

Staging with Class

A Specification for Typed Template Haskell

NINGNING XIE, University of Cambridge, United Kingdom

MATTHEW PICKERING, Well-Typed LLP

ANDRES LÖH, Well-Typed LLP

NICOLAS WU, Imperial College London, United Kingdom

JEREMY YALLOP, University of Cambridge, United Kingdom

MENG WANG, University of Bristol, United Kingdom

Multi-stage programming using typed code quotation is an established technique for writing optimizing code generators with strong type-safety guarantees. Unfortunately, quotation in Haskell interacts poorly with type classes, making it difficult to write robust multi-stage programs.

We study this unsound interaction and propose a resolution, *staged type class constraints*, which we formalize in a source calculus $\lambda[\![\Rightarrow]\!]$ that elaborates into an explicit core calculus $F[\![\!]\!]$. We show type soundness of both calculi, establishing that well-typed, well-staged source programs always elaborate to well-typed, well-staged core programs, and prove beta and eta rules for code quotations.

Our design allows programmers to incorporate type classes into multi-stage programs with confidence. Although motivated by Haskell, it is also suitable as a foundation for other languages that support both overloading and quotation.

CCS Concepts: • **Software and its engineering** → **Functional languages; Semantics**; • **Theory of computation** → **Type theory**;

Additional Key Words and Phrases: Staging, Type Classes, Template Haskell

ACM Reference Format:

Ningning Xie, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. 2021. Staging with Class: A Specification for Typed Template Haskell. 1, 1 (November 2021), 74 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

Producing code with predictable performance is a difficult task that is greatly assisted by *staging annotations*, a technique which has been extensively studied and implemented in a variety of languages [Kiselyov 2014; Rompf and Odersky 2010; Taha and Sheard 2000] and used to eliminate abstraction overhead in many domains [Jonnalagedda et al. 2014; Krishnaswami and Yallop 2019; Schuster et al. 2020; Willis et al. 2020; Yallop 2017]. These annotations give programmers fine control over performance by instructing the compiler to generate code in one stage of compilation that can be used in another.

Authors' addresses: Ningning Xie, University of Cambridge, United Kingdom, nx213@cl.cam.ac.uk; Matthew Pickering, Well-Typed LLP, matthew@well-typed.com; Andres Löh, Well-Typed LLP, andres@well-typed.com; Nicolas Wu, Imperial College London, United Kingdom, n.wu@imperial.ac.uk; Jeremy Yallop, University of Cambridge, United Kingdom, jeremy.yallop@cl.cam.ac.uk; Meng Wang, University of Bristol, United Kingdom, meng.wang@bristol.ac.uk.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2021 Copyright held by the owner/author(s).

XXXX-XXXX/2021/11-ART

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

The classic example of staging is $\text{power } n \ k$, where the value n^k can be efficiently computed for a fixed k by generating code where the required multiplications have been unrolled and inlined. The qpower function shows its corresponding staged version where *Code* annotates the types of values that will be present dynamically at run time. Since k is to be provided as a fixed value at compile time, it remains a value of type *Int*.

```
power :: Int → Int → Int          qpower :: Int → Code Int → Code Int
power 0 n = 1                      qpower 0 qn = [ 1 ]
power k n = n * power (k - 1) n    qpower k qn = [ $(qn) * $(qpower (k - 1) qn) ]
```

Then in the definition of power5 , we can quote $n :: \text{Int}$ to create $[n] :: \text{Code Int}$, and splice the expression $\$(\text{qpower } 5 \ [n])$ to generate $n * (n * (n * (n * (n * 1))))$. By using the staged function, static information can be eliminated by partially evaluating the function at compile-time.

```
power5 :: Int → Int
power5 n = $(qpower 5 [n])  -- power5 n = n * n * n * n * n * 1
```

The code above is restricted to a fixed type *Int*, and it is natural to hope for a more generic version.

The incarnation of staged programming in Typed Template Haskell promises the benefits of *type classes*, one of the distinguishing features of Haskell [Hall et al. 1996; Peyton Jones et al. 1997], allowing a definition to be reused for any type that is qualified to be numeric:

```
npower :: Num a ⇒ Int → a → a      qnpower :: Num a ⇒ Int → Code a → Code a
npower 0 n = 1                      qnpower 0 qn = [ 1 ]
npower k n = n * power (k - 1) n    qnpower k qn = [ $(qn) * $(qnpower (k - 1) qn) ]
```

Thanks to type class polymorphism, this works when n has any fixed type that satisfies the *Num* interface, such as *Integer*, *Double* and countless other types.

It is somewhat surprising, then, that the following function fails to compile in the latest implementation of Typed Template Haskell in GHC 9.0.1:

```
npower5 :: Num a ⇒ a → a
npower5 n = $(qnpower 5 [n])  -- Error!
```

Currently, GHC complains that there is no instance for *Num a* available, which is strange because the type signature explicitly states that *Num a* may be assumed. But this is not the only problem with this simple example: in the definition of qnpower , the constraint is bound outside a quotation but is used inside. As we will see, this discrepancy leads to subtle inconsistencies, which can be used to show that the current implementation of type classes in Typed Template Haskell is *unsound*.

This paper sets out to formally answer the question of how a language with polymorphism and qualified types should interact with a multi-stage programming language, while preserving type soundness. In particular, inspired by Typed Template Haskell, we offer the following contributions:

- We formalize a source calculus $\lambda^{\llbracket \Rightarrow \rrbracket}$, which models two key features of Typed Template Haskell, type classes and multi-stage programming, and includes a novel construct, *staged type class constraints* that resolves the subtle interaction between the two (§3).
- We formalize a core calculus $F^{\llbracket \rrbracket}$, a polymorphic lambda calculus extended with quotations as a compilation target for multi-stage languages (§4). *Splice environments*, a key innovation in $F^{\llbracket \rrbracket}$, make evaluation order evident, avoiding the need for level-indexed evaluation, and support treating quotations opaquely, giving more implementation freedom about their form.
- We present a type-directed elaboration from $\lambda^{\llbracket \Rightarrow \rrbracket}$ to $F^{\llbracket \rrbracket}$, which combines our two key ideas: *dictionary-passing elaboration* of staged type class constraints, and elaboration of splices into splice environments (§5).

- We prove key properties of our formalism: (a) $F^{\llbracket \cdot \rrbracket}$ enjoys type soundness (§4.4), (b) *well-typed, well-staged source programs always elaborate to well-typed, well-staged core programs*, and thus $\lambda^{\llbracket \cdot \rrbracket}$ also enjoys type soundness (§5.4) and (c) splices and quotations are *dual*, building on the *axiomatic semantics* of $F^{\llbracket \cdot \rrbracket}$ (§6).

§7 provides a detailed comparison of our work here to the current implementation of Template Haskell in GHC. The full proofs of the stated lemmas and theorems are available in the appendix included in the supplementary materials. While this work has been motivated by Typed Template Haskell, we believe our work will be useful to designers and implementors of other languages which combine similar features and share many of the same challenges.

2 OVERVIEW

This section gives an overview of our work. We start by introducing the fundamental concepts of multi-staged programming, in the context of Typed Template Haskell.

2.1 Multi-Stage Programming

Multi-stage programming provides two standard staging annotations that allow construction and combination of program fragments:

- A *quotation* expression $\llbracket e \rrbracket$ is a representation of the expression e as program fragment in a future stage. If $e :: a$, then $\llbracket e \rrbracket :: \text{Code } a$.
- A *splice* expression $\$(e)$ extracts the expression from its representation e . If $e :: \text{Code } a$, then $\$(e) :: a$. By splicing expressions inside quotations we can construct larger quotations from smaller ones.

Given these definitions, it may seem that quotes and splices can be used freely so long as the types align; well-typed problems don't go wrong, as the old adage says. Unfortunately, things are not so simple: *type soundness* in multi-staged programming also requires programs to be *well-staged*.

2.2 The Level Restriction

The definition of well-stagedness depends on the notion of a *level*. Levels indicate the evaluation order of expressions, and well-stagedness ensures that program can be evaluated in the order of their levels, so that an expression at a particular level can only be evaluated when all expressions it depends on at previous levels have been evaluated. Formally, the *level* of an expression is an integer given by the number of quotes that surround it, minus the number of splices. In other words, starting from level zero, quotation increases the level of an expression while splicing decreases it. The level of an expression indicates when the expression is evaluated: (1) programs of negative levels are evaluated at compile time; (2) programs of level 0 are evaluated at runtime; and (3) programs of positive levels are at future unevaluated stages.

In the simplest setting, a program is well-staged if each variable is used only at the level in which it is bound (hereafter referred to as *the level restriction*). Using a variable in a different stage may simply be impossible, or at least require special attention. The following three example programs are all well-typed, but only the first, *timely*, is well-staged:

$timely :: \text{Code } (Int \rightarrow Int)$	$hasty :: \text{Code } Int \rightarrow Int$	$tardy :: Int \rightarrow \text{Code } Int$
$timely = \llbracket \lambda x \rightarrow x \rrbracket$	$hasty = \lambda y \rightarrow \(y)	$tardy = \lambda z \rightarrow \llbracket z \rrbracket$

In *timely*, the variable x is both introduced and used at level 1. (Similarly, in the well-staged example, *qpower*, in the introduction, the variables *qpower*, k and qn are introduced and used at level 0.) In the second program, *hasty*, the variable y is introduced at level 0, but used at level -1 . Evaluating the program would get stuck, because its value is not yet known at level -1 . In the third program,

tardy, the variable z is introduced at level 0, but used at level 1. Using a variable at a later stage in this way requires additional mechanisms to persist its value from one stage to another.

Relaxing the level restriction. Designers of multi-stage languages have developed several mechanisms for relaxing the level restriction to allow references to variables from previous stages [Hanada and Igarashi 2014; Taha and Sheard 1997]. *Lifting* makes a variable available to future stages by copying its value into a future-stage representation. Since lifting is akin to serialisation, it can be done easily for first-order types such as strings and integers, but not higher-order types. *Cross-stage persistence* (CSP) is more general than lifting: it supports embedding references to heap-resident values into quotations. Since it does not involve serialisation, CSP also supports persisting non-serialisable values such as functions and file handles. *Path-based persistence* is a restricted form of CSP for top-level¹ identifiers. Rather than persisting references to heap values, path-based persistence stores identifiers themselves, which can be resolved in the same top-level environment in future stages. For example, the top-level function *power* can be persisted in this way.

This work considers only path-based persistence. Fully-general CSP is limited to systems in which all stages are evaluated in the same process, since it requires sharing of heaps between stages; it is not available in systems such as Typed Template Haskell. Lifting is more broadly applicable, but it is straightforward to add separately as a local rewriting of programs. For example, GHC provides the *Lift* type class with a method *lift*, and instances of *Lift* for basic types like *Int*. Using these facilities, the ill-staged *tardy* can be rewritten into the well-staged *timelyLift*:

```
class Lift a where                timelyLift :: Int → Code Int
    lift :: a → Code a           timelyLift = λx → [ $(lift x) ]
```

2.3 Type Classes and the Level Restriction

The examples in the previous section demonstrate the importance of levels in a well-staged program in the simplest setting. However, other features found in real-world languages sometimes interact in non-trivial ways with multi-stage programming support. One such feature is *type classes* [Wadler and Blott 1989], a structured approach to overloading. Unfortunately, naive integration of type classes and staging poses a threat to type soundness. This section presents the problem, after a brief introduction to type classes and their dictionary-passing elaboration.

Type classes and dictionary-passing elaboration. The following presents the elements of type classes: the *Show* class offers an interface with one method *show*, the *Show Int* instance provides an implementation of *Show* for the type *Int* with a primitive *primShowInt*, and the *print* function uses the class method *show*; its type indicates that it can be used at any type *a* that has a *Show* instance.

```
class Show a where               instance Show Int where      print :: Show a ⇒ a → String
    show :: a → String           show = primShowInt         print x = show x
```

Type classes do not have direct operational semantics; rather, they are implemented by *dictionary-passing elaboration* into a simpler language without type classes (e.g. System F). After elaboration, a type class definition becomes a *dictionary* (i.e. a record type with a field for each class member), an instance becomes a value of the dictionary type, and each function that uses class methods acquires an extra parameter for the corresponding dictionary:

```
data ShowDict a = ShowDict      showInt = ShowDict          print' :: ShowDict a → a → String
    { show' :: a → String }      { show' = primShowInt }      print' dShow x = show dShow x
```

¹Do not confuse this use of “top-level” with the staging level.

The problem of staging type class methods. Constraints introduced by type classes have the potential to break type soundness, as implicit dictionary passing may not adhere to the level restriction. For example, in the following program, the class method *show* appears inside a quotation. Note the change of the function return type from $a \rightarrow \text{String}$ to $\text{Code } (a \rightarrow \text{String})^2$.

```
print1 :: Show a  $\Rightarrow$  Code (a  $\rightarrow$  String)
print1 = [ show ]
```

(C1)

Is *print1* well-staged? It appears so, since *print1* only uses the top-level class method *show*, which is path-based persisted. However, a subtle problem reveals itself after type class elaboration:

```
print1' :: ShowDict a  $\rightarrow$  Code (a  $\rightarrow$  String)
print1' dShow = [ show dShow ]
```

After elaboration, *print1'* takes an additional dictionary argument $dShow :: \text{ShowDict } a$. Notice that the dictionary variable *dShow* is introduced at level 0, but is used at level 1! Naively elaborating without considering the *levels of constraints* has introduced a cross-stage reference, making *print1* ill-staged. As §2.2 outlined, one possible remedy is to persist *dShow* between stages, a solution once advocated by [Pickering et al. 2019]. Although dictionaries are typically higher-order, they are ultimately constructed from path-persistable top-level values. However, the additional run-time overhead associated with this approach has led its erstwhile advocates to abandon it as impractical.

In contrast, the following monomorphic definition of *printInt* remains well-staged even after dictionary-passing elaboration into *printInt'*, since the constraint is resolved to a global instance *showInt* (which can be path-based persisted) rather than abstracted as a local variable. But of course this version does not enjoy all the benefits of type classes.

```
printInt :: Code (Int  $\rightarrow$  String)
printInt = [ show ]

printInt' :: Code (Int  $\rightarrow$  String)
printInt' = [ show showInt ]
```

(C2)

The problem of splicing type class methods. The interaction of *splicing* and dictionary-passing elaboration can also be subtle. In particular, splices that appear in top-level definitions may require class constraints to be used at levels prior to the ones where they are introduced. Consider the definition of *topLift*:

```
data C = C
topLift :: Lift C  $\Rightarrow$  C
topLift = $(lift C)

topLift' :: LiftDict C  $\rightarrow$  C
topLift' dLift = $(lift dLift C)
```

(TS1)

As with C1, although *topLift* appears to be well-staged, elaboration reveals that it is not, since it produces a future-stage reference inside the splice: the dictionary *dLift* is introduced at level 0 but is used at level -1. Unlike the case of C1, there is no remedy here, and the code should be rejected, as *dLift* is not known until runtime, and thus cannot be used in compile-time evaluation.

2.4 Staging Type Classes: an Exploration of the Design Space

Up to this point we have focused on the problems of type unsoundness arising from the interaction between quotation/splicing and type classes. We now turn to an exploration of potential solutions. Since there is little formal work in this area, our remarks here focus on designs that have been implemented in GHC. This section discusses the problems with each of these designs, and §7 includes a more detailed comparison with GHC.

Delaying type class elaboration until splicing. One approach to resolving Example C1 is to delay dictionary-passing elaboration until the program is spliced. With this approach, code values

²This example is an eta-reduced version of $\text{print1} = [\lambda x \rightarrow \text{show } x]$. For simplicity, we omit the argument *x*.

represent *source* programs rather than *elaborated* programs. For **C1** this means that *print1* is not elaborated, and so the problem with its ill-staged elaboration *print1'* does not arise. Instead, splicing *print1* first inserts its source code and then performs dictionary-passing elaboration, at which point we can provide the dictionary as per normal.

```
universe :: String                universe' :: String
universe = $(print1) 42          universe' = show showInt 42
```

However, as [Pickering et al. \[2019\]](#) observe, not preserving dictionary information in quotations can also threaten soundness. For example, the *readInt* function below uses the built-in function *read* :: *Read a* \Rightarrow *String* \rightarrow *a*, which converts a *String* into some *Read* instance (e.g. *Int*).

```
printInt :: Code (Int  $\rightarrow$  String)    readInt :: Code (String  $\rightarrow$  Int)
printInt = [ show ]                 readInt = [ read ]
```

Like Example **C2**, we expect that the global instance *readIntPrim* can be used to resolve *Read Int* in *readInt*. If so, then the following function composition would have a clear meaning, which trims spacing around a string representing an integer by first reading it into an integer and then print it:

```
trim :: Code (String  $\rightarrow$  String)
trim = [ $(printInt)  $\cdot$  $(readInt) ]
```

(A1)

Unfortunately, if dictionary information is not preserved in quotations, and we only do dictionary-passing elaboration when splicing *trim*, i.e., in $\$(trim)$, then any use of $\$(trim)$ would be rejected, as its spliced result *print* \cdot *read* is a typical example of an *ambiguous type scheme* [\[Jones 1993\]](#), i.e., *print* \cdot *read* is of type $(Show\ a, Read\ a) \Rightarrow Code\ (String \rightarrow String)$, where the dictionary to be used cannot be decided deterministically. Moreover, even when there is no such ambiguity, this approach may still accidentally change the semantics of a program, for example when the definition site and the splicing site have different instances³.

Excluding local constraints for top-level splices. One tempting solution to address the problem of splicing-type-class-methods mentioned above (Example **TS1**) is to exclude local constraints from the scope inside top-level splices. After all, top-level splices require compile time evaluation, and local constraints will not be available during compile time. While this approach can correctly reject **TS1**, it unfortunately cannot handle the combination of quotations and splices properly. In particular, programs like the following may be unnecessarily rejected.

```
cancel :: Show a  $\Rightarrow$  a  $\rightarrow$  String
cancel = [ show ]
```

(A2)

In this case, the body of the top-level splice is a simple quotation of the *show* method. This method requires an *Show* constraint which is provided by the context on *cancel*. The constraint is introduced at level 0 and also used at level 0, as the splice and the quotation cancel each other out. It is therefore perfectly fine to use the dictionary passed to *cancel* to satisfy the requirements of *Show*.

Impredicativity. Forthcoming versions of GHC are expected to feature *impredicativity*, allowing type variables to be instantiated by polymorphic types [\[Serrano et al. 2020\]](#). At a first glance, impredicativity appears to resolve the difficulty; furthermore, it naturally extends to include other features such as *quantified constraints* [\[Bottu et al. 2017\]](#).

For our example, impredicativity allows *print* to be given the following type, indicating that the code returned is polymorphic in the *Show* instance:

³In GHC, this requires language pragmas for *overlapping instances*, which allows resolving class constraints using more specific instances, and is not uncommon in practice. For example, a module can have both `instance Eq [Int]` and `instance Eq [a]`, and the former will be used to resolve `Eq [Int]`, and the latter can resolve, for example, `Eq [Bool]`.


```

295 printImp :: Code (Show a  $\Rightarrow$  a  $\rightarrow$  String)
296 printImp =  $\llbracket$  show  $\rrbracket$ 
297

```

At a small scale, this neatly solves the problem: the type indicates that the constraint *Show a* elaborates to a level 1 parameter, making the generated code well-staged. However, in larger examples, using impredicativity in this way severely limits the flexibility of staged functions. For example, here is an alternative definition of *qnpower* using impredicativity:

```

302
303 qnpowerImp :: Int  $\rightarrow$  Code (Num a  $\Rightarrow$  a)  $\rightarrow$  Code (Num a  $\Rightarrow$  a)
304 qnpowerImp 0 qn =  $\llbracket$  1  $\rrbracket$ 
305 qnpowerImp k qn =  $\llbracket$  $(qn) * (qnpowerImp (k - 1) qn)  $\rrbracket$ 
306

```

As with *printImp*, the types indicate that *qnpowerImp* is well-staged: the positions of the *Num a* constraints beneath *Code* indicate that they elaborate to level 1 parameters. Unfortunately, the type of the parameter *qn* now places additional demands on callers. The unstaged polymorphic *npower* function accepts an expression of any numeric type *a* as its second argument, and it would be convenient for its staged counterpart to accept an expression of any future-stage numeric type *Code a*. Instead, *qnpowerImp* demands an argument of type *Code* (*Num a* \Rightarrow *a*): even if it is called at a monomorphic type such as *Int*, the argument must still have type *Code* (*Num Int* \Rightarrow *Int*). This requirement has unfortunate effects on usability: such arguments cannot be of type *Code Int*, since *Code Int* is not a subtype of *Code* (*Num Int* \Rightarrow *Int*) (in the latest GHC). This is a significant loss of flexibility for callers. Further studies, beyond the scope of this paper, would be needed to support such subtyping while preserving impredicativity. Moreover, the requirement also leads to reduced control over generated code, which will be strewn with many additional dictionary abstractions and applications in generated code involving type classes. It may be possible to eliminate some of these in subsequent compiler passes but many of those passes are based on heuristics. Relying on compiler optimizations does not produce predictable program performance: it is almost impossible to tell by inspection how a program will be optimized.

2.5 Our Proposal: Staged Type Class Constraints

As we have seen, the interaction of staging and type-class elaboration is complicated, which cannot be managed by simply imposing additional restrictions on either one. A targeted solution that properly combines the two processes and restores type soundness is therefore needed.

Our proposal is to introduce *staged type class constraints*, a new constraint form *CodeC C* indicating that the constraint *C* has been *staged*. That is, we can use the staged constraint *CodeC C* to prove a constraint *C* in the next stage. With staged type class constraints, we can establish type soundness by enforcing well-stageness of constraints and dictionaries, and thus ill-staged use of constraints (e.g. *print1* and *topLift*) can be correctly rejected. To illustrate the idea, let us reconsider the problematic example *print1* in C1. We rewrite the example to *print2* with a staged type class constraint in its new type signature as follows.

```

335 print2 :: CodeC (Show a)  $\Rightarrow$  Code (a  $\rightarrow$  String) -- originally Show a  $\Rightarrow$  Code (a  $\rightarrow$  String) (S1)
336 print2 =  $\llbracket$  show  $\rrbracket$ 
337

```

This example illustrates the key idea of staged type class constraints. First, during typing, we use the *CodeC* (*Show a*) constraint to resolve the *Show a* constraint raised by *show*. Notably, the *CodeC* (*Show a*) constraint is introduced at level 0 but the *Show a* constraint is resolved at level 1. That means, staged type classes have the static semantics that *a constraint CodeC C at level n is equivalent to a constraint C at level n + 1*.

Second, in order to elaborate the expression with dictionary-passing, we need a dictionary representation of $\text{CodeC } C$. Fortunately, we already have all necessary machinery within the language – since dictionaries become regular data structures after elaboration, staging annotations can effectively convert between a dictionary for $\text{CodeC } C$ and a dictionary for C . That means, staged type class constraints have the simple elaboration semantics that *a dictionary for a constraint $\text{CodeC } C$ is a representation of the dictionary for a constraint C* .

Applying this elaboration semantics to *print2* produces the following code:

```
print2' :: Code (ShowDict a) → Code (a → String)
print2' cdShow = [ show $(cdShow) ]
```

The type $\text{Code } (\text{ShowDict } a)$ is the elaboration of the constraint $\text{CodeC } (\text{Show } a)$, and so *cdShow* is the representation of a dictionary, and can be spliced inside the quote as the dictionary argument to *show*. Crucially, the reference to *cdShow* is at the correct level, and so the program is type-safe.

The power function revisited. Recall the *qnpower* example in the introduction (§1):

```
qnpower :: Num a ⇒ Int → Code a → Code a
```

Just as *print1* in Example C1, the definition had to be rejected because of the ill-stagedness of the constraints. Using staged class constraints, we argue that the function *power* should instead have the constraint $\text{CodeC } (\text{Num } a)$, which then gets elaborated to $\text{Code } (\text{NumDict } a)$:

<pre>qnpower :: CodeC (Num a) ⇒ Int → Code a → Code a qnpower 0 cn = [1] qnpower k cn = [\$(cn) * \$(qnpower (k - 1) cn)]</pre>	<pre>qnpower' :: Code (NumDict a) → Int → Code a → Code a qnpower' cdNum 0 cn = [1] qnpower' cdNum k cn = [(*) \$(cdNum) \$(cn) \$(qnpower (k - 1) cn)]</pre>
---	---

(S2)

The elaboration of *npower5* then shows how C can be converted into $\text{CodeC } C$ by quoting:

<pre>npower5 :: Num a ⇒ a → a npower5 n = \$(qnpower 5 [n])</pre>	<pre>npower5' :: NumDict a → a → a npower5' dNum n = \$(qnpower' [dNum] 5 [n])</pre>
---	--

In this case, by quoting *dNum*, the argument to *qnpower'* is a representation of a dictionary (i.e., $[\text{dNum}] :: \text{Code } (\text{NumDict } a)$) as will be required by the elaborated type of $\text{CodeC } (\text{Num } a)$. Moreover, all variables in the definitions are well-staged.

2.6 Staging with Levels at Runtime

Besides formalizing staged type class constraints, our work also offers a guideline for implementation. In order to provide a robust basis for real-world languages such as Typed Template Haskell, we want our formalism to be easy to implement and to stay close to existing implementations.

One question, then, is how to evaluate staging programs. The *level* of an expression, described earlier, indicates when the expression is evaluated: expressions with negative levels are evaluated at compile time, those with level 0 at runtime, and those with positive levels in future stages. Ensuring a well-staged evaluation order involves access the level information during evaluation. For example, evaluating the following expression at runtime (level 0) involves evaluating e_1 and e_3 , but not e_2 :

$$(e_1, [e_2 \$(e_3)])$$

This is often done by level-indexing the reduction relation [Calcagno et al. 2003; Taha and Sheard 1997]. For example, during evaluation, we can traverse the quotation $[e_2 \$(e_3)]$, modifying the level (initially 0) when quotations or splices are encountered, looking for expressions of level 0 to

evaluate. This approach requires tracking of levels during runtime, adding complexity to implementations. Furthermore, as the above example illustrates, it requires inspecting and evaluating inside quotations. But in realistic implementations, quotations are compiled to a representation form for which implementing substitution can be difficult. In particular, previous implementations with low-level representations of quotations [Pickering et al. 2019; Roubinchtein 2015] maintain separate environments for free variables which can be substituted into without having to implement substitution in terms of the low-level representation.

2.7 Our Design: Splice Environments

We present a formalism that is easy to implement and reason about, by introducing quotations with *splice environments* in our core calculus F^{\square} . Splice environments capture *splices inside quotations*, avoiding the need to traverse quotations before splicing them into programs, and allowing quotations to be treated in an opaque manner that imposes few constraints on their representation. Splice environments also make the evaluation order of the core calculus evident, avoiding the need for level-indexed reduction. Using splice environments is reminiscent of the approach taken in logically inspired languages by Nanevski [2002] and Davies and Pfenning [2001].

A quotation with a splice environment is denoted $\llbracket e \rrbracket_{\phi}$, where e is a quoted expression and ϕ the splices it contains. ϕ consists of a list of *splice variables*, with each splice variable s represented as a *closure*. For example, our previous expression $(e_1, \llbracket e_2 \$(e_3) \rrbracket)$ is represented as follows in F^{\square} (assuming e_1, e_2 and e_3 contain no other splices).

$$(e_1, \llbracket e_2 s \rrbracket_{\bullet \uparrow^0 s : \tau = e_3})$$

There are several points to note. First, the splice $\$(e_3)$ is replaced by a fresh splice variable s , bound in the splice environment of the quotation. All splices in quotations will be similarly lifted, so that quotations no longer contain splices; in fact, F^{\square} has no splices, only splice environments.

Second, the splice variable s captures four elements:

- (1) the spliced expression (e_3).
- (2) the type context (\bullet). Here the type context is empty, but in general the expression may contain free variables, which the type context tracks.
- (3) the level of the expression. Here, e_3 is of level 0.
- (4) the type (τ) after splicing. If e_3 is of type $\text{Code } \tau$ then $\$(e_3)$ is of type τ .

Those elements imply that the splice variable s , representing $\$(e_3)$, is at level 1 and of type τ .

Finally, the splice environment contains only expressions of level 0, and is itself bound to a quotation of the same level (i.e., the whole quotation $\llbracket e_2 s \rrbracket_{\bullet \uparrow^0 s : \tau = e_3}$ is at level 0). This is an invariant maintained in the core calculus: a splice is bound immediately to the innermost surrounding quotation at the same level.

Now evaluation can be described straightforwardly, without the need to track levels or inspect quotations. Evaluation initially proceeds as if there is no staging. When it encounters a quotation $\llbracket e \rrbracket_{\phi}$, rather than inspecting e , it evaluates its splice environments ϕ , which are exactly those splices inside the quotation that should be evaluated in the current stage. In the above example, at level 0, evaluation starts with e_1 , then proceeds to the quotation $\llbracket e_2 s \rrbracket_{\bullet \uparrow^0 s : \tau = e_3}$ and moves to its splice environment $\bullet \uparrow^0 s : \tau = e_3$, which in turn evaluates e_3 . As this description makes clear, evaluating the expression evaluates e_1 and e_3 as desired. In more complex examples, nested quotations and splices produce nested quotations and splice environments, but the evaluation principle is the same.

Compile-time evaluation and top-level splice definitions. As we have said, splice environments bind each splice to the innermost surrounding quotation at the same level. This scheme does not account for the case of splices of negative levels which have no such enclosing quotation, such as

top-level splices. Since splices of negative levels are exactly those expressions that are evaluated at compile-time, we lift the corresponding splice environments to top-level as *splice definitions*

$$\text{spdef} \bullet \vdash^{-1} s : \tau = e$$

and put them *before* the rest of the program. This also gives meaning to *compile-time evaluation* in our formalism, where it is modeled using top-level splice definitions, whose evaluation happens before the rest of the program. We might also imagine a post-elaboration process which partially evaluates a program to a residual by computing and removing these splice definitions. Such a process can be easily implemented separately, so we do not include it in the formalism.

3 $\lambda^{\llbracket \Rightarrow \rrbracket}$: MULTI-STAGE PROGRAMMING WITH TYPE CLASSES

We present $\lambda^{\llbracket \Rightarrow \rrbracket}$, which has been designed to incorporate the essential features of a language with staging and qualified types, with the key novelty in the formalism of staged type class constraints.

3.1 Syntax

Figure 1 presents the syntax of our source calculus $\lambda^{\llbracket \Rightarrow \rrbracket}$. The syntax of type classes follows closely that of Bottu et al. [2017]; Chakravarty et al. [2005]; Jones [1994].

A source program *pgm* is a sequence of top-level definitions \mathcal{D} , type class declarations \mathcal{C} , and instance definitions \mathcal{I} , followed by an expression e . Top-level definitions \mathcal{D} ($k = e$) model path-based cross-stage persistence: only variables previously defined in a top-level definition can be referenced at arbitrary levels. The syntax of type class declarations \mathcal{C} is largely simplified to avoid clutter in the presentation. In particular, type class definitions $\text{TC } a \text{ where } \{k : \rho\}$ have precisely one method and no superclasses. Instance definitions $\overline{C}_i^i \Rightarrow \text{TC } \tau \text{ where } \{k = e\}$ are permitted to have an instance context, which is interpreted that τ is an instance of the type class TC with the method implementation $k = e$, if \overline{C}_i^i holds. The expression language e is a standard λ -calculus extended with multi-stage annotations, and includes literals i , top-level variables k , variables x , lambdas $\lambda x : \tau. e$, applications $e_1 e_2$, as well as quotations $\llbracket e \rrbracket$ and splicing $\$e$.

Following Jones [1994], the type language distinguishes between monotypes τ , qualified types ρ , and polymorphic types σ . Monotypes τ include type variables a , the integer type Int , function types $\tau_1 \rightarrow \tau_2$ and code representation $\text{Code } \tau$. Qualified types ρ qualify over monotypes with a list of constraints ($C \Rightarrow \rho$). Polymorphic types σ are qualified types with universal quantifiers ($\forall a. \sigma$). Finally, type class constraints are normal constraints $\text{TC } \tau$, or staged constraints $\text{CodeC } C$.

The program theory Θ is a context of type information for names introduced by top-level definitions $k : \sigma$, and the type class axioms introduced by instance declarations $\forall \overline{a}_i^i. \overline{C}_j^j \Rightarrow C$. The context Γ is used for locally introduced information, including value variables $x : (\tau, n)$, type variables a , and local type class axioms (C, n) . The context keeps track of the (integer) level n that value and constraint variables are introduced at; the typing rules will ensure that the variables are only used at the current level.

3.2 Typing Expressions

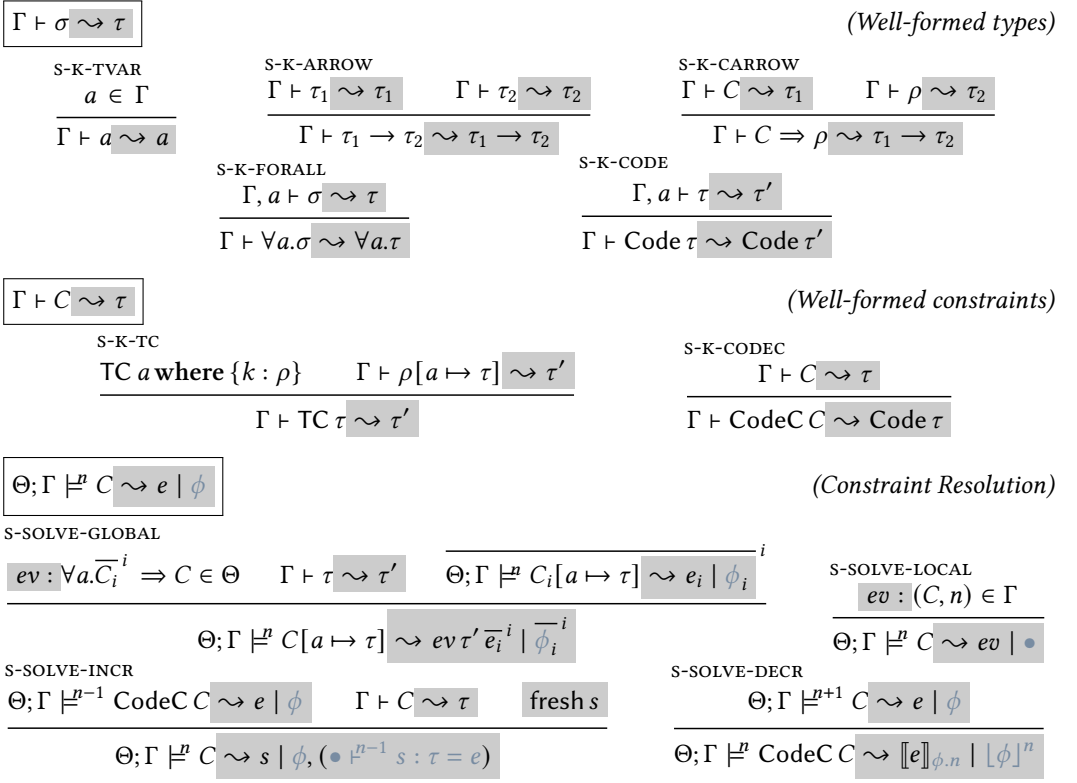
Figure 1 also presents the typing rules for expressions. The judgment $\Theta; \Gamma \vdash^n e : \sigma$ says that under the program theory Θ , the context Γ , and the current level n , the expression e has type σ . The gray parts are for elaboration (§5) and can be ignored until then.

Most typing rules are standard [Bottu et al. 2017; Chakravarty et al. 2005], except that rules are indexed by a level. As emphasized before, level-indexed typing rules ensure that variables and constraint can only be used at the level they are introduced. Literals and top-level variables can be used at any level (rules **S-LIT** and **S-KVAR**), as they can be persisted. Importantly, rule **S-VAR**

491	program	pgm	$::=$	$\text{def } \mathcal{D}; pgm \mid \text{class } C; pgm \mid \text{inst } \mathcal{I}; pgm \mid e$
492	definition	\mathcal{D}	$::=$	$k = e$
493	class	C	$::=$	$\text{TC } a \text{ where } \{k : \rho\}$
494	instance	\mathcal{I}	$::=$	$\overline{C}_i^i \Rightarrow \text{TC } \tau \text{ where } \{k = e\}$
495	expression	e	$::=$	$i \mid k \mid x \mid \lambda x : \tau. e \mid e_1 e_2 \mid \llbracket e \rrbracket \mid \e
496				
497	monotype	τ	$::=$	$a \mid \text{Int} \mid \tau_1 \rightarrow \tau_2 \mid \text{Code } \tau$
498	qualified type	ρ	$::=$	$C \Rightarrow \rho \mid \tau$
499	polymorphic type	σ	$::=$	$\forall a. \sigma \mid \rho$
500	constraint	C	$::=$	$\text{TC } \tau \mid \text{CodeC } C$
501				
502	program context	Θ	$::=$	$\bullet \mid \Theta, k : \sigma \mid \Theta, \forall \overline{a}_i^i. \overline{C}_j^j \Rightarrow C$
503	context	Γ	$::=$	$\bullet \mid \Gamma, x : (\tau, n) \mid \Gamma, a \mid \Gamma, (C, n)$
504				
505	$\Theta; \Gamma \vdash^n e : \sigma \rightsquigarrow e \mid \phi$			
506				
507				
508				
509				
510				
511				
512				
513				
514				
515				
516				
517				
518				
519				
520				
521				
522				
523				
524				
525				
526				
527				
528				
529				
530				
531				
532				
533				
534				
535				
536				
537				
538				
539				

Fig. 1. Syntax and typing rules of $\lambda[\Rightarrow]$

says that if a variable x is introduced at level n , then it is well-typed at level n . Rules **S-CABS** and **S-CAPP** handle generalization and instantiation of type class constraints. If an expression e can be type-checked under a local type class assumption C , then e has a qualified type $C \Rightarrow \rho$. Otherwise, if a constraint C can be resolved (§3.3), then an expression of type $C \Rightarrow \rho$ can be typed ρ .


 Fig. 2. Well-formed types, well-formed constraints and constraint resolution in $\lambda^{\llbracket \Rightarrow \rrbracket}$

Rules **S-QUOTE** and **S-SPLICE** type-check staging annotations. In particular, rule **S-QUOTE** increases the level by one and gives $\llbracket e \rrbracket$ type $\text{Code } \tau$ when e has type τ , while rule **S-SPLICE** decreases the level by one and gives e type τ when $\$e$ has type $\text{Code } \tau$.

Well-formed types and constraints. Typing rules (e.g., rule **S-ABS**) refer to well-formed rules for types and for constraints as given in Figure 2. The type well-formedness judgment $\Gamma \vdash \sigma$ simply checks that all type variables are well-scoped. The constraint well-formedness constraint $\Gamma \vdash C$ checks that the class method type is well-formed after substituting the variable a with τ .

3.3 Constraint Resolution

The typing rule (rule **S-CAPP**) also makes use of constraint resolution, whose rules are given at the bottom of Figure 2. The judgment $\Theta; \Gamma \models^{\sharp} C$ reads that under the program theory Θ , the context Γ , and the current level C , the type class constraint C can be resolved. The definition of constraint resolution in $\lambda^{\llbracket \Rightarrow \rrbracket}$ has two key novelties: (1) *level-indexing*, which allows us to guarantee well-stagedness of constraints; (2) resolution of staged type class constraints.

Rule **S-SOLVE-GLOBAL** resolves a type class constraint using an instance definition. If Θ contains the instance definition $\forall a. \overline{C_i}^i \Rightarrow C$, we can resolve $C[a \mapsto \tau]$ by resolving $\overline{C_i[a \mapsto \tau]}^i$. Rule **S-SOLVE-LOCAL** resolves a constraint using the local type class axiom.

$$\begin{array}{c}
\boxed{\Theta \vdash \text{pgm} : \sigma} \quad \text{(Typing programs)} \\
\begin{array}{c}
\text{S-PGM-DEF} \quad \frac{\Theta_1 \vdash \mathcal{D} \dashv \Theta_2 \quad \Theta_2 \vdash \text{pgm} : \sigma}{\Theta_1 \vdash \text{def } \mathcal{D}; \text{pgm} : \sigma} \quad \text{S-PGM-CLS} \quad \frac{\Theta_1 \vdash \mathcal{C} \dashv \Theta_2 \quad \Theta_2 \vdash \text{pgm} : \sigma}{\Theta_1 \vdash \text{class } \mathcal{C}; \text{pgm} : \sigma} \\
\text{S-PGM-INST} \quad \frac{\Theta_1 \vdash \mathcal{I} \dashv \Theta_2 \quad \Theta_2 \vdash \text{pgm} : \sigma}{\Theta_1 \vdash \text{inst } \mathcal{I}; \text{pgm} : \sigma} \quad \text{S-PGM-EXPR} \quad \frac{\Theta; \bullet \vdash^0 e : \sigma \quad \Theta \vdash e \mid \phi \quad \bullet \vdash \sigma \rightsquigarrow \tau \quad e : \tau \vdash^1 \phi \rightsquigarrow \text{pgm}}{\Theta \vdash e : \sigma \rightsquigarrow \text{pgm}}
\end{array} \\
\boxed{\Theta_1 \vdash \mathcal{D} \dashv \Theta_2} \quad \text{(Typing definitions)} \quad \boxed{\Theta_1 \vdash \mathcal{C} \dashv \Theta_2} \quad \text{(Typing class declarations)} \\
\begin{array}{c}
\text{S-DEF} \quad \frac{\Theta; \bullet \vdash^0 e : \sigma}{\Theta \vdash k = e \dashv \Theta, k : \sigma} \quad \text{S-CLS} \quad \frac{a \vdash \rho}{\Theta \vdash \text{TC } a \text{ where } \{k : \rho\} \dashv \Theta, k : \forall a. \text{TC } a \Rightarrow \rho}
\end{array} \\
\boxed{\Theta_1 \vdash \mathcal{I} \dashv \Theta_2} \quad \text{(Typing class instances)} \\
\text{S-INST} \quad \frac{\text{TC } a \text{ where } \{k : \rho\} \quad \overline{b_j^j} = \text{ftv}(\tau) \quad \overline{b_j^j} \vdash C_i^i \quad \Theta; \overline{b_j^j}, (\overline{C_i}, 0)^i \vdash^0 e : \rho[a \mapsto \tau]}{\Theta \vdash \overline{C_i}^i \Rightarrow \text{TC } \tau \text{ where } \{k = e\} \dashv \Theta, \forall \overline{b_j^j}. \overline{C_i}^i \Rightarrow \text{TC } \tau}
\end{array}$$

Fig. 3. Program typing in $\lambda^{\llbracket \Rightarrow \rrbracket}$

Rules **S-SOLVE-DECR** and **S-SOLVE-INCR** are specific to our system. In particular, rule **S-INCR** says that a staged type class constraint `CodeC` C at level $n - 1$ can be used to resolve C at level n , which is essentially what enables us to have constraint inside quotations. Similarly, rule **S-DECR** says that a normal type class constraint C at level $n + 1$ can be used to resolve `CodeC` C at level n . We can thus use these two rules to convert back and forth between `CodeC` C and C .

