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 and Bryn Mawr College, USA

It is often the case that designing a language feature requires an arbitrary choice between two possibilities, both of which are similarly expressive. Given that proper scientific user studies are generally impractical, we propose using *stability* as a way of making this decision. 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). Our analysis is applicable and relevant to any functional language that mixes implicit and explicit instantiation and generalisation.

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 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?

More generally than thinking only about type variable instantiation, we can consider the specification of implicit arguments. Type variables are just one form of implicit argument, but many languages support default arguments or other implicits. The questions we float above can become even more pressing when passing implicit arguments that are relevant at runtime. We use the term *instantiation* broadly to include both the selection of concrete types for type variables and also the supply of runtime-relevant implicit arguments.

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

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

Authors' addresses: Gert-Jan Bottu, Computer Science Department, KU Leuven, Belgium, gertjan.bottu@kuleuven.be; Richard A. Eisenberg, Tweag I/O, Paris, France, Computer Science Department, Bryn Mawr College, 101 N. Merion Ave. Bryn Mawr, PA, 19010, USA, rae@richarde.dev.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

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.

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 identify a novel language property: *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.

The concept of stability becomes even more relevant in the context of high-level languages with implicit features. If we imagine a fully explicit language—say, System F—stability is essentially guaranteed; the explicit nature of the language means that programmers say exactly what they mean. Even in the more implicit Hindley-Milner type system [Hindley 1969; Milner 1978], we have stability, via careful design of the type system. One could say that the success and ease of Hindley-Milner is precisely due to their design being stable. However, beyond the scope of Hindley-Milner, using a mix of implicit and explicit features (such as visible type instantiation [Eisenberg et al. 2016]), stability becomes an important and non-trivial concern. For this reason, this paper defines a concrete notion of stability, and then uses it to analyse a type system parameterised on the way it instantiates implicit arguments.

The main contribution of this work is fourfold:

- A discussion of stability properties for functional programming languages, and their interaction with type instantiation design choices. (Section 4)
- A type system generator, describing a basic Hindley-Milner-style type system, but with the ability to visibly instantiate type variables. It is parameterised over the type instantiation flavour. (Sections 5–6)
- 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 (introduced in Section 3) is the most stable. (Section 7)
- An application of our results to the concrete use case of GHC/Haskell. (Section 8)

We use Haskell both as a *lingua franca* throughout this paper, and also as the language of study in Section 8; though much of our analysis is easily applicable to other languages.

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

2 MIXING IMPLICIT AND EXPLICIT FEATURES IS UNSTABLE

The concept of stability is important in languages that have a mix of implicit and explicit features. This section walks through how a mixing of implicit and explicit features in Haskell[1], Idris[2], and Agda[3] 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 particular.

2.1 Explicit Instantiation

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.

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 $\mathbf{type}\ b \Leftarrow a = a \to b$ (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.

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.

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

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

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.

