An Existential Crisis Resolved
Type inference for first-class existential types

RICHARD A. EISENBERG, Tweag I/O, France
GUILLAUME DUBOC, ENS Lyon, France and Tweag I/O, 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 with projection functions, not weak existentials accessible only by pattern-matching.

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:

```haskell
-- 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 m 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: dropWhile :: (a → Bool) → Vec n a → ∃m. 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.

Authors’ addresses: Richard A. Eisenberg, rae@richarde.dev, Tweag I/O, 207 Rue de Bercy, 75012, Paris, France; Guillaume Duboc, guillaume.duboc@ens-lyon.fr, ENS Lyon, Lyon, France, Tweag I/O, France; Stephanie Weirich, sweirich@seas.upenn.edu, University of Pennsylvania, 3330 Walnut St., Philadelphia, PA, 19104, USA; Daniel Lee, laniel@seas.upenn.edu, University of Pennsylvania, 3330 Walnut St., Philadelphia, PA, 19104, USA.

2020. 2475-1421/2020/3-ART $15.00
https://doi.org/10.1145/nmnnnn.nnmmn

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 language, \( \mathcal{X} \), is thus a superset of Hindley-Milner.
- Section 5 presents a core language based on System F, \( \mathcal{F}_X \), that is a suitable target of compilation (Section 6) for \( \mathcal{X} \). We prove \( \mathcal{F}_X \) is type-safe, and it is designed in a way that is compatible with the existing System FC [Sulzmann et al. 2007] language used internally within the Glasgow Haskell Compiler (GHC).
- Section 7 includes an analysis of our algorithm, showing that it enjoys several stability properties [Anonymous ICFP Author(s) 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.
- Type inference in Haskell goes beyond what is possible in a Hindley-Milner-style language. Section 8 shows that our algorithm is compatible with the latest version of GHC’s type inference, supporting impredicative type inference via the Quick Look algorithm [Serrano et al. 2020].

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.

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 fancy types.

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

-- XStandaloneKindSignatures, new in GHC 8.10

infixr 5 :>
```

```
data Vec :: Nat → Type → Type

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:
The implementation of `filter` using today’s existentials and the version possible with our new ideas; see Figure 1.

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

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
```

![Fig. 1. Implementations of `filter` over vectors (a) in today’s Haskell, and (b) with our extensions](image)

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

```haskell
dropWhile :: (a -> Bool) -> Vec n a -> ∀(m :: Nat). 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 ∃m. 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. 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 \textit{strong} existential packages, where the existentially-packed type can be projected out from the existential package, 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:

\texttt{class Pretty a where}
\texttt{\hspace{1em} pretty :: a \rightarrow Doc}

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

\texttt{vcat :: \ [ \exists a. Pretty a \land a ] \rightarrow Doc}
\texttt{vcat [] = empty}
\texttt{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.

2.4 Richly typed data structures

Suppose we wish to design a datatype whose inhabitants are well-formed by construction. If the well-formedness constraints 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.\footnote{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] extends 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].}

The idea is encapsulated in these definitions:

\texttt{data Ty = Ty ::\rightarrow Ty | \ldots \hspace{1em} -- base types elided}
\texttt{type Exp :: \ [ Ty ] \hspace{1em} -- types of in-scope variables}
\hspace{1em} \hspace{1em} \hspace{1em} \rightarrow Ty \hspace{1em} -- type of expression
\hspace{1em} \rightarrow Type
\texttt{data Exp ctx ty where}
\hspace{1em} \texttt{App :: Exp ctx (arg ::\rightarrow result) \rightarrow Exp ctx arg \rightarrow Exp ctx result}
\hspace{1em} \ldots

An expression of type $\text{Exp}\ ctx\ ty$ is guaranteed to be well-formed: note that a function application requires the function to have a function type $\text{arg} :\rightarrow \text{result}$ and the argument to have type $\text{arg}$. (The ctx is a list of the types of in-scope variables; using de Bruijn indices means we do not need to map names.)

However, if we are to use $\text{Exp}$ 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 $\text{Exp}$? We must type-check it.

Assuming a type $\text{UExp}$ (“unchecked expression”) that is like $\text{Exp}$ but without its indices, we would write the following:

\[
\text{typecheck} \colon (\text{ctx} :: [\ Ty]) \rightarrow \text{UExp} \rightarrow \text{Maybe} (\exists \ ty. \text{Exp ctx ty})
\]

\[
\text{typecheck ctx (UApp fun arg)} = \text{do} -- \text{using the Maybe monad}
\]

\[
\text{fun'} \leftarrow \text{typecheck ctx fun}
\]

\[
\text{arg'} \leftarrow \text{typecheck ctx arg}
\]

\[
(\text{expectedArgTy}, \_\text{resultTy}) \leftarrow \text{checkFunctionTy (typeOf fun')}
\]

\[
\text{Refl} \leftarrow \text{checkEqual expectedArgTy (typeOf arg')}
\]

\[
\text{return (App fun' arg')}
\]

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 to accept the final use of $\text{App}$. 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.

2.5 Refinement types

Refinement types [Rushby et al. 1998] are a convenient way for a programmer to express preconditions and postconditions of a function. For example, we might want $\text{abs} :: (\text{Ord a, Num a}) \Rightarrow a \rightarrow \{ v :: a \mid v \geq 0 \}$, where the annotation in the result type verifies that the result of $\text{abs}$ is non-negative. The Liquid Haskell verification system [Vazou et al. 2014] has had great success in using refinement types in exactly this way.

Yet Liquid Haskell, as currently implemented, works only at the surface level. Once type-checking is complete, the refinements are discarded. This means, for example, that the optimizer may not take advantage of any information written in the refinements. For example, if the refinements indicate that a value cannot be zero, a division primitive might be able to skip the check whether a denominator is 0—yet this opportunity is lost if the refinements are unseen by the optimizer.

Instead, we can imagine understanding the type for $\text{abs}$ above as ending in $\exists (v :: a) \mid v \geq 0$, where this is new syntax (note: a pipe, not a dot, after the $\exists$ binding) indicating that the value of this type is $v$, carrying some constraints written after the pipe. With lightweight existentials, we can easily support a syntax such as this, and accordingly connect a surface-language type system with refinement types to a core language supporting only existentials.

We do not work out the details of this connection in this paper. Yet the techniques described here, along with the use of singletons to mimic dependent types [Eisenberg and Weirich 2012; Monnier and Haguenauer 2010], are powerful enough to accept an encoding of this idea. Full dependent

---

2This 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.
types would be necessary to realize the full promise of this idea, but lightweight existentials appear to be a necessary stepping stone to robust support for refinement types.

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 datatype-based 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) = ∀(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 this existential, we must use pattern matching. We cannot use a straightforward projection function: it would allow the abstracted type variable to escape its scope. 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.

3.1 Strong existentials via `pack` and `open`

Our core language `FX` adopts the following constructs for introducing and eliminating existential types:4

<table>
<thead>
<tr>
<th>Pack</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\Gamma \vdash e : \tau_2[\tau_1/a]]</td>
<td>[\Gamma \vdash e : \exists a.\tau]</td>
</tr>
<tr>
<td>[\Gamma \vdash \text{pack } \tau_1, e \text{ as } \exists a.\tau_2 : \exists a.\tau_2]</td>
<td>[\Gamma \vdash \text{open } e : \tau[[e : \exists a.\tau] / a]]</td>
</tr>
</tbody>
</table>

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

4These rules are slightly simplified. The full rules appear in Section 5.
The `pack` typing rule is fairly standard [Pierce 2002, Chapter 24]. This term creates an existential package, hiding a type $\tau_1$ 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 `opens` 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, $\tau_0$) with an expression of type $\tau[\tau_0 / a]$. The type $\lfloor e : \exists a.\tau \rfloor$ projects out the type $\tau_0$ from the pair.

This simple idea 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.\footnote{Readers may be alarmed at that sentence: how could $\lfloor \bot : \exists a.\tau \rfloor$ 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.}

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 might have multiple, different types. (For example, $\lambda x \rightarrow x$ has both type $\text{Int} \rightarrow \text{Int}$ and type $\text{Bool} \rightarrow \text{Bool}$.) 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:

\[
\begin{align*}
\text{Unpack} & \\
\Gamma \vdash e_1 : \exists a.\tau_2 \\
\Gamma, a, x : \tau_2 \vdash e_2 : \tau \\
\Gamma, a \not\in \text{fv}(\tau) & \\
\Gamma \vdash \text{unpack } e_1 \text{ as } a, x \text{ in } e_2 : \tau
\end{align*}
\]

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 `case e_1 of Ex (x :: a) → e_2`, where $x$ and $a$ are brought into scope in $e_2$.\footnote{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:

\[
\text{unpack } \text{pack } \tau_1, e_1 \text{ as } \exists a.\tau_2 \text{ as } a, x \text{ in } e_2 \rightarrow e_1 [e_1 / x][\tau_1 / a]
\]

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 $a$.\footnote{Readers may be alarmed at that sentence: how could $\lfloor \bot : \exists a.\tau \rfloor$ 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.}
Let us rewrite the \textit{filter} example from Section 2.2, making more details explicit so that we can see why this is an issue.

\begin{verbatim}
filter :: \forall n a. (a \to \text{Bool}) \to \text{Vec} n a \to \exists m. \text{Vec} m a
filter = \Lambda n a \to \lambda(p :: a \rightarrow \text{Bool}) \times \text{Vec} n a \rightarrow 
\text{case} \ 	ext{vec} \ of 
\begin{align*}
& (\rightarrow) \ n1 (x :: a) (xs :: \text{Vec} n1 a) \quad -- \text{vec} \ is \ x \rightarrow xs \\
& | p \ x \ \rightarrow \ ...
& | \text{otherwise} \rightarrow \text{filter} \ n1 \ a \ p \ xs \\
& \text{Nil} \rightarrow \text{pack} \ Zero, \Nil \ as \ \exists m. \text{Vec} m a \quad -- \text{vec} \ is \ Nil
\end{align*}
\end{verbatim}

The treatment above makes all type abstraction and application explicit. Note that the pattern-match for the cons operator $\rightarrow$ 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 $\ldots$ in the case where $p \ x$ holds? One possibility is to apply the $(\rightarrow)$ 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 \texttt{unpack} cannot help us here, as \texttt{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 \texttt{unpack} (and \texttt{pack}) before calling the $(\rightarrow)$ operator. Specifically, we can write

\begin{verbatim}
\texttt{unpack filter n1 a p xs as n2, ys in pack n2, (\rightarrow) n2 x ys as \exists m. Vec m a}
\end{verbatim}

This use of \texttt{unpack} is type-correct, but we have lost the laziness of \textit{filter} we so prized in Section 2.2.

On the other hand, \texttt{open} allows us to fill in the $\ldots$ with the following code, using the the existential projection to access the new (type-level) length for the arguments to \texttt{pack} and to $\rightarrow$.

\begin{verbatim}
let ys :: \exists m. Vec m a -- usual lazy let
ys = filter n1 a p xs
in pack [ys], (\rightarrow) [ys] x (open ys as \exists m. Vec m a)
\end{verbatim}

In this typeset example, we have dropped the (redundant) type annotation in $[ys]$. As we expand on in the next subsection, we do not have to \texttt{let}-bind $ys$; instead, we could just repeat the sub-expression \texttt{filter n1 a p xs}.

### 3.3 The importance of strength

Beyond the peculiarities of the \textit{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:

\begin{verbatim}
data \texttt{Counter} a = \texttt{Counter} \{ \texttt{zero} :: a, \texttt{succ} :: a \to a, \texttt{toInt} :: a \to \text{Int} \}

\texttt{mkCounter :: String \to \exists a. \texttt{Counter} a} \quad -- \text{a counter with a hidden representation}
\texttt{mkCounter = ...}

\texttt{initial1 :: \text{Int}}
\texttt{initial1 = \text{let} c = \texttt{mkCounter "hello" in (toInt c) (zero c)}}

\texttt{initial2 :: \text{Int}}
\texttt{initial2 = (toInt (\texttt{mkCounter "hello")}) (zero (\texttt{mkCounter "hello")})}
\end{verbatim}
We would like our language to accept both initial1 and initial2. In both cases, 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.

In order 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.

4 INFERRING EXISTENTIALS

In this section we present the surface language, 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 \( \exists a.e \) 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.

\[ a, b ::= \ldots \]
\[ \sigma ::= \epsilon | \forall a.\sigma \]
\[ \epsilon ::= \rho | \exists b.e \]
\[ \rho ::= \tau | \sigma_1 \rightarrow \sigma_2 \]
\[ \tau ::= a | \text{Int} | \tau_1 \rightarrow \tau_2 | \lfloor e : \epsilon \rfloor \]
\[ \Gamma ::= \emptyset | \Gamma, a | \Gamma, x:\sigma \]

Fig. 2. Type stratification
Polytypes $\sigma$ 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 $\sigma$-types and $\epsilon$-types. Note that the type $\exists a.\forall b.\tau$ is ruled out.\footnote{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.}

Top-level monotypes $\rho$ have no top-level quantification. Monotypes $\tau$ 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 $\mathcal{X}$ are defined as follows:

- $x ::= \ldots$ \hspace{1cm} \text{term variable}
- $n ::= \ldots$ \hspace{1cm} \text{integer literal}
- $e ::= h\pi | \lambda x.e | \text{let } x = e_1 \text{ in } e_2 | n$ \hspace{1cm} \text{expression}
- $h ::= x | e | e :: \sigma$ \hspace{1cm} \text{expression head}
- $\pi ::= e | \sigma$ \hspace{1cm} \text{argument}

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

This language is a fairly small $\lambda$-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 $\pi_i$ 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.\footnote{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.}

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 must address how to treat type equality. For example, suppose term variable $x$ (of type Int) is free in a type $\tau$; is $\tau[(\lambda y.y) 1 / x]$ equal to $\tau[1 / x]$? That is, does type equality respect $\beta$-reduction? Our answer is “no”: we restrict type equality in our language to be syntactic equality (modulo $\alpha$-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. Adding such an equality relation is orthogonal to the concerns around existential types that draw our focus.\footnote{Our core language $\mathcal{FX}$ does need to think harder about this question, in order to prove type safety. See Section 5.1.}

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 $\Gamma$, the program $e$ has the inferred type $\rho$, while $\Gamma \vdash e \Leftarrow \rho$ means that, in the type environment $\Gamma$, $e$ is checked to have type $\rho$. 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 $\Leftarrow$ with either $\Rightarrow$ or $\Leftarrow$ in both the conclusion and premises. Although the rules are fairly close to the standard rules of a typed $\lambda$-calculus, handling existentials through packing and opening has an impact on the rules $\text{LET}$ and $\text{GEN}$.