Example 3.1 ($\lambda^{\llbracket \Rightarrow \rrbracket}$ typing). Let us illustrate the typing rules and the constraint resolution rules by revisiting the example $\llbracket \text{show} \rrbracket$ (Example S1). Below we give its typing derivation. For this example we assume the primitive type `String`, and the program environment Θ to contain the type of `show`.

$\Theta = \text{show} : \forall a. \text{Show } a \Rightarrow a \rightarrow \text{String}$

$\Gamma = a, (\text{CodeC } (\text{Show } a), 0)$

$$\begin{array}{c}
\frac{\text{show} : \forall a. \text{Show } a \Rightarrow a \rightarrow \text{String} \in \Theta}{\Theta; \Gamma \vdash^1 \text{show} : \forall a. \text{Show } a \Rightarrow a \rightarrow \text{String}} \text{S-KVAR} \quad \frac{(\text{CodeC } (\text{Show } a), 0) \in \Gamma}{\Theta; \Gamma \vdash^0 \text{CodeC } (\text{Show } a)} \text{S-SOLVE-LOCAL} \\
\frac{\Theta; \Gamma \vdash^1 \text{show} : \forall a. \text{Show } a \Rightarrow a \rightarrow \text{String}}{\Theta; \Gamma \vdash^1 \text{show} : \text{Show } a \Rightarrow a \rightarrow \text{String}} \text{S-TAPP} \quad \frac{\Theta; \Gamma \vdash^0 \text{CodeC } (\text{Show } a)}{\Theta; \Gamma \vdash^1 \text{Show } a} \text{S-SOLVE-INCR} \\
\frac{\Theta; \Gamma \vdash^1 \text{show} : \text{Show } a \Rightarrow a \rightarrow \text{String} \quad \Theta; \Gamma \vdash^1 \text{show} : (a \rightarrow \text{String})}{\Theta; \Gamma \vdash^1 \text{show} : (a \rightarrow \text{String})} \text{S-CAPP} \\
\frac{\Theta; \Gamma \vdash^1 \text{show} : (a \rightarrow \text{String})}{\Theta; \Gamma \vdash^0 \llbracket \text{show} \rrbracket : \text{Code } (a \rightarrow \text{String})} \text{S-QUOTE} \\
\frac{\Theta; \Gamma \vdash^0 \llbracket \text{show} \rrbracket : \text{Code } (a \rightarrow \text{String})}{\Theta; a \vdash^0 \llbracket \text{show} \rrbracket : \text{CodeC } (\text{Show } a) \Rightarrow \text{Code } (a \rightarrow \text{String})} \text{S-CABS} \\
\frac{\Theta; a \vdash^0 \llbracket \text{show} \rrbracket : \text{CodeC } (\text{Show } a) \Rightarrow \text{Code } (a \rightarrow \text{String})}{\Theta; \bullet \vdash^0 \llbracket \text{show} \rrbracket : \forall a. \text{CodeC } (\text{Show } a) \Rightarrow \text{Code } (a \rightarrow \text{String})} \text{S-TABS}
\end{array}$$

Let us go through the derivation bottom-up. First, by applying rules **S-TABS** and **S-CABS**, we introduce the type variable a and the staged type class constraint `CodeC` $(\text{Show } a)$ at level 0 into the context. Then by rule **S-QUOTE**, our goal becomes $\Theta; \Gamma \vdash^1 \text{show} : (a \rightarrow \text{String})$ at level 1. At this point, rule **S-KVAR** allows us to use `show` from Θ at level 1, but we need to further apply rule **S-TAPP** and **S-CAPP**, and the latter requires us to prove `Show` a at level 1. To this end, rule **S-SOLVE-LOCAL** first gets `CodeC` `Show` a at level 0, and rule **S-SOLVE-INCR** then converts it into `Show` a at level 1.

3.4 Program Typing

As we have seen from the syntax, a program is a sequence of top-level definitions, class and instance declarations followed by an expression. Figure 3 presents the typing rules for programs. The judgment $\Theta \vdash \text{pgm} : \sigma$ reads that under the program theory Θ , the source program pgm has type σ . Most rules are standard. Top-level definitions (rule **S-PGM-DEF**) and declaration forms (rules **S-PGM-CLS** and **S-PGM-INST**) extend the program theory Θ which is used to type-check subsequent definitions. Rule **S-PGM-EXPR** makes it clear that the top-level of the program is level 0 and that the expression is checked in an empty local environment.

Rules **S-DEF**, **S-CLS**, and **S-INST** type-check top-level definitions, class and instance declarations, respectively. Rule **S-DEF** extends the list of top-level definitions available at all stages. Rule **S-CLS** extends the program theory with the qualified class method. Rule **S-INST** checks that the class method is of the type specified in the class definition.

4 F^\square : MULTI-STAGE CORE CALCULUS WITH SPLICE ENVIRONMENTS

We describe an explicitly typed core language F^\square , which extends System F with quotations, *splice environments* and *top-level splice definitions*. F^\square does not contain splices themselves as they are modeled using the splice environments, which are attached to quotations, and top-level splice definitions. As such, quotations can be considered opaque until spliced, and F^\square serves as a suitable compilation target for multi-staging programming.

4.1 Syntax

The syntax for F^\square is presented at the top of Figure 4. To reduce notational clutter, we reuse notation from $\lambda^{\square \Rightarrow}$ for expressions and types, making it clear from the context which calculus we refer to.

A program (ρgm) is a sequence of top-level definitions (\mathcal{D}) and top-level splice definitions (\mathcal{S}) followed by an expression (e). Top-level definitions $k : \tau = e$ are the same as for $\lambda^{\square \Rightarrow}$, except that, since F^\square is explicitly typed, k is associated with its type τ . There is no syntax for type classes or instances, which will be represented using top-level definitions after dictionary-passing elaborations. Top-level splice definitions $\Delta \text{ } \iota^{\text{J}} s : \tau = e$ are used to support compile-time evaluation, where the *splice variable* s captures the local type environment Δ , the level n , the type after splicing τ , and the expression to be spliced e . As we will see, the typing rules will ensure that that expression e has type Code τ at level n under type context Δ . The purpose of the environment Δ is to support open code representations which lose their lexical scoping when floated out from the quotation.

Expressions e include literals i , top-level variables k , splice variables s , variables x , lambdas $\lambda x : \tau. e$, applications $e_1 e_2$, type abstractions $\Lambda a. e$ and type applications $e \tau$, and quotations with splice environment $\llbracket e \rrbracket_\phi$, which are quotations with an associated splice environment. The splice environment ϕ is essentially a list of splice definitions $(\Delta \text{ } \iota^{\text{J}} s : \tau = e)$, which binds a splice variable s for each splice point within the quoted expression. A splice point is where the result of evaluating a splice will be inserted. One example we have seen from §2.7 is that the expression $\llbracket e_2 \$(e_3) \rrbracket$ can be represented in F^\square as $\llbracket e_2 s \rrbracket_{\bullet \text{ } \iota^{\text{J}} s : \tau = e_3}$ which, when spliced, will insert the result of splicing e_3 in the place of the splice variable s .

The program context Θ records the type of top-level definitions $k : \tau$ and top-level splice definitions $s : (\Delta, \tau, n)$. We distinguish between two type contexts Δ and Γ , where Γ is Δ extended with types for splice variables. The syntax distinction makes it clear that splice definitions (\mathcal{S}) and environments (ϕ) only capture Δ , which are type contexts elaborated from the source language and so contain no splice variables.

687	program	ρgm	$::=$	$\mathbf{def} \mathcal{D}; \rho gm \mid \mathbf{spdef} \mathcal{S}; \rho gm \mid e : \tau$
688	definition	\mathcal{D}	$::=$	$k : \tau = e$
689	splice definition	\mathcal{S}	$::=$	$\Delta \vdash^n s : \tau = e$
690	expression	e	$::=$	$i \mid k \mid s \mid x \mid \lambda x : \tau. e \mid e_1 e_2 \mid \Lambda a. e \mid e \tau \mid \llbracket e \rrbracket_\phi$
691	splice environment	ϕ	$::=$	$\bullet \mid \phi, \Delta \vdash^n s : \tau = e$
692				
693	type	τ	$::=$	$a \mid \text{Int} \mid \tau_1 \rightarrow \tau_2 \mid \forall a. \tau \mid \text{Code } \tau$
694	program context	Θ	$::=$	$\bullet \mid \Theta, k : \tau \mid \Theta, s : (\Delta, \tau, n)$
695	context	Δ	$::=$	$\bullet \mid \Delta, x : (\tau, n) \mid \Delta, a$
696		Γ	$::=$	$\bullet \mid \Gamma, x : (\tau, n) \mid \Gamma, a \mid \Gamma, s : (\Delta, \tau, n)$
697	$\Theta \vdash \rho gm$			
698				
699				
700				
701				
702				
703				
704				
705				
706				
707				
708				
709				
710				
711				
712				
713				
714				
715				
716				
717				
718				
719				
720				
721				
722				
723				
724				
725				
726				
727				
728				
729				
730				
731				
732				
733				
734				
735				

Fig. 4. Syntax and typing in F^\square

4.2 Typing Rules

Figure 4 presents the typing rules for F^\square . The judgment $\Theta \vdash \rho gm$ type-checks a core program. As before, top-level definitions (rule **C-PGM-DEF**) and top-level splice definitions (rule **C-PGM-SPDEF**)

extend the program theory Θ which is used to type-check subsequent definitions. Rule **C-PGM-EXPR** type-checks the expression.

Rule **C-SPDEF** checks top-level splice definitions $\Delta \vdash^n s : \tau = e$ by checking that e has type Code τ at level n under the current program context Θ and the context Δ . Notice that the program context Θ is extended with $s : (\Delta, \tau, n + 1)$. The level of s is $n + 1$ as it represents the *spliced* expression. In the example of $\llbracket e_2 s \rrbracket_{\bullet}^{\tau=e_3}$ which expresses $\llbracket e_2 \$(e_3) \rrbracket$, the splice variable s stands for $\$(e_3)$. The precondition $\Delta \succ n$ ensures that all variables in Δ have levels greater than n (§4.4.1). We use *dotted binary operators* (e.g., $\dot{>}$, $\dot{=}$ etc) to indicate level comparison.

The expression typing rules for the core expressions are for the most part the same as those in the source language. One observation is that since the language does not contain splicings, the level during typing can only increase (when typing quotations in rule **C-QUOTE**) but never decrease.

Rules **C-SVAR** and **C-TOP-SVAR** retrieve the type of splice variables from the context. Note that, as with expression variables, splice variables must be used at the level where they are introduced. Moreover, the local type context Δ captured by s must be a subset of the current type context Γ so that all free variables in e remain well-typed after substituting s with e . Γ may contain more variables, including splice variables that are not in Δ .

Rule **C-QUOTE**, which type-checks quotations with splice environments, is of particular interest. First, it checks that a splice environment is well-typed by the judgment $\Theta; \Gamma \vdash^n \phi$, which is based on the judgment $\Theta; \Gamma \vdash \phi$ but in addition requires ϕ to contain only splice variables of level n (§4.4.1). An empty splice environment is always well-typed (rule **C-S-EMPTY**). Otherwise the splice environment is well-typed if each of definition is well-typed (rule **C-S-CONS**), where the context Γ is extended with the local type context Δ to type-check e .

After type-checking ϕ , rule **C-QUOTE** converts the splice environment ϕ into a list of splice variables ϕ^Γ . The definition of ϕ^Γ is straightforward and is given in the same figure. Then, rule **C-QUOTE** adds all those splice variables ϕ^Γ into the context Γ , as they may be used inside e . One way to think about splice environments is that they attach splice variable *bindings* to the quotation whose body is e . And thus their concrete names do not matter and we can consider quotations equivalent up to alpha-renaming, e.g., $\llbracket s \rrbracket_{\Delta \vdash^m s : \tau = e_1} =_\alpha \llbracket s' \rrbracket_{\Delta \vdash^m s' : \tau = e_1}$. Finally, the rule type-checks e at level $n + 1$, and concludes with the type Code τ .

4.3 Dynamic Semantics

Figure 5 presents the definition of values and dynamic semantics in $F\llbracket \cdot \rrbracket$. Note that evaluation is not level-indexed, as splice environments make the evaluation order of the core calculus evident.

Values v include literals i , lambdas $\lambda x : \tau. e$, type abstractions $\Lambda a. e$, and quotations with splice environments $\llbracket e \rrbracket_{\phi_v}$. Notably, quotation values ($\llbracket e \rrbracket_{\phi_v}$) can quote arbitrary expressions (e), but require the splice environment to be a value (ϕ_v). A splice environment value ϕ_v simply requires all bindings to be values (i.e. $\Delta \vdash^m s : \tau = v$). As we will see from the dynamic semantics shortly, this avoids the need to look inside quotations, as the splice environment corresponds exactly to the splices inside quotations that need to be evaluated.

The program evaluation judgment ($\rho gm_1 \longrightarrow \rho gm_2$) evaluates declarations in turn from top to bottom. Top-level definitions are evaluated (rule **CE-PGM-DEF**) to values and substituted into the rest of the program (rule **CE-PGM-DBETA**). Similarly, rule **CE-PGM-SPDEF** evaluates a top-level splice definition to a value of the form $\llbracket e \rrbracket_{\phi}$. We must then insert splices back into the program, which is done in rule **CE-PGM-SPBETA** by substituting s with $[\phi_v]e$. The notation $[\phi_v]e$, defined at the top of the figure, further inserts splices in ϕ_v back into the expression e . To understand the process, let us first consider the case when ϕ_v is empty, giving us $[\bullet]e = e$, and suppose $n = -1$ then we have:

$$\text{spdef } \Delta \vdash^{-1} s : \tau = \llbracket e \rrbracket_{\bullet} ; \rho gm \longrightarrow \rho gm[s \mapsto e]$$

785	value	$v ::= i \mid \lambda x : \tau. e \mid \Lambda a. e \mid \llbracket e \rrbracket_{\phi_v}$
786	splice environment value	$\phi_v ::= \bullet \mid \phi_v, \Delta \vdash^s s : \tau = v$
787		
788	$[\phi_v]e$ inserts splices in ϕ_v back into e .	
789	$[\bullet]e$	$= e$
790	$[\phi_v, \Delta \vdash^s s : \tau = \llbracket e' \rrbracket_{\phi_v'}]e$	$= [\phi_v](e[s \mapsto [\phi_v']e'])$
791		
792	$\boxed{\rho gm_1 \longrightarrow \rho gm_2}$	(Program reduction)
793	CE-PGM-DEF	
794	$\mathcal{D} \longrightarrow \mathcal{D}'$	
795	$\text{def } \mathcal{D}; \rho gm \longrightarrow \text{def } \mathcal{D}'; \rho gm$	CE-PGM-DBETA
796	CE-PGM-SPDEF	$\text{def } k : \tau = v; \rho gm \longrightarrow \rho gm[k \mapsto v]$
797	$S \longrightarrow S'$	CE-PGM-EXPR
798	$\text{spdef } S; \rho gm \longrightarrow \text{spdef } S'; \rho gm$	$e \longrightarrow e'$
799		$e : \tau \longrightarrow e' : \tau$
800	$\boxed{\mathcal{D}_1 \longrightarrow \mathcal{D}_2}$	(Definition reduction)
801	CE-DEF	$\boxed{S_1 \longrightarrow S_2}$
802	$e \longrightarrow e'$	CE-SPDEF
803	$k : \tau = e \longrightarrow k : \tau = e'$	$\Delta \vdash^s s : \tau = e \longrightarrow \Delta \vdash^s s : \tau = e'$
804		
805	$\boxed{e_1 \longrightarrow e_2}$	CE-BETA
806		CE-TBETA
807	$(\lambda x : \tau. e_1) e_2 \longrightarrow e_1[x \mapsto e_2]$	(Reduction)
808	CE-APP	$(\Lambda a. e) \tau \longrightarrow e[a \mapsto \tau]$
809	$e_1 \longrightarrow e'_1$	CE-TAPP
810	$e_1 e_2 \longrightarrow e'_1 e_2$	CE-QUOTE
811	$\boxed{\phi_1 \longrightarrow \phi_2}$	$\phi \longrightarrow \phi'$
812	CE-S-HEAD	$\llbracket e \rrbracket_{\phi} \longrightarrow \llbracket e \rrbracket_{\phi'}$
813	$\phi \longrightarrow \phi'$	(Splice environment reduction)
814	$\phi, \Delta \vdash^s s : \tau = e \longrightarrow \phi', \Delta \vdash^s s : \tau = e$	CE-S-TAIL
815		$e \longrightarrow e'$
816		$\phi_v, \Delta \vdash^s s : \tau = e \longrightarrow \phi_v, \Delta \vdash^s s : \tau = e'$

Fig. 5. Values and dynamic semantics in $F\llbracket\cdot\rrbracket$

Essentially, $\Delta \vdash^s s : \tau = \llbracket e \rrbracket_{\bullet}$ corresponds to the expression $\llbracket e \rrbracket$ in the source level, whose splicing result is bound to s . The position of s inside ρgm indicates where the source program $\llbracket e \rrbracket$ was originally found, and by substituting s with e we successfully insert the splicing result back into that position. Rule **CE-PGM-SPBETA** deals with the more general case where ϕ_v can be non-empty, which corresponds to *nested* splices, i.e., the source expression e (as in $\llbracket e \rrbracket$) may itself contain more splices, and those splices (of the corresponding level, in this case -1) are reflected as the splice environment ϕ_v associated to $\llbracket e \rrbracket_{\phi_v}$. In this case, we need to first insert those splice definitions back into the expression, i.e., as $[\phi_v]e$, and then we conclude by substituting s with $[\phi_v]e$.

After we evaluate all definitions and splice definitions, we can then start evaluating the expression (rule **CE-PGM-EXPR**). Expression reductions ($e_1 \longrightarrow e_2$) are mostly standard. Rule **CE-BETA** uses call-by-name, though the exact choice of the evaluation strategy does not matter. Of particular interest is rule **CE-QUOTE**, which says that to evaluate $\llbracket e \rrbracket_{\phi}$, we leave e as is, and all we need to do is to evaluate ϕ , which simply evaluates all expressions it binds (rules **CE-S-HEAD** and **CE-S-TAIL**).

Note that there is no reduction rule which reduces inside a quotation. Now the benefits of splice environments become clear: we can treat a quoted expression (the e part in $\llbracket e \rrbracket_\phi$) opaquely, giving the implementation freedom about its concrete form.

4.4 Well-stagedness and Type Soundness

In this section, we discuss the metatheory of F^\square . Before we present the type soundness result, we first discuss well-stagedness of splice environments.

4.4.1 Well-staged Splice Definitions and Environments. Our typing rules are designed carefully to allow only well-staged programs. As splice definitions and environments are novel in this calculus, great care needs to be taken to guarantee their well-stagedness. To this end, the typing rules have imposed the following restrictions on levels of splice definitions and environments:

- (1) A splice definition $\Delta \vdash^s s : \tau = e$ requires $\Delta \succ n$ as in rule **C-SPDEF** (similarly, rule **C-S-CONS**). That is, all splice variables in the local type context captured by a splice variable must have a level greater than that of the expression captured by the splice variable.
- (2) A well-staged quotation $\Theta; \Delta \vdash^s \llbracket e \rrbracket_\phi$ requires $\Theta; \Gamma \vdash^s \phi$, as in rule **C-QUOTE**, which implies $\phi \doteq n$. That is, all splice variables that bind level n are introduced at level n .⁴

Example 4.1 (Counterexamples to well-staged splices). The following examples are rejected.

- (a) $\bullet; \bullet \vdash^0 \llbracket e \rrbracket_{x:(\text{Code Int}, 0) \vdash^0 s : \text{Int} = x} : \text{Code } \tau$ breaks (1) as $x : (\text{Code Int}, 0) \not\succ 0$
- (b) $\bullet; \bullet \vdash^0 \llbracket \llbracket e \rrbracket_{\bullet \vdash^0 s : \text{Int} = (\lambda y : \text{Code Int}. y) (\llbracket 2 \rrbracket_\bullet)} \rrbracket_\bullet : \text{Code } (\text{Code } \tau)$ breaks (2) as $\bullet \vdash^0 s : \text{Int} \neq 1$

Essentially, the first restriction applies the *level restriction* of variables described in §2.1 to splice definition and environments; and the second lifts the level restriction to splice variables. In particular, consider the counterexample (a). What happens is that in the splice environment x is used at level 0, but inside e we can never introduce x at level 0 (recall that during typing the level monotonically increases)! So such an example is rejected because x is not well-staged.⁵

The level restriction to splice variables requires that a splice variable that binds level n is introduced at level n . The splice variable level restriction ensures that splice variables are evaluated at the right stage. Consider counterexample (b). If we evaluate the program at level 0, then because the splice environment is a value and we do not inspect inside the quotations, we will conclude that it is a value. But note that s is bound at level 0, which means the expression $(\lambda y : \text{Code Int}. y) (\llbracket 2 \rrbracket_\bullet)$ is at level 0 and so should get reduced when the expression is evaluated at level 0! We thus reject this example as s is not well-staged.

4.4.2 Type Soundness. With well-staged splice definitions and environments, we can now prove that F^\square enjoys type soundness, by proving type preservation and progress.

First, we show that any reduction preserves the type information. For space reasons, we only present the theorem for expressions and programs, but the theorem holds for all other forms.

Theorem 4.2 (Progress). (1) If $\bullet; \bullet \vdash^s e : \tau$, then either e is a value, or $e \longrightarrow e'$ for some e' .
 (2) If $\bullet \vdash^s \rho \text{pgm}$, then either ρpgm is $v : \tau$, or $\rho \text{pgm} \longrightarrow \rho \text{pgm}'$ for some $\rho \text{pgm}'$.

⁴An alternative is to represent a splice environment entry as $\Delta \vdash s : \tau = e$ (i.e. without levels), and then rule **C-QUOTE**, just like rule **C-ABS**, could directly take the current level from the typing judgment (which also means ϕ^Γ would need to take a level as input). However, that representation does not work for global splice variables (i.e. in rule **C-SPDEF** where typing is not level-indexed). Moreover, the representation of ϕ is also used during elaboration, where it is important to track the levels. Therefore, we prefer to have a consistent representation and preserve the level information in the core.

⁵It may seem like we can introduce x outside of the quotation, making x well-staged. However, if x is introduced outside of the quotation (and thus the splice environment), then it should *not* be captured by the splice variable, as it is in the scope of the splice environment (i.e. is not *local*). For example, the well-typed source program $\lambda x : \text{Code Int}. \llbracket \$x \rrbracket$ elaborates to $\lambda x : \text{Code Int}. \llbracket s \rrbracket_{\bullet \vdash^0 s : \text{Int} = x}$, while the source program $\llbracket \lambda x : \text{Int}. \$\llbracket x \rrbracket \rrbracket$ elaborates to $\llbracket \lambda x : \text{Code Int}. s \rrbracket_{x:(\text{Int}, 1) \vdash^0 s : \text{Int} = \llbracket x \rrbracket_\bullet}$.

$$\begin{array}{c}
\boxed{\phi_1 \dashv\vdash \Delta \rightsquigarrow \phi_2} \\
\text{S-INJ-EMPTY} \\
\bullet \dashv\vdash \Delta \rightsquigarrow \bullet \\
\hline
\boxed{\rho gm_1 \Vdash^n \phi \rightsquigarrow \rho gm_2} \\
\text{S-CLAP-EMPTY} \\
\rho gm \Vdash^n \bullet \rightsquigarrow \rho gm \\
\hline
\end{array}
\quad
\begin{array}{c}
\text{S-INJ-CONS} \\
\frac{\phi_1 \dashv\vdash \Delta_2 \rightsquigarrow \phi_2}{\phi_1, \Delta_1 \Vdash^n s : \tau = e \dashv\vdash \Delta_2 \rightsquigarrow \phi_2, (\Delta_2, \Delta_1 \Vdash^n s : \tau = e)} \\
\hline
\text{S-CLAP-REC} \\
\frac{\text{spdef } \phi.n; \rho gm_1 \Vdash^{n-1} [\phi]^n \rightsquigarrow \rho gm_2}{\rho gm_1 \Vdash^n \phi \rightsquigarrow \rho gm_2}
\end{array}
\begin{array}{c}
\text{(Injection)} \\
\\
\text{(Collapse)}
\end{array}$$

Fig. 6. Auxiliary definitions used in elaboration: injection used in Figure 1, and collapse used in Figure 3

Now we show that well-typed programs cannot go wrong, by proving that a well-typed expression (and definition / program respectively) is either a value, or can take a step.

Theorem 4.3 (Type Preservation). (1) If $\Theta; \Delta \Vdash e : \tau$, and $e \longrightarrow e'$, then $\Theta; \Delta \Vdash e' : \tau$.
(2) If $\Theta \vdash \rho gm$, and $\rho gm \longrightarrow \rho gm'$, then $\Theta \vdash \rho gm'$.

5 ELABORATION FROM $\lambda[\![\Rightarrow]\!]$ TO $F[\![\!]\!]$

In this section we describe the process of type-directed elaboration from the source language $\lambda[\![\Rightarrow]\!]$ into the core language $F[\![\!]\!]$. There are three key aspects of the elaboration procedure :

- (1) Splices are removed in favour of a splice environment. The elaboration process returns a splice environment which is attached to the quotation form (§5.1).
- (2) Type class constraints are converted to explicit dictionary passing. We describe how to understand staged type class constraints *CodeC* *C* in terms of quotation (§5.2).
- (3) Splices at non-positive levels that are not attached to a corresponding quotation are elaborated to top-level splice definitions, which are put before the rest of the program (§5.3).

5.1 Elaborating Expressions with Splice Environments

The elaboration of expressions appears in gray with the source typing rules in Figure 1. The judgment $\Theta; \Gamma \Vdash e : \sigma \rightsquigarrow e \mid \phi$ states that, under the program context Θ and the context Γ , the source expression e at level n with type σ is elaborated into a core expression e whilst producing the splice environment ϕ . As we will see, since splices at level n create splice variables at level $n - 1$, and quotations at level n capture all inner splice variables at level n , we maintain the invariant on the judgment that $\phi \prec n$ (§5.4.1).

At a high level, all splice variables are initially added to the splice environment when elaborating splices (rule **S-SPLICE**), and then propagated through the rules, until captured by quotations (rule **S-QUOTE**); uncaptured splice variables are discussed in §5.3. Let us first take a look at rule **S-SPLICE**. To elaborate a source splice $\$e$, rule **S-SPLICE** generates a fresh splice variable s which is returned as the elaboration result. It then extends the splice environment ϕ with s that binds an empty local context (as every variable is still in the scope of the splice at this moment), the level of the expression $n - 1$, the core type τ' , and the core expression e . This way we effectively insert s as a splice point, with the expression to be spliced bound to s in the splice environment. Splice environments are captured by quotations in rule **S-QUOTE**. In particular, a quotation at level n captures only the splices at level n ; the notation $\phi.n$ denotes the projection of the splices contained in ϕ at level n . We then truncate ϕ by removing $\phi.n$ from it using the notation $[\phi]^n$.

Importantly, we need to ensure well-scopedness of splice environments during this process. When a splice variable gets out of a scope, e.g. in rule **S-ABS**, we cannot directly return ϕ_1 , as ϕ_1 may refer to x and directly returning ϕ_1 would cause it to be ill-typed! To this end, whenever a splice variable gets out of a scope, it captures the scope in its local context. In other words, a splice variable captures the local context from its introduction point up to the point where it is bound by a quotation. This is done by the injection judgment $\phi_1 \dashv\vdash \Delta \rightsquigarrow \phi_2$, defined at the top of Figure 6, and is used in for example rule **S-ABS**. Specifically, the judgment $\phi_1 \dashv\vdash \Delta \rightsquigarrow \phi_2$ inserts Δ into the local context of each splice variable in ϕ_1 , producing a new splice environment ϕ_2 . As we will prove, the injection process is crucial to establish elaboration soundness.

The remaining rules elaborate source expressions in an expected way, while propagating splice environments, e.g. rule **S-APP** elaborates a source application into a core application, and collects splice environments from preconditions. We talk more about elaborating type classes (rules **S-CABS** and **S-CAPP**) in the next section.

5.2 Dictionary-passing Elaboration of Constraints

Figure 2 presents the elaboration of types and constraints. Well-formed source types elaborate to well-formed core types ($\Gamma \vdash \sigma \rightsquigarrow \tau$).

Type classes are translated away by dictionary-passing elaboration [Jones 1994]. In particular, well-formed constraints elaborate to well-formed core types ($\Gamma \vdash C \rightsquigarrow \tau$). Note that a class constraint $TC \tau$ elaborates to its method type, as an instance of the constraint provides an implementation of the method.⁶ Accordingly, rule **S-CABS** elaborates an expression with a constraint into a dictionary-taking function, and rule **S-CAPP** elaborates class resolution as function applications.

The last judgment $\Theta; \Gamma \models^{\mathcal{L}} C \rightsquigarrow e \mid \phi$ is of particular interest: resolving a type class constraint C returns an expression e as evidence for the constraint, with a splice environment ϕ . Rules **S-SOLVE-GLOBAL** and **S-SOLVE-LOCAL** are standard elaboration rules of normal type class resolution, where the former uses an instance declaration in the program context, and the latter uses a local instance (as introduced in rule **S-CABS**).

Rules **S-SOLVE-INC** and **S-SOLVE-DEC** concern staged type class constraints. Rule **S-SOLVE-DEC** elaborates staged type class constraints into values of type $\text{Code } \tau$. Therefore resolution elaboration of staged type class constraints must be understood in terms of quotations. Rule **S-SOLVE-DEC** is implemented by a simple quotation and thus similar to typing quotations (i.e., rule **S-QUOTE**). Rule **S-SOLVE-INC** conceptually introduces a splice; as in rule **S-SPICE**, it achieves this by extending the splice environment, since the core language does not have splices. These rules explain the necessity of level-indexing constraints in the source language: the elaboration would not be well-staged if the stage discipline was not enforced.

5.3 Elaborating Programs with Top-level Splice Definitions

We elaborate programs as shown in gray in Figure 3. For space reasons, we only present the elaboration for programs of the form $e : \tau$ (rule **S-PGM-EXPR**); elaborations of other forms apply the same idea to the standard elaboration of type class and instance declarations [Bottu et al. 2017; Jones 1994]. The full rules can be found in the appendix.

If a splice occurs at a non-positive level without corresponding surrounding quotations, then it should be evaluated at compile time, and in our formalism, it becomes a top-level splice definition.⁷

⁶This is a simplification of elaboration for multi-method type classes, which produces a *record* with a field for each method.

⁷In general, non-positive splices can still have surrounding quotations. There are two cases. (1) The quotation is not at the corresponding level, then the splice is lifted to top-level splice definition. For example, $\llbracket \$(\$e) \rrbracket$ elaborates to $\text{spdef } \bullet \vdash^{-1} s_2 : \text{Code Int} = e; \llbracket s_1 \rrbracket_{\bullet} \phi_{s_1 : \text{Int} = s_2} : \text{Code Int}$, where s_2 has a surrounding quotation but becomes a *spdef*. (2) The quotation is at the corresponding level, then the splice will be attached to a quotation even if it is non-positive. For example,

This process can be seen from rule **S-PGM-EXPR**, where we start by elaborating the source expression e at the default level 0, which returns the core expression e and the splice environment ϕ . As we have mentioned in §5.1, elaborating expression at level n maintains the invariant $\phi \dot{<} n$ (§5.4.1). Since in this case the expression is elaborated at level 0, we have $\phi \dot{<} 0$; namely, the result ϕ returned from elaborating the expression contains non-positive splice variables that should be evaluated at compile time. Hence, we turn those splice environments into top-level splice definitions and put them before $e : \tau$, using the collapse judgment $\rho gm_1 \Vdash \phi \rightsquigarrow \rho gm_2$, given in Figure 6. The collapse process takes the current program ρgm_1 , and creates top-level splice declarations for each splice in ϕ , generating ρgm_2 . To guarantee a stage-correct execution, the splices are inserted in order of their levels, decreasing from n ; for rule **S-PGM-EXPR**, we have $n = -1$. Now ρgm returned from rule **S-PGM-EXPR** contains exactly what we want: a sequence of top-level splice definitions, followed by the elaborated core expression.

Example 5.1 (Elaboration). The derivation below shows the elaboration of a source program $\$(k)$, where k is a top-level definition defined as $\llbracket \text{show} \rrbracket$ whose typing derivation has been given in Example 3.1. This illustrates two particular points of interest: CodeC (Show a) is elaborated into quoted evidence using rule **C-SOLVE-DECR**, and the injection ensures the splices are well-typed.

$$\begin{array}{c}
\Theta = k : \forall a. \text{CodeC}(\text{Show } a) \Rightarrow \text{Code}(a \rightarrow \text{String}) \quad \phi_1 = \bullet \vdash^1 s : a \rightarrow \text{String} = k a \llbracket ev \rrbracket. \\
\Gamma = a, \text{ev} : (\text{Show } a, 0) \quad \phi_2 = ev : (a \rightarrow \text{String}, 0) \vdash^1 s : a \rightarrow \text{String} = k a \llbracket ev \rrbracket. \\
\phi_3 = a, ev : (a \rightarrow \text{String}, 0) \vdash^1 s : a \rightarrow \text{String} = k a \llbracket ev \rrbracket. \\
\\
\frac{\begin{array}{c} k : \forall a. \text{CodeC}(\text{Show } a) \Rightarrow \text{Code}(a \rightarrow \text{String}) \in \Theta \\ \Theta; \Gamma \vdash^1 k : \forall a. \text{CodeC}(\text{Show } a) \Rightarrow \text{Code}(a \rightarrow \text{String}) \rightsquigarrow k \mid \bullet \end{array} \quad \text{S-KVAR} \quad \frac{\text{ev} : (\text{Show } a, 0) \in \Gamma}{\Theta; \Gamma \vdash^0 \text{Show } a \rightsquigarrow ev \mid \bullet} \text{S-SOLVE-LOCAL}}{\Theta; \Gamma \vdash^1 k : \text{CodeC}(\text{Show } a) \Rightarrow \text{Code}(a \rightarrow \text{String}) \rightsquigarrow k a \mid \bullet} \text{S-TAPP} \quad \frac{\Theta; \Gamma \vdash^1 \text{CodeC}(\text{Show } a) \rightsquigarrow \llbracket ev \rrbracket. \mid \bullet}{\Theta; \Gamma \vdash^1 \text{CodeC}(\text{Show } a) \rightsquigarrow \llbracket ev \rrbracket. \mid \bullet} \text{S-SOLVE-DECR} \\
\\
\frac{\Theta; \Gamma \vdash^1 k : \text{CodeC}(\text{Show } a) \Rightarrow \text{Code}(a \rightarrow \text{String}) \rightsquigarrow k a \mid \bullet \quad \Theta; \Gamma \vdash^1 k : \text{Code}(a \rightarrow \text{String}) \rightsquigarrow k a \llbracket ev \rrbracket. \mid \bullet}{\Theta; \Gamma \vdash^1 k : \text{Code}(a \rightarrow \text{String}) \rightsquigarrow k a \llbracket ev \rrbracket. \mid \bullet} \text{S-CAPP} \\
\\
\frac{\Theta; \Gamma \vdash^0 \$(k) : a \rightarrow \text{String} \rightsquigarrow s \mid \phi_1 \quad \phi_1 \rightsquigarrow ev : (a \rightarrow \text{String}, 0) \rightsquigarrow \phi_2}{\Theta; \Gamma \vdash^0 \$(k) : a \rightarrow \text{String} \rightsquigarrow s \mid \phi_1} \text{S-SPICE} \quad \frac{\phi_1 \rightsquigarrow ev : (a \rightarrow \text{String}, 0) \rightsquigarrow \phi_2}{\phi_1 \rightsquigarrow ev : (a \rightarrow \text{String}, 0) \rightsquigarrow \phi_2} \text{S-INJ-CONS} \\
\\
\frac{\Theta; a \vdash^0 \$(k) : \text{Show } a \Rightarrow a \rightarrow \text{String} \rightsquigarrow \lambda ev. a \rightarrow \text{String}. s \mid \phi_2 \quad \phi_2 \rightsquigarrow a \rightsquigarrow \phi_3}{\Theta; a \vdash^0 \$(k) : \text{Show } a \Rightarrow a \rightarrow \text{String} \rightsquigarrow \lambda ev. a \rightarrow \text{String}. s \mid \phi_2} \text{S-CABS} \quad \frac{\phi_2 \rightsquigarrow a \rightsquigarrow \phi_3}{\phi_2 \rightsquigarrow a \rightsquigarrow \phi_3} \text{S-INJ-CONS} \\
\\
\frac{\Theta; \bullet \vdash^0 \$(k) : \forall a. \text{Show } a \Rightarrow a \rightarrow \text{String} \rightsquigarrow \Lambda a. \lambda ev. a \rightarrow \text{String}. s \mid \phi_3}{\Theta; \bullet \vdash^0 \$(k) : \forall a. \text{Show } a \Rightarrow a \rightarrow \text{String} \rightsquigarrow \Lambda a. \lambda ev. a \rightarrow \text{String}. s \mid \phi_3} \text{S-TABS}
\end{array}$$

Having obtained the main expression, we can apply rule **S-PGM-EXPR** and use collapse to turn ϕ_3 into a top-level splice definition and form the resulting program:

$$(\Lambda a. \lambda ev. a \rightarrow \text{String}. s) : \forall a. (a \rightarrow \text{String}) \rightarrow a \rightarrow \text{String} \vdash^1 \phi_3 \rightsquigarrow \text{spdef } a, ev : (a \rightarrow \text{String}, 0) \vdash^1 s : a \rightarrow \text{String} = k a \llbracket ev \rrbracket.; \\
(\Lambda a. \lambda ev. a \rightarrow \text{String}. s) : \forall a. (a \rightarrow \text{String}) \rightarrow a \rightarrow \text{String}$$

5.4 Elaboration Soundness

In this section, we prove that elaboration preserves types, which, together with type soundness of F^\square , establishes type soundness of $\lambda \llbracket \cdot \rrbracket$. To this end, we first need to show how the well-stagedness restrictions in F^\square (§4.4.1) are satisfied during elaboration.

5.4.1 Well-staged Splice Environments. The first restriction says that every $\Delta \Vdash s : \tau = e$ has $\Delta \dot{>} n$ (rules **C-SPDEF** and **C-S-CONS**). During elaboration, we have seen that a splice variable captures the local context from its introduction point up to the point where it is bound by a quotation. The restriction holds trivially when a splice variable is created with an empty local context, but since the local context can later be extended by injection we must prove that injection respects the

$\llbracket \text{se} \rrbracket$ elaborates to $\text{spdef } \bullet \vdash^1 s_4 : \text{Int} = \llbracket s_3 \rrbracket. \bullet \vdash^1 s_3 : \text{Int} = e; s_4 : \text{Int}$, where s_3 appears at non-positive level but is attached to a quotation. Note that the evaluation order is still correct: since s_4 is evaluated at level -1 , its splice environment is evaluated at -1 , and thus s_3 is evaluated at -1 .

restriction. This can be shown by first proving the invariant that the splice environment produced from typing has level smaller than the current typing level:

Lemma 5.2 (Level Correctness of ϕ). *If $\Theta; \Gamma \vdash^{\tau} e : \tau \rightsquigarrow e \mid \phi$, then $\phi \dot{<} n$.*

This can be easily seen from rule **S-SPLICE** that produces only splice variables with smaller levels; and rule **S-QUOTE** captures all splices at the current level.