2.2 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 an example of this in Agda:

```
  -- ∋ allows a prefix type annotation
_∋_ : (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 appear to 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. 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 (λ{*A*} x_1 x_2 → *A* ∋ x_1), we can write (λ{*anything* = *A*} x_1 x_2 → *A* ∋ 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 seems "obvious" that we should accept a definition like *id* = λ @*a* (*x* :: *a*) → *a* without a type signature. It will turn out that, in the end, we can do this only when we instantiate lazily—see Section 7.

2.3 Implicit Generalisation

Our languages also all support 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.[4] Suppose we have defined $const\ x\ y = x$, without a signature. What type do we infer? It could be $\forall\ a\ b.\ a \to b \to a$ 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 $tr :: TypeRep\ (a :: k)$, where $TypeRep :: \forall\ k.\ k \to Type$ allows runtime type representation and is part of GHC's standard library. A naive left-to-right extraction of type variables would yield $\forall\ a\ k.\ TypeRep\ (a :: k)$, which is ill-scoped when we consider that a depends on k. Instead, we must reorder to $\forall\ k\ a.\ TypeRep\ (a :: k)$. 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 A 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? 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[5].

Agda. Recent versions of Agda support a new **variable** keyword[6]. 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 $A : Set$. 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

[4] 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 $f\ x = \textbf{let}\ g\ y = (y, x)\ \textbf{in}\ (g\ \texttt{'a'}, g\ True)$ is not generalised, because its body mentions the captured 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.

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

[6] 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.

order of quantification when Agda does generalisation. Often, this order is derived directly from the **variable** block—but not always. Consider this (contrived) declaration:

$$property : length\ l_2 + length\ l_1 \equiv length\ l_1 + length\ l_2$$

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 algorithm—not unlike 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.

2.4 Provenance

Because we use Haskell as our main object of study, we introduce Haskell's concept of variable *provenance*. This feature is 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.

Haskell's concept of provenance was invented solely to improve the stability of the language, although we are the first to describe it that way. As tends to happen in Haskell, the feature has attracted new uses, also to mitigate existing problems around stability; GHC now allows users to explicitly choose the provenance of a variable [Eisenberg 2017]. The classic illustrative example is this:

$$typeRep :: Typeable\ a \Rightarrow TypeRep\ a$$
$$typeRep = \ldots$$

The *typeRep* function allows users to produce a runtime type representation of a type a; the $Typeable\ a$ constraint asserts that this information is indeed available at runtime. Peyton Jones et al. [2016] describe the details.

Type representations are available for types of arbitrary kind. Accordingly, *typeRep* is kind-polymorphic, as are *Typeable* and *TypeRep*: its full type is $\forall\ k\ (a::k).\ Typeable$ @$k\ a \Rightarrow TypeRep$ @$k\ a$. However, making the kind polymorphism in *typeRep*'s type signature explicit would hurt clients, because it would change the provenance of k. In the signature for *typeRep* above, k is not mentioned; it is thus an inferred type variable. A client wanting the representation for *Int* can get it with *typeRep* @*Int*. However, writing, for example, $typeRep :: Typeable\ a \Rightarrow TypeRep\ (a :: k)$ both makes the kind polymorphism explicit (helpful for readers of documentation) and changes k's provenance to be specified. With this changed signature, clients would need to instantiate k before instantiating a, writing *typeRep* @*Type* @*Int* or *typeRep* @_ @*Int* to get the representation for *Int*. This degrades the user interface of *typeRep* considerably.

The solution comes from Eisenberg [2017]: allow users to explicitly mark variables as inferred (somewhat paradoxically). This proposal is implemented in GHC 8.12, and now we can write

typeRep :: ∀ {*k*} (*a* :: *k*). *Typeable* @*k* *a* ⇒ *TypeRep* @*k* *a*, both making the kind polymorphism explicit and keeping a tidy user interface.

The lesson here is that Haskell's design around provenance is unstable: by making the kind polymorphism in a type signature explicit (a seemingly innocuous, local change), that function's interface is also altered. We thus must add a new feature to Haskell—explicit provenance marking—in order to mitigate this instability.

2.5 Stability

Before diving into further details about type instantiation—the main technical exploration of this paper—let us reflect on our notion of stability. We have seen in this section how Haskell, Idris, and Agda all have a mix of explicit and implicit features around type instantiation and generalisation. 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.

Our goal here 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, novel 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 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 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, then (in Section 4) build up a set of program similarities we wish to respect, and finally (in Section 5) develop typing rules 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 will also showcase several examples where these choices make a real difference.

3.1 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 $f :: \forall\ a.\ a \to (\forall\ b.\ b \to b) \to \forall\ c.\ c \to c$. Instantiating f's type shallowly 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 8.12 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.

3.2 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 *myPair2*:

$$myPair2\ x = pair\ x$$

What type do we expect to infer for *myPair2*? 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 *myPair2*. 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 generalise this type (and take the parameter x into account) to produce the inferred type $myPair2 :: \forall\ a.\ a \to \forall\ b.\ b \to (a, b)$. 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, *myPair2* @*Bool* @*Char True* `'x'` is accepted, whereas with lazy instantiation, *myPair2* @*Bool True* @*Char* `'x'` is correct.

3.3 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.

Example 1: myId. 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/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. 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: myPair2. The *myPair2* example above acquires a new entanglement once we account for provenance. Recall that we define *myPair2* with this:

$$myPair2\ x = pair\ x$$

We infer these types:

myPair2	eager	lazy
deep/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: myId2. The end of Section 2.2 describes a mechanism for explicit binding of implicit variables. We might wish to define an identity function using this mechanism, thus:

$$myId2 = \lambda\ @a\ (x :: a) \to x$$

However, if we instantiate eagerly, the λ-expression gets instantiated right away, meaning that we must re-generalise to get a polymorphic type. We thus get the same types as *myId*, above:

myId2	eager	lazy
deep/shallow	$\forall\ \{a\}.\ a \to a$	$\forall\ a.\ a \to a$

Here, though, we encounter another complication: we might imagine using these inferred types to write a type signature in our program; an IDE might also do this automatically. If we have eager instantiation, though, adding the type signature would lead to trouble.

Imagine checking the expression $\lambda\ @a\ (x :: a) \to x$ against the type $\forall\ \{a\}.\ a \to a$. The quantified type variable a is marked as inferred, and so it is not available for explicit instantiation or abstraction—indeed, the *raison d'être* of inferred variables is that their presence or ordering cannot impact the typability of a program. The expression thus fails to type-check. This is very disappointing; the language implementation rejects the very type it has itself inferred.

Examples 5 and 6: boolId1 and boolId2. 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/shallow	$\forall\,\{a\}.\ \mathit{Bool} \to a \to a$	$\mathit{Bool} \to \forall\ a.\ a \to a$
