Seeking Stability by being Lazy and Shallow

Lazy and shallow instantiation is user friendly

GERT-JAN BOTTU*, KU Leuven, Belgium
RICHARD A. EISENBERG, Tweag I/O, France

Designing a language feature often requires a choice between several, similarly expressive possibilities. Given that user studies are generally impractical, we propose using *stability* as a way of making such decisions. Stability is a measure of whether the meaning of a program alters under small, seemingly innocuous changes in the code.

Directly motivated by a need to pin down a feature in GHC/Haskell, we apply this notion of stability to analyse four approaches to the instantiation of polymorphic types, concluding that the most stable approach is lazy (instantiate a polytype only when absolutely necessary) and shallow (instantiate only top-level type variables, not variables that appear after explicit arguments).

1 INTRODUCTION

Programmers naturally wish to get the greatest possible utility from their work. They thus embrace *polymorphism*: the idea that one function can work with potentially many types. A simple example is $const :: \forall\ a\ b.\ a \to b \to a$, which returns its first argument, ignoring its second. The question then becomes: what concrete types should *const* work with at a given call site? For example, if we say *const True* `'x'`, then a compiler needs to figure out that a should become *Bool* and b should become *Char*. The process of taking a type variable and substituting in a concrete type is called *instantiation*. Choosing a correct instantiation is important; for *const*, the choice of $a \mapsto Bool$ means that the return type of *const True* `'x'` is *Bool*. A context expecting a different type would lead to a type error.

In the above example, the choices for a and b in the type of *const* were inferred. Many languages also give programmers the opportunity to *specify* the instantiation for these arguments. For example, we might say (in today's Haskell) *const* @*Bool* @*Char True* `'x'` (choosing the instantiations for both a and b) or *const* @*Bool True* `'x'` (still allowing inference for b). However, once we start allowing user-directed instantiation, many thorny design issues arise. For example, will **let** f = *const* **in** f @*Bool True* `'x'` be accepted?

This paper considers both implicit and explicit instantiation, and how design decisions around instantiation can affect the ease of programming in a language. Every language that supports polymorphism must make decisions around instantiation. It is our opinion that many designers grapple with these issues; this paper aims to unpack these issues and lay out a framework in which one can assess what design choices are best for a language.

Our concerns are rooted in design questions for Haskell, as embodied by the Glasgow Haskell Compiler (GHC). Specifically, as Haskell increasingly has features in support of type-level programming, how should its instantiation behave? Should instantiating a type like $Int \to \forall\ a.\ a \to a$

*This work was partially completed while Bottu was an intern at Tweag I/O.

Bottu worked on all aspects of the paper, including designing the typing rules, writing all of the proofs, and composing the text. Eisenberg organized and guided the project, critiqued the typing rules, provided GHC/Haskell expertise, and substantially contributed to writing.

Authors' addresses: Gert-Jan Bottu, Computer Science Department, KU Leuven, Belgium, gertjan.bottu@kuleuven.be; Richard A. Eisenberg, Tweag I/O, Paris, France, rae@richarde.dev.

2021. 2475-1421/2021/8-ART1 $15.00
https://doi.org/

yield $Int \to \alpha \to \alpha$ (where α is a unification variable), or should instantiation stop at the regular argument of type Int? This is a question of the *depth* of instantiation. Suppose $f :: Int \to \forall\ a.\ a \to a$. Should $f\ 5$ have type $\forall\ a.\ a \to a$ or $\alpha \to \alpha$? This is a question of the *eagerness* of instantiation. As we explore in Section 5.3, these questions have real impact on users of the language.

Unlike much type-system research, our goal is *not* simply to make a type-safe language. Type-safe instantiation is well understood [e.g., Damas and Milner 1982; Reynolds 1974]. Instead, we wish to examine the *usability* of a design around instantiation. Unfortunately, proper scientific studies around usability are essentially intractable, as we would need pools of comparable experts in several designs executing a common task. Instead of usability, then, we draw a fresh focus to a property we name *stability*.

Intuitively, a language is *stable* if small changes to the input of the language implementation do not cause large changes to the output (i.e., the compiled executable or the behaviour of an interpreter). Yet we do not mean exactly that: changing an index in a program from 0 to 1 might reasonably drastically change a program's behaviour. Instead, we identify a set of program changes that programmers might expect to have no effect; we call these *program similarities*. A language is stable if changing one program into a similar one produces no change in behaviour. We say a language *respects* a similarity if similar programs indeed have the same behaviour. For example, we generally expect that $a + b$ in a program text has the same meaning as $b + a$; we would thus say that $a + b$ is similar to $b + a$. Yet a language with implicit type conversions might not always respect this similarity, perhaps leading the two expressions to have different types.

This paper proceeds by explaining *stability*, our metric for understanding and evaluating language design. The concept of stability and our general approach of analysis with respect to stability is language- and feature-agnostic. We then identify how mixing implicit with explicit instantiation can lead to instability in several languages (Haskell, Agda, and Idris); accordingly, choosing a design around instantiation is a key design step in any language mixing implicit and explicit instantiation. Having laid out our approach (stability) and our problem (how to design instantiation), we then explore how to apply this approach to the concrete setting of the problem in Haskell, concluding that *shallow*, *lazy* instantiation leads to the most stable design. This result came as something of a surprise: as recently as GHC 8.10, Haskell used a deep, eager strategy. GHC 9.0 updated this to use shallow and eager. We advocate here to take a further step to be shallow and lazy.

Our contributions are as follows:

- The introduction of stability properties for functional programming languages, and their interaction with type instantiation design choices, along with examples of how these properties affect programmer experience. (Sections 2–5)
- A family of type systems, based on the bidirectional type-checking algorithm implemented in GHC [Eisenberg et al. 2016; Peyton Jones et al. 2007; Serrano et al. 2020] but with the ability to visibly instantiate type variables. It is parameterised over the type instantiation flavour. (Sections 6–7)
- An analysis of how different choices of instantiation flavour either respect or do not respect the similarities we identify. We conclude that lazy, shallow instantiation is the most stable. (Section 8)
- An application of our results to the concrete use case of GHC/Haskell (Section 9)

We use Haskell both as a *lingua franca* throughout this paper, and also as inspiration for the main technical development in this paper. Stability issues are necessarily language-specific, as authors in different languages will naturally have different expectations as to what programs are considered similar. However, our general approach—examining a language though the lens of *stability*—is

portable across languages, and our work serves as a template for how one might perform this analysis in a different setting.

The appendices mentioned in the text are included as anonymised supplementary material.

2 STABILITY

We have described stability as a measure of how small transformations—call them *similarities*—in user-written code might drastically change the behaviour of a program. This section lays out the specific similarities we will consider with respect to our instantiation flavours. There are naturally *many* transformations one might think of applying to a source program. We have chosen ones that relate best to instantiation; others (e.g. does a function behave differently in curried form as opposed to uncurried form?) do not distinguish among our flavours and are thus less interesting in our concrete context. We include examples demonstrating each of these—and how instantiation causes trouble—in Section 5. After presenting our formal model of Haskell instantiation, we check our instantiation flavours against these similarities in Section 8.

Before listing the similarities, we must extract out one salient detail buried in the definition of stability: what we mean by the behaviour of a program. We will analyse two different notions of behaviour, both the *static* semantics of a program (that is, whether the program is accepted and what types are assigned to its variables) and its *dynamic* semantics (that is, what the program does at runtime, assuming it is still well typed). We write, for example, $\overset{S+D}{\Longleftrightarrow}$ to denote a similarity that we expect to respect both static and dynamic semantics, whereas $\overset{D}{\Longleftrightarrow}$ is one that we expect only to respect dynamic semantics, but may change static semantics.

A key concern for us is around **let**-inlining and -extraction. That is, if we bind an expression to a new variable and use that variable instead of the original expression, does our program change meaning? Or if we inline a definition, does our program change meaning? These notions are captured in Similarity 1:[1]

Similarity 1.

$$\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \overset{S+D}{\Longleftrightarrow} [e_1/x]\ e_2$$

The second similarity annotates a let binding with the inferred type σ of the bound expression e_1. We expect this similarity to be one-directional, as dropping a type annotation may indeed change the static semantics of a program, as we hope programmers expect.

Similarity 2.

$$\overline{f\ \overline{\pi}_i = e_i}^{\,i} \overset{S+D}{\Longrightarrow} f : \sigma; \overline{f\ \overline{\pi}_i = e_i}^{\,i},\ \textit{where } \sigma \textit{ is the inferred type of } f$$

Changing a type signature should not affect dynamic semantics—except in the case of type classes (or other feature that interrupts parametricity). Because our paper elides type classes, we can state this similarity quite generally; more fleshed-out settings would require a caveat around the lack of type-class constraints.

Similarity 3.

$$f : \sigma_1; \overline{f\ \overline{\pi}_i = e_i}^{\,i} \overset{D}{\Longleftrightarrow} f : \sigma_2; \overline{f\ \overline{\pi}_i = e_i}^{\,i}$$

The fourth similarity represents changing variable patterns (written to the left of the = in a function definition) into λ-binders (written on the right of the =), and vice versa. Here, we assume the patterns $\overline{\pi}$ contain only (expression and type) variables. The operator *wrap* is unsurprising,

[1] A language with a strict **let** construct will observe a runtime difference between a let binding and its expansion, but this similarity would still hold with respect to type-checking.

and just wraps the patterns around the expression in lambda binders. Its definition can be found in Appendix C.

Similarity 4.

$$\textbf{let}\ x\ \overline{\pi} = e_1\ \textbf{in}\ e_2 \overset{S+D}{\Longleftrightarrow} \textbf{let}\ x = e_1'\ \textbf{in}\ e_2,\ \textit{where wrap}\ (\overline{\pi}; e_1 \sim e_1')$$

Our language model includes the ability to define a function by specifying multiple equations. We would want the addition of new equations in a function definition not to affect types. While normally new equations for a function would vary the patterns compared to existing equations, we simply repeat the existing equation twice; after all, the particular choice of (well-typed) pattern should not affect static semantics at all.

Similarity 5.

$$f\ \overline{\pi} = e \overset{S}{\Longleftrightarrow} f\ \overline{\pi} = e, f\ \overline{\pi} = e$$

And lastly, we want η-expansion not to affect types. (This change *can* reasonably affect runtime behaviour, so we would never want to assert that η-expansion maintains dynamic semantics.)

Similarity 6.

$$e \overset{S}{\Longleftrightarrow} \lambda x.e\ x,\ \textit{where } e \textit{ has a function type}$$

We now fix the definition of stability we will work toward in this paper:

Definition (Stability). *A language is considered stable when all of the program similarities above are respected.*

We note here that the idea of judging a language by its robustness in the face of small transformations is not new; see, for example, Le Botlan and Rémy [2009] or Schrijvers et al. [2019], who also consider a similar property. However, we believe ours is the first work to focus on it as the primary criterion of evaluation.

Our goal in this paper is not to eliminate instability, which would likely be too limiting, leaving us potentially with either the Hindley-Milner implicit type system or a System F explicit one. Both are unsatisfactory. Instead, our goal is to make the consideration of stability a key guiding principle in language design. The rest of this paper uses the lens of stability to examine design choices around ordered explicit type instantiation. We hope that this treatment serves as an exemplar for other language design tasks and provides a way to translate vague notions of an "intuitive" design into concrete language properties that can be proved or disproved. Furthermore, we believe that instantiation is an interesting subject of study, as any language with polymorphism must consider these issues, making them less esoteric than they might first appear.

3 MIXING IMPLICIT AND EXPLICIT FEATURES IS UNSTABLE

The concept of stability is important in languages that have a mix of implicit and explicit features—a very common combination, appearing in Coq, Agda, Idris, modern Haskell, C++, Java, C#, Scala, F#, and Rust, among others. This section walks through how a mixing of implicit and explicit features in Haskell[2], Idris[3], and Agda[4] causes instability. We use these languages to show how the issues we describe are likely going to arise in any language mixing implicit and explicit features—and how stability is a worthwhile metric in examining these features—not to critique these languages in

[2]Our work is tested with GHC 8.10.1, and we use "Haskell" to refer to the language accepted by GHC.
[3]We work with Idris 2, as available from https://github.com/idris-lang/Idris2, at commit a7d5a9a7fdfbc3e7ee8995a07b90e6a454209cd8.
[4]We work with Agda 2.6.0.1.

particular. This section considers the instantiation features in these languages; see Appendix A for discussion of other sources of instability in these languages.

All three of our example languages feature explicit instantiation of implicit arguments, allowing the programmer to manually instantiate a polymorphic type, for example. Explicit instantiation broadly comes in two flavours: ordered or named parameters.

3.1 Haskell

Haskell supports ordered instantiation [Eisenberg et al. 2016]: Given $const :: \forall\ a\ b.\ a \to b \to a$, we can write $const\ @Int\ @Bool$, which instantiates the type variables, giving us an expression of type $Int \to Bool \to Int$. If a user wants to visibly instantiate a later type parameter (say, b) without choosing an earlier one, they can write @_ to skip a parameter. The expression $const\ @\_\ @Bool$ has type $\alpha \to Bool \to \alpha$, for any type α. Ordered parameters have two advantages: they are concise and do not leak the choice of variable names from the library to its client.

Haskell's treatment of instantiation is unstable even without reference to other features described below, because Haskell type signatures do not need to explicitly quantify their variables. For example, we can declare $const$ with $const :: a \to b \to a$, with no $\forall\ a\ b$, and we can still write $const\ @Int\ @Bool$ to instantiate the function. The policy is to quantify the variables according to their left-to-right occurrence in the type (ignoring duplicate occurrences). This means, though, that if we define **type** $b \Leftarrow a = a \to b$ (the lexeme $\leftarrow$ is reserved) and then write $const :: (b \to a) \Leftarrow a$, we now require all clients of $const$ to also reverse the order of their type instantiations. As a conscientious library author, we could choose to explicitly include $\forall\ a\ b$ to keep the old variable ordering, but it would be easy to think that the use of a type synonym in a type signature is a purely internal matter and that no special care need be taken.

3.2 Idris

Idris, on the other hand, supports named parameters. If we define $const : \{a, b : Type\} \to a \to b \to a$ (this syntax is the Idris equivalent of the Haskell type $\forall\ a\ b.\ a \to b \to a$), then we can write $const\ \{b = Bool\}$ to instantiate only the second type parameter or $const\ \{a = Int\}\ \{b = Bool\}$ to instantiate both. Order does not matter; $const\ \{b = Bool\}\ \{a = Int\}$ works as well as the previous example. Named parameters may be easier to read than ordered parameters and are robust to the addition of new type variables.

Idris's approach suffers from an instability inherent with named parameters. Unlike Haskell, the order of quantified variables does not matter. Yet now, the choice of *names* of the parameters naturally *does* matter. Thus $const : c \to d \to c$ (taking advantage of the possibility of omitting explicit quantification in Idris) has a different interface than $const : a \to b \to a$, despite the fact that the type variables scope over only the type signature they appear in.

3.3 Agda

Agda accepts both ordered and named parameters. After defining $const : \{a\ b : Set\} \to a \to b \to a$, we can write expressions like $const\ \{Int\}$ (instantiating only a), $const\ \{b = Bool\}$, or $const\ \{\_\}\ \{Bool\}$. Despite using named parameters, order *does* matter: we cannot instantiate earlier parameters after later ones. Naming is useful for skipping parameters that the user does not wish to instantiate. Because Agda requires explicit quantification of variables used in types (except as allowed for in implicit generalisation, below), the ordering of variables must be fixed by the programmer. However, like Idris, Agda suffers from the fact that the choice of name of these local variables leaks to clients.

3.4 Conclusion

We see here how Haskell, Idris, and Agda all have a mix of explicit and implicit features around type instantiation. This mix raises the potential for language instability: they make it possible for small, apparently insignificant changes to code to affect the semantics of programs. All three languages examined here have taken steps to mitigate this instability, but none have eliminated it entirely.

4 PROVENANCE IN HASKELL

Because we use Haskell as our main object of study, we briefly introduce Haskell's concept of variable *provenance*. This feature is directly motivated by the desire for stability, though we are the first to describe it that way. It was originally introduced by Eisenberg et al. [2016] and is frequently called *specificity* in the context of GHC; however, we find here that *provenance* is a more meaningful moniker. The key idea is that quantified type variables are either written by the user (these are called *specified*) or invented by the compiler (these are called *inferred*). A specified variable is available for explicit instantiation using, e.g., $@Int$; an inferred variable may not be explicitly instantiated.

Following GHC, we use braces to denote inferred variables. Thus, if we have the Haskell program

$$id_1 :: a \to a$$
$$id_1\ x = x$$
$$id_2\ x = x$$

then we would write that $id_1 :: \forall\ a.\ a \to a$ (with a specified a) and $id_2 :: \forall\ \{a\}.\ a \to a$ (with an inferred a). Accordingly, $id_1\ @Int$ is a function of type $Int \to Int$, while $id_2\ @Int$ is a type error.

5 TYPE INSTANTIATION

We now turn to our main feature of study: implicit instantiation. We lay out several alternative ways of designing this feature, observe that each can cause instability, and then (in Section 6) develop typing rules (based on the current behaviour of GHC/Haskell, with key inspiration from Eisenberg et al. [2016]; Peyton Jones et al. [2007]; Serrano et al. [2020]) so that we can prove how our choices of instantiation technique either respect or do not respect the similarities.

Our key question: Given a polymorphic function f, how and when should we instantiate it to concrete types?

We introduce several different flavours of instantiation algorithm, differing in two orthogonal axes: when types are instantiated, and the depth to which binders are instantiated. We also showcase several examples where these choices make a real difference.

5.1 Deep vs. Shallow Instantiation

The *depth* of instantiation determines which type variables get instantiated. Concretely, shallow instantiation affects only the type variables bound before any explicit arguments. Deep instantiation, on the other hand, also instantiates all variables bound after an arbitrary number of explicit arguments. For example, consider a function $f :: \forall\ a.\ a \to (\forall\ b.\ b \to b) \to \forall\ c.\ c \to c$. A shallow instantiation of f's type instantiates only a, whereas deep instantiation would also instantiate c, which occurs after two explicit arguments, one of type a and one of type $\forall\ b.\ b \to b$. Note that neither instantiation flavour touches the b variable, as doing so would actually make the function type more general, thus breaking type safety.

Versions of GHC up to 8.10 perform deep instantiation, as originally introduced by Peyton Jones et al. [2007], but GHC 9.0 changes this design, as proposed by Peyton Jones [2019] and inspired by Serrano et al. [2020]. In this paper, we study this change through the lens of stability.

5.2 Eager vs. Lazy Instantiation

The *eagerness* of instantiation determines the location in the code where instantiation happens. Eager instantiation immediately instantiates a polymorphic type variable as soon as it is mentioned. In contrast, lazy instantiation holds off instantiation as long as possible until instantiation is necessary in order to, say, allow a variable to be applied to an argument.

For example, consider this function:

$$pair :: \forall\ a.\ a \to \forall\ b.\ b \to (a, b)$$

Note that *pair* first quantifies over the type a, followed by a value of type a, and only then quantifies over the type b.

Now, consider this definition of *myPairX*:

$$myPairX\ x = pair\ x$$

What type do we expect to infer for *myPairX*? With eager instantiation, the type of a polymorphic expression is instantiated as soon as it occurs. Thus, *pair x* will have a type $\beta \to (\alpha, \beta)$, assuming we have guessed $x :: \alpha$. (We use Greek letters to denote unification variables.) With neither α nor β constrained, we will generalise both (ignoring provenance for now), and infer $\forall\ a\ b.\ a \to b \to (a, b)$ for *myPairX*. Crucially, this type is *different* than the type of *pair*.

Now, let us now replay this process with lazy instantiation. The variable *pair* has type $\forall\ a.\ a \to \forall\ b.\ b \to (a, b)$. In order to apply *pair* of that type to x, we must instantiate the first quantified type variable a to a fresh unification variable α, yielding the type $\alpha \to \forall\ b.\ b \to (\alpha, b)$. This is indeed a function type, so we can consume the argument x, yielding $pair\ x :: \forall\ b.\ b \to (\alpha, b)$. We have now type-checked the expression *pair x*, and thus we take the parameter x into account and generalise this type to produce the inferred type $myPairX :: \forall\ a.\ a \to \forall\ b.\ b \to (a, b)$. This is the same as the type given for *pair*.

As we have seen, thus, the choice of eager or lazy instantiation can change the inferred types of definitions. In a language that allows visible instantiation of type variables, the difference between these types is user-visible. With eager instantiation, *myPairX* @*Bool* @*Char True* `'x'` is accepted, whereas with lazy instantiation, only *myPairX* @*Bool True* @*Char* `'x'` is correct.

5.3 Thorny Examples

Relating the general ideas of instantiation depth and eagerness to the concrete Haskell implementation, with its notion of provenance, we present a number of examples showing how instantiation can become muddled. While these examples are described in terms of types inferred for definitions that have no top-level signature, many of the examples can be adapted to situations that do not depend on the lack of a signature. Each example is annotated with the similarity it illustrates.

Example 1: myId (Similarity 1). The Haskell standard library defines $id :: \forall\ a.\ a \to a$ as the identity function. Suppose we made a synonym of this, with the following:

$$myId = id$$

Note that there is no type signature. Even in this simple example, our choice of instantiation eagerness changes the type we infer:

myId	eager	lazy
deep or shallow	$\forall\ \{a\}.\ a \to a$	$\forall\ a.\ a \to a$

Under eager instantiation, the mention of *id* is immediately instantiated, and thus we must re-generalise in order to get a polymorphic type for *myId*. Generalising always produces inferred variables, and so the inferred type for *myId* starts with $\forall\ \{a\}$, meaning that *myId* cannot be a

drop-in replacement for *id*, which might be used with explicit type instantiation. On the other hand, lazy instantiation faithfully replicates the type of *id* and uses it as the type of *myId*.

Example 2: myPair (Similarity 1). This problem gets even worse if the original function has a non-prenex type, like our *pair*, above. Our definition is now:

$$myPair = pair$$

With this example, both design axes around instantiation matter:

myPair	eager	lazy
deep	$\forall\ \{a\}\ \{b\}.\ a \to b \to (a, b)$	$\forall\ a.\ a \to \forall\ b.\ b \to (a, b)$
shallow	$\forall\ \{a\}.\ a \to \forall\ b.\ b \to (a, b)$	$\forall\ a.\ a \to \forall\ b.\ b \to (a, b)$

All we want is to define a simple synonym, and yet reasoning about the types requires us to consider both depth and eagerness of instantiation.

Example 3: myPairX (Similarity 1). The *myPairX* example above acquires a new entanglement once we account for provenance. We define *myPairX* with this:

$$myPairX\ x = pair\ x$$

We infer these types:

myPairX	eager	lazy
deep or shallow	$\forall\ \{a\}\ \{b\}.\ a \to b \to (a, b)$	$\forall\ \{a\}.\ a \to \forall\ b.\ b \to (a, b)$

Unsurprisingly, the generalised variables end up as inferred, instead of specified.

Example 4: infer (Similarity 2). Though not yet implemented, we consider a version of Haskell that includes the ability to abstract over type variables, the subject of an approved proposal for GHC [Eisenberg 2018]. With this addition, we can imagine writing *infer*:

$$infer = \lambda\ @a\ (x :: a) \to x$$

We would infer these types:

infer	eager	lazy
deep or shallow	$\forall\ \{a\}.\ a \to a$	$\forall\ a.\ a \to a$

Note that the eager variant will infer a type containing an inferred quantified variable $\{a\}$. this is because the expression $\lambda\ @a\ (x :: a) \to x$ is instantly instantiated; it is then **let**-generalised to get the type in the table above.

If we change our program to include these types as annotations, the eager type, with its inferred variable, will be rejected. The problem is that we cannot check an abstraction $\lambda\ @a \to \ldots$ against an expected type $\forall\ \{a\}.\ \ldots$: the whole point of having an inferred provenance is to prevent such behaviour, as an inferred variable should not correspond to either abstractions or applications in the term.

Example 5: swizzle (Similarity 3). Suppose we have this function defined[5]:

$$undef :: \forall\ a.\ Int \to a \to a$$
$$undef = undefined$$

Now, we write a synonym but with a slightly different type:

$$swizzle :: Int \to \forall\ a.\ a \to a$$
$$swizzle = undefined$$

[5]This example is inspired by Peyton Jones [2019].

Shockingly, *undef* and *swizzle* have different runtime behaviour: forcing *undef* diverges (unsurprisingly), but forcing *swizzle* has no effect. The reason is that the definition of *swizzle* is not as simple as it looks. In the System-F-based core language used within GHC, we have $swizzle = \lambda(n :: Int) \to \Lambda(a :: Type) \to undef\ @a\ n$. Accordingly, *swizzle* is a function, which is already a value.

Under shallow instantiation, *swizzle* would simply be rejected, as its type is different than *undef*'s. The only way *swizzle* can be accepted is if it is deeply skolemised (see *Function Application* in Section 6), a necessary consequence of deep instantiation.

swizzle	eager or lazy
deep	converges
shallow	rejected

Example 6: infer2, again (Similarity 4). Returning to the *infer* example, we might imagine moving the abstraction to the left of the =, yielding:

$$infer2\ @a\ (x :: a) = x$$

Under all instantiation schemes, *infer2* will be assigned the type $\forall\ a.\ a \to a$. Accordingly, under eager instantiation, the choice of whether to bind the variables before the = or afterwards matters.

Example 7: boolId1 and boolId2 (Similarity 5). Consider these two definitions:

$$boolId1\ (\_ :: Bool) = id$$
$$boolId2\ False = id$$
$$boolId2\ True\ = id$$

Both of these functions ignore their input and return the polymorphic identity function. (The strictness of the functions differs, but that need not concern us here.) Let us look at their types:

		eager	lazy
boolId1	deep or shallow	$\forall\ \{a\}.\ Bool \to a \to a$	$Bool \to \forall\ a.\ a \to a$
boolId2	deep or shallow	$\forall\ \{a\}.\ Bool \to a \to a$	$\forall\ \{a\}.\ Bool \to a \to a$

The lazy case for *boolId1* is the odd one out: we see that the definition of *boolId1* has type $\forall\ a.\ a \to a$, do not instantiate it, and then prepend the *Bool* parameter. In the eager case, we see that both definitions instantiate *id* and then re-generalise.

However, the most interesting case is the treatment of *boolId2* under lazy instantiation. The reason the type of *boolId2* here differs from that of *boolId1* is that the pattern-match forces the instantiation of *id*. As each branch of a multiple-branch pattern-match must result in the same type, we have to seek the most general type that is still less general than each branch's type. Pattern matching thus performs an instantiation step (regardless of eagerness), in order to find this common type.

In the scenario of *boolId2*, however, this causes trouble: the match instantiates *id*, and then the type of *boolId2* is re-generalised. This causes *boolId2* to have a different inferred type than *boolId1*. Here, once again, we witness an instability: we do not expect the form of η-expansion we see here to change the type of a definition.