We then use Lemma 5.2 to show that injection produces well-staged splice environments. Consider rule **S-ABS** as an example. By Lemma 5.2 we have $\phi_1 \dot{<} n$, and therefore $\phi_1 \dot{<} x : (\tau, n)$, so injection as in $\phi_1 \dashv\vdash x : (\tau, n) \rightsquigarrow \phi_2$ preserves the restriction. Formally, we can prove

Lemma 5.3 (Context Injection). *If $\Theta; \Delta_1, \Delta_2 \vdash \phi_1$, and $\phi_1 \dot{<} \Delta_2$, and $\phi_1 \dashv\vdash \Delta_2 \rightsquigarrow \phi_2$, then $\Theta; \Delta_1 \vdash \phi_2$.*

The second restriction requires that an elaborated quotation $\Theta; \Delta \vdash^{\tau} \llbracket e \rrbracket_{\phi}$ has $\Theta; \Gamma \vdash^{\tau} \phi$. We generate quotations at rule **S-QUOTE**. As the rule binds $\phi.n$ which by construction has level n , we only need to show $\Theta; \Delta \vdash \phi$, which can be proved making use of Lemma 5.3. In the following lemma statement, the notations $\Theta \rightsquigarrow \Theta$ and $\Gamma \rightsquigarrow \Delta$ elaborate contexts in an unsurprising way; their definitions can be found in the appendix.

Lemma 5.4 (Well-staged ϕ). *If $\Theta; \Gamma \vdash^{\tau} e : \tau \rightsquigarrow e \mid \phi$, and $\Theta \rightsquigarrow \Theta$, and $\Gamma \rightsquigarrow \Delta$, then $\Theta; \Delta \vdash \phi$.*

5.4.2 Elaboration Soundness. Now that we have established the key well-stagedness properties of splice environments, we are ready to prove that $\lambda^{\llbracket \Rightarrow \rrbracket}$ is type-safe by proving elaboration soundness, which formally establishes our goal: well-typed, well-staged source programs always elaborate to well-typed, well-staged core programs.

Theorem 5.5 (Elaboration Soundness).

- (1) *If $\Theta; \Gamma \vdash^{\tau} e : \tau \rightsquigarrow e \mid \phi$, and $\Theta \rightsquigarrow \Theta$, and $\Gamma \rightsquigarrow \Delta$, and $\Gamma \vdash \tau \rightsquigarrow \tau$, then $\Theta; \Delta, \phi^{\Gamma} \vdash^{\tau} e : \tau$.*
- (2) *If $\Theta \vdash \text{pgm} : \sigma \rightsquigarrow \rho \text{gm}$, and $\Theta \rightsquigarrow \Theta$, then $\Theta \vdash \rho \text{gm}$.*

6 AXIOMATIC SEMANTICS

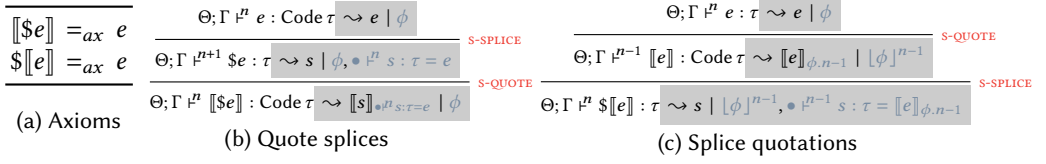
Our goal in designing $\lambda^{\llbracket \Rightarrow \rrbracket}$ and $F^{\llbracket \Rightarrow \rrbracket}$ is to provide a theoretical foundation for multi-stage programming. It is thus important to show that our formalism enjoys desirable properties. One such property is that splices and quotations are dual to each other, which provides a simple reasoning principle for multi-stage programming, and allows programmers to *cancel splices and quotations out* without worrying about changing the semantics of programs.

In this section, we prove this crucial property by establishing axioms and axiomatic semantics of $\lambda^{\llbracket \Rightarrow \rrbracket}$ and $F^{\llbracket \Rightarrow \rrbracket}$ respectively, and show that canceling out splices and quotations leads to *contextually equivalent* programs. The definitions of axiomatic semantics and the proofs in this section follow Taha et al. [1998] and Taha [1999], with key novelties in that (1) $\lambda^{\llbracket \Rightarrow \rrbracket}$ has elaboration-based semantics, and thus the correctness of its axioms are built on that of $F^{\llbracket \Rightarrow \rrbracket}$, and this indirection poses extra complexities in the proofs; and (2) for $F^{\llbracket \Rightarrow \rrbracket}$, we define the axiomatic semantics and extend the proofs for our novel splice environments and top-level splice definitions.

6.1 Duality of Splices and Quotations in $\lambda^{\llbracket \Rightarrow \rrbracket}$

The property we seek to establish is captured by the two axioms of $\lambda^{\llbracket \Rightarrow \rrbracket}$ given in Figure 7a, which state that splicing a quotation or quoting a splice is equivalent to the original expression: they respectively represent eta and beta laws for *Code*. These axioms form part of the equational theory of $\lambda^{\llbracket \Rightarrow \rrbracket}$; they can be thought of as context-independent pattern-based rewriting rules.

Consider an axiomatic equivalence relation between $\lambda^{\llbracket \Rightarrow \rrbracket}$ programs that is the contextual and equivalence closure of the axioms, which we denote as $\text{pgm}_1 =_{ax} \text{pgm}_2$. Our goal now is to prove

Fig. 7. Axioms and elaboration derivations in $\lambda[\Rightarrow]$

axiomatically equivalent source programs are *contextually equivalent*, i.e. they always produce the same result and thus can be used in an interchangeable way. As the dynamic semantics of $\lambda[\Rightarrow]$ is defined based on elaboration to $F[]$, we build the proofs based on the axiomatic semantics of $F[]$.

6.2 Axiomatic Semantics of $F[]$

The axiomatic semantics of $F[]$ is guided by the elaboration of the $\lambda[\Rightarrow]$ axioms. Supposing source e elaborates to core e with ϕ , Figures 7b and 7c present elaboration derivations of $\llbracket \$e \rrbracket$ and $\llbracket \$e \rrbracket$ respectively. Looking first at Figure 7b, what is needed to show the first $\lambda[\Rightarrow]$ axiom is a $F[]$ axiom that models the equivalence between expression $\llbracket s \rrbracket_{\bullet \vdash^n s : \tau = e}$ with ϕ (the elaboration result of $\llbracket \$e \rrbracket$) and e with ϕ (the elaboration result of e). Since the two ϕ s are the same, it is sufficient to introduce a core axiom $\llbracket s \rrbracket_{\bullet \vdash^n s : \tau = e} =_{ax} e$.

The case for splicing quotations (Figure 7c) is more challenging: in this case we cannot directly compare the elaborated expressions, as the generated splice environments are different. Instead, we need to consider equivalence between two core quotations where the splice environments are bound. To derive the axiom, let us first consider the case where both expressions are bound immediately to a quotation. That leads to $\llbracket s \rrbracket_{\llbracket \phi \rrbracket^{n-1}, \bullet \vdash^{n-1} s : \tau = \llbracket e \rrbracket_{\phi, n-1}} =_{ax} \llbracket e \rrbracket_{\llbracket \phi \rrbracket^{n-1}, \phi, n-1}$. Abstracting over the specific shape of splice environments gives us $\llbracket s \rrbracket_{\phi_1, \bullet \vdash^n s : \tau = \llbracket e \rrbracket_{\phi}} =_{ax} \llbracket e \rrbracket_{\phi_1, \phi}$. In the case when s is not immediately bound, we then have $\llbracket e_1 \rrbracket_{\phi_1, \bullet \vdash^n s : \tau = \llbracket e \rrbracket_{\phi}} =_{ax} \llbracket e_1[s \mapsto e] \rrbracket_{\phi_1, \phi}$. However, there are still some wrinkles to this axiom. First, s could have a non-empty splice environment ϕ_2 to its right, as until s is bound there can be more splices. Second, s could have a non-empty local context Δ , as until s is bound it may have got out of some scopes and so have applied the injection process. Finally, if s has a non-empty local context, then after it is substituted away on the right hand side, we cannot directly discard its local context Δ and leave ϕ , since ϕ now becomes ill-typed as it loses the scope of the variables from Δ . Therefore, we need to inject Δ into ϕ .

Summarizing our discussion, we end up with the axiomatic semantic of $F[]$ as defined below. Note that splicing quotations also leads to the equivalence axiom between **spdef**.

Definition 6.1 (Axiomatic Semantics of $F[]$). *Axiomatic semantics of $F[]$ models β -equivalence, as well as the following axioms.*

$\llbracket s \rrbracket_{\bullet \vdash^n s : \tau = e}$	$=_{ax}$	e	
$\llbracket e_1 \rrbracket_{\phi_1, \Delta \vdash^n s : \tau = \llbracket e \rrbracket_{\phi, \phi_2}}$	$=_{ax}$	$\llbracket e_1[s \mapsto e] \rrbracket_{\phi_1, \phi', \phi_2}$	where $\phi \mapsto \Delta \leadsto \phi'$
spdef $\Delta \vdash^n s : \tau = \llbracket e \rrbracket_{\phi}; \rho gm$	$=_{ax}$	spdef $\phi'; \rho gm[s \mapsto e]$	where $\phi \mapsto \Delta \leadsto \phi'$

Now consider an axiomatic equivalence relation between $F[]$ programs that is the contextual and equivalence closure of the axioms, denoted as:

$$\Theta \vdash \rho gm_1 \simeq_{ax} \rho gm_2 \triangleq \Theta \vdash \rho gm_1 \wedge \Theta \vdash \rho gm_2 \wedge \rho gm_1 =_{ax} \rho gm_2$$

To show that our definition of axiomatic semantics of F^\square indeed captures the desirable duality between splices and quotations, we prove that axiomatically equivalent source programs elaborate to axiomatically equivalent core programs.

Lemma 6.2 ($\lambda^{\square} \Rightarrow_{ax}$ to $F^\square \simeq_{ax}$). *If $pgm_1 =_{ax} pgm_2$, where $\Theta \vdash pgm_1 : \sigma \rightsquigarrow \rho gm_1$, and $\Theta \vdash pgm_2 : \sigma \rightsquigarrow \rho gm_2$, and $\Theta \rightsquigarrow \Theta$, then $\Theta \vdash \rho gm_1 \simeq_{ax} \rho gm_2$.*

With this lemma, now we can use core axiomatic equivalence as an intermediate step to show that source axiomatic equivalence derives core contextual equivalence.

6.3 Contextual Equivalence

We define contextual equivalence in F^\square as below.

Definition 6.3 (Contextual Equivalence in F^\square).

$$\begin{aligned} \bullet; \Gamma \Vdash e_1 \simeq_{ctx} e_2 : \tau &\triangleq \bullet; \Gamma \Vdash e_1 : \tau \wedge \bullet; \Gamma \Vdash e_2 : \tau \\ &\wedge (\forall \mathbb{C} : \bullet; \Gamma \Vdash \tau \rightsquigarrow \bullet; \bullet \Vdash^0 \text{Int}, \mathbb{C}[e_1] \longrightarrow^* i \iff \mathbb{C}[e_2] \longrightarrow^* i) \\ \Theta \vdash \rho gm_1 \simeq_{ctx} \rho gm_2 : \tau &\triangleq \Theta \vdash \rho gm_1 \wedge \Theta \vdash \rho gm_2 \wedge (\forall \overline{S_i}, \overline{\mathcal{D}_j}^{i,j} : \Theta \vdash \tau \longrightarrow \bullet \vdash \tau, \\ &(\overline{\text{spdef } S_i; \text{def } \mathcal{D}_j}^{i,j} ; \rho gm_1 \longrightarrow^* e_1 : \tau \iff \overline{\text{spdef } S_i; \text{def } \mathcal{D}_j}^{i,j} ; \rho gm_2 \longrightarrow^* e_2 : \tau) \\ &\wedge (\bullet; \bullet \Vdash^0 e_1 \simeq_{ctx} e_2 : \tau)) \end{aligned}$$

Expression contextual equivalence says that two core expressions e_1 and e_2 are contextually equivalent, if for any *computation context* \mathbb{C} , $\mathbb{C}[e_1]$ and $\mathbb{C}[e_2]$ evaluate to the same value. A computation context \mathbb{C} is a core expression with a hole in it, and we use the notation $\mathbb{C}[e]$ to plug in the expression e into the hole of \mathbb{C} . The notation $\mathbb{C} : \bullet; \Gamma \Vdash \tau \rightsquigarrow \bullet; \bullet \Vdash^0 \text{Int}$ means that if $\bullet; \Gamma \Vdash e : \tau$ then $\bullet; \bullet \Vdash^0 \mathbb{C}[e] : \text{Int}$. Program contextual equivalence is defined in a similar manner and is built using expression contextual equivalence.

The final piece in our proof is to show that axiomatically equivalent core programs are contextually equivalent, then with Lemma 6.2 we can prove that axiomatically equivalent source programs elaborate to contextually equivalent core programs. The proofs follow those of [Taha et al. \[1998\]](#) and [Taha \[1999\]](#), which are omitted for space reasons. At a high level, this lemma requires us to build *parallel reduction* of F^\square to prove the Church-Rosser property, which is then used to prove equivalence between F^\square axiomatic semantics and operational semantics.

Lemma 6.4 ($F^\square \simeq_{ax}$ to $F^\square \simeq_{ctx}$). *If $\Theta \vdash \rho gm_1 \simeq_{ax} \rho gm_2$, then $\Theta \vdash \rho gm_1 \simeq_{ctx} \rho gm_2 : \tau$.*

Combining Lemma 6.2 and Lemma 6.4 yields our final goal:

Theorem 6.5 ($\lambda^{\square} \Rightarrow_{ax}$ to $F^\square \simeq_{ctx}$). *If $pgm_1 =_{ax} pgm_2$, where $\Theta \vdash pgm_1 : \sigma \rightsquigarrow \rho gm_1$, and $\Theta \vdash pgm_2 : \sigma \rightsquigarrow \rho gm_2$, and $\Theta \rightsquigarrow \Theta$, and $\bullet \vdash \sigma \rightsquigarrow \tau$, then $\Theta \vdash \rho gm_1 \simeq_{ctx} \rho gm_2 : \tau$.*

7 TODAY'S TYPED TEMPLATE HASKELL

The behavior of Typed Template Haskell in GHC differs from our calculus. Figure 8 summarizes the key examples from §2, comparing the results from the latest GHC (9.0.1) to λ^{\square} . The Haskell code examples are in the appendix.

At a high level, GHC's implementation is close to the description in §2.4: it delays type class elaboration until splicing, and excludes local constraints for top-level splices. This is sufficient to accept the definitions of *print1* (and *qnpower*) and *trim*, but it restricts their use: *print1* can only be spliced with a monomorphic type signature, and *trim* can never be spliced, despite its semantics being clear. The central guarantee of typed code quotations is that well-typed code values represent well-typed programs; we view GHC's behavior, in which splicing a well-typed code value can

	<i>print1</i>	<i>printInt</i>	<i>print2</i>	<i>topLift</i>	<i>trim</i>	<i>cancel</i>	<i>qnpower/npower5</i>	
	C1	C2	S1	TS1	A1	A2	§1	S2
Well-staged?	✗	✓	✓	✗	✓	✓	✗/✗	✓/✓
$\lambda[\Rightarrow]$	✗	✓	✓	✗	✓	✓	✗/✗	✓/✓
GHC 9.0.1	✗	✓	<i>O</i>	✗	✗	✗	✗/✗	<i>O</i>

Fig. 8. Examples comparison. Well-staged? indicates well-stagedness after dictionary-passing elaboration. ✗ means the definition itself is accepted but its use is restricted; and *O* means not applicable.

raise a type error, as unsound. Even where it does not lead to unexpected splice-time type errors, delaying type class elaboration can unexpectedly change the semantics of a program when the definition site and the splicing site have different instances in scope.

Finally, because GHC excludes local constraints for top-level splices, it (accidentally) correctly rejects *topLift* (and *npower5*) but wrongly rejects *cancel*. We argue that *topLift* should be rejected because it is ill-staged, and *cancel* should be accepted both because it is well-staged, and because canceling a splice-quotation pair should preserve semantics.

8 INTEGRATION INTO GHC

The goal of this work is to formally study the interaction of type classes and staging, along with the formalism of splice environments, and so we have focused on a source calculus that captures their essence. Integrating our solution into GHC will require additional steps, which we touch on briefly here.

Type inference. We anticipate that type inference for staged constraints will be straightforward to integrate into existing constraint solving algorithms (e.g. Vytiniotis et al. [2011]). The key modification is to track the level of constraints and only solve goals with evidence at the right level. In our formalism, constraints can be solved either by rule **S-SOLVE-INCR** or by rule **S-SOLVE-DECR**. In practice, the implementation only needs to keep track of the level of normal constraints (e.g. when given *CodeC* *C* at level 0, the context can record the spliced evidence for *C* at level 1) so that constraint solving only needs to consider rule **S-SOLVE-DECR**.

Local constraints. Local constraints can be introduced by (for example) pattern matching on GADTs [Peyton Jones et al. 2006], and we anticipate that they can be treated similarly to type class constraints: the constraint solver needs to keep track of the level at which a constraint is introduced and ensure that the constraint is only used at that level.

Quantified constraints. The full Haskell language supports more elaborate forms of type classes than the essence modeled in $\lambda[\Rightarrow]$. For example, GHC supports *quantified constraints* [Bottu et al. 2017], which include forms such as $\forall x. \text{Show } x \Rightarrow \text{Show } (f \ x)$, a constraint that converts *Show* instances for *x* into *Show* instances for *f x*. Future work is required to study more formally the interaction between staged constraints and implication constraints; we envisage that constraint entailment should deduce that *CodeC* ($C_1 \Rightarrow C_2$) entails *CodeC* $C_1 \Rightarrow \text{CodeC } C_2$.

Representation of quotations. In today's GHC implementation, untyped code representations are built compositionally using combinators, and type-checked at splice sites. With our development, code representations contain type information, especially dictionaries, and must therefore correspond to one of GHC's post-typechecking term representations. One option is GHC Core terms, which is the simplest representation that retains type information and has existing serialization support (for inlining definitions across modules).

Our development also requires changing the implementation of splicing to support performing substitution at splices inside quotations. In today's GHC, substitution is performed implicitly during translation from expressions to combinators. With the new representation of quotations, the substitution needs to be represented explicitly and performed explicitly during deserialization of the quotation body. Substituting splices takes two steps. First, a quotation body is traversed and each splice is replaced by a splice variable where the evaluated splice term needs to be inserted. The splice variable is maintained in the splice environment. Second, the splicing operation itself involves checking the splice environment for each splice variable and performing the substitution.

9 RELATED WORK

Since its introduction [Taha and Sheard 1997, 2000] multi-stage programming with quotation has attracted both theoretical and practical interest. Several languages, including MetaOCaml [Kiselyov 2014], Haskell and Scala [Stucki et al. 2018], include implementations of typed quotations.

Considering that implementations of multi-stage languages have supported polymorphism from the very beginning, there is surprisingly little work that formally combines multi-stage programming with polymorphism: most multi-staged calculi are simply-typed. An exception, by Kokaji and Kameyama [2011], involves a language with polymorphism and control effects; their primary concern is the interaction of the value restriction and staging. Another, by Kiselyov [2017], considers the tripartite interaction of polymorphism, cross-stage persistence and mutable cells.

Several works examine the interaction of quotation with individual language features, particularly with various forms of effects, such as control operators [Oishi and Kameyama 2017] and mutable cells [Kiselyov et al. 2016]. Work by Yallop and White [2015] is more closely related to the present work, since there is a well-known correspondence between ML modules and type classes [Wehr and Chakravarty 2008]; it examines the interaction between typed compile-time staging and modules. However, since modules are written explicitly rather than introduced by elaboration, the dictionary level problem does not arise. In a similar vein, Radanne [2017] studies the interaction of ML modules with a different modality, client-server programming, where the distinction between client and server functors corresponds to our distinction between unstaged and staged type class constraints.

Several researchers have combined multi-stage programming and dependent types. Kawata and Igarashi [2019] impose a stage discipline on type variables as on term variables, reflecting the fact that checking types involves evaluating expressions. Pašalic [2004] defines a dependently-typed multi-stage language Meta-D but doesn't consider constraints or parametric polymorphism. Concoction [Fogarty et al. 2007] extends MetaOCaml to support Coq terms within types; it is based on the dependently-typed λ_{HO} [Pašalic et al. 2002], which is motivated by removing tags in generated programs. Brady and Hammond [2006] combine dependent types and multi-stage programming to turn a well-typed interpreter into a verified compiler, but do not consider either parametric polymorphism or constraints.

We are not aware of any work that considers the implications of relevant implicit arguments formally, but there is an informal characterization by Pickering et al. [2019], who also advocated persisting dictionaries between stages, using the fact that dictionary values have top-level names. Unfortunately, that scheme, based on extending the constraint solver to select dictionary representations using both type and level, does not readily extend to local constraints. An alternative approach that the authors later considered, passing constraint derivation trees to allow local construction of future-stage dictionaries, was judged to carry too much run-time overhead to be practical.

Formalising Template Haskell. Sheard and Jones [2002] give a brief description of Untyped Template Haskell. The language is simply-typed and does not account for multiple levels. The language has since diverged: untyped quotations are no longer typechecked before conversion

into their representation. Some aspects of their formalism, notably the Q monad which supports reification of types and declarations, are more suited to the untyped metaprogramming than the typed multi-stage programming we consider here. [Berger et al. 2017] give a more formal study of a core calculus that models some aspects of Untyped Template Haskell, focusing on levels and evaluation rather than these additional features.

Code generators often make use of effects such as let insertion or error reporting so it is useful for to consider the interaction of quotation with effects. In GHC releases since 8.12, the type of quotations is generalised [Pickering 2019] from $Q (TExp\ a)$ to a minimal interface $\forall m. Quote\ m \Rightarrow m\ (TExp\ a)$ giving users more control over which effects are allowed in code generators. We leave formalising this extension to future work.

Modal Type Systems. Several type systems motivated by modal logics have modeled the interaction of modal operators and polymorphism. Attention has turned recently to investigating dependent modal type theories and the complex interaction of modal operators in such theories [Gratzer et al. 2020]. It seems likely that ideas from this research can give a formal account of the interaction of the code modality [Davies and Pfenning 2001] and the parametric quantification from System F which can also be regarded as a modality [Nuyts and Devriese 2018; Pfenning 2001].

10 CONCLUSION

We have proposed a resolution to a longstanding problem in Typed Template Haskell arising from the interaction between two key features, code quotation and type classes. In our view, the mysterious failures that can arise when writing large-scale multi-stage programs are one reason for the limited adoption of Typed Template Haskell. Although it is used in a few developments (e.g. [Pickering et al. 2020]; [Willis et al. 2020]; [Yallop et al. 2018]), take-up is low, despite the many use cases for type-safe optimizing code generators. We hope that the resolution of the shortcomings we have described and the reasoning principles we have established will encourage broader adoption.

Although our work is inspired by Haskell, there is reason to believe that it has wider applications. The recent release of Scala 3 added support for typed code quotations to the language [Stucki et al. 2018]. Preliminary experiments suggest that these quotations suffer from surprising interactions with implicit arguments: implicit resolution within quotations sometimes fails mysteriously. Similarly, it is anticipated that OCaml will soon acquire support both for typed code quotations [Yallop and White 2015] and for implicit arguments [White et al. 2014]. We hope that our work will help to guide the integration of these features and avoid problems with unsoundness from the outset.

ACKNOWLEDGMENTS

We thank Dimitrios Vytiniotis, and the anonymous reviewers for their insightful comments. The work is partly supported by EPSRC Grant *SCOPE: Scoped Contextual Programming with Effects* (EP/S028129/1) and Grant *EXHIBIT: Expressive High-Level Languages for Bidirectional Transformations* (EP/T008911/1).

REFERENCES

- Martin Berger, Laurence Tratt, and Christian Urban. 2017. Modelling Homogeneous Generative Meta-Programming. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs)*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:23. <https://doi.org/10.4230/LIPIcs.ECOOP.2017.5>
- Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified Class Constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Oxford, UK) (Haskell 2017)*. Association for Computing Machinery, New York, NY, USA, 148–161. <https://doi.org/10.1145/3122955.3122967>
- Edwin Brady and Kevin Hammond. 2006. A Verified Staged Interpreter is a Verified Compiler. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (Portland, Oregon, USA) (GPCE '06)*. Association for Computing Machinery, New York, NY, USA, 111–120. <https://doi.org/10.1145/1173706.1173724>

- Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing multi-stage languages using ASTs, Gensym, and reflection. In *Proceedings of the 2nd international conference on Generative programming and component engineering* (Erfurt Germany) (GPCE03). Association for Computing Machinery, 57–76. <https://doi.org/10.5555/954186.954190>
- Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated Type Synonyms. *SIGPLAN Not.* 40, 9 (Sept. 2005), 241–253. <https://doi.org/10.1145/1090189.1086397>
- Rowan Davies and Frank Pfenning. 2001. A Modal Analysis of Staged Computation. *J. ACM* 48, 3 (May 2001), 555–604. <https://doi.org/10.1145/382780.382785>
- Seth Fogarty, Emir Pašalic, Jeremy Siek, and Walid Taha. 2007. Concoction: Indexed Types Now!. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Nice, France) (PEPM '07). Association for Computing Machinery, New York, NY, USA, 112–121. <https://doi.org/10.1145/1244381.1244400>
- Daniel Gratzer, GA Kavvos, Andreas Nuyts, and Lars Birkedal. 2020. Multimodal Dependent Type Theory. (2020). In submission.
- Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (1996), 109–138. <https://doi.org/10.1145/227699.227700>
- Yuichiro Hanada and Atsushi Igarashi. 2014. On Cross-Stage Persistence in Multi-Stage Programming. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science)*, Michael Codish and Eijiro Sumii (Eds.), Vol. 8475. Springer, 103–118. https://doi.org/10.1007/978-3-319-07151-0_7
- M.P. Jones. 1993. *Coherence for qualified types*. Research Report YALEU/DCS/RR-989. Yale University, Dept. of Computer Science.
- Mark P. Jones. 1994. *Qualified Types: Theory and Practice*. Cambridge University Press.
- Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. 2014. Staged Parser Combinators for Efficient Data Processing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) (OOPSLA '14). Association for Computing Machinery, New York, NY, USA, 637–653. <https://doi.org/10.1145/2660193.2660241>
- Akira Kawata and Atsushi Igarashi. 2019. A Dependently Typed Multi-stage Calculus. In *Asian Symposium on Programming Languages and Systems*. Springer, 53–72. https://doi.org/10.1007/978-3-030-34175-6_4
- Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming, Michael Codish and Eijiro Sumii (Eds.)*. Springer International Publishing, Cham, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6
- Oleg Kiselyov. 2017. Generating Code with Polymorphic let: A Ballad of Value Restriction, Copying and Sharing. *Electronic Proceedings in Theoretical Computer Science* 241 (Feb 2017), 1–22. <https://doi.org/10.4204/eptcs.241.1>
- Oleg Kiselyov, Yuki Yoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers - Type- and Scope-Safe Code Generation with Mutable Cells. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science)*, Atsushi Igarashi (Ed.), Vol. 10017. 271–291. https://doi.org/10.1007/978-3-319-47958-3_15
- Yuichiro Kokaji and Yuki Yoshi Kameyama. 2011. Polymorphic multi-stage language with control effects. In *Asian Symposium on Programming Languages and Systems*. Springer, 105–120. https://doi.org/10.1007/978-3-642-25318-8_11
- Neelakantan R. Krishnaswami and Jeremy Yallop. 2019. A Typed, Algebraic Approach to Parsing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) (PLDI 2019). Association for Computing Machinery, New York, NY, USA, 379–393. <https://doi.org/10.1145/3314221.3314625>
- Aleksandar Nanevski. 2002. Meta-Programming with Names and Necessity. (2002), 206–217. <https://doi.org/10.1145/581478.581498>
- Andreas Nuyts and Dominique Devriese. 2018. Degrees of Relatedness: A Unified Framework for Parametricity, Irrelevance, Ad Hoc Polymorphism, Intersections, Unions and Algebra in Dependent Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) (LICS '18). Association for Computing Machinery, New York, NY, USA, 779–788. <https://doi.org/10.1145/3209108.3209119>
- Junpei Oishi and Yuki Yoshi Kameyama. 2017. Staging with control: type-safe multi-stage programming with control operators. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017*, Matthew Flatt and Sebastian Erdweg (Eds.). ACM, 29–40. <https://doi.org/10.1145/3136040.3136049>
- Emir Pašalic. 2004. *The role of type equality in meta-programming*. Ph.D. Dissertation. OGI School of Science & Engineering at OHSU.
- Emir Pašalic, Walid Taha, and Tim Sheard. 2002. Tagless Staged Interpreters for Typed Languages. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA, USA) (ICFP '02). Association for Computing Machinery, New York, NY, USA, 218–229. <https://doi.org/10.1145/581478.581499>

- Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space. In *Haskell Workshop*. Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple Unification-based Type Inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA) (ICFP '06). ACM, New York, NY, USA, 50–61. <https://doi.org/10.1145/1159803.1159811>
- Frank Pfenning. 2001. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 221–230.
- Matthew Pickering. 2019. Overloaded Quotations. GHC proposal. <https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0246-overloaded-bracket.rst>
- Matthew Pickering, Andres Löb, and Nicolas Wu. 2020. Staged sums of products. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2020, Virtual Event, USA, August 7, 2020*, Tom Schrijvers (Ed.). ACM, 122–135. <https://doi.org/10.1145/3406088.3409021>
- Matthew Pickering, Nicolas Wu, and Csongor Kiss. 2019. Multi-Stage Programs in Context. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell* (Berlin, Germany) (Haskell 2019). Association for Computing Machinery, New York, NY, USA, 71–84. <https://doi.org/10.1145/3331545.3342597>
- Gabriel Radanne. 2017. *Tierless Web programming in ML. (Programmation Web sans-étages en ML)*. Ph.D. Dissertation. Paris Diderot University, France. <https://tel.archives-ouvertes.fr/tel-01788885>
- Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) (GPCE '10). ACM, New York, NY, USA, 127–136. <https://doi.org/10.1145/1868294.1868314>
- Evgeny Roubinchtein. 2015. *IR-MetaOCaml: (re)implementing MetaOCaml*. Master's thesis. University of British Columbia. <https://doi.org/10.14288/1.0166800>
- Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Zero-cost Effect Handlers by Staging. (2020). In submission.
- Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A Quick Look at Impredicativity. *Proc. ACM Program. Lang.* 4, ICFP, Article 89 (Aug. 2020), 29 pages. <https://doi.org/10.1145/3408971>
- Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) (Haskell '02). ACM, New York, NY, USA, 1–16. <https://doi.org/10.1145/581690.581691>
- Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A Practical Unification of Multi-Stage Programming and Macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Boston, MA, USA) (GPCE 2018). Association for Computing Machinery, New York, NY, USA, 14–27. <https://doi.org/10.1145/3278122.3278139>
- Walid Taha and Tim Sheard. 1997. Multi-stage Programming with Explicit Annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* (Amsterdam, The Netherlands) (PEPM '97). ACM, New York, NY, USA, 203–217. <https://doi.org/10.1145/258993.259019>
- Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. [https://doi.org/10.1016/S0304-3975\(00\)00053-0](https://doi.org/10.1016/S0304-3975(00)00053-0)
- Walid Taha, Tim Sheard, et al. 1998. Multi-stage programming: Axiomatization and type safety. In *International Colloquium on Automata, Languages, and Programming*. Springer, 918–929.
- Walid Mohamed Taha. 1999. *Multistage programming: its theory and applications*. Oregon Graduate Institute of Science and Technology.
- Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X) Modular Type Inference with Local Assumptions. *J. Funct. Program.* 21, 4-5 (Sept. 2011), 333–412. <https://doi.org/10.1017/S0956796811000098>
- Phillip Wadler and Stephen Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) (POPL '89). ACM, New York, NY, USA, 60–76. <https://doi.org/10.1145/75277.75283>
- Stefan Wehr and Manuel M. T. Chakravarty. 2008. ML Modules and Haskell Type Classes: A Constructive Comparison. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings (Lecture Notes in Computer Science)*, G. Ramalingam (Ed.), Vol. 5356. Springer, 188–204. https://doi.org/10.1007/978-3-540-89330-1_14
- Leo White, Frédéric Bour, and Jeremy Yallop. 2014. Modular implicits. In *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014 (EPTCS)*, Oleg Kiselyov and Jacques Garrigue (Eds.), Vol. 198. 22–63. <https://doi.org/10.4204/EPTCS.198.2>
- Jamie Willis, Nicolas Wu, and Matthew Pickering. 2020. Staged selective parser combinators. *Proc. ACM Program. Lang.* 4, ICFP (2020), 120:1–120:30. <https://doi.org/10.1145/3409002>

- Jeremy Yallop. 2017. Staged Generic Programming. *Proc. ACM Program. Lang.* 1, ICFP, Article 29 (Aug. 2017), 29 pages. <https://doi.org/10.1145/3110273>
- Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. 2018. Partially-Static Data as Free Extension of Algebras. *Proc. ACM Program. Lang.* 2, ICFP, Article 100 (July 2018), 30 pages. <https://doi.org/10.1145/3236795>
- Jeremy Yallop and Leo White. 2015. Modular Macros. OCaml Users and Developers Workshop.

A APPENDIX OVERVIEW

Appendix B presents the code examples used for testing in §7. Appendix C includes a preliminary experiment with Scala.

Appendix D include those omitted rules from the main paper.

The rest sections are for proofs. Appendix E proves type soundness of $F^{\llbracket \cdot \rrbracket}$, and Appendix F proves elaboration soundness from $\lambda^{\llbracket \cdot \rrbracket}$ to $F^{\llbracket \cdot \rrbracket}$.

Appendix G gives an overview of the axiomatic semantics, and Appendix H includes the list of lemmas and Appendix I presents the proofs.

The correspondence between lemmas in the paper and proofs in the appendix are given below.

Lemmas in the paper	Lemmas in the appendix
Theorem 4.2	Theorem E.1
Theorem 4.3	Theorem E.2
Lemma 5.2	Lemma 5.2
Lemma 5.3	Lemma 5.3
Lemma 5.4	Theorem F.4
Theorem 5.5	Theorem F.4
Lemma 6.4	Lemma H.4
Theorem 6.5	Theorem H.5

B CODE EXAMPLES IN TYPED TEMPLATE HASKELL

The Haskell code used for tests in §7 is given below. Path-based cross-stage persistence is modeled in our calculi using top-level definitions, and is implemented in GHC using the *module restriction*, which dictates that only identifiers bound in other modules can be used inside top-level splices. Therefore, the examples are based on two modules: *Toplevel* and *Examples*.

```
-- Toplevel.hs
-- Separated compiled because of module restrictions.
{-# LANGUAGE TemplateHaskell #-}

module Toplevel where

import Language.Haskell.TH
import Language.Haskell.TH.Syntax

data C = C

print1 :: (Quote m, Show a) => Code m (a -> String)
print1 = [| show |]

printInt :: (Quote m) => Code m (Int -> String)
printInt = [| show |]
```

```

1471 readInt :: (Quote m) ⇒ Code m (String → Int)
1472 readInt = [| read |]
1473
1474 trim :: Quote m ⇒ Code m (String → String)
1475 trim = [| $(printInt) . $(readInt) |]
1476
1477 qnpower :: (Quote m, Num a) ⇒ Int → Code m a → Code m a
1478 qnpower 0 qn = [| 1 |]
1479 qnpower k qn = [| ( $(qn) * $(qnpower (k - 1) qn)) |]
1480
1481 class MyShow a where
1482   myshow :: a → String
1483
1484 instance Show a ⇒ MyShow [a] where
1485   myshow = show
1486
1487 printListInt :: (Quote m) ⇒ Code m ([Int] → String)
1488 printListInt = [| myshow |]
1489
1490 -- Examples.hs
1491 {-# LANGUAGE TemplateHaskell, FlexibleInstances, FlexibleContexts #-}
1492
1493 module Examples where
1494
1495 import Language.Haskell.TH
1496 import Language.Haskell.TH.Syntax
1497 import Toplevel
1498
1499 -- rejected:
1500 -- No instance for (Show a) arising from a use of 'print1'
1501 -- In the expression: print1
1502 splicePolyPrint1 :: Show a ⇒ a → String
1503 splicePolyPrint1 = $(print1)
1504
1505 -- Monomorphic splice is OK
1506 spliceMonoPrint1 :: Int → String
1507 spliceMonoPrint1 = $(print1)
1508
1509 -- rejected:
1510 -- No instance for (Lift C) arising from a use of 'liftTyped'
1511 topLift :: Lift C ⇒ C
1512 topLift = $(liftTyped C)
1513
1514 -- rejected:
1515 -- No instance for (Show a) arising from a use of 'show'
1516 cancel :: Show a ⇒ a → String
1517 cancel = $( [| show | ])
1518
1519

```

```

1520 -- rejected:
1521 -- Ambiguous type variable 'b0' arising from a use of 'show'
1522 -- prevents the constraint '(Show b0)' from being solved.
1523 strim :: String
1524 strim = ($(trim) "123")
1525
1526 -- The module Toplevel defines an instance for MyShow [a] using normal show.
1527 -- This example is to show the inconsistent behavior when the splicing site and
1528 -- the definition site has given different instances.
1529 instance {-# OVERLAPPING #-} MyShow [Int] where
1530     myshow _ = "hello"
1531
1532 usePrintListInt :: String
1533 usePrintListInt = $(printListInt) [1,2,3] -- "hello"
1534
1535 -- rejected:
1536 -- No instance for (Num a) arising from a use of 'qnpower'
1537 -- In the expression: qnpower 5 ([| n |])
1538 qnpower5 :: Num a => a -> a
1539 qnpower5 n = $(qnpower 5 ([| n |])) -- Error!
1540

```

C PRELIMINARY EXPERIMENTS IN SCALA

We have tested examples with implicits in Scala3. While implicits are rather different to type classes, we observe similar difficult-to-explain behaviors of interaction between implicits and staging. Specifically, we discuss our attempts to define the *power* example introduced in the introduction.

Scala does a good job rejecting the directly translated *power* example. As we can see below, the implicit argument is introduced explicitly as a binding, so Scala can identify the ill-stagedness.

```

1548 import scala.quoted.*
1549 import math.Numeric.Implicits.infixNumericOps
1550
1551 // rejected:
1552 // case k => '{ ${cn} * ${power (k - 1, cn) } }
1553 //           ^
1554 //           access to parameter num from wrong staging level:
1555 //           - the definition is at level 0,
1556 //           - but the access is at level 1.
1557 def power [A] (using Quotes) (x : Int, cn : Expr[A])(implicit num: Numeric[A],
1558     t:Type[A]) : Expr[A] =
1559     x match
1560     case 0 => '{ num.fromInt(1) }
1561     case k => '{ ${cn} * ${power (k - 1, cn) } }
1562

```

We then tried different ways to move the parameter at level 1. Scala is not happy about the following definition. In this case, Scala complains about a type mismatch between quotations, while the *implicit num* is bound in a well-staged manner.

```

1566 // rejected:
1567 // Found:    quoted.Expr[(Numeric[A]) ?=> A]
1568

```



```

1569 // Required: quoted.Expr[A]
1570 import scala.quoted.*
1571 import math.Numeric.Implicits.infixNumericOps
1572
1573 def power [A] (using Quotes) (x : Int, cn : Expr[A]) (implicit t:Type[A]) :
1574   Expr[Numeric[A] => A] =
1575   '{ implicit num : Numeric[A] =>
1576     ${x match
1577       case 0 => '{ num.fromInt(1) }
1578       case k => '{ ${cn} * ${power (k - 1, cn) } } }}
1579
1580 Surprisingly, Scala accepts the code if the implicit argument is supplied explicitly.
1581
1582 // accepted
1583 def power[A: Type](using Quotes)(x: Int, cn: Expr[A]): Expr[Numeric[A] ?=> A] =
1584   x match
1585     case 0 => '{ Numeric[A].fromInt(1) }
1586     case k => '{ ${cn} * ${power(k - 1, cn)}(using Numeric[A]) }
1587   // OR
1588   // case k => '{ num ?=> ${cn} * ${power(k - 1, cn)}(using num) }

```

Our preliminary conclusion is that like Typed Template Haskell, the interaction between quotation and overloading haven't been fully worked out in Scala, either. More systematic investigations are needed to identify the exact problem and possible solutions.

D COMPLETE RULES

This section contains the omitted rules for $\lambda^{\llbracket \Rightarrow \rrbracket}$ and $F^{\llbracket \rrbracket}$.

D.1 Complete Rules for $\lambda^{\llbracket \Rightarrow \rrbracket}$

program context	$\Theta ::=$	$\bullet \mid \Theta, k : \sigma \mid \Theta, \text{ev} : \forall \bar{a}_i^i. \bar{C}_j^j \Rightarrow C$
context	$\Gamma ::=$	$\bullet \mid \Gamma, x : (\tau, n) \mid \Gamma, a \mid \Gamma, (C, n)$

D.1.1 Elaborating Contexts.