boolId2	deep/shallow	$\forall\,\{a\}.\ \mathit{Bool} \to a \to a$	$\forall\,\{a\}.\ \mathit{Bool} \to a \to a$

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*. The result of a multiple-branch pattern-match may not be a polytype. To see why, consider a case where each branch is inferred to be well typed at *different* polytypes. Because each branch must result in the same type, we would have to take the different polytypes and partially instantiate them, seeking some most general type that is still less general than each branch's type. It is unclear whether, in the generality of Haskell, this is well defined. We thus have the restriction, unobtrusive in practice, that pattern-match branches must be monotypes.

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 7: swizzle. Suppose we have this function defined[7]:

$$\mathit{undef} :: \forall\ a.\ \mathit{Int} \to a$$
$$\mathit{undef} = \mathit{undefined}$$

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

$$\mathit{swizzle} :: \mathit{Int} \to \forall\ a.\ a \to a$$
$$\mathit{swizzle} = \mathit{undef}$$

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 $\mathit{swizzle} = \lambda(n :: \mathit{Int}) \to \Lambda(a :: \mathit{Type}) \to \mathit{undef}\ @a\ n$. 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 5), a necessary consequence of deep instantiation.

swizzle	eager/lazy
deep	converges
shallow	rejected

Example 8: eta. Consider these two definitions, where $\mathit{id} :: \forall\ a.\ a \to a$:

$$\mathit{noEta} = \mathit{id}$$
$$\mathit{eta} \quad = \lambda x \to \mathit{id}\ x$$

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

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

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

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.

3.4 Conclusion

The examples in this section show that the choice of instantiation scheme matters—and that no individual choice is clearly the best. How can we choose how to design our language? We use the notion stability to inform our decision, as we pin down in the next section.

4 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 specific similarities we will consider with respect to the instantiation flavours laid out in the previous section. There are naturally *many* transformations one might think of applying to a source program; the ones here relate to the examples in the previous section and seem to capture the challenges associated with implicit argument instantiation. We check our instantiation flavours against these similarities in Section 7.

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.

Examples 1–3 above all concern **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:

Similarity 1.

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

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 semantics of a program, as we hope programmers expect. This is illustrated by Example 4.

Similarity 2.

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

Reordering implicit variables should not affect dynamic semantics. This change does, naturally, affect static semantics, but assuming that both the old and new version are well typed, the runtime behaviour should be unaffected. This is illustrated by Example 7.

Similarity 3.

$$f : \sigma_1; \overline{f\ \overline{p}_i = e_i}^{\,i} \overset{D}{\Longleftrightarrow} f : \sigma_2; \overline{f\ \overline{p}_i = e_i}^{\,i},\ \textit{where } \sigma_1 \textit{ and } \sigma_2 \textit{ differ only in the ordering of implicit arguments}$$

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 $\overline{p}$ only contains (expression and type) variable patterns for this similarity. The operator *wrap* is unsurprising, and just wraps the patterns around the expression in lambda binders. Its definition can be found in Appendix B.

$$\begin{array}{rcll} \delta & ::= & \mathcal{S} \mid \mathcal{D} & \textit{Depth} \\ \epsilon & ::= & \mathcal{E} \mid \mathcal{L} & \textit{Eagerness} \end{array}$$

$$\begin{array}{rcll} \tau & ::= & a \mid \tau_1 \to \tau_2 \mid T\,\overline{\tau} & \textit{Monotype} \\ \rho & ::= & \tau \mid \sigma_1 \to \phi_2^{\delta} & \textit{Instantiated type} \\ \sigma & ::= & \rho \mid \forall\, \overline{a}.\sigma \mid \sigma_1 \to \sigma_2 & \textit{Type scheme} \\ \phi^{\delta} & ::= & \rho \quad (\delta = \mathcal{D}) & \textit{Instantiated result} \\ & \mid & \sigma \quad (\delta = \mathcal{S}) & \\ \eta^{\epsilon} & ::= & \rho \quad (\epsilon = \mathcal{E}) & \textit{Synthesised type} \\ & \mid & \sigma \quad (\epsilon = \mathcal{L}) & \\ \psi & ::= & \tau & \textit{Partial type} \end{array}$$

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

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

Similarity 4.

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

Following Examples 5 and 6, we would want the addition of new equations in a function definition not to affect types:

Similarity 5.

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

And lastly, inspired by Example 8, we want η-expansion not to affect types either. (This change *can* reasonably affect 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 1 (Stability). *A language is considered stable when all of the program similarities above are respected.*

Having laid down our goal—stability—we now build up formal model of our instantiation flavours to see how we can achieve that goal.

5 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 develops 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 is 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 type system *generator*: 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 **l**avender. Similarly, instantiation under the **s**hallow scheme is drawn using a **s**triped line, as in $\Gamma \vdash \phi^{\delta} \xrightarrow{inst\ \mathcal{S}} \rho$.

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 $T\,\overline{\tau}$[8]. 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 they 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 $\Gamma \vdash e \Rightarrow \eta$, 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 must be instantiated types ρ: any top-level quantified variable would have been instantiated away.

Finally, an argument type ψ represents a type synthesised from analysing a function argument pattern. In the IPLC, these are just monotypes, but we will see this change in the next section. Argument types are assembled into type schemes σ with the $type\,(\overline{\psi}; \sigma_0) = \sigma$ 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 declaration *decl*, which may optionally have a type signature $x : \sigma$, followed by the definition $\overline{x\,\overline{p}_i = e_i}^{\,i}$. The patterns $\overline{p}$ on the left of the equals sign each be either a simple variable x or a saturated data constructor $K\,\overline{p}$.

Contexts. Typing contexts Γ are entirely standard, storing both the term variables x with their type 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 and the 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

[8]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)*

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

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

$$\textsc{Tm-InfApp}\ \frac{\Gamma \vdash e_1 \Rightarrow \eta^\epsilon \qquad \Gamma \vdash \eta^\epsilon \xrightarrow{inst\ \delta} \sigma_1 \to \phi_2^\delta \qquad \Gamma \vdash \sigma_1 \xrightarrow{skol\ \delta} \rho_1; \Gamma' \qquad \Gamma' \vdash e_2 \Leftarrow \rho_1 \qquad \Gamma \vdash \phi_2^\delta \overset{inst\ \mathcal{S}}{\dashrightarrow} \eta_2^\epsilon}{\Gamma \vdash e_1\ e_2 \Rightarrow \eta_2^\epsilon}$$

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

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

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

$$\textsc{Tm-CheckAbs}\ \frac{\Gamma \vdash \phi_2^\delta \overset{skol\ \mathcal{S}}{\dashrightarrow} \rho_2; \Gamma' \qquad \Gamma', x : \sigma_1 \vdash e \Leftarrow \rho_2}{\Gamma \vdash \lambda x.e \Leftarrow \sigma_1 \to \phi_2^\delta} \qquad \textsc{Tm-CheckLet}\ \frac{\Gamma \vdash decl \Rightarrow \Gamma' \qquad \Gamma' \vdash e \Leftarrow \rho}{\Gamma \vdash let\ decl\ in\ e \Leftarrow \rho}$$

$$\textsc{Tm-CheckInf}\ \frac{\Gamma \vdash e \Rightarrow \eta^\epsilon \qquad \Gamma \vdash \eta^\epsilon \xrightarrow{inst\ \delta} \rho \qquad \text{No other rules match}}{\Gamma \vdash e \Leftarrow \rho}$$

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

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

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

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