Example 8: eta (Similarity 6). Consider these two definitions, where $id :: \forall\ a.\ a \to a$:

$$noEta = id$$
$$eta\ \ \ = \lambda x \to id\ x$$

The two right-hand sides should have identical meaning, as *eta* is simply the η-expansion of *noEta*. Yet, under lazy instantiation, these two will have different types:

$$\delta ::= \mathcal{S} \mid \mathcal{D} \qquad \textit{Depth}$$
$$\epsilon ::= \mathcal{E} \mid \mathcal{L} \qquad \textit{Eagerness}$$

$$\begin{array}{lcll}
\tau & ::= & a \mid \tau_1 \to \tau_2 \mid T\,\overline{\tau} & \textit{Monotype} \\
\rho & ::= & \tau \mid \sigma_1 \to \phi^{\delta}_2 & \textit{Instantiated type} \\
\sigma & ::= & \rho \mid \forall\, \overline{a}.\sigma \mid \sigma_1 \to \sigma_2 & \textit{Type scheme} \\
\phi^{\delta} & ::= & \rho \quad (\delta = \mathcal{D}) & \textit{Instantiated result} \\
 & \mid & \sigma \quad (\delta = \mathcal{S}) & \\
\eta^{\epsilon} & ::= & \rho \quad (\epsilon = \mathcal{E}) & \textit{Synthesised type} \\
 & \mid & \sigma \quad (\epsilon = \mathcal{L}) & \\
\psi & ::= & \tau & \textit{Arg. descriptor}
\end{array}$$

$$\begin{array}{lcll}
e & ::= & x \mid \lambda x.e \mid e_1\, e_2 \mid K \mid e : \sigma & \textit{Expression} \\
 & \mid & \textbf{let}\ \textit{decl}\ \textbf{in}\ e & \\
\textit{decl} & ::= & x : \sigma; \overline{x\,\overline{\pi}_i = e_i}^{\,i} \mid \overline{x\,\overline{\pi}_i = e_i}^{\,i} & \textit{Declaration} \\
\pi & ::= & x \mid K\,\overline{\pi} & \textit{Pattern} \\
 & & & \\
\Sigma & ::= & \cdot \mid \Sigma, T\,\overline{a} \mid \Sigma, K : \overline{a}; \overline{\sigma}; T & \textit{Static context} \\
\Gamma, \Delta & ::= & \Sigma \mid \Gamma, x : \sigma \mid \Gamma, a & \textit{Context}
\end{array}$$

Fig. 1. Implicit Polymorphic λ-Calculus (IPLC) Syntax

		eager	lazy
noEta	deep or shallow	$\forall\,\{a\}.\, a \to a$	$\forall\, a.\, a \to a$
eta	deep or shallow	$\forall\,\{a\}.\, a \to a$	$\forall\,\{a\}.\, a \to a$

The problem is that the η-expansion instantiates the occurrence of *id* in *eta*, despite the lazy instantiation strategy. Under eager instantiation, the instantiation happens regardless.

5.4 Conclusion

The examples in this section show that the choice of instantiation scheme matters—and that no individual choice is clearly the best. To summarise, each of our possible schemes runs into trouble with some example; this table lists the numbers of the examples that witness a problem:

	eager	lazy
deep	1, 2, 3, 4, 5, 6	5, 7, 8
shallow	1, 2, 3, 4, 6	7, 8

At this point, the best choice is unclear. Indeed, these examples are essentially where we started our exploration of this issue—with failures in each quadrant of this table, how should we design instantiation in GHC? To understand this better, we present in the next two sections a formalisation of GHC's type-checking algorithm, parameterised over the choice of depth and eagerness. Section 8 then presents properties derived from the similarities of Section 2 and checks which variants of our type system uphold which properties. The conclusion becomes clear: lazy, shallow instantiation respects the most similarities.

Let us dive into the details to see how this all works.

6 THE IMPLICIT POLYMORPHIC λ-CALCULUS

In order to assess our design choices against the transformations we desire to ensure stability, this section and the next develop detailed calculi to model our language. We start with a simpler calculus in this section: a polymorphic, stratified λ-calculus with implicit polymorphism (only). We call it the Implicit Polymorphic λ-calculus, or IPLC. By "implicit", we mean that terms cannot explicitly instantiate or bind types. However, type signatures are allowed in the calculus, and the bidirectional type system [Pierce and Turner 2000] permits higher-rank [Odersky and Läufer 1996] functions. Some other features, such as local **let** declarations defining functions with multiple equations, are added to bring this closer to our real-world application of Haskell.

We have built this system to support flexibility in both of our axes of instantiation scheme design. That is, the calculus is parameterised over choices of instantiation depth and eagerness. In this way,

our calculus is essentially a *family* of type systems: choose your design, and you can instantiate our rules accordingly.

The syntax for IPLC is shown in Figure 1. We define two meta parameters δ and ϵ denoting the depth and eagerness of instantiation respectively. In the remainder of this paper, grammar and relations which are affected by one of these parameters will be annotated as such. A good example of this are types ϕ^δ and η^ϵ, as explained below.

Keeping all the moving pieces straight can be challenging. We thus offer some mnemonics to help the reader: In the remainder of the paper, aspects pertaining to **e**ager instantiation are highlighted in **e**merald, while **l**azy features are highlighted in lavender. Similarly, instantiation under the **s**hallow scheme is drawn using a **s**triped line, as in $\Gamma \vdash \sigma \overset{inst\ \mathcal{S}}{\dashrightarrow} \rho$.

Types. Our presentation of the IPLC contains several different type categories, which we explain here. These categories are introduced to support the different instantiation schemes discussed in this paper, altering the behaviour of the typing rules accordingly. Monotypes τ represent simple ground types without any polymorphism. They are entirely standard, incorporating type variables a, functions $\tau_1 \to \tau_2$ and saturated type constructors $T\,\overline{\tau}$[6]. On the other side of the spectrum are type schemes σ, which can be fully polymorphic, with forall binders on the top-level, and nested on both sides of the function arrow.

So far, none of these definitions depend on the instantiation flavour. This changes with instantiated types ρ. These types cannot have any top-level polymorphism, as their quantified type variables have already been instantiated. However, depending on whether instantiation happens shallowly or deeply, they may or may not feature nested foralls on the right of function arrows. This dependency on the depth δ of type instantiation is denoted using an instantiated result type ϕ^δ on the right of the function arrow. This type becomes a full type scheme σ under shallow instantiation ($\mathcal{S}$)—thus allowing foralls on the right of the function arrow—and becomes an instantiated type ρ under deep instantiation ($\mathcal{D}$)—thus disallowing foralls on the right of the function arrow. No matter the instantiation flavour, function arguments within an instantiated type ρ can still be polymorphic, as instantiating a function does not affect polymorphism to the left of an arrow. We also have synthesised types η^ϵ to denote the output of the type synthesis judgement $\Gamma \vdash e \Rightarrow \eta^\epsilon$, which infers a type from an expression. This type depends on the eagerness ϵ of type instantiation: under lazy instantiation ($\mathcal{L}$) inference can produce full type schemes σ, but under eager instantiation ($\mathcal{E}$) synthesised types η^ϵ must be instantiated types ρ: any top-level quantified variable would have been instantiated away.

Finally, an argument descriptor ψ represents a type synthesised from analysing a function argument pattern. While this might not seem very useful in the IPLC (as they are just monotypes), argument descriptors will be extended beyond regular types in the next section. Descriptors are assembled into type schemes σ with the $type\,(\overline{\psi}; \sigma_0 \sim \sigma)$ judgement, in Figure 5.

Expressions. Expressions e are standard, except for **let**-expressions, which are modelled on the syntax of Haskell. These contain a single (non recursive) declaration *decl*, which may optionally have a type signature $x : \sigma$, followed by the definition $\overline{x\,\overline{\pi}_i = e_i}^{\,i}$. The patterns $\overline{\pi}$ on the left of the equals sign can each be either a simple variable x or a saturated data constructor $K\,\overline{\pi}$.

Contexts. Typing contexts Γ are entirely standard, storing both the term variables x with their types and the type variables a in scope; these type variables may not appear in the term (remember: this is an entirely *implicit* calculus), but they may be in scope in types. The type constructors

[6]We work only with saturated type constructors, as we wish to avoid complications arising from a kind system. Such concerns would be orthogonal to our main object of study: instantiation.

$\boxed{\Gamma \vdash e \Rightarrow \eta^\epsilon}$ *(Term Type Synthesis)*

$$\textsc{Tm-InfVar}\ \frac{x : \sigma \in \Gamma \qquad \Gamma \vdash \sigma \xrightarrow{inst\ \delta} \eta^\epsilon}{\Gamma \vdash x \Rightarrow \eta^\epsilon} \qquad \textsc{Tm-InfCon}\ \frac{K : \overline{a}; \overline{\sigma}; T \in \Gamma \qquad \Gamma \vdash \forall\, \overline{a}.\overline{\sigma} \to T\,\overline{a} \xrightarrow{inst\ \delta} \eta^\epsilon}{\Gamma \vdash K \Rightarrow \eta^\epsilon}$$

$$\textsc{Tm-InfAbs}\ \frac{\Gamma, x : \tau \vdash e \Rightarrow \eta^\epsilon}{\Gamma \vdash \lambda x.e \Rightarrow \tau \to \eta^\epsilon}$$

$$\textsc{Tm-InfApp}\ \frac{\Gamma \vdash e_1 \Rightarrow \eta^\epsilon \qquad \Gamma \vdash \eta^\epsilon \xrightarrow{inst\ \delta} \sigma_1 \to \phi_2^\delta \qquad \Gamma \vdash e_2 \Leftarrow \sigma_1 \qquad \Gamma \vdash \phi_2^\delta \xrightarrow{inst\ \mathcal{S}} \eta_2^\epsilon}{\Gamma \vdash e_1\ e_2 \Rightarrow \eta_2^\epsilon}$$

$$\textsc{Tm-InfLet}\ \frac{\Gamma \vdash decl \Rightarrow \Gamma' \qquad \Gamma' \vdash e \Rightarrow \eta^\epsilon}{\Gamma \vdash \textbf{let}\ decl\ \textbf{in}\ e \Rightarrow \eta^\epsilon}$$

$$\textsc{Tm-InfAnn}\ \frac{\Gamma \vdash e \Leftarrow \sigma \qquad \Gamma \vdash \sigma \xrightarrow{inst\ \delta} \eta^\epsilon}{\Gamma \vdash e : \sigma \Rightarrow \eta^\epsilon}$$

$\boxed{\Gamma \vdash e \Leftarrow \sigma}$ *(Term Type Checking)*

$$\textsc{Tm-CheckAbs}\ \frac{\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \sigma_1 \to \phi_2^\delta; \Gamma' \qquad \Gamma', x : \sigma_1 \vdash e \Leftarrow \phi_2^\delta}{\Gamma \vdash \lambda x.e \Leftarrow \sigma}$$

$$\textsc{Tm-CheckLet}\ \frac{\Gamma \vdash decl \Rightarrow \Gamma' \qquad \Gamma' \vdash e \Leftarrow \sigma}{\Gamma \vdash \textbf{let}\ decl\ \textbf{in}\ e \Leftarrow \sigma}$$

$$\textsc{Tm-CheckInf}\ \frac{\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho; \Gamma' \qquad \Gamma' \vdash e \Rightarrow \eta^\epsilon \qquad \Gamma' \vdash \eta^\epsilon \xrightarrow{inst\ \delta} \rho \qquad e \neq \lambda, let}{\Gamma \vdash e \Leftarrow \sigma}$$

Fig. 2. Term Typing for Implicit Polymorphic λ-Calculus

and data constructors are stored in a static context Σ, which forms the basis of typing contexts Γ. This static context contains the data type definitions by storing both type constructors $T\,\overline{a}$ and data constructors $K : \overline{a}; \overline{\sigma}; T$. Data constructor types contain the list of quantified variables $\overline{a}$, the argument types $\overline{\sigma}$, and the resulting type T; when $K : \overline{a}; \overline{\sigma}; T$, then the use of K in an expression would have type $\forall\, \overline{a}.\overline{\sigma} \to T\,\overline{a}$, abusing syntax slightly to write a list of types $\overline{\sigma}$ to the left of an arrow.

6.1 Typing rules

Figures 2, 3, 4 and 5 show the typing rules for our the IPLC. In order to support both type synthesis and checking, we use a bidirectional type system. We review the high-level role of the judgements and then examine details.

Type Synthesis. Synthesis for expressions is performed by the $\Gamma \vdash e \Rightarrow \eta^\epsilon$ relation: "Under typing context Γ, the expression e is assigned the type η^ϵ". As mentioned previously, the shape of the output type η^ϵ is parameterised over the eagerness of type instantiation ϵ. Under lazy instantiation, type synthesis produces a type scheme σ; under eager instantiation, the output will be an instantiated type ρ. Similar to expressions, pattern synthesis is performed by the $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta$ relation: "Under typing context Γ, patterns $\overline{\pi}$ are assigned the argument descriptors $\overline{\psi}$, producing the context extension Δ". Declaration synthesis is performed by the $\Gamma \vdash decl \Rightarrow \Gamma'$ relation: "The declaration *decl* extends the typing context Γ into Γ'".

Type Checking. Checking for expressions is performed by the $\Gamma \vdash e \Leftarrow \sigma$ relation: "Under typing context Γ, the expression e is checked to have the type σ". The type serves as an input here: an

$\boxed{\Gamma \vdash \mathit{decl} \Rightarrow \Gamma'}$ *(Declaration Checking)*

$$\textsc{Decl-NoAnnSingle}\ \frac{\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \quad \Gamma, \Delta \vdash e \Rightarrow \eta^\epsilon \quad \mathit{type}\,(\overline{\psi}; \eta^\epsilon \sim \sigma) \quad \overline{a} = f_v(\sigma) \setminus \mathit{dom}\,(\Gamma)}{\Gamma \vdash x\,\overline{\pi} = e \Rightarrow \Gamma, x : \forall\,\overline{a}.\sigma}$$

$$\textsc{Decl-NoAnnMulti}\ \frac{i > 1 \quad \overline{\Gamma \vdash^P \overline{\pi}_i \Rightarrow \overline{\psi}; \Delta}^{\,i} \quad \overline{\Gamma, \Delta \vdash e_i \Rightarrow \eta_i^\epsilon}^{\,i} \quad \overline{\Gamma_i' \vdash \eta_i^\epsilon \xrightarrow{\mathit{inst}\ \delta} \rho}^{\,i} \quad \mathit{type}\,(\overline{\psi}; \rho \sim \sigma) \quad \overline{a} = f_v(\sigma) \setminus \mathit{dom}\,(\Gamma) \quad \sigma' = \forall\,\overline{a}.\sigma}{\Gamma \vdash \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma, x : \sigma'}$$

$$\textsc{Decl-Ann}\ \frac{\overline{\Gamma \vdash^P \overline{\pi}_i \Leftarrow \sigma \Rightarrow \sigma_i'; \Delta}^{\,i} \quad \overline{\Gamma, \Delta \vdash e_i \Leftarrow \sigma_i'}^{\,i}}{\Gamma \vdash x : \sigma; \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma, x : \sigma}$$

Fig. 3. Declaration Checking for Implicit Polymorphic λ-Calculus

$\boxed{\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta}$ *(Pattern Synthesis)*

$$\textsc{Pat-InfEmpty}\ \frac{}{\Gamma \vdash^P \cdot \Rightarrow \cdot; \cdot} \qquad \textsc{Pat-InfVar}\ \frac{\Gamma, x : \tau_1 \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta}{\Gamma \vdash^P x, \overline{\pi} \Rightarrow \tau_1, \overline{\psi}; x : \tau_1, \Delta}$$

$$\textsc{Pat-InfCon}\ \frac{K : \overline{a}_0; \overline{\sigma}_0; T \in \Gamma \quad \Gamma \vdash^P \overline{\pi} \Leftarrow [\overline{\tau}_0/\overline{a}_0]\,(\overline{\sigma}_0 \to T\,\overline{a}_0) \Rightarrow T\,\overline{\tau}; \Delta_1 \quad \Gamma, \Delta_1 \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}; \Delta_2}{\Gamma \vdash^P (K\ \overline{\pi}), \overline{\pi}' \Rightarrow T\,\overline{\tau}, \overline{\psi}; \Delta_1, \Delta_2}$$

$\boxed{\Gamma \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta}$ *(Pattern Checking)*

$$\textsc{Pat-CheckEmpty}\ \frac{}{\Gamma \vdash^P \cdot \Leftarrow \sigma \Rightarrow \sigma; \cdot} \qquad \textsc{Pat-CheckVar}\ \frac{\Gamma, x : \sigma_1 \vdash^P \overline{\pi} \Leftarrow \sigma_2 \Rightarrow \sigma'; \Delta}{\Gamma \vdash^P x, \overline{\pi} \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma'; x : \sigma_1, \Delta}$$

$$\textsc{Pat-CheckCon}\ \frac{K : \overline{a}_0; \overline{\sigma}_0; T \in \Gamma \quad \Gamma \vdash \sigma_1 \xrightarrow{\mathit{inst}\ \delta} \rho_1 \quad \Gamma \vdash^P \overline{\pi} \Leftarrow [\overline{\tau}_0/\overline{a}_0]\,(\overline{\sigma}_0 \to T\,\overline{a}_0) \Rightarrow \rho_1; \Delta_1 \quad \Gamma, \Delta_1 \vdash^P \overline{\pi}' \Leftarrow \sigma_2 \Rightarrow \sigma_2'; \Delta_2}{\Gamma \vdash^P (K\ \overline{\pi}), \overline{\pi}' \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma_2'; \Delta_1, \Delta_2}$$

$$\textsc{Pat-CheckForall}\ \frac{\Gamma, a \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \quad \overline{\pi} \neq \cdot}{\Gamma \vdash^P \overline{\pi} \Leftarrow \forall\,a.\sigma \Rightarrow \sigma'; a, \Delta}$$

Fig. 4. Pattern Typing for Implicit Polymorphic λ-Calculus

algorithmic interpretation of this relation would simply return a pass/fail result. Similarly, pattern checking is performed by the $\Gamma \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta$ relation: "Under typing context Γ, patterns $\overline{\pi}$ match element-wise against σ, producing the residual type σ' and the context extension Δ".

Type Instantiation. The behaviour of both the instantiation $\Gamma \vdash \sigma \xrightarrow{\mathit{inst}\ \delta} \rho$ and skolemisation $\Gamma \vdash \sigma \xrightarrow{\mathit{skol}\ \delta} \rho; \Gamma'$ judgements depends on whether the depth δ is deep $\mathcal{D}$ or shallow $\mathcal{S}$; see Figure 5. The relations gather the binders of σ and instantiates or skolemises these with monotypes $\overline{\tau}$ or fresh type variables, respectively. Both invoke $\mathit{binders}^\delta(\sigma) = \overline{a}; \rho$, which splits a type scheme σ into a list of its bound type variables $\overline{a}$ and the remaining instantiated type ρ. Under shallow instantiation

$\boxed{binders^{\delta}(\sigma) = \overline{a}; \rho}$ *(Binders)*

Bndr-Mono
$$\frac{}{binders^{\mathcal{D}}(\tau) = \cdot; \tau}$$

Bndr-Function
$$\frac{binders^{\mathcal{D}}(\sigma_2) = \overline{a}; \rho_2}{binders^{\mathcal{D}}(\sigma_1 \to \sigma_2) = \overline{a}; \sigma_1 \to \rho_2}$$

Bndr-Forall
$$\frac{binders^{\mathcal{D}}(\sigma) = \overline{b}; \rho}{binders^{\mathcal{D}}(\forall\, \overline{a}.\sigma) = \overline{a}, \overline{b}; \rho}$$

Bndr-Shallow
$$\frac{}{binders^{\mathcal{S}}(\forall\, \overline{a}.\rho) = \overline{a}; \rho}$$

$\boxed{\Gamma \vdash \sigma \xrightarrow{inst\ \delta} \rho}$ *(Type instantiation)*

Inst-Inst
$$\frac{binders^{\delta}(\sigma) = \overline{a}; \rho}{\Gamma \vdash \sigma \xrightarrow{inst\ \delta} [\overline{\tau}/\overline{a}]\ \rho}$$

$\boxed{\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho; \Gamma'}$ *(Type skolemisation)*

Skol-Skol
$$\frac{binders^{\delta}(\sigma) = \overline{a}; \rho}{\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho; \Gamma, \overline{a}}$$

$\boxed{type\ (\overline{\psi}; \sigma \sim \sigma')}$ *(Telescope Type Construction)*

Type-Empty
$$\frac{}{type\ (\cdot; \sigma \sim \sigma)}$$

Type-Var
$$\frac{type\ (\overline{\psi}; \sigma_2 \sim \sigma_2')}{type\ (\tau_1, \overline{\psi}; \sigma_2 \sim \tau_1 \to \sigma_2')}$$

Fig. 5. Type Instantiation and Skolemisation

(rule Bndr-Shallow), this relation just returns all type variables bound at the top of the input type scheme. Under deep instantiation however (which is based directly on relations in Peyton Jones et al. [2007, Section 4.6.2]), this gathers variables bound both at the top level and on the right of function arrows.

The instantiation and skolemisation relations as presented here work properly only when going from a type scheme σ to an instantiated type ρ. However, we see some rules relate other types. For example, rule Tm-InfVar instantiates the type scheme σ assigned to a variable from a context to a synthesised type η^{ϵ} (the ϵ is included to remind us on the importance of the eagerness parameter). Because a synthesised type *is* a type scheme when we work with lazy instantiation, this instantiation in rule Tm-InfVar is actually a no-op in that scenario; however, under eager instantiation, a synthesised type is the same as an instantiated type (with no $\forall$s at the top), and we must actually instantiate. This is why the use of instantiation in rule Tm-InfVar is coloured in emerald. Conversely, examine rule Tm-InfApp. This rule handles when we are applying one expression to another. If we have instantiated eagerly, then we need take no action before applying the function to the argument: any type variables in the function's type have already been instantiated. Under lazy instantiation, however, function application is a point where we might need to finally instantiate. Accordingly, the rule instantiates the synthesised η^{ϵ} to $\sigma_1 \to \phi_2^{\delta}$, which is an instantiated type, meta variable ρ. Recall that, under eager instantiation, a synthesised type η^{ϵ} is *already* an instantiated type ρ, and then this instantiation is a no-op. The instantiation applies only under the lazy scheme, and thus is coloured in lavender.

Data constructors. Typing a data constructor (rule Tm-InfCon) works identically to typing variables, after assembling the data constructor type from its pieces.

λ-expressions. Synthesis for abstractions (rule Tm-InfAbs) works as usual, by assigning a monotype τ to the variable x when checking the expression e. Type checking for abstractions (rule Tm-CheckAbs) works similarly, but also allows a full type scheme as argument type. Note that we first need to skolemise the given type σ into a function type $\sigma_1 \to \phi_2^\delta$ (if this is not possible, type checking fails). Skolemisation takes the available binders of σ and simply adds them to the context Γ'. These variables behave as skolem constants when checking e.

Function Application. Let us examine rule Tm-InfApp more closely, which synthesises the type for an application $e_1\ e_2$. First, the type of e_1 is synthesised. Under eager instantiation, this produces an instantiated type which should be a function type $\sigma_1 \to \phi_2^\delta$; if it's not, type synthesis fails. Under lazy instantiation, and as described above, the type of e_1 is now instantiated. Then we must check the argument e_2 against known type σ_1. Having checked the application itself, we must now perhaps instantiate the result type ϕ_2^δ. This would happen only with eager instantiation, and there can be variables to instantiate only when the previous eager instantiation did not find them—that is, when instantiation is shallow. Thus the final premise to rule Tm-InfApp applies only in the eager, shallow scheme, marked in emerald and with stripes.

Mode Changing. Going from type synthesis to type checking is achieved using an annotation. Typing an annotated expression $e : \sigma$ (rule Tm-InfAnn) works by checking the expression e under the given type σ. Note that under eager instantiation, the type for this annotated expression is not the type scheme σ, but an instantiated form of this type.

Going from checking to synthesis mode is also allowed (rule Tm-CheckInf), but only when e is not a lambda or let expression, and thus when no other type checking rule would apply. We first skolemise the given type σ to bind any new type variables in Γ'. We then infer a type for the expression e and match it against the skolemised type ρ. Note that under lazy instantiation, the synthesised type still needs to be instantiated in order to match ρ.

Let Binders. As mentioned previously, **let**-expressions **let** *decl* **in** e define a single variable, with or without a type signature. See Figure 3, which defines the judgement $\Gamma \vdash \mathit{decl} \Rightarrow \Gamma'$. This judgement examines a declaration to produce a new context Γ', extended with a binding for the declared variable.

Rules Decl-NoAnnSingle and Decl-NoAnnMulti distinguish between a single equation without a type signature and multiple equations. In the former case, we synthesise the types of the patterns using the $\vdash^P$ judgement and then the type of the right-hand side. We assemble the complete type with *type*, and then generalise. The multiple-equation case is broadly similar, synthesising types for the patterns (note that each equation must yield the *same* types $\overline{\psi}$) and then synthesising types for the right-hand side. These types are then instantiated (only necessary under lazy instantiation—eager instantiation would have already done this step). This additional instantiation step is the only difference between the single-equation case and the multiple-equation case. The reason is that rule EDecl-NoAnnMulti needs to construct a single type that subsumes the types of every branch. Following GHC, we simplify this process by first instantiating the types.

Rule Decl-Ann checks a declaration with a type signature. It works by first checking the patterns $\overline{\pi}_i$ on the left of the equals sign against the provided type σ. The right-hand sides e_i are then checked against the remaining type σ_i'.

Patterns. The pattern synthesis relation $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta$ is presented in Figure 4. As the full type is not yet available, it produces argument descriptors $\overline{\psi}$ and a typing context extension Δ. Inferring a type for a variable pattern (rule Pat-InfVar) works similarly to expression inference, by constructing a monotype and placing it in the context. Inferring a type for a data constructor

pattern (rule Pat-InfCon) works by looking up the type $\forall\, \overline{a}_0.\overline{\sigma}_0 \to T\, \overline{a}_0$ of the constructor K in the typing context, and checking the applied patterns $\overline{\pi}$ under this instantiated type. Note that as there are no forall binders to the right of an arrow, instantiation simply needs to supply types $\overline{\tau}_0$ for the top level $\overline{a}_0$ variables. The remaining type should be the result type for the constructor, meaning that the constructor always needs to be fully applied. Finally, the remaining patterns are typed under this extended environment.

Checking a variable pattern in rule Pat-CheckVar works by placing the variable in the typing context, with its type σ_1, extracted from the known type of the function we are checking. The same holds for checking against a polytype (rule Pat-CheckForall), which gets skolemised by placing it in the typing context. Note that in order for the rules to be syntax-directed, rule Pat-CheckForall does not apply when the patterns are empty. Checking a data constructor pattern (rule Pat-CheckCon) works similarly to synthesis mode. The type $\forall\, \overline{a}_0.\overline{\sigma}_0 \to T\, \overline{a}_0$ of the constructor is looked up in the typing context. Then, the patterns $\overline{\pi}$ are checked under an instantiated form of $\forall\, \overline{a}_0.\overline{\sigma}_0 \to T\, \overline{a}_0$. The remaining type should then be equal to an instantiated form of the provided type σ_1. Finally, the remaining patterns $\overline{\pi}'$ are checked under the extended context[7].

6.2 Alternative Formalisation

Section 7 presents an extension of our λ-calculus formalisation to a more advanced language, mixing implicit and explicit polymorphism. Notably, this covers features like recursive let bindings, visible type application and user-defined provenance. However, under eager instantiation, explicit type application poses an issue with our current formalisation, as it requires temporarily postponing type instantiation during inference. Here, we replace our typing rules based on nested applications from Section 6.1 with an alternative formalisation inspired by the structure from Serrano et al. [2020]. The reason for this will become clear in Section 7, as this representation can easily be extended with explicit type application.

Figure 6 shows the updated system. These rules implement the *same* language as the rules we reviewed previously, but it is recast in a way easier to extend later.

Application. In order to simplify the introduction of explicit type application, applications are now modelled as applying a head h to a (maximally long) list of arguments $\overline{arg}$. This concept of modelling applications is inspired by Serrano et al. [2020], which also features visible type application and eager instantiation. The main idea is that under eager instantiation, type instantiation for the head is postponed until it has been applied to its arguments. A head h is thus defined to be either a variable x, a data constructor K, an annotated expression $e : \sigma$ or a simple expression e. This last form will not be typed with a type scheme under eager instantiation—that is, we will not be able to use explicit instantiation—but is required to enable application of a lambda expression. At the moment, as we have only term application (not type application), an argument arg is defined to just be an expression e. This will be extended in Section 7 to include type application as well.

When inferring a type for an application $h\, \overline{arg}$ (rule Etm-InfApp), a type scheme σ is inferred for the head h. The arguments $\overline{arg}$ are then checked under this type σ, producing a residual type σ'[8]. Finally, after the head has been applied to all its arguments, the remaining type σ' is instantiated (under eager instantiation), producing a type η^{ϵ} for the final expression.

Note that as applications are always typed in this fashion, head typing needs only a synthesis mode, and argument typing needs to be defined only for checking mode. Type synthesis for heads $\Gamma \vdash^{H} h \Rightarrow \sigma$ works identically to expressions from the basic system, except that the types are not

[7] Extending the context for later patterns is not used in the basic system, but it would be required for extensions like view patterns.

