Spine-local Type Inference

Christopher Jenkins
University of Iowa
christopher-jenkins@uiowa.edu

Aaron Stump
University of Iowa
aaron-stump@uiowa.edu

ABSTRACT

We present *spine-local* type inference, a method of partially inferring omitted type annotations for System F based on local type inference. *Local type inference* relies on bidirectional rules to propagate type information into and out of adjacent nodes of the AST and restricts type-argument inference to a single node. Spine-local inference relaxes this restriction, allowing it to occur only within an *application spine*, and improves upon it by using *contextual type-argument inference*. As our goal is to explore the design space of local type inference, we show that, relative to other variants, spine-local type inference better supports desirable features such as first-class curried applications and partial type applications, and it has the ability to infer types for some terms not otherwise possible. Our approach enjoys usual properties of a bidirectional system of having a specification for our inference algorithm and predictable requirements for typing annotations, and in particular maintains some advantages of local type inference such as a relatively simple implementation and a tendency to produce good-quality error messages when type inference fails.

CCS CONCEPTS

• **Software and its engineering → Language features**;

KEYWORDS

bidirectional typechecking, polymorphism, type errors

ACM Reference Format:
Christopher Jenkins and Aaron Stump. 2018. Spine-local Type Inference. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3310232.3310233

1 INTRODUCTION

Local type inference[16] is a simple yet effective partial method for inferring types for programs. In contrast to complete methods (e.g. the Damas-Milner type system[3]) which can type programs without any annotations by restricting the type language, *partial* methods require explicit annotations in some cases and, in exchange, are suitable for languages with rich type features such as impredicativity and subtyping[13, 16], dependent types[23], and higher-rank types[14], where complete type inference may be undecidable.

IFL 2018, September 5–7, 2018, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09…$15.00
https://doi.org/10.1145/3310232.3310233

Local type inference is also contrasted with *global* inference methods (usually based on unification) which are able to infer more missing annotations by solving typing constraints generated from the entire program. Though more powerful, global inference methods can also be more difficult for programmers to use when type inference fails, as they can generate type errors whose root cause is distant from the location the error is reported[9]. For example, for the expression $\lambda\, x : \mathbb{B}.\, x$ an error message like

`error: expected` $\mathbb{N} \to \mathbb{N}$ `, got` $\mathbb{B} \to \mathbb{B}$

requires users to work backwards to understand why the typechecker tried to unify these two types, which can be non-trivial.

Local type inference address this issue by only propagating typing information between adjacent nodes of the abstract syntax tree (AST), allowing programmers to reason *locally* about type errors. It achieves this by using two main techniques: *bidirectional type inference rules* and *local type-argument inference*.

The first of these techniques, bidirectional type inference, is not unique to local type inference (see [5, 14, 21]), and uses two main judgment forms, often called *synthesis* and *checking* mode. When a term t synthesizes type T, we view this typing information as coming up and out of t and as available for use in typing nearby terms; when t checks against type T (called in this paper the *contextual* type), this information is being pushed down and in to t and is provided by nearby terms. The second of these techniques, local type-argument inference, finds the missing types arguments in polymorphic function applications by using only the type information available at an application node of the AST. For a simple example, consider the expression `id` z where `id` has type $\forall\, X.\, X \to X$ and z has type $\mathbb{N}$. Here we can perform *synthetic* type-argument inference by synthesizing the type of z and comparing this to the type of `id` to infer we should instantiate type variable X with $\mathbb{N}$.

Local type inference has a number of desirable properties. Using just these two techniques it can in practice infer a good deal of type annotations, and those it cannot are predictable and coincide with programmers' expectations that they serve as useful and machine-checked documentation[7, 16]. Without further instrumentation, local type inference already tends to report errors close to where further annotations are required; recently, it has been used as the basis for developing autonomous type-driven debugging and error explanations[17]. The type inference algorithms of [13, 16] admit a specification for their behavior, helping programmers understand the system without requiring knowledge of its implementation. Add to this its relative simplicity and robustness when extended to richer type systems and it seems unsurprising that it is a popular choice for type inference in programming languages.

Unfortunately, local type inference can fail in unituitive ways even allowing for their restricted scope of inferece. Consider trying to check that the expression `pair` $(\lambda\, x.\, x)\; z$ has type $\langle(\mathbb{N} \to \mathbb{N}) \times \mathbb{N}\rangle$, assuming `pair` has type $\forall\, X.\, \forall\, Y.\, X \to Y \to \langle X \times Y\rangle$). The inference systems presented in [13, 16] will fail here because $\lambda\, x.\, x$ does not

synthesize a type. The techniques proposed in the literature for dealing with such cases include classifying and avoiding such "hard-to-synthesize" terms[7] and utilizing the partial type information provided by polymorphic functions[13]; the former was dismissed as unsatisfactory by the same authors that introduced it and the latter is of no help in this situation, since the type of `pair` tells us nothing about the expected type of $\lambda\, x.\, x$. What we need in this case is *contextual* type-argument inference, utilizing the information available from the contextual type of the result of the application to know argument $\lambda\, x.\, x$ is expected to have type $\mathbb{N} \to \mathbb{N}$.

Additionally, languages using local type inference usually use fully-uncurried applications in order to maximize the notion of "locality" for type-argument inference, improving its effectiveness. The programmer can still use curried applications if desired, but "they are second-class in this respect."[16]. It is also usual for type arguments to be given in an "all or nothing" fashion in such languages, meaning that even if only *one* cannot be inferred, all must be provided. We believe that currying and partial type applications are useful idioms for functional programming and wish to support them as first-class language features.

1.1 Contributions

In this paper, we explore the design space of local type inference in the setting of System F[6] by developing *spine-local* type inference, an approach that both expands the locality of type-argument inference to an *application spine* and augments its effectiveness by using the *contextual* type of the spine. In doing so, we

- show that we better support first-class currying and partial type applications, and can infer types for some "hard-to-synthesize" terms that other local systems would not type;
- provide a specification for contextual type-argument inference with respect to which we show our algorithm is sound and complete
- give a weak completeness theorem indicating where the programmer can expect type inference to succeed and where it needs additional annotations when it fails.

Spine-local type inference is implemented in Cedille[18], a functional programming language based on the pure type-theory CDLE that contains System F as a sub-language and aims to be a simple language in which one can derive inductive datatypes through impredicative lambda-encodings rather than baking them into its core theory. Therefore, type-inference systems for Cedille need not address these features explicitly, as focusing on the more fundamental task of inferring type annotations improves their usability. Though the setting for this paper is only a sub-language of Cedille, experience tells us that spine-local type inference already makes using Cedille's rich type features much more convenient.

The rest of this paper is organized as follows: in Section 2 we cover the syntax and some useful terminology for our setting; in Section 3 we present the type inference rules constituting a specification for contextual type-argument inference, consider its annotation requirements, and illustrate its use, limitations, and the type errors it presents to users; in Section 4 we show the prototype-matching algorithm implementing contextual type-argument inference; and in Section 5 we discuss how this work compares to other approaches to type inference.

2 INTERNAL AND EXTERNAL LANGUAGE

Type inference can be viewed as a relation between an *internal* language of terms, where all needed typing information is present, and an *external* language, in which programmers work directly and where some of this information can be omitted for their convenience. Under this view, type inference for the external language not only associates a term with some type but also with some *elaborated* term in the internal language in which all missing type information has been restored. In this section, we present the syntax for our internal and external languages as well as introduce some terminology that will be used throughout the rest of this paper.

2.1 Syntax

We take as our internal language explicitly typed System F (see [6]); we review its syntax below:

$$\begin{array}{rc} \textbf{Types} & S, T, U, V ::= X, Y, Z \mid S \to T \mid \forall X.\, T \\ \textbf{Contexts} & \Gamma ::= \cdot \mid \Gamma, X \mid \Gamma, x{:}T \\ \textbf{Internal Terms} & e, p ::= x \mid \lambda\, x{:}T.\, e \mid \Lambda X.\, e \mid e\ e' \mid e[T] \end{array}$$

Types consist of type variables, arrow types, and type quantification, and typing contexts consist of the empty context, type variables (also called the context's *declared* type variables), and term variables associated with their types. The internal language of terms consists of variables, λ-abstractions with annotations on bound variables, Λ-abstractions for polymorphic terms, and term and type applications. Our notational convention in this paper is that term meta-variable e indicates an elaborated term for which all type arguments are known, and p indicates a *partially* elaborated term where some elaborated type arguments are type meta-variables (discussed in Section 3).

The external language contains the same terms as the internal language as well as bare λ-abstractions – that is, λ-abstractions missing an annotation on their bound variable:

$$\textbf{External Terms} \quad t, t' ::= x \mid \lambda\, x{:}T.\, t \mid \lambda\, x.\, t \mid \Lambda X.\, t \mid t\ t' \mid t[T]$$

Types and contexts are the same as for the internal language.