$\Gamma \rightsquigarrow \Delta$	(Elaborating Contexts)	
S-CTX-EMPTY	S-CTX-VAR	S-CTX-TVVAR
$\bullet \rightsquigarrow \bullet$	$\frac{\Gamma \rightsquigarrow \Delta \quad \Gamma \vdash \tau \rightsquigarrow \tau'}{\Gamma, x : (\tau, n) \rightsquigarrow \Delta, x : (\tau', n)}$	$\frac{\Gamma \rightsquigarrow \Delta}{\Gamma, a \rightsquigarrow \Delta, a}$
	S-CTX-EV	
	$\frac{\Gamma \rightsquigarrow \Delta \quad \Gamma \vdash C \rightsquigarrow \tau}{\Gamma, \text{ev} : (C, n) \rightsquigarrow \Delta, \text{ev} : (\tau, n)}$	
$\Theta \rightsquigarrow \Theta$	(Elaborating Program Contexts)	
S-PCTX-EMPTY	S-PCTX-KVAR	S-PCTX-EV
$\bullet \rightsquigarrow \bullet$	$\frac{\Theta \rightsquigarrow \Theta \quad \bullet \vdash \sigma \rightsquigarrow \tau}{\Theta, k : \sigma \rightsquigarrow \Theta, k : \tau}$	$\frac{\Theta \rightsquigarrow \Theta \quad \overline{\bar{a}_i^i \vdash C_j \rightsquigarrow \tau_j^j}^j \quad \bar{a}_i^i \vdash C \rightsquigarrow \tau}{\Theta, \text{ev} : \forall \bar{a}_i^i. \bar{C}_j^j \Rightarrow C \rightsquigarrow \Theta, \text{ev} : \bar{\tau}_j^j \rightarrow \tau}$

D.1.2 Elaborating Programs.

$\Theta \vdash \text{pgm} : \sigma \rightsquigarrow \rho \text{gm}$ (Typing programs)

$$\begin{array}{c}
 \text{S-PGM-DEF} \\
 \frac{\rho \text{gm}_1; \Theta_1 \vdash \mathcal{D} \dashv \Theta_2 \rightsquigarrow \rho \text{gm}_2 \quad \Theta_2 \vdash \text{pgm} : \sigma \rightsquigarrow \rho \text{gm}_1}{\Theta_1 \vdash \text{def } \mathcal{D}; \text{pgm} : \sigma \rightsquigarrow \rho \text{gm}_2} \\
 \text{S-PGM-CLS} \\
 \frac{\rho \text{gm}_1; \Theta_1 \vdash C \dashv \Theta_2 \rightsquigarrow \rho \text{gm}_2 \quad \Theta_2 \vdash \text{pgm} : \sigma \rightsquigarrow \rho \text{gm}_1}{\Theta_1 \vdash \text{class } C; \text{pgm} : \sigma \rightsquigarrow \rho \text{gm}_2} \\
 \text{S-PGM-INST} \\
 \frac{\rho \text{gm}_1; \Theta_1 \vdash I \dashv \Theta_2 \rightsquigarrow \rho \text{gm}_2 \quad \Theta_2 \vdash \text{pgm} : \sigma \rightsquigarrow \rho \text{gm}_1}{\Theta_1 \vdash \text{inst } I; \text{pgm} : \sigma \rightsquigarrow \rho \text{gm}_2} \\
 \text{S-PGM-EXPR} \\
 \frac{\Theta; \bullet \vdash^0 e : \sigma \rightsquigarrow e \mid \phi \quad \bullet \vdash \sigma \rightsquigarrow \tau \quad e : \tau \vdash^1 \phi \rightsquigarrow \rho \text{gm}}{\Theta \vdash e : \sigma \rightsquigarrow \rho \text{gm}}
 \end{array}$$

$\rho \text{gm}_1; \Theta_1 \vdash \mathcal{D} \dashv \Theta_2 \rightsquigarrow \rho \text{gm}_2$ (Typing definitions)

$$\begin{array}{c}
 \text{S-DEF} \\
 \frac{\Theta; \bullet \vdash^0 e : \sigma \rightsquigarrow e \mid \phi \quad \bullet \vdash \sigma \rightsquigarrow \tau \quad \text{def } k : \tau = e; \rho \text{gm}_1 \vdash^1 \phi \rightsquigarrow \rho \text{gm}_2}{\rho \text{gm}_1; \Theta \vdash k = e \dashv \Theta, k : \sigma \rightsquigarrow \rho \text{gm}_2}
 \end{array}$$

$\rho \text{gm}_1; \Theta_1 \vdash C \dashv \Theta_2 \rightsquigarrow \rho \text{gm}_2$ (Typing class definitions)

$$\begin{array}{c}
 \text{S-CLS} \\
 \frac{a \vdash \rho \rightsquigarrow \tau}{\rho \text{gm}; \Theta \vdash \text{TC } a \text{ where } \{k : \rho\} \dashv \Theta, k : \forall a. \text{TC } a \Rightarrow \rho \rightsquigarrow \text{def } k : \forall a. \tau \rightarrow \tau = \Lambda a. \lambda x : \tau. x; \rho \text{gm}}
 \end{array}$$

$\rho \text{gm}_1; \Theta_1 \vdash I \dashv \Theta_2 \rightsquigarrow \rho \text{gm}_2$ (Typing instance definitions)

$$\begin{array}{c}
 \text{S-INST} \\
 \frac{\begin{array}{c} \text{TC } a \text{ where } \{k : \rho\} \\ \overline{b_j^j} = \text{ftv}(\tau) \quad \overline{b_j^j} \vdash C_i \rightsquigarrow \tau_i^i \quad \Theta; \overline{b_j^j}, \overline{ev_i} : (C_i, 0)^i \vdash^0 e : \rho[a \mapsto \tau] \rightsquigarrow e \mid \phi_1 \\ \bullet \vdash \rho[a \mapsto \tau] \rightsquigarrow \tau \quad \text{fresh } \overline{ev_i}^i \quad \phi_1 \dashv (\overline{b_j^j}, \overline{ev_i} : (\tau_i, 0)^i) \rightsquigarrow \phi_2 \\ \text{def } ev : \forall \overline{b_j^j}. \overline{\tau_i}^i \rightarrow \tau = \Lambda \overline{b_j^j}. \lambda \overline{ev_i} : \overline{\tau_i}^i. e; \rho \text{gm}_1 \vdash^1 \phi_2 \rightsquigarrow \rho \text{gm}_2 \quad \text{fresh } ev \end{array}}{\rho \text{gm}_1; \Theta \vdash \overline{C_i}^i \Rightarrow \text{TC } \tau \text{ where } \{k = e\} \dashv \Theta, \overline{ev} : \forall \overline{b_j^j}. \overline{C_i}^i \Rightarrow \text{TC } \tau \rightsquigarrow \rho \text{gm}_2}
 \end{array}$$

D.2 Complete Rules for F^\square

context	$\mathbb{C} ::= \square \mid \lambda x : \tau. \mathbb{C} \mid \mathbb{C} e \mid e \mathbb{C} \mid \Lambda a. \mathbb{C} \mid \mathbb{C} \tau \mid [\mathbb{C}]_\phi \mid [e]_s$
splice context	$\mathbb{S} ::= \phi, \Delta \vdash^n s : \tau = \mathbb{C} \mid \mathbb{S}, \Delta \vdash^n s : \tau = e$

D.2.1 Axiomatic equivalence.

Axioms	
$\llbracket e_1 \rrbracket_{\phi_1, \Delta \vdash^{\text{H}} s : \tau} =_{ax} \llbracket e_2 \rrbracket_{\phi_2, \phi_1 \rightsquigarrow \phi_2}$	$=_{ax} \llbracket e_1[s \mapsto e_2] \rrbracket_{\phi_1, \phi_1', \phi_2}$ where $\phi \rightsquigarrow \Delta \rightsquigarrow \phi'$
$\llbracket s \rrbracket_{\bullet \vdash^{\text{H}} s : \tau = e}$	$=_{ax} e$
$(\lambda x : \tau. e_1) e_2$	$=_{ax} e_1[x \mapsto e_2]$
$(\Lambda a. e) \tau$	$=_{ax} e[a \mapsto \tau]$

$e_1 =_{ax} e_2$ is the axiomatic equivalence relation between $F^{\llbracket \cdot \rrbracket}$ expressions that is the contextual and equivalence closure of the axioms.

$\boxed{e_1 =_{ax} e_2}$ (Axiomatic equality)

EQ-REFL	EQ-SYMM	EQ-TRANS	EQ-CTX
$\frac{}{e =_{ax} e}$	$\frac{e_1 =_{ax} e_2}{e_2 =_{ax} e_1}$	$\frac{e_1 =_{ax} e_2 \quad e_2 =_{ax} e_3}{e_1 =_{ax} e_3}$	$\frac{e_1 =_{ax} e_2 \quad \mathbb{C}_1 =_{ax} \mathbb{C}_2}{\mathbb{C}_1[e_1] =_{ax} \mathbb{C}_2[e_2]}$

$\rho gm_1 =_{ax} \rho gm_2$ is the axiomatic equivalence relation between $F^{\llbracket \cdot \rrbracket}$ programs.

$\boxed{\rho gm_1 =_{ax} \rho gm_2}$ (Axiomatic equality)

PEQ-DEF	PEQ-EXPR
$\frac{e_1 =_{ax} e_2 \quad \rho gm_1 =_{ax} \rho gm_2}{\text{def } k : \tau = e_1; \rho gm_1 =_{ax} \text{def } k : \tau = e_2; \rho gm_2}$	$\frac{e_1 =_{ax} e_2}{e_1 : \tau =_{ax} e_2 : \tau}$
$\frac{e_1 =_{ax} e_2 \quad \rho gm_1 =_{ax} \rho gm_2}{\text{spdef } \Delta \vdash^{\text{H}} s : \tau = e_1; \rho gm_1 =_{ax} \text{spdef } \Delta \vdash^{\text{H}} s : \tau = e_2; \rho gm_2}$	
PEQ-SPDEF	PEQ-SPDEF-AX
$\frac{\phi \rightsquigarrow \Delta \rightsquigarrow \phi'}{\text{spdef } \Delta \vdash^{\text{H}} s : \tau = \llbracket e \rrbracket_{\phi}; \rho gm =_{ax} \text{spdef } \phi'; \rho gm[s \mapsto e]}$	

E PROOFS FOR TYPE SOUNDNESS OF $F^{\llbracket \cdot \rrbracket}$

E.1 Progress

Theorem E.1 (Progress).

- (1) If $\bullet; \Delta \vdash^{\text{H}} e : \tau$, where $\Delta \succ n$, then either e is a value, or $e \longrightarrow e'$ for some e' .
- (2) If $\bullet; \Delta \vdash^{\text{H}} \phi$, where $\Delta \succ n$, then either ϕ is ϕ_v , or $\phi \longrightarrow \phi'$ for some ϕ' .
- (3) If $\bullet \vdash \mathcal{D} \dashv \Theta$, then either \mathcal{D} is $k : \tau = v$, or $\mathcal{D} \longrightarrow \mathcal{D}'$ for some \mathcal{D}' .
- (4) If $\bullet \vdash \mathcal{S} \dashv \Theta$, then either \mathcal{S} is $\Delta \vdash^{\text{H}} s : \tau = \llbracket e \rrbracket_{\phi_v}$, or $\mathcal{S} \longrightarrow \mathcal{S}'$ for some \mathcal{S}' .
- (5) If $\bullet \vdash \rho gm$, then either ρgm is $v : \tau$, or $\rho gm \longrightarrow \rho gm'$ for some $\rho gm'$.

PROOF. By induction on typing.

Part 1 • Case rule **C-LIT**. i is a value.

- Case rule **C-VAR**. Impossible case, since Δ has no level- n items.
- Case rule **C-KVAR**. Impossible case, since the program environment is \bullet .
- Case rule **C-SVAR**. Impossible case, since Δ has no level- n items.
- Case rule **C-TOP-SVAR**. Impossible case, since the program environment is \bullet .
- Case rule **C-ABS**. The expression $\lambda x : \tau. e$ is a value.
- Case rule **C-APP**.

$$\frac{\Theta; \Gamma \vdash^{\text{H}} e_1 : \tau_1 \longrightarrow \tau_2 \quad \Theta; \Gamma \vdash^{\text{H}} e_2 : \tau_1}{\Theta; \Gamma \vdash^{\text{H}} e_1 e_2 : \tau_2}$$

By I.H., we have either e_1 is a value, or $e_1 \longrightarrow e'_1$ for some e'_1 .

– e_1 is a value. Then we know that e_1 must be $\lambda x : \tau. e$ for some e . So by rule **CE-BETA** we have $(\lambda x : \tau. e) e_2 \longrightarrow e[x \mapsto e_2]$.

– $e_1 \longrightarrow e'_1$. By rule **CE-APP** we have $e_1 e_2 \longrightarrow e'_1 e_2$.

• Case rule **C-TABS**. The expression $\Lambda a. e$ is a value.

• Case rule **C-TAPP**.

$$\frac{\text{C-TAPP} \quad \Theta; \Gamma \vdash^n e : \forall a. \tau_2}{\Theta; \Gamma \vdash^n e \tau_1 : \tau_2[a \mapsto \tau_1]}$$

By I.H., we have either e is a value, or $e \longrightarrow e'$ for some e' .

– e is a value. We know that e must be $\Lambda a. e_1$ for some e . So by rule **CE-TBETA**

– $e \longrightarrow e'$. By rule **CE-TAPP** we have $e \tau_1 \longrightarrow e' \tau_1$.

• Case rule **C-QUOTE**.

$$\frac{\text{C-QUOTE} \quad \Theta; \Gamma \vdash^n \phi \quad \Theta; \Gamma, \phi^\Gamma \vdash^{n+1} e : \tau}{\Theta; \Gamma \vdash^n \llbracket e \rrbracket_\phi : \text{Code } \tau}$$

By Part 2, we know that either ϕ is ϕ_v , or $\phi \longrightarrow \phi'$ for some ϕ' . In the first case, the expression $\llbracket e \rrbracket_{\phi_v}$ is a value. In the second case, by rule **CE-QUOTE** we have $\llbracket e \rrbracket_\phi \longrightarrow \llbracket e \rrbracket_{\phi'}$.

Part 2 • Case rule **C-S-EMPTY**. • is ϕ_v .

• Case rule **C-S-CONS**.

$$\frac{\text{C-S-CONS} \quad \Theta; \Gamma \vdash \phi \quad \Theta; \Gamma, \Delta \vdash^n e : \text{Code } \tau \quad \Delta \succ n}{\Theta; \Gamma \vdash \phi, (\Delta \vdash^n s : \tau = e)}$$

By I.H., we know either ϕ is some ϕ_v , or $\phi \longrightarrow \phi'$. In the first case, by Part 1, we know that either e is a value, or $e \longrightarrow e'$ for some e . If e is a value, we know that $\phi_v, \Delta \vdash^n s : \tau = e$ is some ϕ_v' . If e reduces, then by rule **CE-S-TAIL** we have $\phi_v, \Delta \vdash^n s : \tau = e \longrightarrow \phi_v, \Delta \vdash^n s : \tau = e'$.

In the second case, by rule **CE-S-HEAD** we have $\phi, \Delta \vdash^n s : \tau = e \longrightarrow \phi', \Delta \vdash^n s : \tau = e$.

Part 3 We have

$$\frac{\text{C-DEF} \quad \Theta; \bullet \vdash^0 e : \tau}{\Theta \vdash k : \tau = e \dashv \Theta_1, k : \tau}$$

By Part 1, we know that e is either a value, or $e \longrightarrow e'$. In the first case, we have proved the goal. In the second case, by rule **CE-DEF** we have $k : \tau = e \longrightarrow k : \tau = e'$.

Part 4 We have

$$\frac{\text{C-SPDEF} \quad \Theta; \Delta \vdash^n e : \text{Code } \tau \quad \Delta \succ n}{\Theta \vdash \Delta \vdash^n s : \tau = e \dashv \Theta, s : (\Delta, \tau, n + 1)}$$

By Part 1, we know that e is either a value, or $e \longrightarrow e'$. In the first case, since e is of type $\text{Code } \tau$, we know that e must be $\llbracket e' \rrbracket_{\phi_v}$, which proves the goal. In the second case, by rule **CE-SPDEF** we have $\Delta \vdash^n s : \tau = e \longrightarrow \Delta \vdash^n s : \tau = e'$.

Part 5 • Case rule **C-PGM-DEF**.

$$\frac{\text{C-PGM-DEF} \quad \Theta_1 \vdash \mathcal{D} \dashv \Theta_2 \quad \Theta_2 \vdash \rho gm}{\Theta_1 \vdash \text{def } \mathcal{D}; \rho gm}$$

By Part 3, we know that either \mathcal{D} is $k : \tau = v$, or $\mathcal{D} \longrightarrow \mathcal{D}'$. In the first case, by rule **CE-PGM-DBETA** we have $\text{def } k : \tau = v; \rho gm \longrightarrow \rho gm[k \mapsto v]$. In the second case, by rule **CE-PGM-DEF** we have $\text{def } \mathcal{D}; \rho gm \longrightarrow \text{def } \mathcal{D}'; \rho gm$.

• Case rule **C-PGM-SPDEF**.

$$\frac{\text{C-PGM-SPDEF} \quad \Theta_1 \vdash \mathcal{S} \dashv \Theta_2 \quad \Theta_2 \vdash \rho gm}{\Theta_1 \vdash \text{spdef } \mathcal{S}; \rho gm}$$

By Part 4, we know that either \mathcal{S} is $\Delta \vdash^{\text{I}} s : \tau = \llbracket e \rrbracket_{\phi_v}$, or $\mathcal{S} \longrightarrow \mathcal{S}'$. In the first case, by rule **CE-PGM-SPBETA** we have $\text{spdef } \Delta \vdash^{\text{I}} s : \tau = \llbracket e \rrbracket_{\phi_v}; \rho gm \longrightarrow \rho gm[s \mapsto [\phi_v]e]$. In the second case, by rule **CE-PGM-SPDEF** we have $\text{spdef } \mathcal{S}; \rho gm \longrightarrow \text{spdef } \mathcal{S}'; \rho gm$.

- Case rule **C-PGM-EXPR**.

$$\frac{\text{C-PGM-EXPR} \quad \Theta; \bullet \vdash^0 e : \tau}{\Theta \vdash e : \tau}$$

By Part 1, we know that either e is a value, or $e \longrightarrow e'$. In the first case, we have $e : \tau$ which proves the goal. In the second case, by rule **CE-PGM-EXPR** we have $e : \tau \longrightarrow e' : \tau$.

□

E.2 Preservation

Theorem E.2 (Preservation).

- (1) If $\Theta; \Delta \vdash^{\text{I}} e : \tau$, and $e \longrightarrow e'$, then $\Theta; \Delta \vdash^{\text{I}} e' : \tau$.
- (2) If $\Theta; \Delta \vdash^{\text{I}} \phi$, and $\phi \longrightarrow \phi'$, then $\Theta; \Delta \vdash^{\text{I}} \phi'$, and $\phi^{\Gamma} = \phi'^{\Gamma}$.
- (3) If $\Theta_1 \vdash \mathcal{D} \dashv \Theta_2$, and $\mathcal{D} \longrightarrow \mathcal{D}'$, then $\Theta_1 \vdash \mathcal{D}' \dashv \Theta_2$.
- (4) If $\Theta_1 \vdash \mathcal{S} \dashv \Theta_2$, and $\mathcal{S} \longrightarrow \mathcal{S}'$, then $\Theta_1 \vdash \mathcal{S}' \dashv \Theta_2$.
- (5) If $\Theta \vdash \rho gm$, and $\rho gm \longrightarrow \rho gm'$, then $\Theta \vdash \rho gm'$.

PROOF. By induction on typing.

Part 1 • Case

$$\frac{\text{CE-APP} \quad e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2}$$

$$\begin{array}{l|l} \Theta; \Delta \vdash^{\text{I}} e_1 e_2 : \tau_2 & \text{given} \\ \Theta; \Delta \vdash^{\text{I}} e_1 : \tau_1 \rightarrow \tau_2 & \text{inversion (rule C-APP)} \\ \Theta; \Delta \vdash^{\text{I}} e_2 : \tau_1 & \\ \Theta; \Delta \vdash^{\text{I}} e'_1 : \tau_1 \rightarrow \tau_2 & \text{I.H.} \\ \Theta; \Delta \vdash^{\text{I}} e'_1 e_2 : \tau_2 & \text{rule C-APP} \end{array}$$

- Case

$$\frac{\text{CE-BETA}}{(\lambda x : \tau. e_1) e_2 \longrightarrow e_1[x \mapsto e_2]}$$

$$\begin{array}{l|l} \Theta; \Delta \vdash^{\text{I}} (\lambda x : \tau. e_1) e_2 : \tau_2 & \text{given} \\ \Theta; \Delta \vdash^{\text{I}} \lambda x : \tau. e_1 : \tau_1 \rightarrow \tau_2 & \text{inversion (rule C-APP)} \\ \Theta; \Delta \vdash^{\text{I}} e_2 : \tau_1 & \\ \Theta; \Delta, x : (\tau_1, n) \vdash^{\text{I}} e_1 : \tau_2 & \text{inversion (rule C-ABS)} \\ \Theta; \Delta \vdash^{\text{I}} e_1[x \mapsto e_2] : \tau_2 & \text{by substitution} \end{array}$$

- Case

$$\frac{\text{CE-TAPP} \quad e \longrightarrow e'}{e \tau \longrightarrow e' \tau}$$

$$\Theta; \Delta \vdash^{\text{I}} e \tau_1 : \tau_2[a \mapsto \tau_1] \quad | \quad \text{given}$$

1814 $\Theta; \Delta \vdash^{\mathfrak{n}} e_1 : \forall a. \tau_2$ | inversion (rule **C-TAPP**)
 1815 $\Theta; \Delta \vdash^{\mathfrak{n}} e'_1 : \forall a. \tau_2$ | I.H.
 1816 $\Theta; \Delta \vdash^{\mathfrak{n}} e'_1 \tau_1 : \tau_2[a \mapsto \tau_1]$ | rule **C-TAPP**
 1817
 1818 • Case
 1819
$$\frac{\text{CE-TBETA}}{(\Lambda a.e) \tau \longrightarrow e[a \mapsto \tau]}$$

 1820
 1821 $\Theta; \Delta \vdash^{\mathfrak{n}} (\Lambda a.e) \tau : \tau_1[a \mapsto \tau]$ | given
 1822 $\Theta; \Delta \vdash^{\mathfrak{n}} \Lambda a.e : \forall a. \tau_1$ | inversion (rule **C-TAPP**)
 1823 $\Theta; \Delta, a \vdash^{\mathfrak{n}} e : \tau_1$ | inversion (rule **C-TABS**)
 1824 $\Theta; \Delta \vdash^{\mathfrak{n}} e[a \mapsto \tau] : \tau_1[a \mapsto \tau]$ | by substitution
 1825

1826 • Case
 1827
$$\frac{\text{CE-QUOTE}}{\frac{\phi \longrightarrow \phi'}{\llbracket e \rrbracket_{\phi} \longrightarrow \llbracket e \rrbracket_{\phi'}}}$$

 1828
 1829
 1830
 1831 $\Theta; \Delta \vdash^{\mathfrak{n}} \llbracket e \rrbracket_{\phi} : \tau$ | given
 1832 $\Theta; \Delta \vdash^{\mathfrak{n}} \phi$ | inversion (rule **C-QUOTE**)
 1833 $\Theta; \Delta, \phi^{\Gamma} \vdash^{\mathfrak{n}+1} e : \tau$
 1834 $\Theta; \Delta \vdash^{\mathfrak{n}} \phi'$ | Part 2
 1835 $\phi^{\Gamma} = \phi'^{\Gamma}$ | Part 2
 1836 $\Theta; \Delta \vdash^{\mathfrak{n}} \llbracket e \rrbracket_{\phi'} : \tau$ | rule **C-QUOTE**

Part 2 • Case
CE-S-HEAD

1837
 1838
$$\frac{\phi \longrightarrow \phi'}{\phi, \Delta \vdash^{\mathfrak{n}} s : \tau = e \longrightarrow \phi', \Delta \vdash^{\mathfrak{n}} s : \tau = e}$$

 1839
 1840
 1841
 1842 $\Theta; \Delta_1 \vdash^{\mathfrak{n}} \phi, \Delta \vdash^{\mathfrak{n}} s : \tau = e$ | given
 1843 $\Theta; \Delta_1 \vdash^{\mathfrak{n}} \phi$ | inversion (rule **C-S-CONS**)
 1844 $\Delta \dot{>} n$
 1845 $\Theta; \Gamma \vdash^{\mathfrak{n}} e : \text{Code } \tau$
 1846 $\Theta; \Delta_1 \vdash^{\mathfrak{n}} \phi'$ | I.H.
 1847 $\Theta; \Delta_1 \vdash^{\mathfrak{n}} \phi', \Delta \vdash^{\mathfrak{n}} s : \tau = e$ | rule **C-S-CONS**

1848 • Case
 1849
$$\frac{\text{CE-S-TAIL}}{e \longrightarrow e'}$$

 1850
 1851
$$\frac{}{\phi_v, \Delta \vdash^{\mathfrak{n}} s : \tau = e \longrightarrow \phi_v, \Delta \vdash^{\mathfrak{n}} s : \tau = e'}$$

 1852
 1853 $\Theta; \Delta_1 \vdash^{\mathfrak{n}} \phi, \Delta \vdash^{\mathfrak{n}} s : \tau = e$ | given
 1854 $\Theta; \Delta_1 \vdash^{\mathfrak{n}} \phi$ | inversion (rule **C-S-CONS**)
 1855 $\Delta \dot{>} n$
 1856 $\Theta; \Gamma \vdash^{\mathfrak{n}} e : \text{Code } \tau$
 1857 $\Theta; \Gamma \vdash^{\mathfrak{n}} e' : \text{Code } \tau$ | Part 1
 1858 $\Theta; \Delta_1 \vdash^{\mathfrak{n}} \phi, \Delta \vdash^{\mathfrak{n}} s : \tau = e'$ | rule **C-S-CONS**
 1859
 1860

Part 3 Case

CE-DEF

$$\frac{e \longrightarrow e'}{k : \tau = e \longrightarrow k : \tau = e'}$$

$$\begin{array}{l|l} \Theta_1 \vdash k : \tau = e \dashv \Theta_1, k : \tau & \text{given} \\ \Theta_1; \bullet \vdash^0 e : \tau & \text{inversion (rule C-DEF)} \\ \Theta_1; \bullet \vdash^0 e' : \tau & \text{Part 1} \\ \Theta_1 \vdash k : \tau = e' \dashv \Theta_1, k : \tau & \text{rule C-DEF} \end{array}$$

Part 4 Case

CE-SPDEF

$$\frac{e \longrightarrow e'}{\Delta \vdash^n s : \tau = e \longrightarrow \Delta \vdash^n s : \tau = e'}$$

$$\begin{array}{l|l} \Theta_1 \vdash (\Delta \vdash^n s : \tau = e) \dashv \Theta_1, s : (\Delta, \tau, n + 1) & \text{given} \\ \Delta \succ n & \text{inversion (rule C-SPDEF)} \\ \Theta_1; \Delta \vdash^n e : \text{Code } \tau & \\ \Theta; \Delta \vdash^n e' : \text{Code } \tau & \text{Part 1} \\ \Theta_1 \vdash (\Delta \vdash^n s : \tau = e') \dashv \Theta_1, s : (\Delta, \tau, n + 1) & \text{rule C-SPDEF} \end{array}$$

Part 5 • Case

CE-PGM-DEF

$$\frac{\mathcal{D} \longrightarrow \mathcal{D}'}{\text{def } \mathcal{D}; \rho gm \longrightarrow \text{def } \mathcal{D}'; \rho gm}$$

$$\begin{array}{l|l} \Theta_1 \vdash \text{def } \mathcal{D}; \rho gm & \text{given} \\ \Theta_1 \vdash \mathcal{D} \dashv \Theta_2 & \text{inversion (rule C-PGM-DEF)} \\ \Theta_2 \vdash \rho gm & \\ \Theta_1 \vdash \mathcal{D}' \dashv \Theta_2 & \text{Part 3} \\ \Theta_1 \vdash \text{def } \mathcal{D}'; \rho gm & \text{rule C-PGM-DEF} \end{array}$$

• Case

CE-PGM-DBETA

$$\frac{}{\text{def } k : \tau = v; \rho gm \longrightarrow \rho gm[k \mapsto v]}$$

$$\begin{array}{l|l} \Theta_1 \vdash \text{def } k : \tau = v; \rho gm & \text{given} \\ \Theta_1 \vdash k : \tau = v \dashv \Theta_1, k : \tau & \text{inversion (rule C-PGM-DEF), rule C-DEF} \\ \Theta_1; \bullet \vdash^0 v : \tau & \text{inversion (rule C-DEF)} \\ \Theta_1, k : \tau \vdash \rho gm & \\ \Theta_1 \vdash \rho gm[k \mapsto v] & \text{by substitution} \end{array}$$

• Case

CE-PGM-SPDEF

$$\frac{S \longrightarrow S'}{\text{spdef } S; \rho gm \longrightarrow \text{spdef } S'; \rho gm}$$

$$\begin{array}{l|l} \Theta_1 \vdash \text{spdef } S; \rho gm & \text{given} \\ \Theta_1 \vdash S \dashv \Theta_2 & \text{inversion (rule C-PGM-SPDEF)} \\ \Theta_2 \vdash \rho gm & \\ \Theta_1 \vdash S' \dashv \Theta_2 & \text{Part 4} \\ \Theta_1 \vdash \text{spdef } S'; \rho gm & \text{rule C-PGM-SPDEF} \end{array}$$

• Case

CE-PGM-SPBETA

$$\text{spdef } \Delta \vdash^n s : \tau = \llbracket e \rrbracket_{\phi_v}; \rho gm \longrightarrow \rho gm[s \mapsto (\llbracket \phi_v \rrbracket e)]$$

$\Theta_1 \vdash \text{spdef } \Delta \vdash^n s : \tau = v; \rho gm$	given
$\Theta_1 \vdash (\Delta \vdash^n s : \tau = \llbracket e \rrbracket_{\phi_v}) \dashv \Theta_1, s : (\Delta, \tau, n + 1)$	inversion (rule C-PGM-SPDEF), rule C-SPDEF
$\Theta_1; \Delta \vdash^n \llbracket e \rrbracket_{\phi_v} : \text{Code } \tau$	inversion (rule C-SPDEF)
$\Theta_1; \Delta \vdash^{n+1} \llbracket \phi_v \rrbracket e : \tau$	by substitution
$\Theta_1, s : (\Delta, \tau, n + 1) \vdash \rho gm$	
$\Theta_1 \vdash \rho gm[s \mapsto \llbracket \phi_v \rrbracket e]$	by substitution

• Case

CE-PGM-EXPR

$$\frac{e \longrightarrow e'}{e : \tau \longrightarrow e' : \tau}$$

$\Theta_1 \vdash e : \tau$	given
$\Theta_1; \bullet^0 e : \tau$	inversion (rule C-PGM-EXPR)
$\Theta_1; \bullet^0 e' : \tau$	Part 1
$\Theta_1 \vdash e' : \tau$	rule C-PGM-EXPR

□

F PROOFS FOR ELABORATION

Definition F.1 (ϕ^Γ and ϕ^Θ).

$$\frac{\bullet^\Gamma}{(\phi, \Delta \vdash^n s : \tau = e)^\Gamma} = \bullet^\Gamma, s : (\Delta, \tau, n + 1) \quad \frac{\bullet^\Theta}{(\phi, \Delta \vdash^n s : \tau = e)^\Theta} = \bullet^\Theta, s : (\Delta, \tau, n + 1)$$

Lemma 5.2 (Level Correctness of ϕ). *If $\Theta; \Gamma \vdash^n e : \tau \rightsquigarrow e \mid \phi$, then $\phi \dot{<} n$.*

PROOF. By induction on typing. Most cases follow straightforwardly from I.H., the only two interesting cases are:

• Case **S-QUOTE**

$$\Theta; \Gamma \vdash^{n+1} e : \tau \rightsquigarrow e \mid \phi$$

$$\Theta; \Gamma \vdash^n \llbracket e \rrbracket : \text{Code } \tau \rightsquigarrow \llbracket e \rrbracket_{\phi.n} \mid \llbracket \phi \rrbracket^n$$

$\phi \dot{<} n + 1$	I.H.
$\llbracket \phi \rrbracket^n \dot{<} n$	by definition

• Case

S-SPLICE

$$\Theta; \Gamma \vdash^{n-1} e : \text{Code } \tau \rightsquigarrow e \mid \phi \quad \Gamma \vdash \tau \rightsquigarrow \tau' \quad \text{fresh } s$$

$$\Theta; \Gamma \vdash^n \$e : \tau \rightsquigarrow s \mid \phi, (\bullet^{n-1} s : \tau' = e)$$

$\phi \dot{<} n - 1$	I.H.
$\phi \dot{<} n$	follows
$\phi, (\bullet^{n-1} s : \tau' = e) \dot{<} n$	by definition

The cases for constraint solving are exactly the same.

□

Lemma 5.3 (Context Injection). *If $\Theta; \Delta_1, \Delta_2 \vdash \phi_1$, and $\phi_1 \dot{<} \Delta_2$, and $\phi_1 \dot{+} \Delta_2 \rightsquigarrow \phi_2$, then $\Theta; \Delta_1 \vdash \phi_2$.*

PROOF. By induction on ϕ .

- $\phi = \bullet$. Then $\Theta; \Delta_1 \vdash \bullet$ by rule **C-S-EMPTY**.
- $\phi = \phi_1, \Delta \vdash^{\text{I}} s : \tau = e$.

$(\phi_1, \Delta \vdash^{\text{I}} s : \tau = e) \dot{+} \Delta_2 \rightsquigarrow \phi_2, (\Delta_2, \Delta \vdash^{\text{I}} s : \tau = e)$	given
$\phi_1 \dot{+} \Delta_2 \rightsquigarrow \phi_2$	inversion (rule S-INJ-CONS)
$\Theta; \Delta_1, \Delta_2 \vdash \phi_1, \Delta \vdash^{\text{I}} s : \tau = e$	given
$\Theta; \Delta_1, \Delta_2 \vdash \phi_1$	inversion (rule C-S-CONS)
$\Delta \dot{>} n$	above
$\Theta; \Delta_1, \Delta_2, \Delta \vdash^{\text{I}} e : \text{Code } \tau$	above
$\Theta; \Delta_1 \vdash \phi_2$	I.H.
$\Delta_2 \dot{>} \phi$	given
$\Delta_2, \Delta \dot{>} n$	follows
$\Theta; \Delta_1 \vdash \phi_2, (\Delta_2, \Delta \vdash^{\text{I}} s : \tau = e)$	rule C-S-CONS

□

Lemma F.2 (ϕ^Γ to ϕ^Θ). *If $\Theta; \phi^\Gamma, \Gamma \vdash^{\text{I}} e : \tau$, then $\Theta, \phi^\Theta; \Gamma \vdash^{\text{I}} e : \tau$. Similarly, if $\Theta; \phi^\Gamma, \Gamma \vdash \phi'$, then $\Theta, \phi^\Theta; \Gamma \vdash \phi'$.*

PROOF. by induction on typing. Most cases are straightforward. The only interesting case is

$$\frac{\text{C-SVAR} \quad s : (\Delta, \tau, n) \in \Gamma \quad \Delta \subseteq \Gamma}{\Theta; \Gamma \vdash^{\text{I}} s : \tau}$$

If $s \in \phi$, then it is now moved to Θ, ϕ^Θ , and we can apply rule **C-TOP-SVAR**; or otherwise we can still apply rule **C-SVAR**.

The left requirement is to show from $\Delta \subseteq \phi^\Gamma, \Gamma$ that $\Delta \subseteq \Gamma$. The observation here is that since Δ does not have any splice variables, so removing ϕ^Γ does not affect the subset requirement.

□

Lemma F.3 (ϕ^Γ moves to left). *If $\Theta; \Gamma_1, \Delta, \phi^\Gamma, \Gamma_2 \vdash^{\text{I}} e : \tau$, and $\Delta \dot{>} \phi$, and $\phi \dot{+} \Delta \rightsquigarrow \phi'$ then $\Theta; \Gamma_1, \phi'^\Gamma, \Delta, \Gamma_2 \vdash^{\text{I}} e : \tau$. Similarly, if $\Theta; \Gamma_1, \Delta, \phi^\Gamma, \Gamma_2 \vdash \phi_1$, and $\Delta \dot{>} \phi$, and $\phi \dot{+} \Delta \rightsquigarrow \phi'$, then $\Theta; \Gamma_1, \phi'^\Gamma, \Delta, \Gamma_2 \vdash \phi_1$.*

PROOF. By induction on typing. Most cases are straightforward. The only interesting cases are the cases for splice variables. Most importantly, we need to show that the subset constraint $\Delta \subseteq \Gamma$ in rules **C-SVAR** and **C-TOP-SVAR** is still satisfied in the modified context.

The observation here is that since Δ does not have splice variables, so moving ϕ^Γ does not affect the subset requirement.

□

Theorem F.4 (Elaboration Soundness).

- (1) If $\Theta; \Gamma \vdash^{\text{I}} e : \tau \rightsquigarrow e \mid \phi$, and $\Theta \rightsquigarrow \Theta$, and $\Gamma \rightsquigarrow \Delta$, then $\Theta; \Delta \vdash \phi$.
- (2) If $\Theta; \Gamma \Vdash C \rightsquigarrow e \mid \phi$, and $\Theta \rightsquigarrow \Theta$, and $\Gamma \rightsquigarrow \Delta$, then $\Theta; \Delta \vdash \phi$. If $\Gamma \vdash C \rightsquigarrow \tau$, then $\Theta; \Delta, \phi^\Gamma \vdash^{\text{I}} e : \tau$.
- (3) If $\Theta; \Gamma \vdash^{\text{I}} e : \tau \rightsquigarrow e \mid \phi$, and $\Theta \rightsquigarrow \Theta$, and $\Gamma \rightsquigarrow \Delta$, and $\Gamma \vdash \tau \rightsquigarrow \tau$, then $\Theta; \Delta, \phi^\Gamma \vdash^{\text{I}} e : \tau$.

- (4) If $\rho gm_1; \Theta_1 \vdash \mathcal{D} \dashv \Theta_2 \rightsquigarrow \rho gm_2$, and $\Theta_1 \rightsquigarrow \Theta_1$, and $\Theta_2 \rightsquigarrow \Theta_2$, and $\Theta_2 \vdash \rho gm_1$, then $\Theta_1 \vdash \rho gm_2$.
- (5) If $\rho gm_1; \Theta_1 \vdash C \dashv \Theta_2 \rightsquigarrow \rho gm_2$, and $\Theta_1 \rightsquigarrow \Theta_1$, and $\Theta_2 \rightsquigarrow \Theta_2$, and $\Theta_2 \vdash \rho gm_1$, then $\Theta_1 \vdash \rho gm_2$.
- (6) If $\rho gm_1; \Theta_1 \vdash I \dashv \Theta_2 \rightsquigarrow \rho gm_2$, and $\Theta_1 \rightsquigarrow \Theta_1$, and $\Theta_2 \rightsquigarrow \Theta_2$, and $\Theta_2 \vdash \rho gm_1$, then $\Theta_1 \vdash \rho gm_2$.
- (7) If $\rho gm_1 \Vdash \phi \rightsquigarrow \rho gm_2$, and $\phi \leq n$, and $\Theta; \bullet \vdash \phi$, and $\Theta, \phi^\Theta \vdash \rho gm_1$, then $\Theta \vdash \rho gm_2$.
- (8) If $\Theta \vdash \rho gm : \sigma \rightsquigarrow \rho gm$, and $\Theta \rightsquigarrow \Theta$, then $\Theta \vdash \rho gm$.

PROOF. By induction on typing.

Part 1 • Case rule **S-LIT**. Follows trivially from rule **C-S-EMPTY**.