[8] The application judgement is inspired by Dunfield and Krishnaswami [2013].

$$
\begin{array}{lcll}
e & ::= & h\,\overline{arg} \mid \lambda x.e & \textit{Expression} \\
 & \mid & \textbf{let}\ decl\ \textbf{in}\ e & \\
h & ::= & x \mid K \mid e : \sigma \mid e & \textit{Application head} \\
arg & ::= & e & \textit{Application argument}
\end{array}
$$

$\boxed{\Gamma \vdash^H h \Rightarrow \sigma}$ *(Head Type Synthesis)*

$$\textsc{H-Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash^H x \Rightarrow \sigma} \qquad \textsc{H-Con}\ \frac{K : \overline{a}; \overline{\sigma}; T \in \Gamma}{\Gamma \vdash^H K \Rightarrow \forall\, \overline{a}.\overline{\sigma} \to T\,\overline{a}} \qquad \textsc{H-Ann}\ \frac{\Gamma \vdash e \Leftarrow \sigma}{\Gamma \vdash^H e : \sigma \Rightarrow \sigma} \qquad \textsc{H-Inf}\ \frac{\Gamma \vdash e \Rightarrow \sigma}{\Gamma \vdash^H e \Rightarrow \sigma}$$

$\boxed{\Gamma \vdash e \Rightarrow \eta^\epsilon}$ *(Term Type Synthesis)*

$$\textsc{ETm-InfAbs}\ \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \eta_2^\epsilon}{\Gamma \vdash \lambda x.e \Rightarrow \tau_1 \to \eta_2^\epsilon} \qquad \textsc{ETm-InfApp}\ \frac{\Gamma \vdash^H h \Rightarrow \sigma \quad \Gamma \vdash^A \overline{arg} \Leftarrow \sigma \Rightarrow \sigma' \quad \Gamma \vdash \sigma' \xrightarrow{\ inst\ \delta\ } \eta^\epsilon}{\Gamma \vdash h\,\overline{arg} \Rightarrow \eta^\epsilon} \qquad \textsc{ETm-InfLet}\ \frac{\Gamma \vdash decl \Rightarrow \Gamma' \quad \Gamma' \vdash e \Rightarrow \eta^\epsilon}{\Gamma \vdash \textbf{let}\ decl\ \textbf{in}\ e \Rightarrow \eta^\epsilon}$$

$\boxed{\Gamma \vdash e \Leftarrow \sigma}$ *(Term Type Checking)*

$$\textsc{ETm-CheckAbs}\ \frac{\Gamma \vdash \sigma \xrightarrow{\ skol\ S\ } \sigma_1 \to \sigma_2; \Gamma_1 \quad \Gamma_1, x : \sigma_1 \vdash e \Leftarrow \sigma_2}{\Gamma \vdash \lambda x.e \Leftarrow \sigma} \qquad \textsc{ETm-CheckLet}\ \frac{\Gamma \vdash decl \Rightarrow \Gamma' \quad \Gamma' \vdash e \Leftarrow \sigma}{\Gamma \vdash \textbf{let}\ decl\ \textbf{in}\ e \Leftarrow \sigma}$$

$$\textsc{ETm-CheckInf}\ \frac{\Gamma \vdash \sigma \xrightarrow{\ skol\ \delta\ } \rho; \Gamma_1 \quad \Gamma_1 \vdash e \Rightarrow \eta^\epsilon \quad \Gamma_1 \vdash \eta^\epsilon \xrightarrow{\ inst\ \delta\ } \rho \quad e \neq \lambda, \Lambda, let}{\Gamma \vdash e \Leftarrow \sigma}$$

$\boxed{\Gamma \vdash^A \overline{arg} \Leftarrow \sigma \Rightarrow \sigma'}$ *(Argument Type Checking)*

$$\textsc{Arg-Empty}\ \frac{}{\Gamma \vdash^A \cdot \Leftarrow \sigma \Rightarrow \sigma} \qquad \textsc{Arg-App}\ \frac{\Gamma \vdash e \Leftarrow \sigma_1 \quad \Gamma \vdash^A \overline{arg} \Leftarrow \sigma_2 \Rightarrow \sigma'}{\Gamma \vdash^A e, \overline{arg} \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma'} \qquad \textsc{Arg-Inst}\ \frac{\Gamma \vdash^A e, \overline{arg} \Leftarrow [\tau_1/a]\,\sigma_2 \Rightarrow \sigma_3}{\Gamma \vdash^A e, \overline{arg} \Leftarrow \forall\, a.\sigma_2 \Rightarrow \sigma_3}$$

Fig. 6. Alternative formulation for the Implicit Polymorphic λ-Calculus

instantiated. Argument type checking $\Gamma \vdash^A \overline{arg} \Leftarrow \sigma \Rightarrow \sigma'$ bears a close resemblance to pattern checking, taking a typing context Γ, a list of arguments $\overline{arg}$ and a type scheme σ and producing a residual type σ'. Concretely, this means that bound type variables in the given type are instantiated (rule Arg-Inst) and the list of arguments are type checked to have the given type.

7 MIXING IMPLICIT AND EXPLICIT TYPES

Figure 7 shows the extension of the language syntax to handle mixed implicit and explicit features. We call this extended system Mixed Polymorphic λ-calculus, or MPLC. Our formalisation is based on Eisenberg et al. [2016] and Serrano et al. [2020], and extends the IPLC with explicit type instantiation and abstraction, along with user-defined type variable provenance. Application arguments are extended with visible type application arguments $@\tau$, and both expressions and patterns are extended with explicit type abstraction $\Lambda a.e$ and $@\tau$, respectively. Type schemes σ now allow

$$\sigma ::= \ldots \mid \forall\, \overline{\{a\}}.\sigma \quad \textit{Type scheme}$$
$$\psi ::= \ldots \mid @a \quad \textit{Arg. descriptor}$$

$$e ::= \ldots \mid \Lambda a.e \quad \textit{Expression}$$
$$\pi ::= \ldots \mid @\sigma \quad \textit{Pattern}$$
$$arg ::= \ldots \mid @\sigma \quad \textit{Application argument}$$

$\boxed{\Gamma \vdash^A \overline{arg} \Leftarrow \sigma \Rightarrow \sigma'}$ *(Argument Type Checking)*

Arg-TyApp
$$\frac{\Gamma \vdash^A \overline{arg} \Leftarrow [\sigma_1/a]\,\sigma_2 \Rightarrow \sigma_3}{\Gamma \vdash^A @\sigma_1, \overline{arg} \Leftarrow \forall\, a.\sigma_2 \Rightarrow \sigma_3}$$

Arg-InfInst
$$\frac{\Gamma \vdash^A \overline{arg} \Leftarrow [\tau_1/a]\,\sigma_2 \Rightarrow \sigma_3}{\Gamma \vdash^A \overline{arg} \Leftarrow \forall\, \{a\}.\sigma_2 \Rightarrow \sigma_3}$$

$\boxed{\Gamma \vdash e \Rightarrow \eta^\epsilon}$ *(Term Type Synthesis)*

ETm-InfTyAbs
$$\frac{\Gamma, a \vdash e \Rightarrow \eta_1^\epsilon \qquad \Gamma \vdash \forall\, a.\eta_1^\epsilon \xrightarrow{inst\ \delta} \eta_2^\epsilon}{\Gamma \vdash \Lambda a.e \Rightarrow \eta_2^\epsilon}$$

$\boxed{\Gamma \vdash e \Leftarrow \sigma}$ *(Term Type Checking)*

ETm-CheckTyAbs
$$\frac{\sigma = \forall\, \overline{\{a\}}.\forall\, a.\sigma' \qquad \Gamma, \overline{a}, a \vdash e \Leftarrow \sigma'}{\Gamma \vdash \Lambda a.e \Leftarrow \sigma}$$

$\boxed{type\,(\overline{\psi}; \sigma \sim \sigma')}$ *(Telescope Construction)*

Type-TyVar
$$\frac{type\,(\overline{\psi}; \sigma \sim \sigma')}{type\,(@a, \overline{\psi}; \sigma \sim \forall\, a.\sigma')}$$

ETm-CheckAbs
$$\frac{\Gamma \vdash \sigma \xrightarrow{skol\ \mathcal{S}} \sigma_1 \to \sigma_2; \Gamma_1 \qquad \Gamma_1, x : \sigma_1 \vdash e \Leftarrow \sigma_2}{\Gamma \vdash \lambda x.e \Leftarrow \sigma}$$

$\boxed{\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta}$ *(Pattern Synthesis)*

Pat-InfTyVar
$$\frac{\Gamma, a \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta}{\Gamma \vdash^P @a, \overline{\pi} \Rightarrow @a, \overline{\psi}; a, \Delta}$$

Pat-InfCon
$$\frac{K : \overline{a}_0; \overline{\sigma}_0; T \in \Gamma \qquad \Gamma \vdash^P \overline{\pi} \Leftarrow [\overline{\sigma}_1, \overline{\tau}_0/\overline{a}_0]\,(\overline{\sigma}_0 \to T\,\overline{a}_0) \Rightarrow T\,\overline{\tau}; \Delta_1 \qquad \Gamma, \Delta_1 \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}; \Delta_2}{\Gamma \vdash^P (K\,@\overline{\sigma}_1\ \overline{\pi}), \overline{\pi}' \Rightarrow T\,\overline{\tau}, \overline{\psi}; \Delta_1, \Delta_2}$$

$\boxed{\Gamma \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta}$ *(Pattern Checking)*

Pat-CheckTyVar
$$\frac{\Gamma, a \vdash^P \overline{\pi} \Leftarrow [a/b]\,\sigma_1 \Rightarrow \sigma_2; \Delta}{\Gamma \vdash^P @a, \overline{\pi} \Leftarrow \forall\, b.\sigma_1 \Rightarrow \sigma_2; a, \Delta}$$

Pat-CheckCon
$$\frac{K : \overline{a}_0; \overline{\sigma}_0; T \in \Gamma \qquad \Gamma \vdash \sigma_1 \xrightarrow{inst\ \delta} \rho_1 \qquad \Gamma \vdash^P \overline{\pi} \Leftarrow [\overline{\sigma}_1, \overline{\tau}_0/\overline{a}_0]\,(\overline{\sigma}_0 \to T\,\overline{a}_0) \Rightarrow \rho_1; \Delta_1 \qquad \Gamma, \Delta_1 \vdash^P \overline{\pi}' \Leftarrow \sigma_2 \Rightarrow \sigma_2'; \Delta_2}{\Gamma \vdash^P (K\,@\overline{\sigma}_1\ \overline{\pi}), \overline{\pi}' \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma_2'; \Delta_1, \Delta_2}$$

Pat-CheckForall
$$\frac{\Gamma, a \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \qquad \overline{\pi} \neq \cdot \ and\ \overline{\pi} \neq @\sigma, \overline{\pi}'}{\Gamma \vdash^P \overline{\pi} \Leftarrow \forall\, a.\sigma \Rightarrow \sigma'; a, \Delta}$$

Pat-CheckInfForall
$$\frac{\Gamma, a \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \qquad \overline{\pi} \neq \cdot}{\Gamma \vdash^P \overline{\pi} \Leftarrow \forall\, \{a\}.\sigma \Rightarrow \sigma'; a, \Delta}$$

Fig. 7. The Mixed Polymorphic λ-calculus (Extension of Figures 1, 3, 5, and 6)

inferred variables $\forall\, \overline{\{a\}}.\sigma$. Note, however, that the context Γ does not differentiate between specified and inferred variables, as the provenance of variables no longer has any impact after skolemising. Argument types ψ are extended with type abstractions $@a$, used in a function definition like $f\ @a\ x = (x : a)$ and which corresponds to a polytype $\forall\, a.\sigma$ when assembled with *type*. The *binders* operation is updated to extract out inferred type variables; and the declaration typing relation receives a minor update, marking generalised variables as inferred. Both updated definitions can be found in Appendices C and D, respectively.

Explicit Type Instantiation. See rule Arg-TyApp, which shows the new argument type checking relation. The alternative formalisation of the language presented in Section 6.2 reduces the introduction of explicit type application—by all accounts a non-trivial language feature—to the addition of a single typing rule. Note that inferred type variables (as presented in Section 4), handled in rule Arg-InfInst, are unconditionally instantiated.

Explicit Type Abstraction. The definition of patterns π now includes type arguments $@\tau$, and expressions e include type abstraction $\Lambda a.e$. Programmers can now explicitly abstract over type variables in both lambda and let expressions.

When synthesising a type for an explicit type abstraction $\Lambda a.e$ (rule ETm-InfTyAbs), the expression e is typed under an extended environment, and the resulting forall type is instantiated under eager instantiation. Type checking (rule ETm-CheckTyAbs) skolemises any top level inferred binders, and recursively checks e under an extended environment. Note that in order to support explicit type abstraction inside a lambda binder $\lambda x.\Lambda a.e$, rule ETm-CheckAbs has been slightly altered to never skolemise after the arrow (even under deep instantiation).

Type abstraction $@a$ in both synthesis and checking mode (rule Pat-InfTyVar and rule Pat-CheckTyVar respectively) produces a type argument descriptor $@a$ and extends the typing environment. In data constructor patterns (rule Pat-InfCon and rule Pat-CheckCon), full type schemes $\overline{\sigma}_1$ are allowed in patterns, where they are used to instantiate the variables $\overline{a}_0$ (possibly extended with guessed monotypes $\overline{\tau}_0$, if there are not enough $\overline{\sigma}_1$). Consider, for example, $f\ (\mathit{Just}\ @\mathit{Int}\ x) = x + 1$, where the $@\mathit{Int}$ refines the type of *Just*, which in turn assigns x the type *Int*. Note that pattern checking allows skolemising bound type variables (rule Pat-CheckInfForall), but only when the patterns are not empty in order not to lose syntax-directedness of the rules. The same holds for rule Pat-CheckForall, which only applies when no other rules match.

8 EVALUATION

This section evaluates the impact of the type instantiation flavour on the stability of the programming language. To this end, we define a set of eleven properties, based on the informal definition of stability from Section 2. Every property is analysed against the four instantiation flavours, the results of which are shown in Table 1, which also cross-references the examples from Section 5.3. Formal proofs for each of the properties are provided in Appendix E.

We do not investigate the type safety of our formalisms, as both the IPLC and MPLC are subsets of System F. We can thus be confident that programs in either language can be assigned a sensible runtime semantics without‘going wrong.

8.1 Contextual Equivalence

Following the approach of GHC, rather than providing a dynamic operational semantics of our type system directly, we instead define an elaboration of the surface language presented in this paper to explicit System F, our core language. It is important to remark that elaborating deep instantiation into this core language, involves semantics-changing η-expansion. This allows us to understand

Sim.	Ex.		Properties		$\mathcal{E}$ $\mathcal{S}$	$\mathcal{E}$ $\mathcal{D}$	$\mathcal{L}$ $\mathcal{S}$	$\mathcal{L}$ $\mathcal{D}$
1	1, 2, 3	1	Let inlining	Static Sem.	✓	✓	✓	✓
	1, 2, 3	2	Let extraction	Static Sem.	✗	✗	✓	✓
		3		Dynamic Sem.	✓	✗	✓	✗
2		4	Signature prop	Static Sem.	✗	✗	✗	✗
	4	4b	*restricted*		✗	✗	✓	✓
		5		Dynamic Sem.	✓	✗	✓	✗
3	5	6	Type signatures	Dynamic Sem.	✓	✗	✓	✗
4	6	7	Pattern inlining	Static Sem.	✗	✗	✓	✓
		8		Dynamic Sem.	✓	✗	✓	✓
	6	9	Pattern extraction	Static Sem.	✗	✗	✓	✓
5	7	10	Single/multi	Static Sem.	✓	✓	✗	✗
6	8	11	η-expansion	Static Sem.	✗	✗	✗	✗
		11b	*restricted*		✗	✓	✗	✗

Table 1. Property Overview

the behaviour of Example 5, *swizzle*, which demonstrates a change in dynamic semantics arising from a type signature. This change is caused by η-expansion, observable only in the core language.

The definition of this core language and the elaboration from MPLC to core are in Appendix D. The meta variable t refers to core terms, and $\rightsquigarrow$ denotes elaboration. In the core language, η-expansion is expressed through the use of an expression wrapper $\dot{t}$, an expression with a hole, which retypes the expression that gets filled in. The full details can be found in Appendix D. We now provide an intuitive definition of contextual equivalence in order to describe what it means for dynamic semantics to remain unchanged.

Definition 1 (Contextual Equivalence). *Two core expressions t_1 and t_2 are contextually equivalent, written $t_1 \simeq t_2$, if there does not exist a context that can distinguish them. That is, t_1 and t_2 behave identically in all contexts.*

Here, we understand a context to be a core expression with a hole, similar to an expression wrapper, which instantiates the free variables of the expression that gets filled in. More concretely, the expression built by inserting t_1 and t_2 to the context should either both evaluate to the same value, or both diverge. A formal definition of contextual equivalence can be found in Appendix E.2.

8.2 Properties

let-*inlining and extraction.* We begin by analysing Similarity 1, which expands to the three properties described in this subsection.

Property 1 (Let Inlining is Type Preserving).

- *If* $\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow \eta^\epsilon$ *then* $\Gamma \vdash [e_1/x]\ e_2 \Rightarrow \eta^\epsilon$
- *If* $\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Leftarrow \sigma$ *then* $\Gamma \vdash [e_1/x]\ e_2 \Leftarrow \sigma$

Property 2 (Let Extraction is Type Preserving).

- *If* $\Gamma \vdash [e_1/x]\ e_2 \Rightarrow \eta_2^\epsilon$ *and* $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ *then* $\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow \eta_2^\epsilon$
- *If* $\Gamma \vdash [e_1/x]\ e_2 \Leftarrow \sigma_2$ *and* $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ *then* $\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Leftarrow \sigma_2$

Property 3 (Let Inlining is Dynamic Semantics Preserving).

- *If* $\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Rightarrow \eta^{\epsilon} \rightsquigarrow t_1$ *and* $\Gamma \vdash [e_1/x]\ e_2 \Rightarrow \eta^{\epsilon} \rightsquigarrow t_2$ *then* $t_1 \simeq t_2$
- *If* $\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Leftarrow \sigma \rightsquigarrow t_1$ *and* $\Gamma \vdash [e_1/x]\ e_2 \Leftarrow \sigma \rightsquigarrow t_2$ *then* $t_1 \simeq t_2$

As an example for why Property 2 does not hold under eager instantiation, consider *id @Int*. Extracting the *id* function into a new **let**-binder fails to type check, because *id* will be instantiated and then re-generalised. This means that explicit type instantiation can no longer work. We discuss this further in Section 8.3.

The dynamic semantics properties (both these and later ones) struggle under deep instantiation. This is demonstrated by Example 5, *swizzle*, where we see that non-prenex quantification can cause η-expansion during elaboration and thus change dynamic semantics.

Signature Property. Similarity 2 gives rise to these properties about signatures.

Property 4 (Signature Property is Type Preserving).
If $\Gamma \vdash \overline{x\ \overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma'$ *and* $x : \sigma \in \Gamma'$ *then* $\Gamma \vdash x : \sigma; \overline{x\ \overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma'$

As an example of how this goes wrong under eager instantiation, consider the definition $x = \Lambda a.\lambda y.(y : a)$. Annotating x with its inferred type $\forall\{a\}.a \to a$ is rejected, because rule ETm-CheckTyAbs requires a *specified* quantified variable, not an *inferred* one.

However, similarly to eager evaluation, even lazy instantiation needs to instantiate the types at some point. In order to type a multi-equation declaration, a single type needs to be constructed that subsumes the types of every branch. In our type system, rule EDecl-NoAnnMulti simplifies this process by first instantiating every branch type (following the example set by GHC), thus breaking Property 4. We thus introduce a simplified version of this property, limited to single equation declarations. This raises a possible avenue of future work: parameterising the type system over the handling of multi-equation declarations.

Property 4b (Signature Property is Type Preserving (Single Equation)).
If $\Gamma \vdash x\ \overline{\pi} = e \Rightarrow \Gamma'$ *and* $x : \sigma \in \Gamma'$ *then* $\Gamma \vdash x : \sigma; x\ \overline{\pi} = e \Rightarrow \Gamma'$

Property 5 (Signature Property is Dynamic Semantics Preserving).
If $\Gamma \vdash \overline{x\ \overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t_1$ *and* $\Gamma \vdash x : \sigma; \overline{x\ \overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t_2$ *then* $t_1 \simeq t_2$

Type Signatures. Similarity 3 gives rise to the following property about dynamic semantics.

Property 6 (Type Signatures are Dynamic Semantics Preserving).
If $\Gamma \vdash x : \sigma_1; \overline{x\ \overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma_1 \rightsquigarrow x : \sigma_1 = t_1$ *and* $\Gamma \vdash x : \sigma_2; \overline{x\ \overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma_1 \rightsquigarrow x : \sigma_2 = t_2$
where $\Gamma \vdash \sigma_1 \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_1$ *and* $\Gamma \vdash \sigma_2 \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_2$, *then* $\dot{t}_1[t_1] \simeq \dot{t}_2[t_2]$

Consider $\mathbf{let}\ x : \forall\, a.Int \to a \to a; x = undefined\ \mathbf{in}\ x\ `seq`\ ()$, which diverges. Yet under deep instantiation, this version terminates: $\mathbf{let}\ x : Int \to \forall\, a.a \to a; x = undefined\ \mathbf{in}\ x\ `seq`\ ()$. Under shallow instantiation, the second program is rejected, because *undefined* cannot be instantiated to the type $Int \to \forall\, a.a \to a$, as that would be impredicative. You can find the typing rules for *undefined* and *seq* in Appendix D.1.

Pattern Inlining and Extraction. The properties in this section come from Similarity 4. Like in that similarity, we assume that the patterns are just variables (either implicit type variables or explicit term variables).

Property 7 (Pattern Inlining is Type Preserving).
If $\Gamma \vdash x\ \overline{\pi} = e_1 \Rightarrow \Gamma'$ *and* $wrap\ (\overline{\pi}; e_1 \sim e_2)$ *then* $\Gamma \vdash x = e_2 \Rightarrow \Gamma'$

The failure of pattern inlining under eager instantiation will feel similar: if we take $id\ @a\ x = x : a$, we will infer a type $\forall\ a.a \to a$. Yet if we write $id = \Lambda a.\lambda x.(x : a)$, then eager instantiation will give us the different type $\forall\ \{a\}.a \to a$.

Property 8 (Pattern Inlining / Extraction is Dynamic Semantics Preserving).
If $\Gamma \vdash x\,\overline{\pi} = e_1 \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t_1$, *wrap* $(\overline{\pi}; e_1 \sim e_2)$, *and* $\Gamma \vdash x = e_2 \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t_2$ *then* $t_1 \simeq t_2$

Property 9 (Pattern Extraction is Type Preserving).
If $\Gamma \vdash x = e_2 \Rightarrow \Gamma'$ *and wrap* $(\overline{\pi}; e_1 \sim e_2)$ *then* $\Gamma \vdash x\,\overline{\pi} = e_1 \Rightarrow \Gamma'$

Single vs. multiple equations. Similarity 5 says that there should be no observable change between the case for a single equation and multiple (redundant) equations with the same right-hand side. That gets formulated into the following property.

Property 10 (Single/multiple Equations is Type Preserving).
If $\Gamma \vdash x\,\overline{\pi} = e \Rightarrow \Gamma, x : \sigma$ *then* $\Gamma \vdash x\,\overline{\pi} = e, x\,\overline{\pi} = e \Rightarrow \Gamma'$

This property favours the otherwise-unloved eager flavour. Imagine $f\ \_ = pair$. Under eager instantiation, this definition is accepted as type synthesis produces an instantiated type. Yet if we simply duplicate this equation under lazy instantiation (realistic scenarios would vary the patterns on the left-hand side, but duplication is simpler to state and addresses the property we want), then rule EDecl-NoAnnMulti will reject as it requires the type to be instantiated.

η-expansion. Similarity 6 leads to the following property.

Property 11 (η-expansion is Type Preserving).

- *If* $\Gamma \vdash e \Rightarrow \eta^\epsilon$ *where* $numargs(\eta^\epsilon) = n$ *then* $\Gamma \vdash \lambda\overline{x}^n.e\,\overline{x}^n \Rightarrow \eta^\epsilon$
- *If* $\Gamma \vdash e \Leftarrow \sigma$ *where* $numargs(\rho) = n$ *then* $\Gamma \vdash \lambda\overline{x}^n.e\,\overline{x}^n \Leftarrow \sigma$

Here, $\overline{x}^n$ represents n variables. We use $numargs(\sigma)$ to count the number of explicit arguments an expression can take, possibly instantiating any intervening implicit arguments. A formal definition can be found in Figure 9 in the appendix. However, in synthesis mode this property fails for every flavour: η^ϵ might be a function type $\sigma_1 \to \sigma_2$ taking a type scheme σ_1 as an argument, while we only synthesise monotype arguments. We thus introduce a restricted version of Property 11, with the additional premise that η^ϵ can not contain any binders to the left of an arrow.

Property 11b (η-expansion is Type Preserving (Monotype Restriction)).

- *If* $\Gamma \vdash e \Rightarrow \eta^\epsilon$ *where* $numargs(\eta^\epsilon) = n$ *and* $\Gamma \vdash \eta^\epsilon \xrightarrow{inst\ \delta} \tau$ *then* $\Gamma \vdash \lambda\overline{x}^n.e\,\overline{x}^n \Rightarrow \eta^\epsilon$
- *If* $\Gamma \vdash e \Leftarrow \sigma$ *where* $numargs(\rho) = n$ *then* $\Gamma \vdash \lambda\overline{x}^n.e\,\overline{x}^n \Leftarrow \sigma$

This (restricted) property fails for all but the eager/deep flavour as η-expansion forces other flavours to instantiate arguments they otherwise would not have.

8.3 Explicit Type Instantiation

The stability properties discussed above predominantly favour lazy instantiation, which is at least partly due to the presence of explicit type instantiation. Limiting the language to implicit type instantiation only implies that Property 2 holds under eager instantiation as well. The reason is that while instantiation prevents any explicit type application afterwards, re-generalised types can still be implicitly instantiated. Furthermore, as the distinction between inferred and specified types becomes irrelevant, Properties 4b, 7 and 9 now also hold under eager instantiation.

Note that the opposite holds true as well: discussing the static semantics stability for a language with only explicit type instantiation is uninteresting as all static properties discussed above trivially hold.

8.4 Conclusion

A brief inspection of Table 1 suggests how we should proceed: choose *lazy*, *shallow* instantiation. While this configuration does not respect all properties, it is the clear winner—even more so when we consider that Property 11b (one of only two that favour another mode) must be artificially restricted in order for any of our flavours to support the property.

We should note here that we authors were surprised by this result. This paper arose from the practical challenge of designing instantiation in GHC. After considerable debate among the authors of GHC, we were unable to remain comfortable with any one decision—as we see here, no choice is perfect, and so any suggestion was met with counter-examples showing how that suggestion was incorrect. Yet we had a hunch that eager instantiation was the right design. We thus formulated the similarities of Section 2 and went about building a formalisation and proving properties. Crucially, we did *not* select the similarities to favour a particular result, though we did choose to avoid reasonable similarities that would not show any difference between instantiation flavours. At an early stage of this work, we continued to believe that eager instantiation was superior. It was only through careful analysis, guided by our proofs and counter-examples, that we realised that lazy instantiation was winning. We are now convinced by our own analysis.

9 INSTANTIATION IN GHC

Given the connection between this work and GHC, we now turn to examine some practicalities of how lazy instantiation might impact the implementation.

9.1 Eagerness

GHC has used eager instantiation from the beginning, echoing Damas and Milner [1982]. Starting with GHC 8.0, however, which contains support for explicit type application, GHC has implemented an uneasy truce, sometimes using lazy instantiation (as advocated by Eisenberg et al. [2016]), and sometimes eager. In particular, all definitions are instantiated before their type in inferred. The truce is there because there is trouble supporting universal lazy instantiation, and any implementation of pervasive lazy instantiation would have to surmount these obstacles.