$$\textsc{Decl-Ann}\ \frac{\overline{\Gamma \vdash^P \overline{p}_i \Leftarrow \sigma \Rightarrow \sigma_i'; \Delta}^{\,i} \qquad \overline{\overline{\Gamma, \Delta \vdash \sigma_i' \xrightarrow{skol\ \delta} \rho; \Gamma_i'}^{\,i} \qquad \overline{\Gamma_i' \vdash e_i \Leftarrow \rho}^{\,i}}{\Gamma \vdash x : \sigma; \overline{x\,\overline{p}_i = e_i}^{\,i} \Rightarrow \Gamma, x : \sigma}$$

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

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.

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

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

$$\textsc{Pat-InfCon}\ \frac{\begin{array}{c} K : \overline{a}_0; \overline{\sigma}_0; T \in \Gamma \\ \mathit{binders}^{\delta}(\forall\, \overline{a}_0.\overline{\sigma}_0 \to T\, \overline{a}_0) = \overline{a}; \rho_0 \\ \Gamma \vdash^P \overline{p} \Leftarrow [\overline{\tau}/\overline{a}]\, \rho_0 \Rightarrow T\, \overline{\tau}; \Delta_1 \\ \Gamma, \Delta_1 \vdash^P \overline{p}' \Rightarrow \overline{\psi}; \Delta_2 \end{array}}{\Gamma \vdash^P (K\, \overline{p}), \overline{p}' \Rightarrow T\, \overline{\tau}, \overline{\psi}; \Delta_1, \Delta_2}$$

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

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

$$\textsc{Pat-CheckCon}\ \frac{\begin{array}{c} K : \overline{a}_0; \overline{\sigma}_0; T \in \Gamma \\ \mathit{binders}^{\delta}(\forall\, \overline{a}_0.\overline{\sigma}_0 \to T\, \overline{a}_0) = \overline{a}; \rho_0 \\ \Gamma \vdash \sigma_1 \xrightarrow{\mathit{skol}\ \delta} \rho_1; \Gamma_1 \\ \Gamma_1 \vdash^P \overline{p} \Leftarrow [\overline{\tau}/\overline{a}]\, \rho_0 \Rightarrow \rho_1; \Delta_1 \\ \Gamma, \Delta_1 \vdash^P \overline{p}' \Leftarrow \sigma_2 \Rightarrow \sigma_2'; \Delta_2 \end{array}}{\Gamma \vdash^P (K\, \overline{p}), \overline{p}' \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma_2'; \Delta_1, \Delta_2}$$

$$\textsc{Pat-CheckForall}\ \frac{\begin{array}{c} \Gamma, a \vdash^P \overline{p} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \\ \text{No other rules match} \end{array}}{\Gamma \vdash^P \overline{p} \Leftarrow \forall\, a.\sigma \Rightarrow \sigma'; a, \Delta}$$

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

5.1 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 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.

Type Checking. Checking is performed by the $\Gamma \vdash e \Leftarrow \rho$ relation: "Under typing context Γ, the expression e is checked to have the type ρ". The type serves as an input here: an algorithmic interpretation of this relation would simply return a pass/fail result.

Type Instantiation. The behaviour of both the instantiation $\Gamma \vdash \sigma \xrightarrow{\mathit{inst}\ \delta} \rho$ and skolemisation $\Gamma \vdash \sigma \xrightarrow{\mathit{skol}\ \delta} \rho; \Delta$ judgements depends on whether the depth δ is deep $\mathcal{D}$ or shallow $\mathcal{S}$; see Figure 5. The relations gathers 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 (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

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

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

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

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

Fig. 5. Type Instantiation and Skolemisation

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; 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 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 mono-type τ to the variable x when checking the remainder of the expression e. Type checking for

abstractions (rule Tm-CheckAbs) works similarly, but also allows a full type scheme as argument type. The provided result type is skolemised before checking the remainder of the expression.

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. However, the checking judgement $\Gamma \vdash e \Leftarrow \rho$ expects a ρ-type; we thus must *skolemise* σ_1 to get ρ_1. Skolemisation takes the available binders of σ_1 and simply adds them to the context Γ'. These variables behave as skolem constants when checking e_2. 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 skolemising the given type σ and checking the expression e under this skolemised type ρ. Note that under eager instantiation, the type for this annotated expression is not the type scheme σ, but an instantiated form of this type: this is the last premise to rule Tm-InfAnn.

Going from checking to synthesis mode is also allowed (rule Tm-CheckInf). This rule applies only when other type checking rules do not. Also note that under lazy instantiation, the synthesised type still needs to be instantiated in order to match the given type 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). The critical difference between the single-equation case and the multiple-equation case is that the right-hand side type must be a monotype τ; the single-equation case does not have this restriction. Indeed, this difference is the only reason we have separated out the single-equation case, to be able to define synonyms of variables with higher-rank or non-prenex types.

Rule Decl-Ann checks a declaration with a type signature. It works by first checking the patterns $\overline{p}_i$ on the left of the equals sign against the provided type σ. The right-hand sides e_i are then checked against the (skolemised) remaining type σ_i'. Note that we require this skolemised type ρ to be identical for each of the equations. The pattern type checking relation $\Gamma \vdash^P \overline{p} \Leftarrow \sigma \Rightarrow \sigma'; \Delta$ (see Figure 4) checks the list of patters $\overline{p}$ under type scheme σ, and produces the remaining type σ' and the typing context extension Δ.

Patterns. 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. Checking a data constructor pattern (rule Pat-CheckCon) is more noteworthy. The type $\forall\, \overline{a}_0.\overline{\sigma}_0 \to \tau$ of the constructor K is looked up in the typing context. Then, the patterns $\overline{p}$ are checked under an instantiated form of $\forall\, \overline{a}_0.\overline{\sigma}_0 \to \tau$. The remaining type should then be equal to the (skolemised) provided type σ_1. Finally, the remaining patterns $\overline{p}'$ are checked under the extended context[9].

The pattern synthesis relation $\Gamma \vdash^P \overline{p} \Rightarrow \overline{\psi}; \Delta$ is presented in Figure 4. As the full type is not yet available, it produces argument types $\overline{\psi}$ 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 works by looking up the type of the constructor σ_0 in the typing context, and checking the applied patterns $\overline{p}$ under this instantiated type. 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.

5.2 Alternative Formalisation

Section 6 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. For this reason, we present an alternative formalisation of our IPLC typing rules, which 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 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 6 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'[10]. Finally, after the head has been applied to all its arguments, the remaining type σ' is instantiated (under eager instantiation), producing a η^ϵ type 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 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

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

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

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

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

$$\textsc{H-Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash^H x \Rightarrow \sigma} \qquad \textsc{H-Con}\ \frac{K : \overline{a}; \overline{\sigma}; T \in \Gamma}{\Gamma \vdash^H K \Rightarrow \forall\, \overline{a}.\overline{\sigma} \to T\,\overline{a}} \qquad \textsc{H-Ann}\ \frac{\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho; \Gamma' \quad \Gamma' \vdash e \Leftarrow \rho}{\Gamma \vdash^H e : \sigma \Rightarrow \sigma} \qquad \textsc{H-Inf}\ \frac{\Gamma \vdash e \Rightarrow \sigma}{\Gamma \vdash^H e \Rightarrow \sigma}$$

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

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

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

$$\textsc{ETm-CheckAbs}\ \frac{\Gamma \vdash \sigma_2 \xrightarrow{skol\ \mathcal{S}} \rho_2; \Gamma_2 \quad \Gamma_2, x : \sigma_1 \vdash e \Leftarrow \rho_2}{\Gamma \vdash \lambda x.e \Leftarrow \sigma_1 \to \sigma_2} \qquad \textsc{ETm-CheckLet}\ \frac{\Gamma \vdash decl \Rightarrow \Gamma' \quad \Gamma' \vdash e \Leftarrow \rho}{\Gamma \vdash \textit{let decl in } e \Leftarrow \rho}$$