We review the rules in Figure 4 here, deferring the most involved rule, $\text{APP}$ until after we discuss the instantiation judgment $\vdash \text{inst}$, in Section 4.3.
\[ \Gamma \vdash^\forall e \iff \sigma \]

\[ \text{(Universal type checking)} \]

\[ \begin{array}{l}
\Gamma, \alpha \vdash e \iff \rho[\bar{\tau} / \bar{b}]
\end{array} \]

\[ \text{Gen} \]

\[ \begin{array}{l}
\forall e \iff \sigma
\end{array} \]

\[ \text{(Type synthesis and type checking)} \]

\[ \begin{array}{l}
\Gamma \vdash e \Rightarrow \rho \\
\Gamma \vdash e \iff \rho
\end{array} \]

\[ \text{App} \]

\[ \begin{array}{l}
\Gamma \vdash h \Rightarrow \sigma \\
\Gamma \vdash^\forall e_i \iff \sigma_i
\end{array} \]

\[ \Gamma \vdash h \bar{\pi} \iff \rho_r \]

\[ \text{H-V ar} \]

\[ \begin{array}{l}
\Gamma \vdash \alpha \Rightarrow \sigma
\end{array} \]

\[ \text{H-Ann} \]

\[ \begin{array}{l}
\Gamma \vdash^\forall e \iff \sigma \\
fv(\tau) \subseteq \text{dom}(\Gamma) \\
\Gamma \vdash (e :: \sigma) \Rightarrow \sigma
\end{array} \]

\[ \text{H-Infer} \]

\[ \Gamma \vdash e \Rightarrow \rho \]

\[ \Gamma \vdash e \iff \rho \]

\[ \text{(Head synthesis)} \]

\[ \begin{array}{l}
\text{Int} \\
\Gamma \vdash n \iff \text{Int}
\end{array} \]

\[ \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \iff \rho_2[e_1 / x] \]

\[ \text{CABs} \]

\[ \begin{array}{l}
\Gamma, x:\sigma_1 \vdash^\forall e \iff \sigma_2 \\
fv(\sigma_1) \subseteq \text{dom}(\Gamma) \\
\Gamma \vdash \lambda x.e \iff \sigma_1 \rightarrow \sigma_2
\end{array} \]

\[ \text{tAbs} \]

\[ \begin{array}{l}
\Gamma, x:\tau \vdash e \Rightarrow \rho \\
fv(\tau) \subseteq \text{dom}(\Gamma) \\
\rho' = \rho[\bar{a} / [\rho]_x]
\end{array} \]

\[ \Gamma \vdash \lambda x.e \iff \sigma_1 \rightarrow \sigma_2 \]

\[ \text{Let} \]

\[ \begin{array}{l}
\Gamma \vdash e_1 \Rightarrow \rho_1 \\
\bar{a} = fv(\rho_1) \setminus \text{dom}(\Gamma) \\
\Gamma, x:\forall \bar{a}.\rho_1 \vdash e_2 \Rightarrow \rho_2
\end{array} \]

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

However, our type system refers to no such \( \leq \) 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 \( \rho_r \) in rule App, when replacing the \( \iff \) in the conclusion with a \( \iff \).) We get away with this because our bidirectional type-checking algorithm works over top-level monotypes \( \rho \), not the more general polytype \( \sigma \). A type \( \rho \) 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].

Checking against a polytype. Rule GEN, the sole rule for the $\Gamma \vdash^V e \Leftarrow \sigma$ judgment, deals with the case when we are checking against a polytype $\sigma$. If we want to ensure that $e$ has type $\sigma$, then we must skolemize any universal variables bound in $\sigma$: 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 $\sigma$, 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 $\tau$ 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 $\exists a. e$, we guess a monotype $\tau$ and check $e$ against the type $e[\tau / a]$. 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 that allows us to guess monotypes during type inference, knowing how to translate this convention into an implementation using unification variables.

Abstractions. Rule IABS synthesizes the type of a $\lambda$-abstraction, by guessing the (mono)type $\tau$ of the bound variable and then inferring the type of the body $e$ to be $\rho$. However, rule IABS also can pack existentials. This is necessary to avoid skolem escape: it is possible that the type $\rho$ 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.

We thus must identify all existential projections within $\rho$ that have $x$ free. These are replaced with fresh variables $\overline{a}$. We use the notation $[\rho]_x$ to denote the list of projections in $\rho$; multiple projections of the same expression (that is, multiple occurrences of $[e_0 : e_0]$ for some $e_0$ and $e_0$) are commoned up in this list. Formally,

$$[\rho]_x = \{[e : e] \mid [e : e] \text{ is a sub-expression of } \rho \land x \text{ is a free variable in } e\}.$$

The notation $\rho[\overline{a} / [\rho]_x]$ denotes the type $\rho$ 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 $\tau \rightarrow \exists \overline{a}, \rho'$.

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

Let skolem-escape. Rule LET deals with $\mathtt{let}$-expressions, both in synthesis and in checking modes. It performs standard let-generalization, computing generalized variables $\overline{a}$ by finding the free variables in $\rho_1$ and removing any variables additionally free in $\Gamma$. Indeed, all that is unexpected in this rule is the type in the bottom-right corner, which has a perhaps-surprising substitution.

The problem, like with rule IABS, is the potential for skolem-escape. The variable $x$ might appear in the type $\rho_2$. However, $x$ is out of scope in the conclusion, and thus it cannot appear in the overall type of the $\mathtt{let}$-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 $\epsilon$ instead of top-level monomorphic types $\rho$; keeping with the simpler $\rho$ 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 $\text{LET}$ rule.

Infering 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 $\Gamma \vdash h \Rightarrow \sigma$ judgment. Variables simply perform a context lookup, annotated expressions check the contained expression against the provided type, and other expressions infer a $\rho$-type. It is understood here that we use rule $\text{H-INFER}$ only when the other rules do not apply, for example, for $\lambda$-abstractions.

4.3 Instantiation semantics

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

The idea is that we use $\vdash \text{inst}$ 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 $\vec{\sigma}$, rule $\text{APP}$ (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.

Application. Rule $\text{ITYARG}$ handles type application by instantiating the bound variable $a$ with the supplied type argument $\sigma'$. Rule $\text{IARG}$ handles routine expression application simply by remembering that the argument should have type $\sigma_1$. Note that we do not check that the argument $e'$ has type $\sigma_1$ here.

Quantifiers. Rule $\text{IALL}$ deals with universal quantifiers in the function’s type by instantiating with a guessed monotype $\tau$. The first premise is to avoid ambiguity with rule $\text{ITYARG}$; we do not wish to guess an instantiation when the user provides it explicitly with a type argument.
Rule \( \textsf{IEXIST} \) 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 \( \textsf{Bool} \rightarrow \exists b. (b, b \rightarrow \textsf{Int}) \), then the function application \( f \textsf{True} \) will be given the opened pair type:

\[
(\{ f \textsf{True} : \exists b. (b, b \rightarrow \textsf{Int}) \}, \{ f \textsf{True} : \exists b. (b, b \rightarrow \textsf{Int}) \} \rightarrow \textsf{Int})
\]

Rule \( \textsf{IRESULT} \) concludes computing the instantiation in a function application by copying the function type to be the result type.

The \textbf{App rule}. Having now understood the instantiation judgment, we turn our attention to rule \( \textsf{APP} \). After inferring the type \( \sigma \) for an application head \( h \), \( \sigma \) gets instantiated, revealing argument types \( \overline{\sigma} \). Each argument \( e_i \) is checked against its corresponding type \( \sigma_i \), where the entire function application expression has type \( \rho_f \). Rule \( \textsf{APP} \) operates in both synthesis and checking modes. When synthesizing, it simply returns \( \rho_f \) from the instantiation judgment; when checking, it ensures that the instantiated type \( \rho_f \) 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 projects \( [e : e] \), 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. 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.

\[
\begin{align*}
B & ::= \rightarrow | \text{Int} | \ldots & \text{base type} \\
\text{t, r, s} & ::= a | B \overline{t} | \forall a.t | \exists a.t | [e] & \text{type} \\
e, h & ::= x | n | \lambda x.t.e | e_1 e_2 | \Lambda a.e | e t | \text{pack } t, e \text{ as } t_2 \\
& | \text{open } e | \text{let } x = e_1 \text{ in } e_2 | e \triangleright \gamma & \text{expression} \\
v & ::= n | \lambda x.t.e | \Lambda a.v | \text{pack } t, v \text{ as } t_2 & \text{value} \\
\gamma & ::= \langle t \rangle | \text{sym } \gamma | \gamma_1 :: \gamma_2 | [\eta] | \gamma_1 @ \gamma_2 | \text{projpack } t, e \text{ as } t_2 | \ldots & \text{type coercion} \\
\eta & ::= e \triangleright \gamma | \text{step } e & \text{expression coercion} \\
G & ::= \emptyset | G, x : t | G, a & \text{typing context}
\end{align*}
\]

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, \texttt{pack} and \texttt{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 \( \texttt{CT-PROJ} \) shows that a projection in a type is well-formed when the expression is well-scoped. (The \( \vdash G \texttt{ok} \)
Fig. 7. Selected typing rules of the core language.
We prove (almost) standard progress and preservation theorems for this language: We want to extract the value \( FX \).

The biggest surprise in the presence of casts is essentially unimportant) and to reduce expressions. Figure 7. These rules allow us to drop casts (supporting a coherence property which states that the equivalence is also a congruence over types. Coercions also include several decomposition relations (rules \( CG\text{-}Refl \), \( CG\text{-}Sym \), and \( CG\text{-}Trans \)), along with several omitted forms showing that the equivalence is also a congruence over types. Coercions also include several decomposition operations; rule \( CG\text{-}InstExists \) shows one, used in our reduction rules. The two forms of interest to use are \( \eta \) (rule \( CG\text{-}Proj \)) and \( projpack \) (rule \( CG\text{-}ProjPack \)). The former injects the equivalence relation on expressions (witnessed by expression coercions \( \eta \)) into the type equivalence relation, and the latter witnesses the equivalence between \( [pack \ t, e \ as \ t] \) 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.

5.1 Coercions

The biggest surprise in \( FX \) is its need for type and expression coercions. The motivation for these can be seen in rule \( CS\text{-}OpenPack \). If we are stepping an expression \( open (pack \ t, v as \exists a. t_2) \), 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 \( pack \ t, v as \exists a. t_2 \) and \( open (pack \ t, v as \exists a. t_2) : t_2 \langle [pack \ t, v as \exists a. t_2] / a \rangle \), according to rule \( CE\text{-}Open \). This last type is not syntactically the same as \( t_0 \), although it must be that \( t_0 = t_2[t / a] \) to satisfy the premises of rule \( CE\text{-}Pack \). Because the type of the opened existential does not match the type of the packed value, a naïve reduction rule like \( G \vdash open (pack \ t, v as t_2) \rightarrow v \) 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 \( [pack \ t, e as t_2] \equiv t \). 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 \( t_1 \equiv t_2 \). 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 \( CS\text{-}OpenPack \) into the proofs.

The alternative approach to a non-trivial equality relation is to use explicit coercions, as we have here. The cost here is clutter. Casts sully our reduction steps, and we need to explicitly shunt coercions in several (omitted, unenlightening) reduction rules—for example, when reducing \( ((\lambda x.t_1) \ b) \ v_2 \) where the cast intervenes between a \( \lambda \)-abstraction and its argument.

Both approaches are essentially equivalent: we can view explicit coercions simply as an encoding of the derivation of an equality judgment.

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

5.2 Metatheory

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

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

1. there exists \( e' \) such that \( G \vdash e \rightarrow e' \);

\footnote{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.}
\( \Gamma \vdash \forall e \equiv \sigma \Rightarrow e \) elaboration of polymorphic expressions

\( \Gamma \vdash e \equiv \rho \Rightarrow e \) elaboration of expressions

\( \Gamma \vdash_h h \Rightarrow \sigma \Rightarrow h \) elaboration of application heads

\( \sigma \Rightarrow s \) elaboration of types

\( \Gamma \Rightarrow G \) elaboration of typing contexts

Fig. 8. Judgments used for elaborating from \( X \) into \( FX \).

(2) \( e \) is a value \( v \); or

(3) \( e \) is a casted value \( v \triangleright \gamma \).

**Theorem 5.2 (Preservation).** If \( G \vdash e : t \) and \( G \vdash e \rightarrow e' \), then \( G \vdash e' : t \).

In addition, we prove that types can still be erased in this language. Let \(|e|\) denote the expression \( e \) with all type abstractions, type applications, packs, opens and casts dropped. Furthermore, overload \( \rightarrow \) to mean the reduction relation over the erased language.

**Theorem 5.3 (Erasure).** If \( G \vdash e \rightarrow^* e' \), then \(|e| \rightarrow^* |e'|\).

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 \( X \) into our core \( FX \). The notation \( \Rightarrow \) denotes elaboration of a surface term, type or context into its core equivalent. Many of our rules appear in Figure 8, where some unsurprising rules are elided. The rest appear in the appendix. In order to aid understanding, we use blue for \( X \) terms and red for \( FX \) terms.

#### 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 \([e : \exists a.\epsilon] \). 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 \([e_0 : \epsilon]\)—and \( e_2 \) indeed has a type mentioning \([e_0 : \epsilon]\)—we cannot be sure that the application remains well-typed after elaboration. After all, type-checking in \( X \) is non-deterministic, given the way it guesses instantiations and the types of \( \lambda \)-bound variables. Another wrinkle is that \([e_0 : \epsilon]\) might appear under binders, making it even easier for type inference to come to two different conclusions when computing \( \Gamma \vdash \forall \epsilon e_0 \equiv \epsilon \).

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:
Now, the elaboration process \( \tau \Rightarrow t \) is indeed deterministic, making \( \Rightarrow \) a function on types \( \tau \) and contexts \( \Gamma \). 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, \( X \) would keep using projections of the form \( \lfloor e : \sigma \rfloor \), where we understand that \( \Gamma \vdash e : \sigma \) in the ambient context \( \Gamma \), while \( FX \) uses the form \( \lfloor e \rfloor \).

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.

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 \( \text{Int} \) and \( \text{Bool} \) as types for \( x \) in our example. In essence, a deterministic elaborator means that we can consider \( \lfloor e : \sigma \rfloor \) as a proxy for \( [e] \). 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 evaluator.

### 7 ANALYSIS

The surface language \( X \) allows us to easily manipulate existentials in a \( \lambda \)-calculus while delegating type consistency to an explicit core language \( FX \). The following theorems establish the soundness of this approach, via the elaboration transformation \( \Rightarrow \), 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 \( \Gamma \vdash e : \sigma \Rightarrow e \), then \( G \vdash e : s \), where \( \Gamma \vdash G \) and \( \sigma \vdash s \).
2. If \( \Gamma \vdash e \Rightarrow \rho \Rightarrow e \), then \( G \vdash e : r \), where \( \Gamma \vdash G \) and \( \rho \vdash r \).
3. If \( \Gamma \vdash e \Rightarrow \rho \Rightarrow e \), then \( G \vdash e : r \), where \( \Gamma \vdash G \) and \( \rho \vdash r \).

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 \( \Gamma \vdash e : \sigma \Rightarrow e \), then \( |e| = |e| \).
2. If \( \Gamma \vdash e \Rightarrow \rho \Rightarrow e \), then \( |e| = |e| \).
(3) If $\Gamma \vdash e \Leftrightarrow \rho \Rightarrow e$, then $|e| = |e|$.

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

### 7.2 Familiarity

Not only do we want our $\mathcal{X}$ programs to be sound, but we also want $\mathcal{X}$ to be a comfortable language to program in. We have captured some elements of this in the next several theorems, stating that our design is backward-compatible and that adding redundant annotations do not disrupt type-checking.

**Theorem 7.3 (Conservative extension of Hindley-Milner).** If $e$ has no type arguments or type annotations, and $\Gamma, e, \tau, \sigma$ contain no existentials, then:

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

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

**Theorem 7.4 (Synthesis implies checking).** If $\Gamma \vdash e \Rightarrow \rho$ then $\Gamma \vdash e \Leftrightarrow \rho$.