2.2 Terminology

In both the internal and external languages, we say that the *applicand* of a term or type application is the term in the function position. A *head* is either a variable or abstraction (term or type), and an *application spine*[2] (or just *spine*) is a view of an application as consisting of a head (called the *spine head*) followed by a sequence of (term and type) arguments. The *maximal application* of a sub-expression is the spine in which it occurs as an applicand, or just the sub-expression itself if it does not. For example, spine $x[S]\ y\ z$ is the maximal application of itself and its applicand sub-expressions x, $x[S]$, and $x[S]\ y$, with x as head of the spine. Predicate $App(t)$ indicates term t is some term or type application (in either language) and we define it formally as $(\exists\, t_1, t_2.\ t = t_1\ t_2) \vee (\exists\, t', S.\ t = t'[S])$.

Turning to definitions for types and contexts, function $DTV(\Gamma)$ calculates the set of *declared type variables* of context Γ and is

defined recursively by the following set of equations:

$$\begin{aligned} DTV(\cdot) &= \varnothing \\ DTV(\Gamma, X) &= DTV(\Gamma) \cup \{X\} \\ DTV(\Gamma, x{:}T) &= DTV(\Gamma) \end{aligned}$$

Predicate $WF(\Gamma, T)$ indicates that type T is *well-formed* under Γ – that is, all free type variables of T occur as declared type variables in Γ (formally $FV(T) \subseteq DTV(\Gamma)$).

3 TYPE INFERENCE SPECIFICATION

The typing rules for our internal language are standard for explicitly typed System F and are omitted (see Ch. 23 of [15] for a thorough discussion of these rules). We write $\Gamma \vdash e : T$ to indicate that under context Γ internal term e has type T. For type inference in the external language, Figure 1 shows judgment $\vdash_\delta$ which consists mostly of standard (except for *AppSyn* and *AppChk*) bidirectional inference rules with elaboration to the internal language, and Figure 2 shows the specification for contextual type-argument inference. Judgment $\vdash^{\mathrm{P}}$ in Figure 2b handles traversing the spine and judgment $\vdash^{\cdot}$ in Figure 2c types its term applications and performs type-argument inference (both synthetic and contextual). Figure 2a gives a "shim" judgment $\vdash^{\mathrm{I}}$ which bridges the bidirectional rules with the specification for rhetorical purposes (discussed below). While these rules are not fully syntax-directed, they *are* subject-directed, meaning that for each judgment the shape of the term we are typing (i.e. the *subject* of typing) uniquely determines which rule applies.

Bidirectional Rules. We now consider more closely each judgment form and its rules starting with $\vdash_\delta$, the point of entry for type inference. The two modes for type inference, checking and synthesizing, are indicated resp. by $\vdash_\Downarrow$ (suggesting pushing a type down and into a term) and $\vdash_\Uparrow$ (suggesting pulling a type up and out of a term). Following the notational convention of Peyton Jones et al.[14] we abbreviate two inference rules that differ only in their direction to one by writing $\vdash_\delta$, where δ is a parameter ranging over $\{\Uparrow, \Downarrow\}$. We read judgment $\Gamma \vdash_\Uparrow t : T \rightsquigarrow e$ as: "under context Γ, term t synthesizes type T and elaborates to e," and a similar reading for checking mode applies for $\vdash_\Downarrow$. When the direction does not matter, we will simply say that we can *infer* t has type T.

Rule *Var* is standard. Rule *Abs* says we can infer missing type annotation T on a λ-abstraction when we have a contextual arrow type $T \to S$. Rules *AAbs* and *TAbs* say that Λ- and annotated λ-abstractions can have their types either checked or synthesized. *TApp* says that a type application $t[S]$ has its type inferred in either mode when the applicand t synthesizes a quantified type. The reason for this asymmetry between the modes of the conclusion and the premise is that even when in checking mode, it is not clear how to work backwards from type $[S/X]T$ to $\forall X.\, T$.

AppSyn and *AppChk* are invoked on maximal applications and are the first non-standard rules. To understand how these rules work, we must 1) explain the "shim" judgment $\vdash^{\mathrm{I}}$ serving as the interface for contextual type-argument inference and 2) define meta-language function *MV*. Read $\Gamma; T_? \vdash^{\mathrm{I}} t\ t' : T \rightsquigarrow (p, \sigma)$ as "under context Γ and with (optional) contextual type $T_?$, we partially infer application $t\ t'$ has type T with elaboration p and solution σ," where $T_?$ indicates either a type T (as in *AppChk*) or no contextual type information ? (as in *AppSyn*) and σ is a substitution mapping a subset of the meta-variables (i.e. the originally omitted type arguments) in p to contextually-inferred type arguments.

In rule *AppSyn*, ? is an artifice provided to $\vdash^{\mathrm{I}}$ solely to indicate no contextual type is available and is not propagated throughout the other rules. We constrain σ to be the identity substitution (notation σ_{id}) and insist the elaborated term has no unsolved meta-variables, matching our intuition that all type arguments must be inferred synthetically. In rule *AppChk*, we provide the contextual type to $\vdash^{\mathrm{I}}$ and check (implicitly) that it equals $\sigma\ T$ and (explicitly) that all remaining meta-variables in p are solved by σ, then elaborate $\sigma\ p$ (the replacement of each meta-variable in p with its mapping in σ). Shared by both is the second premise of the (anonymous) rule introducing $\vdash^{\mathrm{I}}$ that σ solves precisely the meta-variables of the partially inferred type T for application $t\ t'$.

Meta-variables. What do we mean by the "meta-variables" of partial elaborations and types? When t is a term application with some type arguments omitted in its spine, then (under context Γ) its partial elaboration p from type-argument inference fills in each missing type argument with either a type (well-formed under Γ) or with a *meta-variable* (a type variable not declared in Γ) depending on whether it was inferred through synthesizing the types of the term arguments of t. For example, if $t = \mathtt{pair}\ (\lambda x.\, x)\ z$ and we wanted to check that it has type $T = \langle(\mathbb{N} \to \mathbb{N}) \times \mathbb{N}\rangle$ under a typing context Γ associating `pair` with type $\forall X.\, \forall Y.\, X \to Y \to \langle X \times Y\rangle$ and z with type $\mathbb{N}$, then we could derive

$$\Gamma; T \vdash^{\mathrm{I}} t : \langle X \times \mathbb{N}\rangle \rightsquigarrow (\mathtt{pair}[X][\mathbb{N}]\ (\lambda x{:}\mathbb{N}.\, x)\ z, [\mathbb{N} \to \mathbb{N}/X])$$

(assuming some base type $\mathbb{N}$, some family of base types $\langle S \times T\rangle$ for all types S and T, and assuming X is not declared in Γ.) Looking at the partial elaboration of t, we would see that type argument X was inferred from its contextual type $\langle(\mathbb{N} \to \mathbb{N}) \times \mathbb{N}\rangle$ and that Y was inferred from the synthesized types of the arguments to `pair` (in this case z).

Meta-variables never occur in a judgment formed by $\vdash_\delta$, only in the judgments of Figure 2. In particular, these rules enforce that meta-variables in a partial elaboration p can occur *only* as type arguments in its spine, not within its head or term arguments. This restriction guarantees *spine-local* type-argument inference and helps to narrow the programmer's focus when debugging type errors. Furthermore, meta-variables correspond to omitted type arguments *injectively*, simplifying the kind of reasoning needed for debugging type errors. We make this precise by defining meta-language function $MV(\Gamma, -)$ which yields the set of meta-variables occurring in its second argument with respect to the context Γ. *MV* is overloaded to take both types and elaborated terms for its second argument: for types we define $MV(\Gamma, T) = FV(T) - DTV(\Gamma)$, the set of free variables in T less the declared type variables of Γ; for terms, $MV(\Gamma, p)$ is defined recursively by the following equations:

$$\begin{aligned} MV(\Gamma, p) &= \varnothing && \text{when } \neg App(p) \\ MV(\Gamma, p[X]) &= MV(\Gamma, p) \cup \{X\} && \text{when } X \notin DTV(\Gamma) \\ MV(\Gamma, p[S]) &= MV(\Gamma, p) && \text{when } WF(\Gamma, S) \\ MV(\Gamma, p\ e) &= MV(\Gamma, p) \end{aligned}$$