$$\textsc{ETm-CheckInf}\ \frac{\Gamma \vdash e \Rightarrow \eta^\epsilon \quad \Gamma \vdash \eta^\epsilon \xrightarrow{inst\ \delta} \rho \quad \text{No other rules match}}{\Gamma \vdash e \Leftarrow \rho}$$

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

$$\textsc{Arg-Empty}\ \frac{}{\Gamma \vdash^A \cdot \Leftarrow \sigma \Rightarrow \sigma} \qquad \textsc{Arg-App}\ \frac{\Gamma \vdash \sigma_1 \xrightarrow{skol\ \delta} \rho_1; \Gamma' \quad \Gamma' \vdash e \Leftarrow \rho_1 \quad \Gamma \vdash^A \overline{arg} \Leftarrow \sigma_2 \Rightarrow \sigma'}{\Gamma \vdash^A e, \overline{arg} \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma'} \qquad \textsc{Arg-Inst}\ \frac{\Gamma \vdash^A e, \overline{arg} \Leftarrow [\tau_1/a]\,\sigma_2 \Rightarrow \sigma_3}{\Gamma \vdash^A e, \overline{arg} \Leftarrow \forall\, a.\sigma_2 \Rightarrow \sigma_3}$$

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

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.

6 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 extension over the IPLC includes 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 $@\tau$. Type schemes σ now allow inferred variables $\forall\, \overline{\{a\}}.\sigma$. This is also reflected in the updated typing context

$$\begin{array}{lcll} \sigma & ::= & \ldots \mid \forall\, \overline{\{a\}}.\sigma & \textit{Type scheme} \\ \psi & ::= & \ldots \mid @a & \textit{Argument type} \\ \Gamma & ::= & \ldots \mid \Gamma, \{a\} & \textit{Context} \end{array} \qquad \begin{array}{lcll} e & ::= & \ldots \mid \Lambda a.e & \textit{Expression} \\ p & ::= & \ldots \mid @\tau & \textit{Pattern} \\ arg & ::= & \ldots \mid @\tau & \textit{Application argument} \end{array}$$

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

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

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

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

ETM-CHECKTYABS
$$\frac{a \in \Gamma \qquad \Gamma \vdash e \Leftarrow \rho}{\Gamma \vdash \Lambda a.e \Leftarrow \rho}$$

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

PAT-INFTYVAR
$$\frac{\Gamma, a \vdash^P \overline{p} \Rightarrow \overline{\psi}; \Delta}{\Gamma \vdash^P @a, \overline{p} \Rightarrow @a, \overline{\psi}; a, \Delta}$$

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

TYPE-TYVAR
$$\frac{type\,(\overline{\psi}; \sigma) = \sigma'}{type\,(@a, \overline{\psi}; \sigma) = \forall\, a.\sigma'}$$

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

PAT-CHECKTYPE
$$\frac{\overline{a} = f_v(\tau) \qquad \Gamma, \overline{a} \vdash^P \overline{p} \Leftarrow [\tau/b]\,\sigma \Rightarrow \sigma'; \Delta}{\Gamma \vdash^P @\tau, \overline{p} \Leftarrow \forall\, b.\sigma \Rightarrow \sigma'; \overline{a}, \Delta}$$

PAT-CHECKINFFORALL
$$\frac{\Gamma, \{a\} \vdash^P \overline{p} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \qquad \text{No other rules match}}{\Gamma \vdash^P \overline{p} \Leftarrow \forall\, \{a\}.\sigma \Rightarrow \sigma'; \{a\}, \Delta}$$

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

grammar $\Gamma, \{a\}$. 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 B and C, 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 5.2 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 2.4), handled in rule ARG-INFINST, are unconditionally instantiated.

Explicit Type Abstraction. The definition of patterns p 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 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 rule ETm-CheckTyAbs) requires that the forall type has been skolemised, and recursively checks the expression e.

Type abstraction $@a$ in synthesis mode (rule Pat-InfTyVar) produces a partial forall type and extends the typing environment. When in checking mode (rule Pat-CheckType), full monotypes are allowed in patterns. Consider, for example, $f\ (\mathbf{Just}\ @\mathbf{Int}\ x) = x + 1$, where the **@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-CheckForall), but only when no other rules match in order not to lose syntax-directedness of the rules.

7 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 ten properties, based on the informal definition of stability from Section 4. Every property is analysed against the four instantiation flavours, the results of which are shown in Table 1. Clarifying examples are provided. A more formal reasoning behind the properties is provided in Appendix D.

7.1 Contextual Equivalence

Instead of providing a dynamic operational semantics of our type system, we instead define an elaboration of the surface language presented in this paper to an explicit variant of System F, which we call our core language. A separate core language allows us to understand the behaviour of Example 7, *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 C. The meta variable t refers to core terms, and $\rightsquigarrow$ denotes elaboration. We can now define contextual equivalence in order to describe what it means for dynamic semantics to remain unchanged.

Definition 2 (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; the hole is filled by t_1 or t_2.

7.2 Properties

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

Property 1 (Let Inlining is Type Preserving).

- $If\ \Gamma \vdash let\ x = e_1\ in\ e_2 \Rightarrow \eta^\epsilon\ then\ \Gamma \vdash [e_1/x]\ e_2 \Rightarrow \eta^\epsilon$
- $If\ \Gamma \vdash let\ x = e_1\ in\ e_2 \Leftarrow \rho\ then\ \Gamma \vdash [e_1/x]\ e_2 \Leftarrow \rho$

Property 2 (Let Inlining is Dynamic Semantics Preserving).

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

Property 3 (Let Extraction is Type Preserving).

- $If\ \Gamma \vdash [e_1/x]\ e_2 \Rightarrow \eta^\epsilon\ then\ \Gamma \vdash let\ x = e_1\ in\ e_2 \Rightarrow \eta^\epsilon$
- $If\ \Gamma \vdash [e_1/x]\ e_2 \Leftarrow \rho\ then\ \Gamma \vdash let\ x = e_1\ in\ e_2 \Leftarrow \rho$

Property 4 (Let Extraction is Dynamic Semantics Preserving).

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

- *If* $\Gamma \vdash [e_1/x]\, e_2 \Leftarrow \rho \rightsquigarrow t_1$ *and* $\Gamma \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \Leftarrow \rho \rightsquigarrow t_2$ *then* $t_1 \simeq t_2$

As an example for why Property 3 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.

The dynamic semantics properties (both these and later ones) struggle under deep instantiation. This is demonstrated by Example 7, *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 5 (Signature Property is Type Preserving).
If $\Gamma \vdash \overline{x\,\overline{p}_i = e_i}^{\,i} \Rightarrow \Gamma'$ *and* $x : \sigma \in \Gamma'$ *then* $\Gamma \vdash x : \sigma; \overline{x\,\overline{p}_i = e_i}^{\,i} \Rightarrow \Gamma'$

As an example of how this goes wrong under eager instantiation, consider the definition $x = \lambda\ @a\ (y :: a) \to y$. Annotating x with its inferred type $\forall\ \{a\}.\ a \to a$ is rejected, because of the $a \in \Gamma$ premise to rule ETm-CheckTyAbs. That premise makes sure that a has been added to the context by skolemising a *specified* type variable, not an inferred one.

Property 6 (Signature Property is Dynamic Semantics Preserving).
If $\Gamma \vdash \overline{x\,\overline{p}_i = e_i}^{\,i} \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t_1$ *and* $\Gamma \vdash x : \sigma; \overline{x\,\overline{p}_i = e_i}^{\,i} \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t_2$ *then* $t_1 \simeq t_2$

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

Property 7 (Type Signatures are Dynamic Semantics Preserving).
If $\Gamma \vdash x : \sigma_1; \overline{x\,\overline{p}_i = e_i}^{\,i} \Rightarrow \Gamma_1 \rightsquigarrow x : \sigma_1 = t_1$ *and* $\Gamma \vdash x : \sigma_2; \overline{x\,\overline{p}_i = e_i}^{\,i} \Rightarrow \Gamma_1 \rightsquigarrow x : \sigma_2 = t_2$ *then* $t_1 \simeq t_2$

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

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

Property 8 (Pattern Inlining is Type Preserving).
If $\Gamma \vdash x\,\overline{p} = e_1 \Rightarrow \Gamma'$ *and* $\mathit{wrap}\,(\overline{p}; e_1) = e_2$ *then* $\Gamma \vdash x = e_2 \Rightarrow \Gamma'$

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

Property 9 (Pattern Inlining / Extraction is Dynamic Semantics Preserving).
If $\Gamma \vdash x\,\overline{p} = e_1 \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t_1$, $\mathit{wrap}\,(\overline{p}; e_1) = e_2$, *and* $\Gamma \vdash x = e_2 \Rightarrow \Gamma' \rightsquigarrow x : \sigma = t_2$ *then* $t_1 \simeq t_2$

Property 10 (Pattern Extraction is Type Preserving).