Displaying inferred types. The types inferred for functions are more exotic with lazy instantiation. For example, defining $f = \lambda\_ \to id$ would infer $f :: \forall\ \{a\}.\ a \to \forall\ b.\ b \to b$. These types, which could be reported by tools (including GHCi), might be confusing for users.

Monomorphism restriction. Eager instantiation makes the monomorphism restriction easier to implement, because relevant constraints are instantiated.

The monomorphism restriction is a peculiarity of Haskell, introduced to avoid unexpected runtime evaluation[9]. It potentially applies whenever a variable is defined without a type annotation and without any arguments to the left of the =. As an example, the monomorphism restriction would apply to **let** *plus* = (+) **in** ... and **let** $x = 5$ **in** ... (recalling that $5 :: Num\ a \Rightarrow a$ in Haskell). In these variable declarations, Haskell requires that the inferred type not be constrained. The reason is that such a declaration looks like it is declaring a variable, not a function. Accordingly, programmers might expect the variable to be evaluated once, and then accessible with no runtime cost many times. However, a constrained variable compiles to a function, taking the type-class

[9]The full description is in the Haskell Report, Section 4.5.5 [Marlow (editor) 2010].

dictionary [Hall et al. 1996] containing the runtime implementations of class methods. Thus, if we inferred $x :: Num\ a \Rightarrow a$, then every occurrence of x would have to be evaluated separately, causing runtime slowdown. The monomorphism restriction avoids this surprise by requiring an unconstrained type; in practice, the type will be defaulted. The examples in this paragraph are inferred to have types $plus :: Integer \to Integer \to Integer$ and $x :: Integer$, unless usage sites of the functions suggest a different numeric type.

Eager instantiation is helpful in implementing the monomorphism restriction, as the implementation of **let**-generalisation can look for unsolved constraints and default the type if necessary. With lazy instantiation, on the other hand, we would have to infer the type and then make a check to see whether it is constrained, instantiating it if necessary. Of course, the monomorphism restriction itself introduces instability in the language (note that *plus* and (+) have different types), and so perhaps revisiting this design choice is worthwhile.

Type application with un-annotated variables. For simplicity, we want all variables without type signatures not to work with explicit type instantiation. (Eisenberg et al. [2016, Section 3.1] expands on this point.) Eager instantiation accomplishes this, because variables without type signatures would get their polymorphism via re-generalisation. On the other hand, lazy instantiation would mean that some user-written variables might remain in a variable's type, like in the type of f, just above.

Yet even with eager instantiation, if instantiation is shallow, we can *still* get the possibility of visible type application on un-annotated variables: the specified variables might simply be hiding under a visible argument.

9.2 Depth

From the introduction of support for higher-rank types in GHC 6.8, GHC has done deep instantiation, as outlined by Peyton Jones et al. [2007], the paper describing the higher-rank types feature. However, deep instantiation has never respected the dynamic semantics of a program; Peyton Jones [2019] has the details. In addition, deep instantiation is required in order to support covariance of result types in the type subsumption judgement ([Peyton Jones et al. 2007, Figure 7]). This subsumption judgement, though, weakens the ability to do impredicative type inference, as described by Serrano et al. [2018] and Serrano et al. [2020]. GHC has thus, for GHC 9.0, changed to use shallow subsumption and shallow instantiation.

9.3 The situation today: Quick Look impredicativity has arrived

The analysis around stability in this paper strongly suggests that GHC should use the lazy, shallow approach to instantiation. Yet the struggles with lazy instantiation above remain. In order to simplify the implementation, GHC has recently (for GHC 9.0) switched to use exclusively eager instantiation. This choice sacrifices stability for convenience in implementation.

A second recent innovation within GHC (due for release in the next version, GHC 9.2) is the implementation of the Quick Look algorithm for impredicative type inference [Serrano et al. 2020]. The design of that algorithm walks a delicate balance between expressiveness and stability. It introduces new instabilities: for example, if $f\ x\ y$ requires impredicative instantiation, (**let** *unused* = 5 **in** f) $x\ y$ will fail. Given that users who opt into impredicative type inference are choosing to lose stability properties, we deemed it more important to study type inference without impredicativity in analysing stability. While our formulation of the inference algorithm is easily integrated with the Quick Look algorithm, we leave an analysis of the stability of the combination as future work.

10 RELATED WORK

The type systems in this paper build most directly from Peyton Jones et al. [2007], Eisenberg et al. [2016], and Serrano et al. [2020]. Each of these papers adds new capabilities to Haskell, and each also decreases the stability of the language. While these papers do consider properties we would consider to be components of stability, stability is not a key criterion in those authors' evaluation. By contrast, our work focuses squarely on stability as a believable proxy for the quality of the user experience.

Many other works on designing type inference algorithms also introduce stability properties, but these properties exist among others—such as completeness—and do not seem to guide the design of the algorithm. We do call out one such work, that by Schrijvers et al. [2019], which revolves around implicit programming systems, and describes a property they call *stability*. In the context of that work, stability is about Haskell's class-instance selection mechanism: we would like the choice of instance to remain stable under substitutions. That is, if $f :: C\ a \Rightarrow a \rightarrow Int$ is called at an argument of type $Maybe\ b$ (for a type variable b), the instance selected for $C\ (Maybe\ b)$ should be the same as the one that would be selected if f were called on an argument of type $Maybe\ Int$. After all $Maybe\ b$ can be substituted to become $Maybe\ Int$; perhaps a small change to the program would indeed cause this substitution, and we would not want a change in runtime behaviour. Accordingly, the stability property, as used by Schrijvers et al. [2019], is what we would also call a stability property, but it is much narrower than the definition we give the term.

In comparison to these other papers on type systems and type inference, the angle of this paper is somewhat different: we are not introducing a new language or type system feature, proving a language type safe, or proving an inference algorithm sound and complete to its declarative specification. Instead, we introduce the concept of *stability* as a new metric for evaluating (new or existing) type systems, and then apply this metric to a system featuring both implicit and explicit instantiation. Because of this novel, and somewhat unconventional topic, we are unable to find further related work.

11 CONCLUSION

This work introduces the concept of *stability* as a proxy for the usability of a language that supports both implicit and explicit arguments. We believe that designers of all languages supporting this mix of features need to grapple with this decision; those other designers may wish to follow our lead in formalising the problem to seek the most stable design. While stability is uninteresting in languages featuring pure explicit or pure implicit instantiation, it turns out to be an important metric in the presence of mixed behaviour.

We introduced a family of type systems, parameterised over the instantiation flavour, and featuring a mix of explicit and implicit behaviour. Using this family, we then evaluated the different flavours of instantiation, against a set of formal stability properties. The results are surprisingly unambiguous: (a) *lazy instantiation* achieves the highest stability for the static semantics, and (b) *shallow instantiation* results in the most stable dynamic semantics.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyon Jones. 2005. Associated Type Synonyms. In *International Conference on Functional Programming* (Tallinn, Estonia) *(ICFP '05)*. ACM.

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

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

Richard A. Eisenberg. 2018. Binding type variables in lambda-expressions. GHC Proposal #155. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0155-type-lambda.rst

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM.

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan Ahmed. 2016. Visible Type Application. In *European Symposium on Programming (ESOP) (LNCS)*. Springer-Verlag.

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

Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). Cambridge University Press.

Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *European Symposium on Programming*.

Didier Le Botlan and Didier Rémy. 2009. Recasting MLF. *Information and Computation* 207, 6 (2009), 726–785. https://doi.org/10.1016/j.ic.2008.12.006

Simon Marlow (editor). 2010. Haskell 2010 Language Report.

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

Simon Peyton Jones. 2019. Simplify subsumption. GHC Proposal #287. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0287-simplify-subsumption.rst

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

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000).

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

Tom Schrijvers, Bruno C. d. S. Oliveira, Philip Wadler, and Koar Marntirosian. 2019. COCHIS: Stable and coherent implicits. *J. Funct. Program.* 29 (2019), e3. https://doi.org/10.1017/S0956796818000242

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

Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded Impredicative Polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 783–796. https://doi.org/10.1145/3192366.3192389

Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. 2010. Let Should Not Be Generalized. In *Types in Language Design and Implementation (TLDI '10)*. ACM.

Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *POPL*. ACM, 60–76.

Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind Equality. In *International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. ACM.

A FURTHER INSTABILITIES AROUND INSTANTIATION

A.1 Explicit Abstraction

Binding implicit variables in named function definitions. If we sometimes want to explicitly instantiate an implicit argument, we will also sometimes want to explicitly abstract over an implicit argument. A classic example of why this is useful is in the *replicate* function for length-indexed vectors, here written in Idris:

```
replicate : {n : Nat} → a → Vect n a
replicate {n = Z}   _ = []
replicate {n = S _} x = x :: replicate x
```

Because a length-indexed vector *Vect* includes its length in its type, we need not always pass the desired length of a vector into the *replicate* function: type inference can figure it out. We thus decide here to make the $n : Nat$ parameter to be implicit, putting it in braces. However, in the definition of *replicate*, we must pattern-match on the length to decide what to return. The solution is to use an explicit pattern, in braces, to match against the argument n.

Idris and Agda both support explicit abstraction in parallel to their support of explicit instantiation: when writing equations for a function, the user can use braces to denote the abstraction over an implicit parameter. Idris requires such parameters to be named, while Agda supports both named and ordered parameters, just as the languages do for instantiation. The challenges around stability are the same here as they are for explicit instantiation.

Haskell has no implemented feature analogous to this. Its closest support is that for scoped type variables, where a type variable introduced in a type signature becomes available in a function body. For example:

```
const :: ∀ a b. a → b → a
const x y = (x :: a)
```

The $\forall\ a\ b$ brings a and b into scope both in the type signature *and* in the function body. This feature in Haskell means that, like in Idris and Agda, changing the name of an apparently local variable in a type signature may affect code beyond that type signature. It also means that the top-level $\forall$ in a type signature is treated specially. For example, neither of the following examples are accepted by GHC:

```
const₁ :: ∀. ∀ a b. a → b → a
const₁ x y = (x :: a)
const₂ :: (∀ a b. a → b → a)
const₂ x y = (x :: a)
```

In $const_1$, the vacuous $\forall.$ (which is, generally, allowed) stops the scoped-type variables mechanism from bringing a into scope; in $const_2$, the parentheses around the type serve the same function. Once again, we see how Haskell is unstable: programmers might reasonably think that syntax like $\forall\ a\ b.$ is shorthand for $\forall\ a. \forall\ b.$ or that outermost parentheses would be redundant, yet neither of these facts is true.

Binding implicit variables in an anonymous function. Sometimes, binding a type variable only in a function declaration is not expressive enough, however—we might want to do this in an anonymous function in the middle of some other expression.

Here is a (contrived) example of this in Agda, where ∋ allows for prefix type annotations:

```
_∋_ : (A : Set) → A → A
A ∋ x = x

ChurchBool : Set₁
ChurchBool = {A : Set} → A → A → A

churchBoolToBit : ChurchBool → ℕ
churchBoolToBit b = b 1 0

one : ℕ
one = churchBoolToBit (λ{A} x₁ x₂ → A ∋ x₁)
```

Here, we bind the implicit variable A in the argument to *churchBoolToBit*. (Less contrived examples are possible; see the Motivation section of Eisenberg [2018].)

Binding an implicit variable in a λ-expression is subtler than doing it in a function clause. Idris does not support this feature at all, requiring a named function to bind an implicit variable. Agda supports this feature, as written above, but with caveats: the construct only works sometimes. For example, the following is rejected:

```
id : {A : Set} → A → A
id = λ{A} x → A ∋ x
```

The fact that this example is rejected, but `id {A} x = A ∋ x` is accepted is another example of apparent instability—we might naïvely expect that writing a function with an explicit λ and using patterns to the left of an = are equivalent. Another interesting aspect of binding an implicit variable in a λ-abstraction is that the name of the variable is utterly arbitrary: instead of writing $(\lambda\{A\}\ x_1\ x_2 \to A \ni x_1)$, we can write $(\lambda\{anything = A\}\ x_1\ x_2 \to A \ni x_1)$. This is an attempt to use Agda's support for named implicits, but the name can be, well, *anything*. This would appear to be a concession to the fact that the proper name for this variable, A as written in the definition of *ChurchBool*, can be arbitrarily far away from the usage of the name, so Agda is liberal in accepting any replacement for it.

An accepted proposal [Eisenberg 2018] adds this feature to Haskell, though it has not been implemented as of this writing. That proposal describes that the feature would be available only when we are *checking* a term against a known type, taking advantage of GHC's bidirectional type system [Eisenberg et al. 2016; Peyton Jones et al. 2007]. One of the motivations that inspired this paper was to figure out whether we could relax this restriction. After all, it would seem plausible that we should accept a definition like $id = \lambda\ @a\ (x :: a) \to a$ without a type signature. (Here, the $@a$ syntax binds a to an otherwise-implicit type argument.) It will turn out that, in the end, we can do this only when we instantiate lazily—see Section 8.

A.2 Implicit Generalisation

All three languages support some form of implicit generalisation, despite the fact that the designers of Haskell famously declared that **let** should not be generalised [Vytiniotis et al. 2010] and that both Idris and Agda require type signatures on all declarations.

Haskell. Haskell's **let**-generalisation is the most active, as type signatures are optional.[10] Suppose we have defined $const\ x\ y = x$, without a signature. What type do we infer? It could be $\forall\ a\ b.\ a \to$

[10] Though not relevant for our analysis, some readers may want the details: Without any language extensions enabled, all declarations without signatures are generalised, meaning that defining $id\ x = x$ will give *id* the type $\forall\ a.\ a \to a$. With the `MonoLocalBinds` extension enabled, which is activated by either of `GADTs` or `TypeFamilies`, local definitions that capture variables from an outer scope are not generalised—this is the effect of the dictum that **let** should not be generalised. As an example, the g in $f\ x =$ **let** $g\ y = (y, x)$ **in** $(g\ \texttt{'a'}, g\ True)$ is not generalised, because its body mentions the captured

$b \to a$ or $\forall\ b\ a.\ a \to b \to a$. This choice matters, because it affects the meaning of explicit type instantiations. A natural reaction is to suggest choosing the former inferred type, following the left-to-right scheme described above. However, in a language with a type system as rich as Haskell's, this guideline does not always work. Haskell supports type synonyms (which can reorder the occurrence of variables), class constraints (whose ordering is arbitrary) [Wadler and Blott 1989], functional dependencies (which mean that a type variable might be mentioned *only* in constraints and not in the main body of a type) [Jones 2000], and arbitrary type-level computation through type families [Chakravarty et al. 2005; Eisenberg et al. 2014]. With all of these features potentially in play, it is unclear how to order the type variables. Thus, in a concession to language stability, Haskell brutally forbids explicit type instantiation on any function whose type is inferred; we discuss the precise mechanism in the next section.

Since GHC 8.0, Haskell allows dependency within type signatures [Weirich et al. 2013], meaning that the straightforward left-to-right ordering of variables—even in a user-written type signature—might not be well-scoped. As a simple example, consider $tr :: \mathit{TypeRep}\ (a :: k)$, where $\mathit{TypeRep} :: \forall\ k.\ k \to \mathit{Type}$ allows runtime type representation and is part of GHC's standard library. A naive left-to-right extraction of type variables would yield $\forall\ a\ k.\ \mathit{TypeRep}\ (a :: k)$, which is ill-scoped when we consider that a depends on k. Instead, we must reorder to $\forall\ k\ a.\ \mathit{TypeRep}\ (a :: k)$. In order to support stability when instantiating explicitly, GHC thus defines a concrete sorting algorithm, called "ScopedSort", that reorders the variables; it has become part of GHC's user-facing specification. Any change to this algorithm may break user programs, and it is specified in GHC's user manual.

Idris. Idris's support for implicit generalisation is harder to trigger; see Appendix B for an example of how to do it. The problem that arises in Idris is predictable: if the compiler performs the quantification, then it must choose the name of the quantified type variable. How will clients know what this name is, necessary in order to instantiate the parameter? They cannot. Accordingly, in order to support stability, Idris uses a special name for generalised variables: the variable name itself includes braces (for example, it might be $\{k : 265\}$) and thus can never be parsed[11].

Agda. Recent versions of Agda support a new **variable** keyword[12]. Here is an example of it in action:

```
variable
  A : Set
  l1 l2 : List A
```

The declaration says that an out-of-scope use of, say, A is a hint to Agda to implicitly quantify over $A : \mathit{Set}$. The order of declarations in a **variable** block is significant: note that l_1 and l_2 depend on A. However, because explicit instantiation by order is possible in Agda, we must specify the order of quantification when Agda does generalisation. Often, this order is derived directly from the **variable** block—but not always. Consider this (contrived) declaration:

```
property : length l2 + length l1 ≡ length l1 + length l2
```

What is the full, elaborated type of *property*? Note that the two lists l_1 and l_2 can have *different* element types A. The Agda manual calls this *nested* implicit generalisation, and it specifies an

---

x. Accordingly, f is rejected, as it uses g at two different types (*Char* and *Bool*). Adding a type signature to g can fix the problem.

[11] Idris 1 does not use an exotic name, but still prevents explicit instantiation, using a mechanism similar to Haskell's provenance mechanism described below.

[12] See https://agda.readthedocs.io/en/v2.6.0.1/language/generalization-of-declared-variables.html in the Agda manual for an description of the feature.

algorithm—similar to GHC's ScopedSort—to specify the ordering of variables. Indeed it must offer this specification, as leaving this part out would lead to instability; that is, it would lead to the inability for a client of *property* to know how to order their type instantiations.

B EXAMPLE OF IMPLICIT GENERALISATION IN IDRIS

It is easy to believe that a language that requires type signatures on all definitions will not have implicit generalisation. However, Idris does allow generalisation to creep in, with just the right definitions.

We start with this:

```
data Proxy : {k : Type} → k → Type where
  P : Proxy a
```

The datatype *Proxy* here is polymorphic; its one explicit argument can be of any type.

Now, we define *poly*:

```
poly : Proxy a
poly = P
```

We have not given an explicit type to the type variable *a* in *poly*'s type. Because *Proxy*'s argument can be of any type, *a*'s type is unconstrained. Idris *generalises* this type, giving *poly* the type $\{k : Type\} \to \{a : k\} \to Proxy\ a$.

At a use site of *poly*, we must then distinguish between the possibility of instantiating the user-written *a* and the possibility of instantiating the compiler-generated *k*. This is done by giving the *k* variable an unusual name, `{k:446}` in our running Idris session.

C ADDITIONAL RELATIONS

$\boxed{binders^{\delta}(\sigma) = \overline{a}; \rho}$ *(Binders)*

$$\textsc{EBndr-ShallowInst}\ \frac{}{binders^{\mathcal{S}}(\rho) = \cdot; \rho} \qquad \textsc{EBndr-ShallowForall}\ \frac{binders^{\mathcal{S}}(\sigma) = \overline{b}; \rho}{binders^{\mathcal{S}}(\forall\, \overline{a}.\sigma) = \overline{a}, \overline{b}; \rho} \qquad \textsc{EBndr-ShallowInfForall}\ \frac{binders^{\mathcal{S}}(\sigma) = \overline{b}; \rho}{binders^{\mathcal{S}}(\forall\, \overline{\{a\}}.\sigma) = \overline{\{a\}}, \overline{b}; \rho}$$

$$\textsc{EBndr-DeepMono}\ \frac{}{binders^{\mathcal{D}}(\tau) = \cdot; \tau} \qquad \textsc{EBndr-DeepFunction}\ \frac{binders^{\mathcal{D}}(\sigma_2) = \overline{a}; \rho_2}{binders^{\mathcal{D}}(\sigma_1 \to \sigma_2) = \overline{a}; \sigma_1 \to \rho_2} \qquad \textsc{EBndr-DeepForall}\ \frac{binders^{\mathcal{D}}(\sigma) = \overline{b}; \rho}{binders^{\mathcal{D}}(\forall\, \overline{a}.\sigma) = \overline{a}, \overline{b}; \rho}$$

$$\textsc{EBndr-DeepInfForall}\ \frac{binders^{\mathcal{D}}(\sigma) = \overline{b}; \rho}{binders^{\mathcal{D}}(\forall\, \overline{\{a\}}.\sigma) = \overline{\{a\}}, \overline{b}; \rho}$$

$\boxed{wrap\ (\overline{\pi}; e_1 \sim e_2)}$ *(Pattern Wrapping)*

$$\textsc{PatWrap-Empty}\ \frac{}{wrap\ (\cdot; e \sim e)} \qquad \textsc{PatWrap-Var}\ \frac{wrap\ (\overline{\pi}; e_1 \sim e_2)}{wrap\ (x, \overline{\pi}; e_1 \sim \lambda x.e_2)} \qquad \textsc{PatWrap-TyVar}\ \frac{wrap\ (\overline{\pi}; e_1 \sim e_2)}{wrap\ (@a, \overline{\pi}; e_1 \sim \Lambda a.e_2)}$$

D CORE LANGUAGE

The dynamic semantics of the languages in Sections 6 and 7 are defined through a translation to System F. While the target language is largely standard, a few interesting remarks can be made. The language supports nested pattern matching through case lambdas $\overline{\textbf{case}\ \overline{\pi_F}_i : \overline{\psi_F} \to t_i}^{\,i}$, where patterns π_F include both term and type variables, as well as nested constructor patterns. Note that while we reuse our type σ grammar for the core language, System F does not distinguish between inferred and specified binders.

We also define two meta-language features to simplify the elaboration, and the proofs: Firstly, in order to support eta-expansion (for translating deep instantiation to System F), we define expression wrappers $\dot{t}$, essentially a limited form of expressions with a hole $\bullet$ in them. An expression t can be filled in for the hole to get a new expression $\dot{t}[t]$. One especially noteworthy wrapper construct is $\lambda t_1.t_2$, explicitly abstracting over and handling the expression to be filled in. Note that, as expression wrappers are only designed to alter the type of expressions through eta-expansion, there is no need to support the full System F syntax.

Secondly, in order to define contextual equivalence, we introduce contexts M. These are again expressions with a hole $\bullet$ in them, but unlike expression wrappers, contexts do cover the entire System F syntax. Typing contexts is performed by the $M : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \sigma_2$ relation: "Given an expression t that has type σ_1 under typing environment Γ_1, then the resulting expression $M[t]$ has type σ_2 under typing environment Γ_2". We will elaborate further on contextual equivalence in Appendix E.2.

$$
\begin{array}{lcll}
t & ::= & x \mid K \mid t_1\ t_2 \mid \lambda x : \sigma.t \mid t\ \sigma \mid \Lambda a.t \mid \mathit{undefined} \mid \mathit{seq} & \textit{Expression} \\
 & \mid & \textbf{case}\ \overline{\pi_F}_i : \overline{\psi_F} \to t_i^{\,i} \mid \mathit{true} \mid \mathit{false} & \\
v & ::= & \lambda x : \sigma.t \mid \Lambda a.t \mid \textbf{case}\ \overline{\pi_F}_i : \overline{\psi_F} \to t_i^{\,i} \mid K\ \overline{t} & \textit{Value} \\
\dot{t} & ::= & \bullet \mid \lambda x : \sigma.\dot{t} \mid \dot{t}\ \sigma \mid \Lambda a.\dot{t} \mid \lambda t_1.t_2 & \textit{Expression Wrapper} \\
M & ::= & \bullet \mid \lambda x : \sigma.M \mid M\ t \mid t\ M \mid \Lambda a.M \mid M\ \sigma & \textit{Context} \\
arg_F & ::= & t \mid \sigma & \textit{Argument} \\
\pi_F & ::= & x : \sigma \mid @a \mid K\ \overline{\pi_F} & \textit{Pattern} \\
\psi_F & ::= & \sigma \mid @a & \textit{Arg. descriptor}
\end{array}
$$

$\boxed{\Gamma \vdash t : \sigma}$ *(System F Term Typing)*

$$\textsc{FTm-Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \textsc{FTm-Con}\ \frac{K : \overline{a}; \overline{\sigma}; T \in \Gamma}{\Gamma \vdash K : \forall\, \overline{a}.\overline{\sigma} \to T\ \overline{a}} \qquad \textsc{FTm-App}\ \frac{\Gamma \vdash t_1 : \sigma_1 \to \sigma_2 \quad \Gamma \vdash t_2 : \sigma_1}{\Gamma \vdash t_1\ t_2 : \sigma_2} \qquad \textsc{FTm-Abs}\ \frac{\Gamma, x : \sigma_1 \vdash t : \sigma_2}{\Gamma \vdash \lambda x : \sigma_1.t : \sigma_1 \to \sigma_2}$$

$$\textsc{FTm-TyApp}\ \frac{\Gamma \vdash t : \forall\, a.\sigma_1}{\Gamma \vdash t\ \sigma_2 : [\sigma_2/a]\ \sigma_1} \qquad \textsc{FTm-TyAbs}\ \frac{\Gamma, a \vdash t : \sigma}{\Gamma \vdash \Lambda a.t : \forall\, a.\sigma} \qquad \textsc{FTm-Undef}\ \frac{}{\Gamma \vdash \mathit{undefined} : \forall\, a.a} \qquad \textsc{FTm-True}\ \frac{}{\Gamma \vdash \mathit{true} : \mathit{Bool}}$$

$$\textsc{FTm-False}\ \frac{}{\Gamma \vdash \mathit{false} : \mathit{Bool}} \qquad \textsc{FTm-Seq}\ \frac{}{\Gamma \vdash \mathit{seq} : \forall\, a.\forall\, b.a \to b \to b} \qquad \textsc{FTm-Case}\ \frac{\overline{\Gamma \vdash^P \overline{\pi_F}_i : \overline{\psi_F}; \Delta}^{\,i} \quad \overline{\Gamma, \Delta \vdash t_i : \sigma_1}^{\,i} \quad \mathit{type}\,(\overline{\psi_F}; \sigma_1 \sim \sigma_2)}{\Gamma \vdash \textbf{case}\ \overline{\pi_F}_i : \overline{\psi_F} \to t_i^{\,i} : \sigma_2}$$

$\boxed{\Gamma \vdash^P \overline{\pi_F} : \overline{\psi_F}; \Delta}$ *(System F Pattern Typing)*

$$\textsc{FPat-Empty}\ \frac{}{\Gamma \vdash^P \cdot : \cdot; \cdot} \qquad \textsc{FPat-Var}\ \frac{\Gamma, x : \sigma \vdash^P \overline{\pi_F} : \overline{\psi_F}; \Delta}{\Gamma \vdash^P x : \sigma, \overline{\pi_F} : \sigma, \overline{\psi_F}; x : \sigma, \Delta} \qquad \textsc{FPat-TyVar}\ \frac{\Gamma, a \vdash^P \overline{\pi_F} : \overline{\psi_F}; \Delta}{\Gamma \vdash^P @a, \overline{\pi_F} : @a, \overline{\psi_F}; a, \Delta}$$

$$\textsc{FPat-Con}\ \frac{K : \overline{a}_0; \overline{\sigma}_0; T \in \Gamma \quad \Gamma \vdash^P \overline{\pi_F} : [\overline{\sigma}_1/\overline{a}_0]\,\overline{\sigma}_0; \Delta_1 \quad \Gamma, \Delta_1 \vdash^P \overline{\pi_F}' : \overline{\psi_F}; \Delta_2}{\Gamma \vdash^P (K\,\overline{\pi_F}), \overline{\pi_F}' : T\,\overline{\sigma}_1, \overline{\psi_F}; \Delta_1, \Delta_2}$$

$\boxed{M : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \sigma_2}$ *(System F Context Typing)*

$$\textsc{FCtx-Hole}\ \frac{}{\bullet : \Gamma; \sigma \mapsto \Gamma; \sigma} \qquad \textsc{FCtx-Abs}\ \frac{M : \Gamma_1; \sigma_2 \mapsto \Gamma_2, x : \sigma_1; \sigma_3}{\lambda x : \sigma_1.M : \Gamma_1; \sigma_2 \mapsto \Gamma_2; \sigma_1 \to \sigma_3} \qquad \textsc{FCtx-AppR}\ \frac{M_1 : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \sigma_2 \to \sigma_3 \quad \Gamma_2 \vdash t_2 : \sigma_2}{M_1\ t_2 : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \sigma_3}$$

$$\textsc{FCtx-AppL}\ \frac{\Gamma_2 \vdash t_1 : \sigma_2 \to \sigma_3 \quad M_2 : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \sigma_2}{t_1\ M_2 : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \sigma_3} \qquad \textsc{FCtx-TyAbs}\ \frac{M : \Gamma_1; \sigma_1 \mapsto \Gamma_2, a; \sigma_2}{\Lambda a.M : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \forall\, a.\sigma_2} \qquad \textsc{FCtx-TyApp}\ \frac{M : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \forall\, a.\sigma_2}{M\,\sigma : \Gamma_1; \sigma_1 \mapsto \Gamma_2; [\sigma/a]\,\sigma_2}$$

$$\textsc{FCtx-Case}\ \frac{\dfrac{\dfrac{}{\Gamma_1 \vdash^P \overline{\pi_F}_i : \overline{\psi_F}; \Delta}\,i}{M_i : \Gamma_1, \Delta; \sigma_1 \mapsto \Gamma_2; \sigma_2}\,i \quad type\,(\overline{\psi_F}; \sigma_2 \sim \sigma_3)}{\mathbf{case}\ \overline{\overline{\pi_F}_i : \overline{\psi_F} \to M_i}^{\,i} : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \sigma_3}$$

Evaluation for our System F target language is defined below.