### 7.3 Stability

The following theorems denote stability properties [Anonymous ICFP Author(s) 2021]. In other words, they ensure that small user-written transformations do not change drastically the static semantics 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 7.5 (let-inlining).** If $x$ is free in $e_2$ then:

- $(\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow \rho)$ implies $(\Gamma \vdash e_2[e_1/x] \Rightarrow \rho)$
- $(\Gamma \vdash^\forall \text{let } x = e_1 \text{ in } e_2 \Leftrightarrow \sigma)$ implies $(\Gamma \vdash^\forall e_2[e_1/x] \Leftrightarrow \sigma)$

**Theorem 7.6 (Order of Quantification does not matter).** Let $\rho'$ (resp. $\sigma'$) be two types that differ from $\rho$ (resp. $\sigma$) 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. $(\Gamma \vdash^\forall e \Leftrightarrow \sigma)$ if and only if $(\Gamma \vdash^\forall e \Leftrightarrow \sigma')$

### 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.

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 $\texttt{iArg}$ is our rule $\texttt{iArg}$, and their rule $\texttt{App}$ is our rule $\texttt{App}$. The only wrinkle in merging these systems is that their presentation uses a notion of *instantiation variable*, which Serrano et al. write as $\kappa$. Given that impredicative instantiation is not a primary goal of our work, we choose not to use this approach in our formalism, instead preferring the more conventional idiom of using guessed $\tau$-types. It would be a straightforward exercise to reframe our judgments in Section 4 to use instantiation variables, similarly to the approach by Serrano et al.

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 $\texttt{Gen}$. That rule skolemizes (makes fresh constants out of) the variables universally quantified in $\sigma$ and guesses $\tau$ to instantiate the existentially quantified variables. In order to allow these instantiations to be impredicative, we must modify the rule, as follows:

\[
\Gamma \vdash \forall e \Leftarrow \sigma
\]

\[
\text{GEN\textsc{Impredicative}}
\]

\[
\begin{align*}
& \kappa \text{ fresh} \\
& \rho' = \rho[\kappa / b] \\
& \Gamma, \bar{a} \vdash e : \rho' \rightsquigarrow \Theta \\
& \rho'' = \Theta \rho' \\
& \text{dom } (\Theta) = \text{fiv } (\rho'') \\
& \Gamma, \bar{a} \vdash e \Leftarrow \theta \rho'' \\
& \Gamma \vdash \forall e \Leftarrow \forall \bar{a}. \exists \bar{b} . \rho
\end{align*}
\]

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

This rule follows broadly the pattern from rule $\texttt{Gen}$, but using instantiation variables $\kappa$ instead of guessing $\tau$. The third premise invokes the Quick Look judgment $\vdash \_ \_ \_ \_ \$ [Serrano et al. 2020, Figure 5] to generate a substitution $\Theta$. Such a substitution $\Theta$ maps instantiation variables $\kappa$ to polytypes $\sigma$; by contrast, a substitution $\theta$ includes only monotypes $\tau$ in its codomain. The next two premises of rule $\text{GEN\textsc{Impredicative}}$ apply the $\Theta$ substitution, and then use $\theta$ to eliminate any remaining instantiation variables $\kappa$: the $\text{fiv } (\rho'')$ extracts all the *free* instantiation variables in $\rho''$. Note that the range of $\theta$ appears unconstrained here; the types in its range are guessed, just like the $\tau$ in rule $\texttt{Gen}$.

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

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

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 \( (\text{True}, \lambda z \rightarrow 1) \): infer the former to have type \( (\text{Int}, \lambda y \rightarrow \text{Int}) \) and the latter to have type \( (\text{Bool}, \alpha \rightarrow \text{Int}) \) for some unknown type \( \alpha \).

The goal then is to find some type \( \tau \) such that \( \tau \) can instantiate to either of these two types: this is anti-unification. The problem is, in this case, \( \alpha \): we get different results depending on whether \( \alpha \) becomes \( \text{Int} \) or \( \text{Bool} \).

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 \( \text{Show \ Int} \)), 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 $\sigma$ can now require proving a constraint: to use $e : Q \Rightarrow \sigma$, the constraint $Q$ must hold. Existentially quantified types $\epsilon$ can now provide the proof of a constraint: the expression $e : Q \land \epsilon$ contains evidence that $Q$ holds. Assumed constraints appear in contexts $\Gamma$.

The surprising feature here is that we have a new form of assumption, $[e : \epsilon]$. This assumption is allowed only when $\epsilon$ has the form $Q \land \epsilon'$; 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 $[e : \epsilon]$. This arbitrary assumption is quite like how rule Gen assumes types $\bar{\epsilon}$ to replace the existential variables $\bar{b}$. To prevent the type system from working in an unbounded search space for assumptions to make, the expressions $\epsilon$ must be sub-expressions of our checked expression $e_0$.

The instantiation judgment $\vdash_{inst}$ must also accommodate constraints. When, in rule IGiven, it comes across an expression whose type includes a packed assumption $Q \land \epsilon$, it checks to make sure that assumption was included in $\Gamma$. 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 $Q \land \epsilon$, 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 $a \land a$ and call pretty ($mk$ True). Calling pretty requires passing the dictionary giving the implementation of the function at the specific type pretty is instantiated at ($[mk$ True $:: $ $\exists a$. Pretty $a \land a$], 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 $Q \land \epsilon$ as the type of pairs of a dictionary for $Q$ and the inhabitant of $\epsilon$: the naive 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 communing 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 $\{v :: Int \mid v \geq 0\}$; 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 $\rightarrow $ $\{v :: $ Int $\mid v \geq 0\}$. This function would return a value $v$ of type Int, along with

---

11Other 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 $\Gamma$. 

a proof that $v \geq 0$: this is a dependent pair, or an existential package. Thus, we can rephrase the type of mk to be $\text{Bool} \rightarrow \exists(v :: \text{Int})$. Proof $(v \geq 0)$, 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 $v :: \text{Int}$ at runtime, discarding the compile-time proof that $v \geq 0$.

The core language presented in this paper cannot, without embellishment, support relevant first components of existentials. In other words, $[e : \epsilon]$ 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 $\Sigma$-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 version of existentials found in GHC, based on datatype declarations, 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. The feature was also implemented in 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, in UHC values of existential types can be opened in place, without the use of an unpack term with a bounded scope for the abstract type. 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 module systems for the ML language. We do not attempt to summarize that field here, but mention a few 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 F₁₀ augmented with pack and unpack. Our approach is similar to theirs, in that we also use a translation of a surface language into our 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 F₁₀ 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.

REFERENCES


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.


Dedicated to H. B. Curry on the occasion of his 80th birthday.


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


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


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.


A ELABORATION RULES

We first extend the \( \mathcal{P} \cal{K} \) grammar to include arguments:

\[
p ::= \ e \mid t \quad \text{argument}
\]

(Elaboration for polymorphic expressions)

\[
\Gamma \vdash \ e \iff \sigma \Rightarrow \ e
\]

(Elaboration for expressions)

Elab-Gen

\[
\begin{align*}
\Gamma, \bar{a} \vdash e \iff \rho[\bar{\tau} / \bar{b}] \Rightarrow e \\
\tau \Rightarrow t & \quad \rho \Rightarrow r \\
\text{fv}(\bar{\tau}) \subseteq \text{dom}(\Gamma, \bar{a})
\end{align*}
\]

\[
\Gamma \vdash \ e \iff \forall \bar{a}, \exists \bar{\rho} \cdot \Lambda \bar{a}. \text{pack} \ t, e \text{ as } \exists \bar{b}. \ r
\]

Elab-App

\[
\begin{align*}
\Gamma \vdash h & \Rightarrow \sigma \Rightarrow h \\
\text{inst}_{h : \sigma} & \Rightarrow \text{inst}_{h : \sigma} \Rightarrow \text{inst}_{\rho_r} \Rightarrow \text{inst}_{e_r}
\end{align*}
\]

\[
\Gamma \vdash \ h \bar{\pi} \iff \rho_r \Rightarrow e_r
\]

(Elaboration for heads)

Elab-cAbs

\[
\begin{align*}
\Gamma, x: \sigma_1 \vdash e & \iff \sigma_2 \Rightarrow e \\
fv(\sigma_1) \subseteq \text{dom}(\Gamma) & \\
\sigma_1 & \Rightarrow s_1
\end{align*}
\]

\[
\Gamma \vdash \lambda x. e \iff \sigma_1 \Rightarrow \sigma_2 \Rightarrow \lambda x : s_1.e
\]

Elab-LetCore

\[
\begin{align*}
\Gamma \vdash e & \Rightarrow \rho_1 \iff e_1 \\
\bar{a} & = \text{fv}(\rho_1) \setminus \text{dom}(\Gamma) \\
\Gamma, x : \forall \bar{a}, \rho_1 \vdash e_2 & \iff \rho_2 \Rightarrow e_2
\end{align*}
\]

\[
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow \rho_2[\Lambda \bar{a}. e_1 / x] \Rightarrow \text{let } x = \Lambda \bar{a}. e_1 \text{ in } e_2
\]

(Elaboration for instantiation)

Elab-Var

\[
\begin{align*}
x : \sigma & \in \Gamma \\
\Gamma \vdash h & \Rightarrow \sigma \Rightarrow x
\end{align*}
\]

Elab-Ann

\[
\begin{align*}
\Gamma \vdash e & \iff \sigma \Rightarrow e \\
fv(\sigma) \subseteq \text{dom}(\Gamma) & \\
\Gamma \vdash \text{let } e : \sigma : \Rightarrow e \iff \sigma \Rightarrow e
\end{align*}
\]

Elab-Inf

\[
\begin{align*}
\Gamma \vdash e & \Rightarrow \rho \Rightarrow e
\end{align*}
\]

(Elaboration for instantiation)
The construction is defined recursively by:

\[
\text{Elab-iAbs} \quad (\text{Elaboration for types})
\]

\[
\text{ElabC-Nil} \quad \text{ElabC-TyVar} \quad \text{ElabC-Var} \quad \text{ElabC-Var} \quad \text{ElabC-TyVar} \quad \text{ElabC-Var}
\]

In a small abuse of notation, we write (for example, in rule \text{Elab-iAbs}) a list of types in a \text{pack} construct to denote nested packs. Formally, for \(e\) of type \(r[\overline{t} / \overline{a}]\), with \(\overline{t} = t_1 ... t_n \) and \(\overline{a} = a_1 ... a_n\), the construction is defined recursively by:

\[
\text{pack} \ t_1 ... t_n, e \ \text{as} \ \exists a_1 ... a_n.r = \text{pack} \ t_1, (\text{pack} \ t_2 ... t_n, e \ \text{as} \ \exists a_2 ... a_n.r[t_1 / a_1]) \ \text{as} \ \exists a_1 a_2 ... a_n.r
\]

Define erasure on \(\|\) terms by the following equations:

\[
|n| = n \\
|x| = x \\
|e :: \sigma| = |e| \\
|h \overline{t}, e| = |h \overline{t}| |e| \\
|h \overline{a}, \sigma| = |h \overline{a}| \\
|\lambda x.e| = \lambda x.|e| \\
|\text{let } x \ = \ e_1 \ \text{in } e_2| = \text{let } x = |e_1| \ \text{in } |e_2|
\]

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

1. If \(\Gamma \vdash^\nu e \iff \sigma \Rightarrow e\), then \(|e| = |e|\).
2. If \(\Gamma \vdash e \Rightarrow \rho \Rightarrow e\), then \(|e| = |e|\).
3. If \(\Gamma \vdash e \Rightarrow \rho \Rightarrow e\), then \(|e| = |e|\).
4. If \(\Gamma \vdash h \Rightarrow \sigma \Rightarrow h\), then \(|h| = |h|\).
5. If \(\Gamma \vdash^\iota e : \sigma \Rightarrow e ; \overline{t} : \sigma \Rightarrow e_0 \) and \(|e| = |e|\), then \(|e \overline{t}| = |e_0|\).

**Proof.** By straightforward induction on the elaboration judgments. \(\square\)
B PROOFS ABOUT OUR SURFACE LANGUAGE, X

Theorem B.1 (Soundness).
(1) If $\Gamma \vdash e : \sigma \iff e$, then $\Gamma \vdash e : s$, where $\Gamma \vdash G$ and $\sigma \Rightarrow s$.
(2) If $\Gamma \vdash e : \rho \Rightarrow e$, then $\Gamma \vdash e : r$, where $\Gamma \vdash G$ and $\rho \Rightarrow r$.
(3) If $\Gamma \vdash e : \rho \Rightarrow e$, then $\Gamma \vdash e : r$, where $\Gamma \vdash G$ and $\rho \Rightarrow r$.
(4) If $\Gamma \vdash h \Rightarrow \sigma \Rightarrow h$, then $\Gamma \vdash h : s$, where $\Gamma \vdash G$ and $\sigma \Rightarrow s$.
(5) If $\Gamma \vdash e \Rightarrow e$, then $\Gamma \vdash e : r$, where $\Gamma \vdash G$, $\sigma \Rightarrow s$ and $\rho_r \Rightarrow r_r$.

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: $\Gamma, \overline{a} \vdash e \Leftarrow \rho[\overline{a} / \overline{b}] \Rightarrow e$, where $\overline{r} \Rightarrow t$ and $\rho \Rightarrow r$. By induction hypothesis, $G, \overline{a} \vdash e : r[\overline{a} / \overline{b}]$. By successive applications of rule CE-Pack we get $G, \overline{a} \vdash \text{pack} \overline{t}, e$ as $\exists \overline{b}. r : \exists \overline{b}. x$. Then by successive applications of rule CE-Tabs we get the result: $G \vdash \overline{A}. \text{pack} \overline{t}, e$ as $\exists \overline{b}. r : \forall \overline{a}. \exists \overline{b}. r$.

Rule Elab-App Inference and synthesis are treated at the same time by mutual induction. By induction hypothesis, $G \vdash h : s$ where $\sigma \Rightarrow s$. Then by induction hypothesis (case (5)), we obtain $G \vdash e_r : r_r$.

Rule Elab-Tabs By induction hypothesis, $G, x : t \vdash e : r$. By applications of rule CE-Pack we obtain $G, x : t \vdash \text{pack} [r]_x, e$ as $\exists \overline{a}. r' : \exists \overline{a}. x$. We conclude by applying rule CE-Tabs where the premise $x \notin \text{fv}(\exists \overline{a}. r')$ is verified by construction of $r'$ and definition of $[r]_x$.

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 CE-Let.

Rule Elab-Var Since $x : \sigma \in \Gamma$, we have $x : s \in G$ 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 $G \vdash h : s$. 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 $\Gamma \vdash e : \sigma \Rightarrow e : s \Rightarrow \overline{s}$).

Rule Elab-Itarg Assuming that $G \vdash e : \forall a : s$, by rule CE-Tapp: $G \vdash e : s[a]$.

Rule Elab-IArg Assuming that $G \vdash e : s_1 \rightarrow s_2$ and $\Gamma \vdash e' \Rightarrow \sigma_1 \Rightarrow e''$. By induction hypothesis, $G \vdash e' : s_1$ where $\sigma_1 \Rightarrow s_1$. By rule CE-App we obtain $G \vdash e : s_2$.

Rule Elab-IAll Assuming that $G \vdash e : \forall a : s$. By rule CE-Tapp, we obtain $G \vdash e : s[t / a]$.

Rule Elab-IExistCore Assuming that $G \vdash e : \exists a : \tau$ where $\epsilon \Rightarrow t$. By rule CE-Open: $G \vdash \epsilon : t[\epsilon / a]$.