If $\Gamma \vdash x = e_2 \Rightarrow \Gamma'$ *and* $\mathit{wrap}\,(\overline{p}; e_1) = e_2$ *then* $\Gamma \vdash x\,\overline{p} = e_1 \Rightarrow \Gamma'$

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 this property:

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

Table 1. Property Overview

Property 11 (Single/multiple Equations is Type Preserving).
If $\Gamma \vdash x\,\overline{p} = e \Rightarrow \Gamma'$ *then* $\Gamma \vdash x\,\overline{p} = e, x\,\overline{p} = e \Rightarrow \Gamma'$

This property surprisingly favours the otherwise-unloved eager/deep flavour. Imagine $f\ \_ = \mathit{pair}$. This definition is accepted, via rule Decl-NoAnnSingle. Yet if we simply duplicate this equation (as a stand-in for differing some patterns on the left-hand side), then rule Decl-NoAnnMulti will reject—except if the right-hand side is deeply instantiated[11] and thus becomes a monotype.

η-expansion. Similarity 6 leads to this property:

Property 12 (η-expansion is Type Preserving).

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

Here, we use $\mathit{numargs}(\sigma)$ to count the number of explicit arguments an expression can take, possibly instantiating any intervening implicit arguments. This property fails for all but the eager/deep flavour: η-expansion forces other flavours to instantiate arguments they otherwise would not have.

8 INSTANTIATION IN GHC

We started this analysis with one goal: to figure out how GHC should instantiate. This section relates the work we have presented in this paper to the real-life setting of GHC.

8.1 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 instantiation, it has used lazy instantiation, as advocated by Eisenberg et al. [2016]. Yet there is trouble supporting universal lazy instantiation:

[11] In the final days of preparation of this paper, GHC bug #18412 was submitted, and a fix supplied. This effectively changes rule Decl-NoAnnMulti to allow polytypes. Under this change, all four flavours would respect this similarity, but there was no time to update the typing rules in this paper correspondingly.

- 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.
- Lazy instantiation makes the monomorphism restriction easier to implement, because relevant constraints are instantiated.
- For simplicity, we want all variables without type signatures not to work with explicit type instantiation. 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.

Because of this trouble, GHC implemented an uneasy truce: instantiation was lazy, but if GHC was about to generalise a type, it instantiated it first. This new "before generalisation" instantiation plan addresses the bullets above, but it has caused extra twists and turns in the implementation, and causes greater instability.

8.2 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 8.12, changed to use shallow subsumption and shallow instantiation.

8.3 The situation today

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 (due for release with GHC 8.12) switched to use exclusively eager instantiation. This choice sacrifices stability for convenience in implementation.

Is this paper an argument for changing to lazy instantiation? Not unreservedly. We attack this design problem from the viewpoint of stability. There are other concerns, however, including ease of implementation. While it would not seem hard to support lazy instantiation, there would be compromises. For example, there might need to be tweaks to the definition of the monomorphism restriction[12]. Instead, we view this paper as a careful study of the stability aspects of the choices around instantiation, so that implementations such as GHC can make an informed decision.

9 RELATED WORK

Our work builds on the development of explicit control of polymorphism, as described in works cited throughout the paper. However, 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

[12] A definition like *plus* = (+) (Example 7 of Section 3.3) is subject to the monomorphism restriction (MR), yet lazy instantiation means it would be hard to apply the MR in this case. However, the MR was not designed with definitions like *plus* in mind—instead, it was intended for definitions like *five* = 2 + 3, in order to avoid the need for recomputation at every use site of *five*. Yet, even with lazy instantiation, the MR could easily apply to *five*, as the usage of (+) there is instantiated anyway. So our change would be not to apply the MR in situations where the body of the definition is uninstantiated—in other words, exactly where the body is already a function. There is no risk of repeated computation in such cases, and so we think this change to the MR would likely be beneficial regardless of stability or other concerns.

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 related work beyond what we have already cited throughout this paper.

10 CONCLUSION

This work introduces the concept of *stability*, as a way of evaluating the usability of type instantiation design decisions. 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 type system generator, parameterised over the instantiation flavour, and featuring a mix of explicit and implicit behaviour. Using this generator, 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 (ICFP '05)*. ACM.

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

Joshua 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. 2017. Explicit specificity in type variable binders. GHC Proposal #99. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0099-explicit-specificity.rst

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 (POPL '14)*. ACM.

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

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

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

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

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).

Simon Peyton Jones, Stephanie Weirich, Richard A. Eisenberg, and Dimitrios Vytiniotis. 2016. A reflection on types. In *A list of successes that can change the world*. Springer. A festschrift in honor of Phil Wadler.

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.

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A quick look at impredicativity. In *International Conference on Functional Programming (ICFP'20)*. ACM, ACM. https://www.microsoft.com/en-us/research/publication/a-quick-look-at-impredicativity/

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 (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 (ICFP '13)*. ACM.

A 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.

B ADDITIONAL RELATIONS

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

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

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

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

$\boxed{wrap\,(\overline{p}; e_1) = e_2}$ *(Pattern Wrapping)*

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

C CORE LANGUAGE

The dynamic semantics of the languages in Sections 5 and 6 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 $\overline{\lambda case\, \overline{p_F}_i : \overline{\psi_F} \to t_i}^{\,i}$, where patterns p_F include both term and type variables, as well as nested constructor patterns. Furthermore, in order to simplify the translations, expression wrappers $\dot{t}$ – essentially expressions

with a hole $\bullet$ in them – are defined. 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.

$$
\begin{array}{lcll}
t & ::= & x \mid K \mid t_1\ t_2 \mid \lambda x : \sigma.t \mid t\ \sigma \mid \Lambda a.t & \textit{Expression} \\
 & \mid & \lambda case\, \overline{\overline{p_F}_i : \overline{\psi_F} \to t_i}^{\,i} \mid \dot{t}[t] & \\
\dot{t} & ::= & \bullet \mid \lambda x : \sigma.\dot{t} \mid \dot{t}\,\tau \mid \Lambda a.\dot{t} \mid \lambda t_1.t_2 & \textit{Expression Wrapper} \\
arg_F & ::= & t \mid \sigma & \textit{Argument} \\
p_F & ::= & x : \sigma \mid @a \mid K\,\overline{p_F} & \textit{Pattern} \\
\psi_F & ::= & \sigma \mid @a & \textit{Partial Type Scheme}
\end{array}
$$

The type erasing evaluation for our System F target language is defined below.

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

FMatch-Var

$$\frac{}{match\,(x : \sigma, \overline{p_F} \to t_1; t_2 : \sigma) \hookrightarrow \overline{p_F} \to [t_2/x]t_1}$$

FMatch-Con

$$\frac{\begin{array}{c}\sigma_2 = \overline{\psi_F}_1 \to \tau_2 \\ t_2 \hookrightarrow^* K\,\overline{t} \\ (\lambda case\, \overline{\overline{p_F}_1 : \overline{\psi_F}_1 \to t_1})\,\overline{t} \hookrightarrow^* t_1'\end{array}}{match\,((K\,\overline{\overline{p_F}_1}), \overline{p_F}_2 \to t_1; t_2 : \sigma_2) \hookrightarrow \overline{p_F}_2 \to t_1'}$$