Using our running example where the subject t is $\mathtt{pair}\ (\lambda x.\, x)\ z$ we can now show how the meta-variable checks are used in rules *AppSyn* and *AppChk*. We have for our partially elaborated term that

$$\boxed{\Gamma \vdash_\delta t : T \rightsquigarrow e} \qquad \frac{}{\Gamma \vdash_\delta x : \Gamma(x) \rightsquigarrow x}\ Var \qquad \frac{\Gamma, x{:}T \vdash_\Downarrow t : S \rightsquigarrow e}{\Gamma \vdash_\Downarrow \lambda x.\, t : T \to S \rightsquigarrow \lambda x{:}T.\, e}\ Abs$$

$$\frac{\Gamma, x{:}T \vdash_\delta t : S \rightsquigarrow e}{\Gamma \vdash_\delta \lambda x{:}T.\, t : T \to S \rightsquigarrow \lambda x{:}T.\, e}\ AAbs \qquad \frac{\Gamma, X \vdash_\delta t : T \rightsquigarrow e}{\Gamma \vdash_\delta \Lambda X.\, t : \forall X.T \rightsquigarrow \Lambda X.\, e}\ TAbs \qquad \frac{\Gamma \vdash_\Uparrow t : \forall X.T \rightsquigarrow e}{\Gamma \vdash_\delta t[S] : [S/X]T \rightsquigarrow e[S]}\ TApp$$

$$\frac{\Gamma; ? \vdash^{\mathrm{I}} t\ t' : T \rightsquigarrow (e, \sigma_{id}) \quad MV(\Gamma, e) = \varnothing}{\Gamma \vdash_\Uparrow t\ t' : T \rightsquigarrow e}\ AppSyn \qquad \frac{\Gamma; \sigma\ T \vdash^{\mathrm{I}} t\ t' : T \rightsquigarrow (p, \sigma) \quad MV(\Gamma, p) = dom(\sigma)}{\Gamma \vdash_\Downarrow t\ t' : \sigma\ T \rightsquigarrow \sigma\ p}\ AppChk$$

Figure 1: Bidirectional inference rules with elaboration

(a) Shim (specification)

$$\frac{\Gamma \vdash^{\mathrm{P}} t\ t' : T \rightsquigarrow (p, \sigma) \quad MV(\Gamma, T) = dom(\sigma) \quad T_? \in \{?, \sigma\ T\}}{\Gamma; T_? \vdash^{\mathrm{I}} t\ t' : T \rightsquigarrow (p, \sigma)}$$

(b) $\boxed{\Gamma \vdash^{\mathbf{P}} t : T \rightsquigarrow (p, \sigma)}$

$$\frac{\neg App(t) \quad \Gamma \vdash_\Uparrow t : T \rightsquigarrow e}{\Gamma \vdash^{\mathrm{P}} t : T \rightsquigarrow (e, \sigma_{id})}\ PHead \qquad \frac{\Gamma \vdash^{\mathrm{P}} t : \forall X.T \rightsquigarrow (p, \sigma)}{\Gamma \vdash^{\mathrm{P}} t[S] : [S/X]T \rightsquigarrow (p[S], \sigma)}\ PTApp \qquad \frac{\Gamma \vdash^{\mathrm{P}} t : T \rightsquigarrow (p, \sigma) \quad \Gamma \vdash^{\cdot} (p{:}T, \sigma) \cdot t' : T' \rightsquigarrow (p', \sigma')}{\Gamma \vdash^{\mathrm{P}} t\ t' : T' \rightsquigarrow (p', \sigma')}\ PApp$$

(c) $\boxed{\Gamma \vdash^{\cdot} (p{:}T, \sigma) \cdot t' : T' \rightsquigarrow (p', \sigma')}$

$$\frac{\sigma'' \in \{\sigma, [S/X] \circ \sigma\} \quad WF(\Gamma, S) \quad \Gamma \vdash^{\cdot} (p[X]{:}T, \sigma'') \cdot t' : T' \rightsquigarrow (p', \sigma')}{\Gamma \vdash^{\cdot} (p{:}\forall X.\, T, \sigma) \cdot t' : T' \rightsquigarrow (p', \sigma')}\ PForall \qquad \frac{MV(\Gamma, \sigma\ S) = \varnothing \quad \Gamma \vdash_\Downarrow t' : \sigma\ S \rightsquigarrow e'}{\Gamma \vdash^{\cdot} (p{:}S \to T, \sigma) \cdot t' : T \rightsquigarrow (p\ e', \sigma)}\ PChk$$

$$\frac{MV(\Gamma, \sigma\ S) = \overline{Y} \neq \varnothing \quad \Gamma \vdash_\Uparrow t' : \overline{[U/Y]}\ \sigma\ S \rightsquigarrow e'}{\Gamma \vdash^{\cdot} (p{:}S \to T, \sigma) \cdot t' : \overline{[U/Y]}\ T \rightsquigarrow ((\overline{[U/Y]}\ p)\ e', \sigma)}\ PSyn$$

Figure 2: Specification for contextual type-argument inference

$MV(\Gamma, \texttt{pair}[X][\mathbb{N}]\ (\lambda x{:}\mathbb{N}.\, x)\ z) = \{X\}$ and also for our type that $MV(\Gamma, \langle X \times \mathbb{N} \rangle) = \{X\}$. If we have a derivation of the judgment above formed by $\vdash^{\mathrm{I}}$ we can then derive with rule *AppChk*

$$\Gamma \vdash_\Downarrow t : \langle (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \rangle \rightsquigarrow \texttt{pair}[\mathbb{N} \to \mathbb{N}][\mathbb{N}]\ (\lambda x{:}\mathbb{N}.\, x)\ z)$$

because substitution $[\mathbb{N} \to \mathbb{N}/X]$ solves the remaining meta-variable X in the elaborated term and type, and when utilized on the partially inferred type $\langle X \times \mathbb{N} \rangle$ yields the contextual type for the term. However, we would not be able to derive with rule *AppSyn*

$$\Gamma \vdash_\Uparrow t : \langle (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \rangle \rightsquigarrow \texttt{pair}[\mathbb{N} \to \mathbb{N}][\mathbb{N}]\ (\lambda x{:}\mathbb{N}.\, x)\ z)$$

since we do not have σ_{id} as our solution and we have meta-variable X remaining in our partial elaboration and type. Together, the checks in *AppSyn* and *AppChk* ensure that meta-variables are never passed up and out of a maximal application during type inference.

Specification Rules. Judgment $\vdash^{\mathrm{I}}$ serves as an interface to spine-local type-argument inference. In Figure 2a it is defined in terms of the specification for contextual type-argument inference given by judgments $\vdash^{\mathrm{P}}$ and $\vdash^{\cdot}$; we call it a "shim" judgment because in Figure 4a we give for it an alternative definition using the algorithmic rules in which the condition $MV(\Gamma, T) = dom(\sigma)$ is not needed. Its purpose, then, is to cleanly delineate what we consider specification and implementation for our inference system.

Though the details of maintaining spine-locality and performing synthetic type-argument inference permeate the inference rules for $\vdash^{\mathrm{P}}$ and $\vdash^{\cdot}$, these rules form a specification in that they fully abstract away the details of contextual type-argument inference, describing how solutions are used but omitting how they are generated. Spine-locality in particular contributes to our specification's size – what would be one or two rules in a fully-uncurried language with all-or-nothing type argument applications is decomposed in our system into multiple inference rules to support currying and partial type applications.

Judgment $\vdash^{\mathrm{P}}$ contains three rules and serves to dig through a spine until it reaches its head, then work back up the spine typing its term and type applications. The reading for it is the same as for $\vdash^{\mathrm{I}}$, less the optional contextual type. Rule *PHead* types the spine head t by deferring to $\vdash_\Uparrow$; our partial solution is σ_{id} since no meta-variables are present in a judgment formed by $\vdash_\Uparrow$. *PTApp* is similar to *TApp* except it additionally propagates solution σ. Rule *PApp* is used for term applications: first it partially synthesizes a type T for the applicand and then it uses judgment $\vdash^{\cdot}$ to ensure that a function of type T can be applied to the argument t'

Judgment $\vdash^{\cdot}$ performs synthetic and contextual type-argument inference and ensures that term applications with omitted type arguments are well-typed. We read $\Gamma \vdash^{\cdot} (p{:}T, \sigma) \cdot t' : T' \rightsquigarrow (p', \sigma')$

as "under context Γ, elaborated applicand p of partial type T together with solution σ can be applied to term t'; the application has type T' and elaborates p' with solution σ'."

Contextual type-argument inference happens in rule *PForall*, which says that when the applicand has type $\forall X.\, T$ we can choose to guess any well-formed S for our contextual type argument by picking $\sigma'' = [S/X] \circ \sigma$ (indicating σ'' contains all the mappings present in σ and an additional mapping S for X), or choose to attempt to synthesize it later from an argument by picking $\sigma'' = \sigma$. *The details of which S to guess, or whether we should guess at all, are not present in this specificational rule.* In both cases, we elaborate the applicand to $p[X]$ of type T and check that it can be applied to t' – we do this even when we guess S for X to maintain the invariant that all partial elaborations p and solutions σ we generate satisfy $dom(\sigma) \subseteq MV(\Gamma, p)$, needed when checking in the (specificational) rule for $\vdash^{\mathrm{I}}$ that these guessed solutions are ultimately justified by the contextual type (if any) of our maximal application.