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 $\rho_r$ such that $G \vdash e_r : r_r$ with $\rho_r \Rightarrow r_r$.

Theorem B.2 (Conservative extension of Clément et al. [1986]). If $e$ has no type arguments or type annotations, and $\Gamma, e, \tau, \sigma$ contain no existentials, then:

(1) $(\Gamma \vdash_{HM} e : \tau)$ implies $(\Gamma \vdash e \Rightarrow \tau)$

(2) $(\Gamma \vdash_{HM} e : \sigma)$ implies $(\Gamma \vdash e \Leftarrow \sigma)$

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 $x: \forall \alpha. \tau' \in \Gamma$, with $\tau = \tau'[\alpha / \emptyset]$. In our type system, we can type $\Gamma \vdash_h x \Rightarrow \forall \alpha. \tau$ with $H\_Var$. Then the instantiation judgment gives us $\Gamma \vdash^{\text{inst}} x : \forall \alpha. \tau' ; [] \Rightarrow [] ; \tau$ as the $\text{IAll}$ rule will be used to instantiate $\forall \alpha. \tau$ with $\tau$. Finally we apply $\text{App}$ to obtain $\Gamma \vdash x \Rightarrow \tau$.

- $e = \lambda x. e'$: Since there are no existentials in $\tau = \tau_1 \rightarrow \tau_2$, hence in $\tau_2$, the $\text{iAbs}$ rule is the same as the usual $\text{C\_Abs}$ rule, therefore we conclude by induction.

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

- **Rule $\text{Let}$ and rule $\text{App}$**: Same rules for synthesis and checking.

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

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

**Proof.** In inference mode, the only rule that packs existentials is rule $\text{iAbs}$. 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 $\rho$ to type $\rho'$.

In checking mode, rule $\text{Gen}$ also does several packs at once, whose ordering does not matter.

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

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

**Proof.** By structural induction on the derivation.

- **Rule $\text{Gen}$**: By inversion, $\Gamma, \overline{\alpha}' \vdash e \Leftarrow \rho[\tau / \overline{\delta}]$. By $\alpha$-equivalence, it is permissible to choose the $\overline{\alpha}'$ fresh, such that $\overline{\alpha}$ and $\overline{\alpha}'$ do not intersect. Hence, we have $\overline{\alpha} \notin \text{dom} (\Gamma, \overline{\alpha}')$ and by induction hypothesis $\overline{\alpha} \notin \text{fv}(e)$.
Rule **APP**: By induction hypothesis, we have $\vec{a} \notin fv(h)$ as well as $\vec{a} \notin fv(e_i)$ for all $i$. Since $\Gamma \vdash^{inst} h : \sigma ; \overline{\tau} \rightsquigarrow \sigma ; \rho_r$, we also know thanks to the scoping rule of rule **ITyArg** that for every $\sigma' \in \overline{\tau}$, $fv(\sigma') \subseteq dom(\Gamma)$. So since $\vec{a} \notin dom(\Gamma)$ we conclude that $\vec{a} \notin fv(h \overline{\tau})$.

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

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

**Rule LETCore**: By induction hypothesis $\vec{a} \notin fv(\epsilon)$ Consider $\vec{a}' = fv(\rho_1) \backslash dom(\Gamma)$ and $\Gamma, x : \forall \vec{a}' . \rho_1 + e_2 \Leftrightarrow e_2$. By definition of the $\vec{a}'$, $\vec{a} \notin dom(\Gamma,x : \forall \vec{a}' . \rho_1)$ so by induction hypothesis $\vec{a} \notin fv(e_2)$ 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.

\[ \square \]

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

1. If $\Gamma \vdash^{\forall} e \leftarrow \sigma \Rightarrow e$, then $\Gamma \vdash^{\forall} e \leftarrow \sigma[\overline{\tau} / \overline{\vec{a}}] \Rightarrow e[\overline{\vec{f}} / \overline{\vec{a}}]$, where $\overline{\vec{f}} \Leftarrow \overline{\vec{f}}$.
2. If $\Gamma \vdash h \Rightarrow \sigma \Rightarrow h$, then $\Gamma \vdash h \Rightarrow \sigma[\overline{\tau} / \overline{\vec{a}}] \Rightarrow h[\overline{\vec{f}} / \overline{\vec{a}}]$, where $\overline{\vec{f}} \Leftarrow \overline{\vec{f}}$.
3. If $\Gamma \vdash e \leftarrow \rho \Rightarrow e$, then $\Gamma \vdash e \leftarrow \rho[\overline{\tau} / \overline{\vec{a}}] \Rightarrow e[\overline{\vec{f}} / \overline{\vec{a}}]$, where $\overline{\vec{f}} \Leftarrow \overline{\vec{f}}$.
4. If $\Gamma \vdash h \Rightarrow \sigma[\overline{\tau} / \overline{\vec{a}}] \Rightarrow h[\overline{\vec{f}} / \overline{\vec{a}}]$ where $\overline{\vec{f}} \Leftarrow \overline{\vec{f}}$ and $\Gamma \vdash^{\forall} h : \sigma \Rightarrow h ; \overline{\tau} \rightsquigarrow \sigma : \rho_r \Rightarrow e_r$, then $\Gamma \vdash^{\forall} h : \sigma[\overline{\tau} / \overline{\vec{a}}] \Rightarrow e[\overline{\vec{f}} / \overline{\vec{a}}] ; \overline{\tau} \rightsquigarrow \sigma[\overline{\tau} / \overline{\vec{a}}] ; \rho_r[\overline{\tau} / \overline{\vec{a}}] \Rightarrow e_r[\overline{\vec{f}} / \overline{\vec{a}}]$ where $\overline{\vec{f}} \Leftarrow \overline{\vec{f}}$.

**Proof**: By structural induction on elaboration derivations.

**Rule ELAB-GEN**: Since $a \notin dom(\Gamma, \overline{\vec{a}})$, by induction hypothesis $\Gamma, \overline{\vec{a}} + e \leftarrow \rho[\overline{\tau} / \overline{\vec{b}}] \Rightarrow e[\overline{\vec{f}} / \overline{\vec{a}}]$ where $\overline{\vec{f}} \Leftarrow \overline{\vec{f}}$. By rule **ELAB-Gen** $\Gamma \vdash^{\forall} e \leftarrow \forall \overline{\vec{a}} . \exists \overline{\vec{b}} . \rho[\overline{\tau} / \overline{\vec{a}}] \Rightarrow \Lambda \overline{\vec{a}} . \text{pack} \overline{\vec{f}}$, $e[\overline{\vec{f}} / \overline{\vec{a}}]$ as $\exists \overline{\vec{b}}. r[\overline{\vec{f}} / \overline{\vec{a}}]$ where $\overline{\vec{f}} \Leftarrow \overline{\vec{f}}$. Since $fv(\tau') \subseteq dom(\Gamma, \overline{\vec{a}}')$ and $\vec{a} \notin dom(\Gamma), \Lambda \overline{\vec{a}} . \text{pack} \overline{\vec{f}}', e[\overline{\vec{f}} / \overline{\vec{a}}]$ as $\exists \overline{\vec{b}}. r[\overline{\vec{f}} / \overline{\vec{a}}] = (\Lambda \overline{\vec{a}} . \text{pack} \overline{\vec{f}}', e[\overline{\vec{f}} / \overline{\vec{a}}])$ which concludes.

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

**Rule ELAB-iABS**: By induction hypothesis $\Gamma, x : \tau + e \Rightarrow \rho[\overline{\tau} / \overline{\vec{a}}] \Rightarrow e[\overline{\vec{f}} / \overline{\vec{a}}]$. We find that, since $fv(\overline{\tau}) \subseteq dom(\Gamma)$, $\rho[\overline{\tau} / \overline{\vec{a}}][\overline{\vec{a}}'] / [\rho[\overline{\tau} / \overline{\vec{a}}]_x] = \rho[\overline{\vec{a}}'] / [\rho[\overline{\vec{a}}]_x]$. By rule **ELAB-iABS**, we obtain $\Gamma + \lambda x . e \Rightarrow \Rightarrow \exists \overline{\vec{a}} . \rho'[\overline{\tau} / \overline{\vec{a}}] \Rightarrow \lambda x : t . \text{pack} [\overline{\tau}]_x, e[\overline{\vec{f}} / \overline{\vec{a}}]$ as $\exists \overline{\vec{a}}' . r'[\overline{\vec{f}} / \overline{\vec{a}}]$ which concludes since $\lambda x : t . \text{pack} [\overline{\tau}]_x, e[\overline{\vec{f}} / \overline{\vec{a}}]$ as $\exists \overline{\vec{a}}' . r'[\overline{\vec{f}} / \overline{\vec{a}}] = (\lambda x : t . \text{pack} [\overline{\tau}]_x, e$ as $\exists \overline{\vec{a}}' . r'[\overline{\vec{f}} / \overline{\vec{a}}]$.

**Rule ELAB-CABS**: By induction hypothesis. We also use $fv(\sigma_1) \subseteq dom(\Gamma)$ to prove $\lambda x : s_1 . e[\overline{\vec{f}} / \overline{\vec{a}}] = (\lambda x : s_1 . e)[\overline{\vec{f}} / \overline{\vec{a}}]$.

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

**Rule ELAB-VAR**: Since $\vec{a} \notin dom(\Gamma)$, this means the $\vec{a}$ do not appear in $\sigma$ hence $\sigma[\overline{\tau} / \overline{\vec{a}}] = \sigma$ 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 $\Gamma \vdash^{\forall} h : \sigma \Rightarrow h ; \overline{\tau} \rightsquigarrow \overline{\vec{a}} ; \rho \Rightarrow e$. And transform it by applying the substitution $[\overline{\tau} / \overline{\vec{a}}]$ at every intermediary step. We show that it 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 $\vec{a} \notin dom(\Gamma)$, we conclude by noting that $\sigma[\overline{\tau} / \overline{\vec{a}}][\sigma' / \overline{\vec{a}}] = \sigma[\sigma' / \overline{\vec{a}}][\overline{\tau} / \overline{\vec{a}}]$. 

Rule **ELAB-IARG**: By case (1) of the Lemma, from $\Gamma \vdash^v e' \iff \sigma_1 \Rightarrow e'$ we obtain $\Gamma \vdash^v e' \iff \sigma_1[\overline{\tau / \overline{\alpha}}] \Rightarrow e'[\overline{\tau / \overline{\alpha}}]$. Hence we correctly have $\Gamma \vdash \text{inst } e e' : \sigma_2[\overline{\tau / \overline{\alpha}}] \Rightarrow (e e')[\overline{\tau / \overline{\alpha}}] : \overline{\tau} \Rightarrow \overline{\sigma}[\overline{\tau / \overline{\alpha}}] ; \rho_r[\overline{\tau / \overline{\alpha}}] \Rightarrow e_r[\overline{\tau / \overline{\alpha}}].$

Rule **ELAB-IALL**: We just notice that, since $fv(\tau) \subseteq dom(\Gamma)$, $\sigma[\overline{\tau / \overline{\alpha}}][\tau / a] = \sigma[\tau / a][\overline{\tau / \overline{\alpha}}].$

Rule **ELAB-IXISTCORE**: The rule applies with $\Gamma \vdash \text{inst } e : [e[\overline{\tau / \overline{\alpha}}]] / a] \Rightarrow \text{open } e[\overline{\tau / \overline{\alpha}}] ; \overline{\tau} \Rightarrow \overline{\sigma}[\overline{\tau / \overline{\alpha}}] ; \rho_r[\overline{\tau / \overline{\alpha}}] \Rightarrow e_r[\overline{\tau / \overline{\alpha}}].$ We conclude by noting that $e[[e[\overline{\tau / \overline{\alpha}}]] / a] = e[[e / a][\overline{\tau / \overline{\alpha}}]$ and $\text{open } e[\overline{\tau / \overline{\alpha}}] = (\text{open } e)[\overline{\tau / \overline{\alpha}}].$

Rule **ELAB-IRESULT**: $\Gamma \vdash \text{inst } e : \rho_r[\overline{\tau / \overline{\alpha}}] \Rightarrow e_r[\overline{\tau / \overline{\alpha}}] ; [\overline{\tau} \Rightarrow \overline{\sigma}[\overline{\tau / \overline{\alpha}}] ; \rho_r[\overline{\tau / \overline{\alpha}}] \Rightarrow e_r[\overline{\tau / \overline{\alpha}}]$ is true.

\[\square\]

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

1. If $\Gamma \vdash^v e \iff \sigma$, then $\Gamma \vdash^v e \iff \sigma[\overline{\tau / \overline{\alpha}}].$
2. If $\Gamma \vdash^v h \Rightarrow \sigma$, then $\Gamma \vdash^v h \Rightarrow \sigma[\overline{\tau / \overline{\alpha}}].$
3. If $\Gamma \vdash^v e \Rightarrow \rho$, then $\Gamma \vdash^v e \Rightarrow \rho[\overline{\tau / \overline{\alpha}}].$
4. If $\Gamma \vdash^v h \Rightarrow \sigma$ and $\Gamma \vdash \text{inst } h : \sigma ; \overline{\tau} \Rightarrow \overline{\sigma} ; \rho_r$, then $\Gamma \vdash \text{inst } h : \sigma[\overline{\tau / \overline{\alpha}}] ; \overline{\tau} \Rightarrow \overline{\sigma}[\overline{\tau / \overline{\alpha}}] ; \rho_r[\overline{\tau / \overline{\alpha}}].$

**Proof.** By corollary of Lemma B.6 \[\square\]

**Lemma B.8 (Substitution).** Suppose $\Gamma_1 \vdash e_1 \Rightarrow \rho_1 \Rightarrow e_1$ and take $\overline{\alpha} = fv(\rho_1) \setminus fv(\Gamma_1)$.

1. If $\Gamma_1, x:\forall \overline{\alpha}, \rho_1, \Gamma_2 \vdash e_2 \Rightarrow \rho_2$, then $\Gamma_1, \Gamma_2[\Lambda \overline{\alpha}.e_1 / x] \vdash e_2[\overline{\tau / \overline{\alpha}}] \Rightarrow \rho_2[\Lambda \overline{\alpha}.e_1 / x].$
2. If $\Gamma_1, x:\forall \overline{\alpha}, \rho_1, \Gamma_2 \vdash \text{inst } e_2 : \sigma$, then $\Gamma_1, \Gamma_2[\Lambda \overline{\alpha}.e_1 / x] \vdash \text{inst } e_2[\overline{\tau / \overline{\alpha}}] \Rightarrow \sigma[\Lambda \overline{\alpha}.e_1 / x].$
3. If $\Gamma_1, x:\forall \overline{\alpha}, \rho_1, \Gamma_2 \vdash \text{inst } h : \sigma ; \overline{\tau} \Rightarrow \overline{\sigma} ; \rho_r$, then $\Gamma_1, \Gamma_2[\Lambda \overline{\alpha}.e_1 / x] \vdash \text{inst } h[\overline{\tau / \overline{\alpha}}] : \sigma[\Lambda \overline{\alpha}.e_1 / x] ; \overline{\tau} \Rightarrow \overline{\sigma}[\overline{\tau / \overline{\alpha}}] ; \rho_r[\overline{\tau / \overline{\alpha}}].$
4. If $\Gamma_1, x:\forall \overline{\alpha}, \rho_1, \Gamma_2 \vdash h \Rightarrow \sigma$, then $\Gamma_1, \Gamma_2[\Lambda \overline{\alpha}.e_1 / x] \vdash h[\overline{\tau / \overline{\alpha}}] \Rightarrow \sigma[\Lambda \overline{\alpha}.e_1 / x].$

**Proof.** By induction on $e_2$.

$e_2 = x$: Then $\Gamma_1, x:\forall \overline{\alpha}, \rho_1, \Gamma_2 \vdash h \Rightarrow \rho_2$ implies $\rho_2 = \rho_1[\overline{\tau / \overline{\alpha}}].$ This means that $\rho_2[\Lambda \overline{\alpha}.e_1 / x] = \rho_1[\overline{\tau / \overline{\alpha}}][\Lambda \overline{\alpha}.e_1 / x].$ Since $x$ does not appear in $\rho_1$ (it is not in $\Gamma_1$, which is used to type $e_1$ with $\rho_1$), we have in fact $\rho_2[\Lambda \overline{\alpha}.e_1 / x] = \rho_1[\overline{\tau / \overline{\alpha}}][\Lambda \overline{\alpha}.e_1 / x] / \overline{\alpha}.$ Thus, since $\Gamma_1 \vdash e \Rightarrow \rho_1$ and $\overline{\alpha} \notin dom(\Gamma_1)$, by Lemma B.7 we obtain $\Gamma_1 \vdash e_1 = \rho_2[\Lambda \overline{\alpha}.e_1 / x]$, and then we conclude by weakening.