- Case rule **S-VAR**. Follows trivially from rule **C-S-EMPTY**.
- Case rule **S-KVAR**. Follows trivially from rule **C-S-EMPTY**.
- Case

$$\frac{\Theta; \Gamma, x : (\tau_1, n) \Vdash e : \tau_2 \rightsquigarrow e \mid \phi_1 \quad \Gamma \vdash \tau_1 \rightsquigarrow \tau'_1 \quad \phi_1 \dashv\vdash x : (\tau'_1, n) \rightsquigarrow \phi_2}{\Theta; \Gamma \Vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2 \rightsquigarrow \lambda x : \tau'_1. e \mid \phi_2}$$

$$\left. \begin{array}{l} \Theta; \Delta, x : (\tau'_1, n) \vdash \phi_1 \\ \phi_1 \leq n \\ \phi_1 \dashv\vdash x : (\tau'_1, n) \rightsquigarrow \phi_2 \\ \Theta; \Delta \vdash \phi_2 \end{array} \right| \begin{array}{l} \text{I.H.} \\ \text{Lemma 5.2} \\ \text{given} \\ \text{Lemma 5.3} \end{array}$$

- Case

$$\frac{\text{S-APP} \quad \Theta; \Gamma \Vdash e_1 : \tau_1 \rightarrow \tau_2 \rightsquigarrow e_1 \mid \phi_1 \quad \Theta; \Gamma \Vdash e_2 : \tau_1 \rightsquigarrow e_2 \mid \phi_2}{\Theta; \Gamma \Vdash e_1 e_2 : \tau_2 \rightsquigarrow e_1 e_2 \mid \phi_1, \phi_2}$$

$$\left. \begin{array}{l} \Theta; \Delta \vdash \phi_1 \\ \Theta; \Delta \vdash \phi_2 \\ \Theta; \Delta \vdash \phi_1, \phi_2 \end{array} \right| \begin{array}{l} \text{I.H.} \\ \text{I.H.} \end{array}$$

- Case

$$\frac{\text{S-TABS} \quad \Theta; \Gamma, a \Vdash e : \sigma \rightsquigarrow e \mid \phi_1 \quad \phi_1 \dashv\vdash a \rightsquigarrow \phi_2}{\Theta; \Gamma \Vdash e : \forall a. \sigma \rightsquigarrow \Lambda a. e \mid \phi_2}$$

$$\left. \begin{array}{l} \Theta; \Delta, a \vdash \phi_1 \\ \phi_1 \leq n \\ \phi_1 \dashv\vdash a \rightsquigarrow \phi_2 \\ \Theta; \Delta \vdash \phi_2 \end{array} \right| \begin{array}{l} \text{I.H.} \\ \text{Lemma 5.2} \\ \text{given} \\ \text{Lemma 5.3} \end{array}$$

- Case

$$\frac{\text{S-TAPP} \quad \Theta; \Gamma \Vdash e : \forall a. \sigma \rightsquigarrow e \mid \phi \quad \Gamma \vdash \tau \rightsquigarrow \tau'}{\Theta; \Gamma \Vdash e : \sigma[a \mapsto \tau] \rightsquigarrow e \tau' \mid \phi}$$

$$\Theta; \Delta \vdash \phi \mid \text{I.H.}$$

• Case

S-QUOTE

$$\frac{\Theta; \Gamma \vdash^{n+1} e : \tau \rightsquigarrow e \mid \phi}{\Theta; \Gamma \vdash^n \llbracket e \rrbracket : \text{Code } \tau \rightsquigarrow \llbracket e \rrbracket_{\phi.n} \mid \llbracket \phi \rrbracket^n}$$

$$\left. \begin{array}{l} \Theta; \Delta \vdash \phi \\ \Theta; \Delta \vdash \llbracket \phi \rrbracket^n \end{array} \right| \text{I.H.}$$

• Case

S-SPLICE

$$\frac{\Theta; \Gamma \vdash^{n-1} e : \text{Code } \tau \rightsquigarrow e \mid \phi \quad \Gamma \vdash \tau \rightsquigarrow \tau' \quad \text{fresh } s}{\Theta; \Gamma \vdash^n \$e : \tau \rightsquigarrow s \mid \phi, (\bullet \vdash^{n-1} s : \tau' = e)}$$

$$\left. \begin{array}{l} \Theta; \Delta \vdash \phi \\ \Theta; \Delta \vdash^{n-1} e : \text{Code } \tau' \\ \Theta; \Delta \vdash \phi, (\bullet \vdash^{n-1} s : \tau' = e) \end{array} \right| \begin{array}{l} \text{I.H.} \\ \text{Part 3} \\ \text{rule C-S-CONS} \end{array}$$

• Case

S-CABS

$$\frac{\Theta; \Gamma, \text{ev} : (C, n) \vdash^n e : \rho \rightsquigarrow e \mid \phi_1 \quad \Gamma \vdash C \rightsquigarrow \tau \quad \phi_1 ++ \text{ev} : (\tau, n) \rightsquigarrow \phi_2 \quad \text{fresh } \text{ev}}{\Theta; \Gamma \vdash^n e : C \Rightarrow \rho \rightsquigarrow \lambda \text{ev} : \tau.e \mid \phi_2}$$

$$\left. \begin{array}{l} \Theta; \Delta, \text{ev} : (\tau, n) \vdash \phi_1 \\ \phi_1 \prec n \\ \phi_1 ++ \text{ev} : (\tau_1, n) \rightsquigarrow \phi_2 \\ \Theta; \Delta \vdash \phi_2 \end{array} \right| \begin{array}{l} \text{I.H.} \\ \text{Lemma 5.2} \\ \text{given} \\ \text{Lemma 5.3} \end{array}$$

• Case

S-CAPP

$$\frac{\Theta; \Gamma \vdash^n e : C \Rightarrow \rho \rightsquigarrow e_1 \mid \phi_1 \quad \Theta; \Gamma \vdash^n C \rightsquigarrow e_2 \mid \phi_2}{\Theta; \Gamma \vdash^n e : \rho \rightsquigarrow e_1 e_2 \mid \phi_1, \phi_2}$$

$$\left. \begin{array}{l} \Theta; \Delta \vdash \phi_1 \\ \Theta; \Delta \vdash \phi_2 \\ \Theta; \Delta \vdash \phi_1, \phi_2 \end{array} \right| \begin{array}{l} \text{I.H.} \\ \text{Part 2} \end{array}$$

Part 2 • Case

S-SOLVE-GLOBAL

$$\frac{\text{ev} : \forall a. \overline{a.C_i}^i \Rightarrow C \in \Theta \quad \Gamma \vdash \tau \rightsquigarrow \tau' \quad \Theta; \Gamma \vdash^n C_i[a \mapsto \tau] \rightsquigarrow e_i \mid \phi_i^i}{\Theta; \Gamma \vdash^n C[a \mapsto \tau] \rightsquigarrow \text{ev } \tau' \overline{e_i}^i \mid \overline{\phi_i}^i}$$

$$\left. \begin{array}{l} \Theta; \Delta \vdash \overline{\phi_i}^i \\ \Gamma \vdash C_i \rightsquigarrow \tau_i \\ \Gamma \vdash C \rightsquigarrow \tau'' \\ \text{ev} : \forall a. \overline{a.C_i}^i \Rightarrow C \in \Theta \end{array} \right| \begin{array}{l} \text{I.H.} \\ \text{let} \\ \text{let} \\ \text{given} \end{array}$$

$$\begin{array}{c}
 2108 \quad \Theta; \Delta, \overline{\phi_i^\Gamma}^i \Vdash^{\mathbf{n}} ev : \forall a. \overline{\tau_i^i} \rightarrow \tau'' \quad \left| \text{rule } \mathbf{C-KVAR} \right. \\
 2109 \quad \hline \\
 2110 \quad \Theta; \Delta, \overline{\phi_i^\Gamma}^i \Vdash^{\mathbf{n}} e_i : \tau_i[a \mapsto \tau'] \quad \left| \text{I.H.} \right. \\
 2111 \quad \Theta; \Delta, \overline{\phi_i^\Gamma}^i \Vdash^{\mathbf{n}} ev \tau' \overline{e_i^i} : \tau''[a \mapsto \tau'] \quad \left| \text{rules } \mathbf{C-TAPP} \text{ and } \mathbf{C-APP} \right. \\
 2112
 \end{array}$$

• Case

$$\begin{array}{c}
 \text{S-SOLVE-LOCAL} \\
 \hline
 ev : (C, n) \in \Gamma \\
 \hline
 \Theta; \Gamma \Vdash^{\mathbf{n}} C \rightsquigarrow ev \mid \bullet
 \end{array}$$

$$\begin{array}{c}
 2118 \quad \Theta; \Delta \vdash \bullet \quad \left| \text{rule } \mathbf{C-S-EMPTY} \right. \\
 2119 \quad \Gamma \vdash C \rightsquigarrow \tau \quad \left| \text{let} \right. \\
 2120 \quad ev : (C, n) \in \Gamma \quad \left| \text{given} \right. \\
 2121 \quad \Theta; \Delta \Vdash^{\mathbf{n}} ev : \tau \quad \left| \text{rule } \mathbf{C-VAR} \right. \\
 2122
 \end{array}$$

• Case

$$\begin{array}{c}
 \text{S-SOLVE-DECR} \\
 \hline
 \Theta; \Gamma \Vdash^{n+1} C \rightsquigarrow e \mid \phi \\
 \hline
 \Theta; \Gamma \Vdash^{\mathbf{n}} \text{CodeC } C \rightsquigarrow \llbracket e \rrbracket_{\phi.n} \mid \lfloor \phi \rfloor^{\mathbf{n}}
 \end{array}$$

$$\begin{array}{c}
 2127 \quad \Theta; \Gamma \Vdash^{n+1} e : \tau \rightsquigarrow e \mid \phi \quad \left| \text{given} \right. \\
 2128 \quad \Theta; \Delta, \phi^\Gamma \Vdash^{n+1} e : \tau \quad \left| \text{I.H.} \right. \\
 2129 \quad \Theta; \Delta \vdash \phi \quad \left| \text{I.H.} \right. \\
 2130 \quad \Theta; \Delta \vdash \lfloor \phi \rfloor^{\mathbf{n}} \quad \left| \text{follows} \right. \\
 2131 \quad \Theta; \Delta \Vdash^{\mathbf{n}} \phi.n \quad \left| \text{follows} \right. \\
 2132 \quad \Theta; \Delta, (\lfloor \phi \rfloor^{\mathbf{n}})^\Gamma \Vdash^{\mathbf{n}} \phi.n \quad \left| \text{weakening} \right. \\
 2133 \quad \phi \prec n + 1 \quad \left| \text{Lemma 5.2} \right. \\
 2134 \quad \phi = \phi.n, \lfloor \phi \rfloor^{\mathbf{n}} \quad \left| \text{follows} \right. \\
 2135 \quad \Theta; \Delta, (\lfloor \phi \rfloor^{\mathbf{n}})^\Gamma \Vdash^{\mathbf{n}} \llbracket e \rrbracket_{\phi.n} : \text{Code } \tau \quad \left| \text{rule } \mathbf{C-QUOTE} \right. \\
 2136
 \end{array}$$

• Case

$$\begin{array}{c}
 \text{S-SOLVE-INCR} \\
 \hline
 \Theta; \Gamma \Vdash^{n-1} \text{CodeC } C \rightsquigarrow e \mid \phi \quad \Gamma \vdash C \rightsquigarrow \tau \quad \text{fresh } s \\
 \hline
 \Theta; \Gamma \Vdash^{\mathbf{n}} C \rightsquigarrow s \mid \phi, (\bullet \Vdash^{n-1} s : \tau = e)
 \end{array}$$

$$\Theta; \Delta, \phi^\Gamma, s : (\bullet, \tau, n) \Vdash^{\mathbf{n}} s : \tau \mid \text{rule } \mathbf{C-SVAR}$$

Part 3 • Case for rule **S-LIT** follows directly from rule **C-LIT**.

• Case

$$\begin{array}{c}
 \text{S-VAR} \\
 \hline
 x : (\tau, n) \in \Gamma \\
 \hline
 \Theta; \Gamma \Vdash^{\mathbf{n}} x : \tau \rightsquigarrow x \mid \bullet
 \end{array}$$

$$\begin{array}{c}
 2150 \quad x : (\tau, n) \in \Gamma \quad \left| \text{given} \right. \\
 2151 \quad x : (\tau', n) \in \Delta \quad \left| \text{follows} \right. \\
 2152 \quad \Theta; \Delta \Vdash^{\mathbf{n}} x : \tau' \quad \left| \text{rule } \mathbf{C-VAR} \right. \\
 2153
 \end{array}$$

• Case

$$\begin{array}{c} \text{S-KVAR} \\ \hline k : \sigma \in \Theta \\ \hline \Theta; \Gamma \Vdash k : \sigma \rightsquigarrow k \mid \bullet \end{array}$$

$k : \sigma \in \Theta$ | given
 $k : \tau \in \Theta$ | follows
 $\Theta; \Delta \Vdash k : \tau$ | rule **C-KVAR**

• Case

$$\begin{array}{c} \text{S-ABS} \\ \hline \Theta; \Gamma, x : (\tau_1, n) \Vdash e : \tau_2 \rightsquigarrow e \mid \phi_1 \quad \Gamma \vdash \tau_1 \rightsquigarrow \tau'_1 \quad \phi_1 \Vdash x : (\tau'_1, n) \rightsquigarrow \phi_2 \\ \hline \Theta; \Gamma \Vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2 \rightsquigarrow \lambda x : \tau'_1. e \mid \phi_2 \end{array}$$

$\Theta; \Gamma, x : (\tau_1, n) \Vdash e : \tau_2 \rightsquigarrow e \mid \phi_1$ | given
 $\Theta; \Delta, x : (\tau'_1, n), \phi_1^\Gamma \Vdash e : \tau'_2$ | I.H.
 $\Theta; \Delta, \phi_1^\Gamma, x : (\tau'_1, n) \Vdash e : \tau'_2$ | context reorder
 $\Theta; \Delta, \phi_2^\Gamma, x : (\tau'_1, n) \Vdash e : \tau'_2$ | strengthening
 $\Theta; \Delta, \phi_2^\Gamma \Vdash \lambda x : \tau'_1. e : \tau'_1 \rightarrow \tau'_2$ | rule **C-ABS**

• Case

$$\begin{array}{c} \text{S-APP} \\ \hline \Theta; \Gamma \Vdash e_1 : \tau_1 \rightarrow \tau_2 \rightsquigarrow e_1 \mid \phi_1 \quad \Theta; \Gamma \Vdash e_2 : \tau_1 \rightsquigarrow e_2 \mid \phi_2 \\ \hline \Theta; \Gamma \Vdash e_1 e_2 : \tau_2 \rightsquigarrow e_1 e_2 \mid \phi_1, \phi_2 \end{array}$$

$\Theta; \Gamma \Vdash e_1 : \tau_1 \rightarrow \tau_2 \rightsquigarrow e_1 \mid \phi_1$ | given
 $\Theta; \Delta, \phi_1^\Gamma \Vdash e_1 : \tau'_1 \rightarrow \tau'_2$ | I.H.
 $\Theta; \Delta, \phi_1^\Gamma, \phi_2^\Gamma \Vdash e_1 : \tau'_1 \rightarrow \tau'_2$ | weakening
 $\Theta; \Gamma \Vdash e_2 : \tau_1 \rightsquigarrow e_2 \mid \phi_2$ | given
 $\Theta; \Delta, \phi_2^\Gamma \Vdash e_2 : \tau'_1$ | I.H.
 $\Theta; \Delta, \phi_1^\Gamma, \phi_2^\Gamma \Vdash e_2 : \tau'_1$ | weakening
 $\Theta; \Delta, (\phi_1, \phi_2)^\Gamma \Vdash e_1 e_2 : \tau'_2$ | rule **C-APP**

• Case

$$\begin{array}{c} \text{S-TABS} \\ \hline \Theta; \Gamma, a \Vdash e : \sigma \rightsquigarrow e \mid \phi_1 \quad \phi_1 \Vdash a \rightsquigarrow \phi_2 \\ \hline \Theta; \Gamma \Vdash e : \forall a. \sigma \rightsquigarrow \Lambda a. e \mid \phi_2 \end{array}$$

$\Theta; \Gamma, a \Vdash e : \sigma \rightsquigarrow e \mid \phi_1$ | given
 $\Theta; \Delta, a, \phi_1^\Gamma \Vdash e : \tau$ | I.H.
 $\Theta; \Delta, \phi_2^\Gamma, a \Vdash e : \tau$ | Lemma **F.3**
 $\Theta; \Delta, \phi_2^\Gamma \Vdash \Lambda a. e : \forall a. \tau$ | rule **C-TABS**

• Case

$$\begin{array}{c} \text{S-TAPP} \\ \hline \Theta; \Gamma \Vdash e : \forall a. \sigma \rightsquigarrow e \mid \phi \quad \Gamma \vdash \tau \rightsquigarrow \tau' \\ \hline \Theta; \Gamma \Vdash e : \sigma[a \mapsto \tau] \rightsquigarrow e \tau' \mid \phi \end{array}$$

2206	$\Theta; \Gamma \Vdash e : \forall a. \sigma \rightsquigarrow e \mid \phi$	given
2207	$\Theta; \Delta, \phi^\Gamma \Vdash e : \forall a. \tau_1$	I.H.
2208	$\Theta; \Delta, \phi^\Gamma \Vdash e \tau : \tau_1[a \mapsto \tau]$	rule C-TAPP
2209		
2210	• Case	
2211		S-QUOTE
2212	$\Theta; \Gamma \Vdash^{n+1} e : \tau \rightsquigarrow e \mid \phi$	
2213	$\Theta; \Gamma \Vdash \llbracket e \rrbracket : \text{Code } \tau \rightsquigarrow \llbracket e \rrbracket_{\phi.n} \mid \llbracket \phi \rrbracket^n$	
2214		
2215	$\Theta; \Gamma \Vdash^{n+1} e : \tau \rightsquigarrow e \mid \phi$	given
2216	$\Theta; \Delta, \phi^\Gamma \Vdash^{n+1} e : \tau$	I.H.
2217	$\Theta; \Delta \vdash \phi$	Part 1
2218	$\Theta; \Delta \Vdash \phi.n$	follows
2219	$\Theta; \Delta, (\llbracket \phi \rrbracket^n)^\Gamma \Vdash \phi.n$	weakening
2220	$\phi \prec n + 1$	Lemma 5.2
2221	$\phi = \phi.n, \llbracket \phi \rrbracket^n$	follows
2222	$\Theta; \Delta, (\llbracket \phi \rrbracket^n)^\Gamma \Vdash \llbracket e \rrbracket_{\phi.n} : \text{Code } \tau$	rule C-QUOTE
2223		
2224	• Case	
2225		S-SPLICE
2226	$\Theta; \Gamma \Vdash^{n-1} e : \text{Code } \tau \rightsquigarrow e \mid \phi \quad \Gamma \vdash \tau \rightsquigarrow \tau' \quad \text{fresh } s$	
2227	$\Theta; \Gamma \Vdash \$e : \tau \rightsquigarrow s \mid \phi, (\bullet \Vdash^{n-1} s : \tau' = e)$	
2228		
2229	$\Theta; \Delta, \phi^\Gamma, s : (\bullet, \tau', n) \Vdash s : \tau' \mid$	rule C-SVAR
2230		
2231	• Case	
2232		S-CABS
2233	$\Theta; \Gamma, \text{ev} : (C, n) \Vdash e : \rho \rightsquigarrow e \mid \phi_1$	
2234	$\Gamma \vdash C \rightsquigarrow \tau \quad \phi_1 \dashv\vdash \text{ev} : (\tau, n) \rightsquigarrow \phi_2 \quad \text{fresh ev}$	
2235	$\Theta; \Gamma \Vdash e : C \Rightarrow \rho \rightsquigarrow \lambda \text{ev} : \tau.e \mid \phi_2$	
2236		
2237	$\Theta; \Delta, \text{ev} : (\tau, n), \phi_1^\Gamma \Vdash e : \tau'$	I.H.
2238	$\Theta; \Delta, \phi_2^\Gamma, \text{ev} : (\tau, n) \Vdash e : \tau'$	Lemma F.3
2239	$\Theta; \Delta, \phi_2^\Gamma \Vdash \lambda \text{ev} : \tau.e : \tau \rightarrow \tau'$	rule C-ABS
2240		
2241	• Case	
2242		S-CAPP
2243	$\Theta; \Gamma \Vdash e : C \Rightarrow \rho \rightsquigarrow e_1 \mid \phi_1 \quad \Theta; \Gamma \Vdash C \rightsquigarrow e_2 \mid \phi_2$	
2244	$\Theta; \Gamma \Vdash e : \rho \rightsquigarrow e_1 e_2 \mid \phi_1, \phi_2$	
2245		
2246	$\Theta; \Delta, \phi_1^\Gamma \Vdash e_1 : \tau_1 \rightarrow \tau_2$	I.H.
2247	$\Theta; \Delta, (\phi_1, \phi_2)^\Gamma \Vdash e : \tau_1 \rightarrow \tau_2$	weakening
2248	$\Theta; \Delta, \phi_2^\Gamma \Vdash e_2 : \tau_1$	Part 2
2249	$\Theta; \Delta, (\phi_1, \phi_2)^\Gamma \Vdash e_2 : \tau_1$	weakening
2250	$\Theta; \Delta, (\phi_1, \phi_2)^\Gamma \Vdash e_1 e_2 : \tau_2$	rule C-APP
2251		
2252		

Part 4 Case

2255	S-DEF	
2256	$\Theta; \bullet \vdash^0 e : \sigma \rightsquigarrow e \mid \phi$	$\bullet \vdash \sigma \rightsquigarrow \tau$
2257		$\text{def } k : \tau = e; \rho gm_1 \vdash^{-1} \phi \rightsquigarrow \rho gm_2$
2258	$\rho gm_1; \Theta \vdash k = e \dashv \Theta, k : \sigma \rightsquigarrow \rho gm_2$	
2259	$\Theta, k : \tau \vdash \rho gm_1$	given
2260	$\Theta; \bullet \vdash \phi$	Part 1
2261	$\Theta; \phi \vdash^{\Gamma} \bullet \vdash^0 e : \tau$	Part 3
2262	$\Theta, \phi^{\Theta}; \bullet \vdash^0 e : \tau$	Lemma F.2
2263	$\Theta, \phi^{\Theta} \vdash k : \tau = e \dashv \Theta, \phi^{\Theta}, k : \tau$	rule C-DEF
2264	$\Theta, \phi^{\Theta}, k : \tau \vdash \rho gm_1$	weakening
2265	$\Theta, \phi^{\Theta} \vdash \text{def } k : \tau = e; \rho gm_1$	rule C-PGM-DEF
2266	$\phi \prec 0$	Lemma 5.2
2267	$\Theta \vdash \rho gm_2$	Part 7

Part 5 Case

S-CLS

2269		$a \vdash \rho \rightsquigarrow \tau$
2270	$\rho gm; \Theta \vdash \text{TC } a \text{ where } \{k : \rho\} \dashv \Theta, k : \forall a. \text{TC } a \Rightarrow \rho \rightsquigarrow \text{def } k : \forall a. \tau \rightarrow \tau = \Lambda a. \lambda x : \tau. x; \rho gm$	
2271		
2272		
2273		
2274	$\bullet \vdash \forall a. \text{TC } a \Rightarrow \rho \rightsquigarrow \forall a. \tau \rightarrow \tau$	rules S-K-FORALL, S-K-CARROW, and S-K-TC
2275	$\Theta, k : \forall a. \tau \rightarrow \tau \vdash \rho gm$	given
2276	$\Theta; \bullet \vdash^0 \Lambda a. \lambda x : \tau. x : \forall a. \tau \rightarrow \tau$	rules C-TABS, C-ABS, and C-VAR
2277	$\Theta \vdash k : \forall a. \tau \rightarrow \tau = \Lambda a. \lambda x : \tau. x \dashv \Theta, k : \forall a. \tau \rightarrow \tau$	rule C-DEF
2278	$\Theta \vdash \text{def } k : \forall a. \tau \rightarrow \tau = \Lambda a. \lambda x : \tau. x; \rho gm$	rule C-PGM-DEF
2279		

Part 6 Case

S-INST

2280		TC a where {k : ρ}
2281		
2282	$\overline{b_j^j} = \text{ftv}(\tau)$	$\overline{b_j^j} \vdash C_i \rightsquigarrow \tau_i^i$
2283	$\Theta; \overline{b_j^j}, \overline{ev_i : (C_i, 0)}^i \vdash^0 e : \rho[a \mapsto \tau] \rightsquigarrow e \mid \phi_1$	
2284	$\bullet \vdash \rho[a \mapsto \tau] \rightsquigarrow \tau$	$\text{fresh } \overline{ev_i}^i$
2285	$\phi_1 \dashv (\overline{b_j^j}, \overline{ev_i : (\tau_i, 0)}^i) \rightsquigarrow \phi_2$	
2286	$\text{def } ev : \forall \overline{b_j^j}. \overline{\tau_i^i} \rightarrow \tau = \Lambda \overline{b_j^j}. \lambda \overline{ev_i : \tau_i^i}. e; \rho gm_1 \vdash^{-1} \phi_2 \rightsquigarrow \rho gm_2$	fresh ev
2287	$\rho gm_1; \Theta \vdash \overline{C_i^i} \Rightarrow \text{TC } \tau \text{ where } \{k = e\} \dashv \Theta, ev : \forall \overline{b_j^j}. \overline{C_i^i} \Rightarrow \text{TC } \tau \rightsquigarrow \rho gm_2$	
2288		
2289		
2290		
2291	$\Theta, ev : \forall \overline{b_j^j}. \overline{\tau_i^i} \rightarrow \tau \vdash \rho gm_1$	given
2292	$\Theta; \overline{b_j^j}, \overline{ev_i : (\tau_i, 0)}^i \vdash \phi_1$	Part 1
2293	$\phi_1 \prec 0$	Lemma 5.2
2294	$\Theta; \bullet \vdash \phi_2$	Lemma 5.3
2295	$\Theta; \overline{b_j^j}, \overline{ev_i : (\tau_i, 0)}^i, \phi_1^{\Gamma} \vdash^0 e : \tau$	Part 3
2296	$\phi_2 \prec 0$	by definition
2297	$\Theta; \phi_2^{\Gamma}, \overline{b_j^j}, \overline{ev_i : (\tau_i, 0)}^i \vdash^0 e : \tau$	Lemma F.3
2298	$\Theta, \phi_2^{\Theta}; \overline{b_j^j}, \overline{ev_i : (\tau_i, 0)}^i \vdash^0 e : \tau$	Lemma F.2
2299	$\Theta, \phi_2^{\Theta}; \bullet \vdash^0 \Lambda \overline{b_j^j}. \lambda \overline{ev_i : \tau_i^i}. e : \forall \overline{b_j^j}. \overline{\tau_i^i} \rightarrow \tau$	rules C-TABS and C-ABS
2300	$\Theta, \phi_2^{\Theta} \vdash ev : \forall \overline{b_j^j}. \overline{\tau_i^i} \rightarrow \tau = \Lambda \overline{b_j^j}. \lambda \overline{ev_i : \tau_i^i}. e \dashv \Theta, \phi_2^{\Theta}, ev : \forall \overline{b_j^j}. \overline{\tau_i^i} \rightarrow \tau$	rule C-DEF
2301		
2302		
2303		

2304	$\Theta, \phi_2^\Theta, ev : \forall \bar{b}_j^j . \bar{\tau}_i^i \rightarrow \tau \vdash \rho gm_1$	weakening rule C-PGM-DEF Part 7
2305	$\Theta, \phi_2^\Theta \vdash \text{def } ev : \forall \bar{b}_j^j . \bar{\tau}_i^i \rightarrow \tau = \Lambda \bar{b}_j^j . \lambda \overline{ev}_i : \bar{\tau}_i^i . e ; \rho gm_1$	
2306	$\Theta \vdash \rho gm_2$	
2307		

Part 7 • The case for rule **S-CLAP-EMPTY** holds trivially.

• Case

$$\frac{\text{S-CLAP-REC} \quad \text{spdef } \phi.n; \rho gm_1 \vdash^{n-1} \lfloor \phi \rfloor^n \rightsquigarrow \rho gm_2}{\rho gm_1 \vdash^n \phi \rightsquigarrow \rho gm_2}$$

2313	$\Theta; \bullet \vdash \phi$	given
2314	$\Theta; \bullet \vdash \lfloor \phi \rfloor^n$	follows
2315	$\Theta; \bullet \vdash^n \phi.n$	follows
2316	$\phi \leq n$	given
2317	$\lfloor \phi \rfloor^n \leq n - 1$	follows
2318	$\Theta, \phi^\Theta \vdash \rho gm_1$	given
2319	$\phi = \lfloor \phi \rfloor^n, \phi.n$	$\phi \leq n$
2320	$\Theta, (\lfloor \phi \rfloor^n)^\Theta, (\phi.n)^\Theta \vdash \rho gm_1$	follows
2321	$\Theta, (\lfloor \phi \rfloor^n)^\Theta \vdash \text{spdef } \phi.n; \rho gm_1$	rule C-PGM-SPDEF
2322	$\Theta \vdash \rho gm_2$	I.H.
2323		

Part 8 • Case

$$\frac{\text{S-PGM-DEF} \quad \rho gm_1; \Theta_1 \vdash \mathcal{D} \dashv \Theta_2 \rightsquigarrow \rho gm_2 \quad \Theta_2 \vdash \text{pgm} : \sigma \rightsquigarrow \rho gm_1}{\Theta_1 \vdash \text{def } \mathcal{D}; \text{pgm} : \sigma \rightsquigarrow \rho gm_2}$$

2329	$\Theta_2 \vdash \rho gm_1$	given
2330	$\Theta_1 \vdash \rho gm_2$	Part 4

• Case

$$\frac{\text{S-PGM-CLS} \quad \rho gm_1; \Theta_1 \vdash C \dashv \Theta_2 \rightsquigarrow \rho gm_2 \quad \Theta_2 \vdash \text{pgm} : \sigma \rightsquigarrow \rho gm_1}{\Theta_1 \vdash \text{class } C; \text{pgm} : \sigma \rightsquigarrow \rho gm_2}$$

2336	$\Theta_2 \vdash \rho gm_1$	given
2337	$\Theta_1 \vdash \rho gm_2$	Part 5

• Case

$$\frac{\text{S-PGM-INST} \quad \rho gm_1; \Theta_1 \vdash I \dashv \Theta_2 \rightsquigarrow \rho gm_2 \quad \Theta_2 \vdash \text{pgm} : \sigma \rightsquigarrow \rho gm_1}{\Theta_1 \vdash \text{inst } I; \text{pgm} : \sigma \rightsquigarrow \rho gm_2}$$

2344	$\Theta_2 \vdash \rho gm_1$	given
2345	$\Theta_1 \vdash \rho gm_2$	Part 6

• Case

$$\frac{\text{S-PGM-EXPR} \quad \Theta; \bullet \vdash^0 e : \sigma \rightsquigarrow e \mid \phi \quad \bullet \vdash \sigma \rightsquigarrow \tau \quad e : \tau \vdash^{-1} \phi \rightsquigarrow \rho gm}{\Theta \vdash e : \sigma \rightsquigarrow \rho gm}$$

2353	$\Theta; \bullet \vdash \phi$	Part 1
2354	$\phi \dot{<} 0$	Lemma 5.2
2355	$\Theta; \phi^\Gamma \vdash^0 e : \tau$	Part 3
2356	$\Theta, \phi^\Theta; \bullet \vdash^0 e : \tau$	Lemma F.2
2357	$\Theta, \phi^\Theta \vdash e : \tau$	Rule C-PGM-EXPR
2358	$\Theta \vdash \rho gm$	Part 7

□

G OVERVIEW OF AXIOMATIC SEMANTICS

In this section we outline the proofs for axiomatic semantics Appendix H includes the list of lemmas, and Appendix I presents the proofs. An overview figure that shows the relation between definitions and lemma is given in Figure 9. First, we present some definitions, and then discuss about the proofs in Appendix G.2.

G.1 Axiomatic Equivalence

We have axioms between F^\square expressions:

$$\frac{\llbracket s \rrbracket_{\Theta \vdash s : \tau = e} =_{ax} e}{\llbracket e_1 \rrbracket_{\phi_1, \Delta \vdash s : \tau = \llbracket e \rrbracket_{\phi, \phi_2}} =_{ax} \llbracket e_1[s \mapsto e] \rrbracket_{\phi_1, \phi', \phi_2}} \quad \text{where } \phi \leftrightarrow \Delta \rightsquigarrow \phi'$$

An axiomatic equivalence relation $e_1 =_{ax} e_2$ between F^\square expressions that is the contextual and equivalence closure of the axioms. In particular, we extend the axioms with

$$\boxed{e_1 =_{ax} e_2} \quad \text{(Axiomatic equality)}$$

$$\frac{}{e =_{ax} e} \quad \text{EQ-REFL} \quad \frac{e_1 =_{ax} e_2}{e_2 =_{ax} e_1} \quad \text{EQ-SYMM} \quad \frac{e_1 =_{ax} e_2 \quad e_2 =_{ax} e_3}{e_1 =_{ax} e_3} \quad \text{EQ-TRANS} \quad \frac{e_1 =_{ax} e_2 \quad \mathbb{C}_1 =_{ax} \mathbb{C}_2}{\mathbb{C}_1[e_1] =_{ax} \mathbb{C}_2[e_2]} \quad \text{EQ-CTX}$$

Similarly, the axiomatic equivalence relation $\rho gm_1 =_{ax} \rho gm_2$ axioms for F^\square programs are the contextual and equivalence closure of the following axioms:

$$\boxed{\rho gm_1 =_{ax} \rho gm_2} \quad \text{(Axiomatic equality)}$$

$$\frac{}{\rho gm_1 =_{ax} \rho gm_2} \quad \text{PEQ-DEF} \quad \frac{e_1 =_{ax} e_2 \quad \rho gm_1 =_{ax} \rho gm_2}{\text{def } k : \tau = e_1; \rho gm_1 =_{ax} \text{def } k : \tau = e_2; \rho gm_2} \quad \text{PEQ-EXPR}$$

$$\frac{e_1 =_{ax} e_2 \quad \rho gm_1 =_{ax} \rho gm_2}{\text{spdef } \Delta \vdash^n s : \tau = e_1; \rho gm_1 =_{ax} \text{spdef } \Delta \vdash^n s : \tau = e_2; \rho gm_2} \quad \text{PEQ-SPDEF} \quad \frac{e_1 =_{ax} e_2}{e_1 : \tau =_{ax} e_2 : \tau} \quad \text{PEQ-SPDEF-AX}$$

$$\frac{\phi \leftrightarrow \Delta \rightsquigarrow \phi'}{\text{spdef } \Delta \vdash^n s : \tau = \llbracket e \rrbracket_{\phi}; \rho gm =_{ax} \text{spdef } \phi'; \rho gm[s \mapsto e]} \quad \text{PEQ-SPDEF-AX}$$

Definition G.1 (Axiomatic Equivalence).

$$\begin{aligned} \Theta \vdash \rho gm_1 &\preceq_{ax} \rho gm_2 \triangleq \Theta \vdash \rho gm_1 \wedge \Theta \vdash \rho gm_2 \wedge \rho gm_1 =_{ax} \rho gm_2 \\ \Theta; \Gamma \vdash^n e_1 &\preceq_{ax} e_2 : \tau \triangleq \Theta; \Gamma \vdash^n e_1 : \tau \wedge \Theta; \Gamma \vdash^n e_2 : \tau \wedge e_1 =_{ax} e_2 \end{aligned}$$

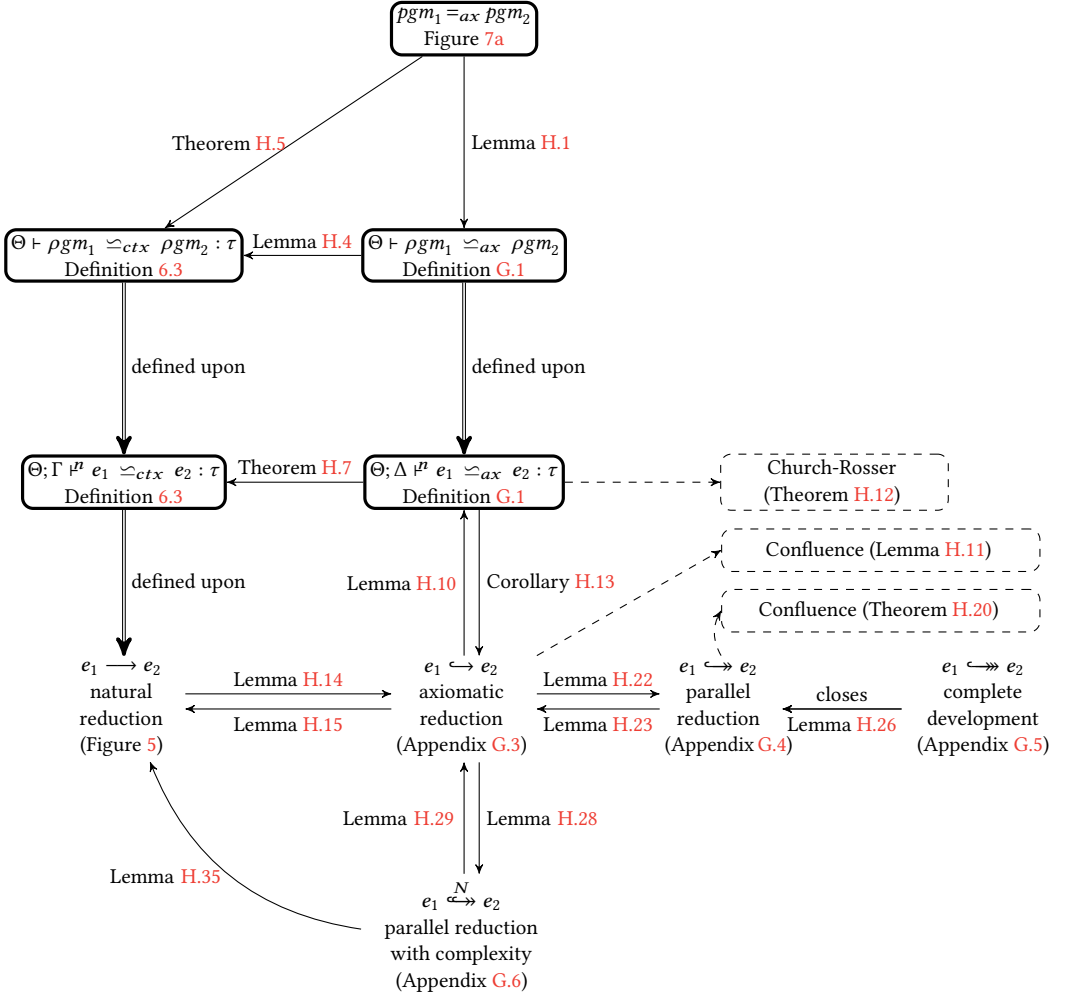


Fig. 9. Overview of the proofs

G.2 Outline

To prove our goal that $\text{source} =_{ax}$ leads to $\text{core} \simeq_{ctx}$, we need two steps:

- (1) $\text{Source} =_{ax}$ leads to $\text{core} =_{ax}$; and
- (2) $\text{Core} \simeq_{ax}$ leads to $\text{core} \simeq_{ctx}$.

The first step is by an inductive step on $\text{source} =_{ax}$. The related lemmas are given in Appendix H.1 (for programs) and Appendix H.2 (for expressions).

The second step is more involved. We first define *axiomatic reduction* (\leftrightarrow) (Appendix G.3) derived from core axiomatic equivalence. Now that we can first relate \simeq_{ax} to \leftrightarrow . This part is proved in Appendix H.3. One important property we need there is *Church-Rosser* (Theorem H.12), which is proved in Appendix H.4. The proof of Church-Rosser is based on the notion of *parallel reduction* (Appendix G.4), whose proofs are based on the notion of *complete development* (Appendix G.5). The

proofs regarding parallel reduction are given in Appendix H.5, and regarding complete development are given in Appendix H.6.

Now we can relate $\text{core} \simeq_{ax}$ to $\text{core} \simeq_{ctx}$ by relating axiomatic reduction to operational semantics. That is done via a definition of *parallel reduction with complexity* (Appendix G.6). And the related lemmas are given in Appendix H.7.