We illustrate the use of *PForall* with an example: if the subject of (i.e. input to) judgment $\vdash^{\cdot}$ is

$$(\texttt{pair} : \forall X.\, \forall Y.\, X \to Y \to \langle X \times Y \rangle, \sigma_{id}) \cdot (\lambda x.\, x)$$

then after two uses of rule *PForall* where we guess $\mathbb{N} \to \mathbb{N}$ for X and decline to guess for Y the subject would be:

$$(\texttt{pair}[X][Y] : X \to Y \to \langle X \times Y \rangle, [\mathbb{N} \to \mathbb{N}/X]) \cdot (\lambda x.\, x)$$

After working through omitted type arguments, $\vdash^{\cdot}$ requires that we eventually reveal some arrow type $S \to T$ to type a term application. When it does we have two cases, handled resp. by *PChk* and *PSyn*: either the domain type S of applicand p together with solution σ provide enough information to fully know the expected type for argument t' (i.e. $MV(\Gamma, \sigma\ p) = \varnothing$), or else they do not and we have some non-empty set of unsolved meta-variables $\overline{Y}$ in S corresponding to type arguments we must synthesize. Having full knowledge, in *PChk* we check t' has type $\sigma\ S$; otherwise, in *PSyn* we try to solve meta-variables $\overline{Y}$ by synthesizing a type for t' and checking it is instantiation $\overline{[U/Y]}$ (vectorized notation for the simultaneous substitution of types $\overline{U}$ for $\overline{Y}$) of $\sigma\ S$. Once done, we conclude with result type $\overline{[U/Y]}\ T$ and elaboration $(\overline{[U/Y]}\ p)\ e$ for the application, as the meta-variables $\overline{Y}$ of p corresponding to omitted type arguments have now been fully solved by type-argument synthesis. Together, *PChk* and *PSyn* prevent meta-variables from being passed down to term argument t', as we require that it either check against or synthesize a well-formed type.

We illustrate the use of rule *PSyn* with and example: suppose that under context Γ the subject of judgment $\vdash^{\cdot}$ is

$$(\texttt{pair}[X][Y]\ (\lambda x : \mathbb{N}.\, x) : Y \to \langle X \times Y \rangle, [\mathbb{N} \to \mathbb{N}/X]) \cdot z$$

and furthermore that $\Gamma \vdash_{\Uparrow} z : \mathbb{N}$. Then we have instantiation $[\mathbb{N}/Y]$ from synthetic type-argument inference and use it to produce for the application the result type $[\mathbb{N}/Y]\ \langle X \times Y \rangle = \langle X \times \mathbb{N} \rangle$ and the elaboration $\texttt{pair}[X][\mathbb{N}]\ (\lambda x : \mathbb{N}.\, x)\ z$. Note that synthesized type arguments are used *eagerly*, meaning that the typing information synthesized from earlier arguments can in some cases be used to infer the types of later arguments in *checking* mode (see Section 3.2). This is reminiscent of *greedy* type-argument inference for type systems with subtyping[1, 4], which is known to cause unintuitive type inference failures due to sub-optimal type arguments (i.e. less general wrt to the subtyping relation) being inferred. As System F lacks subtyping, this problem does not affect our type inference system and we can happily utilize synthesized type arguments eagerly (see Section 5 for more discussion of this).

3.1 Soundness, Weak Completeness, and Annotation Requirements

The inference rules in Figure 2 for our external language are *sound* with respect to the typing rules for our internal language (i.e. explicitly typed System F), meaning that elaborations of typeable external terms are typeable at the same type[1]:

Theorem 1. *(Soundness of $\vdash_{\delta}$):*
If $\Gamma \vdash_{\delta} t : T \rightsquigarrow e$ *then* $\Gamma \vdash e : T$.

Our inference rules also enjoy a trivial form of completeness that serves as a sanity-check with respect to the internal language: since any fully annotated internal language term e is in the external language, we expect that e should be typeable using the typing rules for external terms:

Theorem 2. *(Trivial Completeness of $\vdash_{\delta}$):*
If $\Gamma \vdash e : T$ *then* $\Gamma \vdash_{\delta} e : T \rightsquigarrow e$

A more interesting form of completeness comes from asking *which* external terms can be typed – after all, this is precisely what a programmer needs to know when trying to debug type inference failures! Since our external language contains terms without any annotations and our type language is impredicative System F, we know from [22] that type inference is in general undecidable. Therefore, to state a completeness theorem for type inference we must first place some restrictions on the set of external terms that can be the subject of typing.

We start by defining what it means for t to be a *partial erasure* of internal term e. The grammar given in Section 2 for the external language does not fully express where we hope our inference rules will restore missing type information. Specifically, the rules in Figures 1 and 2 will try to infer annotations on bare λ-abstractions and only try to infer missing type arguments that occur in the applicand of a term application. For example, given (well-typed) internal term $x[S_1][S_2]\ y[T]$ and external term $x\ y$, our inference rules will try to infer the missing type arguments S_1 and S_2 but *will not* try to infer the missing T.

A more artificial restriction on partial erasures is that the sequence of type arguments occurring between two terms in an application can only be erased in a right-to-left fashion. For example, given internal term $x[S_1][S_2]\ y[T_1][T_2]\ z$, the external term $x\ y[T_1]\ z$ is a valid erasure (S_1 and S_2 are erased between x and y, and between y and z rightmost T_2 is erased), but term $x[S_2]\ y[T_2]\ z$ is not. This restriction helps preserve soundness of the external type inference rules by ensuring that every explicit type argument preserved in an erasure of an internal term e instantiates the same type variable it did in e; it is artificial because we could instead have introduced notation for "explicitly erased" type arguments in the external language, such as $x[\_][S_2]\ y$, to indicate the first type argument has been erased, or allow type arguments to be given *by*

[1] A complete list of proofs for non-trivial theorems can be found in the proof appendix at http://homepage.cs.uiowa.edu/~cwjnkins/Papers/JS18_Spine-local/proof-appendix.pdf

name such as $x[?X_2 = S_2]\ y$, but chose not to do so to simplify the presentation of our inference rules and language.

The above restrictions for partial erasure are made precise by the functions $\lfloor \_ \rfloor$ and $\lfloor \_ \rfloor_a$ which map an internal term e to sets of partial erasures $\lfloor e \rfloor$. They are defined mutually recursively below:

$$\begin{aligned}
\lfloor \lambda\, x{:}T.\, e \rfloor &= \{\lambda\, x{:}T.\, t \mid t \in \lfloor e \rfloor\} \cup \{\lambda\, x.\, t \mid t \in \lfloor e \rfloor\} \\
\lfloor \Lambda\, X.\, e \rfloor &= \{\Lambda\, X.\, t \mid t \in \lfloor e \rfloor\} \\
\lfloor e\ e' \rfloor &= \{t\ t' \mid t \in \lfloor e \rfloor_a \wedge t' \in \lfloor e' \rfloor\} \\
\lfloor e[S] \rfloor &= \{t[S] \mid t \in \lfloor e \rfloor\} \\
\\
\lfloor e[S] \rfloor_a &= \{t \mid t \in \lfloor e \rfloor_a\} \cup \{t[S] \mid t \in \lfloor e \rfloor\} \\
\lfloor e \rfloor_a &= \lfloor e \rfloor \text{ otherwise}
\end{aligned}$$

We are now ready to state a weak completeness theorem for typing terms in the external language which over-approximates the annotations required for type inference to succeed (we write $\forall\, \overline{X}.\, T$ to mean some number of type quantifications over type T)

Theorem 3. *(Weak completeness of $\vdash_{\Uparrow}$):*
Let e be a term of the internal language and t be an external language term such that $t \in \lfloor e \rfloor$. If $\Gamma \vdash e : T$ then $\Gamma \vdash_{\Uparrow} t : T \rightsquigarrow e$ when the following conditions hold for each sub-expression e' of e, corresponding sub-expression t' of t, and corresponding sub-derivation $\Gamma' \vdash e' : T'$ of $\Gamma \vdash e : T$:

1. *If $e' = \lambda\, x : S.\, e''$ for some S and e'', then $t' = \lambda\, x : S.\, t''$ for some t'*
2. *If e' is a maximal term application in e and if $\Gamma' \vdash^P t' : T'' \rightsquigarrow (p, \sigma_{id})$ for some T'' and p, then $MV(\Gamma, p) = \varnothing$.*
3. *If e' is a term application and $t' = t_1\ t_2$ for some t_1 and t_2, and if $\Gamma' \vdash^P t_1 : T'' \rightsquigarrow (p, \sigma_{id})$ for some T'' and p, then $T'' = \forall\, \overline{X}.\, S_1 \to S_2$ for some S_1 and S_2.*
4. *If e' is a type application and $t' = t''[S]$ for some t'' and S, and $\Gamma' \vdash^P t'' : T'' \rightsquigarrow (p, \sigma_{id})$ for some T'' and p, then $T'' = \forall\, X.\, S'$ for some S'.*

Theorem 3 only considers synthetic type-argument inference, and in practice condition (1) is too conservative thanks to contextual type-argument inference. Though a little heavyweight, our weak completeness theorem can be translated into a reasonable guide for where type annotations are required when type synthesis fails. Conditions (3) and (4) suggest that when the applicand of a term or type application already partially synthesizes some type, the programmer should give enough annotations to at least reveal it has the appropriate shape (resp. a type arrow or quantification) for the application. (2) indicates that type variables that do not occur somewhere corresponding to a term argument of an application should be instantiated explicitly, as there is no way for synthetic type-argument inference to do so. For example, in the expression $f\ z$ if f has type $\forall\, X.\, \forall\, Y.\, Y \to X$ there is no way to instantiate X from synthesizing argument z. Finally, condition (1) we suggest as the programmer's last resort: if the above advice does not help it is because some λ-abstractions need annotations.