$e_2 = e :: \sigma$: By inversion on rules **APP** and rule **H-ANN**, we get $\Gamma_1, x:\forall \overline{\alpha}, \rho_1, \Gamma_2 \vdash e \iff \sigma$. By induction hypothesis, $\Gamma_1 \vdash^v e[\overline{\tau / \overline{\alpha}}] \Rightarrow \sigma[\Lambda \overline{\alpha}.e_1 / x].$ Then, since projections do not appear in type arguments, $\sigma[\Lambda \overline{\alpha}.e_1 / x] = \sigma$ and $\Gamma_1 \vdash^v e[\overline{\tau / \overline{\alpha}}] \Rightarrow \sigma$, and we conclude by applying rule **APP**.

$e_2 = \lambda y.e$: By inversion on rule **1ABS** and induction hypothesis, $\Gamma_1, y:\forall \overline{\alpha}, \rho_1, \Gamma_2 \vdash e[\overline{\tau / \overline{\alpha}}] \Rightarrow \rho[\Lambda \overline{\alpha}.e_1 / x].$ Hence $\Gamma_1 \vdash \lambda y.e[\overline{\tau / \overline{\alpha}}] \Rightarrow (\tau \Rightarrow \exists b.\rho)[\Lambda \overline{\alpha}.e_1 / x].$

$e_2 = \text{let } y = e_3 \text{ in } e_4$: By the induction hypothesis.

$e_2 = h \overline{\tau}$ with non-empty $\overline{\tau}$: By the induction hypothesis.

\[\square\]

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

1. $(\Gamma \vdash e_1 \Rightarrow \rho) \text{ implies } (\Gamma \vdash e_2[\overline{\tau / \overline{\alpha}}] \Rightarrow \rho)$
2. $(\Gamma \vdash^v e_1 \Rightarrow \sigma) \text{ implies } (\Gamma \vdash^v e_2[\overline{\tau / \overline{\alpha}}] \Rightarrow \sigma)$
Proof. (1) By inversion on the LetCore rule, we have

\[
\begin{align*}
\Gamma &\vdash e_1 \Rightarrow \rho_1 \Rightarrow e_1 \\
\Gamma, x \forall \overline{\alpha}. \rho_1 &\vdash e_2 \Rightarrow \rho' \\
\overline{\alpha} &\equiv fv(\rho_1) \setminus \text{dom}(\Gamma) \\
\rho &\equiv \rho'[\Lambda \overline{\alpha}. e_1 / x]
\end{align*}
\]

By Lemma B.8 we obtain \( \Gamma \vdash e_2[e_1 / x] \Rightarrow \rho'[\Lambda \overline{\alpha}. e_1 / x] \).

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

\[
\begin{align*}
\Gamma &\vdash e_1 \Rightarrow \rho_1 \Rightarrow e_1 \\
\Gamma, x \forall \overline{\alpha}. \rho_1 &\vdash e_2 \Leftarrow \rho' \\
\overline{\alpha} &\equiv fv(\rho_1) \setminus \text{dom}(\Gamma) \\
\rho &\equiv \rho'[\Lambda \overline{\alpha}. e_1 / x]
\end{align*}
\]

By Lemma B.8, we obtain \( \Gamma \vdash e_2[e_1 / x] \Leftarrow \rho'[\Lambda \overline{\alpha}. e_1 / x] \) i.e. \( \Gamma \vdash e_2[e_1 / x] \Leftarrow \rho \). We conclude by rule Gen.

\[\square\]

C DETAILS AND PROOFS ABOUT THE CORE LANGUAGE, \( \mathbb{F} \)

C.1 Typing rules

\[
\begin{align*}
\text{CE-Var} &\quad G \vdash \text{ok} \quad x : t \in G \quad \Rightarrow G \vdash x : t \\
\text{CE-Int} &\quad G \vdash \text{ok} \quad \Rightarrow G \vdash n : \text{Int} \\
\text{CE-Abs} &\quad G, x : t_1 \vdash e : t_2 \quad x \notin fv(t_2) \\
&\quad \Rightarrow G \vdash \lambda x : t_1 . e : t_1 \to t_2 \\
\text{CE-App} &\quad G \vdash e : \text{type} \quad G \vdash t_2 : t_1 \\
&\quad \Rightarrow G \vdash e \, \triangleright t_2 : t_1 \to t_2 \\
\text{CE-ForAll} &\quad G \vdash \exists a. t_2 : \text{type} \\
&\quad \Rightarrow G \vdash \exists a. t_2 : \exists a. t_2 \\
\text{CE-Exists} &\quad G \vdash e : \text{type} \\
&\quad \Rightarrow G \vdash \exists a. t_2 : \exists a. t_2 \\
\text{CE-Let} &\quad G \vdash e_1 : t_1 \\
&\quad G, x : t_1 \vdash e_2 : t_2 \\
&\quad \Rightarrow G \vdash \text{let } x = e_1 \text{ in } e_2 : t_2[e_1 / x] \\
\text{CE-Cast} &\quad G \vdash e : t_1 \\
&\quad G \vdash \gamma : t_2 \quad \sim t_2 \\
&\quad \Rightarrow G \vdash e \triangleright \gamma : t_2 \\
\text{CE-Var} &\quad G \vdash \text{ok} \quad a \in G \\
&\quad \Rightarrow G \vdash a : \text{type} \\
\text{CT-Base} &\quad G \vdash \text{ok} \\
&\quad \Rightarrow G \vdash t_1 : \text{type} \\
\text{CT-Abs} &\quad G, a : t \vdash t_2 : \text{type} \\
&\quad \Rightarrow G \vdash \forall a. t : \text{type} \\
\text{CT-Exists} &\quad G, a : t \vdash t_2 : \text{type} \\
&\quad \Rightarrow G \vdash \exists a. t : \text{type} \\
\text{CT-ForAll} &\quad G \vdash e : \text{type} \\
&\quad \Rightarrow G \vdash \text{fv}(e) \subseteq \text{dom}(G) \\
&\quad \Rightarrow G \vdash \text{ok} \\
\text{CT-Proj} &\quad G \vdash \text{ok} \\
&\quad \Rightarrow G \vdash \text{ok}
\end{align*}
\]

(Core expression typing)

(Core type well-formedness)
\[ G \vdash \gamma : t_1 \sim t_2 \]

(Core coercion typing)

\[
\begin{array}{ll}
\text{CG-REFL} & G \vdash t : \text{type} \\
G \vdash (t) : t \sim t & G \vdash \text{sym} \gamma : t_2 \sim t_1 \\
\text{CG-FORALL} & G, a \vdash \gamma : t_1 \sim t_2 \\
G \vdash \forall a \gamma : (\forall a.t_1) \sim (\forall a.t_2) & G \vdash \exists a \gamma : (\exists a.t_1) \sim (\exists a.t_2) \\
\text{CG-EXISTS} & G, a \vdash \gamma : t_3 \sim t_4 \\
G \vdash \eta : e_1 \sim e_2 & G \vdash \text{projpack} t, e \ as t_2 : t_2 \sim t_2 \\
\text{CG-INSTEXISTS} & G \vdash \gamma_1 : (\exists a.t_1) \sim (\exists a.t_2) \\
G \vdash \eta \mathbin{\vdash} x = a \mathbin{\vdash} \gamma_2 : t_3 \sim t_4 & G \vdash \text{nth}_n \gamma : t_n \sim t_n \\
\text{CG-INSTFORALL} & G \vdash \gamma_1 : (\forall a.t_1) \sim (\forall a.t_2) \\
G \vdash \gamma_2 : t_3 \sim t_4 & G \vdash \eta \mathbin{\vdash} \gamma_1 \mathbin{\vdash} x = a \mathbin{\vdash} \gamma_2 : t_3 \sim t_4 \\
\text{CG-INSTFORALL} & G \vdash \gamma : t_3 \sim t_4 \\
G \vdash \eta \mathbin{\vdash} x = a \mathbin{\vdash} \gamma_2 : t_3 \sim t_4 & G \vdash \eta \mathbin{\vdash} \gamma_1 \mathbin{\vdash} x = a \mathbin{\vdash} \gamma_2 : t_3 \sim t_4 \\
\text{CG-NTH} & G \vdash \gamma : t_3 \sim t_4 \\
G \vdash \text{projpack} t, e \ as t_2 : t_2 \sim t_2 & G \vdash \text{nth}_n \gamma : t_n \sim t_n \\
\end{array}
\]

(Core expression coercion typing)

\[
\begin{array}{ll}
\text{CH-COHERENCE} & G \vdash e : t_1 \mathbin{\vdash} \gamma : t_1 \sim t_2 \\
G \vdash e \triangleright \gamma : e \sim e \triangleright \gamma & G \vdash \text{step} e : e \sim e' \\
\end{array}
\]

(Core context well-formedness)

\[
\begin{array}{ll}
\text{C-NIL} & G \vdash \text{ok} \\
\text{C-TYPE} & a \notin \text{dom}(G) \\
G \vdash \text{ok} & G \vdash \text{ok} \\
\end{array}
\]

(Core operational semantics)

\[
\begin{array}{ll}
\text{CS-BETA} & G \vdash (\lambda x : t_1 e_1) e_2 \rightarrow e_1[e_2 / x] \\
G \vdash e_1 e_2 \rightarrow e'_1 e_2 & G \vdash e_1 e_2 \rightarrow e'_1 e_2 \\
\text{CS-TABS} & G \vdash \Lambda a. e \rightarrow \Lambda a. e' \\
G \vdash \Lambda a. (v \triangleright \gamma) \rightarrow (\Lambda a. v) \triangleright \gamma & G \vdash (\Lambda a. v) t \rightarrow v[t / a] \\
\text{CS-CODE} & G \vdash e t \rightarrow e't \\
G \vdash (v \triangleright \gamma) t \rightarrow v t \triangleright (\gamma \triangleright (v t)) & G \vdash \text{pack} t, e \ as t_2 \rightarrow \text{pack} t, e' as t_2 \\
\end{array}
\]

G \vdash v = \lambda x : t_1 e_0 \\
G \vdash \gamma_1 = \text{sym}(\text{nth}_0 \gamma) \\
G \vdash \gamma_2 = \text{nth}_1 \gamma \\
G \vdash (v \triangleright \gamma) e \rightarrow (v (e \triangleright \gamma_1)) \triangleright \gamma_2 \\
G \vdash \text{pack} t, e \ as t_2 \rightarrow \text{pack} t, e' as t_2
Richard A. Eisenberg, Guillaume Duboc, Stephanie Weirich, and Daniel Lee

CS-OpenPack

\[ G \vdash \text{open} \ (\text{pack} \ t, v \ as \ t_2) \rightarrow v \triangleright \langle t_2 \rangle \circ \text{(sym} \ (\text{projpack} \ t, v \ as \ t_2)) \]

CS-OpenPackCasted

\[ G \vdash \text{open} \ (\text{pack} \ t, (v \triangleright \gamma) \ as \ t_2) \rightarrow (v \triangleright \gamma) \triangleright \langle t_2 \rangle \circ \text{(sym} \ (\text{projpack} \ t, (v \triangleright \gamma) \ as \ t_2)) \]

CS-OpenCong

\[ G \vdash e : t \quad G \vdash e \rightarrow e' \quad G \vdash \text{open} \ e \rightarrow \text{open} \ e' \triangleright \langle t \rangle \circ \text{(sym} \ [\text{step} \ e]) \]

CS-OpenPull

\[ v = \text{pack} \ t_1, v_0 \ as \ \exists \ a. t_0 \quad G \vdash \text{open} \ (v \triangleright \gamma) \rightarrow (\text{open} \ v) \triangleright \gamma \circ [v \triangleright \gamma] \]

CS-Let

\[ G \vdash \text{let} \ x = e_1 \ in \ e_2 \rightarrow e_2[e_1 / x] \]

CS-CastCong

\[ G \vdash e \rightarrow e' \quad G \vdash e \triangleright \gamma \rightarrow e' \triangleright \gamma \]

CS-CastTrans

\[ G \vdash (v \triangleright \gamma_1) \triangleright \gamma_2 \rightarrow v \triangleright (\gamma_1 \triangleright \gamma_2) \]

C.2 Structural properties

Lemma C.1 (Context regularity).

1. If \( G \vdash e : t \), then \( G \vdash G \ ok \).
2. If \( G \vdash t : \text{type} \), then \( G \vdash G \ ok \).
3. If \( G \vdash \gamma : t_1 \sim t_2 \), then \( G \vdash G \ ok \).
4. If \( G \vdash \eta : e_1 \sim e_2 \), then \( G \vdash G \ ok \).

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 \( G, G' \ ok \), then \( G \ ok \).

Proof. Straightforward induction on the structure of \( G' \). □

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

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

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

Proof. By straightforward induction on \( G \vdash t : \text{type} \). In the case for rule \text{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 \( G, G'' \ ok \) and \( G' \vdash G' \ ok \), then \( G', G'' \ ok \).


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

1. If \( G, G' \ ok \), then \( G, a, x : t, G' \ ok \).
2. If \( G, G' \ ok \), then \( G, a, G' \ ok \).

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

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

\section*{Lemma C.7 (Permutation in contexts).} If $\vdash G_1, G_2, a, G_3 \text{ ok}$, then $\vdash G_1, a, G_2, G_3 \text{ ok}$.


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


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

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

\textbf{Rule CT-Var:} By appeal to Lemma C.8 and rule CT-Var.

\textbf{Rule CT-Base:} By the induction hypothesis and Lemma C.8.

\textbf{Rule CT-ForAll:} By the induction hypothesis.

\textbf{Rule CT-Exists:} By the induction hypothesis.

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

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

(1) If $\vdash e : t$, then $G' \vdash e : t$.

(2) If $\vdash \gamma : t_1 \sim t_2$, then $G' \vdash \gamma : t_1 \sim t_2$.

(3) If $\vdash \eta : e_1 \sim e_2$, then $G' \vdash \eta : e_1 \sim e_2$.

(4) If $\vdash e \rightarrow e'$, then $G' \vdash e \rightarrow e'$.

\textbf{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.

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

(1) If $\vdash e : t$, then $G, G' \vdash e : t$.