G.3 Axiomatic Reduction

From axioms we can derive a reduction semantics:

$$\boxed{e_1 \hookrightarrow e_2} \quad (\text{Axiomatic Reduction})$$

$$\begin{array}{c}
 \text{CE-AX-SPLICEQUOTE} \\
 \frac{\phi \uparrow \Delta \rightsquigarrow \phi'}{\llbracket e_1 \rrbracket_{\phi_1, \Delta \uparrow^M S; \tau = e_2} \llbracket e_2 \rrbracket_{\phi, \phi_2} \hookrightarrow \llbracket e_1 [s \mapsto e_2] \rrbracket_{\phi_1, \phi', \phi_2}} \\
 \text{CE-AX-BETA} \quad \text{CE-AX-TBETA} \quad \text{CE-AX-QUOTE SPLICE} \\
 \frac{}{(\lambda x : \tau. e_1) e_2 \hookrightarrow e_1 [x \mapsto e_2]} \quad \frac{}{(\Lambda a. e) \tau \hookrightarrow e [a \mapsto \tau]} \quad \frac{\llbracket s \rrbracket_{\bullet \uparrow^M S; \tau = e} \hookrightarrow e}{\text{CE-AX-CTX} \quad e_1 \hookrightarrow e_2} \\
 \frac{}{\mathbb{C}[e_1] \hookrightarrow \mathbb{C}[e_2]}
 \end{array}$$

We write $e_1 \hookrightarrow^* e_2$ to mean the reflexive, transitive and context closure of \hookrightarrow . Formally,

$$\boxed{e_1 \hookrightarrow^* e_2} \quad (\text{Reduction})$$

$$\begin{array}{c}
 \text{CE-AX-C-REFL} \quad \text{CE-AX-C-TRANS} \\
 \frac{}{e \hookrightarrow^* e} \quad \frac{e_1 \hookrightarrow e_2 \quad e_2 \hookrightarrow^* e_3}{e_1 \hookrightarrow^* e_3}
 \end{array}$$

G.4 Parallel Reduction

$$\boxed{e_1 \twoheadrightarrow e_2} \quad (\text{Parallel Reduction})$$

$$\begin{array}{c}
 \text{CE-AX-PA-LIT} \quad \text{CE-AX-PA-VAR} \quad \text{CE-AX-PA-SVAR} \quad \text{CE-AX-PA-KVAR} \\
 \frac{}{i \twoheadrightarrow i} \quad \frac{}{x \twoheadrightarrow x} \quad \frac{}{s \twoheadrightarrow s} \quad \frac{}{k \twoheadrightarrow k} \\
 \text{CE-AX-PA-ABS} \quad \text{CE-AX-PA-TABS} \quad \text{CE-AX-PA-APP} \quad \text{CE-AX-PA-TAPP} \\
 \frac{}{e_1 \twoheadrightarrow e_2} \quad \frac{}{e_1 \twoheadrightarrow e_2} \quad \frac{}{e_1 \twoheadrightarrow e_3 \quad e_2 \twoheadrightarrow e_4} \quad \frac{}{e_1 \twoheadrightarrow e_2} \\
 \lambda x : \tau. e_1 \twoheadrightarrow \lambda x : \tau. e_2 \quad \Lambda a. e_1 \twoheadrightarrow \Lambda a. e_2 \quad \frac{e_1 e_2 \twoheadrightarrow e_3 e_4}{\text{CE-AX-PA-TBETA}} \quad e_1 \tau \twoheadrightarrow e_2 \tau \\
 \text{CE-AX-PA-BETA} \\
 \frac{e_1 \twoheadrightarrow e_3 \quad e_2 \twoheadrightarrow e_4}{(\lambda x : \tau. e_1) e_2 \twoheadrightarrow e_3 [x \mapsto e_4]} \quad \frac{e_1 \twoheadrightarrow e_2}{(\Lambda a. e_1) \tau \twoheadrightarrow e_2 [a \mapsto \tau]} \\
 \text{CE-AX-PA-SPLICEQUOTE} \\
 e \twoheadrightarrow e' \quad \phi_i = \begin{cases} \phi_i'' & \text{where } e_i \twoheadrightarrow \llbracket e_i'' \rrbracket_{\phi_i'} \wedge \phi_i' \uparrow \Delta_i \rightsquigarrow \phi_i'' \\ \Delta_i \uparrow^{n_i} S_i : \tau_i = e_i' & \text{where } e_i \twoheadrightarrow e_i' \end{cases} \\
 \frac{\llbracket e \rrbracket_{\Delta_i \uparrow^{n_i} S_i; \tau_i = e_i} \twoheadrightarrow \llbracket e' [s_i \mapsto e_i''] \rrbracket_{\phi_i}^i}{\text{CE-AX-PA-QUOTE SPLICE} \quad e_1 \twoheadrightarrow e_2} \\
 \frac{}{\llbracket s \rrbracket_{\bullet \uparrow^M S; \tau = e_1} \twoheadrightarrow e_2}
 \end{array}$$

G.5 Complete Development

$e_1 \rightsquigarrow e_2$	(Complete Development)			
	CE-AX-CP-LIT	CE-AX-CP-VAR	CE-AX-CP-SVAR	CE-AX-CP-KVAR
	$\overline{i \rightsquigarrow i}$	$\overline{x \rightsquigarrow x}$	$\overline{s \rightsquigarrow s}$	$\overline{k \rightsquigarrow k}$
	CE-AX-CP-ABS	CE-AX-CP-TABS	CE-AX-CP-APP	
	$\overline{e_1 \rightsquigarrow e_2}$	$\overline{e_1 \rightsquigarrow e_2}$	$\overline{e_1 \rightsquigarrow e_3} \quad \overline{e_2 \rightsquigarrow e_4}$	$\overline{e_1 \neq \lambda x : \tau. e}$
	$\overline{\lambda x : \tau. e_1 \rightsquigarrow \lambda x : \tau. e_2}$	$\overline{\Lambda a. e_1 \rightsquigarrow \Lambda a. e_2}$	$\overline{e_1 e_2 \rightsquigarrow e_3 e_4}$	
	CE-AX-CP-TAPP	CE-AX-CP-BETA	CE-AX-CP-TBETA	
	$\overline{e_1 \rightsquigarrow e_2} \quad \overline{e_1 \neq \Lambda a. e}$	$\overline{e_1 \rightsquigarrow e_3} \quad \overline{e_2 \rightsquigarrow e_4}$	$\overline{e_1 \rightsquigarrow e_2}$	
	$\overline{e_1 \tau \rightsquigarrow e_2 \tau}$	$\overline{(\lambda x : \tau. e_1) e_2 \rightsquigarrow e_3 [x \mapsto e_4]}$	$\overline{(\Lambda a. e_1) \tau \rightsquigarrow e_2 [a \mapsto \tau]}$	
	CE-AX-CP-SPLICEQUOTE			
	$e \rightsquigarrow e'$			
$\phi_i = \begin{cases} \phi_i'' & \text{where } e_i \rightsquigarrow \llbracket e_i'' \rrbracket_{\phi_i'} \wedge \phi_i' \vdash \Delta_i \rightsquigarrow \phi_i'' \\ \Delta_i \vdash^{n_i} s_i : \tau_i = e_i' & \text{where } e_i \rightsquigarrow e_i' \wedge e_i' \neq \llbracket e_i'' \rrbracket_{\phi_i'} \end{cases}$	$\llbracket e \rrbracket_{\Delta_i \vdash^{n_i} s_i : \tau_i = e_i'}^i \neq \llbracket s \rrbracket_{\bullet \vdash^{n_i} s : \tau = e'''}^i$			
$\llbracket e \rrbracket_{\Delta_i \vdash^{n_i} s_i : \tau_i = e_i'}^i \rightsquigarrow \llbracket e' [s_i \mapsto e_i''] \rrbracket_{\phi_i'}^i$				
CE-AX-CP-QUOTE SPLICE				
$\overline{e_1 \rightsquigarrow e_2}$				
$\overline{\llbracket s \rrbracket_{\bullet \vdash^{n_i} s : \tau = e_1}^i \rightsquigarrow e_2}$				

G.6 Parallel Reduction with Complexity

$e_1 \xrightarrow{N} e_2$	<i>(Parallel Reduction with Derivation Complexity)</i>			
CE-AX-PPA-LIT	CE-AX-PPA-VAR	CE-AX-PPA-SVAR	CE-AX-PPA-KVAR	
$\overline{i \xrightarrow{0} i}$	$\overline{x \xrightarrow{0} x}$	$\overline{s \xrightarrow{0} s}$	$\overline{k \xrightarrow{0} k}$	
CE-AX-PPA-ABS	CE-AX-PPA-TABS	CE-AX-PPA-APP	CE-AX-PPA-TAPP	
$\overline{e_1 \xrightarrow{N} e_2}$	$\overline{e_1 \xrightarrow{N} e_2}$	$\overline{e_1 \xrightarrow{M} e_3} \quad \overline{e_2 \xrightarrow{N} e_4}$	$\overline{e_1 \xrightarrow{N} e_2}$	
$\overline{\lambda x : \tau. e_1 \xrightarrow{N} \lambda x : \tau. e_2}$	$\overline{\Lambda a. e_1 \xrightarrow{N} \Lambda a. e_2}$	$\overline{e_1 e_2 \xrightarrow{M+N} e_3 e_4}$	$\overline{e_1 \tau \xrightarrow{N} e_2 \tau}$	
CE-AX-PPA-BETA		CE-AX-PPA-TBETA		
$\overline{e_1 \xrightarrow{M} e_3} \quad \overline{e_2 \xrightarrow{N} e_4}$		$\overline{e_1 \xrightarrow{N} e_2}$		
$\overline{(\lambda x : \tau. e_1) e_2 \xrightarrow{M+\#(x, e_3)*N+1} e_3[x \mapsto e_4]}$		$\overline{(\Lambda a. e_1) \tau \xrightarrow{N+1} e_2[a \mapsto \tau]}$		
CE-AX-PPA-SPLICEQUOTE				
	$e \xrightarrow{N} e'$			
$\phi_i = \begin{cases} \phi_i'' & \text{where } e_i = \llbracket e_i'' \rrbracket_{\phi_i} \wedge e_i'' \xrightarrow{N_i} e_i''' \wedge \phi_i \xrightarrow{L_i} \phi_i' \wedge \phi_i' \vdash \Delta_i \rightsquigarrow \phi_i'' \\ \Delta_i \vdash^{n_i} s_i : \tau_i = e_i' & \text{where } e_i \xrightarrow{M_i} e_i' \end{cases}$				
$\llbracket e \rrbracket_{\Delta_i \vdash^{n_i} s_i : \tau_i = e_i'}^i \xrightarrow{N+\#(s_i, e')*\overline{N_i}+\overline{M_i}+\overline{L_i}+1} \llbracket e' [s_i \mapsto e_i'''] \rrbracket_{\phi_i'}^i$				
CE-AX-PPA-QUOTE SPLICE				
$\overline{e_1 \xrightarrow{N} e_2}$				
$\llbracket s \rrbracket_{\bullet \vdash^{n_i} s : \tau = e_1}^i \xrightarrow{N+1} e_2$				

$$\boxed{\phi_1 \xrightarrow{N} \phi_2}$$

(Parallel Reduction with Derivation Complexity)

$$\frac{\text{CE-AX-PPA-S-EMPTY} \quad \frac{0}{\bullet \xrightarrow{} \bullet} \quad \frac{\text{CE-AX-PPA-S-CONS} \quad \frac{\phi_1 \xrightarrow{N} \phi_2 \quad e_1 \xrightarrow{M} e_2}{\phi_1, \Delta \vdash^n s : \tau = e_1 \xrightarrow{N+M} \phi_2, \Delta \vdash^n s : \tau = e_2}}{\bullet \xrightarrow{} \bullet}}$$

For simplicity we also write $e_1 \xrightarrow{} e_2$ when the absolute complexity does not matter.

H LIST OF LEMMAS FOR AXIOMATIC SEMANTICS

H.1 Elaboration of Source Programs

Lemma H.1 ($\lambda \llbracket \Rightarrow \rrbracket =_{ax}$ to $F \llbracket \simeq_{ax} \rrbracket$).

- If $\text{pgm}_1 =_{ax} \text{pgm}_2$, where $\Theta \vdash \text{pgm}_1 : \sigma \rightsquigarrow \rho \text{gm}_1$, and $\Theta \vdash \text{pgm}_2 : \sigma \rightsquigarrow \rho \text{gm}_2$, and $\Theta \rightsquigarrow \Theta$, then $\Theta \vdash \rho \text{gm}_1 \simeq_{ax} \rho \text{gm}_2$.
- If $e_1 =_{ax} e_2$, where $\Theta; \Gamma \vdash^n e_1 : \sigma \rightsquigarrow e_1 \mid \phi_1$, and $\Theta; \Gamma \vdash^n e_2 : \sigma \rightsquigarrow e_2 \mid \phi_2$, and $\Theta \rightsquigarrow \Theta$, and $\Gamma \rightsquigarrow \Delta$, and $\bullet \vdash \sigma \rightsquigarrow \tau$ then if $e_1 : \tau \vdash^{n-1} \phi_1 \rightsquigarrow \rho \text{gm}_1$, and $e_2 : \tau \vdash^{n-1} \phi_2 \rightsquigarrow \rho \text{gm}_2$, then $\rho \text{gm}_1 =_{ax} \rho \text{gm}_2$.

Lemma H.2. If $\bullet; \bullet \vdash^n e_1 : \tau$, and $e_1 \longrightarrow e_2$, then $\bullet; \bullet \vdash^n e_1 \simeq_{ax} e_2 : \tau$.

Lemma H.3 (\longrightarrow Preserves \simeq_{ax}). • Given $\bullet \vdash \rho \text{gm}_1 \simeq_{ax} \rho \text{gm}_2$, if $\rho \text{gm}_1 \longrightarrow^* e_1 : \tau$ or $\rho \text{gm}_2 \longrightarrow^* e_2 : \tau$, then there exists $\rho \text{gm}'_1$ and $\rho \text{gm}'_2$, such that (1) either $\rho \text{gm}'_1 = \rho \text{gm} = v_1 : \tau$, or $\rho \text{gm}_1 \longrightarrow^+ \rho \text{gm}'_1$; (2) either $\rho \text{gm}'_2 = \rho \text{gm} = v_2 : \tau$, or $\rho \text{gm}_2 \longrightarrow^+ \rho \text{gm}'_2$; (3) and $\bullet \vdash \rho \text{gm}'_1 \simeq_{ax} \rho \text{gm}'_2$.

• Given $\bullet; \bullet \vdash^n e_1 \simeq_{ax} e_2 : \tau$, if $e_1 \longrightarrow^* v_1$, then $e_2 \longrightarrow^* v_2$, and $\bullet; \bullet \vdash^n v_1 \simeq_{ax} v_2 : \tau$, and vice versa.

Lemma H.4 ($F \llbracket \simeq_{ax} \rrbracket$ to $F \llbracket \simeq_{ctx} \rrbracket$). If $\Theta \vdash \rho \text{gm}_1 \simeq_{ax} \rho \text{gm}_2$, then $\Theta \vdash \rho \text{gm}_1 \simeq_{ctx} \rho \text{gm}_2 : \tau$.

Theorem H.5 ($\lambda \llbracket \Rightarrow \rrbracket =_{ax}$ to $F \llbracket \simeq_{ctx} \rrbracket$). If $\text{pgm}_1 =_{ax} \text{pgm}_2$, where $\Theta \vdash \text{pgm}_1 : \sigma \rightsquigarrow \rho \text{gm}_1$, and $\Theta \vdash \text{pgm}_2 : \sigma \rightsquigarrow \rho \text{gm}_2$, and $\Theta \rightsquigarrow \Theta$, and $\bullet \vdash \sigma \rightsquigarrow \tau$, then $\Theta \vdash \rho \text{gm}_1 \simeq_{ctx} \rho \text{gm}_2 : \tau$.

H.2 Elaboration of Source Expressions

Lemma H.6 (Substitution for $=_{ax}$).

- If $\rho \text{gm}_1 =_{ax} \rho \text{gm}_2$, and $v_1 =_{ax} v_2$, then $\rho \text{gm}_1[k \mapsto v_1] =_{ax} \rho \text{gm}_2[k \mapsto v_2]$.
- If $\rho \text{gm}_1 =_{ax} \rho \text{gm}_2$, and $v_1 =_{ax} v_2$, then $\rho \text{gm}_1[s \mapsto v_1] =_{ax} \rho \text{gm}_2[s \mapsto v_2]$.
- If $\llbracket e_1 \rrbracket_{\phi_{v_1}} =_{ax} \llbracket e_2 \rrbracket_{\phi_{v_2}}$, then $\llbracket \phi_{v_1} \rrbracket e_1 =_{ax} \llbracket \phi_{v_2} \rrbracket e_2$.

Theorem H.7. If $\bullet; \Gamma \vdash^n e_1 \simeq_{ax} e_2 : \tau$, then $\bullet; \Gamma \vdash^n e_1 \simeq_{ctx} e_2 : \tau$.

H.3 Axiomatic Reduction

Lemma H.8 (Transitivity). If $e_1 \hookrightarrow^* e_2$ and $e_2 \hookrightarrow^* e_3$, then $e_1 \hookrightarrow^* e_3$.

Lemma H.9 (Congruence). If $e_1 \hookrightarrow^* e_2$, then $\mathbb{C}[e_1] \hookrightarrow^* \mathbb{C}[e_2]$.

Lemma H.10 (\hookrightarrow to $=_{ax}$). Given $\Theta; \Delta \vdash^n e_1 : \tau$, if $e_1 \hookrightarrow e_2$ then $\Theta; \Delta \vdash^n e_1 \simeq_{ax} e_2 : \tau$.

Lemma H.11 (Confluence). *Given $\Theta; \Delta \Vdash e : \tau$, if $e \hookrightarrow^* e_1$ and $e \hookrightarrow^* e_2$, then there exists e' such that $e_1 \hookrightarrow^* e'$ and $e_2 \hookrightarrow^* e'$.*

Theorem H.12 (Church-Rosser). *If $\Theta; \Delta \Vdash e_1 \simeq_{ax} e_2 : \tau$, then there exists e such that $e_1 \hookrightarrow^* e$ and $e_2 \hookrightarrow^* e$.*

Corollary H.13. *Given $\Theta; \Delta \Vdash e : \text{Int}$, if $\Theta; \Delta \Vdash e \simeq_{ax} i : \text{Int}$ then $e \hookrightarrow^* i$.*

Lemma H.14. *If $e \longrightarrow^* v$, then $e \hookrightarrow^* v$.*

Lemma H.15. *Given $\Theta; \Delta \Vdash e : \tau$, if $e \hookrightarrow^* v$, then $e \longrightarrow^* v'$ for some v' .*

Corollary H.16. *Given $\Theta; \Delta \Vdash e : \text{Int}$, then we have $\Theta; \Delta \Vdash e \simeq_{ax} i : \text{Int}$ if and only if $e \longrightarrow^* i$.*

H.4 Church-Rosser

Lemma H.17 (Substitution).

- If $e_1 \hookrightarrow e_2$, and $e_3 \hookrightarrow e_4$, then $e_1[x \mapsto e_3] \hookrightarrow e_2[x \mapsto e_4]$.
- If $e_1 \hookrightarrow e_2$, then $e_1[a \mapsto \tau] \hookrightarrow e_2[a \mapsto \tau]$.
- If $e_1 \hookrightarrow e_2$, and $e_3 \hookrightarrow e_4$, then $e_1[s \mapsto e_3] \hookrightarrow e_2[s \mapsto e_4]$.

Lemma H.18 (Diamond Lemma). *Given $\Theta; \Delta \Vdash e : \tau$, if $e \hookrightarrow e_1$, and $e \hookrightarrow e_2$, then there exists e' such that $e_1 \hookrightarrow e'$ and $e_2 \hookrightarrow e'$.*

Lemma H.19 (Strip Lemma). *Given $\Theta; \Delta \Vdash e : \tau$, if $e \hookrightarrow e_1$, and $e \hookrightarrow^* e_2$, then there exists e' such that $e_1 \hookrightarrow^* e'$ and $e_2 \hookrightarrow e'$.*

Theorem H.20 (Confluence of \hookrightarrow^*). *Given $\Theta; \Delta \Vdash e : \tau$, if $e \hookrightarrow^* e_1$, and $e \hookrightarrow^* e_2$, then there exists e' such that $e_1 \hookrightarrow^* e'$ and $e_2 \hookrightarrow^* e'$.*

H.5 Parallel Reduction

Lemma H.21 (Reflexivity). $e \hookrightarrow e$.

Lemma H.22 (\hookrightarrow simulates \hookrightarrow). *If $e_1 \hookrightarrow e_2$, then $e_1 \hookrightarrow e_2$.*

Lemma H.23 (\hookrightarrow^* simulates \hookrightarrow). *If $e_1 \hookrightarrow e_2$, then $e_1 \hookrightarrow^* e_2$.*

Theorem H.24 (Equivalence of Parallel Reduction and Axiomatic Semantics). $e_1 \hookrightarrow^* e_2$ if and only if $e_1 \hookrightarrow^* e_2$.

H.6 Complete Development

Lemma H.25 (\hookrightarrow exists). *For any e , there exists e' such that $e \hookrightarrow e'$.*

Lemma H.26 (\hookrightarrow closes \hookrightarrow). *Given $\Theta; \Delta \Vdash e : \tau$, if $e \hookrightarrow e_1$, and $e \hookrightarrow e_2$, then $e_2 \hookrightarrow e_1$.*

H.7 Parallel Reduction with Complexity

Lemma H.27 (Reflexivity). $e \rightsquigarrow e$.

Lemma H.28 (\rightsquigarrow simulates \hookrightarrow). If $e_1 \hookrightarrow e_2$, then $e_1 \rightsquigarrow e_2$.

Lemma H.29 (\hookrightarrow^* simulates \rightsquigarrow). If $e_1 \rightsquigarrow e_2$, then $e_1 \hookrightarrow^* e_2$.

Lemma H.30 (Substitution).

- If $e_1 \xrightarrow{N_1} e_2$, and $e_3 \xrightarrow{N_2} e_4$, then there exists M , such that $e_1[x \mapsto e_3] \xrightarrow{M} e_2[x \mapsto e_4]$, where $M \leq N_1 + \#(x, e_2) * N_2$.
- If $e_1 \xrightarrow{N} e_2$, then $e_1[a \mapsto \tau] \xrightarrow{N} e_2[a \mapsto \tau]$.

Lemma H.31 (Monotonicity). If $v \rightsquigarrow e$, then e is also a value.

Lemma H.32 (Transition). If $e \rightsquigarrow v$, then there exists v_2 , such that $e \longrightarrow^* v_2$, and $v_2 \rightsquigarrow v$.

Lemma H.33 (Permutation). Given $\Theta; \Delta \Vdash e_1 : \tau$, if $e_1 \rightsquigarrow e_2$, and $e_2 \longrightarrow^* e_3$, then there exists e_4 , such that $e_1 \longrightarrow^* e_4$, and $e_4 \rightsquigarrow e_3$.

Lemma H.34 (Push Back). Given $\Theta; \Delta \Vdash e_1 : \tau$, if $e_1 \rightsquigarrow e_2$, and $e_2 \longrightarrow^* v_1$, then there exists v_2 , such that $e_1 \longrightarrow^* v_2$, and $v_2 \rightsquigarrow v_1$.

Lemma H.35 (\longrightarrow^* simulates \rightsquigarrow^*). Given $\Theta; \Delta \Vdash e : \tau$, if $e \rightsquigarrow^* v$, then there exists v_2 such that $e \longrightarrow^* v_2$, and $v_2 \rightsquigarrow^* v$.

I PROOFS FOR AXIOMATIC SEMANTICS

I.1 Elaboration of Source Programs

Lemma H.6 (Substitution for $=_{ax}$).

- If $\rho gm_1 =_{ax} \rho gm_2$, and $v_1 =_{ax} v_2$, then $\rho gm_1[k \mapsto v_1] =_{ax} \rho gm_2[k \mapsto v_2]$.
- If $\rho gm_1 =_{ax} \rho gm_2$, and $v_1 =_{ax} v_2$, then $\rho gm_1[s \mapsto v_1] =_{ax} \rho gm_2[s \mapsto v_2]$.
- If $\llbracket e_1 \rrbracket_{\phi_{v_1}} =_{ax} \llbracket e_2 \rrbracket_{\phi_{v_2}}$, then $\llbracket \phi_{v_1} \rrbracket e_1 =_{ax} \llbracket \phi_{v_2} \rrbracket e_2$.

The first two parts follows straightforward by induction on ρgm_1 and ρgm_2 . The third part can then be proved by repeating part 2. \square

Lemma H.1 ($\lambda \llbracket \Rightarrow \rrbracket =_{ax}$ to $F \llbracket \rrbracket \simeq_{ax}$).

- If $\rho gm_1 =_{ax} \rho gm_2$, where $\Theta \vdash \rho gm_1 : \sigma \rightsquigarrow \rho gm_1$, and $\Theta \vdash \rho gm_2 : \sigma \rightsquigarrow \rho gm_2$, and $\Theta \rightsquigarrow \Theta$, then $\Theta \vdash \rho gm_1 \simeq_{ax} \rho gm_2$.
- If $e_1 =_{ax} e_2$, where $\Theta; \Gamma \Vdash e_1 : \sigma \rightsquigarrow e_1 \mid \phi_1$, and $\Theta; \Gamma \Vdash e_2 : \sigma \rightsquigarrow e_2 \mid \phi_2$, and $\Theta \rightsquigarrow \Theta$, and $\Gamma \rightsquigarrow \Delta$, and $\bullet \vdash \sigma \rightsquigarrow \tau$ then if $e_1 : \tau \Vdash^{-1} \phi_1 \rightsquigarrow \rho gm_1$, and $e_2 : \tau \Vdash^{-1} \phi_2 \rightsquigarrow \rho gm_2$, then $\rho gm_1 =_{ax} \rho gm_2$.

PROOF. Part 1 By induction on $\rho gm_1 =_{ax} \rho gm_2$.

- $\rho gm_1 = \text{def } k = e_1; \rho gm_3$, and $\rho gm_2 = \text{def } k = e_2; \rho gm_4$, and $e_1 =_{ax} e_2$, and $\rho gm_3 =_{ax} \rho gm_4$.

$\Theta \vdash \text{def } k = e_1; \rho gm_3 : \sigma \rightsquigarrow \rho gm_1$

$\rho gm_3; \Theta \vdash k = e_1 \dashv \Theta, k : \tau \rightsquigarrow \rho gm_1$

given

inversion (rule S-PGM-DEF)

2696	$\Theta, k : \tau \vdash \text{pgm}_3 : \sigma \rightsquigarrow \rho \text{gm}_3$	
2697	$\Theta; \bullet \vdash^0 e_1 : \sigma_1 \rightsquigarrow e_1 \mid \phi_1$	inversion (rule S-DEF)
2698	$\bullet \vdash \sigma \rightsquigarrow \tau$	
2699	$\text{def } k : \tau = e_1; \rho \text{gm}_3 \vdash^1 \phi_1 \rightsquigarrow \rho \text{gm}_1$	inversion
2700	$\rho \text{gm}_1 = \text{spdef } \phi'_1; \text{def } k : \tau = e_1; \rho \text{gm}_3$	given
2701	$\Theta \vdash \text{def } k = e_2; \text{pgm}_4 : \sigma \rightsquigarrow \rho \text{gm}_2$	inversion (rule S-PGM-DEF)
2702	$\rho \text{gm}_4; \Theta \vdash k = e_2 \vdash \Theta, k : \tau \rightsquigarrow \rho \text{gm}_2$	
2703	$\Theta, k : \tau \vdash \text{pgm}_3 : \sigma \rightsquigarrow \rho \text{gm}_4$	
2704	$\Theta; \bullet \vdash^0 e_2 : \sigma_1 \rightsquigarrow e_2 \mid \phi_2$	inversion (rule S-DEF)
2705	$\bullet \vdash \sigma \rightsquigarrow \tau$	
2706	$\text{def } k : \tau = e_2; \rho \text{gm}_4 \vdash^1 \phi_2 \rightsquigarrow \rho \text{gm}_2$	inversion
2707	$\rho \text{gm}_2 = \text{spdef } \phi'_2; \text{def } k : \tau = e_2; \rho \text{gm}_4$	I.H.
2708	$\Theta \vdash \rho \text{gm}_3 \sqsubseteq_{\text{ax}} \rho \text{gm}_4$	Part 2
2709	$e_1 : \tau \vdash^1 \phi_1 \rightsquigarrow \text{spdef } \phi'_1; e_1 : \tau$	above
2710	$e_2 : \tau \vdash^1 \phi_2 \rightsquigarrow \text{spdef } \phi'_2; e_2 : \tau$	above
2711	$\text{spdef } \phi'_1; e_1 : \tau =_{\text{ax}} \text{spdef } \phi'_2; e_2 : \tau$	
2712	$\phi'_1, \phi'_2 \text{ fresh w.r.t. } \rho \text{gm}_3 \text{ and } \rho \text{gm}_4$	
2713	$\Theta \vdash \text{spdef } \phi'_1; \text{def } k : \tau = e_1; \rho \text{gm}_3 \sqsubseteq_{\text{ax}} \text{spdef } \phi'_2; \text{def } k : \tau = e_2; \rho \text{gm}_4$	follows
2714		
2715		
2716	$\bullet \text{pgm}_1 = \text{class TC } a \text{ where } \{k : \rho\}; \text{pgm}_3, \text{ and } \text{pgm}_2 = \text{class TC } a \text{ where } \{k : \rho\}; \text{pgm}_4, \text{ and } \text{pgm}_3 =_{\text{ax}} \text{pgm}_4.$	
2717		
2718		
2719	$\Theta \vdash \text{class TC } a \text{ where } \{k : \rho\}; \text{pgm}_3 : \sigma \rightsquigarrow \rho \text{gm}_1$	given
2720	$\Theta, k : \forall a. \text{TC } a \Rightarrow \rho \vdash \text{pgm}_3 : \sigma \rightsquigarrow \rho \text{gm}_3$	inversion (rule S-PGM-CLS)
2721	$\rho \text{gm}_3; \Theta \vdash \text{TC } a \text{ where } \{k : \rho\} \vdash \Theta, k : \forall a. \text{TC } a \Rightarrow \rho \rightsquigarrow \text{def } k : \forall a. \tau \rightarrow \tau = \Lambda a. \lambda x : \tau. x; \rho \text{gm}_3$	inversion (rule S-CLS)
2722		
2723	$a \vdash \rho \rightsquigarrow \tau$	
2724	$\rho \text{gm}_1 = \text{def } k : \forall a. \tau \rightarrow \tau = \Lambda a. \lambda x : \tau. x; \rho \text{gm}_3$	given
2725	$\Theta \vdash \text{class TC } a \text{ where } \{k : \rho\}; \text{pgm}_4 : \sigma \rightsquigarrow \rho \text{gm}_2$	inversion (rule S-PGM-CLS)
2726	$\Theta, k : \forall a. \text{TC } a \Rightarrow \rho \vdash \text{pgm}_4 : \sigma \rightsquigarrow \rho \text{gm}_4$	
2727	$\rho \text{gm}_4; \Theta \vdash \text{TC } a \text{ where } \{k : \rho\} \vdash \Theta, k : \forall a. \text{TC } a \Rightarrow \rho \rightsquigarrow \text{def } k : \forall a. \tau \rightarrow \tau = \Lambda a. \lambda x : \tau. x; \rho \text{gm}_4$	inversion (rule S-PGM-CLS)
2728		
2729	$\rho \text{gm}_2 = \text{def } k : \forall a. \tau \rightarrow \tau = \Lambda a. \lambda x : \tau. x; \rho \text{gm}_4$	I.H.
2730	$\Theta, k : \forall a. \tau \rightarrow \tau \vdash \rho \text{gm}_3 \sqsubseteq_{\text{ax}} \rho \text{gm}_4$	follows
2731	$\Theta \vdash \rho \text{gm}_1 \sqsubseteq_{\text{ax}} \rho \text{gm}_2$	
2732		
2733		
2734	$\bullet \text{pgm}_1 = \text{inst } \overline{C}_i^i \Rightarrow \text{TC } \tau \text{ where } \{k = e_1\}; \text{pgm}_3, \text{ and } \text{pgm}_2 = \text{inst } \overline{C}_i^i \Rightarrow \text{TC } \tau \text{ where } \{k = e_2\}; \text{pgm}_4, \text{ and } e_1 =_{\text{ax}} e_2, \text{ and } \text{pgm}_3 =_{\text{ax}} \text{pgm}_4.$	
2735		
2736		
2737	$\Theta \vdash \text{inst } \overline{C}_i^i \Rightarrow \text{TC } \tau \text{ where } \{k = e_1\}; \text{pgm}_3 : \sigma \rightsquigarrow \rho \text{gm}_1$	given
2738	$\Theta, \forall b_j^j. \overline{C}_i^i \Rightarrow \text{TC } \tau \vdash \text{pgm}_3 : \sigma \rightsquigarrow \rho \text{gm}_3$	inversion (rule S-PGM-INST)
2739	$\rho \text{gm}_3; \Theta \vdash \overline{C}_i^i \Rightarrow \text{TC } \tau \text{ where } \{k = e_1\} \vdash \Theta, \forall b_j^j. \overline{C}_i^i \Rightarrow \text{TC } \tau \rightsquigarrow \rho \text{gm}_1$	inversion (rule S-INST)
2740		
2741	$\Theta; \overline{b}_j^j, \overline{ev_i : (C_i, 0)}^i \vdash^0 e_1 : \rho[a \mapsto \tau] \rightsquigarrow e_1 \mid \phi_1$	
2742		
2743		
2744		

2745	$\phi_1 \dashv\vdash (\overline{b_j^j}, \overline{ev_i : (\tau_i, 0)^i}) \rightsquigarrow \phi_3$	
2746	$\rho gm_1 = \mathbf{spdef} \phi'_3; \mathbf{def} \text{ ev} : \forall \overline{b_j^j} . \overline{\tau_i^i} \rightarrow \tau = \Lambda \overline{b_j^j} . \lambda \overline{ev_i : \tau_i^i} . e_1; \rho gm_3$	
2747	$\Theta \vdash \mathbf{inst} \overline{C_i^i} \Rightarrow \text{TC } \tau \text{ where } \{k = e_2\}; \rho gm_4 : \sigma \rightsquigarrow \rho gm_2$	given
2748	$\Theta, \forall \overline{b_j^j} . \overline{C_i^i} \Rightarrow \text{TC } \tau \vdash \rho gm_4 : \sigma \rightsquigarrow \rho gm_4$	inversion (rule S-PGM-INST)
2749	$\rho gm_4; \Theta \vdash \overline{C_i^i} \Rightarrow \text{TC } \tau \text{ where } \{k = e_2\} \dashv\vdash \Theta, \forall \overline{b_j^j} . \overline{C_i^i} \Rightarrow \text{TC } \tau \rightsquigarrow \rho gm_2$	
2750		inversion (rule S-INST)
2751	$\Theta; \overline{b_j^j}, \overline{ev_i : (C_i, 0)^i} \vdash^0 e_2 : \rho[a \mapsto \tau] \rightsquigarrow e_2 \mid \phi_2$	
2752	$\phi_2 \dashv\vdash (\overline{b_j^j}, \overline{ev_i : (\tau_i, 0)^i}) \rightsquigarrow \phi_4$	
2753	$\rho gm_2 = \mathbf{spdef} \phi'_4; \mathbf{def} \text{ ev} : \forall \overline{b_j^j} . \overline{\tau_i^i} \rightarrow \tau = \Lambda \overline{b_j^j} . \lambda \overline{ev_i : \tau_i^i} . e_2; \rho gm_4$	
2754	$\Theta, \text{ev} : \forall \overline{b_j^j} . \overline{\tau_i^i} \rightarrow \tau \vdash \rho gm_3 \simeq_{ax} \rho gm_4$	I.H.
2755	$e_1 : \tau \vdash^{-1} \phi_1 \rightsquigarrow \mathbf{spdef} \phi'_1; e_1 : \tau$	Part 2
2756	$e_2 : \tau \vdash^{-1} \phi_2 \rightsquigarrow \mathbf{spdef} \phi'_2; e_2 : \tau$	above
2757	$\mathbf{spdef} \phi'_1; e_1 : \tau =_{ax} \mathbf{spdef} \phi'_2; e_2 : \tau$	above
2758	ϕ'_3, ϕ'_4 fresh w.r.t. ρgm_3 and ρgm_4	
2759	$\mathbf{spdef} \phi'_3; \mathbf{def} \text{ ev} : \forall \overline{b_j^j} . \overline{\tau_i^i} \rightarrow \tau = \Lambda \overline{b_j^j} . \lambda \overline{ev_i : \tau_i^i} . e_1; \rho gm_3$	
2760	$=_{ax} \mathbf{spdef} \phi'_4; \mathbf{def} \text{ ev} : \forall \overline{b_j^j} . \overline{\tau_i^i} \rightarrow \tau = \Lambda \overline{b_j^j} . \lambda \overline{ev_i : \tau_i^i} . e_2; \rho gm_4$	follows
2761	$\Theta \vdash \rho gm_1 \simeq_{ax} \rho gm_2$	namely
2762		
2763	$\bullet \rho gm_1 = e_1, \text{ and } \rho gm_2 = e_2. \text{ and } e_1 =_{ax} e_2.$	
2764		
2765	$\Theta \vdash e_1 : \sigma \rightsquigarrow \rho gm_1$	given
2766	$\bullet \vdash \sigma \rightsquigarrow \tau$	inversion (rule S-PGM-EXPR)
2767	$\Theta; \bullet \vdash^0 e_1 : \sigma \rightsquigarrow e_1 \mid \phi_1$	
2768	$e_1 : \tau \vdash^{-1} \phi_1 \rightsquigarrow \rho gm_1$	
2769	$\Theta \vdash e_2 : \sigma \rightsquigarrow \rho gm_2$	given
2770	$\Theta; \bullet \vdash^0 e_2 : \sigma \rightsquigarrow e_2 \mid \phi_2$	inversion (rule S-PGM-EXPR)
2771	$e_2 : \tau \vdash^{-1} \phi_2 \rightsquigarrow \rho gm_2$	
2772	$\rho gm_1 =_{ax} \rho gm_2$	Part 2
2773	$\Theta \vdash \rho gm_1 \simeq_{ax} \rho gm_2$	follows
2774		

Part 2 By induction on $e_1 =_{ax} e_2$.

- The case for $e_1 = e_2 = i$, and x , and k are trivial.
- $e_1 = \lambda x : \tau. e_3$, and $e_2 = \lambda x : \tau. e_4$, and $e_3 =_{ax} e_4$.

2782	$\Theta; \Gamma, x : (\tau, n) \vdash^n e_3 : \tau_2 \rightsquigarrow e_1 \mid \phi_1$	given
2783	$\phi_1 \dashv\vdash x : (\tau'_1, n) \rightsquigarrow \phi_3$	
2784	$\Theta; \Gamma, x : (\tau, n) \vdash^n e_4 : \tau_2 \rightsquigarrow e_2 \mid \phi_2$	given
2785	$\phi_2 \dashv\vdash x : (\tau'_1, n) \rightsquigarrow \phi_4$	
2786	$e_1 : \tau \vdash^{n-1} \phi_1 \rightsquigarrow \mathbf{spdef} \phi'_1; e_1 : \tau$	ϕ'_1 is sorted ϕ_1
2787	$e_2 : \tau \vdash^{n-1} \phi_2 \rightsquigarrow \mathbf{spdef} \phi'_2; e_2 : \tau$	ϕ'_2 is sorted ϕ_2
2788	$\mathbf{spdef} \phi'_1; e_1 : \tau =_{ax} \mathbf{spdef} \phi'_2; e_2 : \tau$	I.H.
2789	$(\lambda x : \tau'. e_1) : \tau' \rightarrow \tau \vdash^{n-1} \phi_3 \rightsquigarrow \mathbf{spdef} \phi'_3; (\lambda x : \tau'. e_1) : \tau' \rightarrow \tau$	ϕ'_3 is sorted ϕ_3
2790	$\phi'_1 \dashv\vdash x : (\tau'_1, n) \rightsquigarrow \phi'_3$	follows
2791	$(\lambda x : \tau'. e_2) : \tau' \rightarrow \tau \vdash^{n-1} \phi_4 \rightsquigarrow \mathbf{spdef} \phi'_4; (\lambda x : \tau'. e_1) : \tau' \rightarrow \tau$	ϕ'_4 is sorted ϕ_4
2792		
2793		

$\phi'_2 \dashv\vdash x : (\tau'_1, n) \rightsquigarrow \phi'_4$ | follows

We can then derive the conclusion $\mathbf{spdef} \phi'_3; (\lambda x : \tau'.e_1) : \tau' \rightarrow \tau =_{ax} \mathbf{spdef} \phi'_4; (\lambda x : \tau'.e_1) : \tau' \rightarrow \tau$ by first applying the same sequence of equivalence rules as for $\mathbf{spdef} \phi'_1; e_1 : \tau =_{ax} \mathbf{spdef} \phi'_2; e_2 : \tau$, until we need to prove the equivalence of two expressions. At that point we apply rule **EQ-CTX** to remove the lambda, then we can apply again the same sequence of equivalence rules as for proving the equivalence of two expressions in $\mathbf{spdef} \phi'_1; e_1 : \tau =_{ax} \mathbf{spdef} \phi'_2; e_2 : \tau$.

- The case for rule **S-TABS** and rule **S-CABS** is the same as the above one.
- $e_1 = e_3 e_4$, and $e_2 = e_5 e_6$, and $e_3 =_{ax} e_5$, $e_4 =_{ax} e_6$.

$\Theta; \Gamma \Vdash e_3 : \tau_1 \rightarrow \tau_2 \rightsquigarrow e_3 \mid \phi_3$	given
$\Theta; \Gamma \Vdash e_4 : \tau_1 \rightsquigarrow e_4 \mid \phi_4$	given
$\Theta; \Gamma \Vdash e_5 : \tau_1 \rightarrow \tau_2 \rightsquigarrow e_5 \mid \phi_5$	given
$\Theta; \Gamma \Vdash e_6 : \tau_1 \rightsquigarrow e_6 \mid \phi_6$	given
$e_3 : \tau_1 \rightarrow \tau_2 \Vdash^{n-1} \phi_3 \rightsquigarrow \mathbf{spdef} \phi'_3; e_3 : \tau_1 \rightarrow \tau_2$	ϕ'_3 is sorted ϕ_3
$e_4 : \tau_1 \Vdash^{n-1} \phi_4 \rightsquigarrow \mathbf{spdef} \phi'_4; e_4 : \tau_1$	ϕ'_4 is sorted ϕ_4
$e_5 : \tau_1 \rightarrow \tau_2 \Vdash^{n-1} \phi_5 \rightsquigarrow \mathbf{spdef} \phi'_5; e_5 : \tau_1 \rightarrow \tau_2$	ϕ'_5 is sorted ϕ_5
$e_6 : \tau_1 \Vdash^{n-1} \phi_6 \rightsquigarrow \mathbf{spdef} \phi'_6; e_6 : \tau_1$	ϕ'_6 is sorted ϕ_6
$\mathbf{spdef} \phi'_3; e_3 : \tau_1 \rightarrow \tau_2 =_{ax} \mathbf{spdef} \phi'_5; e_5 : \tau_1 \rightarrow \tau_2$	I.H.
$\mathbf{spdef} \phi'_4; e_4 : \tau_1 =_{ax} \mathbf{spdef} \phi'_6; e_6 : \tau_1$	I.H.
$e_3 e_4 : \tau_2 \Vdash^{n-1} (\phi_3, \phi_4) \rightsquigarrow \rho gm_1$	let
$e_5 e_6 : \tau_2 \Vdash^{n-1} (\phi_5, \phi_6) \rightsquigarrow \rho gm_2$	let

Assume the level range of $\phi_3, \phi_4, \phi_5, \phi_6$ is n to n' , then $\mathbf{spdef} \phi'_3; e_3 : \tau_1 \rightarrow \tau_2$ can be represented as:

$\mathbf{spdef} \phi_3.n; \mathbf{spdef} \phi_3.n+1; \dots; \mathbf{spdef} \phi_3.n'; e_3 : \tau_1 \rightarrow \tau_2$.
 So $\rho gm_1 = \mathbf{spdef} \phi_3.n; \mathbf{spdef} \phi_4.n; \mathbf{spdef} \phi_3.n+1; \mathbf{spdef} \phi_4.n+1; \dots;$
 $\mathbf{spdef} \phi_3.n'; \mathbf{spdef} \phi_4.n'; e_3 e_4 : \tau_2$
 $\rho gm_2 = \mathbf{spdef} \phi_5.n; \mathbf{spdef} \phi_6.n; \mathbf{spdef} \phi_5.n+1; \mathbf{spdef} \phi_6.n+1; \dots;$
 $\mathbf{spdef} \phi_5.n'; \mathbf{spdef} \phi_6.n'; e_5 e_6 : \tau_2$

Our goal is to prove $\rho gm_1 =_{ax} \rho gm_2$.