Note that in conditions (2), (3), and (4) we are not circularly assuming type synthesis for sub-expressions of partial erasure t succeeds in order to show that it succeeds for t, only that *if* a certain sub-expression can be typed *then* we can make some assumptions about the shape of its type or elaboration. Conditions (3) and (4) in particular are a direct consequence of a design choice we made for our algorithm to maintain injectivity of meta-variables to omitted type arguments. As an alternative, we could instead refine meta-variables when we know something about the shape of their instantiation. For example, if we encountered a term application whose applicand has a meta-variable type X, we know it must have some arrow type and could refine X to $X_1 \to X_2$, where X_1 and X_2 are fresh meta-variables. However, doing so means type errors may now require non-trivial reasoning from users to determine why some meta-variables were introduced in the first place.

Still, we find it somewhat inelegant that our characterization of annotation requirements for type inference is not fully independent of the inference system itself. For programmers using these guidelines, this implies that there must be some way to interactively query the type-checker for different sub-expressions of a program during debugging. Fortunately, many programming languages offer just such a feature in the form of a REPL, meaning that in practice this is not too onerous a requirement to make.

Theorem 3 only states when an external term will synthesize its type, but what about when a term can be *checked* against a type? It is clear from the typing rules in Figure 1 that some terms that fail to synthesize a type may still be successfully checked against a type. Besides typing bare λ-abstractions (which can only have their type checked), checking mode can also reduce the annotation burden implied by condition (2) of Theorem 3: consider again the example $f\ z$ where f has type $\forall\, X.\, \forall\, Y.\, Y \to X$. If instead of attempting type synthesis we were to check that it has some type T then we would not need to provide an explicit type argument to instantiate X.

From these observations and our next result, we have that checking mode of our type inference system can infer the types of strictly more terms than can synthesizing mode – whenever a term synthesizes a type, it can be checked against the same type.

Theorem 4. *(Checking extends synthesizing):*
If $\Gamma \vdash_{\Uparrow} t : T \rightsquigarrow e$ then $\Gamma \vdash_{\Downarrow} t : T \rightsquigarrow e$

3.2 Examples

Successful Type Inference. We conclude this section with some example programs for which the type inference system in Figures 1 and 2 will and will not be able to type. We start with the motivating example from the introduction of checking that the expression $\texttt{pair}\ (\lambda\, x.\, x)\ z$ has type $\langle(\mathbb{N} \to \mathbb{N}) \times \mathbb{N}\rangle$, which is not possible in other variants of local type inference. For convenience, we assume the existence of a base type $\mathbb{N}$ and a family of base types $\langle S \times T \rangle$ for all types S and T. These assumptions are admissible as we could define these types using lambda encodings. A full derivation for typing this program is given in Figure 3, including the following abbreviations:

$$\begin{aligned}
I_\times\ X\ Y &= X \to Y \to \langle X \times Y \rangle \\
\Gamma &= \texttt{pair}{:}\forall\, X.\, \forall\, Y.\, I_\times\ X\ Y,\ z{:}\mathbb{N} \\
\sigma &= [\mathbb{N} \to \mathbb{N}/X] \\
p &= \texttt{pair}[X][\mathbb{N}]\ (\lambda\, x{:}\mathbb{N}.\, x)\ z
\end{aligned}$$

To type this application $\texttt{pair}\ (\lambda\, x.\, x)\ z$ we first dig through the spine, reach the head pair, and synthesize type $\forall\, X.\, \forall\, Y.\, \mathtt{I}_\times\ X\ Y$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\Uparrow} \mathtt{pair} : \forall X.\, \forall Y.\, I_\times\ X\ Y \rightsquigarrow \mathtt{pair}}\ Var}{\Gamma \vdash^{\mathrm{P}} \mathtt{pair} : \forall X.\, \forall Y.\, I_\times\ X\ Y \rightsquigarrow (\mathtt{pair}, \sigma_{id})}\ PHead \quad \mathcal{D}_1}{\Gamma \vdash^{\mathrm{P}} \mathtt{pair}\ (\lambda\, x.\, x)\ : Y \to \langle X \times \mathbb{N} \rangle \rightsquigarrow (\mathtt{pair}[X][\mathbb{N}]\ (\lambda\, x{:}\mathbb{N}.\, x), \sigma)}\ PApp \quad \mathcal{D}_2}{\Gamma \vdash^{\mathrm{P}} \mathtt{pair}\ (\lambda\, x.\, x)\ z : \langle X \times \mathbb{N} \rangle \rightsquigarrow (p, \sigma)}\ PApp \quad MV(\Gamma, X \times \mathbb{N}) = dom(\sigma)}{\Gamma \vdash^{\mathrm{I}} \mathtt{pair}\ (\lambda\, x.\, x)\ z : \langle X \times \mathbb{N} \rangle \rightsquigarrow (p, \sigma)} \quad MV(\Gamma, p) = dom(\sigma)}{\Gamma \vdash_{\Downarrow} \mathtt{pair}\ (\lambda\, x.\, x)\ z : \langle (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \rangle \rightsquigarrow \mathtt{pair}[\mathbb{N} \to \mathbb{N}][\mathbb{N}]\ (\lambda\, x{:}\mathbb{N}.\, x)\ z}\ AppChk$$

$$\mathcal{D}_1 = \dfrac{\dfrac{\dfrac{MV(\Gamma, \sigma\ X) = \varnothing \quad \dfrac{\dfrac{}{\Gamma, x{:}\mathbb{N} \vdash_{\Downarrow} x : \mathbb{N} \rightsquigarrow x}\ Var}{\Gamma \vdash_{\Downarrow} \lambda\, x.\, x : \mathbb{N} \to \mathbb{N} \rightsquigarrow \lambda\, x{:}\mathbb{N}.\, x}\ Abs}{\Gamma \vdash^{\cdot} (\mathtt{pair}[X][Y]{:}I_\times\ X\ Y, \sigma) \cdot (\lambda\, x.\, x) : Y \to \langle X \times Y \rangle \rightsquigarrow (\mathtt{pair}[X][Y]\ (\lambda\, x{:}\mathbb{N}.\, x), \sigma)}\ PChk}{\Gamma \vdash^{\cdot} (\mathtt{pair}[X]{:}\forall Y.\, I_\times\ X\ Y, \sigma) \cdot (\lambda\, x.\, x) : Y \to \langle X \times Y \rangle \rightsquigarrow (\mathtt{pair}[X][Y]\ (\lambda\, x{:}\mathbb{N}.\, x), \sigma)}\ PForall}{\Gamma \vdash^{\cdot} (\mathtt{pair}{:}\forall X.\, \forall Y.\, I_\times\ X\ Y, \sigma_{id}) \cdot (\lambda\, x.\, x) : Y \to \langle X \times Y \rangle \rightsquigarrow (\mathtt{pair}[X][Y]\ (\lambda\, x{:}\mathbb{N}.\, x), \sigma)}\ PForall$$

$$\mathcal{D}_2 = \dfrac{MV(\Gamma, Y) = \{Y\} \quad \dfrac{}{\Gamma \vdash_{\Uparrow} z : \mathbb{N} \rightsquigarrow z}\ Var}{\Gamma \vdash^{\cdot} (\mathtt{pair}[X][Y]\ (\lambda\, x{:}\mathbb{N}.\, x){:}Y \to \langle X \times Y \rangle, \sigma) \cdot z : \langle X \times \mathbb{N} \rangle \rightsquigarrow (\mathtt{pair}[X][\mathbb{N}]\ (\lambda\, x{:}\mathbb{N}.\, x)\ z, \sigma)}\ PSyn$$

$$\text{where} \quad \begin{array}{rclcrcl} I_\times\ X\ Y & = & X \to Y \to \langle X \times Y \rangle & \quad & \Gamma & = & \mathtt{pair}{:}\forall X.\, \forall Y.\, I_\times\ X\ Y,\ z{:}\mathbb{N} \\ \sigma & = & [\mathbb{N} \to \mathbb{N}/X] & & p & = & \mathtt{pair}[X][\mathbb{N}]\ (\lambda\, x{:}\mathbb{N}.\, x)\ z \end{array}$$

Figure 3: Example typing derivation with the specification rules

No meta-variables are generated by judgment $\vdash_{\Uparrow}$ and thus there can be no meta-variable solutions, so we generate solution σ_{id}.