$\boxed{match\,(\overline{\pi_F}_1 \to t_1; t_2 : \sigma_2) \hookrightarrow \overline{\pi_F}_2 \to t_1'}$ *(Core Pattern Matching)*

$$\textsc{FMatch-Var}\ \frac{}{match\,(x : \sigma, \overline{\pi_F} \to t_1; t_2 : \sigma) \hookrightarrow \overline{\pi_F} \to [t_2/x]t_1}$$

$$\textsc{FMatch-Con}\ \frac{\sigma_2 = \overline{\psi_F}_1 \to \tau_2 \quad t_2 \hookrightarrow^{\Downarrow} K\,\overline{t} \quad (\mathbf{case}\ \overline{\pi_F}_1 : \overline{\psi_F}_1 \to t_1)\,\overline{t} \hookrightarrow^{\Downarrow} v}{match\,((K\,\overline{\pi_F}_1), \overline{\pi_F}_2 \to t_1; t_2 : \sigma_2) \hookrightarrow \overline{\pi_F}_2 \to v}$$

$\boxed{t_1 \hookrightarrow t_2}$ *(Core Evaluation)*

$$\textsc{FEval-App}\ \frac{t_1 \hookrightarrow t_1'}{t_1\ t_2 \hookrightarrow t_1'\ t_2} \qquad \textsc{FEval-AppAbs}\ \frac{}{(\lambda x : \sigma.t_1)\ t_2 \hookrightarrow [t_2/x]t_1} \qquad \textsc{FEval-Seq}\ \frac{t_1 \hookrightarrow t_1'}{seq\ t_1\ t_2 \hookrightarrow seq\ t_1'\ t_2} \qquad \textsc{FEval-SeqVal}\ \frac{}{seq\ v_1\ t_2 \hookrightarrow t_2}$$

$$\textsc{FEval-CaseEmpty}\ \frac{}{\mathbf{case}\ \overline{\cdot : \cdot \to t_i}^{\,i} \hookrightarrow t_0} \qquad \textsc{FEval-CaseMatch}\ \frac{\forall j \in v\ \mathbf{where}\ (match\ (\overline{\pi_F}_j \to t_j; t_2 : \sigma) \hookrightarrow \overline{\pi_F}_j' \to t_j')}{(\mathbf{case}\ \overline{\overline{\pi_F}_i : \sigma, \overline{\psi_F} \to t_i}^{\,i<v}\ )\ t_2 \hookrightarrow \mathbf{case}\ \overline{\overline{\pi_F}_j' : \overline{\psi_F} \to t_j'}^{\,j<w}}$$

$$\textsc{FEval-TyApp}\ \frac{t_1 \hookrightarrow t_1'}{t_1\ \sigma \hookrightarrow t_1'\ \sigma} \qquad \textsc{FEval-TyAppAbs}\ \frac{}{(\Lambda a.t_1)\ \sigma \hookrightarrow [\sigma/a]\ t_1} \qquad \textsc{FEval-Undef}\ \frac{}{undefined \hookrightarrow undefined}$$

$$\textsc{FEval-TyAbsCase}\ \frac{}{(\mathbf{case}\ \overline{@a_i, \overline{\pi_F}_i : @a, \overline{\psi_F} \to t_i}^{\,i}\ )\ \sigma \hookrightarrow \mathbf{case}\ \overline{[\sigma/a]\overline{\pi_F}_i : [\sigma/a]\ \overline{\psi_F} \to [\sigma/a]\ t_i}^{\,i}}$$

$\boxed{t \hookrightarrow^{\Downarrow} v}$ *(Big Step Evaluation)*

$$\textsc{FEvalBigStep-Step}\ \frac{t \hookrightarrow t' \qquad t' \hookrightarrow^{\Downarrow} v}{t \hookrightarrow^{\Downarrow} v} \qquad \textsc{FEvalBigStep-Done}\ \frac{}{v \hookrightarrow^{\Downarrow} v}$$

D.1 Translation from the Mixed Polymorphic λ-calculus

$\boxed{\Gamma \vdash^H h \Rightarrow \sigma \rightsquigarrow t}$ *(Head Type Synthesis)*

$$\textsc{H-Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash^H x \Rightarrow \sigma \rightsquigarrow x} \qquad \textsc{H-Con}\ \frac{K : \overline{a}; \overline{\sigma}; T \in \Gamma}{\Gamma \vdash^H K \Rightarrow \forall\, \overline{a}.\overline{\sigma} \to T\,\overline{a} \rightsquigarrow K} \qquad \textsc{H-Ann}\ \frac{\Gamma \vdash e \Leftarrow \sigma \rightsquigarrow t}{\Gamma \vdash^H e : \sigma \Rightarrow \sigma \rightsquigarrow t}$$

$$\textsc{H-Undef}\ \frac{}{\Gamma \vdash^H undefined \Rightarrow \forall\, a.a \rightsquigarrow undefined} \qquad \textsc{H-Seq}\ \frac{}{\Gamma \vdash^H seq \Rightarrow \forall\, a.\forall\, b.a \to b \to b \rightsquigarrow seq}$$

$$\textsc{H-Inf}\ \frac{\Gamma \vdash e \Rightarrow \sigma \rightsquigarrow t}{\Gamma \vdash^H e \Rightarrow \sigma \rightsquigarrow t}$$

$\boxed{\Gamma \vdash e \Rightarrow \eta^{\epsilon} \rightsquigarrow t}$ *(Term Type Synthesis)*

ETm-InfAbs
$$\frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \eta_2^{\epsilon} \rightsquigarrow t_1}{\Gamma \vdash \lambda x.e \Rightarrow \tau_1 \to \eta_2^{\epsilon} \rightsquigarrow \lambda x : \tau_1.t_1}$$

ETm-InfTyAbs
$$\frac{\Gamma, a \vdash e \Rightarrow \eta_1^{\epsilon} \rightsquigarrow t \qquad \Gamma \vdash \forall\, a.\eta_1^{\epsilon} \xrightarrow{inst\ \delta} \eta_2^{\epsilon} \rightsquigarrow \dot{t}}{\Gamma \vdash \Lambda a.e \Rightarrow \eta_2^{\epsilon} \rightsquigarrow \dot{t}[\Lambda a.t]}$$

ETm-InfApp
$$\frac{\Gamma \vdash^{H} h \Rightarrow \sigma \rightsquigarrow t \qquad \Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma \Rightarrow \sigma' \rightsquigarrow \overline{arg_F} \qquad \Gamma \vdash \sigma' \xrightarrow{inst\ \delta} \eta^{\epsilon} \rightsquigarrow \dot{t}}{\Gamma \vdash h\,\overline{arg} \Rightarrow \eta^{\epsilon} \rightsquigarrow \dot{t}[t\,\overline{arg_F}]}$$

ETm-InfLet
$$\frac{\Gamma \vdash decl \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t_1 \qquad \Gamma' \vdash e \Rightarrow \eta^{\epsilon} \rightsquigarrow t_2}{\Gamma \vdash \mathbf{let}\ decl\ \mathbf{in}\ e \Rightarrow \eta^{\epsilon} \rightsquigarrow (\lambda x : \sigma.t_2)\ t_1}$$

ETm-InfTrue
$$\frac{}{\Gamma \vdash true \Rightarrow Bool \rightsquigarrow true}$$

ETm-InfFalse
$$\frac{}{\Gamma \vdash false \Rightarrow Bool \rightsquigarrow false}$$

$\boxed{\Gamma \vdash e \Leftarrow \sigma \rightsquigarrow t}$ *(Term Type Scheme Checking)*

ETm-CheckAbs
$$\frac{\Gamma \vdash \sigma \xrightarrow{skol\ \mathcal{S}} \sigma_1 \to \sigma_2; \Gamma_1 \rightsquigarrow \dot{t} \qquad \Gamma_1, x : \sigma_1 \vdash e \Leftarrow \sigma_2 \rightsquigarrow t_1}{\Gamma \vdash \lambda x.e \Leftarrow \sigma \rightsquigarrow \dot{t}[\lambda x : \sigma_1.t_1]}$$

ETm-CheckTyAbs
$$\frac{\sigma = \forall\, \overline{\{a\}}.\forall\, a.\sigma' \qquad \Gamma, \overline{a}, a \vdash e \Leftarrow \sigma' \rightsquigarrow t}{\Gamma \vdash \Lambda a.e \Leftarrow \sigma \rightsquigarrow \Lambda \overline{a}.\Lambda a.t}$$

ETm-CheckLet
$$\frac{\Gamma \vdash decl \Rightarrow \Gamma' \rightsquigarrow x : \sigma_1 = t_1 \qquad \Gamma' \vdash e \Leftarrow \sigma \rightsquigarrow t_2}{\Gamma \vdash \mathbf{let}\ decl\ \mathbf{in}\ e \Leftarrow \sigma \rightsquigarrow (\lambda x : \sigma_1.t_2)\ t_1}$$

ETm-CheckInf
$$\frac{\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho; \Gamma_1 \rightsquigarrow \dot{t}_1 \qquad \Gamma_1 \vdash e \Rightarrow \eta^{\epsilon} \rightsquigarrow t \qquad \Gamma_1 \vdash \eta^{\epsilon} \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_2 \qquad e \neq \lambda, \Lambda, let}{\Gamma \vdash e \Leftarrow \sigma \rightsquigarrow \dot{t}_1[\dot{t}_2[t]]}$$

$\boxed{\Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma \Rightarrow \sigma' \rightsquigarrow \overline{arg_F}}$ *(Argument Type Checking)*

Arg-Empty
$$\frac{}{\Gamma \vdash^{A} \cdot \Leftarrow \sigma \Rightarrow \sigma \rightsquigarrow \cdot}$$

Arg-App
$$\frac{\Gamma \vdash e \Leftarrow \sigma_1 \rightsquigarrow t \qquad \Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma_2 \Rightarrow \sigma' \rightsquigarrow \overline{arg_F}}{\Gamma \vdash^{A} e, \overline{arg} \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma' \rightsquigarrow t, \overline{arg_F}}$$

Arg-Inst
$$\frac{\Gamma \vdash^{A} e, \overline{arg} \Leftarrow [\tau_1/a]\,\sigma_2 \Rightarrow \sigma_3 \rightsquigarrow \overline{arg_F}}{\Gamma \vdash^{A} e, \overline{arg} \Leftarrow \forall\, a.\sigma_2 \Rightarrow \sigma_3 \rightsquigarrow \overline{arg_F}}$$

Arg-TyApp
$$\frac{\Gamma \vdash^{A} \overline{arg} \Leftarrow [\sigma_1/a]\,\sigma_2 \Rightarrow \sigma_3 \rightsquigarrow \overline{arg_F}}{\Gamma \vdash^{A} @\sigma_1, \overline{arg} \Leftarrow \forall\, a.\sigma_2 \Rightarrow \sigma_3 \rightsquigarrow \tau_1, \overline{arg_F}}$$

Arg-InfInst
$$\frac{\Gamma \vdash^{A} \overline{arg} \Leftarrow [\tau_1/a]\,\sigma_2 \Rightarrow \sigma_3 \rightsquigarrow \overline{arg_F}}{\Gamma \vdash^{A} \overline{arg} \Leftarrow \forall\, \{a\}.\sigma_2 \Rightarrow \sigma_3 \rightsquigarrow \overline{arg_F}}$$

$\boxed{\Gamma \vdash \sigma \xrightarrow{\text{inst } \delta} \rho \rightsquigarrow \dot{t}}$ *(Type Instantiation)*

InstT-SInst

$$\frac{}{\Gamma \vdash \rho \xrightarrow{\text{inst } \mathcal{S}} \rho \rightsquigarrow \bullet}$$

InstT-SForall

$$\frac{\Gamma \vdash [\tau/a]\,\sigma \xrightarrow{\text{inst } \mathcal{S}} \rho \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\, a.\sigma \xrightarrow{\text{inst } \mathcal{S}} \rho \rightsquigarrow \lambda t.(\dot{t}[t\,\tau])}$$

InstT-SInfForall

$$\frac{\Gamma \vdash [\tau/a]\,\sigma \xrightarrow{\text{inst } \mathcal{S}} \rho \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\, \{a\}.\sigma \xrightarrow{\text{inst } \mathcal{S}} \rho \rightsquigarrow \lambda t.(\dot{t}[t\,\tau])}$$

InstT-Mono

$$\frac{}{\Gamma \vdash \tau \xrightarrow{\text{inst } \mathcal{D}} \tau \rightsquigarrow \bullet}$$

InstT-Function

$$\frac{\Gamma \vdash \sigma_2 \xrightarrow{\text{inst } \mathcal{D}} \rho_2 \rightsquigarrow \dot{t}}{\Gamma \vdash \sigma_1 \to \sigma_2 \xrightarrow{\text{inst } \mathcal{D}} \sigma_1 \to \rho_2 \rightsquigarrow \lambda t.\lambda x : \sigma_1.(\dot{t}[t\,x])}$$

InstT-Forall

$$\frac{\Gamma \vdash [\tau/a]\,\sigma \xrightarrow{\text{inst } \mathcal{D}} \rho \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\, a.\sigma \xrightarrow{\text{inst } \mathcal{D}} \rho \rightsquigarrow \lambda t.(\dot{t}[t\,\tau])}$$

InstT-InfForall

$$\frac{\Gamma \vdash [\tau/a]\,\sigma \xrightarrow{\text{inst } \mathcal{D}} \rho \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\, \{a\}.\sigma \xrightarrow{\text{inst } \mathcal{D}} \rho \rightsquigarrow \lambda t.(\dot{t}[t\,\tau])}$$

$\boxed{\Gamma \vdash \sigma \xrightarrow{\text{skol } \delta} \rho; \Gamma' \rightsquigarrow \dot{t}}$ *(Type Skolemisation)*

SkolT-SInst

$$\frac{}{\Gamma \vdash \rho \xrightarrow{\text{skol } \mathcal{S}} \rho; \Gamma \rightsquigarrow \bullet}$$

SkolT-SForall

$$\frac{\Gamma, a \vdash \sigma \xrightarrow{\text{skol } \mathcal{S}} \rho; \Gamma_1 \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\, a.\sigma \xrightarrow{\text{skol } \mathcal{S}} \rho; \Gamma_1 \rightsquigarrow \Lambda a.\dot{t}}$$

SkolT-SInfForall

$$\frac{\Gamma, a \vdash \sigma \xrightarrow{\text{skol } \mathcal{S}} \rho; \Gamma_1 \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\, \{a\}.\sigma \xrightarrow{\text{skol } \mathcal{S}} \rho; \Gamma_1 \rightsquigarrow \Lambda a.\dot{t}}$$

SkolT-Mono

$$\frac{}{\Gamma \vdash \tau \xrightarrow{\text{skol } \mathcal{D}} \tau; \Gamma \rightsquigarrow \bullet}$$

SkolT-Function

$$\frac{\Gamma \vdash \sigma_2 \xrightarrow{\text{skol } \mathcal{D}} \rho_2; \Gamma_1 \rightsquigarrow \dot{t}}{\Gamma \vdash \sigma_1 \to \sigma_2 \xrightarrow{\text{skol } \mathcal{D}} \sigma_1 \to \rho_2; \Gamma_1 \rightsquigarrow \lambda t.\lambda x : \sigma_1.(\dot{t}[t\,x])}$$

SkolT-Forall

$$\frac{\Gamma, a \vdash \sigma \xrightarrow{\text{skol } \mathcal{D}} \rho; \Gamma_1 \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\, a.\sigma \xrightarrow{\text{skol } \mathcal{D}} \rho; \Gamma_1 \rightsquigarrow \Lambda a.\dot{t}}$$

SkolT-InfForall

$$\frac{\Gamma, a \vdash \sigma \xrightarrow{\text{skol } \mathcal{D}} \rho; \Gamma_1 \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\, \{a\}.\sigma \xrightarrow{\text{skol } \mathcal{D}} \rho; \Gamma_1 \rightsquigarrow \Lambda a.\dot{t}}$$

$\boxed{\Gamma \vdash \mathit{decl} \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t}$ *(Declaration Checking)*

EDecl-NoAnnSingle

$$\frac{\begin{array}{c}\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F} \\ \Gamma, \Delta \vdash e \Rightarrow \eta^{\epsilon} \rightsquigarrow t \\ \mathit{type}\,(\overline{\psi}; \eta^{\epsilon} \sim \sigma) \\ \overline{a} = f_v(\sigma) \setminus \mathit{dom}\,(\Gamma)\end{array}}{\Gamma \vdash x\,\overline{\pi} = e \Rightarrow \Gamma, x : \forall\,\overline{\{a\}}.\sigma \rightsquigarrow x : \forall\,\overline{\{a\}}.\sigma = \mathbf{case}\,\overline{\pi_F} : \overline{\psi_F} \rightarrow t}$$

EDecl-NoAnnMulti

$$\frac{\begin{array}{c}i > 1 \\ \overline{\Gamma \vdash^P \overline{\pi}_i \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F}_i : \overline{\psi_F}}^{\,i} \\ \overline{\Gamma, \Delta \vdash e_i \Rightarrow \eta_i^{\epsilon} \rightsquigarrow t_i}^{\,i} \\ \overline{\Gamma, \Delta \vdash \eta_i^{\epsilon} \xrightarrow{\mathit{inst}\ \delta} \rho \rightsquigarrow \dot{t}_i}^{\,i} \\ \mathit{type}\,(\overline{\psi}; \rho \sim \sigma) \\ \overline{a} = f_v(\sigma) \setminus \mathit{dom}\,(\Gamma) \\ \sigma' = \forall\,\overline{\{a\}}.\sigma\end{array}}{\Gamma \vdash \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma, x : \sigma' \rightsquigarrow x : \sigma' = \mathbf{case}\,\overline{\overline{\pi_F}_i : \overline{\psi_F} \rightarrow \dot{t}_i[t_i]}^{\,i}}$$

EDecl-Ann

$$\frac{\begin{array}{c}\overline{\Gamma \vdash^P \overline{\pi}_i \Leftarrow \sigma \Rightarrow \sigma_i'; \Delta \rightsquigarrow \overline{\pi_F}_i : \overline{\psi_F}}^{\,i} \\ \overline{\Gamma, \Delta \vdash e_i \Leftarrow \sigma_i' \rightsquigarrow t_i}^{\,i}\end{array}}{\Gamma \vdash x : \sigma; \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma, x : \sigma \rightsquigarrow x : \sigma = \mathbf{case}\,\overline{\overline{\pi_F}_i : \overline{\psi_F} \rightarrow t_i}^{\,i}}$$

$\boxed{\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}}$ *(Pattern Synthesis)*

Pat-InfEmpty

$$\frac{}{\Gamma \vdash^P \cdot \Rightarrow \cdot; \cdot \rightsquigarrow \cdot : \cdot}$$

Pat-InfVar

$$\frac{\Gamma, x : \tau_1 \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}}{\Gamma \vdash^P x, \overline{\pi} \Rightarrow \tau_1, \overline{\psi}; x : \tau_1, \Delta \rightsquigarrow x : \tau_1, \overline{\pi_F} : \tau_1, \overline{\psi_F}}$$

Pat-InfCon

$$\frac{\begin{array}{c}K : \overline{a}_0; \overline{\sigma}_0; T \in \Gamma \\ \Gamma \vdash^P \overline{\pi} \Leftarrow [\overline{\sigma}_1, \overline{\tau}_0/\overline{a}_0]\,(\overline{\sigma}_0 \rightarrow T\,\overline{a}_0) \Rightarrow T\,\overline{\tau}; \Delta_1 \rightsquigarrow \overline{\pi_F}_1 : \overline{\psi_F}_1 \\ \Gamma, \Delta_1 \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}; \Delta_2 \rightsquigarrow \overline{\pi_F}_2 : \overline{\psi_F}_2\end{array}}{\Gamma \vdash^P (K\,@\overline{\sigma}_1\,\overline{\pi}), \overline{\pi}' \Rightarrow T\,\overline{\tau}, \overline{\psi}; \Delta_1, \Delta_2 \rightsquigarrow (K\,\overline{\pi_F}_1), \overline{\pi_F}_2 : T\,\overline{\tau}, \overline{\psi_F}_2}$$

Pat-InfTyVar

$$\frac{\Gamma, a \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}}{\Gamma \vdash^P @a, \overline{\pi} \Rightarrow @a, \overline{\psi}; a, \Delta \rightsquigarrow @a, \overline{\pi_F} : @a, \overline{\psi_F}}$$

$$\boxed{\Gamma \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}} \qquad \textit{(Pattern Checking)}$$

Pat-CheckEmpty

$$\frac{}{\Gamma \vdash^P \cdot \Leftarrow \sigma \Rightarrow \sigma; \cdot \rightsquigarrow \cdot : \cdot}$$

Pat-CheckVar

$$\frac{\Gamma, x : \sigma_1 \vdash^P \overline{\pi} \Leftarrow \sigma_2 \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}}{\Gamma \vdash^P x, \overline{\pi} \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma'; x : \sigma_1, \Delta \rightsquigarrow x : \sigma_1, \overline{\pi_F} : \sigma_1, \overline{\psi_F}}$$

Pat-CheckCon

$$\frac{\begin{array}{c} K : \overline{a}_0; \overline{\sigma}_0; T \in \Gamma \\ \Gamma \vdash \sigma_1 \xrightarrow{inst\ \delta} \rho_1 \rightsquigarrow \dot{t} \\ \Gamma \vdash^P \overline{\pi} \Leftarrow [\overline{\sigma}_1, \overline{\tau}_0/\overline{a}_0]\,(\overline{\sigma}_0 \to T\,\overline{a}_0) \Rightarrow \rho_1; \Delta_1 \rightsquigarrow \overline{\pi_F}_1 : \overline{\psi_F}_1 \\ \Gamma, \Delta_1 \vdash^P \overline{\pi}' \Leftarrow \sigma_2 \Rightarrow \sigma_2'; \Delta_2 \rightsquigarrow \overline{\pi_F}_2 : \overline{\psi_F}_2 \end{array}}{\Gamma \vdash^P (K\,@\overline{\sigma}_1\ \overline{\pi}), \overline{\pi}' \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma_2'; \Delta_1, \Delta_2 \rightsquigarrow (K\,\overline{\pi_F}_1), \overline{\pi_F}_2 : \sigma_1, \overline{\psi_F}_2}$$

Pat-CheckForall

$$\frac{\Gamma, a \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F} \quad \overline{\pi} \neq \cdot \textit{ and } \overline{\pi} \neq @\sigma, \overline{\pi}'}{\Gamma \vdash^P \overline{\pi} \Leftarrow \forall\, a.\sigma \Rightarrow \sigma'; a, \Delta \rightsquigarrow \overline{\pi_F} : @a, \overline{\psi_F}}$$

Pat-CheckTyVar

$$\frac{\Gamma, a \vdash^P \overline{\pi} \Leftarrow [a/b]\,\sigma_1 \Rightarrow \sigma_2; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}}{\Gamma \vdash^P @a, \overline{\pi} \Leftarrow \forall\, b.\sigma_1 \Rightarrow \sigma_2; a, \Delta \rightsquigarrow \overline{\pi_F} : @a, \overline{\psi_F}}$$

Pat-CheckInfForall

$$\frac{\Gamma, a \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F} \quad \overline{\pi} \neq \cdot}{\Gamma \vdash^P \overline{\pi} \Leftarrow \forall\, \{a\}.\sigma \Rightarrow \sigma'; a, \Delta \rightsquigarrow \overline{\pi_F} : @\overline{a}, \overline{\psi_F}}$$

E PROOFS

This section provides the formal proofs for the properties discussed in Section 8.

E.1 Let-Inlining and Extraction

Property 1 (Let Inlining is Type Preserving).

- *If* $\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow \eta^\epsilon$ *then* $\Gamma \vdash [e_1/x]\ e_2 \Rightarrow \eta^\epsilon$
- *If* $\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Leftarrow \sigma$ *then* $\Gamma \vdash [e_1/x]\ e_2 \Leftarrow \sigma$

Before proving Property 1, we first introduce a number of helper lemmas:

Lemma E.1 (Expression Inlining is Type Preserving (Synthesis)).
If $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma_1, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash e_2 \Rightarrow \eta_2^\epsilon$ *where* $\overline{a} = f_v(\eta_1^\epsilon) \setminus dom\,(\Gamma_1)$
then $\Gamma_1, \Gamma_2 \vdash [e_1/x]\ e_2 \Rightarrow \eta_2^\epsilon$

Lemma E.2 (Expression Inlining is Type Preserving (Checking)).
If $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma_1, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash e_2 \Leftarrow \sigma_2$ *where* $\overline{a} = f_v(\eta_1^\epsilon) \setminus dom\,(\Gamma_1)$
then $\Gamma_1, \Gamma_2 \vdash [e_1/x]\ e_2 \Leftarrow \sigma_2$

Lemma E.3 (Head Inlining is Type Preserving).
If $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma_1, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash^H h \Rightarrow \sigma_2$ *where* $\overline{a} = f_v(\eta_1^\epsilon) \setminus dom\,(\Gamma_1)$
then $\Gamma_1, \Gamma_2 \vdash^H [e_1/x]\ h \Rightarrow \sigma_2$

Lemma E.4 (Argument Inlining is Type Preserving).
If $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma_1, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash^A \overline{arg} \Leftarrow \sigma_1 \Rightarrow \sigma_2$ *where* $\overline{a} = f_v(\eta_1^\epsilon) \setminus dom\,(\Gamma_1)$
then $\Gamma_1, \Gamma_2 \vdash^A [e_1/x]\ \overline{arg} \Leftarrow \sigma_1 \Rightarrow \sigma_2$

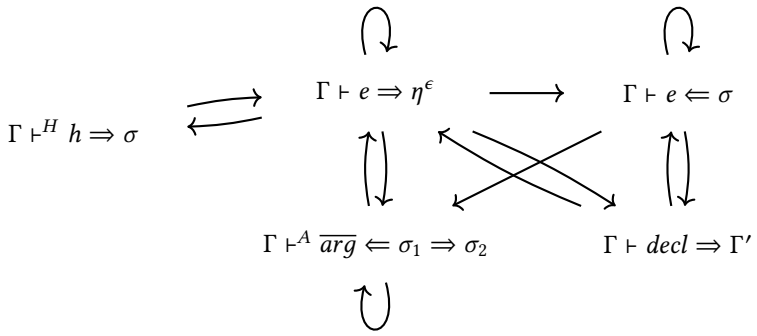

Fig. 8. Relation dependencies

Lemma E.5 (Declaration Inlining is Type Preserving).
If $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma_1, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash decl \Rightarrow \Gamma_3$ *where* $\overline{a} = f_v(\eta_1^\epsilon) \setminus dom\,(\Gamma_1)$
then $\Gamma_1, \Gamma_2 \vdash [e_1/x]\ decl \Rightarrow \Gamma_3$

Figure 8 shows the dependencies between the different relations, and by extension the different helper lemmas. An arrow from A to B denotes that B depends on A. Note that these 5 lemmas need to be proven through mutual induction. The proof proceeds by structural induction on the second typing derivation. While the number of cases gets quite large, each case is entirely trivial.