We can proceed by applying the interleaving sequence of rules used to prove the equivalence of the splice definitions for $\mathbf{spdef} \phi'_3; e_3 : \tau_1 \rightarrow \tau_2 =_{ax} \mathbf{spdef} \phi'_5; e_5 : \tau_1 \rightarrow \tau_2$ and $\mathbf{spdef} \phi'_4; e_4 : \tau_1 =_{ax} \mathbf{spdef} \phi'_6; e_6 : \tau_1$, until we need to prove the equivalence of expressions, which are applications.

Note that since every time we generate fresh splice variables, substituting splice variables in ϕ_3 with their expressions in e_4 keeps e_4 unchanged. Similarly, substituting ϕ_4 in e_3 and substituting ϕ_5 to e_6 , and substituting ϕ_6 to e_5 will keep the expression unchanged.

Therefore at the point when we need to prove the equivalence of the applications, the application we get is simply $e_3 e_4$ with e_3 substituted by some splice variables with their expressions in ϕ_3 , e_4 substituted by some splice variables with their expressions in ϕ_4 ; and $e_5 e_6$ with similar substitutions. Note the result expressions (substituted e_3, e_4, e_5, e_6) are the same as the substituted one we got in the derivation tree in I.H.

Now we can first apply rule **EQ-TRANS**, rule **EQ-CTX** to split the applications into two subexpressions and prove the equivalence of the two subexpressions respectively, i.e., the equivalence between substituted e_3 and e_5 , and between substituted e_4 and e_6 .

Now we can again apply the same sequence of rules applied in the I.H. to complete the proof.

- The case for rule **S-TAPP** and rule **S-CAPP** is the same as the above one. The case for rule **S-CAPP** requires a similar lemma for constraint solving. As the form of rules for constraint solving is essentially the same as expression typing, the lemma can be proved in a similar way.
- $e_1 = \llbracket e_3 \rrbracket$, and $e_2 = \llbracket e_4 \rrbracket$, and $e_3 =_{ax} e_4$.

$\Theta; \Gamma \vdash^{n+1} e_3 : \tau$	$\tau \rightsquigarrow e_3 \mid \phi_3$	given
$\Theta; \Gamma \vdash^n \llbracket e_3 \rrbracket : \text{Code } \tau$	$\tau \rightsquigarrow \llbracket e_3 \rrbracket_{\phi_3.n} \mid \lfloor \phi_3 \rfloor^n$	given
$\Theta; \Gamma \vdash^{n+1} e_4 : \tau$	$\tau \rightsquigarrow e_4 \mid \phi_4$	given
$\Theta; \Gamma \vdash^n \llbracket e_4 \rrbracket : \text{Code } \tau$	$\tau \rightsquigarrow \llbracket e_4 \rrbracket_{\phi_4.n} \mid \lfloor \phi_4 \rfloor^n$	given
$e_3 : \tau \vdash^n \phi_3 \rightsquigarrow \rho gm_3$		let
$e_4 : \tau \vdash^n \phi_4 \rightsquigarrow \rho gm_4$		let
$\rho gm_3 =_{ax} \rho gm_4$		I.H.

According to the definition, suppose ϕ_5 is the sorted $\lfloor \phi_3 \rfloor^n$ and ϕ_6 is the sorted $\lfloor \phi_4 \rfloor^n$. Then ρgm_3 can be represented as:

spdef $\phi_5; \text{spdef } \phi_3.n; e_3 : \tau$.

And ρgm_4 can be represented as:

spdef $\phi_6; \text{spdef } \phi_4.n; e_4 : \tau$.

Now our goal is to prove

spdef $\phi_5; \llbracket e_3 \rrbracket_{\phi_3.n} : \text{Code } \tau =_{ax} \text{spdef } \phi_6; \llbracket e_4 \rrbracket_{\phi_4.n} : \text{Code } \tau$

We can proceed by applying the sequence of rules used to prove the equivalence of the splice definitions for $\rho gm_3 =_{ax} \rho gm_4$, until we need to prove

$\llbracket e'_3 \rrbracket_{\phi'_3.n} : \text{Code } \tau =_{ax} \llbracket e'_4 \rrbracket_{\phi'_4.n} : \text{Code } \tau$

whereas in I.H., we have

spdef $\phi'_3.n; e'_3 : \tau =_{ax} \text{spdef } \phi'_4.n; e'_4 : \tau$.

where $\phi'_3.n, \phi'_4.n, e'_3$ and e'_4 are ϕ_3, ϕ_4, e_3 and e_4 after the substitution caused by rule **PEQ-SPDEF-AX**.

At this point, I.H. will further apply a mix of rule **PEQ-SPDEF** and rule **PEQ-SPDEF-AX** (with refl, symm, trans, congruence in between). We can correspondingly apply rule **EQ-CTX** (with rule **EQ-TRANS**) and rule **EQ-SPLICEQUOTE** (and refl etc respectively).

If the I.H. applies rule **PEQ-SPDEF**, for example between **spdef** $\Delta \vdash^s s : \tau = e_5; \text{spdef } \phi'_3.n; e'_3 : \tau =_{ax} \text{spdef } \Delta \vdash^s s : \tau = e_6; \text{spdef } \phi'_4.n; e'_4 : \tau$ where $e_5 =_{ax} e_6$, then our goal is to prove

$\llbracket e'_3 \rrbracket_{\Delta \vdash^s s : \tau = e_5, \phi'_3.n} : \text{Code } \tau =_{ax} \llbracket e'_4 \rrbracket_{\Delta \vdash^s s : \tau = e_6, \phi'_4.n} : \text{Code } \tau$.

We then apply rule **EQ-TRANS** with an intermediate expression $\llbracket e'_3 \rrbracket_{\Delta \vdash^s s : \tau = e_6, \phi'_3.n} : \text{Code } \tau$.

Note that $\llbracket e'_3 \rrbracket_{\Delta \vdash^s s : \tau = e_5, \phi'_3.n} : \text{Code } \tau =_{ax} \llbracket e'_3 \rrbracket_{\Delta \vdash^s s : \tau = e_6, \phi'_3.n} : \text{Code } \tau$ holds by rule **EQ-CTX**. And

now our goal is to prove $\llbracket e'_3 \rrbracket_{\Delta \vdash^s s : \tau = e_6, \phi'_3.n} : \text{Code } \tau =_{ax} \llbracket e'_4 \rrbracket_{\Delta \vdash^s s : \tau = e_6, \phi'_4.n} : \text{Code } \tau$.

In this case, we have assumed s is the first splice variable in the splice definition of e'_3 and e'_4 , but it does not have to be. That means, the s may appear in the middle of the splice definitions.

Note that while rule **PEQ-SPDEF** eliminates one definition at a time, to prove our goal we don't eliminate the splice definition but we introduce an intermediate expression so that our new goal will have the same splice definition at that place eliminated by rule **PEQ-SPDEF** (like in the above case).

On the other hand, if the I.H. applies rule **PEQ-SPDEF-AX**, then we will apply rule **EQ-SPLICEQUOTE**. Note the similarity between rule **PEQ-SPDEF-AX** and rule **EQ-SPLICEQUOTE**. The only difference is that rule **EQ-SPLICEQUOTE** allows splice definitions in front of the s the rule is applied on. But since in the first step we have already make all splice definitions in the front of s equivalent, we can safely apply rule **EQ-SPLICEQUOTE**.

Of course if the I.H. applies the refl, symm or trans, we will apply refl, symm, trans correspondingly.

Through this sequence of rules we can finally end up comparing the expressions e_3'' and e_4'' (which are e_3' and e_4' after further substitution caused by rule **PEQ-SPDEF-AX** and rule **EQ-SPLICEQUOTE** respectively). And we can then apply the same rules used in the I.H. to prove our final result.

- $e_1 = \$e_3$, and $e_2 = \$e_4$, and $e_3 =_{ax} e_4$.

$\Theta; \Gamma \vdash^{n-1} e_3 : \text{Code } \tau \rightsquigarrow e_3 \mid \phi_3$	given
$\Theta; \Gamma \vdash^n \$e_3 : \tau \rightsquigarrow s \mid \phi_3, \bullet \vdash^{n-1} s : \tau = e_3$	given
$\Theta; \Gamma \vdash^{n-1} e_4 : \text{Code } \tau \rightsquigarrow e_4 \mid \phi_3$	given
$\Theta; \Gamma \vdash^n \$e_4 : \tau \rightsquigarrow s \mid \phi_4, \bullet \vdash^{n-1} s : \tau = e_4$	given
$e_3 : \text{Code } \tau \vdash^n \phi_3 \rightsquigarrow \rho gm_3$	let
$e_4 : \text{Code } \tau \vdash^n \phi_4 \rightsquigarrow \rho gm_4$	let
$\rho gm_3 =_{ax} \rho gm_4$	I.H.

According to the definition, suppose ϕ_5 is the sorted $\lfloor \phi_3 \rfloor^n$ and ϕ_6 is the sorted $\lfloor \phi_4 \rfloor^n$. Then ρgm_3 can be represented as:

spdef $\phi_5; e_3 : \text{Code } \tau$.

And ρgm_4 can be represented as:

spdef $\phi_6; e_4 : \text{Code } \tau$.

So our I.H. is

spdef $\phi_5; e_3 : \text{Code } \tau =_{ax} \text{spdef } \phi_6; e_4 : \text{Code } \tau$.

Now our goal is to prove

spdef $\phi_5; \text{spdef } \bullet \vdash^{n-1} s : \tau = e_3; s : \tau =_{ax} \text{spdef } \phi_6; \text{spdef } \bullet \vdash^{n-1} s : \tau = e_4; s : \tau$

We can prove our goal by first following the proof of the I.H., until we need to prove

spdef $\bullet \vdash^{n-1} s : \tau = e_3'; s : \tau =_{ax} \text{spdef } \bullet \vdash^{n-1} s : \tau = e_4'; s : \tau$

where e_3' and e_4' are e_3 and e_4 after the substitution introduced by rule **PEQ-SPDEF-AX**.

At this point we can apply rule **PEQ-SPDEF** and uses the same proof used by the I.H. to prove $e_3 =_{ax} e_4$. Furthermore we have $s =_{ax} s$ by rule **EQ-REFL**. That concludes our proof.

- $e_1 = \$[e]$, and $e_2 = e$.

$\Theta; \Gamma \vdash^n e : \tau \rightsquigarrow e_1 \mid \phi_1$	given
$\Theta; \Gamma \vdash^n \$[e] : \tau \rightsquigarrow s \mid \lfloor \phi_1 \rfloor^{n-1}, \bullet \vdash^{n-1} s : \tau = \llbracket e_1 \rrbracket_{\phi_1, n-1}$	given
$e_1 : \tau \vdash^n \phi_1 \rightsquigarrow \text{spdef } \lfloor \phi_1' \rfloor^{n-1}; \text{spdef } \phi_1.n - 1; e_1 : \text{Code } \tau$	let
$s : \tau \vdash^n \lfloor \phi_1 \rfloor^{n-1}, \bullet \vdash^{n-1} s : \tau = \llbracket e_1 \rrbracket_{\phi_1, n-1} \rightsquigarrow \text{spdef } \phi_1'.n - 1; \text{spdef } \bullet \vdash^{n-1} s : \tau = \llbracket e_1 \rrbracket_{\phi_1, n-1}; s : \tau$	let

Our goal is to prove

spdef $\lfloor \phi_1' \rfloor^{n-1}; \text{spdef } \phi_1.n - 1; e_1 : \text{Code } \tau$

$=_{ax} \text{spdef } \phi_1'.n - 1; \text{spdef } \bullet \vdash^{n-1} s : \tau = \llbracket e_1 \rrbracket_{\phi_1, n-1}; s : \tau$.

We can prove the goal by a sequence of rule **PEQ-SPDEF**, followed by one rule **PEQ-SPDEF-AX**.

- $e_1 = \llbracket \$e \rrbracket$, and $e_2 = e$.

$\Theta; \Gamma \Vdash e : \text{Code } \tau \rightsquigarrow e_1 \mid \phi_1$	given
$\Theta; \Gamma \Vdash \llbracket \$e \rrbracket : \text{Code } \tau \rightsquigarrow \llbracket s \rrbracket_{\bullet \Vdash s : \tau = e_1} \mid \phi_1$	given
$e_1 : \text{Code } \tau \Vdash \phi_1 \rightsquigarrow \text{spdef } \phi'_1; e_1 : \text{Code } \tau$	let
$\llbracket s \rrbracket_{\bullet \Vdash s : \tau = e_1} : \text{Code } \tau \Vdash \phi_1 \rightsquigarrow \text{spdef } \phi'_1; \llbracket s \rrbracket_{\bullet \Vdash s : \tau = e_1} : \text{Code } \tau$	let

Our goal is to prove

$\text{spdef } \phi'_1; e_1 : \text{Code } \tau =_{ax} \text{spdef } \phi'_1; \llbracket s \rrbracket_{\bullet \Vdash s : \tau = e_1} : \text{Code } \tau$.

We can prove the goal by a sequence of rule **PEQ-SPDEF**, followed by one rule **PEQ-EXPR**, which is then proved by rule **EQ-QUOTESPLICE**.

□

Lemma H.2. *If $\bullet; \bullet \Vdash e_1 : \tau$, and $e_1 \longrightarrow e_2$, then $\bullet; \bullet \Vdash e_1 \simeq_{ax} e_2 : \tau$.*

PROOF. By a straightforward induction on $e_1 \longrightarrow e_2$, making use of rule **EQ-TRANS**.

□

Lemma H.3 (\longrightarrow Preserves \simeq_{ax}). • Given $\bullet \vdash \rho gm_1 \simeq_{ax} \rho gm_2$, if $\rho gm_1 \longrightarrow^* e_1 : \tau$ or $\rho gm_2 \longrightarrow^* e_2 : \tau$, then there exists $\rho gm'_1$ and $\rho gm'_2$, such that (1) either $\rho gm'_1 = \rho gm = v_1 : \tau$, or $\rho gm_1 \longrightarrow^+ \rho gm'_1$; (2) either $\rho gm'_2 = \rho gm = v_2 : \tau$, or $\rho gm_2 \longrightarrow^+ \rho gm'_2$; (3) and $\bullet \vdash \rho gm'_1 \simeq_{ax} \rho gm'_2$.

• Given $\bullet; \bullet \Vdash e_1 \simeq_{ax} e_2 : \tau$, if $e_1 \longrightarrow^* v_1$, then $e_2 \longrightarrow^* v_2$, and $\bullet; \bullet \Vdash v_1 \simeq_{ax} v_2 : \tau$, and vice versa.

PROOF. Part 1 By case analysis on $=_{ax}$.

– Case

PEQ-DEF

$$\frac{e_1 =_{ax} e_2 \quad \rho gm_1 =_{ax} \rho gm_2}{\text{def } k : \tau = e_1; \rho gm_1 =_{ax} \text{def } k : \tau = e_2; \rho gm_2}$$

$e_1 \longrightarrow^* v_1$

$e_2 \longrightarrow^* v_2$

$\bullet; \bullet \Vdash v_1 \simeq_{ax} v_2 : \tau$

$\text{def } k : \tau = e_1; \rho gm_1 \longrightarrow^* \text{def } k : \tau = v_1; \rho gm_1$

$\text{def } k : \tau = e_2; \rho gm_2 \longrightarrow^* \text{def } k : \tau = v_2; \rho gm_2$

$\text{def } k : \tau = v_1; \rho gm_1 \longrightarrow \rho gm_1[k \mapsto v_1]$

$\text{def } k : \tau = v_2; \rho gm_2 \longrightarrow \rho gm_2[k \mapsto v_2]$

Lemma H.6

Part 2

By rule **CE-PGM-DEF**

By rule **CE-PGM-DEF**

rule **CE-PGM-DBETA**

rule **CE-PGM-DBETA**

– Case

PEQ-SPDEF

$$\frac{e_1 =_{ax} e_2 \quad \rho gm_1 =_{ax} \rho gm_2}{\text{spdef } \Delta \Vdash s : \tau = e_1; \rho gm_1 =_{ax} \text{spdef } \Delta \Vdash s : \tau = e_2; \rho gm_2}$$

$e_1 \longrightarrow^* \llbracket e'_1 \rrbracket_{\phi_{v_1}}$

$e_2 \longrightarrow^* \llbracket e'_2 \rrbracket_{\phi_{v_2}}$

$\bullet; \bullet \Vdash \llbracket e'_1 \rrbracket_{\phi_{v_1}} \simeq_{ax} \llbracket e'_2 \rrbracket_{\phi_{v_2}} : \text{Code } \tau$

$\text{spdef } \Delta \Vdash s : \tau = e_1; \rho gm_1 \longrightarrow^* \text{spdef } \Delta \Vdash s : \tau = \llbracket e'_1 \rrbracket_{\phi_{v_1}}; \rho gm_1$

$\text{spdef } \Delta \Vdash s : \tau = e_2; \rho gm_2 \longrightarrow^* \text{spdef } \Delta \Vdash s : \tau = \llbracket e'_2 \rrbracket_{\phi_{v_2}}; \rho gm_2$

Part 2

By rule **CE-PGM-DEF**

By rule **CE-PGM-DEF**

$\text{spdef } \Delta \vdash^{\text{N}} s : \tau = \llbracket e'_1 \rrbracket_{\phi_{v_1}} ; \rho gm_1 \longrightarrow \rho gm_1[s \mapsto [\phi_{v_1}]e'_1]$ rule **CE-PGM-SPBETA**
 $\text{spdef } \Delta \vdash^{\text{N}} s : \tau = \llbracket e'_2 \rrbracket_{\phi_{v_2}} ; \rho gm_2 \longrightarrow \rho gm_2[s \mapsto [\phi_{v_2}]e'_2]$ rule **CE-PGM-SPBETA**
 Lemma H.6

– Case

PEQ-EXPR

$$\frac{e_1 =_{ax} e_2}{e_1 : \tau =_{ax} e_2 : \tau}$$

- * If $e_1 : \tau = v_1 : \tau$ and $e_2 : \tau = v_2 : \tau$, then by rule **PEQ-EXPR** we have $\bullet; \bullet \vdash^{\text{N}} v_1 \simeq_{ax} v_2 : \tau$ and we are done.
- * If $e_1 : \tau = v_1 : \tau$ and e_2 is not a value. Then by progress we have $e_2 \longrightarrow e'_2$. By preservation we have $\bullet; \bullet \vdash^{\text{N}} e'_2 : \tau$. By Part 2, we have $\bullet; \bullet \vdash^{\text{N}} e_2 \simeq_{ax} e'_2 : \tau$. By rule **EQ-TRANS**, we have $\bullet; \bullet \vdash^{\text{N}} e_1 \simeq_{ax} e'_2 : \tau$. Then by rule **PEQ-EXPR** we have $\bullet \vdash v_1 : \tau \simeq_{ax} e'_2 : \tau$ and we are done.
- * The case when e_1 is not a value and e_2 is a value is the same as the previous case.
- * If neither e_1 nor e_2 is a value, then similar as the above case, we have $e_1 \longrightarrow e'_1$ and $\bullet; \bullet \vdash^{\text{N}} e_1 \simeq_{ax} e'_1 : \tau$, and also $e_2 \longrightarrow e'_2$ and $\bullet; \bullet \vdash^{\text{N}} e_2 \simeq_{ax} e'_2 : \tau$. By rule **EQ-TRANS**, we have $\bullet; \bullet \vdash^{\text{N}} e'_1 \simeq_{ax} e'_2 : \tau$. Then by rule **PEQ-EXPR** we have $\bullet \vdash e'_1 : \tau \simeq_{ax} e'_2 : \tau$ and we are done.

– Case

PEQ-SPDEF-AX

$$\frac{\phi \dashv\vdash \Delta \rightsquigarrow \phi'}{\text{spdef } \Delta \vdash^{\text{N}} s : \tau = \llbracket e \rrbracket_{\phi} ; \rho gm =_{ax} \text{spdef } \phi' ; \rho gm[s \mapsto e]}$$

$\phi \longrightarrow^* \phi_v$
 $\text{spdef } \Delta \vdash^{\text{N}} s : \tau = \llbracket e \rrbracket_{\phi} ; \rho gm$
 $\longrightarrow^* \text{spdef } \Delta \vdash^{\text{N}} s : \tau = \llbracket e \rrbracket_{\phi_v} ; \rho gm$
 $\longrightarrow^* \rho gm[s \mapsto [\phi_v]e]$

Part 2

rules **CE-PGM-SPDEF** and **CE-QUOTE**
 rule **CE-PGM-SPBETA**

On the right:

$\phi' \longrightarrow^* \phi'_v$
 $\phi_v \dashv\vdash \Delta \rightsquigarrow \phi'_v$
 $\text{spdef } \phi' ; \rho gm[s \mapsto e]$
 $\longrightarrow^* \text{spdef } \phi'_v ; \rho gm[s \mapsto e]$
 $(\text{dom } (\phi'_v) \# \text{fv } (\rho gm))$
 $\longrightarrow^* \rho gm[s \mapsto [\phi'_v]e]$
 $= \rho gm[s \mapsto [\phi_v]e]$

$$\phi \dashv\vdash \Delta \rightsquigarrow \phi'$$

rule **CE-PGM-SPDEF**
 program well-typed, $\text{dom } (\phi'_v) = \text{dom } (\phi)$
 rule **CE-PGM-SPBETA**
 $\phi_v \dashv\vdash \Delta \rightsquigarrow \phi'_v$

Part 2 By induction on $e_1 =_{ax} e_2$, making use of Lemma H.2.

□

Lemma H.4 ($F \llbracket \cdot \rrbracket \simeq_{ax}$ to $F \llbracket \cdot \rrbracket \simeq_{ctx}$). If $\Theta \vdash \rho gm_1 \simeq_{ax} \rho gm_2$, then $\Theta \vdash \rho gm_1 \simeq_{ctx} \rho gm_2 : \tau$.

PROOF. We prove the direction from ρgm_1 to ρgm_2 , and the other direction is the same.

$\Theta \vdash \rho gm_1 \simeq_{ax} \rho gm_2$ given
 $\frac{S_i, \mathcal{D}_j^{i,j} \in \Theta}{\text{spdef } S_i; \text{def } \mathcal{D}_j^{i,j} ; \rho gm_1 \longrightarrow^* e_1 : \tau}$ assume
 $\rho gm'_1 = \text{spdef } S_i; \text{def } \mathcal{D}_j^{i,j} ; \rho gm_1$ let

3039	$\rho gm'_2 = \overline{\text{spdef } \mathcal{S}_i; \text{def } \mathcal{D}_j}^{i,j}; \rho gm_2$	let
3040	$\bullet \vdash \rho gm'_1 \simeq_{ax} \rho gm'_2$	follows
3041	$\rho gm'_1 \longrightarrow^* e_1 : \tau$	after substitution
3042	$\rho gm'_1 \longrightarrow^* e'_1 : \tau$	Lemma H.3
3043	$e_1 : \tau \longrightarrow^* e'_1 : \tau$	
3044	$\rho gm'_2 \longrightarrow^* e_2 : \tau$	
3045	$\bullet \vdash e'_1 : \tau \simeq_{ax} e_2 : \tau$	above
3046	$\bullet; \bullet \vdash^0 e'_1 \simeq_{ax} e_2 : \tau$	by inversion
3047	$\bullet; \bullet \vdash^0 e_1 \simeq_{ax} e'_1 : \tau$	by Lemma H.2
3048	$\bullet; \bullet \vdash^0 e_1 \simeq_{ax} e_2 : \tau$	by rule EQ-TRANS
3049	$\bullet; \bullet \vdash^0 e_1 \simeq_{ctx} e_2 : \tau$	Theorem H.7

3050

3051

Theorem H.5 ($\lambda \models =_{ax}$ to $F \models \simeq_{ctx}$). If $\rho gm_1 =_{ax} \rho gm_2$, where $\Theta \vdash \rho gm_1 : \sigma \rightsquigarrow \rho gm_1$, and $\Theta \vdash \rho gm_2 : \sigma \rightsquigarrow \rho gm_2$, and $\Theta \rightsquigarrow \Theta$, and $\bullet \vdash \sigma \rightsquigarrow \tau$, then $\Theta \vdash \rho gm_1 \simeq_{ctx} \rho gm_2 : \tau$.

PROOF. Follows by Lemma H.1 and Lemma H.4.

3057 I.2 Elaboration of Source Expressions

Theorem H.7. If $\bullet; \Gamma \vdash^0 e_1 \simeq_{ax} e_2 : \tau$, then $\bullet; \Gamma \vdash^0 e_1 \simeq_{ctx} e_2 : \tau$.

PROOF. We prove the direction from e_1 to e_2 , and the other direction is the same.

3061	$\bullet; \Gamma \vdash^0 e_1 \simeq_{ax} e_2 : \tau$	given
3062	$\bullet; \bullet \vdash^0 \mathbb{C}[e_1] \simeq_{ax} \mathbb{C}[e_2] : \text{Int}$	follows
3063	$\mathbb{C}[e_1] \longrightarrow^* i$	assume
3064	$\bullet; \bullet \vdash^0 \mathbb{C}[e_1] \simeq_{ax} i : \text{Int}$	Corollary H.16
3065	$\bullet; \bullet \vdash^0 \mathbb{C}[e_2] \simeq_{ax} i : \text{Int}$	by rules EQ-TRANS and EQ-SYMM
3066	$\mathbb{C}[e_2] \longrightarrow^* i$	Corollary H.16

3068

3069

3070 I.3 Axiomatic Reduction

Lemma H.8 (Transitivity). If $e_1 \hookrightarrow^* e_2$ and $e_2 \hookrightarrow^* e_3$, then $e_1 \hookrightarrow^* e_3$.

PROOF. By a straightforward induction on $e_1 \hookrightarrow^* e_2$.

Lemma H.9 (Congruence). If $e_1 \hookrightarrow^* e_2$, then $\mathbb{C}[e_1] \hookrightarrow^* \mathbb{C}[e_2]$.

PROOF. By a straightforward induction on $e_1 \hookrightarrow^* e_2$.

Lemma H.10 (\hookrightarrow to $=_{ax}$). Given $\Theta; \Delta \vdash^0 e_1 : \tau$, if $e_1 \hookrightarrow e_2$ then $\Theta; \Delta \vdash^0 e_1 \simeq_{ax} e_2 : \tau$.

PROOF. As \hookrightarrow is the semantics derived from $=_{ax}$, the goal follows straightforwardly. Note the type is preserved according to Preservation (Theorem E.2).

Lemma H.11 (Confluence). Given $\Theta; \Delta \vdash^0 e : \tau$, if $e \hookrightarrow^* e_1$ and $e \hookrightarrow^* e_2$, then there exists e' such that $e_1 \hookrightarrow^* e'$ and $e_2 \hookrightarrow^* e'$.

PROOF. Given $e \hookrightarrow^* e_1$ and $e \hookrightarrow^* e_2$, by Theorem H.24, we have $e \twoheadrightarrow^* e_1$ and $e \twoheadrightarrow^* e_2$. By confluence of \twoheadrightarrow (Theorem H.20), we know there exists an e' such that $e_1 \twoheadrightarrow^* e'$ and $e_2 \twoheadrightarrow^* e'$. By Theorem H.24, we have $e_1 \hookrightarrow^* e'$ and $e_2 \hookrightarrow^* e'$ and we are done.

Theorem H.12 (Church-Rosser). *If $\Theta; \Delta \vdash^{\text{I}} e_1 \simeq_{ax} e_2 : \tau$, then there exists e such that $e_1 \hookrightarrow^* e$ and $e_2 \hookrightarrow^* e$.*

PROOF. By induction on $=_{ax}$.

- For the four axioms, the goal follows directly by choose $e = e_2$, as $e_1 \hookrightarrow e_2$.
- Rule **EQ-REFL**. The goal follows trivially as $e_1 = e_2$.
- Rule **EQ-SYMM**. The goal follows directly from I.H..
- Rule **EQ-TRANS**. There is one e_3 such that $e_1 =_{ax} e_3$ and $e_3 =_{ax} e_2$. By I.H., there exists e' such that $e_1 \hookrightarrow^* e'$ and $e_3 \hookrightarrow^* e'$. Also by I.H., there exists e'' such that $e_3 \hookrightarrow^* e''$ and $e_2 \hookrightarrow^* e''$. By Lemma H.11, there exists e , $e' \hookrightarrow^* e$ and $e'' \hookrightarrow^* e$. Therefore $e_1 \hookrightarrow^* e$ and $e_2 \hookrightarrow^* e$.
- Rule **EQ-CTX**. By I.H., there exists e' such that $e_1 \hookrightarrow^* e'$ and $e_2 \hookrightarrow^* e'$. By rules **CE-AX-C-TRANS** and **CE-AX-CTX**, we have $\mathbb{C}[e_1] \hookrightarrow^* \mathbb{C}[e']$ and $\mathbb{C}[e_2] \hookrightarrow^* \mathbb{C}[e']$.

□

Corollary H.13. *Given $\Theta; \Delta \vdash^{\text{I}} e : \text{Int}$, if $\Theta; \Delta \vdash^{\text{I}} e \simeq_{ax} i : \text{Int}$ then $e \hookrightarrow^* i$.*

PROOF. Follows directly from Theorem H.12.

□

Lemma H.14. *If $e \longrightarrow^* v$, then $e \hookrightarrow^* v$.*

PROOF. The goal can be derived from: if $e_1 \longrightarrow e_2$, then $e_1 \hookrightarrow e_2$. The later can be proved by a straightforward induction on $e_1 \longrightarrow e_2$.

□

Lemma H.15. *Given $\Theta; \Delta \vdash^{\text{I}} e : \tau$, if $e \hookrightarrow^* v$, then $e \longrightarrow^* v'$ for some v' .*

PROOF. Given $e \hookrightarrow^* v$, by Lemma H.28 we know $e \twoheadrightarrow^* v$. By Lemma H.35, we have $e \longrightarrow^* v'$ for some v' .

□

Corollary H.16. *Given $\Theta; \Delta \vdash^{\text{I}} e : \text{Int}$, then we have $\Theta; \Delta \vdash^{\text{I}} e \simeq_{ax} i : \text{Int}$ if and only if $e \longrightarrow^* i$.*

PROOF. From right to left follows directly from Lemma H.14 and Corollary H.13. From left to right:

$\Theta; \Delta \vdash^{\text{I}} e \simeq_{ax} i : \text{Int}$	given
$e \hookrightarrow^* i$	Corollary H.13
for some v_1	Lemma H.15
$e \longrightarrow^* v_1$	
$e \hookrightarrow^* v_1$	Lemma H.14
for some v_2	Lemma H.11
$i \hookrightarrow^* v_2$	above
$v_1 \hookrightarrow^* v_2$	above
$v_2 = i$	by inversion
$v_1 \hookrightarrow^* i$	by substitution
$v_1 = i$	follows

□

1.4 Church-Rosser

Lemma H.17 (Substitution).

- If $e_1 \twoheadrightarrow e_2$, and $e_3 \twoheadrightarrow e_4$, then $e_1[x \mapsto e_3] \twoheadrightarrow e_2[x \mapsto e_4]$.
- If $e_1 \twoheadrightarrow e_2$, then $e_1[a \mapsto \tau] \twoheadrightarrow e_2[a \mapsto \tau]$.
- If $e_1 \twoheadrightarrow e_2$, and $e_3 \twoheadrightarrow e_4$, then $e_1[s \mapsto e_3] \twoheadrightarrow e_2[s \mapsto e_4]$.

This can be proved using the similar techniques as the substitution lemma for parallel reduction with complexity (Lemma H.30). \square

Lemma H.18 (Diamond Lemma). *Given $\Theta; \Delta \vdash^{\text{p}} e : \tau$, if $e \hookrightarrow e_1$, and $e \hookrightarrow e_2$, then there exists e' such that $e_1 \hookrightarrow e'$ and $e_2 \hookrightarrow e'$.*

PROOF. Suppose $e \hookrightarrow e_3$ (Lemma H.25). Let $e' = e_3$. By Lemma H.26, we know $e_1 \hookrightarrow e_3$ and $e_2 \hookrightarrow e_3$. \square

Lemma H.19 (Strip Lemma). *Given $\Theta; \Delta \vdash^{\text{p}} e : \tau$, if $e \hookrightarrow e_1$, and $e \hookrightarrow^* e_2$, then there exists e' such that $e_1 \hookrightarrow^* e'$ and $e_2 \hookrightarrow e'$.*

PROOF. By induction on $e \hookrightarrow^* e_2$.

- Case $e = e_2$ and $e \hookrightarrow^* e$. Let $e' = e_1$ and we are done.
- Case $e \hookrightarrow e_3$ and $e_3 \hookrightarrow^* e_2$. By Lemma H.18, there exists e_4 such that $e_1 \hookrightarrow e_4$ and $e_3 \hookrightarrow e_4$. By I.H., there exists e' such that $e_4 \hookrightarrow^* e'$ and $e_2 \hookrightarrow e'$. By $e_1 \hookrightarrow e_4$ and $e_4 \hookrightarrow^* e'$, we have $e_1 \hookrightarrow^* e'$ so we are done.

Theorem H.20 (Confluence of \hookrightarrow^*). *Given $\Theta; \Delta \vdash^{\text{p}} e : \tau$, if $e \hookrightarrow^* e_1$, and $e \hookrightarrow^* e_2$, then there exists e' such that $e_1 \hookrightarrow^* e'$ and $e_2 \hookrightarrow^* e'$.*

PROOF. By induction on $e \hookrightarrow^* e_1$.

- Case $e = e_1$ and $e \hookrightarrow^* e$. Let $e' = e_2$ and we are done.
- Case $e \hookrightarrow e_3$ and $e_3 \hookrightarrow^* e_1$. By lemma H.19, there exists e_4 such that $e_3 \hookrightarrow^* e_4$ and $e_2 \hookrightarrow e_4$. By I.H., there exists e' such that $e_1 \hookrightarrow^* e'$ and $e_4 \hookrightarrow^* e'$. By $e_2 \hookrightarrow e_4$ and $e_4 \hookrightarrow^* e'$ we have $e_2 \hookrightarrow^* e'$ so we are done.

1.5 Parallel Reduction

Lemma H.21 (Reflexivity). $e \hookrightarrow e$.

PROOF. By a straightforward induction on e . \square

Lemma H.22 (\hookrightarrow simulates \hookrightarrow^*). *If $e_1 \hookrightarrow e_2$, then $e_1 \hookrightarrow^* e_2$.*

PROOF. By induction on $e_1 \hookrightarrow e_2$. The key observation is that in $e_1 \hookrightarrow e_2$ fewer subterms are reduced, so we employ Lemma H.21 to fill in necessary identity reductions to obtain $e_1 \hookrightarrow^* e_2$. \square

Lemma H.23 (\hookrightarrow^* simulates \hookrightarrow). *If $e_1 \hookrightarrow^* e_2$, then $e_1 \hookrightarrow e_2$.*

PROOF. By induction on $e_1 \hookrightarrow^* e_2$, making use of Lemma H.8 and Lemma H.9.

- Cases for rules **CE-AX-PA-VAR**, **CE-AX-PA-SVAR**, and **CE-AX-PA-KVAR** follow directly by rule **CE-AX-C-REFL**.
- Rule **CE-AX-PA-ABS** where $\lambda x : \tau. e_1 \hookrightarrow \lambda x : \tau. e_2$. By I.H., we have $e_1 \hookrightarrow^* e_2$. By Lemma H.9, we have $\lambda x : \tau. e_1 \hookrightarrow^* \lambda x : \tau. e_2$.
- The case for rule **CE-AX-PA-TABS** is similar as the previous one.
- Rule **CE-AX-PA-APP** where $e_1 e_2 \hookrightarrow e_3 e_4$. By I.H., we have $e_1 \hookrightarrow^* e_3$, and $e_2 \hookrightarrow^* e_4$. By Lemma H.9, we have $e_1 e_2 \hookrightarrow^* e_3 e_2$. Also by Lemma H.9, we have $e_3 e_2 \hookrightarrow^* e_3 e_4$. Thus by Lemma H.8, we have $e_1 e_2 \hookrightarrow^* e_3 e_4$.
- The case for rule **CE-AX-PA-TAPP** is similar as the previous one.

- Rule **CE-AX-PA-BETA** where $(\lambda x : \tau.e_1) e_2 \hookrightarrow^* e_3[x \mapsto e_4]$. By I.H., we have $e_1 \hookrightarrow^* e_3$ and $e_2 \hookrightarrow^* e_4$. So by Lemma H.9, we have $(\lambda x : \tau.e_1) e_2 \hookrightarrow^* (\lambda x : \tau.e_3) e_2$, and also $(\lambda x : \tau.e_3) e_2 \hookrightarrow^* (\lambda x : \tau.e_3) e_4$. Further $(\lambda x : \tau.e_3) e_4 \hookrightarrow^* e_3[x \mapsto e_4]$ by rules **CE-AX-C-REFL** and **CE-AX-BETA**. So by Lemma H.8, we have $(\lambda x : \tau.e_1) e_2 \hookrightarrow^* e_3[x \mapsto e_4]$.
- The cases for rule **CE-AX-PA-TBETA** is similar as the previous case.
- Rule **CE-AX-PA-SPLICEQUOTE** where $\llbracket e \rrbracket_{\Delta_i \vdash^i s_i; \tau_i = e_i} \hookrightarrow^* \llbracket e' [\overline{s_i \mapsto e''^i}] \rrbracket_{\phi_i}^i$. By I.H., we have $e \hookrightarrow^* e'$, and $e_i \hookrightarrow^* e''^i$ for expressions going through the first branch, and $e_i \hookrightarrow^* e_i'$ for expressions going through the second branch. Then through Lemma H.9, Lemma H.8, and rule **CE-AX-SPLICEQUOTE**, we can get $\llbracket e \rrbracket_{\Delta_i \vdash^i s_i; \tau_i = e_i} \hookrightarrow^* \llbracket e' [\overline{s_i \mapsto e''^i}] \rrbracket_{\phi_i}^i$.
- Rule **CE-AX-PA-QUOTE SPLICE**, where $\llbracket s \rrbracket_{\bullet \vdash^i s; \tau = e_1} \hookrightarrow^* e_2$. By I.H., we have $e_1 \hookrightarrow^* e_2$. By Lemma H.9, we have $\llbracket s \rrbracket_{\bullet \vdash^i s; \tau = e_1} \hookrightarrow^* \llbracket s \rrbracket_{\bullet \vdash^i s; \tau = e_2}$. Then, by rule **CE-AX-QUOTE SPLICE**, we have $\llbracket s \rrbracket_{\bullet \vdash^i s; \tau = e_2} \hookrightarrow^* e_2$. Therefore by Lemma H.8, we have $\llbracket s \rrbracket_{\bullet \vdash^i s; \tau = e_1} \hookrightarrow^* e_2$.

□

Theorem H.24 (Equivalence of Parallel Reduction and Axiomatic Semantics). $e_1 \hookrightarrow^* e_2$ if and only if $e_1 \hookrightarrow^* e_2$.

PROOF. Follows directly by Lemma H.22 and Lemma H.23.

□

1.6 Complete Development

Lemma H.25 (\hookrightarrow^* exists). For any e , there exists e' such that $e \hookrightarrow^* e'$.

PROOF. By straightforward induction on e . Of particular interest is when e is a quotation. Then depending on its shape we can apply rule **CE-AX-CP-SPLICEQUOTE** or rule **CE-AX-CP-QUOTE SPLICE** correspondingly.

□

Lemma H.26 (\hookrightarrow^* closes \hookrightarrow). Given $\Theta; \Delta \vdash^i e : \tau$, if $e \hookrightarrow^* e_1$, and $e \hookrightarrow e_2$, then $e_2 \hookrightarrow e_1$.

PROOF. By induction on $e \hookrightarrow^* e_1$.