Next we type the first application, `pair` $(\lambda\, x.\, x)$, shown in sub-derivation $\mathcal{D}_1$. In the first invocation of rule *PForall* we guess solution σ for X, and in the second invocation we decline to guess an instantiation for Y (in this example we could have also guessed $\mathbb{N}$ for Y as this information is also available from the contextual type, but choose not to in order to demonstrate the use of all three rules of $\vdash^{\cdot}$). Then using rule *PChk* we check argument $\lambda\, x.\, x$ against $\sigma\ X = \mathbb{N} \to \mathbb{N}$. *This is the point at which the local type inference systems of [13, 16] will fail*: as a bare λ-abstraction this argument will not synthesize a type, and the expected type X as provided by the applicand `pair` alone does not tell us what the missing type annotation should be. However, by using the information provided by the contextual type of the entire application we know it must have type $\mathbb{N} \to \mathbb{N}$. The resulting partial type of the application is $Y \to \langle X \times Y \rangle$, and we propagate solution σ to the rest of the derivation. Note that we elaborate the argument $\lambda\, x.\, x$ of this application to $\lambda\, x{:}\mathbb{N}.\, x$ – we never pass down meta-variables to term arguments, keeping type-argument inference local to the spine.

In sub-derivation $\mathcal{D}_2$ we type (`pair` $(\lambda\, x.\, x)$) z (parentheses added) where our applicand has partial type $Y \to \langle X \times Y \rangle$. We find that we have unsolved meta-variable Y as the expected type for z, so we use rule *PSyn* and synthesize the type $\mathbb{N}$ for z. Using solution $[\mathbb{N}/Y]$, we produce $\langle X \times \mathbb{N} \rangle$ for the resulting type of the application and elaborate the application to $\mathtt{pair}[X][\mathbb{N}]\ (\lambda\, x{:}\mathbb{N}.\, x)\ z$, wherein type argument Y is replaced by $\mathbb{N}$ in the original elaborated applicand $\mathtt{pair}[X][Y]\ (\lambda\, x{:}\mathbb{N}.\, x)$.

Finally, in rule *AppChk* we confirm that the meta-variables remaining in our partial type synthesis of the application is precisely those for which we knew the solutions contextually. For this example, the only remaining meta-variable in both the partially synthesized type and elaboration is X, which is also the only mapping in σ, so type inference succeeds. We use σ to replace all occurrences of X with $\mathbb{N} \to \mathbb{N}$ in the type and elaboration and conclude that term `pair` $(\lambda\, x.\, x)\ z$ can be checked against type $\langle (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \rangle$.

The next example shows how eager use of *synthetic* type-argument inference can type some terms not possible in other variants of local type inference. Consider checking that the expression `rapp` $x\ \lambda\, y.\, y$ has type $\mathbb{N}$, where `rapp` has type $\forall X.\, \forall Y.\, X \to (X \to Y) \to Y$ and x has type $\mathbb{N}$. From the contextual type we know that Y should be instantiated to $\mathbb{N}$, and when we reach application `rapp` x, we learn that X should be instantiated to $\mathbb{N}$ from the synthesized type of x. Together, this gives us enough information to know that argument $\lambda\, y.\, y$ should have type $\mathbb{N} \to \mathbb{N}$. Such eager instantiation is neither novel, nor is it necessarily desirable when extended to richer type languages or more powerful methods of inference (see Section 5), but in our setting it is a useful technique that we can use to infer types for expressions like the one above.

Type Inference Failures. To see where type inference can fail, we again use `pair` $(\lambda\, x.\, x)\ z$ but now ask that it *synthesize* its type. Rule *AppSyn* insists that we make no guesses for meta-variables (as there is no contextual type for the application that they could have come from), so we would need to synthesize a type for argument $\lambda\, x.\, x$ – but our rules do not permit this! In this case the user can expect an error message like the following:

```
expected type: ?X
        error: We are not in checking mode, so bound
               variable x must be annotated
```

where ?X indicates an unsolved meta-variable corresponding to type variable X in the type of `pair`. The situation above corresponds to condition (1) of Theorem 3: in general, if there is not enough information from the type of an applicand and the contextual type of the application spine in which it occurs to fully know the expected types of arguments that are λ-abstractions, then such arguments require explicit type annotations.

We next look at an example corresponding to condition (2) of Theorem 3, namely that the type variables of a polymorphic function that do not correspond to term arguments in an application should be instantiated explicitly. Here we will assume a family of base types $S + T$ for every type S and T, a variable `right` of type $\forall X. \forall Y. Y \to (X + Y)$, and a variable z of type $\mathbb{N}$. In trying to synthesize a type for the application `right` z the user can expect an error message like:

```
synthesized type: (?X + ℕ)
           error: Unsolved meta-variable ?X
```

indicating that type variable X requires an explicit type argument be provided. Fortunately for the programmer, and unlike the local type inference systems of [13, 16], our system supports partial explicit type application, meaning that X can be instantiated without also explicitly (and redundantly) instantiating Y. On the other hand, these systems are in the setting of System $F_{\leq}$ and can succeed in typing `right` z *without* additional type arguments, as they will instantiate X to the minimal type (with respect to their subtyping relation) `Bot`. Partial type application, then, is more useful for our setting of System F where picking an instantiation in this situation would be arbitrary.

A similar error can occur when synthesizing types for partial (term) applications. For example, the expression `pair` $\lambda x : \mathbb{N}. x$ would raise the message

```
synthesized type: ?Y → ⟨ℕ → ℕ, ?Y⟩
           error: Unsolved meta-variable ?Y
```

As discussed in Section 3.1, our system is artificially sensitive to the order of quantifiers and would require instantiating type variable X to instantiate Y. However, another option is available to users in this case that is *not* available to other forms of local type inference: if this partial application were *checked* against a type like $\mathbb{N} \to (\mathbb{N} \to \mathbb{N}, \mathbb{N})$, then contextual type-argument inference can infer the instantiation for Y and thus the expression would be well-typed.

A more subtle point of failure for our algorithm corresponds to conditions (3) and (4) of Theorem 3. Even when the head and all arguments of an application spine can synthesize their types, the programmer may still be required to provide some additional type arguments. Consider the expression `bot z`, where `bot`:$\forall X. X$ and `z`:$\mathbb{N}$. Even with some contextual type for this expression, type inference still fails because the rules in Figure 2c require that the type of the applicand of a term application reveals some arrow, which $\forall X. X$ does not. The programmer would be met with the following error message:

```
applicand type: ?X
         error: The type of an applicand in a term
                application must reveal an arrow
```

prompting the user to provide an explicit type argument for X. To make expression `bot` z typeable, the programmer could write `bot`$[\mathbb{N} \to \mathbb{N}]\ z$, or even `bot`$[\forall Y. Y \to Y]\ z$ – our inference rules are able to solve meta-variables introduced by explicit (and even synthetically-inferred) type arguments, as long as there is enough information to reveal a quantifier or arrow in the type of a term or type applicand.

For our last type error example, we consider the situation where the programmer has written an ill-typed program. Local type inference enjoys the property that type errors can be understood *locally*, without any "spooky action" from a distant part of the program. In particular, with local type inference we would like to avoid error messages like the following:

```
synthesized type: 𝔹 → 𝔹
   expected type: ?X := ℕ → ℕ
           error: type mismatch
```

From this error message alone the programmer has no indication of why the expected type is $\mathbb{N} \to \mathbb{N}$! In our type system we have expanded the distance information travels by allowing it to flow from the contextual type of an application to its arguments – so can programmers expect now to see such error messages?. As an example, the error message above could have been generated when checking the expression `pair` $(\lambda x : \mathbb{B}. x)\ z$ against type $\langle(\mathbb{N} \to \mathbb{N}) \times \mathbb{N}\rangle$, specifically when inferring the type of the first argument. Fortunately, our notion of locality is still quite small and we can easily demystify the type error:

```
synthesized type: 𝔹 → 𝔹
   expected type: ?X := ℕ → ℕ
contextual match: ⟨?X × ?Y⟩ := ⟨(ℕ → ℕ) × ℕ⟩
```

where `contextual match` tells the programmer to compare the partially synthesized and contextual return types of the application spine to determine why X was instantiated to $\mathbb{N} \to \mathbb{N}$. A similar field, `synthetic match`, could tell the programmer that the type of an earlier argument informs the expected type of current one.