Using these additional lemmas, we then continue proving Property 1. By case analysis on the premise (rule ETm-InfLet or rule ETm-CheckLet, followed by rule EDecl-NoAnnSingle), we learn that $\Gamma \vdash x = e_1 \Rightarrow \Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon$, $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$, and either $\Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon \vdash e_2 \Rightarrow \eta^\epsilon$ or $\Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon \vdash e_2 \Leftarrow \sigma$. Both parts of the goal now follow trivially from Lemma E.1 and E.2 respectively. □

Property 2 (Let Extraction is Type Preserving).

- *If* $\Gamma \vdash [e_1/x]\ e_2 \Rightarrow \eta_2^\epsilon$ *and* $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ *then* $\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow \eta_2^\epsilon$
- *If* $\Gamma \vdash [e_1/x]\ e_2 \Leftarrow \sigma_2$ *and* $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ *then* $\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Leftarrow \sigma_2$

Similarly to before, we start by introducing a number of helper lemmas:

Lemma E.6 (Expression Extraction is Type Preserving (Synthesis)).
If $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma \vdash [e_1/x]\ e_2 \Rightarrow \eta_2^\epsilon$
then $\Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon \vdash e_2 \Rightarrow \eta_2^\epsilon$ *where* $\overline{a} = f_v(\eta_1^\epsilon) \setminus dom\,(\Gamma)$

Lemma E.7 (Expression Extraction is Type Preserving (Checking)).
If $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma \vdash [e_1/x]\ e_2 \Leftarrow \sigma_2$
then $\Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon \vdash e_2 \Leftarrow \sigma_2$ *where* $\overline{a} = f_v(\eta_1^\epsilon) \setminus dom\,(\Gamma)$

Lemma E.8 (Head Extraction is Type Preserving).
If $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma \vdash^H [e_1/x]\ h \Rightarrow \sigma_2$
then $\Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon \vdash^H h \Rightarrow \sigma_2$ *where* $\overline{a} = f_v(\eta_1^\epsilon) \setminus dom\,(\Gamma)$

Lemma E.9 (Argument Extraction is Type Preserving).
If $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma \vdash^A [e_1/x]\ \overline{arg} \Leftarrow \sigma_1 \Rightarrow \sigma_2$
then $\Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon \vdash^A \overline{arg} \Leftarrow \sigma_1 \Rightarrow \sigma_2$ *where* $\overline{a} = f_v(\eta_1^\epsilon) \setminus dom\,(\Gamma)$

Lemma E.10 (Declaration Extraction is Type Preserving).
If $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma \vdash [e_1/x]\ decl \Rightarrow \Gamma, \Gamma'$
then $\Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon \vdash decl \Rightarrow \Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon, \Gamma'$ *where* $\overline{a} = f_v(\eta_1^\epsilon) \setminus dom\,(\Gamma)$

In addition to these helper lemmas, we also introduce two typing context lemmas:

Lemma E.11 (Environment Variable Shifting is Type Preserving).

- *If* $\Gamma_1, x_1 : \sigma_1, x_2 : \sigma_2, \Gamma_2 \vdash e \Rightarrow \eta^\epsilon$ *then* $\Gamma_1, x_2 : \sigma_2, x_1 : \sigma_1, \Gamma_2 \vdash e \Rightarrow \eta^\epsilon$
- *If* $\Gamma_1, x_1 : \sigma_1, x_2 : \sigma_2, \Gamma_2 \vdash e \Leftarrow \sigma$ *then* $\Gamma_1, x_2 : \sigma_2, x_1 : \sigma_1, \Gamma_2 \vdash e \Leftarrow \sigma$

Lemma E.12 (Environment Type Variable Shifting is Type Preserving).

- *If* $\Gamma_1, a, x : \sigma, \Gamma_2 \vdash e \Rightarrow \eta^\epsilon$ *and* $\cdot = f_v(\sigma) \setminus dom\,(\Gamma_1)$ *then* $\Gamma_1, x : \sigma, a, \Gamma_2 \vdash e \Rightarrow \eta^\epsilon$
- *If* $\Gamma_1, a, x : \sigma, \Gamma_2 \vdash e \Leftarrow \sigma$ *and* $\cdot = f_v(\sigma) \setminus dom\,(\Gamma_1)$ *then* $\Gamma_1, x : \sigma, a, \Gamma_2 \vdash e \Leftarrow \sigma$
- *If* $\Gamma_1, x : \sigma, a, \Gamma_2 \vdash e \Rightarrow \eta^\epsilon$ *then* $\Gamma_1, a, x : \sigma, \Gamma_2 \vdash e \Rightarrow \eta^\epsilon$
- *If* $\Gamma_1, x : \sigma, a, \Gamma_2 \vdash e \Leftarrow \sigma$ *then* $\Gamma_1, a, x : \sigma, \Gamma_2 \vdash e \Leftarrow \sigma$

Lemmas E.11 and E.12 are folklore, and can be proven through straightforward induction.

Now we can go about proving Lemmas E.6 till E.10. Similarly to the Property 1 helper lemmas, they have to be proven using mutual induction. Most cases are quite straightforward, and we will focus only on Lemma E.8. We start by performing case analysis on h:

Case $h = y$ **where** $y = x$

By evaluating the substitution, we know from the premise that $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ and $\Gamma \vdash^H e_1 \Rightarrow \sigma_2$, while the goal remains $\Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon \vdash^H x \Rightarrow \sigma_2$. It is clear from rule H-Var that in order for the goal to hold, $\sigma_2 = \forall\, \overline{\{a\}}.\eta_1^\epsilon$. We proceed by case analysis on the second derivation:

case rule H-Var $e_1 = x'$ **:** The rule premise tells us that $x' : \sigma_2 \in \Gamma$. The goal follows directly under lazy instantiation. However, under eager instantiation, rule ETm-InfApp instantiates the type $\Gamma \vdash \sigma_2 \xrightarrow{inst\ \delta} \eta_1^\epsilon$ making the goal invalid.

case rule H-Con $e_1 = K$**, rule** H-Ann $e_1 = e_3 : \sigma_3$**, rule** H-Inf $e_1 = e_1$**, rule** H-Undef $e_1 = undefined$**, or rule** H-Seq $e_1 = seq$ **:**

Similarly to the previous case, the goal is only valid under eager instantiation.

Case $h = y$ **where** $y \neq x$

This case is trivial, as the substitution $[e_1/x]$ does not alter h. The result thus follows from weakening.

Case $h = K$, $h = undefined$, **or** $h = seq$

Similarly to the previous case, as the substitution does not alter h, the result thus follows from weakening.

Case $h = e : \sigma$

The result follows by applying Lemma E.7.

Case $h = e$

The result follows by applying Lemma E.6. □

Using these lemmas, both Property 2 goals follow straightforwardly using rule EDecl-NoAnnSingle, in combination with rule ETm-InfLet and Lemma E.6 or rule ETm-CheckLet and Lemma E.7, respectively. □

E.2 Contextual Equivalence

As we've now arrived at properties involving the dynamic semantics of the language, we first need to formalise our definition of contextual equivalence, and introduce a number of useful lemmas.

Definition 2 (Contextual Equavalence).

$$\begin{aligned} t_1 \simeq t_2 \equiv \Gamma \vdash t_1 : \sigma_1 \quad \wedge \quad & \Gamma \vdash \sigma_1 \xrightarrow{inst\ \delta} \rho_3 \rightsquigarrow \dot{t}_1 \\ \wedge\ \Gamma \vdash t_2 : \sigma_2 \quad \wedge \quad & \Gamma \vdash \sigma_2 \xrightarrow{inst\ \delta} \rho_3 \rightsquigarrow \dot{t}_2 \\ \wedge\ \forall M : \Gamma; \rho_3 \mapsto \cdot; & Bool, \\ \exists v : M[\dot{t}_1[t_1]] \hookrightarrow^{\Downarrow} v \quad \wedge \quad & M[\dot{t}_2[t_2]] \hookrightarrow^{\Downarrow} v \end{aligned}$$

This definition for contextual equivalence is modified from Harper [2016, Chapter 46]. Two core expressions are thus contextually equivalent, if a common type exists to which both their types instantiate, and if no (closed) context can distinguish between them. This can either mean that both applied expressions evaluate to the same value v or both diverge. Note that while we require the context to map to a closed, Boolean expression, other base types, like *Int*, would have been valid alternatives as well.

We first introduce reflexivity, commutativity and transitivity lemmas:

Lemma E.13 (Contextual Equivalence Reflexivity).
If $\Gamma \vdash t : \sigma$ *then* $t \simeq t$

The proof follows directly from the definition of contextual equivalence, along with the determinism of System F evaluation.

Lemma E.14 (Contextual Equivalence Commutativity).
If $t_1 \simeq t_2$ *then* $t_2 \simeq t_1$

Trivial proof by unfolding the definition of contextual equivalence.

Lemma E.15 (Contextual Equivalence Transitivity).
If $t_1 \simeq t_2$ *and* $t_2 \simeq t_3$ *then* $t_1 \simeq t_3$

Trivial proof by unfolding the definition of contextual equivalence.

Furthermore, we also introduce a number of compatibility lemmas for the contextual equivalence relation, along with two helper lemmas:

Lemma E.16 (Compatibility Term Abstraction).
If $t_1 \simeq t_2$ *then* $\lambda x : \sigma.t_1 \simeq \lambda x : \sigma.t_2$

Lemma E.17 (Compatibility Term Application).
If $t_1 \simeq t_2$ *and* $t_1' \simeq t_2'$ *then* $t_1\ t_1' \simeq t_2\ t_2'$

Lemma E.18 (Compatibility Type Abstraction).
If $t_1 \simeq t_2$ *then* $\Lambda a.t_1 \simeq \Lambda a.t_2$

Lemma E.19 (Compatibility Type Application).
If $t_1 \simeq t_2$ *then* $t_1\ \sigma \simeq t_2\ \sigma$

Lemma E.20 (Compatibility Case Abstraction).
If $\forall\ i : t_{1\,i} \simeq t_{2\,i}$ *then* $\mathbf{case}\ \overline{\pi_{F\,i} : \overline{\psi_F} \to t_{1\,i}}^{\,i} \simeq \mathbf{case}\ \overline{\pi_{F\,i} : \overline{\psi_F} \to t_{2\,i}}^{\,i}$

Lemma E.21 (Compatibility Expression Wrapper).
If $t_1 \simeq t_2$ *then* $\dot{t}[t_1] \simeq \dot{t}[t_2]$

Lemma E.22 (Compatibility Helper Forwards).
If $M[t_1] \hookrightarrow^{\Downarrow} v$ *and* $t_1 \hookrightarrow t_2$ *then* $M[t_2] \hookrightarrow^{\Downarrow} v$

Lemma E.23 (Compatibility Helper Backwards).
If $M[t_2] \hookrightarrow^{\Downarrow} v$ *and* $t_1 \hookrightarrow t_2$ *then* $M[t_1] \hookrightarrow^{\Downarrow} v$

The helper lemmas are proven by straightforward induction on the evaluation step derivation. We will prove Lemma E.18 as an example, as it is non-trivial. The other compatibility lemmas are proven similarly.

We start by unfolding the definition of contextual equivalence in both the premise: $\Gamma \vdash t_1 : \sigma_1$, $\Gamma \vdash \sigma_1 \xrightarrow{inst\ \delta} \rho_3 \rightsquigarrow \dot{t}_1$, $\Gamma \vdash t_2 : \sigma_2$, $\Gamma \vdash \sigma_2 \xrightarrow{inst\ \delta} \rho_3 \rightsquigarrow \dot{t}_2$, $\forall M : \Gamma; \rho_3 \mapsto \cdot; Bool, \exists v : M[\dot{t}_1[t_1]] \hookrightarrow^{\Downarrow} v$ and $M[\dot{t}_2[t_2]] \hookrightarrow^{\Downarrow} v$. Unfolding the definition reduces the goal to be proven to $\Gamma' \vdash \Lambda a.t_1 : \sigma'_1$, $\Gamma' \vdash \sigma'_1 \xrightarrow{inst\ \delta} \rho'_3 \rightsquigarrow \dot{t}'_1$, $\Gamma' \vdash \Lambda a.t_2 : \sigma'_2$, $\Gamma' \vdash \sigma'_2 \xrightarrow{inst\ \delta} \rho'_3 \rightsquigarrow \dot{t}'_2$, $\forall M' : \Gamma'; \rho'_3 \mapsto \cdot; Bool, \exists v' : M'[\dot{t}'_1[\Lambda a.t_1]] \hookrightarrow^{\Downarrow} v'$ and $M'[\dot{t}'_2[\Lambda a.t_2]] \hookrightarrow^{\Downarrow} v'$.

The typing judgement goals follow directly from rule FTm-TyAbs, where we take $\sigma'_1 = \forall\, a.\sigma_1$, $\sigma'_2 = \forall\, a.\sigma_2$ and $\Gamma' = [\tau/a]\,\Gamma$ for some τ.

As we know $\Gamma \vdash \sigma_1 \xrightarrow{inst\ \delta} \rho_3 \rightsquigarrow \dot{t}_1$, it is easy to see that $[\tau/a]\,\Gamma \vdash [\tau/a]\,\sigma_1 \xrightarrow{inst\ \delta} [\tau/a]\,\rho_3 \rightsquigarrow [\tau/a]\,\dot{t}_1$, and similarly for $[\tau/a]\,\sigma_2$. Using this, the instantiation goals follow from rule InstT-SForall and rule InstT-Forall with $\rho'_3 = [\tau/a]\,\rho_3$, $\dot{t}'_1 = \lambda t.([\tau/a]\,\dot{t}_1[t\,\tau])$ and $\dot{t}'_2 = \lambda t.([\tau/a]\,\dot{t}_2[t\,\tau])$.

Finally, by inlining the definitions, the first halve of the third goal becomes $M'[(\lambda t.([\tau/a]\,\dot{t}_1[t\,\tau]))[\Lambda a.t_1]] \hookrightarrow v'$. This reduces to $M'[[\tau/a]\,\dot{t}_1[(\Lambda a.t_1)\,\tau]] \hookrightarrow^{\Downarrow} v'$. By lemma E.22 (note that we can consider the combination of a context and an expression wrapper as a new context): $M'[[\tau/a]\,\dot{t}_1[[\tau/a]\,t_1]] \hookrightarrow^{\Downarrow} v'$. We can now bring the substitutions to the front, and reduce the goal (by Lemma E.23) $M''[\dot{t}_1[t_1]] \hookrightarrow^{\Downarrow} v'$ where we define $M'' = \lambda t.M'[(\Lambda a.t)\,\tau]$ (note that we use λt as meta-notation here, to simplify our definition of M''). We perform the same derivation for the second halve of the goal: $M''[\dot{t}_2[t_2]] \hookrightarrow^{\Downarrow} v'$. As $M'' : \Gamma; \rho_3 \mapsto \cdot; Bool$, the goal follows directly from the unfolded premise, where $v' = v$. □

We introduce an additional lemma stating that instantiating the type of expressions does not alter their behaviour:

Lemma E.24 (Type Instantiation is Dynamic Semantics Preserving).
If $\Gamma \vdash t : \sigma$ *and* $\Gamma \vdash \sigma \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}$ *then* $t \simeq \dot{t}[t]$

The proof proceeds by induction on the instantiation relation:

Case rule InstT-SInst $\dot{t} = \bullet$ **:**

Trivial case, as $\dot{t}[t] = t$, the goal follows directly from Lemma E.13.

Case rule InstT-SForall $\dot{t} = \lambda t_1.(\dot{t}'[t_1\,\tau])$ **:**

We know from the first premise, along with rule FTm-TyApp that $\Gamma \vdash t\,\tau : [\tau/a]\,\sigma'$ where $\sigma = \forall\, a.\sigma'$. By applying the induction hypothesis we get $t\,\tau \simeq \dot{t}'[t\,\tau]$. The goal to be proven is $t \simeq (\lambda t_1.(\dot{t}'[t_1\,\tau]))[t]$, which reduces to $t \simeq \dot{t}'[t\,\tau]$. By unfolding the definition of contextual equivalence in both the goal and the induction hypothesis result (using Lemma E.15), the remaining goals are:

- $\Gamma \vdash t : \sigma_1$: follows directly from the first premise.
- $\Gamma \vdash \forall\, a.\sigma' \xrightarrow{inst\ \mathcal{S}} \rho' \rightsquigarrow \dot{t}_1$ and $\Gamma \vdash \rho' \xrightarrow{inst\ \mathcal{S}} \rho \rightsquigarrow \dot{t}_2$: follows directly from the premise if we take $\rho' = \rho$, $\dot{t}_1 = \dot{t}$ and $\dot{t}_2 = \bullet$.
- $M[\dot{t}_1[t]] \hookrightarrow^{\Downarrow} v$ and $M[\dot{t}[t]] \hookrightarrow^{\Downarrow} v$: trivial as both sides are identical and evaluation is deterministic.

Case rule InstT-SInfForall $\dot{t} = \lambda t_1.(\dot{t}'[t_1\,\tau])$ **:**

The proof follows analogously to the previous case. We have thus proven Lemma E.24 under shallow instantiation.

Case rule InstT-Mono $\dot{t} = \bullet$ **:**

Trivial case, as $\dot{t}[t] = t$, the goal follows directly from Lemma E.13.

Case rule InstT-Function $\dot{t} = \lambda t_1.\lambda x : \sigma_1.(\dot{t}'[t_1\ x])$ **:**

It is clear that the goal does not hold in this case. Under deep instantiation, full eta expansion is performed, which alters the evaluation behaviour. Consider for example *undefined* and its expansion $\lambda x : \sigma.\mathit{undefined}\ x$. □

Finally, we introduce a lemma stating that evaluation preserves contextual equivalence. However, in order to prove it, we first need to introduce the common preservation lemma:

Lemma E.25 (Preservation).
If $\Gamma \vdash t : \sigma$ *and* $t \hookrightarrow t'$ *then* $\Gamma \vdash t' : \sigma$

The preservation proof for System F is folklore, and proceeds by straightforward induction on the evaluation relation.

Lemma E.26 (Evaluation is Contextual Equivalence Preserving).
If $t_1 \simeq t_2$ *and* $t_2 \hookrightarrow t_2'$ *then* $t_1 \simeq t_2'$

The proof follows by Lemma E.25 (to cover type preservation) and Lemma E.22 (to cover the evaluation aspect).

E.3 Let-Inlining and Extraction, Continued

Property 3 (Let Inlining is Dynamic Semantics Preserving).

- *If* $\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Rightarrow \eta^{\epsilon} \rightsquigarrow t_1$ *and* $\Gamma \vdash [e_1/x]\ e_2 \Rightarrow \eta^{\epsilon} \rightsquigarrow t_2$ *then* $t_1 \simeq t_2$
- *If* $\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Leftarrow \sigma \rightsquigarrow t_1$ *and* $\Gamma \vdash [e_1/x]\ e_2 \Leftarrow \sigma \rightsquigarrow t_2$ *then* $t_1 \simeq t_2$

We first need typing preservation lemmas before we can prove Property 3.

Lemma E.27 (Expression Typing Preservation (Synthesis)).
If $\Gamma \vdash e \Rightarrow \eta \rightsquigarrow t$ *then* $\Gamma \vdash t : \eta$

Lemma E.28 (Expression Typing Preservation (Checking)).
If $\Gamma \vdash e \Leftarrow \sigma \rightsquigarrow t$ *then* $\Gamma \vdash t : \sigma$

Lemma E.29 (Head Typing Preservation).
If $\Gamma \vdash^{H} h \Rightarrow \sigma \rightsquigarrow t$ *then* $\Gamma \vdash t : \sigma$

Lemma E.30 (Argument Typing Preservation).
If $\Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma \Rightarrow \sigma' \rightsquigarrow \overline{arg_F}$ *then* $\forall t_i \in \overline{arg_F} : \Gamma \vdash t_i : \sigma_i$

Lemma E.31 (Declaration Typing Preservation).
If $\Gamma \vdash \mathit{decl} \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t$ *then* $\Gamma \vdash t : \sigma$

Similarly to the helper lemmas for Property 1, these lemmas need to be proven using mutual induction. The proofs follow through straightforward induction on the typing derivation.

We continue by introducing another set of helper lemmas:

Lemma E.32 (Expression Inlining is Dynamic Semantics Preserving (Synthesis)).
If $\Gamma_1, x : \forall\ \overline{\{a\}}.\eta_1^{\epsilon}, \Gamma_2 \vdash e_2 \Rightarrow \eta_2^{\epsilon} \rightsquigarrow t_2$, $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^{\epsilon} \rightsquigarrow t_1$ *and* $\Gamma_1, \Gamma_2 \vdash [e_1/x]\ e_2 \Rightarrow \eta_2^{\epsilon} \rightsquigarrow t_3$ *where* $\overline{a} = f_v(\eta_1^{\epsilon}) \setminus \mathit{dom}(\Gamma_1)$ *then* $t_3 \simeq (\lambda x : \forall\ \overline{a}.\eta_1^{\epsilon}.t_2)\ t_1$

Lemma E.33 (Expression Inlining is Dynamic Semantics Preserving (Checking)).
If $\Gamma_1, x : \forall\ \overline{\{a\}}.\eta_1^{\epsilon}, \Gamma_2 \vdash e_2 \Leftarrow \sigma_2 \rightsquigarrow t_2$, $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^{\epsilon} \rightsquigarrow t_1$ *and* $\Gamma_1, \Gamma_2 \vdash [e_1/x]\ e_2 \Leftarrow \sigma_2 \rightsquigarrow t_3$ *where* $\overline{a} = f_v(\eta_1^{\epsilon}) \setminus \mathit{dom}(\Gamma_1)$ *then* $t_3 \simeq (\lambda x : \forall\ \overline{a}.\eta_1^{\epsilon}.t_2)\ t_1$

Lemma E.34 (Head Inlining is Dynamic Semantics Preserving).
If $\Gamma_1, x : \forall \overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash^H h \Rightarrow \sigma \rightsquigarrow t_2$, $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon \rightsquigarrow t_1$ *and* $\Gamma_1, \Gamma_2 \vdash^H [e_1/x]\, h \Rightarrow \sigma \rightsquigarrow t_3$ *where* $\overline{a} = f_v(\eta_1^\epsilon) \setminus dom(\Gamma_1)$ *then* $t_3 \simeq (\lambda x : \forall \overline{a}.\eta_1^\epsilon.t_2)\ t_1$

Lemma E.35 (Argument Inlining is Dynamic Semantics Preserving).
If $\Gamma_1, x : \forall \overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash^A \overline{arg} \Leftarrow \sigma_1 \Rightarrow \sigma_2 \rightsquigarrow \overline{arg_{F_1}}$, $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon \rightsquigarrow t_1$
and $\Gamma_1, \Gamma_2 \vdash^A [e_1/x]\, \overline{arg} \Leftarrow \sigma_1 \Rightarrow \sigma_2 \rightsquigarrow \overline{arg_{F_2}}$ *where* $\overline{a} = f_v(\eta_1^\epsilon) \setminus dom(\Gamma_1)$
then $\forall t_i \in \overline{arg_{F_1}},\ t_i' \in \overline{arg_{F_2}} : t_i' \simeq (\lambda x : \forall \overline{a}.\eta_1^\epsilon.t_i)\ t_1$

Lemma E.36 (Declaration Inlining is Dynamic Semantics Preserving).
If $\Gamma_1, x : \forall \overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash decl \Rightarrow \Gamma_3 \rightsquigarrow y : \sigma_2 = t_2$, $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon \rightsquigarrow t_1$ *and* $\Gamma_1, \Gamma_2 \vdash [e_1/x]\, decl \Rightarrow \Gamma_3 \rightsquigarrow y : \sigma_2 = t_3$ *where* $\overline{a} = f_v(\eta_1^\epsilon) \setminus dom(\Gamma_1)$ *then* $t_3 \simeq (\lambda x : \forall \overline{a}.\eta_1^\epsilon.t_2)\ t_1$

As is probably clear by now, these lemmas are proven through mutual induction. The proof proceeds by structural induction on the first typing derivation. We will focus on the non-trivial cases:

Case rule H-Var $h = y$ **where** $y = x$ **:**
The goal reduces to $t_1 \simeq (\lambda x : \forall \overline{a}.\eta_1^\epsilon.x)\ t_1$, which follows directly from Lemmas E.13 and E.26.

Case rule H-Var $h = y$ **where** $y \neq x$ **:**
The goal reduces to $y \simeq (\lambda x : \forall \overline{a}.\eta_1^\epsilon.y)\ t_1$. Since $(\lambda x : \forall \overline{a}.\eta_1^\epsilon.y)\ t_1 \hookrightarrow y$, the goal follows directly from Lemmas E.13 and E.26.

Case rule ETm-InfAbs $e_2 = \lambda y.e_4$ **:**
The premise tells us $\Gamma_1, x : \forall \overline{\{a\}}.\eta_1^\epsilon, \Gamma_2, y : \tau_1 \vdash e_4 \Rightarrow \eta_4^\epsilon \rightsquigarrow t_4$ and $\Gamma_1, \Gamma_2, y : \tau_1 \vdash [e_1/x]\, e_4 \Rightarrow \eta_4^\epsilon \rightsquigarrow t_5$. Applying the induction hypothesis gives us $t_5 \simeq (\lambda x : \forall \overline{a}.\eta_1^\epsilon.t_4)\ t_1$. The goal reduces to $\lambda y : \tau_1.t_5 \simeq (\lambda x : \forall \overline{a}.\eta_1^\epsilon.\lambda y : \tau_1.t_4)\ t_1$. In order not to clutter the proof too much, we introduce an additional helper lemma E.37. The goal then follows from Lemmas E.16 and E.37.

Case rule ETm-InfTyAbs $e_2 = \Lambda a.e_4$ **:**
The premise tells us $\Gamma_1, x : \forall \overline{\{a\}}.\eta_1^\epsilon, \Gamma_2, a \vdash e_4 \Rightarrow \eta_4^\epsilon \rightsquigarrow t_4$, $\Gamma_1, \Gamma_2, a \vdash [e_1/x]\, e_4 \Rightarrow \eta_4^\epsilon \rightsquigarrow t_5$ and $\Gamma_1, \Gamma_2 \vdash \forall a.\eta_4^\epsilon \xrightarrow{inst\ \delta} \eta_5^\epsilon \rightsquigarrow \dot{t}$. Applying the induction hypothesis gives us $t_5 \simeq (\lambda x : \forall \overline{a}.\eta_1^\epsilon.t_4)\ t_1$. The goal reduces to $\dot{t}[\Lambda a.t_5] \simeq (\lambda x : \forall \overline{a}.\eta_1^\epsilon.\dot{t}[\Lambda a.t_4])\ t_1$. Similarly to before, we avoid cluttering the proof by introducing an additional helper lemma E.38. The goal then follows from Lemmas E.18, E.24 and E.38. □

Lemma E.37 (Property 3 Term Abstraction Helper).
If $\Gamma \vdash \lambda x : \sigma_2.((\lambda y : \sigma_1.t_2)\ t_1) : \sigma_3$ *and* $\Gamma \vdash t_1 : \sigma_1$ *then* $\lambda x : \sigma_2.((\lambda y : \sigma_1.t_2)\ t_1) \simeq (\lambda y : \sigma_1.\lambda x : \sigma_2.t_2)\ t_1$

Lemma E.38 (Property 3 Type Abstraction Helper).
If $\Gamma \vdash \Lambda a.((\lambda x : \sigma_1.t_2)\ t_1) : \sigma_2$ *and* $a \notin f_v(\sigma_1)$ *then* $\Lambda a.((\lambda x : \sigma_1.t_2)\ t_1) \simeq (\lambda x : \sigma_1.\Lambda a.t_2)\ t_1$

Both lemmas follow from the definition of contextual equivalence.