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

FEval-App

$$\frac{t_1 \hookrightarrow t_1'}{t_1\ t_2 \hookrightarrow t_1'\ t_2}$$

FEval-AppAbs

$$\frac{}{(\lambda x : \sigma.t_1)\ t_2 \hookrightarrow [t_2/x]t_1}$$

FEval-CaseEmpty

$$\frac{}{\lambda case\, \overline{\cdot : \cdot \to t_i}^{\,i} \hookrightarrow t_0}$$

FEval-CaseMatch

$$\frac{\forall\, j \in v\ where\ (match\,(\overline{p_F}_j \to t_j; t_2 : \sigma) \hookrightarrow \overline{p_F}_j' \to t_j')}{(\lambda case\, \overline{\overline{p_F}_i : \sigma, \overline{\psi_F} \to t_i}^{\,i<v})\ t_2 \hookrightarrow \lambda case\, \overline{\overline{p_F}_j' : \overline{\psi_F} \to t_j'}^{\,j<w}}$$

FEval-TyApp

$$\frac{}{t_1\ \sigma \hookrightarrow t_1}$$

FEval-TyAbs

$$\frac{}{\Lambda a.t_1 \hookrightarrow t_1}$$

FEval-TyAbsCase

$$\frac{}{\lambda case\, \overline{@a_i, \overline{p_F}_i : @a, \overline{\psi_F} \to t_i}^{\,i} \hookrightarrow \lambda case\, \overline{\overline{p_F}_i : \overline{\psi_F} \to t_i}^{\,i}}$$

C.1 Translation from the Mixed Polymorphic λ-calculus

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

H-Var

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash^H x \Rightarrow \sigma \rightsquigarrow x}$$

H-Con

$$\frac{K : \overline{a}; \overline{\sigma}; T \in \Gamma}{\Gamma \vdash^H K \Rightarrow \forall\, \overline{a}.\overline{\sigma} \to T\,\overline{a} \rightsquigarrow K}$$

H-Ann

$$\frac{\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho; \Gamma' \rightsquigarrow \dot{t} \qquad \Gamma' \vdash e \Leftarrow \rho \rightsquigarrow t}{\Gamma \vdash^H e : \sigma \Rightarrow \sigma \rightsquigarrow \dot{t}[t]}$$

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

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

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

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

ETm-CheckAbs
$$\frac{\Gamma \vdash \sigma_2 \xrightarrow{skol\ \mathcal{S}} \rho_2; \Gamma_2 \rightsquigarrow \dot{t} \qquad \Gamma_2, x : \sigma_1 \vdash e \Leftarrow \rho_2 \rightsquigarrow t_1}{\Gamma \vdash \lambda x.e \Leftarrow \sigma_1 \to \sigma_2 \rightsquigarrow \lambda x : \sigma_1.(\dot{t}[t_1])}$$

ETm-CheckTyAbs
$$\frac{a \in \Gamma \qquad \Gamma \vdash e \Leftarrow \rho \rightsquigarrow t}{\Gamma \vdash \Lambda a.e \Leftarrow \rho \rightsquigarrow \Lambda a.t}$$

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

ETm-CheckInf
$$\frac{\Gamma \vdash e \Rightarrow \eta^{\epsilon} \rightsquigarrow t \qquad \Gamma \vdash \eta^{\epsilon} \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t} \qquad \text{No other rules match}}{\Gamma \vdash e \Leftarrow \rho \rightsquigarrow \dot{t}[t]}$$

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

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

Arg-App
$$\frac{\Gamma \vdash \sigma_1 \xrightarrow{skol\ \delta} \rho_1; \Gamma' \rightsquigarrow \dot{t} \qquad \Gamma' \vdash e \Leftarrow \rho_1 \rightsquigarrow t \qquad \Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma_2 \Rightarrow \sigma' \rightsquigarrow \overline{arg_F}}{\Gamma \vdash^{A} e, \overline{arg} \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma' \rightsquigarrow \dot{t}[t], \overline{arg_F}}$$

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

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

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

InstT-Shallow

$$\frac{}{\Gamma \vdash \forall\, \overline{a}.\rho \xrightarrow{\ inst\ \mathcal{S}\ } [\overline{\tau}/\overline{a}]\,\rho \rightsquigarrow \bullet\, \overline{\tau}}$$

InstT-Mono

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

InstT-Function

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

InstT-Forall

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

InstT-InfForall

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

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

SkolT-Shallow

$$\frac{}{\Gamma \vdash \forall\, \overline{a}.\rho \xrightarrow{\ skol\ \mathcal{S}\ } \rho; \Gamma, \overline{a} \rightsquigarrow \Lambda \overline{a}.\bullet}$$

SkolT-Mono

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

SkolT-Function

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

SkolT-Forall

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

SkolT-InfForall

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

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

EDecl-NoAnnSingle

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

EDecl-NoAnnMulti

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

EDecl-Ann

$$\frac{\begin{array}{c}\overline{\Gamma \vdash^P \overline{p}_i \Leftarrow \sigma \Rightarrow \sigma_i'; \Delta \rightsquigarrow \overline{p_F}_i : \overline{\psi_F}}^{\,i} \\ \overline{\Gamma, \Delta \vdash \sigma_i' \xrightarrow{\mathit{skol}\ \delta} \rho; \Gamma_i' \rightsquigarrow \dot{t}_i}^{\,i} \\ \overline{\Gamma_i' \vdash e_i \Leftarrow \rho \rightsquigarrow t_i}^{\,i}\end{array}}{\Gamma \vdash x : \sigma; \overline{x\,\overline{p}_i = e_i}^{\,i} \Rightarrow \Gamma, x : \sigma \rightsquigarrow x : \sigma = \lambda \mathit{case}\,\overline{\overline{p_F}_i : \overline{\psi_F} \rightarrow \dot{t}_i[t_i]}^{\,i}}$$

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

Pat-InfEmpty

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

Pat-InfVar

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

Pat-InfCon

$$\frac{\begin{array}{c}K : \overline{a}_0; \overline{\sigma}_0; T \in \Gamma \\ \mathit{binders}^{\delta}(\forall\,\overline{a}_0.\overline{\sigma}_0 \rightarrow T\,\overline{a}_0) = \overline{a}; \rho_0 \\ \Gamma \vdash^P \overline{p} \Leftarrow [\overline{\tau}/\overline{a}]\,\rho_0 \Rightarrow T\,\overline{\tau}; \Delta_1 \rightsquigarrow \overline{p_F}_1 : \overline{\psi_F}_1 \\ \Gamma, \Delta_1 \vdash^P \overline{p'} \Rightarrow \overline{\psi}; \Delta_2 \rightsquigarrow \overline{p_F}_2 : \overline{\psi_F}_2\end{array}}{\Gamma \vdash^P (K\,\overline{p}), \overline{p'} \Rightarrow T\,\overline{\tau}, \overline{\psi}; \Delta_1, \Delta_2 \rightsquigarrow (K\,\overline{p_F}_1), \overline{p_F}_2 : T\,\overline{\tau}, \overline{\psi_F}_2}$$

Pat-InfTyVar

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

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

PAT-CHECKEMPTY

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

PAT-CHECKVAR

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

PAT-CHECKCON

$$\frac{\begin{array}{c} K : \overline{a}_0; \overline{\sigma}_0; T \in \Gamma \\ \mathit{binders}^{\delta}(\forall\, \overline{a}_0.\overline{\sigma}_0 \to T\, \overline{a}_0) = \overline{a}; \rho_0 \\ \Gamma \vdash \sigma_1 \xrightarrow{\mathit{skol}\ \delta} \rho_1; \Gamma_1 \rightsquigarrow \dot{t} \\ \Gamma_1 \vdash^P \overline{p} \Leftarrow [\overline{\tau}/\overline{a}]\, \rho_0 \Rightarrow \rho_1; \Delta_1 \rightsquigarrow \overline{p_F}_1 : \overline{\psi_F}_1 \\ \Gamma, \Delta_1 \vdash^P \overline{p'} \Leftarrow \sigma_2 \Rightarrow \sigma_2'; \Delta_2 \rightsquigarrow \overline{p_F}_2 : \overline{\psi_F}_2 \end{array}}{\Gamma \vdash^P (K\, \overline{p}), \overline{p'} \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma_2'; \Delta_1, \Delta_2 \rightsquigarrow (K\, \overline{p_F}_1), \overline{p_F}_2 : \sigma_1, \overline{\psi_F}_2}$$

PAT-CHECKFORALL

$$\frac{\begin{array}{c} \Gamma, a \vdash^P \overline{p} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{p_F} : \overline{\psi_F} \\ \text{No other rules match} \end{array}}{\Gamma \vdash^P \overline{p} \Leftarrow \forall\, a.\sigma \Rightarrow \sigma'; a, \Delta \rightsquigarrow \overline{p_F} : @a, \overline{\psi_F}}$$

PAT-CHECKTYPE

$$\frac{\begin{array}{c} \overline{a} = f_v(\tau) \\ \Gamma, \overline{a} \vdash^P \overline{p} \Leftarrow [\tau/b]\, \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{p_F} : \overline{\psi_F} \end{array}}{\Gamma \vdash^P @\tau, \overline{p} \Leftarrow \forall\, b.\sigma \Rightarrow \sigma'; \overline{a}, \Delta \rightsquigarrow \overline{p_F} : @\overline{a}, \overline{\psi_F}}$$

PAT-CHECKINFFORALL

$$\frac{\begin{array}{c} \Gamma, \{a\} \vdash^P \overline{p} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{p_F} : \overline{\psi_F} \\ \text{No other rules match} \end{array}}{\Gamma \vdash^P \overline{p} \Leftarrow \forall\, \{a\}.\sigma \Rightarrow \sigma'; \{a\}, \Delta \rightsquigarrow \overline{p_F} : @\overline{a}, \overline{\psi_F}}$$

D EVALUATION REASONINGS

This section provides the reasonings behind the analysis of Section 7, laying the groundwork for more formal proofs in the future.

Property 1: Let Inlining is Type Preserving.