4 ALGORITHMIC INFERENCE RULES

The type inference system presented in Section 3 do not constitute an algorithm. Though the rules forming judgment $\vdash^{\cdot}$ indicate *where* and *how* we use contextually-inferred type arguments, they do not specify *what* their instantiations are or even *whether* this information is available to use, and it is not obvious how to work backwards from the second premise in Figure 2a to develop an algorithm.

Figure 4 shows the algorithmic rules implementing contextual type-argument inference. The full algorithm for spine-local type inference, then, consists of the rules in Figure 1 with the shim judgment $\vdash^{\mathrm{I}}$ as defined in Figure 4a. At the heart of the implementation is our prototype matching algorithm; to understand the details of how we implement contextual type-argument inference, we must first discuss this algorithm and the two new syntactic categories it introduces, prototypes and decorated types.

4.1 Prototype Matching

Figure 4d lists the rules for the prototype matching algorithm. We read the judgment $\overline{X} \Vdash^{:=} T := P \Rightarrow (\sigma, W)$ as: "solving for meta-variables $\overline{X}$, we match type T to prototype P and generate solution σ and decorated type W," and we maintain the invariant that $dom(\sigma) \subseteq \overline{X}$. Meta-variables can only occur in T, thus these are

(a) Shim (algorithm)

$$\frac{\Gamma; T_? \Vdash^? t\ t' : T \rightsquigarrow (p, \sigma) \quad T_? \in \{?, \sigma\ T\}}{\Gamma; T_? \vdash^! t\ t' : T \rightsquigarrow (p, \sigma)}$$

(b) $\boxed{\Gamma; P \Vdash^? t : W \rightsquigarrow (p, \sigma)}$

$$\frac{\neg App(t) \quad \Gamma \Vdash_{\Uparrow} t : T \rightsquigarrow e \quad \varnothing \Vdash^{:=} T := ? \to P \Rightarrow (\sigma_{id}, W)}{\Gamma; ? \to P \Vdash^? t : W \rightsquigarrow (e, \sigma_{id})}\ ?Head \qquad \frac{\Gamma; ? \to P \Vdash^? t : \forall X{=}R.\ W \rightsquigarrow (p, \sigma) \quad R \in \{X, S\}}{\Gamma; ? \to P \Vdash^? t[S] : [S/X]W \rightsquigarrow (p[S], \sigma)}\ ?TApp$$

$$\frac{\Gamma; ? \to P \Vdash^? t : W \rightsquigarrow (p, \sigma) \quad \Gamma \Vdash^{\cdot} (p{:}W, \sigma) \cdot t' : W' \rightsquigarrow (p', \sigma')}{\Gamma; P \Vdash^? t\ t' : W' \rightsquigarrow (p', \sigma')}\ ?App$$

(c) $\boxed{\Gamma \Vdash^{\cdot} (p{:}W, \sigma) \cdot t' : W \rightsquigarrow (p', \sigma')}$

$$\frac{\sigma'' = \text{ if } R{=}X \text{ then } \sigma \text{ else } [R/X]{\circ}\sigma \quad \Gamma \Vdash^{\cdot} (p[X]{:}W, \sigma'') \cdot t' : W' \rightsquigarrow (p', \sigma')}{\Gamma \Vdash^{\cdot} (p{:}\forall X{=}R.\ W, \sigma) \cdot t' : W' \rightsquigarrow (p', \sigma')}\ ?Forall \qquad \frac{MV(\Gamma, \sigma\ S) = \varnothing \quad \Gamma \Vdash_{\Downarrow} t' : S \rightsquigarrow e}{\Gamma \Vdash^{\cdot} (p{:}S \to W, \sigma) \cdot t' : W \rightsquigarrow (p\ e', \sigma)}\ ?Chk$$

$$\frac{MV(\Gamma, \sigma\ S) = \overline{Y} \neq \varnothing \quad \Gamma \Vdash_{\Uparrow} t : \overline{[U/Y]}\ \sigma\ S \rightsquigarrow e}{\Gamma \Vdash^{\cdot} (p{:}S \to W) \cdot t' : \overline{[U/Y]}\ W \rightsquigarrow ((\overline{[U/Y]}\ p)\ e', \sigma)}\ ?Syn$$

(d) $\boxed{\overline{X} \Vdash^{:=} T := P \Rightarrow (\sigma, W)}$

$$\frac{\overline{X} \Vdash^{:=} T := P \Rightarrow (\sigma, W)}{\overline{X} \Vdash^{:=} S \to T := ? \to P \Rightarrow (\sigma, S \to W)}\ MArr \qquad \frac{\overline{X}, X \Vdash^{:=} T := ? \to P \Rightarrow (\sigma, W)}{\overline{X} \Vdash^{:=} \forall X.\ T := ? \to P \Rightarrow (\sigma - X, \forall X{=}\sigma(X).\ W)}\ MForall$$

$$\frac{\overline{Y} = FV(T) \cap \overline{X} \quad FV(\overline{U}) \cap (BTV(S) \cup \overline{X}) = \varnothing \quad \overline{[U/Y]}\ T = S}{\overline{X} \Vdash^{:=} T := S \Rightarrow (\overline{[U/Y]}, T)}\ MType \qquad \frac{}{\overline{X} \Vdash^{:=} T := ? \Rightarrow (\sigma_{id}, T)}\ M? \qquad \frac{X \in \overline{X}}{\overline{X} \Vdash^{:=} X := ? \to P \Rightarrow (\sigma_{id}, (X, ? \to P))}\ MCurr$$

Figure 4: Algorithm for contextual type argument inference

matching (not unification) rules. The grammar for prototypes and decorated types is given below:

$$\begin{array}{rl} \textbf{Prototypes} & P ::= ? \mid T \mid ? \to P \\ \textbf{Decorated Types} & W ::= T \mid S \to W \mid \forall X{=}X.\ W \mid \forall X{=}S.\ W \\ & \quad \mid (X, ? \to P) \end{array}$$

Prototypes carry the contextual type of the maximal application of a spine. In the base case they are either the uninformative ? (as in *AppSyn*), indicating no contextual type, or they are informative of type T (as in *AppChk*). In this way, prototypes generalize the syntactic category $T_?$ we introduced earlier for optional contextual types. We use the last prototype former $? \to$ as we work our way down an application spine to track the expected arity of its head. For example, if we wished to check that the expression `id suc x` has type $\mathbb{N}$, then when we reached the head `id` using the rules in Figure 4b we would generate for it prototype $? \to ? \to \mathbb{N}$

Decorated types consist of types (also called *plain-decorated* types), an arrow with a regular type as the domain (as prototypes only carry the result type of a maximal application, not of the types of arguments), quantified types whose bound variable X may be decorated with the type to which we expect to instantiate it, and "stuck" decorations. On quantifiers, decoration $X = X$ indicates that P did not inform us of an instantiation for X – we sometimes abbreviate the two cases as $\forall X{=}R.\ W$, where $R \in \{X, S\}$ and $S \neq X$. Finally, we define operation $\lfloor W \rfloor$ erasing decorations from W and producing a type in the expected way, $\lfloor \forall X{=}R.\ W \rfloor = \forall X.\ \lfloor W \rfloor$ and $\lfloor (X, ? \to P) \rfloor = X$

To explain the role of stuck decorations, consider again `id suc x`. Assuming `id` has type $\forall X.\ X \to X$, matching this with prototype $? \to ? \to \mathbb{N}$ generates decorated type $\forall X = X.\ X \to (X, ? \to \mathbb{N})$, meaning that we only know that X will be instantiated to some type that matches $? \to \mathbb{N}$. Stuck decorations occur when the expected arity of a spine head (as tracked by a given prototype) is greater than the arity of the head's synthesized type and are the mechanism by which we propagate a contextual type to a head that is "over-applied" – a not-uncommon occurrence in languages with curried applications!

Turning to the prototype matching algorithm in Figure 4d, rule *MArr* says that we match an arrow type and prototype when we can match their codomains. Rule *MType* says that when the prototype is some type S we must find an instantiation $\overline{[U/Y]}$ (where $\overline{Y} \subseteq \overline{X}$) such that $\overline{[U/Y]}\ T = S$, and rule *M?* says that any type matches with ? with no solutions generated (thus we call ? "uninformative"). In rule *MForall* we match a quantified type with a prototype by

adding bound variable X to our meta-variables and matching the body T to the same prototype; the substitution in the conclusion, $\sigma - X$, is the solution generated from this match less its mapping for X, which is placed in the decoration $X = \sigma(X)$. For example, matching $\forall X.\forall Y.\, X \to Y \to X$ with prototype $? \to ? \to \mathbb{N}$ generates decorated type $\forall X{=}\mathbb{N}.\forall Y{=}Y.\, X \to Y \to X$. Finally, rule *MCurr* applies when there is incomplete information (in the form of $? \to P$) on how to instantiate a meta-variable; we generate a stuck decoration with identity solution σ_{id}.