(2) If $\vdash \gamma : t_1 \sim t_2$, then $G, G' \vdash \gamma : t_1 \sim t_2$.

(3) If $\vdash \eta : e_1 \sim e_2$, then $G, G' \vdash \eta : e_1 \sim e_2$.

(4) If $\vdash e \rightarrow e'$, then $G, G' \vdash e \rightarrow e'$.

\textbf{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.

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

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

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

\textbf{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 \( \text{fv}(e) \subseteq \text{dom}(G) \).
2. If \( G \vdash \gamma : t_1 \sim t_2 \), then \( \text{fv}(\gamma) \subseteq \text{dom}(G) \).
3. If \( G \vdash \eta : e_1 \sim e_2 \), then \( \text{fv}(\eta) \subseteq \text{dom}(G) \).

**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 \( G, a, G_1 \vdash t_1 : \text{type} \) and \( G_1 \vdash t_2 : \text{type} \), then \( G, G_2[t_2 / a] \vdash t_1[t_2 / a] : \text{type} \).
2. If \( G, a, G_1 \) ok and \( G_1 \vdash t_2 : \text{type} \), then \( G, G_2[t_2 / a] \) 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 \( \vdash G_1, a, G_2 \) ok. The induction hypothesis tells us that \( \vdash G_1, G_2[t_2 / a] \) ok. We now have three cases:

- \( a' \in G_1 \): We must prove \( G_1, G_2[t_2 / a] \vdash a' : \text{type} \). This comes straight from \( \vdash G_1, G_2[t_2 / a] \) ok and \( a' \in G_1 \), by rule CT-VAR.
- \( a' = a \): We must prove \( G_1, G_2[t_2 / a] \vdash t_2 : \text{type} \). We are done by Lemma C.3.
- \( a' \in G_2 \): We must prove \( G_1, G_2[t_2 / a] \vdash a' : \text{type} \). This comes straight from \( \vdash G_1, G_2[t_2 / a] \) 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 \( G_1, a, G_2, a' \vdash t_0 : \text{type} \). We now use the induction hypothesis to get \( G_1, G_2[t_2 / a], a' \vdash t_0[t_2 / a] : \text{type} \) and finish with rule CT-EXISTS to get \( G_1, G_2[t_2 / a] \vdash \exists a'.t_0[t_2 / a] : \text{type} \) as desired.

**Rule CT-PROJ:** We know \( t_1 = [e] \), and inversion tells us that \( \vdash G_1, a, G_2 \text{ ok} \) and \( \text{fv}(e) \subseteq \text{dom}(G_1, a, G_2) \). We must prove \( G_1, G_2[t_2 / a] \vdash [e[t_2 / a]] : \text{type} \). The induction hypothesis tells us that \( \vdash G_1, G_2[t_2 / a] \) ok, so (using rule CT-PROJ) we must prove only that \( \text{fv}(e[t_2 / a]) \subseteq \text{dom}(G_1, G_2[t_2 / a]) \). This must be true, because \( a \) cannot be free in \( e[t_2 / a] \) and \( \text{dom}(G_2[t_2 / a]) = \text{dom}(G_2) \).

**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 \( G, x : t_2, G_1 \vdash t_1 : \text{type} \) and \( G_1 \vdash e_2 : t_2 \), then \( G_1, G_2[e_2 / x] \vdash t_1[e_2 / x] : \text{type} \).
2. If \( \vdash G_1, x : t_2, G_2 \text{ ok} \) and \( G_1 \vdash e_2 : t_2 \), then \( \vdash G_1, G_2[e_2 / x] \) 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 \( \vdash G_1, x : t_2, G_2 \text{ ok} \) and \( a \in G_1, x : t_2, G_2 \). We must prove \( G_1, G_2[e_2 / x] \vdash a : \text{type} \). The induction hypothesis
An Existential Crisis Resolved

... gives us that \( \vdash G_1, G_2[e_2/x] \text{ ok} \). And, noting that substitutions do not affect type variable bindings, we must have \( a \in G_1, G_2[e_2/x] \). 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 \( t_1 = [e] \); we must prove \( G_1, G_2[e_2/x] \vdash [e][e_2/x] : \text{ type} \).

We know

| \( \vdash G_1, x : t_2, G_2 \text{ ok} \) | inversion of rule CT-PROJ |
| \( \vdash \) G_1, \( \vdash \) G_2[e_2/x] \text{ ok} | \( f v(e) \subseteq dom(G_1, x : t_2, G_2) \) |
| \( \vdash \) G_1, \( \vdash \) G_2[e_2/x] \text{ ok} | inversion of rule CT-PROJ |
| \( \vdash \) G_1, \( \vdash \) G_2[e_2/x] \text{ ok} | induction hypothesis |
| \( \vdash \) G_1, \( \vdash \) G_2[e_2/x] \text{ ok} | \( f v(e[e_2/x]) \subseteq f v(e) \cup f v(e_2)\backslash \{x\} \) |
| \( \vdash \) G_1, \( \vdash \) G_2[e_2/x] \text{ ok} | \( f v(e[e_2/x]) \subseteq dom(G_1, G_2[e_2/x]) \) |
| \( \vdash \) G_1, G_2[e_2/x] \text{ ok} | rules of \( \subseteq \) |
| \( \vdash \) G_1, G_2[e_2/x] \text{ ok} | 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.

\[ \square \]

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

**Proof**. Straightforward induction on the definition of values.

...
rule CE-ABS, and we are thus done (noting that it must be that $fv(e_2)$ 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, $e_1 = \text{pack} \ t, e$ as $\exists \ a. t'$. We must show $G_1, G_2[e_2 / x] \vdash pack \ t[e_2 / x], e[e_2 / x] \ 
\exists \ a. t'[e_2 / x]$. Lemma C.15 gives us the first two premises of rule CE-PACK. We must show $G_1, G_2[e_2 / x] \vdash e[e_2 / x] : t'[e_2 / x][t[e_2 / x] / a]$. By the algebra of substitutions, the object of this judgment equals $t'[t / a][e_2 / x]$. By inversion on our original assumption, we know $G_1, x : t_2, G_2 \vdash e : t'[t / a]$. We are thus done by the induction hypothesis.

**Rule CE-OPEN:** Here, $e_1 = \text{open} e$, where $G_1, x : t_2, G_2 \vdash e : \exists \ a. t$ and $t_1 = [e] / a$. We must show $G_1, G_2[e_2 / x] \vdash \text{open} e[e_2 / x] : t[[e] / a][e_2 / x]$. The object of this judgment equals $t[e_2 / x][[e][e_2 / x] / a]$. To use rule CE-OPEN, we must show $G_1, G_2[e_2 / x] \vdash e[e_2 / x] : \exists \ a. t[e_2 / x]$. 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 $e_1 = (\lambda x_0 : t. e_3) \ e_4$ and $e'_1 = e_3[e_4 / x_0]$. We must show $G_1, G_2[e_2 / x] \vdash (\lambda x_0 : t[e_2 / x]. e_3[e_2 / x]) \ e_4[e_2 / x] \rightarrow e_3[e_4 / x_0][e_2 / x]$. Rule CS-BETA tells us $G_1, G_2[e_2 / x] \vdash (\lambda x_0 : t[e_2 / x]. e_3[e_2 / x]) \ e_4[e_2 / x] \rightarrow e_3[e_2 / x][e_4[e_2 / x] / x_0]$. 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-APPULL:** 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-TAPPULL:** 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.
Lemma C.18 (Type substitution in terms). Suppose \( G_1 \vdash t_2 : \text{type} \).

1. If \( G_1, a, G_2 \vdash e_1 : t_1 \), then \( G_1, G_2[t_2 / a] \vdash e_1[t_2 / a] : t_1[t_2 / a] \).
2. If \( G_1, a, G_2 \vdash \gamma_1 : t_0 \sim t_1 \), then \( G_1, G_2[t_2 / a] \vdash \gamma_1[t_2 / a] : t_0[t_2 / a] \sim t_1[t_2 / a] \).
3. If \( G_1, a, G_2 \vdash \eta_1 : e_0 \sim e_1 \), then \( G_1, G_2[t_2 / a] \vdash \eta_1[t_2 / a] : e_0[t_2 / a] \sim e_1[t_2 / a] \).
4. If \( G_1, a, G_2 \vdash e \longrightarrow e' \), then \( G_1, G_2[t_2 / a] \vdash e[t_2 / a] \longrightarrow e'[t_2 / a] \).

Proof. By mutual induction on the structure of \( G_1, a, G_2 \vdash e_1 : t_1 \). Similarto the case for rule CS-BETA.

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.19 (Object regularity). Suppose \( G_1 \vdash t : \text{type} \).

1. If \( G \vdash e : t \), then \( G \vdash t : \text{type} \).


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.

(2) If \( G \vdash \gamma : t_1 \sim t_2 \), then \( G \vdash t_1 : \text{type} \) and \( G \vdash t_2 : \text{type} \).

(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 \( \vdash G \text{ok} \) 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 \( \vdash G, x : t_1 \text{ok} \) by Lemma C.1. Thus, by Lemma C.12, we have \( G \vdash t_1 : \text{type} \). The induction hypothesis gives us \( G, x : t_1 \vdash t_2 : \text{type} \), but we also know that \( x \not\in \text{fv}(t_2) \). We can use Lemma C.9 to get \( G \vdash t_2 : \text{type} \), 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 \( G \vdash t_2 : \text{type} \). We must show \( G \vdash t_1[t_2 / a] : \text{type} \); we are thus done by Lemma C.14.

**Rule CE-PACK:** By inversion.

**Rule CE-OPEN:** We know \( e = \text{open} e_0 \), and (by inversion) \( G \vdash e_0 : \exists a.t_0 \). We must prove \( G \vdash t_0[\{e_0\} / a] : \text{type} \). The induction hypothesis tells us that \( G \vdash \exists a.t_0 : \text{type} \). Inversion by rule CT-EXISTS then tells us \( G, a \vdash t_0 : \text{type} \). To use Lemma C.14, we must now show \( G \vdash \{e_0\} : \text{type} \). To use rule CT-PROJ, we must now show the following:

\[
\vdash G \text{ok} \quad \text{This is from Lemma C.1.}
\]

\( \text{fv}(e_0) \subseteq \text{dom}(G) \): This is from Lemma C.13.

**Rule CT-PROJ** gives us \( G \vdash \{e_0\} : \text{type} \) and then Lemma C.14 gives us \( G \vdash t_0[\{e_0\} / a] : \text{type} \) 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, \( \gamma = \text{projpack} t_3, e \text{ as } t_4 \), and we must show \( G \vdash \{\text{pack} t_3, e \text{ as } t_4 \} : \text{type} \) and \( G \vdash t_3 : \text{type} \). Inversion on the typing judgment gives us \( G \vdash \text{pack} t_3, e \text{ as } t_4 : t_4 \). This can be so only by rule CE-PACK. We can thus invert again to get \( G \vdash t_3 : \text{type} \). 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 \( G \vdash t_3[t_5 / a] : \text{type} \) and \( G \vdash t_4[t_6 / a] : \text{type} \). Let’s focus on the first of these.

<table>
<thead>
<tr>
<th>We know</th>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G \vdash \forall a.t_3 : \text{type} )</td>
<td>induction hypothesis</td>
</tr>
<tr>
<td>( G, a \vdash t_3 : \text{type} )</td>
<td>inversion of rule CT-FORALL</td>
</tr>
<tr>
<td>( G \vdash t_5 : \text{type} )</td>
<td>induction hypothesis</td>
</tr>
<tr>
<td>( G, a \vdash t_3[t_5 / a] : \text{type} )</td>
<td>Lemma C.14</td>
</tr>
<tr>
<td>The derivation for ( G \vdash t_4[t_6 / a] : \text{type} ) is similar.</td>
<td></td>
</tr>
</tbody>
</table>
Rule **CG-InstExists**: In this case, we know \( \gamma = \gamma_1 \circ \gamma_2 \), with inversion giving us \( G \vdash \gamma_1 : (\exists a.t_3) : (\exists a.t_4) \) and \( G \vdash \gamma_2 : t_5 \rightarrow t_6 \). We must show \( G \vdash t_3 [t_5 / a] : \text{type} \) and \( G \vdash t_4 [t_6 / a] : \text{type} \). Let’s focus on the first of these.

<table>
<thead>
<tr>
<th>We know</th>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G \vdash \exists a.t_3 : \text{type} )</td>
<td>induction hypothesis</td>
</tr>
<tr>
<td>( G, a \vdash t_3 : \text{type} )</td>
<td>inversion of rule \text{CT-Exists}</td>
</tr>
<tr>
<td>( G \vdash t_5 : \text{type} )</td>
<td>induction hypothesis</td>
</tr>
<tr>
<td>( G \vdash t_3 [t_5 / a] : \text{type} )</td>
<td>Lemma C.14</td>
</tr>
</tbody>
</table>

The derivation for \( G \vdash t_4 [t_6 / a] : \text{type} \) is similar.

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

**Rule CH-Coherence**: By inversion, using rule \text{CE-Cast}.

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

\[ \square \]

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

**Proof.** By induction on the structure of \( G \vdash e \rightarrow e' \).

**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 \text{CE-Abs}, and we can thus conclude \( G, x : t_1 \vdash e_1 : t_2 \) and \( x \not\in fv(t_2) \). Lemma C.17 tells us \( G \vdash e_1[e_2 / x] : t_2[e_2 / x] \). But since \( x \not\in fv(t_2) \), 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 \( e = (v \triangleright y) e_2 \), where \( v = \lambda x : t_0.e_0 \).

<table>
<thead>
<tr>
<th>We know</th>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t = t_2 )</td>
<td>inversion on rule \text{CE-App}</td>
</tr>
<tr>
<td>( G \vdash (v \triangleright y) : t_1 \rightarrow t_2 )</td>
<td>inversion on rule \text{CE-App}</td>
</tr>
<tr>
<td>( G \vdash e_2 : t_1 )</td>
<td>inversion on rule \text{CE-App}</td>
</tr>
<tr>
<td>( G \vdash v : t_3 )</td>
<td>inversion on rule \text{CE-Cast}</td>
</tr>
<tr>
<td>( t_3 = t_4 \rightarrow t_5 )</td>
<td>inversion on rule \text{CE-Abs}  (using ( v = \lambda x : t_0.e_0 ))</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>We know</th>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G \vdash \Lambda a. (v \triangleright \gamma) : t$</td>
<td>assumption</td>
</tr>
<tr>
<td>$G, a \vdash v \triangleright \gamma : t_1$</td>
<td>inversion of rule CE-TAbs</td>
</tr>
<tr>
<td>$t = \forall a. t_1$</td>
<td>inversion of rule CE-TAbs</td>
</tr>
<tr>
<td>$G, a \vdash v : t_2$</td>
<td>inversion of rule CE-CAST</td>
</tr>
<tr>
<td>$G, a \vdash \gamma : t_2 \sim t_1$</td>
<td>inversion of rule CE-CAST</td>
</tr>
<tr>
<td>$G \vdash \forall a. \gamma : (\forall a. t_2) \sim (\forall a. t_1)$</td>
<td>rule CG-FORALL</td>
</tr>
<tr>
<td>$G \vdash \Lambda a. v : \forall a. t_2$</td>
<td>rule CE-TAbs</td>
</tr>
<tr>
<td>$G \vdash (\Lambda a. v) \triangleright \forall a. \gamma : \forall a. t_1$</td>
<td>rule CE-CAST</td>
</tr>
</tbody>
</table>

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 $e = (v \triangleright \gamma) t_0$ where $G \vdash v : \forall a. t_2$, and we must prove $G \vdash v t_0 \triangleright (\gamma @ \langle t_0 \rangle) : t$.