Repeated case analysis on the given derivation (rule ETM-INFLET or rule ETM-CHECKLET, followed by rule EDECL-NOANNSINGLE) results in $\Gamma \vdash x = e_1 \Rightarrow \Gamma, x : \forall\, \overline{\{a\}}.\eta_1^{\epsilon} \rightsquigarrow x : \sigma = t, \Gamma \vdash e_1 \Rightarrow \eta_1^{\epsilon} \rightsquigarrow t$, and either $\Gamma, x : \forall\, \overline{\{a\}}.\eta_1^{\epsilon} \vdash e_2 \Rightarrow \eta^{\epsilon} \rightsquigarrow t$ or $\Gamma, x : \forall\, \overline{\{a\}}.\eta_1^{\epsilon} \vdash e_2 \Leftarrow \rho \rightsquigarrow t$. It is not hard to see that the result follows from this, as typing the variable x (through rule ETM-INFAPP) or the expression e_1 directly, results in an identical typing.

Property 3: Let Extraction is Type Preserving.
Unlike Property 1, Property 3 only holds under lazy instantiation. The reason is that type inference under eager instantiation, produces an instantiated type for e_1 before generalising and placing it in the context. This means that while explicit type application could be possible for e_1 directly, it gets rejected for x.

Property 2: Let Inlining is Dynamic Semantics Preserving.
Repeated case analysis on the first premise (rule ETM-INFLET or rule ETM-CHECKLET, followed by rule EDECL-NOANNSINGLE) tells us that $t_1 = (\lambda x : \sigma.t_3)\, t_4$, where $\Gamma \vdash e_1 \Rightarrow \eta_1^{\epsilon} \rightsquigarrow t_4$, $\sigma = \forall\, \overline{\{a\}}.\eta_1^{\epsilon}$, and $\Gamma, x : \sigma \vdash e_2 \Rightarrow \eta^{\epsilon} \rightsquigarrow t_3$ or $\Gamma, x : \sigma \vdash e_2 \Leftarrow \rho \rightsquigarrow t_3$. Typing e_2 using the variable x (and substituting afterwards) or using the inlined expression e_1 results in a different translation. When typing the variable directly, the type gets instantiated in rule ETM-INFAPP (under eager instantiation)

or rule ETm-CheckInf (under lazy instantiation). Under shallow instantiation, this results in additional type applications to the translated expression, which are erased during evaluation. Under deep instantiation, however, eta expansion is performed on the translated expression, potentially altering the evaluation outcome.

Property 4: Let Extraction is Dynamic Semantics Preserving.
Analogous to Property 2.

Property 5: Signature Property is Type Preserving.
Before proving Property 5, we first introduce two helper properties.

Property 13 (Pattern Checking is Type Preserving).
If $\Gamma \vdash^P \overline{p} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{p_F} : \overline{\psi_F}$ *then* $\Gamma \vdash^P \overline{p} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{p_F} : \overline{\psi_F}$ *where* $type(\overline{\psi}; \sigma') = \sigma$

Property 13 follows by structural induction on the pattern typing derivation.

Property 14 (Expression Checking is Type Preserving).
If $\Gamma \vdash e \Rightarrow \eta^\epsilon \rightsquigarrow t$ *then* $\Gamma \vdash e \Leftarrow \rho \rightsquigarrow \dot{t}[t]$ *where* $\Gamma \vdash \eta^\epsilon \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}$

Under lazy instantiation, Properties 5 and 14 are proven straightforwardly through mutual induction, using Property 13. Neither property holds under eager instantiation, however. The problem lies in the explicit type abstraction case: rule ETm-InfTyAbs eagerly instantiates the type, which is then generalised in rule EDecl-NoAnnMulti, resulting in an inferred forall type. The expression is not accepted by rule ETm-CheckTyAbs under this same inferred type, though. As the type gets skolemised, the inferred variable is placed in the typing environment, but this does not match the specified type variable in rule ETm-CheckTyAbs.

Property 6: Signature Property is Dynamic Semantics Preserving.
Similarly to Property 5, we first introduce two helper properties.

Property 15 (Pattern Checking is Dynamic Semantics Preserving).
If $\Gamma \vdash^P \overline{p} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{p_{F_1}} : \overline{\psi_{F_1}}$ *and* $\Gamma \vdash^P \overline{p} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{p_{F_2}} : \overline{\psi_{F_2}}$ *and* $type(\overline{\psi}; \sigma') = \sigma$ *then* $\overline{p_{F_1}} = \overline{p_{F_2}}$ *and* $\overline{\psi_{F_1}} = \overline{\psi_{F_2}}$

Property 15 follows by structural induction on the pattern typing derivation.

Property 16 (Expression Checking is Dynamic Semantics Preserving).
If $\Gamma \vdash e \Rightarrow \eta^\epsilon \rightsquigarrow t_1$ *and* $\Gamma \vdash e \Leftarrow \rho \rightsquigarrow t_2$ *and* $\Gamma \vdash \eta^\epsilon \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}$ *then* $t_1 \simeq \dot{t}[t_2]$

Under shallow instantiation, properties 6 and 16 are proven straightforwardly using mutual induction. Note that while rule ETm-InfTyAbs instantiates the type eagerly, while rule ETm-CheckTyAbs does not. Under shallow instantiation, this results in additional type applications, which are erased during evaluation (rule FEval-TyApp). However, under deep instantiation, neither property holds, as rule ETm-InfTyAbs performs eta expansion on the translated expression, altering its behaviour.

Property 7: Type Signatures are Dynamic Semantics Preserving.
Similarly to before, we first introduce a new helper property.

Property 17 (Expression Signatures are Dynamic Semantics Preserving).
If $\Gamma \vdash e \Leftarrow \rho_1 \rightsquigarrow t_1$ *and* $\Gamma \vdash e \Leftarrow \rho_2 \rightsquigarrow t_2$ *then* $t_1 \simeq t_2$

Under shallow instantiation, Properties 7 and 17 are proven through straightforwardly mutual induction. Neither property holds under deep instantiation, though, as the skolemisation step in rule EDecl-Ann performs eta expansion on the translated expression, altering its behaviour depending on the provided type signature.

Property 8: Pattern Inlining is Type Preserving.

Case analysis on the given derivation (rule EDecl-NoAnnMulti) tells us that $\overline{\Gamma \vdash^P \overline{p}_i \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{p_F}_i : \overline{\psi_F}}^i$, $\overline{\Gamma, \Delta \vdash e_{i1} \Rightarrow \eta^{\epsilon}_{i1} \rightsquigarrow t_{i1}}^i$, $\overline{\Gamma, \Delta \vdash \eta^{\epsilon}_{i1} \xrightarrow{inst\ \delta} \rho_1 \rightsquigarrow \dot{t}_{i1}}^i$ and $type(\overline{\psi}; \rho_1) = \sigma$. Proving the goal (rule EDecl-NoAnnMulti) requires us to show $\overline{\Gamma \vdash e_{i2} \Rightarrow \eta^{\epsilon}_{i2} \rightsquigarrow t_{i2}}^i$ and $\overline{\Gamma \vdash \eta^{\epsilon}_{i2} \xrightarrow{inst\ \delta} \rho_2 \rightsquigarrow \dot{t}_{i2}}^i$, where $\sigma = \rho_2$. Under lazy instantiation, this results follows straightforwardly from rule ETm-InfAbs and rule ETm-InfTyAbs, as the patterns have simply been moved to the righthandside, and are typed in the exact same way. However, under eager instantiation, this fails as rule ETm-InfTyAbs performs an additional instantiation. Concretely, this means that while explicit type abstractions on the left hand side remain uninstantiated, they are instantiated eagerly under a lambda binder.

Property 10: Pattern Extraction is Type Preserving.

Analogous to Property 8.

Property 9: Pattern Inlining / Extraction is Dynamic Semantics Preserving.

Case analysis on both typing derivations (rule EDecl-NoAnnMulti) tells us that $t_1 = \lambda case\ \overline{\overline{p_F}_i : \overline{\psi_F} \rightarrow \dot{t}_{i1}[t_{i1}]}^i$ and $t_2 = \lambda case\ \overline{\cdot : \cdot \rightarrow \dot{t}_{i2}[t_{i2}]}^i$, where $\overline{\Gamma \vdash^P \overline{p}_i \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{p_F}_i : \overline{\psi_F}}^i$, $\overline{\Gamma, \Delta \vdash e_{i1} \Rightarrow \eta^{\epsilon}_{i1} \rightsquigarrow t_{i1}}^i$, and $\overline{\Gamma \vdash e_{i2} \Rightarrow \eta^{\epsilon}_{i2} \rightsquigarrow t_{i2}}^i$. As we would expect, the patterns from $\overline{p_F}_{i1}$ have been shifted into the expression t_{i1}, resulting in t_{i2}. For expression variable patterns, the context equivalence follows trivially, from the evaluation relations rule FMatch-Var and rule FEval-AppAbs. Type variable patterns are a bit more involved, as they are eagerly instantiated in a lambda binder (rule ETm-InfTyAbs), but not inside of a declaration (rule EDecl-NoAnnMulti). Under shallow instantiation, the goal follows through simple induction, as the instantiation only results in additional type applications, which are erased during evaluation. However, under deep instantiation, rule ETm-InfTyAbs results in an eta expansion of the translated expression, altering the dynamic semantics in the process.