We now return to proving Property 3. By case analysis (Either rule ETm-InfLet or rule ETm-CheckLet, followed by rule EDecl-NoAnnSingle) we know $\Gamma, x : \forall \overline{\{a\}}.\eta_1^\epsilon \vdash e_2 \Rightarrow \eta^\epsilon \rightsquigarrow t_3$ or $\Gamma, x : \forall \overline{\{a\}}.\eta_1^\epsilon \vdash e_2 \Leftarrow \sigma \rightsquigarrow t_3$ where $t_1 = (\lambda x : \forall \overline{a}.\eta_1^\epsilon.t_3)\ t_4$, $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon \rightsquigarrow t_4$ and $\overline{a} = f_v(\eta_1^\epsilon) \setminus dom(\Gamma)$. The goal thus follows directly from Lemma E.32 or E.33. However, as Lemma E.24 only holds under shallow instantiation, we cannot prove Property 3 under deep instantiation. □

E.4 Type Signatures

Property 4b (Signature Property is Type Preserving).
If $\Gamma \vdash x\,\overline{\pi} = e \Rightarrow \Gamma'$ *and* $x : \sigma \in \Gamma'$ *then* $\Gamma \vdash x : \sigma; x\,\overline{\pi} = e \Rightarrow \Gamma'$

Before proving Property 4b, we first introduce a number of helper lemmas:

Lemma E.39 (Skolemisation Exists).
If $f_v(\sigma) \in \Gamma$ *then* $\exists \rho, \Gamma'$ *such that* $\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho; \Gamma'$

The proof follows through careful examination of the skolemisation relation.

Lemma E.40 (Skolemisation Implies Instantiation).
If $\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho; \Gamma'$ *then* $\Gamma' \vdash \sigma \xrightarrow{inst\ \delta} \rho$

The proof follows by straightforward induction on the skolemisation relation. Note that as skolemisation binds all type variables in Γ', they can then be used for instantiation.

Lemma E.41 (Inferred Type Binders Preserve Expression Checking).
If $\Gamma \vdash e \Leftarrow \sigma$ *then* $\Gamma \vdash e \Leftarrow \forall\, \overline{\{a\}}.\sigma$

The proof follows by straightforward induction on the typing derivation.

Lemma E.42 (Pattern Synthesis Implies Checking).
If $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta$ *then* $\forall \sigma', \exists \sigma : \Gamma \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta$ *where type* $(\overline{\psi}; \sigma' \sim \sigma)$

The proof follows by straightforward induction on the pattern typing derivation.

Lemma E.43 (Expression Synthesis Implies Checking).
If $\Gamma \vdash e \Rightarrow \eta^\epsilon$ *then* $\Gamma \vdash e \Leftarrow \eta^\epsilon$

The proof follows by induction on the typing derivation. We will focus on the non-trivial cases below:

Case rule ETm-InfAbs $e = \lambda x.e'$ **:**

We know from the premise of the typing rule that $\Gamma, x : \tau_1 \vdash e' \Rightarrow \eta_2^\epsilon$ where $\eta^\epsilon = \tau_1 \to \eta_2^\epsilon$. By rule ETm-CheckAbs, the goal reduces to $\Gamma \vdash \tau_1 \to \eta_2^\epsilon \xrightarrow{skol\ \mathcal{S}} \tau_1 \to \eta_2^\epsilon; \Gamma$ (which follows directly by rule SkolT-SInst) and $\Gamma, x : \tau_1 \vdash e' \Leftarrow \eta_2^\epsilon$ (which follows by the induction hypothesis).

Case rule ETm-InfTyAbs $e = \Lambda a.e'$ **:**

The typing rule premise tells us that $\Gamma, a \vdash e' \Rightarrow \eta_1^\epsilon$ and $\Gamma \vdash \forall\, a.\eta_1^\epsilon \xrightarrow{inst\ \delta} \eta_2^\epsilon$. By rule ETm-CheckTyAbs, the goal reduces to $\eta_2^\epsilon = \forall\, \overline{\{a\}}.\forall\, a.\sigma'$ and $\Gamma, \overline{\{a\}}, a \vdash e' \Leftarrow \sigma'$. It is now clear that this property can never hold under eager instantiation, as the forall type in $\forall\, a.\eta_1^\epsilon$ would always be instantiated away. We will thus focus solely on lazy instantiation from here on out, where $\eta_2^\epsilon = \forall\, a.\eta_1^\epsilon$. In this case, the goal follows directly from the induction hypothesis.

Case rule ETm-InfApp $e = h\, \overline{arg}$ **:**

We know from the typing rule premise that $\Gamma \vdash^H h \Rightarrow \sigma, \Gamma \vdash^A \overline{arg} \Leftarrow \sigma \Rightarrow \sigma'$ and $\Gamma \vdash \sigma' \xrightarrow{inst\ \delta} \eta^\epsilon$. Note that as we assume lazy instantiation, $\eta^\epsilon = \sigma'$. By rule ETm-CheckInf, the goal reduces to $\Gamma \vdash \eta^\epsilon \xrightarrow{skol\ \delta} \rho; \Gamma'$ (follows by Lemma E.39), $\Gamma' \vdash h\, \overline{arg} \Rightarrow \eta_1^\epsilon$ (follows by performing environment weakening on the premise, with $\eta_1^\epsilon = \eta^\epsilon$) and $\Gamma' \vdash \eta_1^\epsilon \xrightarrow{inst\ \delta} \rho$ (given that $\eta_1^\epsilon = \eta^\epsilon$, this follows by Lemma E.40). □

We now proceed with proving Property 4b, through case analysis on the declaration typing derivation (rule EDecl-NoAnnSingle):

We know from the typing rule premise that $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta, \Gamma, \Delta \vdash e \Rightarrow \eta^\epsilon$, $type\ (\overline{\psi}; \eta^\epsilon \sim \sigma_1)$ and $\sigma = \forall\, \overline{\{a\}}.\sigma_1$ where $\overline{a} = f_v(\sigma_1) \setminus dom\ (\Gamma)$. By rule EDecl-Ann, the goal reduces to $\Gamma \vdash^P \overline{\pi} \Leftarrow \forall\, \overline{\{a\}}.\sigma_1 \Rightarrow \sigma_2; \Delta_2$ and $\Gamma, \Delta_2 \vdash e \Leftarrow \sigma_2$. We know from Lemma E.42 that $\Gamma \vdash^P \overline{\pi} \Leftarrow \sigma_1 \Rightarrow \sigma_3; \Delta$ where $type\ (\overline{\psi}; \sigma_3 \sim \sigma_1)$. Furthermore, from Lemma E.43 we get $\Gamma, \Delta \vdash e \Leftarrow \eta^\epsilon$. Note that we thus only prove Property 4b under lazy instantiation. We now proceed by case analysis on $\overline{\pi}$:

Case $\overline{\pi} = \cdot$:

The first goal now follows trivially by rule Pat-CheckEmpty with $\sigma_2 = \forall\, \overline{\{a\}}.\sigma_1$, $\sigma_1 = \eta^\epsilon$ and $\Delta = \Delta_2 = \cdot$. The second goal follows by Lemma E.41.

Case $\overline{\pi} \neq \cdot$:

The first goal follows by repeated application of rule Pat-CheckInfForall with $\sigma_2 = \sigma_3 = \eta^\epsilon$. The second goal then follows directly from Lemma E.43. □

Property 5 (Signature Property is Dynamic Semantics Preserving).
If $\Gamma \vdash \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t_1$ *and* $\Gamma \vdash x : \sigma; \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t_2$ *then* $t_1 \simeq t_2$

We start by introducing a number of helper lemmas:

Lemma E.44 (Pattern Typing Mode Preserves Translation).
If $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_{F_1}} : \overline{\psi_{F_1}}$ *and* $\Gamma \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{\pi_{F_2}} : \overline{\psi_{F_2}}$ *where type* $(\overline{\psi}; \sigma' \sim \sigma)$
then $\overline{\pi_{F_1}} = \overline{\pi_{F_2}}$ *and* $\overline{\psi_{F_1}} = \overline{\psi_{F_2}}$

The proof follows by straightforward induction on the pattern type inference derivation.

Lemma E.45 (Compatibility One-Sided Type Abstraction).
If $t_1 \simeq t_2$ *then* $t_1 \simeq \Lambda a.t_2$

The proof follows by the definition of contextual equivalence. Note that while the left and right hand sides have different types, they still instantiate to a single common type.

Lemma E.46 (Partial Skolemisation Preserves Type Checking and Dynamic Semantics).
If $\Gamma \vdash e \Leftarrow \forall\, \overline{\{a\}}.\sigma \rightsquigarrow t_1$ *then* $\Gamma, \overline{a} \vdash e \Leftarrow \sigma \rightsquigarrow t_2$ *where* $t_1 \simeq t_2$.

The proof proceeds by induction on the type checking derivation. Note that every case performs a (limited) form of skolemisation. Every case proceeds by applying the induction hypothesis, followed by Lemma E.45.

Lemma E.47 (Typing Mode Preserves Dynamic Semantics).
If $\Gamma \vdash e \Rightarrow \eta^\epsilon \rightsquigarrow t_1$ *and* $\Gamma \vdash e \Leftarrow \sigma \rightsquigarrow t_2$ *where* $\Gamma \vdash \eta^\epsilon \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_1$ *and* $\Gamma \vdash \sigma \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_2$
then $t_1 \simeq t_2$

The proof proceeds by induction on the first typing derivation. Each case follows straightforwardly by applying the induction hypothesis, along with the corresponding compatibility lemma (Lemmas E.16 till E.20).

We now turn to proving property 5, through case analysis on the first declaration typing derivation:

Case rule EDecl-NoAnnSingle **:**

We know from the premise of the first derivation that $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_{F_1}} : \overline{\psi_{F_1}}, \Gamma, \Delta \vdash e \Rightarrow \eta^\epsilon \rightsquigarrow t_1'$, $type\,(\overline{\psi}; \eta^\epsilon \sim \sigma_1)$, $t_1 = \mathbf{case}\, \overline{\pi_{F_1}} : \overline{\psi_{F_1}} \to t_1'$ and $\sigma = \forall\, \overline{\{a\}}.\sigma_1$ where $\overline{a} = f_v(\sigma_1) \setminus dom\,(\Gamma)$. By case analysis on the second derivation (rule EDecl-Ann), we get $\Gamma \vdash^P \overline{\pi} \Leftarrow \forall\, \overline{\{a\}}.\sigma_1 \Rightarrow \sigma_2; \Delta \rightsquigarrow \overline{\pi_{F_2}} : \overline{\psi_{F_2}}, \Gamma, \Delta \vdash e \Leftarrow \sigma_2 \rightsquigarrow t_2'$ and $t_2 = \mathbf{case}\, \overline{\pi_{F_2}} : \overline{\psi_{F_2}} \to t_2'$.

We proceed by case analysis on the patterns $\overline{\pi}$:

case $\overline{\pi} = \cdot$: We know from rule Pat-InfEmpty, rule Pat-CheckEmpty and rule Type-Empty that $\sigma_2 = \forall\, \overline{\{a\}}.\sigma_1 = \forall\, \overline{\{a\}}.\eta^\epsilon$. By applying Lemma E.46, we get $\Gamma, \overline{a} \vdash e \Leftarrow \eta^\epsilon \rightsquigarrow t_3$ where $t_2' \simeq t_3$. The goal now follows by Lemma E.47 (after environment weakening, where $\sigma = \rho = \eta^\epsilon$), and Lemma E.15.

case $\overline{\pi} \neq \cdot$: By case analysis on the pattern checking derivation (rule Pat-CheckInfForall), we know that $\Gamma, \overline{a} \vdash^P \overline{\pi} \Leftarrow \sigma_1 \Rightarrow \sigma_2; \Delta' \rightsquigarrow \overline{\pi_{F_2}} : \overline{\psi_{F_2}'}$ where $\Delta = \overline{a}, \Delta'$ and $\overline{\psi_{F_2}} = @\overline{a}, \overline{\psi_{F_2}'}$.

By Lemma E.42 (where we take $\sigma = \sigma_1$), we know that $type\,(\overline{\psi}; \sigma_2 \sim \sigma_1)$. This thus means that $\sigma_2 = \eta^\epsilon$. By Lemma E.44, the goal reduces to $\mathbf{case}\,\overline{\pi_{F1}} : \overline{\psi_{F1}} \to t_1' \simeq \mathbf{case}\,\overline{\pi_{F1}} : \overline{\psi_{F1}} \to t_2'$. Applying Lemma E.20 reduces this goal further to $t_1' \simeq t_2'$. This follows directly from Lemma E.47 (where $\sigma = \rho = \eta^\epsilon$).

Case rule EDecl-NoAnnMulti **:**

We know from the premise of the first derivation that $\forall i : \Gamma \vdash^P \overline{\pi}_i \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_{F\,i}} : \overline{\psi_F}{}_i$, $\Gamma, \Delta \vdash e_i \Rightarrow \eta_i^\epsilon \rightsquigarrow t_i$ and $\Gamma, \Delta \vdash \eta_i^\epsilon \xrightarrow{inst\ \delta} \rho' \rightsquigarrow \dot{t}_i$. Furthermore, $t_1 = \mathbf{case}\,\overline{\pi_{F\,i}} : \overline{\psi_F} \to \dot{t}_i[t_i]{}_i$, $type\,(\overline{\psi}; \rho' \sim \sigma')$ and $\sigma = \forall\,\overline{\{a\}}.\sigma'$ where $\overline{a} = f_v(\sigma') \setminus dom\,(\Gamma)$. By case analysis on the second derivation (rule EDecl-Ann), we know that $\forall i : \Gamma \vdash^P \overline{\pi}_i \Leftarrow \forall\,\overline{\{a\}}.\sigma' \Rightarrow \sigma_i; \Delta \rightsquigarrow \overline{\pi_{F\,i}'} : \overline{\psi_F}'$, $\Gamma, \Delta \vdash e_i \Leftarrow \sigma_i \rightsquigarrow t_i'$ and $t_2 = \mathbf{case}\,\overline{\pi_{F\,i}'} : \overline{\psi_F}' \to t_i'{}_i$.

We again perform case analysis on the patterns $\overline{\pi}$:

case $\overline{\pi} = \cdot$ **:** Similarly to last time, we know that $\sigma' = \rho'$ and $\forall i : \sigma_i = \forall\,\overline{\{a\}}.\rho'$. We know by Lemma E.46 that $\forall i : \Gamma, \overline{a} \vdash e_i \Leftarrow \rho' \rightsquigarrow t_i''$ where $t_i' \simeq t_i''$. The goal now follows by Lemma E.47 (where we take $\sigma = \rho = \rho'$) and Lemma E.15.

case $\overline{\pi} \neq \cdot$ **:** Similarly to the previous case, we can derive that $\forall i : \Gamma, \overline{a} \vdash^P \overline{\pi} \Leftarrow \sigma' \Rightarrow \sigma_i; \Delta' \rightsquigarrow \overline{\pi_{F\,i}'} : \overline{\psi_F}''$ where $\Delta = \overline{a}, \Delta'$ and $\overline{\psi_F}' = @\overline{a}, \overline{\psi_F}''$. We again derive by Lemma E.42 that $type\,(\overline{\psi}; \sigma_i \sim \sigma')$ and thus that $\sigma_i = \rho'$. By Lemma E.44, the goal reduces to $\mathbf{case}\,\overline{\pi_{F\,i}} : \overline{\psi_F} \to \dot{t}_i[t_i]{}_i \simeq \mathbf{case}\,\overline{\pi_{F\,i}} : \overline{\psi_F} \to t_i'{}_i$. We reduce this goal further by applying Lemma E.20 to $\forall i : \dot{t}_i[t_i] \simeq t_i'$. This follows directly from Lemma E.47 (where $\sigma = \rho = \rho'$).

Note however, that as Lemma E.47 only holds under shallow instantiation, that the same holds true for Property 5. □

Property 6 (Type Signatures are Dynamic Semantics Preserving).
If $\Gamma \vdash x : \sigma_1; \overline{x\,\overline{\pi}_i = e_i}{}_i \Rightarrow \Gamma_1 \rightsquigarrow x : \sigma_1 = t_1$ *and* $\Gamma \vdash x : \sigma_2; \overline{x\,\overline{\pi}_i = e_i}{}_i \Rightarrow \Gamma_1 \rightsquigarrow x : \sigma_2 = t_2$
where $\Gamma \vdash \sigma_1 \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_1$ *and* $\Gamma \vdash \sigma_2 \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_2$ *then* $\dot{t}_1[t_1] \simeq \dot{t}_2[t_2]$

We start by introducing a number of helper lemmas:

Lemma E.48 (Substitution in Expressions is Type Preserving (Synthesis)).
If $\Gamma, a \vdash e \Rightarrow \eta^\epsilon \rightsquigarrow t$ *then* $\Gamma \vdash [\tau/a]\,e \Rightarrow [\tau/a]\,\eta^\epsilon \rightsquigarrow [\tau/a]\,t$

Lemma E.49 (Substitution in Expressions is Type Preserving (Checking)).
If $\Gamma, a \vdash e \Leftarrow \sigma \rightsquigarrow t$ *then* $\Gamma \vdash [\tau/a]\,e \Leftarrow [\tau/a]\,\sigma \rightsquigarrow [\tau/a]\,t$

Lemma E.50 (Substitution in Heads is Type Preserving).
If $\Gamma, a \vdash^H h \Rightarrow \sigma \rightsquigarrow t$ *then* $\Gamma \vdash^H [\tau/a]\,h \Rightarrow [\tau/a]\,\sigma \rightsquigarrow [\tau/a]\,t$

Lemma E.51 (Substitution in Arguments is Type Preserving).
If $\Gamma, a \vdash^A \overline{arg} \Leftarrow \sigma \Rightarrow \sigma' \rightsquigarrow \overline{arg_F}$ *then* $\Gamma \vdash^A [\tau/a]\,\overline{arg} \Leftarrow [\tau/a]\,\sigma \Rightarrow [\tau/a]\,\sigma' \rightsquigarrow [\tau/a]\,\overline{arg_F}$

Lemma E.52 (Substitution in Declarations is Type Preserving).
If $\Gamma, a \vdash decl \Rightarrow \Gamma, a, x : \sigma \rightsquigarrow x : \sigma = t$ *then* $\Gamma \vdash [\tau/a]\,decl \Rightarrow \Gamma, x : [\tau/a]\,\sigma \rightsquigarrow x : \sigma = [\tau/a]\,t$

The proof proceeds by mutual induction on the typing derivation. While the number of cases gets pretty large, each is quite straightforward.

Lemma E.53 (Type Instantiation Produces Equivalent Expressions (Synthesis)).
If $\Gamma_1 \vdash e \Rightarrow \eta_1^\epsilon \rightsquigarrow t_1$, $\Gamma_2 \vdash e \Rightarrow \eta_2^\epsilon \rightsquigarrow t_2$ *and* $\exists\,\overline{a} \subseteq f_v(\eta_1^\epsilon) \cup f_v(\eta_2^\epsilon)$
such that $\Gamma' = [\overline{\tau}/\overline{a}]\,\Gamma_1 = [\overline{\tau}/\overline{a}]\,\Gamma_2$ *and* $\Gamma' \vdash \forall\,\overline{a}.\eta_1^\epsilon \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_1$ *and* $\Gamma' \vdash \forall\,\overline{a}.\eta_2^\epsilon \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_2$
then $\dot{t}_1[\Lambda\overline{a}.t_1] \simeq \dot{t}_2[\Lambda\overline{a}.t_2]$

Lemma E.54 (Type Instantiation Produces Equivalent Expressions (Checking)).
If $\Gamma_1 \vdash e \Leftarrow \sigma_1 \rightsquigarrow t_1$ *and* $\Gamma_2 \vdash e \Leftarrow \sigma_2 \rightsquigarrow t_2$ *and* $\exists\, \overline{a} \subseteq f_v(\sigma_1) \cup f_v(\sigma_2)$
such that $\Gamma' = [\overline{\tau}/\overline{a}]\,\Gamma_1 = [\overline{\tau}/\overline{a}]\,\Gamma_2$ *and* $\Gamma' \vdash \forall\, \overline{a}.\sigma_1 \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_1$ *and* $\Gamma' \vdash \forall\, \overline{a}.\sigma_2 \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_2$
then $\dot{t}_1[\Lambda \overline{a}.t_1] \simeq \dot{t}_2[\Lambda \overline{a}.t_2]$

Lemma E.55 (Type Instantiation Produces Equivalent Expressions (Head Judgement)).
If $\Gamma_1 \vdash^H h \Rightarrow \sigma_1 \rightsquigarrow t_1$, $\Gamma_2 \vdash^H h \Rightarrow \sigma_2 \rightsquigarrow t_2$ *and* $\exists\, \overline{a} \subseteq f_v(\eta_1^\epsilon) \cup f_v(\eta_2^\epsilon)$
such that $\Gamma' = [\overline{\tau}/\overline{a}]\,\Gamma_1 = [\overline{\tau}/\overline{a}]\,\Gamma_2$ *and* $\Gamma' \vdash \forall\, \overline{a}.\sigma_1 \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_1$ *and* $\Gamma' \vdash \forall\, \overline{a}.\sigma_2 \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_2$
then $\dot{t}_1[\Lambda \overline{a}.t_1] \simeq \dot{t}_2[\Lambda \overline{a}.t_2]$

Note that we define $[\tau/a]\,\Gamma$ as removing a from the environment Γ and substituting any occurrence of a in types bound to term variables. Furthermore, we use $\overline{a}_1 \cup \overline{a}_2$ as a shorthand for list concatenation, removing duplicates. The proof proceeds by induction on the first typing derivation. Note that Lemmas E.53, E.54 and E.55 have to be proven using mutual induction. However, the proof for Lemma E.55 is trivial, as every case besides rule H-Inf is deterministic. As usual, we will focus on the non-trivial cases:

Case rule ETm-CheckAbs $e = \lambda x.e'$ **:**

We know from the premise of the first and second (as the relation is syntax directed) typing derivation that $\Gamma_1 \vdash \sigma_1 \overset{skol\ \mathcal{S}}{\dashrightarrow} \sigma_4 \to \sigma_5; \Gamma_1' \rightsquigarrow \dot{t}_1'$, $\Gamma_2 \vdash \sigma_2 \overset{skol\ \mathcal{S}}{\dashrightarrow} \sigma_4' \to \sigma_5'; \Gamma_2' \rightsquigarrow \dot{t}_2'$, $\Gamma_1', x : \sigma_4 \vdash e' \Leftarrow \sigma_5 \rightsquigarrow t_3$ and $\Gamma_2', x : \sigma_4' \vdash e' \Leftarrow \sigma_5' \rightsquigarrow t_4$, where $t_1 = \dot{t}_1'[\lambda x : \sigma_4.t_3]$ and $t_2 = \dot{t}_2'[\lambda x : \sigma_4'.t_4]$.

At this point, it is already clear that Lemma E.54 can not hold under deep instantiation, as instantiation performs full eta expansion. We will thus focus on shallow instantiation from here on out.

By case analysis on the skolemisation and instantiation premises, it is clear that $\Gamma_1' = \Gamma_1, \overline{a}_1$, $\Gamma_2' = \Gamma_2, \overline{a}_2$ and $\rho = [\overline{\tau}_1/\overline{a}_1]\,(\sigma_4 \to \sigma_5) = [\overline{\tau}_2/\overline{a}_2]\,(\sigma_4' \to \sigma_5') = \sigma_3 \to \sigma_3'$. In order to apply the induction hypothesis, we take $\overline{a}'$ as $\overline{a} \cup \overline{a}_1 \cup \overline{a}_2$. Note that this does not alter the instantiation to ρ in any way, as these variables would already have been instantiated. We apply the induction hypothesis with $\Gamma_1 \vdash \forall\, \overline{a}'.\sigma_5 \xrightarrow{inst\ \delta} \sigma_3' \rightsquigarrow \dot{t}_3$ and $\Gamma_2 \vdash \forall\, \overline{a}'.\sigma_5' \xrightarrow{inst\ \delta} \sigma_3' \rightsquigarrow \dot{t}_4$ (after weakening), producing $\dot{t}_3[\Lambda \overline{a}'.t_3] \simeq \dot{t}_4[\Lambda \overline{a}'.t_4]$. Under shallow instantiation, these two instantiations follow directly from the premise with $\dot{t}_3 = \dot{t}_1$ and $\dot{t}_4 = \dot{t}_2$.

The goal reduces to $\dot{t}_1[\Lambda \overline{a}.\dot{t}_1'[\lambda x : \sigma_4.t_3]] \simeq \dot{t}_2[\Lambda \overline{a}.\dot{t}_2'[\lambda x : \sigma_4'.t_4]]$. By the definition of skolemisation, this further reduces to $\dot{t}_1[\Lambda \overline{a}.\Lambda \overline{a}_1.\lambda x : \sigma_4.t_3] \simeq \dot{t}_2[\Lambda \overline{a}.\Lambda \overline{a}_2.\lambda x : \sigma_4'.t_4]$. Finally, the goal follows by the induction hypothesis and compatibility Lemmas E.18, E.16 and E.21, along with transitivity Lemma E.15.

Case rule ETm-CheckTyAbs $e = \Lambda a.e'$ **:**

We know the premise of the typing derivation that $\sigma_1 = \forall\, \overline{\{a\}_1}.\forall\, a.\sigma_1'$, $\sigma_2 = \forall\, \overline{\{a\}}.\forall\, a.\sigma_2'$, $\Gamma_1, \overline{a}_1, a \vdash e' \Leftarrow \sigma_1' \rightsquigarrow t_1'$, $\Gamma_2, \overline{a}_2, a \vdash e' \Leftarrow \sigma_2' \rightsquigarrow t_2'$, $t_1 = \Lambda \overline{a}_1.\Lambda a.t_1'$ and $t_2 = \Lambda \overline{a}_2.\Lambda a.t_2'$. By case analysis on the type instantiation (rule InstT-SForall and rule InstT-SInfForall), we get $\Gamma' \vdash [\overline{\tau}_1/\overline{a}]\,[\overline{\tau}_1'/\overline{a}_1]\,[\tau_1/a]\,\sigma_1' \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_1'$ and $\Gamma' \vdash [\overline{\tau}_2/\overline{a}]\,[\overline{\tau}_2'/\overline{a}_2]\,[\tau_2/a]\,\sigma_2' \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_2'$ where $\dot{t}_1 = \lambda t.(\dot{t}_1'[t\,\overline{\tau}_1\,\overline{\tau}_1'\,\tau_1])$ and $\dot{t}_2 = \lambda t.(\dot{t}_2'[t\,\overline{\tau}_2\,\overline{\tau}_2'\,\tau_2])$.

The goal to be proven is $\dot{t}_1[\Lambda \overline{a}.\Lambda \overline{a}_1.\Lambda a.t_1'] \simeq \dot{t}_2[\Lambda \overline{a}.\Lambda \overline{a}_2.\Lambda a.t_2']$. This reduces to $\dot{t}_1'[(\Lambda \overline{a}.\Lambda \overline{a}_1.\Lambda a.t_1')\,\overline{\tau}_1\,\overline{\tau}_1'\,\tau_1] \simeq \dot{t}_2'[(\Lambda \overline{a}.\Lambda \overline{a}_2.\Lambda a.t_2')\,\overline{\tau}_2\,\overline{\tau}_2'\,\tau_2]$.

We now define a substitution $\theta = [\overline{\tau}_1/\overline{a}].[\overline{\tau}_2/\overline{a}].[\overline{\tau}_1'/\overline{a}_1].[\overline{\tau}_2'/\overline{a}_2].[\tau_1/a].[\tau_2/a]$. From the instantiation relation (and the fact that both types instantiate to the same type ρ, we conclude that if $[\tau_i/a] \in \theta$ and $[\tau_j/a] \in \theta$ that $\tau_i = \tau_j$. By applying Lemma E.49, we transform the premise to $[\overline{\tau}_1/\overline{a}]\,\Gamma_1 \vdash \theta\, e' \Leftarrow \theta\, \sigma_1' \rightsquigarrow \theta\, t_1'$ and $[\overline{\tau}_2/\overline{a}]\,\Gamma_2 \vdash \theta\, e' \Leftarrow \theta\, \sigma_2' \rightsquigarrow \theta\, t_2'$.