<table>
<thead>
<tr>
<th>We know</th>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G \vdash (v \triangleright \gamma) t_0 : t$</td>
<td>assumption</td>
</tr>
<tr>
<td>$G \vdash v \triangleright \gamma : \forall a. t_1$</td>
<td>inversion of rule CE-TApp</td>
</tr>
<tr>
<td>$G \vdash t_0 : \mathsf{type}$</td>
<td>inversion of rule CE-TApp</td>
</tr>
<tr>
<td>$t = t_1[t_0 / a]$</td>
<td>inversion of rule CE-TApp</td>
</tr>
<tr>
<td>$G \vdash \forall a. t_2) \sim (\forall a. t_1)$</td>
<td>inversion of rule CE-CAST</td>
</tr>
<tr>
<td>$G \vdash \langle t_0 \rangle : t_0 \sim t_0$</td>
<td>rule CG-REFL</td>
</tr>
<tr>
<td>$G \vdash \gamma @ \langle t_0 \rangle : t_2[t_0 / a] \sim t_1[t_0 / a]$</td>
<td>rule CG-INSTFORALL</td>
</tr>
<tr>
<td>$G \vdash v t_0 : t_2[t_0 / a]$</td>
<td>rule CE-TAPP</td>
</tr>
<tr>
<td>$G \vdash v t_0 \triangleright (\gamma @ \langle t_0 \rangle) : t_1[t_0 / a]$</td>
<td>rule CE-CAST</td>
</tr>
</tbody>
</table>

Rule CS-PackCong: By the induction hypothesis.
**Rule CS-OpenPack:** Here, we have $e = \text{open} (\text{pack} t_1, v_0 \text{ as } t_0)$.

<table>
<thead>
<tr>
<th>We know</th>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G \vdash \text{open} (\text{pack} t_1, v_0 \text{ as } t_0) : t$</td>
<td>assumption</td>
</tr>
<tr>
<td>$G \vdash \text{pack} t_1, v_0 \text{ as } t_0 : \exists a.t_2$</td>
<td>inversion of rule CE-Open</td>
</tr>
</tbody>
</table>

$t = t_2[[\text{pack} t_1, v_0 \text{ as } t_0]/a]$

| $G \vdash v_0 : t_2[t_1 / a]$ | inversion of rule CE-Pack |
| | inversion of rule CE-Pack |
| | inversion of rule CE-Pack |

$t_0 = \exists a.t_2$

| $G \vdash t_0 : \text{type}$ | |

| $G \vdash \langle t_0 \rangle : (\exists a.t_2) \sim (\exists a.t_2)$ | rule CG-Refl |
| $G \vdash \text{projpack} t_1, v_0 \text{ as } t_0 : [\text{pack} t_1, v_0 \text{ as } t_0] \sim t_1$ | rule CG-ProjPack |
| $G \vdash \text{sym} (\text{projpack} t_1, v_0 \text{ as } t_0) : t_3 \sim [\text{pack} t_1, v_0 \text{ as } t_0]$ | rule CG-Sym |
| $G \vdash \langle t_0 \rangle @(\text{sym} (\text{projpack} t_1, v_0 \text{ as } t_0)) : t_2[t_1 / a] \sim t_2[[\text{pack} t_1, v_0 \text{ as } t_0]/a]$ | rule CG-InstExists |
| $G \vdash v_0 \triangleright \langle t_0 \rangle @(\text{sym} (\text{projpack} t_1, v_0 \text{ as } t_0)) : t_2[[\text{pack} t_1, v_0 \text{ as } t_0]/a]$ | 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 $e = \text{open } e_0$. Inverting rule CE-Open in the derivation for $G \vdash \text{open } e_0 : t$ tells us $G \vdash e_0 : \exists a.t_2$ and $t = t_2[[e_0]/a]$. Given $G \vdash e_0 \rightarrow e_0'$, we must now show $G \vdash \text{open } e_0' \triangleright (\exists a.t_2) @ (\text{sym } \text{step } e) : t_2[[e_0]/a]$. 

---

We know

\[ G \vdash e_0' : \exists a.t_2 \]
\[ G \vdash \text{step} e_0 : e_0 \sim e_0' \]
\[ G \vdash [\text{step} e_0] : [e_0] \sim [e_0'] \]
\[ G \vdash \text{sym} [\text{step} e_0] : [e_0'] \sim [e_0] \]
\[ G \vdash \exists a.t_2 : \text{type} \]
\[ G \vdash \langle \exists a.t_2 \rangle : (\exists a.t_2) \sim (\exists a.t_2) \]
\[ G \vdash \langle \exists a.t_2 \rangle \circ (\text{sym} [\text{step} e_0]) : t_2[[e_0'] / a] \sim t_2[[e_0] / a] \]

How

induction hypothesis
rule CH-STEP
rule CG-Proj
rule CG-Sym
Lemma C.19
rule CG-RefL
rule CG-InstExists
rule CE-Open
rule CE-Cast

We are done with this case.

Rule CS-OpenPull: We have \( e = \text{open} (v \triangleright \gamma) \), where \( v = \text{pack} t_0, v_0 \) as \( \exists a.t_1 \).

We know

\[ G \vdash \text{open} (v \triangleright \gamma) : t \]
\[ G \vdash v \triangleright \gamma : \exists a.t_2 \]
\[ t = t_2[[v \triangleright \gamma] / a] \]
\[ G \vdash v : t_3 \]
\[ t_3 = \exists a.t_1 \]
\[ G \vdash \gamma : (\exists a.t_1) \sim (\exists a.t_2) \]
\[ G \vdash v \triangleright \gamma : v \sim v \triangleright \gamma \]
\[ G \vdash [v \triangleright \gamma] : [v] \sim [v \triangleright \gamma] \]
\[ G \vdash \gamma \circ [v \triangleright \gamma] : t_1[[v] / a] \sim t_2[[v \triangleright \gamma] / a] \]
\[ G \vdash \text{open} v : t_1[[v] / a] \]
\[ G \vdash \text{open} v \circ \gamma \circ [v \triangleright \gamma] : t_2[[v \triangleright \gamma] / a] \]

How

assumption
inversion of rule CE-Open
inversion of rule CE-Open
inversion of rule CE-Cast
inversion of rule CE-Pack
inversion of rule CE-Cast
use of rule CH-Cohere
rule CG-Proj
rule CG-InstExists
rule CE-Open
rule CE-Cast

Rule CS-Let: We have \( e = \text{let} x = e_1 \in e_2 \).

We know

\[ G \vdash \text{let} x = e_1 \in e_2 : t \]
\[ G \vdash e_1 : t_1 \]
\[ G, x : t_1 \vdash e_2 : t_2 \]
\[ t = t_2[e_1 / x] \]
\[ G \vdash e_2[e_1 / x] : t_2[e_1 / x] \]

How

assumption
inversion of rule CE-Let
inversion of rule CE-Let
inversion of rule CE-Let
Lemma C.17

Rule CS-CastCong: We have \( e = e_0 \triangleright \gamma \), where \( G \vdash e_0 \longrightarrow e_0' \). We must show \( G \vdash e_0' \triangleright \gamma : t \).

We know

\[ G \vdash e_0 : t_0 \]
\[ G \vdash \gamma : t_0 \sim t \]
\[ G \vdash e_0' : t_0 \]
\[ G \vdash e_0' \triangleright \gamma : t \]

How

inversion of rule CE-Cast
inversion of rule CE-Cast
induction hypothesis
rule CE-Cast

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

We know

\[ G \vdash v \triangleright \gamma_1 : t_1 \]
\[ G \vdash \gamma_2 : t_1 \sim t \]
\[ G \vdash v : t_2 \]
\[ G \vdash \gamma_1 : t_2 \sim t \]
\[ G \vdash \gamma_1 ; ; \gamma_2 : t_2 \sim t \]
\[ G \vdash v \triangleright (\gamma_1 ; ; \gamma_2) : t \]

How

inversion of rule CE-Cast
inversion of rule CE-Cast
inversion of rule CE-Cast
inversion of rule CE-Cast
rule CG-Trans
rule CE-Cast

\[ \square \]
### C.4 Progress

**Definition C.21 (Rewrite relation).** Define rewrite relations on types $t_1 \Rightarrow t_2$ and terms $e_1 \Rightarrow e_2$ with the rules below.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RT-Refl</strong></td>
<td>$t \Rightarrow t$</td>
</tr>
<tr>
<td><strong>RT-Base</strong></td>
<td>$t \Rightarrow t'$</td>
</tr>
<tr>
<td><strong>RT-ForAll</strong></td>
<td>$\forall a.t \Rightarrow \forall a.t'$</td>
</tr>
<tr>
<td><strong>RT-Exists</strong></td>
<td>$\exists a.t \Rightarrow \exists a.t'$</td>
</tr>
<tr>
<td><strong>RT-Proj</strong></td>
<td>$e \Rightarrow e'$</td>
</tr>
<tr>
<td><strong>RT-ProjPack</strong></td>
<td>$e \Rightarrow e'$</td>
</tr>
<tr>
<td><strong>RE-TApp</strong></td>
<td>$a \Rightarrow a'$</td>
</tr>
<tr>
<td><strong>RE-TBeta</strong></td>
<td>$t \Rightarrow t'$</td>
</tr>
<tr>
<td><strong>RE-Open</strong></td>
<td>$e \Rightarrow e'$</td>
</tr>
<tr>
<td><strong>RE-Let</strong></td>
<td>$e \Rightarrow e'$</td>
</tr>
<tr>
<td><strong>RE-TAbs</strong></td>
<td>$e \Rightarrow e'$</td>
</tr>
<tr>
<td><strong>RE-DropCo</strong></td>
<td>$e \Rightarrow e'$</td>
</tr>
<tr>
<td><strong>RE-AddCo</strong></td>
<td>$e \Rightarrow e'$</td>
</tr>
<tr>
<td><strong>RE-Cast</strong></td>
<td>$e \Rightarrow e'$</td>
</tr>
<tr>
<td><strong>RE-Beta</strong></td>
<td>$e \Rightarrow e'$</td>
</tr>
</tbody>
</table>

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

1. If $t_1 \Rightarrow t_2$, then $t_1 \Rightarrow^* t_2$.
2. If $e_1 \Rightarrow e_2$, then $e_1 \Rightarrow^* e_2$.

**Proof.** By mutual induction on the structure of $t_1 \Rightarrow t_2$ or $e_1 \Rightarrow e_2$. 

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

1. If $t_1 \Rightarrow^* t_2$, then $t_1 \Rightarrow^* t_2$.
2. If $e_1 \Rightarrow^* e_2$, then $e_1 \Rightarrow^* e_2$.

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

**Lemma C.25 (Substitution in rewrite relation).**

1. If $t_1 \Rightarrow t_2$, then $t_1 \Rightarrow t_2$.
2. If $e_1 \Rightarrow e_2$, then $e_1 \Rightarrow e_2$.

**Proof.** By mutual induction on the structure of $t_1 \Rightarrow t_2$ or $e_1 \Rightarrow e_2$. 

---

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

1. If $t_1 \Rightarrow^* t_2$, then $t_1[e_3/x] \Rightarrow^* t_2[e_3/x]$.
2. If $e_1 \Rightarrow^* e_2$, then $e_1[e_3/x] \Rightarrow^* e_2[e_3/x]$.

Proof. By induction on the length of the reduction.

Lemma C.27 (Lifting in rewrite relation). Assume $t_1 \Rightarrow t_2$.

1. For every $t_3$, $t_3[t_1/a] \Rightarrow t_3[t_2/a]$.
2. For every $e_3$, $e_3[t_1/a] \Rightarrow e_3[t_2/a]$.

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

$t_3 = a'$: We have two cases:

- $a' = a$: We are done by assumption.
- $a' \neq a$: We are done by rule RT-REFL.

$t_3 = B\tilde{t}$: By the induction hypothesis and rule RT-BASE.

$t_3 = \forall a'.t_4$: By the induction hypothesis and rule RT-FORALL.

$t_3 = \exists a'.t_4$: By the induction hypothesis and rule RT-EXISTS.

$t_3 = [e]$: By the induction hypothesis and rule RT-PROJ.

$e_3 = x$: By rule RE-REFL.

$e_3 = \lambda x:t.e$: By the induction hypothesis and rule RE-ABS.

$e_3 = e_1 e_2$: By the induction hypothesis and rule RE-APP.

$e_3 = \Lambda a.e$: By the induction hypothesis and rule RE-TABS.

$e_3 = e t$: By the induction hypothesis and rule RE-TAPP.

$e_3 = \text{pack } t e \text{ as } t'$: By the induction hypothesis and rule RE-PACK.

$e_3 = \text{open } e$: By the induction hypothesis and rule RE-OPEN.

$e_3 = \text{let } x = e_1 \text{ in } e_2$: By the induction hypothesis and rule RE-LETCONG.

$e_3 = e \triangleright y$: 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 $t_1 \Rightarrow^* t_2$.

1. For every $t_3$, $t_3[t_1/a] \Rightarrow^* t_3[t_2/a]$.
2. For every $e_3$, $e_3[t_1/a] \Rightarrow^* e_3[t_2/a]$.

Proof. By induction on the length of the reduction.

Lemma C.29 (Parallel substitution of a type). Assume $t_1 \Rightarrow t_2$.

1. If $t_3 \Rightarrow t_4$, then $t_3[t_1/a] \Rightarrow t_4[t_2/a]$.
2. If $e_3 \Rightarrow e_4$, then $e_3[t_1/a] \Rightarrow e_4[t_2/a]$.

Proof. By mutual induction on $t_3 \Rightarrow t_4$ or $e_3 \Rightarrow e_4$.

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.

Lemma C.30 (Parallel Substitution). Assume $e_1 \Rightarrow e_2$.

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 $t_1 = \text{[pack} t, e \text{as } \exists a.t_0] \text{ and } t_2 = [e'_0], \text{ where pack} t, e \text{ as } \exists a.t_0 \Rightarrow e'_0$. We further have $t_3 = t'$ where $t \Rightarrow t'$.

We know
\[
e'_0 = \text{pack} t'', e'' \text{ as } \exists a.t'_0''
\]
\[
t \Rightarrow t''
\]
\[
t'' \text{ such that } t' \Rightarrow t'' \text{ and } t'' \Rightarrow t''
\]
\[
\text{choose } t_4 = t''
\]
\[
t_2 \Rightarrow t''
\]

How
\[
\text{inversion of rule RE-PACK}
\]
\[
\text{inversion of rule RE-PACK}
\]
\[
\text{induction hypothesis}
\]
\[
\text{rule RT-PROJPACK}
\]

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 $e_1 \Rightarrow e_3$ was rewritten:

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

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

Rule RE-Abs: By the induction hypothesis.
Rule RE-App: We have two cases, depending on how $e_1 \Rightarrow e_3$ was rewritten:

- 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 $t_1 \Rightarrow t_2$, $e_5 \Rightarrow e_7$, and $e_6 \Rightarrow e_8$ (inverting rule RE-Abs)), and $e_3 = e_9[e_{10} / x]$ (where $e_5 \Rightarrow e_9$ and $e_6 \Rightarrow e_{10}$).