- Rule **CE-AX-CP-VAR**. Then $e = e_1 = e_2 = x$. The goal follows by rule **CE-AX-PA-VAR**.
- Cases for rules **CE-AX-CP-SVAR** and **CE-AX-CP-KVAR** are similar as the previous case.
- Rule **CE-AX-CP-ABS** where $e = \lambda x : \tau.e'$ and $e_1 = \lambda x : \tau.e'_1$

$$\begin{array}{l|l}
 \lambda x : \tau.e' \hookrightarrow^* \lambda x : \tau.e'_1 & \text{given} \\
 e' \hookrightarrow^* e'_1 & \text{inversion (rule CE-AX-CP-ABS)} \\
 e_2 = \lambda x : \tau.e'_2 & \text{inversion (rule CE-AX-PA-ABS)} \\
 e' \hookrightarrow e'_2 & \text{above} \\
 e'_2 \hookrightarrow e'_1 & \text{I.H.} \\
 \lambda x : \tau.e'_2 \hookrightarrow \lambda x : \tau.e'_1 & \text{rule CE-AX-PA-ABS}
 \end{array}$$

- The case for rule **CE-AX-CP-TABS** is similar as the previous case.
- Rule **CE-AX-CP-APP** where $e = e_3 e_4$ and $e_1 = e_5 e_6$.

$$\begin{array}{l|l}
 e_3 e_4 \hookrightarrow^* e_5 e_6 & \text{given} \\
 e_3 \hookrightarrow^* e_5 & \text{inversion (rule CE-AX-CP-APP)} \\
 e_4 \hookrightarrow^* e_6 & \text{above} \\
 e_3 \neq \lambda x : \tau.e' & \text{above} \\
 e_2 = e_7 e_8 & \text{inversion (rule CE-AX-PA-APP)} \\
 e_3 \hookrightarrow e_7 & \text{above} \\
 e_4 \hookrightarrow e_8 & \text{above}
 \end{array}$$

3235 $e_7 \hookrightarrow e_5$ | I.H.
 3236 $e_8 \hookrightarrow e_6$ | I.H.
 3237 $e_7 e_8 \hookrightarrow e_5 e_6$ | rule **CE-AX-PA-APP**

- 3238
- The cases for rule **CE-AX-CP-TAPP** is similar as the previous case.
 - Rule **CE-AX-CP-BETA**, where $e = (\lambda x : \tau. e_3) e_4$, and $e_1 = e_5[x \mapsto e_6]$.

3241 $(\lambda x : \tau. e_3) e_4 \hookrightarrow e_5[x \mapsto e_6]$ | given
 3242 $e_3 \hookrightarrow e_5$ | inversion (rule **CE-AX-CP-BETA**)
 3243 $e_4 \hookrightarrow e_6$ | above

3245 There are two subcases for the derivation $e \hookrightarrow e_2$.

3246 (1) Rule **CE-AX-PA-BETA**

3248 $e_2 = e_7[x \mapsto e_8]$ | inversion (rule **CE-AX-PA-BETA**)
 3249 $e_3 \hookrightarrow e_7$ | above
 3250 $e_4 \hookrightarrow e_8$ | above
 3251 $e_7 \hookrightarrow e_5$ | I.H.
 3252 $e_8 \hookrightarrow e_6$ | I.H.
 3253 $(\lambda x : \tau. e_3) e_4 \hookrightarrow e_7[x \mapsto e_8]$ | rule **CE-AX-PA-BETA**
 3254 $e_7[x \mapsto e_8] \hookrightarrow e_5[x \mapsto e_6]$ | Lemma H.17

3255 (2) Rule **CE-AX-PA-APP**

3257 $e_2 = e_7 e_8$ | inversion (rule **CE-AX-PA-BETA**)
 3258 $e_3 \hookrightarrow e_7$ | above
 3259 $e_4 \hookrightarrow e_8$ | above
 3260 $e_7 \hookrightarrow e_5$ | I.H.
 3261 $e_8 \hookrightarrow e_6$ | I.H.
 3262 $e_7 e_8 \hookrightarrow e_5[x \mapsto e_6]$ | rule **CE-AX-PA-BETA**

- 3264
- The cases for rule **CE-AX-CP-TBETA** is similar as the previous case.
 - e_1 goes through rule **CE-AX-CP-SPLICEQUOTE** and e_2 goes through rule **CE-AX-PA-QUOTE SPLICE**. Impossible case as rule **CE-AX-CP-SPLICEQUOTE** rules out the form of e that rule **CE-AX-PA-QUOTE SPLICE** can be applied.
 - Rule **CE-AX-CP-SPLICEQUOTE** where $e = \llbracket e_3 \rrbracket_{\Delta_i \vdash^{P_i} s_i : \tau_i = e_i^i}$, and e_2 also goes through rule **CE-AX-PA-SPLICEQUOTE**.

3270 We have $e_1 = \llbracket e'_1[\overline{s_i \mapsto e_{1i}^i}] \rrbracket_{\phi_{1i}^i}$ and $e_3 \hookrightarrow e'_1$. Also, $e_2 = \llbracket e'_2[\overline{s_i \mapsto e_{2i}^i}] \rrbracket_{\phi_{2i}^i}$ and $e_3 \hookrightarrow e'_2$.

3271 By I.H., $e'_2 \hookrightarrow e'_1$.

3272 In this case, we aim to show that by one step of rule **CE-AX-PA-SPLICEQUOTE**, e_2 can get the same set of substitutions and quotation environments as e_1 , then the goal can be established by the substitution lemma (Lemma H.17).

3273 We know that during the derivation, for each $\Delta_i \vdash^{P_i} s_i : \tau_i = e_i$, it may go through the first branch in rule **CE-AX-CP-SPLICEQUOTE**, or the second branch.

3274 Namely, for some e_i , $e_i \hookrightarrow \llbracket e''_i \rrbracket_{\phi_i^i} \wedge \phi_i^i \vdash \Delta_i \rightsquigarrow \phi_i^{i'}$; for some e_i , $\Delta_i \vdash^{P_i} s_i : \tau_i = e_i^i$.

3275 For each $\Delta_i \vdash^{P_i} s_i : \tau_i = e_i$, we discuss the cases of its reduction in e_1 and in e_2 . There are four subcases.

- 3276 (1) e_1 goes through the first branch in rule **CE-AX-CP-SPLICEQUOTE** and rule **CE-AX-PA-SPLICEQUOTE** respectively.

3283

Then for e_1 , it has applied the substitution $s_i \mapsto e_i''$, and left the splice environment ϕ_i'' .
 Then for e_2 , it has applied the substitution $s_i \mapsto e_i'''$, and left the splice environment ϕ_i''' .

$e_i \hookrightarrow \llbracket e_i'' \rrbracket_{\phi_i''}$	inversion (rule CE-AX-CP-SPLICEQUOTE)
$\phi_i'' \vdash \Delta_i \leadsto \phi_i''$	above
$e_i \hookrightarrow \llbracket e_i''' \rrbracket_{\phi_i'''}$	inversion (rule CE-AX-PA-SPLICEQUOTE)
$\phi_i''' \vdash \Delta_i \leadsto \phi_i'''$	above
$\llbracket e_i''' \rrbracket_{\phi_i'''} \hookrightarrow \llbracket e_i'' \rrbracket_{\phi_i''}$	I.H.

- Suppose $\llbracket e_i''' \rrbracket_{\phi_i'''} \hookrightarrow \llbracket e_i'' \rrbracket_{\phi_i''}$ has gone through rule **CE-AX-PA-SPLICEQUOTE**.
 Then in the goal when applying rule **CE-AX-PA-SPLICEQUOTE** we will choose the same branch for each splice variable in ϕ_i''' as we did for it in ϕ_i'' . Then effectively we can generate a bunch of substitutions that when applied turns the substitution $s_i \mapsto e_i'''$ into $s_i \mapsto e_i''$. This also leaves us with the environment ϕ_i'' .
 - On the other hand, suppose $\llbracket e_i''' \rrbracket_{\phi_i'''} \hookrightarrow \llbracket e_i'' \rrbracket_{\phi_i''}$ has gone through rule **CE-AX-PA-QUOTE SPLICE**.
 Then in this case we have $\llbracket e_i''' \rrbracket_{\phi_i'''} = \llbracket s_j \rrbracket_{\phi_i'''} \bullet \vdash^{n_i} s_j : \tau = e_i''''$, and $e_i'''' \hookrightarrow \llbracket e_i'' \rrbracket_{\phi_i''}$.
 Namely, e_2 has made a substitution $s_i \mapsto s_j$ and left the splice environment $\Delta_i \vdash^{n_i} s_j : \tau = e_i''''$.
 Then in the goal when applying rule **CE-AX-PA-SPLICEQUOTE** we will choose the first branch for s_j with $e_i'''' \hookrightarrow \llbracket e_i'' \rrbracket_{\phi_i''}$.
 This generates a substitution $s_j \mapsto e_i''$, which when applied turns the substitution $s_i \mapsto s_j$ into $s_i \mapsto e_i''$, and this also leaves us ϕ_i'' .
- (2) e_i goes through the first branch in rule **CE-AX-CP-SPLICEQUOTE**, and goes through the second branch in rule **CE-AX-PA-SPLICEQUOTE**.
 Then for e_1 , it has applied the substitution $s_i \mapsto e_i''$, and left the splice environment ϕ_i'' .
 Then for e_2 , it has applied no substitution, but left the splice environment $\Delta_i \vdash^{n_i} s_i : \tau_i = e_i'''$.

$e_i \hookrightarrow \llbracket e_i'' \rrbracket_{\phi_i''}$	inversion (rule CE-AX-CP-SPLICEQUOTE)
$\phi_i'' \vdash \Delta_i \leadsto \phi_i''$	above
$e_i \hookrightarrow e_i'''$	inversion (rule CE-AX-PA-SPLICEQUOTE)
$e_i''' \hookrightarrow \llbracket e_i'' \rrbracket_{\phi_i''}$	I.H.

Then by applying the first branch in rule **CE-AX-PA-SPLICEQUOTE**, we can obtain the substitutions $s_i \mapsto e_i''$ and the splice environment ϕ_i'' .

- (3) e_i goes through the second branch in rule **CE-AX-CP-SPLICEQUOTE** and rule **CE-AX-PA-SPLICEQUOTE** respectively.
 Then for e_1 , it has applied no substitution, but left the splice environment $\Delta_i \vdash^{n_i} s_i : \tau_i = e_i''$.
 Then for e_2 , it has applied no substitution, but left the splice environment $\Delta_i \vdash^{n_i} s_i : \tau_i = e_i'''$.

$e_i \hookrightarrow e_i''$	inversion (rule CE-AX-CP-SPLICEQUOTE)
e_i'' is not a quotation	above
$e_i \hookrightarrow e_i'''$	inversion (rule CE-AX-PA-SPLICEQUOTE)
$e_i''' \hookrightarrow e_i''$	I.H.

Then by applying the second branch in rule **CE-AX-PA-SPLICEQUOTE**, we can transform from $\Delta_i \vdash^{n_i} s_i : \tau_i = e_i'''$ to $\Delta_i \vdash^{n_i} s_i : \tau_i = e_i''$.

- (4) e_i goes through the second branch in rule **CE-AX-CP-SPLICEQUOTE**, and goes through the first branch in rule **CE-AX-PA-SPLICEQUOTE**.
 Then for e_1 , it has applied no substitution, but left the splice environment $\Delta_i \vdash^{n_i} s_i : \tau_i = e_i''$.
 Then for e_2 , it has applied the substitution $s_i \mapsto e_i'''$, and left the splice environment ϕ_i''' .

3333	$e_i \hookrightarrow e_i''$	inversion (rule CE-AX-CP-SPLICEQUOTE)
3334	e_i'' is not a quotation	above
3335	$e_i \hookrightarrow \llbracket e_i''' \rrbracket_{\phi_i'''}$	inversion (rule CE-AX-PA-SPLICEQUOTE)
3336	$\phi_i''' \vdash \Delta_i \leadsto \phi_i''''$	above
3337	$\llbracket e_i''' \rrbracket_{\phi_i'''} \hookrightarrow e_i''$	I.H.

3338
3339 By inversion we know $\llbracket e_i''' \rrbracket_{\phi_i'''} \hookrightarrow e_i''$ goes through rule **CE-AX-QUOTE SPLICE**. Therefore,
3340 $e_i''' = s'$. As the expression is well-typed, so s' must have the same level and type as s_i . We
3341 then rewrite the above reasoning as:

3342	$e_i \hookrightarrow e_i''$	inversion (rule CE-AX-CP-SPLICEQUOTE)
3343	e_i'' is not a quotation	above
3344	$e_i \hookrightarrow \llbracket s' \rrbracket_{\bullet \vdash^{\tau_i} s' : \tau_i = e_i'}$	inversion (rule CE-AX-PA-SPLICEQUOTE)
3345	$\bullet \vdash^{\tau_i} s' : \tau_i = e_i' \vdash \Delta_i \leadsto \Delta_i \vdash^{\tau_i} s' : \tau_i = e_i'$	above
3346	$\llbracket s' \rrbracket_{\bullet \vdash^{\tau_i} s' : \tau_i = e_i'} \hookrightarrow e_i''$	I.H.
3347	$e_i' \hookrightarrow e_i''$	by inversion (rule CE-AX-QUOTE SPLICE)

3348
3349 Namely for e_2 , it has applied the substitution $s_i \mapsto s'$, and left the splice environment
3350 $\Delta_i \vdash^{\tau_i} s' : \tau_i = e_i'$.

3351 By α renaming, this is equivalent to that for e_2 , it has applied no substitution, and left the
3352 splice environment $\Delta_i \vdash^{\tau_i} s : \tau_i = e_i'$.

3353 Then by applying rule **CE-AX-PA-SPLICEQUOTE**, we can transform from $\Delta_i \vdash^{\tau_i} s_i : \tau_i = e_i'$ to
3354 $\Delta_i \vdash^{\tau_i} s_i : \tau_i = e_i'$.

- 3355 • Rule **CE-AX-CP-QUOTE SPLICE** where $e = \llbracket s \rrbracket_{\bullet \vdash^{\tau} s : \tau = e_3}$.

3357	$e_1 = e_4$	inversion (rule CE-AX-CP-QUOTE SPLICE)
3358	$e_3 \hookrightarrow e_4$	above

3359
3360 There are two subcases for the derivation $e \hookrightarrow e_2$.

- 3361 (1) Rule **CE-AX-PA-QUOTE SPLICE**

3362	$e_2 = e_5$	inversion (rule CE-AX-PA-QUOTE SPLICE)
3363	$e_3 \hookrightarrow e_5$	above
3364	$e_5 \hookrightarrow e_4$	I.H.

- 3365
3366 (2) Rule **CE-AX-PA-SPLICEQUOTE**. There are further two subcases.

3367 – The first branch

3368	$e_2 = \llbracket s[s \mapsto e_5] \rrbracket_{\phi} = \llbracket e_5 \rrbracket_{\phi}$	inversion (rule CE-AX-PA-SPLICEQUOTE)
3369	$e_3 \hookrightarrow \llbracket e_5 \rrbracket_{\phi}$	above
3370	$\llbracket e_5 \rrbracket_{\phi} \hookrightarrow e_4$	I.H.

3371
3372 – the second branch

3373	$e_2 = \llbracket s \rrbracket_{\bullet \vdash^{\tau} s : \tau = e_5}$	inversion (rule CE-AX-PA-SPLICEQUOTE)
3374	$e_3 \hookrightarrow e_5$	above
3375	$e_5 \hookrightarrow e_4$	I.H.
3376	$\llbracket s \rrbracket_{\bullet \vdash^{\tau} s : \tau = e_5} \hookrightarrow e_4$	rule CE-AX-PA-QUOTE SPLICE

3377
3378
3379
3380 □

3381

1.7 Parallel Reduction with Complexity

Lemma H.27 (Reflexivity). $e \longleftrightarrow e$.

PROOF. By a straightforward induction on e . □

Lemma H.28 (\longleftrightarrow simulates \hookrightarrow). If $e_1 \hookrightarrow e_2$, then $e_1 \longleftrightarrow e_2$.

PROOF. By induction on $e_1 \hookrightarrow e_2$. Just like for Lemma H.22, the key observation is that in $e_1 \hookrightarrow e_2$ fewer subterms are reduced, so we employ Lemma H.27 to fill in necessary identity reductions to obtain $e_1 \longleftrightarrow e_2$. □

Lemma H.29 (\hookrightarrow^* simulates \longleftrightarrow). If $e_1 \longleftrightarrow e_2$, then $e_1 \hookrightarrow^* e_2$.

PROOF. By induction on $e_1 \longleftrightarrow e_2$, making use of Lemma H.8 and Lemma H.9. The proof is the same as Lemma H.23. □

Lemma H.30 (Substitution).

- If $e_1 \xrightarrow{N_1} e_2$, and $e_3 \xrightarrow{N_2} e_4$, then there exists M , such that $e_1[x \mapsto e_3] \xrightarrow{M} e_2[x \mapsto e_4]$, where $M \leq N_1 + \#(x, e_2) * N_2$.
- If $e_1 \xrightarrow{N} e_2$, then $e_1[a \mapsto \tau] \xrightarrow{N} e_2[a \mapsto \tau]$.

PROOF. Part 1 By induction on the size of e_1 , then we do a case analysis on $e_1 \xrightarrow{N_1} e_2$.

- rule **CE-AX-PPA-LIT**, where $e_1 = i = e_2$, and $N_1 = 0$.
So $i[x \mapsto e_3] = i$, and $e_2[x \mapsto e_4] = i$.
The goal follows by rule **CE-AX-PPA-LIT** where $M = 0$.
- rule **CE-AX-PPA-VAR**, where $e_1 = x = e_2$, and $N_1 = 0$.
So $N_1 + \#(x, e_2) * N_2 = 0 + 1 * N_2 = N_2$.
The goal follows by letting $M = N_2$.
- The cases for rule **CE-AX-PPA-VAR** where $e_1 = y \neq x$, and for rule **CE-AX-PPA-SVAR**, for rule **CE-AX-PPA-KVAR** are similar as the case for rule **CE-AX-PPA-LIT**.
- rule **CE-AX-PPA-ABS**, where $e_1 = \lambda y : \tau. e_5$, $e_2 = \lambda y : \tau. e_6$, and $e_5 \xrightarrow{N_1} e_6$.
By I.H., $e_5[x \mapsto e_3] \xrightarrow{M} e_6[x \mapsto e_4]$, where $M \leq N_1 + \#(x, e_6) * N_2$.
By rule **CE-AX-PPA-ABS**, we have $(\lambda y : \tau. e_5)[x \mapsto e_3] \xrightarrow{M} (\lambda y : \tau. e_6)[x \mapsto e_4]$. As $\#(x, e_6) = \#(x, \lambda y : \tau. e_6)$, we have $M \leq N_1 + \#(x, e_6) * N_2$.
- The case for rule **CE-AX-PPA-TABS** is similar as the previous case.
- rule **CE-AX-PPA-APP**, where $e_1 = e_5 e_6$, $e_2 = e_7 e_8$, and $e_5 \xrightarrow{N_3} e_7$, $e_6 \xrightarrow{N_4} e_8$, and $N_1 = N_3 + N_4$.
By I.H., $e_5[x \mapsto e_3] \xrightarrow{M_1} e_7[x \mapsto e_4]$, and $M_1 \leq N_3 + \#(x, e_7) * N_2$.
Also by I.H., $e_6[x \mapsto e_3] \xrightarrow{M_2} e_8[x \mapsto e_4]$ and $M_2 \leq N_4 + \#(x, e_8) * N_2$.
So by rule **CE-AX-PPA-APP**, we have $(e_5 e_6)[x \mapsto e_3] \xrightarrow{M_1 + M_2} (e_7 e_8)[x \mapsto e_4]$. Let $M = M_1 + M_2$, we have $M \leq N_3 + N_4 + \#(x, e_7 e_8) * N_2$.
- The case for rule **CE-AX-PPA-TAPP** is similar as the previous case.
- rule **CE-AX-PPA-BETA**, where $e_1 = (\lambda y : \tau. e_5) e_6$, $e_2 = e_7[y \mapsto e_8]$, and $e_5 \xrightarrow{N_3} e_7$, $e_6 \xrightarrow{N_4} e_8$, and $N_1 = N_3 + \#(y, e_7) * N_4 + 1$.
By I.H., $e_5[x \mapsto e_3] \xrightarrow{M_1} e_7[x \mapsto e_4]$, where $M_1 \leq N_3 + \#(x, e_7) * N_2$.
Also by I.H., $e_6[x \mapsto e_3] \xrightarrow{M_2} e_8[x \mapsto e_4]$ and $M_2 \leq N_4 + \#(x, e_8) * N_2$.

By rule **CE-AX-PPA-BETA**, $(\lambda y : \tau. e_5[x \mapsto e_3]) (e_6[x \mapsto e_3]) \xrightarrow{M_1 + \#(y, e_7[x \mapsto e_4]) * M_2 + 1} (e_7[x \mapsto e_4])[y \mapsto e_8[x \mapsto e_4]]$

With y fresh (alpha-renaming), we have $\#(y, e_7[x \mapsto e_4]) = \#(y, e_7)$

By substitution, we have $(e_7[y \mapsto e_8])[x \mapsto e_4] = (e_7[x \mapsto e_4])[y \mapsto e_8[x \mapsto e_4]]$.

Namely, $((\lambda y : \tau. e_5) e_6)[x \mapsto e_3] \xrightarrow{M_1 + \#(y, e_7) * M_2 + 1} (e_7[y \mapsto e_8])[x \mapsto e_4]$.

$$\begin{aligned} & M_1 + \#(y, e_7) * M_2 + 1 \\ & \leq N_3 + \#(x, e_7) * N_2 + \#(y, e_7) * (N_4 + \#(x, e_8) * N_2) + 1 \\ & = N_3 + \#(y, e_7) * N_4 + 1 + (\#(x, e_7) + \#(y, e_7) * \#(x, e_8)) * N_2 \\ & = N_3 + \#(y, e_7) * N_4 + 1 + \#(x, e_7[y \mapsto e_8]) * N_2 \end{aligned}$$

– The case for rule **CE-AX-PPA-TBETA** is similar as the previous case.

– rule **CE-AX-PPA-SPLICEQUOTE**, where $e_1 = \llbracket e \rrbracket_{\Delta_i \vdash^{\mu_i} s_i : \tau_i = e_i}^i$, $e_2 = \llbracket e' [s_i \mapsto e_i'''] \rrbracket_{\phi_i}^i$, and $e \xrightarrow{N} e'$. and $N_1 = N + \overline{\#(s_i, e')} * \overline{N_i}^i + \overline{M_i}^i + \overline{L_i}^i + 1$.

By I.H., $e[x \mapsto e_3] \xrightarrow{M_1} e'[x \mapsto e_4]$, where $M_1 \leq N + \#(x, e') * N_2$.

(1) If $\phi_i = \phi_i''$, where $e_i = \llbracket e_i'' \rrbracket_{\phi_i}^i$, $e_i'' \xrightarrow{N_i'} e_i'''$, $\phi_i \xrightarrow{N_i'} \phi_i'$, and $\phi_i' \vdash \Delta_i \leadsto \phi_i''$.

By I.H., $e_i''[x \mapsto e_3] \xrightarrow{N_i'} e_i'''[x \mapsto e_4]$ and $N_i' \leq N_i + \#(x, e_i''') * N_2$.

By Part 2, $\phi_i[x \mapsto e_3] \xrightarrow{L_i'} \phi_i'[x \mapsto e_4]$ and $L_i' \leq L_i + \#(x, \phi_i') * N_2$.

(2) If $\phi_i = \Delta_i \vdash^{\mu_i} s_i : \tau_i = e_i'$, where $e_i \xrightarrow{M_i} e_i'$.

By I.H., $e_i[x \mapsto e_3] \xrightarrow{M_i} e_i'[x \mapsto e_4]$ and $M_i' \leq M_i + \#(x, e_i') * N_2$.

By substitution, we have $(e' [s_i \mapsto e_i''] [x \mapsto e_4]) = ((e' [x \mapsto e_4]) [s_i \mapsto e_i'' [x \mapsto e_4]])$.

By rule **CE-AX-PPA-SPLICEQUOTE**, $\llbracket e[x \mapsto e_3] \rrbracket_{\Delta_i \vdash^{\mu_i} s_i : \tau_i = e_i}^i \xrightarrow{M} \llbracket (e' [s_i \mapsto e_i''] [x \mapsto e_4]) \rrbracket_{\phi_i[x \mapsto e_4]}^i$,

where $M = M_1 + \#(s_i, e' [x \mapsto e_4]) * \overline{N_i'}^i + \overline{M_i'}^i + \overline{L_i'}^i + 1$

With s_i fresh (alpha-renaming), we have $\#(s_i, e' [x \mapsto e_4]) = \#(s_i, e')$.

Also, $\#(x, \phi_i') = \#(x, \phi_i'')$.

M

$$\begin{aligned} & = M_1 + \#(s_i, e' [x \mapsto e_4]) * \overline{N_i'}^i + \overline{M_i'}^i + \overline{L_i'}^i + 1 \\ & = M_1 + \#(s_i, e') * \overline{N_i'}^i + \overline{M_i'}^i + \overline{L_i'}^i + 1 \\ & \leq N + \#(x, e') * N_2 + \#(s_i, e') * (N_i + \#(x, e_i''') * N_2) + \overline{M_i}^i + \#(x, e_i') * N_2 + \overline{L_i}^i + \#(x, \phi_i') * N_2 + 1 \\ & = N + \#(s_i, e') * \overline{N_i}^i + \overline{M_i}^i + \overline{L_i}^i + 1 + (\#(x, e') + \#(s_i, e') * \#(x, e_i''') + \#(x, e_i') + \#(x, \phi_i'')) * N_2 \\ & = N + \#(s_i, e') * \overline{N_i}^i + \overline{M_i}^i + \overline{L_i}^i + 1 + (\#(x, \llbracket e' [s_i \mapsto e_i'''] \rrbracket_{\phi_i}^i)) * N_2 \end{aligned}$$

– rule **CE-AX-PPA-QUOTE SPLICE**, where $e_1 = \llbracket s \rrbracket_{\bullet \vdash^{\mu} s : \tau = e_5}^{\mu}$, and $e_5 \xrightarrow{N} e_2$. and $N_1 = N + 1$.

By I.H., $e_5[x \mapsto e_3] \xrightarrow{M_1} e_2[x \mapsto e_4]$, where $M_1 \leq N + \#(x, e_2) * N_2$.

By rule **CE-AX-PPA-QUOTE SPLICE**, we have $\llbracket s \rrbracket_{\bullet \vdash^{\mu} s : \tau = e_5}^{\mu} [x \mapsto e_4] \xrightarrow{M_1 + 1} e_2[x \mapsto e_4]$.

$M_1 + 1 \leq N + \#(x, e_2) * N_2 + 1$.

Part 2 By a straightforward induction on the size of e_1 and a case analysis on $e_1 \xrightarrow{N} e_2$.

□

Lemma H.31 (Monotonicity). *If $v \hookrightarrow e$, then e is also a value.*

PROOF. By a straightforward induction on $v \hookrightarrow e$. □

Lemma H.32 (Transition). *If $e \hookrightarrow v$, then there exists v_2 , such that $e \rightarrow^* v_2$, and $v_2 \hookrightarrow v$.*

PROOF. By induction on the derivation complexity of $e \hookrightarrow v$.

- Case rule **CE-AX-PPA-LIT**. The goal follows directly by letting $v_2 = e$. The cases for rule **CE-AX-PPA-ABS**, rule **CE-AX-PPA-TABS** are similar.
- The cases for rule **CE-AX-PPA-VAR**, rule **CE-AX-PPA-SVAR**, rule **CE-AX-PPA-KVAR**, rule **CE-AX-PPA-APP**, rule **CE-AX-PPA-TAPP** are impossible cases as they don't result into values.
- Case rule **CE-AX-PPA-BETA**, where $e = (\lambda x : \tau. e_1) e_2$, $e_1 \hookrightarrow e_3$, $e_2 \hookrightarrow e_4$, and $e_3[x \mapsto e_4] = v$. According to Lemma H.30, $e_1[x \mapsto e_2] \hookrightarrow e_3[x \mapsto e_4] = v$. Then by I.H., we have $e_1[x \mapsto e_2] \rightarrow^* v_2 \hookrightarrow v$. Therefore $(\lambda x : \tau. e_1) e_2 \rightarrow e_1[x \mapsto e_2] \rightarrow^* v_2 \hookrightarrow v$. Namely, $(\lambda x : \tau. e_1) e_2 \rightarrow^* v_2 \hookrightarrow v$.

- The case for rule **CE-AX-PPA-TBETA** is similar as the above case.

- Case rule **CE-AX-PPA-SPLICEQUOTE**, where $e \hookrightarrow v$ is

$$\llbracket e \rrbracket_{\Delta_i \vdash^i s_i : \tau_i = e_i} \hookrightarrow \llbracket e' [s_i \mapsto e_i'] \rrbracket_{\phi_i}^i.$$

Since the right hand side is a value, we know ϕ_i are value splice environments.

- (1) If $\phi_i = \phi_i'$, where $e_i = \llbracket e_i' \rrbracket_{\phi_i}$, and $e_i' \hookrightarrow e_i''$, and $\phi_i \hookrightarrow \phi_i'$, and $\phi_i' + \Delta_i \leadsto \phi_i''$, then ϕ_i' are also value splice environments.

By I.H. on every expression in ϕ_i , we get $\phi_i \rightarrow^* \phi_{v_i}$, and $\phi_{v_i} \hookrightarrow \phi_i'$.

Let $v_i = \llbracket e_i' \rrbracket_{\phi_{v_i}}$.

- (2) If $\phi_i = \Delta_i \vdash^i s_i : \tau_i = e_i'$, where $e_i \hookrightarrow e_i'$, then as we know ϕ_i is a value splice environment, we know e_i' is a value.

By I.H., $e_i \rightarrow^* v_i$, and $v_i \hookrightarrow e_i'$.

So we have $\llbracket e \rrbracket_{\Delta_i \vdash^i s_i : \tau_i = e_i} \rightarrow^* \llbracket e \rrbracket_{\Delta_i \vdash^i s_i : \tau_i = v_i} \hookrightarrow \llbracket e' [s_i \mapsto e_i'] \rrbracket_{\phi_i}^i$.

- Case rule **CE-AX-PPA-QUOTE SPLICE**, where $\llbracket s \rrbracket_{\Delta_i \vdash^i s : \tau = e} \hookrightarrow e$, and $e \hookrightarrow v$.

Then by I.H., we have $e \rightarrow^* v_2 \hookrightarrow v$.

Therefore $\llbracket s \rrbracket_{\Delta_i \vdash^i s : \tau = e} \rightarrow^* \llbracket s \rrbracket_{\Delta_i \vdash^i s : \tau = v_2} \hookrightarrow v$. □

Lemma H.33 (Permutation). *Given $\Theta; \Delta \vdash^i e_1 : \tau$, if $e_1 \hookrightarrow e_2$, and $e_2 \rightarrow e_3$, then there exists e_4 , such that $e_1 \rightarrow^* e_4$, and $e_4 \hookrightarrow e_3$.*

PROOF. By induction on the derivation complexity of $e_1 \hookrightarrow e_2$, and then on the size of e_1 . We do a case analysis on $e_1 \hookrightarrow e_2$.

- Case rule **CE-AX-PPA-LIT**. The case is impossible, as there is no e_3 such that $i \rightarrow e_3$.
- The cases for rule **CE-AX-PPA-VAR**, rule **CE-AX-PPA-SVAR**, rule **CE-AX-PPA-KVAR**, rule **CE-AX-PPA-ABS**, rule **CE-AX-PPA-TABS** are all impossible.
- Case rule **CE-AX-PPA-APP**, where $e_1 = e_5 e_6$, $e_5 \hookrightarrow e_7$, and $e_6 \hookrightarrow e_8$, and $e_2 = e_7 e_8$.

Now we do a case analysis on $e_2 \rightarrow e_3$.

– $e_3 = e_9 e_8$, where $e_7 \rightarrow e_9$.

Then by I.H., we have $e_5 \rightarrow^* e_{10} \hookrightarrow e_9$.

Therefore $e_5 e_6 \rightarrow^* e_{10} e_6 \hookrightarrow e_9 e_8$.

– $e_7 = \lambda x : \tau. e_9$, and $e_3 = e_9[x \mapsto e_8]$.

We know $e_5 \hookrightarrow e_7 = \lambda x : \tau. e_9$. By Lemma H.32, we have $e_5 \rightarrow^* v$, and $v \hookrightarrow \lambda x : \tau. e_9$. By analyzing $v \hookrightarrow \lambda x : \tau. e_9$, we know that it must be $v = \lambda x : \tau. e_{10}$ (v cannot be a quotation which is ill-typed.), and $e_{10} \hookrightarrow e_9$.

Therefore $e_5 e_6 \rightarrow^* (\lambda x : \tau. e_{10}) e_6 \rightarrow e_{10}[x \mapsto e_6]$.

By Lemma H.30, we have $e_{10}[x \mapsto e_6] \hookrightarrow e_9[x \mapsto e_8]$.

Namely $e_5 e_6 \rightarrow^* e_{10}[x \mapsto e_6] \hookrightarrow e_9[x \mapsto e_8]$.

- The case for rule **CE-AX-PPA-TAPP** is similar as the previous one.
- Case rule **CE-AX-PPA-BETA** where $e_1 = (\lambda x : \tau. e_5) e_6$, $e_5 \hookrightarrow e_7$, and $e_6 \hookrightarrow e_8$, and $e_2 = e_7[x \mapsto e_8]$.

Then $e_1 \rightarrow e_5[x \mapsto e_6]$.

By Lemma H.30, $e_5[x \mapsto e_6] \hookrightarrow e_7[x \mapsto e_8] \rightarrow e_3$.

By I.H., $e_5[x \mapsto e_6] \rightarrow^* e_9 \hookrightarrow e_3$ for some e_9 .

Therefore $e_1 \rightarrow e_5[x \mapsto e_6] \rightarrow^* e_9 \hookrightarrow e_3$.

Namely $e_1 \rightarrow^* e_9 \hookrightarrow e_3$.

- Case rule **CE-AX-PPA-TBETA** where $e_1 = (\Lambda a. e_5) \tau$, $e_5 \hookrightarrow e_6$, and $e_2 = e_6[a \mapsto \tau]$.

Then $e_1 \rightarrow e_5[a \mapsto \tau]$.

By Lemma H.30, $e_5[a \mapsto \tau] \hookrightarrow e_6[a \mapsto \tau] \rightarrow e_3$.

By I.H., $e_5[a \mapsto \tau] \rightarrow^* e_7 \hookrightarrow e_3$ for some e_7 .

Therefore $e_1 \rightarrow e_5[a \mapsto \tau] \rightarrow^* e_7 \hookrightarrow e_3$.

Namely $e_1 \rightarrow^* e_7 \hookrightarrow e_3$.

- Case rule **CE-AX-PPA-SPLICEQUOTE**, where $e_1 \hookrightarrow e_2$ is

$$\llbracket e \rrbracket_{\Delta_i \mu^i s_i : \tau_i = e_i} \hookrightarrow \llbracket e' [s_i \mapsto e_i'''] \rrbracket_{\phi_i}^i.$$

We have $\llbracket e' [s_i \mapsto e_i'''] \rrbracket_{\phi_i}^i \rightarrow e_3$. Then it must be $e_3 = \llbracket e' [s_i \mapsto e_i'''] \rrbracket_{\phi_i}^i$, and $\phi_i^i \rightarrow \phi_i'^i$.

According to reduction, ϕ_i^i is ϕ_i^i , except for one $\Delta_5 \mu^5 s_5 : \tau_5 = e_5 \in \phi_i$, we have $e_5 \rightarrow e_5'$.

- (1) $e_i = \llbracket e_i'' \rrbracket_{\phi_i'}$, and $e_i'' \hookrightarrow e_i'''$, and $\phi_i'' \hookrightarrow \phi_i'''$ and $\phi_i''' \vdash \Delta_i \leadsto \phi_i$.

We denote ϕ_i'''' as the splice environment ϕ_i''' after $e_5 \rightarrow e_5'$. Then $\phi_i'''' \vdash \Delta_i \leadsto \phi_i'$.

Therefore $\phi_i'' \hookrightarrow \phi_i''' \rightarrow \phi_i''''$.

By I.H. on e_5 , we know there is ϕ_i'''' , such that $\phi_i'' \rightarrow^* \phi_i'''' \hookrightarrow \phi_i''''$.

Let $\overline{e_i''''}^i = \overline{e_i}^i$ for every index, except for i , where $e_i'''' = \llbracket e_i'' \rrbracket_{\phi_i''''}$.

Therefore

$$\llbracket e \rrbracket_{\Delta_i \mu^i s_i : \tau_i = e_i} \rightarrow^* \llbracket e \rrbracket_{\Delta_i \mu^i s_i : \tau_i = e_i''''} \hookrightarrow \llbracket e' [s_i \mapsto e_i'''] \rrbracket_{\phi_i}^i.$$

- (2) $\phi_i = \Delta_5 \mu^5 s_5 : \tau_5 = e_5$, where $e_i \hookrightarrow e_5$.

Namely, $e_i \hookrightarrow e_5 \rightarrow e_5'$.

By I.H., $e_i \rightarrow^* e_6$, and $e_6 \hookrightarrow e_5'$.

Let $\overline{e_i'}^i = \overline{e_i}^i$ for every index, except for i , where $e_i' = e_6$.

Therefore

$$\llbracket e \rrbracket_{\Delta_i \mu^i s_i : \tau_i = e_i} \rightarrow^* \llbracket e \rrbracket_{\Delta_i \mu^i s_i : \tau_i = e_i'} \hookrightarrow e_3 \llbracket e' [s_i \mapsto e_i'''] \rrbracket_{\phi_i}^i.$$

- Case rule **CE-AX-PPA-QUOTE SPLICE**, where $e = \llbracket s \rrbracket_{\bullet \mu^i s : \tau = e_5}$, and $e_5 \hookrightarrow e_2$, and $e_2 \rightarrow e_3$.

By I.H., $e_5 \rightarrow^* e_4 \hookrightarrow e_3$.

Therefore $\llbracket s \rrbracket_{\bullet \mu^i s : \tau = e_5} \rightarrow^* \llbracket s \rrbracket_{\bullet \mu^i s : \tau = e_4}$.

If e_3 is not a quotation, then by rule **CE-AX-PPA-QUOTE SPLICE**, or otherwise by rule **CE-AX-PPA-SPLICEQUOTE**, $\llbracket s \rrbracket_{\bullet \mu^i s : \tau = e_4} \hookrightarrow e_3$.

□

Lemma H.34 (Push Back). *Given $\Theta; \Delta \Vdash e_1 : \tau$, if $e_1 \rightsquigarrow e_2$, and $e_2 \longrightarrow^* v_1$, then there exists v_2 , such that $e_1 \longrightarrow^* v_2$, and $v_2 \rightsquigarrow v_1$.*

PROOF. We first virtualize the hypothesis, where $e_2 \longrightarrow^* v_1$ corresponds to a chain of evaluation:

$$e_1 \rightsquigarrow e_2 \longrightarrow e_3 \longrightarrow \dots e_{i-1} \longrightarrow e_i \longrightarrow v_1$$

Apply Lemma H.33, we have

$$e_1 \longrightarrow^* e'_2 \rightsquigarrow e_3 \longrightarrow \dots e_{i-1} \longrightarrow e_i \longrightarrow v_1$$

Keep applying Lemma H.33 $i - 2$ times, then we get

$$e_1 \longrightarrow^* e'_2 \longrightarrow^* e'_3 \longrightarrow^* \dots e'_{i-1} \longrightarrow^* e'_i \rightsquigarrow v_1$$

Apply Lemma H.32, we have

$$e_1 \longrightarrow^* e'_2 \longrightarrow^* e'_3 \longrightarrow^* \dots e'_{i-1} \longrightarrow^* e'_i \longrightarrow^* v_2 \rightsquigarrow v_1$$

Namely

$$e_1 \longrightarrow^* v_2 \rightsquigarrow v_1$$

□

Lemma H.35 (\longrightarrow^* simulates \rightsquigarrow^*). *Given $\Theta; \Delta \Vdash e : \tau$, if $e \rightsquigarrow^* v$, then there exists v_2 such that $e \longrightarrow^* v_2$, and $v_2 \rightsquigarrow^* v$.*

PROOF. If e is a value, then let $v_2 = e$ and we are done.

Otherwise, by Lemma H.31, $e \rightsquigarrow^* v$ can be visualized as a chain of parallel reduction:

$$e \rightsquigarrow e_1 \rightsquigarrow \dots \rightsquigarrow e_i \rightsquigarrow v_1 \rightsquigarrow^* v$$

where e_1 to e_i are non-values.

By Lemma H.32, we have

$$e \rightsquigarrow e_1 \rightsquigarrow \dots \rightsquigarrow e_i \longrightarrow^* v_2 \rightsquigarrow v_1 \rightsquigarrow^* v$$

Namely,

$$e \rightsquigarrow e_1 \rightsquigarrow \dots \rightsquigarrow e_i \longrightarrow^* v_2 \rightsquigarrow^* v$$

By Lemma H.34, we get

$$e \rightsquigarrow e_1 \rightsquigarrow \dots \rightsquigarrow e_{i-1} \longrightarrow^* v_3 \rightsquigarrow v_2 \rightsquigarrow^* v$$

Namely,

$$e \rightsquigarrow e_1 \rightsquigarrow \dots \rightsquigarrow e_{i-1} \longrightarrow^* v_3 \rightsquigarrow^* v$$

Keep applying Lemma H.34, then we get

$$e \longrightarrow^* v_{i+2} \rightsquigarrow^* v$$

□