By applying the induction hypothesis, we get that $\dot{t}_1'[\theta\, t_1'] \simeq \dot{t}_2'[\theta\, t_2']$. The goal follows directly from the definition of θ.

Case rule ETm-CheckInf **:**

We know from the premise of the typing derivation that $\Gamma_1 \vdash \sigma_1 \xrightarrow{skol\ \delta} \rho_1; \Gamma_1' \rightsquigarrow \dot{t}_1'$, $\Gamma_2 \vdash \sigma_2 \xrightarrow{skol\ \delta} \rho_2; \Gamma_2' \rightsquigarrow \dot{t}_2'$, $\Gamma_1' \vdash e \Rightarrow \eta_1^\epsilon \rightsquigarrow t_1'$, $\Gamma_2' \vdash e \Rightarrow \eta_2^\epsilon \rightsquigarrow t_2'$, $\Gamma_1' \vdash \eta_1^\epsilon \xrightarrow{inst\ \delta} \rho_1 \rightsquigarrow \dot{t}_1''$, $\Gamma_2' \vdash \eta_2^\epsilon \xrightarrow{inst\ \delta} \rho_2 \rightsquigarrow \dot{t}_2''$, $t_1 = \dot{t}_1'[\dot{t}_1''[t_1']]$ and $t_2 = \dot{t}_2''[\dot{t}_2'[t_2']]$. The goal to be proven is thus $\dot{t}_1[\Lambda\overline{a}.\dot{t}_1'[\dot{t}_1''[t_1']]] \simeq \dot{t}_2[\Lambda\overline{a}.\dot{t}_2'[\dot{t}_2''[t_2']]]$.

From the definition of shallow skolemisation, we know that $\Gamma_1' = \Gamma_1, \overline{a}_1$, $\Gamma_2' = \Gamma_2, \overline{a}_2$, $\dot{t}_1' = \lambda t.\Lambda\overline{a}_1.t$ and $\dot{t}_2' = \lambda t.\Lambda\overline{a}_2.t$. We now take $\overline{a}' = \overline{a} \cup \overline{a}_1 \cup \overline{a}_2$. As σ_1 and σ_2 instantiate to the same type ρ, it is not hard to see from the definition of skolemisation that $\Gamma_1' \vdash \forall\,\overline{a}'.\eta_1^\epsilon \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_3$ and $\Gamma_2' \vdash \forall\,\overline{a}'.\eta_2^\epsilon \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_4$. By applying Lemma E.53, we thus get $\dot{t}_3[\Lambda\overline{a}'.t_1'] \simeq \dot{t}_4[\Lambda\overline{a}'.t_2']$. The goal follows through careful examination of the skolemisation and instantiation premises. □

Lemma E.56 (Pattern Checking Implies Synthesis).
If $\Gamma \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}$ *then* $\exists\overline{\psi} : \Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}$ *where type* $(\overline{\psi}; \sigma' \sim \sigma)$

The proof follows by straightforward induction on the pattern typing derivation.

We now go back to proving Property 6, and proceed by case analysis on both typing derivations (rule EDecl-Ann). We know from the premise that $\Gamma \vdash^P \overline{\pi_i} \Leftarrow \sigma_1 \Rightarrow \sigma_{i1}; \Delta_1 \rightsquigarrow \overline{\pi_{Fi}} : \overline{\psi_{F_1}}$, $\Gamma \vdash^P \overline{\pi_i} \Leftarrow \sigma_2 \Rightarrow \sigma_{i2}; \Delta_2 \rightsquigarrow \overline{\pi_{Fi}} : \overline{\psi_{F_2}}$, $\Gamma, \Delta_1 \vdash e_i \Leftarrow \sigma_{i1} \rightsquigarrow t_{i1}$, $\Gamma, \Delta_2 \vdash e_i \Leftarrow \sigma_{i2} \rightsquigarrow t_{i2}$, $t_1 = \overline{\textbf{case}\ \overline{\pi_{Fi}} : \overline{\psi_{F_1}} \rightarrow t_{i1}}^{\,i}$ and $t_2 = \overline{\textbf{case}\ \overline{\pi_{Fi}} : \overline{\psi_{F_2}} \rightarrow t_{i2}}^{\,i}$. The goal to be proven is $\dot{t}_1[\overline{\textbf{case}\ \overline{\pi_{Fi}} : \overline{\psi_{F_1}} \rightarrow t_{i1}}^{\,i}] \simeq \dot{t}_2[\overline{\textbf{case}\ \overline{\pi_{Fi}} : \overline{\psi_{F_2}} \rightarrow t_{i2}}^{\,i}]$. Lemma E.20 reduces this to $\forall i : \dot{t}_1[t_{i1}] \simeq \dot{t}_2[t_{i2}]$.

We take $\overline{a} = dom\,(\Delta_1) \cup dom\,(\Delta_2) \setminus dom\,(\Gamma)$, and apply weakening to get $\Gamma, \overline{a} \vdash e_i \Leftarrow \sigma_{i1} \rightsquigarrow t_{i1}$ and $\Gamma, \overline{a} \vdash e_i \Leftarrow \sigma_{i2} \rightsquigarrow t_{i2}$. The goal now follows directly from Lemma E.54 with $\overline{a} = \cdot$, if we can show that $\Gamma, \overline{a} \vdash \sigma_{i1} \xrightarrow{inst\ \delta} \rho' \rightsquigarrow \dot{t}_1$ and $\Gamma, \overline{a} \vdash \sigma_{i2} \xrightarrow{inst\ \delta} \rho' \rightsquigarrow \dot{t}_2$ for some ρ' (Note that Lemma E.54 only holds under shallow instantiation).

We know from Lemma E.56 that $\exists\overline{\psi_F} : \Gamma \vdash^P \overline{\pi_i} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_{Fi}} : \overline{\psi_F}$ such that $type\,(\overline{\psi}; \sigma_{i1} \sim \sigma_1)$ and $type\,(\overline{\psi}; \sigma_{i2} \sim \sigma_2)$. The remaining goal follows from the definition of the type relation, and shallow instantiation. □

E.5 Pattern Inlining and Extraction

Property 7 (Pattern Inlining is Type Preserving).
If $\Gamma \vdash x\,\overline{\pi} = e_1 \Rightarrow \Gamma'$ *and wrap* $(\overline{\pi}; e_1 \sim e_2)$ *then* $\Gamma \vdash x = e_2 \Rightarrow \Gamma'$

We first introduce a helper lemma to prove pattern inlining in expressions preserves the type:

Lemma E.57 (Pattern Inlining in Expressions is Type Preserving (Synthesis)).
If $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta$ *and* $\Gamma, \Delta \vdash e_1 \Rightarrow \eta_1^\epsilon$ *where wrap* $(\overline{\pi}; e_1 \sim e_2)$
then $\Gamma \vdash e_2 \Rightarrow \eta_2^\epsilon$ *and type* $(\overline{\psi}; \eta_1^\epsilon \sim \eta_2^\epsilon)$

The proof proceeds by induction on the pattern typing derivation. We will focus on the non-trivial cases below. Note that the rule Pat-InfCon is an impossible case as $wrap\,(K\,\overline{\pi}; e_1 \sim e_2)$ is undefined.

Case rule Pat-InfVar $\overline{\pi} = x, \overline{\pi}'$, $\overline{\psi} = \tau_1, \overline{\psi}'$ **and** $\Delta = x : \tau_1, \Delta'$ **:**

We know from the rule premise that $\Gamma, x : \tau_1 \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}'; \Delta'$. Furthermore, by inlining the definitions of Δ and $\overline{\pi}$ in the lemma premise, we get $\Gamma, x : \tau_1, \Delta' \vdash e_1 \Rightarrow \eta_1^\epsilon$ and $wrap\,(x, \overline{\pi}'; e_1 \sim \lambda x.e_2')$ and thus (by rule PatWrap-Var) $wrap\,(\overline{\pi}'; e_1 \sim e_2')$. By the induction hypothesis, we get $\Gamma, x : \tau_1 \vdash e_2' \Rightarrow \eta_3^\epsilon$ and $type\,(\overline{\psi}'; \eta_1^\epsilon \sim \eta_3^\epsilon)$. The goal follows by rule ETm-InfAbs and rule Type-Var.

Case rule Pat-InfTyVar $\overline{\pi} = @a, \overline{\pi}'$, $\overline{\psi} = @a, \overline{\psi}'$ **and** $\Delta = a, \Delta'$ **:**

We know from the rule premise that $\Gamma, a \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}'; \Delta'$. Again, by inlining the definitions in the lemma premise, we get $\Gamma, a, \Delta' \vdash e_1 \Rightarrow \eta_1^\epsilon$ and $wrap\,(@a, \overline{\pi}'; e_1 \sim \Lambda a.e_2')$ and thus (by rule PatWrap-TyVar) $wrap\,(\overline{\pi}'; e_1 \sim e_2')$. By the induction hypothesis, we get $\Gamma, a \vdash e_2' \Rightarrow \eta_3^\epsilon$ and $type\,(\overline{\psi}'; \eta_1^\epsilon \sim \eta_3^\epsilon)$.

The goal to be proven is $\Gamma \vdash \Lambda a.e_2' \Rightarrow \forall\, a.\eta_3^\epsilon$ where $type\,(@a, \overline{\psi}'; \eta_1^\epsilon \sim \forall\, a.\eta_3^\epsilon)$ (follows by rule Type-TyVar). However, under eager instantiation, this goal can never hold as rule ETm-InfTyAbs would instantiate the forall binder away. We can thus only prove this lemma under lazy instantiation, where the goal follows trivially from rule ETm-InfTyAbs. □

We now proceed with proving Property 7, through case analysis on the declaration typing relation (rule EDecl-NoAnnSingle). We know from the premise of the first derivation that $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta$, $\Gamma, \Delta \vdash e_1 \Rightarrow \eta_1^\epsilon$, $type\,(\overline{\psi}; \eta_1^\epsilon \sim \sigma)$ and $\Gamma' = \Gamma, x : \forall\, \overline{\{a\}}.\sigma$ where $\overline{a} = f_v(\sigma) \setminus dom\,(\Gamma)$. The goal to be proven thus becomes $\Gamma \vdash^P \cdot \Rightarrow \cdot; \cdot$ (follows directly from rule Pat-InfEmpty) and $\Gamma \vdash e_2 \Rightarrow \eta_2^\epsilon$ where $\eta_2^\epsilon = \sigma$ (follows from Lemma E.57). Note that as we require Lemma E.57, we can only prove Property 7 under lazy instantiation. □

Property 9 (Pattern Extraction is Type Preserving).
If $\Gamma \vdash x = e_2 \Rightarrow \Gamma'$ *and* $wrap\,(\overline{\pi}; e_1 \sim e_2)$ *then* $\Gamma \vdash x\,\overline{\pi} = e_1 \Rightarrow \Gamma'$

We first introduce another helper lemma to prove that pattern extraction from expressions preserves the typing:

Lemma E.58 (Pattern Extraction from Expressions is Type Preserving (Synthesis)).
If $\Gamma \vdash e_2 \Rightarrow \eta_2^\epsilon$ *and* $\exists\, e_1, \overline{\pi}$ *such that* $wrap\,(\overline{\pi}; e_1 \sim e_2)$
then $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta$ *and* $\Gamma, \Delta \vdash e_1 \Rightarrow \eta_1^\epsilon$ *where* $type\,(\overline{\psi}; \eta_1^\epsilon \sim \eta_2^\epsilon)$

The proof proceeds by induction on the e_2 typing derivation. As usual, we will focus on the non-trivial cases:

Case rule ETm-InfAbs $e_2 = \lambda x.e_2'$ **and** $\eta_2^\epsilon = \tau_2 \to \eta_3^\epsilon$ **:**

We know from the rule premise that $\Gamma, x : \tau_2 \vdash e_2' \Rightarrow \eta_3^\epsilon$. It is clear by case analysis on $wrap\,(\overline{\pi}; e_1 \sim \lambda x.e_2')$ that $\overline{\pi} = x, \overline{\pi}'$ and $wrap\,(\overline{\pi}'; e_1 \sim e_2')$. By applying the induction hypothesis, we get $\Gamma, x : \tau_2 \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}'; \Delta', \Gamma, x : \tau_2, \Delta' \vdash e_1 \Rightarrow \eta_1^\epsilon$ and $type\,(\overline{\psi}'; \eta_1^\epsilon \sim \eta_3^\epsilon)$. The goal thus follows straightforwardly by rule Pat-InfVar and rule Type-Var.

Case rule ETm-InfTyAbs $e_2 = \Lambda a.e_2'$ **:**

We know from the rule premise that $\Gamma, a \vdash e_2' \Rightarrow \eta_3^\epsilon$ and $\Gamma \vdash \forall\, a.\eta_3^\epsilon \xrightarrow{inst\ \delta} \eta_2^\epsilon$. Furthermore, it is clear by case analysis on $wrap\,(\overline{\pi}; e_1 \sim \Lambda a.e_2')$ that $\overline{\pi} = @a, \overline{\pi}'$ and $wrap\,(\overline{\pi}'; e_1 \sim e_2')$. By the induction hypothesis, we get $\Gamma, a \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}'; \Delta', \Gamma, a, \Delta' \vdash e_1 \Rightarrow \eta_1^\epsilon$ and $type\,(\overline{\psi}'; \eta_1^\epsilon \sim \eta_3^\epsilon)$.

The goal to be proven is $\Gamma \vdash^P @a, \overline{\pi}' \Rightarrow @a, \overline{\psi}'; a, \Delta'$ (follows by rule Pat-InfTyVar), $\Gamma, a, \Delta' \vdash e_1 \Rightarrow \eta_1^\epsilon$ (follows by the induction hypothesis) and $type\,(@a, \overline{\psi}'; \eta_1^\epsilon \sim \eta_2^\epsilon)$. However, it is clear that this final goal does not hold under eager instantiation, as rule ETm-InfTyAbs instantiates the forall binder away. Under lazy instantiation, the remaining goal follows directly from the premise.

Case rule ETm-InfApp $e_2 = h\,\overline{arg}$ **and** $\overline{arg} = \cdot$ **and** $h = e$ **:**

The goal follows directly by the induction hypothesis.

Case rule ETm-InfApp $e_2 = h\,\overline{arg}$ **and** $\overline{arg} \neq \cdot$ **or** $h \neq e$ **:**

It is clear from the definition of $wrap\,(\overline{\pi}; e_1 \sim h\,\overline{arg})$ that $\overline{\pi} = \cdot$. The goal thus follows trivially. □

We now return to prove Property 9 by case analysis on the declaration typing derivation (rule EDecl-NoAnnSingle). We know from the derivation premise that $\Gamma \vdash e_2 \Rightarrow \eta_2^\epsilon$ and $\sigma = \forall\, \overline{\{a\}}.\eta_2^\epsilon$ where $\overline{a} = f_v(\eta_2^\epsilon) \setminus dom\,(\Gamma)$. The goal follows directly from Lemma E.58. Note that as Lemma E.58 only holds under lazy instantiation, the same holds true for Property 9. □

Property 8 (Pattern Inlining / Extraction is Dynamic Semantics Preserving).
If $\Gamma \vdash x\,\overline{\pi} = e_1 \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t_1$, *wrap* $(\overline{\pi}; e_1 \sim e_2)$, *and* $\Gamma \vdash x = e_2 \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t_2$ *then* $t_1 \simeq t_2$

We start by introducing a helper lemma, proving pattern inlining preserves the dynamic semantics for expressions.

Lemma E.59 (Pattern Inlining in Expressions is Dynamic Semantics Preserving).
If $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}$ *and* $\Gamma, \Delta \vdash e_1 \Rightarrow \eta_1^\epsilon \rightsquigarrow t_1$ *and* $\Gamma \vdash e_2 \Rightarrow \eta_2^\epsilon \rightsquigarrow t_2$ *where wrap* $(\overline{\pi}; e_1 \sim e_2)$ *then* $\mathbf{case}\,\overline{\pi_F} : \overline{\psi_F} \to t_1 \simeq t_2$

The proof proceeds by induction on the pattern typing derivation. We will focus on the non-trivial cases. Note that, as *wrap* $(K\,\overline{\pi}; e_1 \sim e_2)$ is undefined, rule Pat-InfCon is an impossible case.

Case rule Pat-InfVar $\overline{\pi} = x, \overline{\pi'}$, $\overline{\psi} = \tau_1, \overline{\psi'}$, $\Delta = x : \tau_1, \Delta'$, $\overline{\pi_F} = x : \tau_1, \overline{\pi_F'}$ **and** $\overline{\psi_F} = \tau_1, \overline{\psi_F'}$ **:**
We know from the pattern typing derivation premise that $\Gamma, x : \tau_1 \vdash^P \overline{\pi'} \Rightarrow \overline{\psi'}; \Delta' \rightsquigarrow \overline{\pi_F'} : \overline{\psi_F'}$. By inlining the definitions and rule PatWrap-Var, we get $e_2 = \lambda x.e_2'$ and *wrap* $(\overline{\pi'}; e_1 \sim e_2')$. By case analysis on the e_2 typing derivation (rule ETm-InfAbs), we know $\Gamma, x : \tau_1 \vdash e_2' \Rightarrow \eta_3^\epsilon \rightsquigarrow t_2'$ where $\eta_2^\epsilon = \tau_1 \to \eta_3^\epsilon$ and $t_2 = \lambda x : \tau_1.t_2'$. By applying the induction hypothesis, we get $\mathbf{case}\,\overline{\pi_F'} : \overline{\psi_F'} \to t_1 \simeq t_2'$. The goal to be proven is $\lambda x : \tau_1.\mathbf{case}\,\overline{\pi_F'} : \overline{\psi_F'} \to t_1 = \lambda x : \tau_1.t_2'$, and follows directly from Lemma E.16.

Case rule Pat-InfTyVar $\overline{\pi} = @a, \overline{\pi'}$, $\overline{\psi} = @a, \overline{\psi'}$, $\Delta = a, \Delta'$, $\overline{\pi_F} = @a, \overline{\pi_F'}$ **and** $\overline{\psi_F} = @a, \overline{\psi_F'}$ **:**
We know from the pattern typing derivation premise that $\Gamma, a \vdash^P \overline{\pi'} \Rightarrow \overline{\psi'}; \Delta' \rightsquigarrow \overline{\pi_F'} : \overline{\psi_F'}$. Similarly to the previous case, by inlining and rule PatWrap-TyVar, we get $e_2 = \Lambda a.e_2'$ and *wrap* $(\overline{\pi'}; e_1 \sim e_2')$. By case analysis on the e_2 typing derivation (rule ETm-InfTyAbs), we get $\Gamma, a \vdash e_2' \Rightarrow \eta_3^\epsilon \rightsquigarrow t_2'$, $\Gamma \vdash \forall\, a.\eta_3^\epsilon \xrightarrow{inst\ \delta} \eta_2^\epsilon \rightsquigarrow \dot{t}$ and $t_2 = \dot{t}[\Lambda a.t_2']$. Applying the induction hypothesis tells us that $\mathbf{case}\,\overline{\pi_F'} : \overline{\psi_F'} \to t_1 \simeq t_2'$.

The goal to be proven is $\Lambda a.\mathbf{case}\,\overline{\pi_F'} : \overline{\psi_F'} \to t_1 \simeq \dot{t}[\Lambda a.t_2']$. By applying Lemma E.18 to the result of the induction hypothesis, we get $\Lambda a.\mathbf{case}\,\overline{\pi_F'} : \overline{\psi_F'} \to t_1 \simeq \Lambda a.t_2'$. Under lazy instantiation, the goal follows directly from this result, as $\dot{t} = \bullet$. Under eager deep instantiation, it is clear that the goal does not hold, as $\dot{t}$ might perform eta expansion, thus altering the dynamic semantics. Under eager shallow instantiation, the goal follows straightforwardly, as $\dot{t}$ can only perform type applications. Note that this implies that $\Lambda a.\mathbf{case}\,\overline{\pi_F'} : \overline{\psi_F'} \to t_1$ and $\dot{t}[\Lambda a.t_2']$ could thus have different types, but can always instantiate to the same type. □

We now return to proving Property 8, by case analysis on the first declaration typing relation (rule EDecl-NoAnnSingle). We know from the derivation premise that $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}$, $\Gamma, \Delta \vdash e_1 \Rightarrow \eta_1^\epsilon \rightsquigarrow t_1'$, $t_1 = \mathbf{case}\,\overline{\pi_F} : \overline{\psi_F} \to t_1'$, *type* $(\overline{\psi}; \eta_1^\epsilon \sim \sigma')$, $\sigma = \forall\,\overline{\{a\}}.\sigma'$ where $\overline{a} = f_v(\sigma') \setminus dom\,(\Gamma)$. The premise of the second declaration typing derivations tells us that $\Gamma \vdash e_2 \Rightarrow \eta_2^\epsilon \rightsquigarrow t_2$. The goal now follows directly from Lemma E.59. Note that as Lemma E.59 does not hold under eager deep instantiation, the same is true for Property 8. □

E.6 Single vs. Multiple Equations

Property 10 (Single/multiple Equations is Type Preserving).
If $\Gamma \vdash x\,\overline{\pi} = e \Rightarrow \Gamma, x : \sigma$ *then* $\Gamma \vdash x\,\overline{\pi} = e, x\,\overline{\pi} = e \Rightarrow \Gamma'$

The proof proceeds by case analysis on the declaration typing derivation (rule EDecl-NoAnnSingle). From the derivation premise, we get $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta, \Gamma, \Delta \vdash e \Rightarrow \eta^\epsilon$, *type* $(\overline{\psi}; \eta^\epsilon \sim \sigma_1)$ and $\sigma = \forall\,\overline{\{a\}}_1.\sigma_1$ where $\overline{a}_1 = f_v(\sigma_1) \setminus dom\,(\Gamma)$. The goal to be proven thus reduces to $\Gamma, \Delta \vdash \eta^\epsilon \xrightarrow{inst\ \delta} \rho$, *type* $(\overline{\psi}; \rho \sim \sigma_2)$ and $\sigma = \forall\,\overline{\{a\}}_2.\sigma_2$ where $\overline{a}_2 = f_v(\sigma_2) \setminus dom\,(\Gamma)$. It is clear that the property can not

$\boxed{numargs\,(\sigma) = m}$ *(Explicit Argument Counting)*

$$\textsc{Numargs-TyVar}\ \frac{}{numargs\,(a) = 0} \qquad \textsc{Numargs-Con}\ \frac{}{numargs\,(T\,\overline{\tau}) = 0} \qquad \textsc{Numargs-Arrow}\ \frac{numargs\,(\sigma_2) = m}{numargs\,(\sigma_1 \to \sigma_2) = m + 1}$$

$$\textsc{Numargs-Forall}\ \frac{numargs\,(\sigma) = m}{numargs\,(\forall\, a.\sigma) = m} \qquad \textsc{Numargs-InfForall}\ \frac{numargs\,(\sigma) = m}{numargs\,(\forall\, \{a\}.\sigma) = m}$$

Fig. 9. Counting Explicit Arguments

hold under lazy instantiation, as rule EDecl-NoAnnMulti performs an additional instantiation step, thus altering the type. Under eager instantiation, η^ϵ is already an instantiated type by the type inference relation, making the instantiation in the goal a no-op (by definition). The goal is thus trivially true. □

E.7 η-expansion

Property 11b (η-expansion is Type Preserving).

- *If* $\Gamma \vdash e \Rightarrow \eta^\epsilon$ *where* $numargs(\eta^\epsilon) = n$ *and* $\Gamma \vdash \eta^\epsilon \xrightarrow{inst\ \delta} \tau$ *then* $\Gamma \vdash \lambda\overline{x}^n.e\,\overline{x}^n \Rightarrow \eta^\epsilon$
- *If* $\Gamma \vdash e \Leftarrow \sigma$ *where* $numargs(\rho) = n$ *then* $\Gamma \vdash \lambda\overline{x}^n.e\,\overline{x}^n \Leftarrow \sigma$

A formal definition of *numargs* is shown in Figure 9. We prove Property 11b by first introducing a slightly more general lemma:

Lemma E.60 (η-expansion is Type Preserving - Generalised).

- *If* $\Gamma \vdash e \Rightarrow \eta^\epsilon$ *where* $0 \leqslant n \leqslant numargs(\eta^\epsilon)$ *and* $\Gamma \vdash \eta^\epsilon \xrightarrow{inst\ \delta} \tau$ *then* $\Gamma \vdash \lambda\overline{x}^n.e\,\overline{x}^n \Rightarrow \eta^\epsilon$
- *If* $\Gamma \vdash e \Leftarrow \sigma$ *where* $0 \leqslant n \leqslant numargs(\rho)$ *then* $\Gamma \vdash \lambda\overline{x}^n.e\,\overline{x}^n \Leftarrow \sigma$

The proof proceeds by induction on the integer n.

Case $n = 0$ **:**

This case is trivial, as it follows directly from the premise.

Case $n = m + 1 \leqslant numargs(\eta^\epsilon)$ **:**

case synthesis mode : We know from the induction hypothesis that $\Gamma \vdash \lambda\overline{x}^m.e\,\overline{x}^m \Rightarrow \eta^\epsilon$. We perform case analysis on this result (m repeated applications of rule ETm-InfAbs) to get $\Gamma, \overline{x_i : \tau_i}^{i<m} \vdash e\,\overline{x}^m \Rightarrow \eta_1^\epsilon$ where $\eta^\epsilon = \overline{\tau_i}^{i<m} \to \eta_1^\epsilon$. Performing case analysis again on this result (rule ETm-InfApp), gives us $\Gamma, \overline{x_i : \tau_i}^{i<m} \vdash^H e \Rightarrow \sigma_1$, $\Gamma, \overline{x_i : \tau_i}^{i<m} \vdash^A \overline{x}^m \Leftarrow \sigma_1 \Rightarrow \sigma_2$ and $\Gamma, \overline{x_i : \tau_i}^{i<m} \vdash \sigma_2 \xrightarrow{inst\ \delta} \eta_1^\epsilon$.

The goal to be proven is $\Gamma \vdash \lambda\overline{x}^{m+1}.e\,\overline{x}^{m+1} \Rightarrow \eta^\epsilon$, which (by rule ETmInfAbs) reduces to $\Gamma, \overline{x_i : \tau_i}^{i<m}, x : \tau \vdash e\,\overline{x}^{m+1} \Rightarrow \eta_2^\epsilon$, where $\eta^\epsilon = \overline{\tau_i}^{i<m} \to \tau \to \eta_2^\epsilon$.

Note that this requires proving that $\eta_1^\epsilon = \tau \to \eta_2^\epsilon$. While we know that $m < numargs(\eta^\epsilon)$, we can only prove this under eager deep instantiation. Under lazy instantiation, type inference does not instantiate the result type at all. Under eager shallow, it is instantiated, but only up to the first function type. From here on out, we will thus assume eager deep instantiation. Furthermore, note that as even deep instantiation does not instantiate argument types, we need the additional premise that η^ϵ instantiates into a monotype, in order to prove this goal.

This result in turn (by rule ETm-InfApp) reduces to $\Gamma, \overline{x_i : \tau_i}^{i<m}, x : \tau \vdash^H e \Rightarrow \sigma_1$ (follows by weakening), $\Gamma, \overline{x_i : \tau_i}^{i<m}, x : \tau \vdash^A x, \overline{x}^m \Leftarrow \sigma_1 \Rightarrow \sigma_3$ (follows by rule Arg-Inst, rule Arg-App

and the fact that $\eta_1^\epsilon = \tau \to \eta_2^\epsilon$) and $\Gamma, \overline{x_i : \tau_i}^{\,i<m}, x : \tau \vdash \sigma_3 \xrightarrow{inst\ \delta} \eta_2^\epsilon$ (follows by the definition of instantiation).

case checking mode : We know from the induction hypothesis that $\Gamma \vdash \lambda \overline{x}^m . e\, \overline{x}^m \Leftarrow \sigma$. The proof proceeds similarly to the synthesis mode case, by case analysis on this result (rule ETm-CheckAbs). One additional step is that rule ETm-CheckInf is applied to type $e\, \overline{x}^m$. The derivation switches to synthesis mode at this point, and becomes completely identical to the previous case. □

The proof for Property 11b now follows directly by Lemma E.60, by taking $n = numargs(\eta^\epsilon)$. □