<table>
<thead>
<tr>
<th>We know</th>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_{11}$ such that $e_7 \Rightarrow e_{11}$ and $e_9 \Rightarrow e_{11}$</td>
<td>induction hypothesis</td>
</tr>
<tr>
<td>$e_{12}$ such that $e_8 \Rightarrow e_{12}$ and $e_{10} \Rightarrow e_{12}$</td>
<td>induction hypothesis</td>
</tr>
<tr>
<td>Choose $e_4 = e_{11}[e_{12} / x]$</td>
<td>rule RE-Beta</td>
</tr>
<tr>
<td>$e_2 \Rightarrow e_4$</td>
<td>Lemma C.30</td>
</tr>
<tr>
<td>$e_3 \Rightarrow e_4$</td>
<td>Lemma C.30</td>
</tr>
</tbody>
</table>

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 $e_1 \Rightarrow e_3$ was rewritten.


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

<table>
<thead>
<tr>
<th>We know</th>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_{11}$ such that $e_7 \Rightarrow e_{11}$ and $e_9 \Rightarrow e_{11}$</td>
<td>induction hypothesis</td>
</tr>
<tr>
<td>$e_{12}$ such that $e_8 \Rightarrow e_{12}$ and $e_{10} \Rightarrow e_{12}$</td>
<td>induction hypothesis</td>
</tr>
<tr>
<td>Choose $e_4 = e_{11}[e_{12} / x]$.</td>
<td>Lemma C.30</td>
</tr>
<tr>
<td>$e_2 \Rightarrow e_4$</td>
<td>Lemma C.30</td>
</tr>
<tr>
<td>$e_3 \Rightarrow e_4$</td>
<td>Lemma C.30</td>
</tr>
</tbody>
</table>

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 \( \exists x. t_1 \Rightarrow^* t_3 \) and \( \exists x. t_2 \Rightarrow^* t_3 \), 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 \( \exists x. t_1 \) and \( \exists x. t_2 \) is rule \( \text{RT-Exists} \). Accordingly, an inductive argument shows that \( t_3 \) must have the form \( \exists x. t_4 \) 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 \( \forall x. t_1 \Rightarrow^* t_3 \) and \( \forall x. t_2 \Rightarrow^* t_3 \), 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 \( B \Gamma \Rightarrow^* t_0 \) and \( B \Gamma' \Rightarrow^* t_0 \), 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 \( G \vdash e_1 \rightarrow e_2 \), then \( e_1 \Rightarrow e_2 \).

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

Rule CS-Beta: By rule \( \text{RE-Beta} \).
Rule CS-AppCong: By the induction hypothesis and rule \( \text{RE-App} \).
Rule CS-AppPull: By rules \( \text{RE-AddCo}, \text{RE-App}, \text{RE-DropCo}, \) and \( \text{RE-AddCo} \).
Rule CS-TabsCong: By the induction hypothesis and rule \( \text{RE-Tabs} \).
Rule CS-TabsPull: By rules \( \text{RE-AddCo}, \text{RE-Tabs}, \) and \( \text{RE-DropCo} \).
Rule CS-TBeta: By rule \( \text{RE-TBeta} \).
Rule CS-TAppCong: By the induction hypothesis and rule \( \text{RE-TApp} \).
Rule CS-TAppPull: By rules \( \text{RE-AddCo}, \text{RE-TApp}, \) and \( \text{RE-DropCo} \).
Rule CS-PackCong: By the induction hypothesis and rule \( \text{RE-Pack} \).
Rule CS-OpenPack: By rules \( \text{RE-OpenPack} \) and \( \text{RE-AddCo} \).
Rule CS-OpenPackCasted: By rules \( \text{RE-OpenPack} \) and \( \text{RE-AddCo} \).
Rule CS-OpenCong: By the induction hypothesis and rule \( \text{RE-Open} \).
Rule CS-OpenPull: By rules \( \text{RE-AddCo}, \text{RE-Open}, \) and \( \text{RE-DropCo} \).
Rule CS-Let: By rule \( \text{RE-Let} \).
Rule CS-CastCong: By the induction hypothesis and rule \( \text{RE-Cast} \).
Rule CS-CastTrans: By rules \( \text{RE-Cast} \) and \( \text{RE-DropCo} \).

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

<table>
<thead>
<tr>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G \vdash \gamma_1 : t_1 \sim t_4$</td>
</tr>
<tr>
<td>$G \vdash \gamma_2 : t_4 \sim t_2$</td>
</tr>
<tr>
<td>$t_5$ such that $t_1 \Rightarrow^* t_5$ and $t_4 \Rightarrow^* t_3$</td>
</tr>
<tr>
<td>$t_6$ such that $t_4 \Rightarrow^* t_6$ and $t_2 \Rightarrow^* t_6$</td>
</tr>
<tr>
<td>$t_7$ such that $t_5 \Rightarrow^* t_7$ and $t_6 \Rightarrow^* t_7$</td>
</tr>
</tbody>
</table>

We are done, as $t_1 \Rightarrow^* t_7$ and $t_2 \Rightarrow^* t_7$.

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-EXIST.

Rule **CG-INSTFORALL**: We are done by rule RE-INST.

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

We know

<table>
<thead>
<tr>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G \vdash \gamma_1 : (\exists a.t_4) \sim (\exists a.t_5)$</td>
</tr>
<tr>
<td>$G \vdash \gamma_2 : t_6 \sim t_7$</td>
</tr>
<tr>
<td>$t_8$ that is the join of $\exists a.t_4$ and $\exists a.t_5$</td>
</tr>
<tr>
<td>$t_9$ that is the join of $t_6$ and $t_7$</td>
</tr>
<tr>
<td>$t_{10}$ that is the join of $t_4$ and $t_5$</td>
</tr>
<tr>
<td>$t_4[t_6 / a] \Rightarrow^* t_{10}[t_6 / a]$</td>
</tr>
<tr>
<td>$t_5[t_7 / a] \Rightarrow^* t_{10}[t_7 / a]$</td>
</tr>
<tr>
<td>$t_{10}[t_6 / a] \Rightarrow^* t_{10}[t_9 / a]$</td>
</tr>
<tr>
<td>$t_{10}[t_7 / a] \Rightarrow^* t_{10}[t_9 / a]$</td>
</tr>
<tr>
<td>$t_{10}[t_6 / a]$ is the join of $t_4[t_6 / a]$ and $t_5[t_7 / a]$</td>
</tr>
</tbody>
</table>

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

\[\square\]

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

1. a base type $B_{\gamma'}$
2. a universal type $\forall a.t'$
3. an existential type $\exists a.t'$

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

\[
\text{head}(B_{\gamma'}) = B \\
\text{head}(\forall a.t) = \forall \\
\text{head}(\exists a.t) = \exists
\]

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

Proof. Straightforward case analysis on the structure of $v$. \[\square\]
Lemma C.41 (Preservation of Value Types). If \( t \) is a value type and \( t \Rightarrow^* t' \), then \( t' \) is a value type and \( \text{head}(t) = \text{head}(t') \).

Proof. By induction over the length of the chain \( t \Rightarrow^* t' \).

Zero steps: Trivial.

\( n + 1 \) steps: We have \( t_0 \) such that \( t \Rightarrow^* t_0 \) in \( n \) steps and that \( t_0 \Rightarrow t' \). The induction hypothesis tells us that \( t_0 \) is a value type and that \( \text{head}(t) = \text{head}(t_0) \). 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 \( \text{head}(t_0) = \text{head}(t') \).

Lemma C.42 (Consistency). If \( G \vdash \gamma : t_1 \leadsto t_2 \) and both \( t_1 \) and \( t_2 \) are value types, then \( \text{head}(t_1) = \text{head}(t_2) \).

Proof. Lemma C.37 gives us \( t_3 \) such that \( t_1 \Rightarrow^* t_3 \) and \( t_2 \Rightarrow^* t_3 \). Lemma C.41 then tells us that \( t_3 \) is a value type with \( \text{head}(t_3) = \text{head}(t_1) \). Another use of Lemma C.41 tells us that \( \text{head}(t_3) = \text{head}(t_2) \). By transitivity of equality, \( \text{head}(t_1) = \text{head}(t_2) \).

Lemma C.43 (Canonical Forms).

1. If \( G \vdash v : t_1 \rightarrow t_2 \), then there exist \( x \) and \( e \) such that \( v = \lambda x : t_1.e \).
2. If \( G \vdash v : \forall a.t \), then there exists \( v_0 \) such that \( v = \Lambda a.v_0 \).
3. If \( G \vdash v : \exists a.t \), then either:
   a. there exists \( t_0, v_0 \) and \( t_1 \) such that \( v = \text{pack}(t_0, v_0) \) as \( t_1 \), or
   b. there exists \( t_0, v_0, y_0 \), and \( t_1 \) such that \( v = \text{pack}(t_0, (v_0 \triangleright y_0)) \) as \( t_1 \).

Proof.

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

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

1. there exists \( e' \) such that \( G \vdash e \rightarrow e' \);
2. \( e \) is a value \( v \); or
3. \( e \) is a casted value \( v \triangleright \gamma \).

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, \( e = n \), a value.
Rule CE-ABS: Here, \( e = \lambda x : t_1.e_1 \), a value.
Rule CE-APP: We know \( e = e_1 e_2 \), with \( G \vdash e_1 : t_1 \rightarrow t_2 \) and \( G \vdash e_2 : t_1 \). Applying the induction hypothesis on the first of these yields three possibilities:

There exists \( e'_1 \) such that \( G \vdash e_1 \rightarrow e'_1 \): In this case, \( e_1 e_2 \) steps by rule CS-AppCong.
\( e_1 = v_1 \). Lemma C.43 tells us that \( v_1 = \lambda x : t_1.e_0 \). Thus, our original expression is \( e = (\lambda x : t_1.e_0) e_2 \), which can reduce by rule CS-Beta.
\( e_1 = v_1 \triangleright y_1 \). Thus, our original expression is \( e = (v_1 \triangleright y_1) e_2 \). In order to use rule CS-AppPull, we need only prove \( v_1 = \lambda x : t_3.e_0 \) for some \( t_3 \) and \( e_0 \).
We know $G \vdash (v_1 \triangleright y_1) e_2 : t$, assumption

$G \vdash v_1 \triangleright y_1 : t_4 \rightarrow t$, inversion of rule CE-APP

$G \vdash v_1 : t_5$, inversion of rule CE-Cast

$G \vdash y_1 : t_5 \sim (t_4 \rightarrow t)$, inversion of rule CE-Cast

$t_5$ is a value type, Lemma C.40

$t_5 = t_6 \rightarrow t_7$, 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 $G, a \vdash e_0 \rightarrow e'_0$:** We are done by rule CS-TABSCong.

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

$e_0 = v_0 \triangleright y_0$: 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 $G \vdash t_0 : \text{type}$. A use of the induction hypothesis on $e_0$ yields three cases:

**There exists $e'_0$ such that $G \vdash e_0 \rightarrow e'_0$:** 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.

$e_0 = v_0 \triangleright y_0$: We have $e = (v_0 \triangleright y_0) t_0$. To use rule CS-TAPPPull, we must show $G \vdash v_0 : \forall a. t_1$.

We know $G \vdash (v_0 \triangleright y_0) t_0 : t$, assumption

$G \vdash v_0 \triangleright y_0 : \forall a. t_3$, inversion of rule CE-TAPP

$G \vdash v_0 : t_4$, inversion of rule CE-Cast

$G \vdash y_0 : t_4 \sim \forall a. t_3$, 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 $e = \text{pack} t_0, e_0$ 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 $G \vdash e_0 \rightarrow e'_0$:** We are done by rule CS-PACKCong.

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

$e_0 = v_0 \triangleright y_0$: In this case, we have $e = \text{pack} t_0, (v_0 \triangleright y_0)$ as $\exists a. t_1$, which is a value.

**Rule CE-OPEN:** We know $e = \text{open} e_0$, 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 $G \vdash e_0 \rightarrow e'_0$:** 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.

$e_0 = v_0 \triangleright y_0$: In this case, we have $e = \text{open} (v_0 \triangleright y_0)$. To use rule CS-OPENPull, we must show only that $v_0 = \text{pack} t_1, v_1$ as $\exists a. t_0$. 

We know

<table>
<thead>
<tr>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G \vdash \text{open}(v_0 \triangleright y_0) : t$</td>
</tr>
<tr>
<td>$G \vdash v_0 \triangleright y_0 : \exists a.t_2$</td>
</tr>
<tr>
<td>$t = t_2[[v_0 \triangleright y_0] / a]$</td>
</tr>
<tr>
<td>$G \vdash v_0 : t_3$</td>
</tr>
<tr>
<td>$G \vdash y_0 : t_3 \sim \exists a.t_2$</td>
</tr>
<tr>
<td>$t_3$ is a value type</td>
</tr>
<tr>
<td>$v_0 = \text{pack} t_1, v_1 \text{ as } \exists a.t_0$</td>
</tr>
</tbody>
</table>

We are thus done by rule CS-OPENPULL.

**Rule CE-LET**: We are done by rule CS-LET.

**Rule CE-CAST**: We know $e = e_0 \triangleright y_0$, where $G \vdash e_0 : t_0$. We use the induction hypothesis on $e_0$ to get three cases:

- **There exists $e'_0$ such that** $G \vdash e_0 \rightarrow e'_0$: We are done by rule CS-CASTCong.
- $e_0 = v_0$: Then $e$ is a casted value $v_0 \triangleright y_0$ and we are done.
- $e_0 = v_0 \triangleright y_1$: We are done by rule CS-CASTTRANS.

□

### C.5 Erasure

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

$$\begin{align*}
\hat{e} &::= x | \lambda x.\hat{e} | \hat{e}_1 \cdot \hat{e}_2 | \text{let } x = \hat{e}_1 \text{ in } \hat{e}_2 | n \\
\hat{o} &::= \lambda x.\hat{e} | n
\end{align*}$$

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

$$\begin{align*}
|\hat{x}| &= x \\
|\lambda x:t.e| &= \lambda x.|e| \\
|\hat{e}_1 e_2| &= |\hat{e}_1| |e_2| \\
|\Lambda a.e| &= |e| \\
|e t| &= |e| \\
|\hat{\text{pack } t, e \text{ as } t_2}| &= |e| \\
|\text{open } e| &= |e| \\
|\text{let } x = e_1 \text{ in } e_2| &= \text{let } x = |e_1| \text{ in } |e_2| \\
|e \triangleright y| &= |e| \\
|n| &= n
\end{align*}$$

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

$$(\text{Single-step operational semantics})$$

<table>
<thead>
<tr>
<th>ES-BETA</th>
<th>ES-APP</th>
<th>ES-LET</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\lambda x.\hat{e}_1) \hat{e}_2 \rightarrow \hat{e}_1[\hat{e}_2 / x]$</td>
<td>$\hat{e}_1 \hat{e}_2 \rightarrow \hat{e}_1' \hat{e}_2'$</td>
<td>$\text{let } x = \hat{e}_1 \text{ in } \hat{e}_2 \rightarrow \hat{e}_2[\hat{e}_1 / x]$</td>
</tr>
</tbody>
</table>

**Lemma C.45 (Erasure Substitution).** For all expressions $e_1$ and $e_2$, $|e_1[|e_2| / x]| = |e_1[|e_2| / x]|$.

**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 ($\Rightarrow$)).** If $G \vdash e \rightarrow e'$, then either $|e| = |e'|$ or $|e| \rightarrow |e'|$.

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

- **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 $G \vdash e \rightarrow^* e'$, then $|e| \rightarrow^* |e'|$.

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