We conclude by showing that our prototype matching rules are *functional*; when $\overline{X}$, T, and P are considered as inputs then there is at most one output pair (σ, W):

Theorem 5. *(Function-ness of $\Vdash^{:=}$):*
Given $\overline{X}$, T, and P, if $\overline{X} \Vdash^{:=} T := P \Rightarrow (\sigma, W)$
and $\overline{X} \Vdash^{:=} T := P \Rightarrow (\sigma', W')$, then $\sigma = \sigma'$ and $W = W'$

It should also be clear that these rules are *syntax-directed* (i.e. that the rules are non-overlapping and that the inputs to premises and the outputs to the conclusions are uniquely determined), meaning they can be straightforwardly translated to an algorithm.

4.2 Decorated Type Inference

We now discuss the rules in Figures 4b and 4c which implement contextual type-argument inference (as specified by Figures 2b and 2c) by using the prototype matching algorithm. We begin by giving a reading for judgments $\Vdash^{?}$ – read $\Gamma; P \Vdash^{?} t : W \rightsquigarrow (p, \sigma)$ as: "under context Γ and with prototype P, t synthesizes decorated type W and elaborates p with solution σ," where σ again represents the contextually-inferred type arguments.

In rule *AppSyn* we required that the solution generated by $\vdash^{\mathrm{I}}$ in its premise is σ_{id}; in *AppChk* we (implicitly) required that the contextual type is equal to $\sigma\ T$; and now with the algorithmic definition for $\vdash^{\mathrm{I}}$ we appear to be requiring in both that the decorated type generated by $\Vdash^{?}$ is a plain-decorated type T. With the algorithmic rules, these are not requirements but *guarantees* that the specification makes of the algorithm:

Lemma 1. *Let $arr_P(P)$ be the number of prototype arrows prefixing P and $arr_W(W)$ be the number of decorated arrows prefixing W. If $\Gamma; P \Vdash^{?} t : W \rightsquigarrow (p, \sigma)$ then $arr_W(W) \leq arr_P(P)$*

Theorem 6. *(Soundness of $\Vdash^{?}$ wrt $\Vdash^{:=}$): If $\Gamma; P \Vdash^{?} t : W \rightsquigarrow (p, \sigma)$ then $MV(\Gamma, p) \Vdash^{:=} \lfloor W \rfloor := P \Rightarrow (\sigma, W)$*

Assuming prototype inference succeeds, when we specialize P in Theorem 6 to $?$ we have immediately by rule $M?$ that $\sigma = \sigma_{id}$; when we specialize it to some contextual type T' for an application, then by the premise of *MType* we have $\sigma\ T = T'$. Theorem 1 and 6 together tell us that we generate plain-decorated types in both cases, as in particular we cannot have leading (decorated) arrows or stuck decorations with prototypes $?$ or T'.

Next we discuss the rules forming judgment $\Vdash^{?}$ in Figure 4b, constituting the algorithmic version of the rules in Figure 2b. In rule *?Head*, after synthesizing a type T for the application head we match this type against expected prototype $? \to P$ (we are guaranteed the prototype has this shape since a maximal term application begins all derivations of $\Vdash^{?}$). No meta-variables occur in T initially – as we perform prototype matching these will be generated by rule *MForall* from quantified type variables in T and their solutions will be left as quantifier decorations in the resulting decorated type W. We are justified in requiring that matching T to $? \to P$ generates empty solution σ_{id} since we have in general that the meta-variables solved by our prototype matching judgment are a subset of the meta-variables it was asked to solve:

Lemma 2. *If $\overline{X} \Vdash^{:=} T := P \Rightarrow (\sigma, W)$ then $dom(\sigma) \subseteq \overline{X}$*

In *?TApp*, we can infer the type of a type application $t[S]$ when t synthesizes a decorated type $\forall X{=}R.\, W$ and R is either an uninformative decoration X or is precisely S (that is, the programmer provided explicitly the type argument the algorithm contextually inferred). We synthesize $[S/X]W$ for the type application, where we extend type substitution to decorated types by the following recursive partial function:

$$\sigma\ S \to W = (\sigma\ S) \to (\sigma\ W)$$
$$\sigma\ \forall X{=}R.\, W = \forall X{=}R.\, \sigma\ W$$
$$\sigma\ (X, ? \to P) = W \qquad \text{if } \varnothing \Vdash^{:=} \sigma(X) :=? \to P \Rightarrow (\sigma_{id}, W)$$

This definition is straightforward except for the case dealing with stuck decorations. Here, σ (representing type arguments given explicitly or inferred synthetically) may provide information on how to instantiate X and this must match our current (though incomplete) information $? \to P$ about this instantiation. For example, if we have decorated type $W = X \to (X, ? \to \mathbb{N})$, then $[\mathbb{N} \to \mathbb{N}/X]\ W$ would require we match $\mathbb{N} \to \mathbb{N}$ with $? \to \mathbb{N}$ and matching would generate (plain) decorated type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$

The definition of substitution on decorated types is partial since prototype matching may fail (consider if we used substitution $[\mathbb{N}/X]$ in the above example instead). When a decorated type substitution $\sigma\ W$ appears in the conclusion of our algorithmic rules, such as in *?TApp* or *?Syn*, we are implicitly assuming an additional premise that the result is defined.

The last rule for judgment $\Vdash^{?}$ is *?App*, and like *PApp* it benefits from a reading for judgment $\Vdash^{\cdot}$ occurring in its premise. We read $\Gamma \Vdash^{\cdot} (p : W, \sigma) \cdot t' : W' \rightsquigarrow (p', W')$ as: "under Γ, elaborated applicand p of decorated type W together with solution σ can be applied to t'; the application has decorated type W' and elaborates p' with solution σ'." Thus, *?App* says that to synthesize a decorated type for a term application $t\ t'$ we synthesize the decorated type of the applicand t and ensure that the resulting elaboration p, along with its decorated type and solution, can be applied to t'.

We now turn to the rules for the last judgment $\Vdash^{\cdot}$ of our algorithm. Rule *?Forall* clarifies the non-deterministic guessing done by the specificational rule *PForall*: the contextually-inferred type arguments we build during contextual type-argument inference are just the accumulation of quantified type decorations. The solution σ'' we provide to the second premise of *?Forall* contains mapping $[R/X]$ if R is an informative decoration, and as we did in rule *PForall* we provide elaborated term $p[X]$ to track the contextually-inferred type arguments separately from those synthetically inferred.

Rule *?Chk* works similarly to *PChk*: when the only meta-variables in the domain S of our decorated type are solved by σ, we can check that argument t' has type $\sigma\ S$. In rule *?Syn* we have some meta-variables $\overline{Y}$ in S not solved by σ – we synthesize a type for the

argument, ensure that it is some instantiation $\overline{[U/Y]}$ of $\sigma\ S$, and use this instantiation on the meta-variables in p as well as the decorated codomain type W, potentially unlocking some stuck decoration to reveal more arrows or decorated type quantifications.

We conclude this section by noting that the specificational and algorithmic type inference system are equivalent, in the sense that they type precisely the same set of terms:

Theorem 7. *(Soundness of $\Vdash_\delta$ wrt $\vdash_\delta$):*
If $\Gamma \Vdash_\delta t : T \rightsquigarrow e$ *then* $\Gamma \vdash_\delta t : T \rightsquigarrow e$

Theorem 8. *(Completeness of $\Vdash_\delta$ wrt $\vdash_\delta$):*
If $\Gamma \vdash_\delta t : T \rightsquigarrow e$ *then* $\Gamma \Vdash_\delta t : T \rightsquigarrow e$

(where $\Vdash_\delta$ indicates $\vdash^{\mathrm{I}}$ is defined as in Figure 4a), and that the algorithmic rules are decidable:

Theorem 9. *(Decidability of Type-checking)*

- *For any context* Γ *and term* t*, it is decidable whether* $\Gamma \Vdash_{\Uparrow} t : T \rightsquigarrow e$ *for some* T *and* e
- *For any context* Γ*, term* t*, and type* T*, it is decidable whether* $\Gamma \Vdash_{\Downarrow} t : T \rightsquigarrow e$ *for some* e

Taken together, Theorems 7 and 8 justify our claim that the rules of Figure 2 constitute a specification for contextual type-argument inference – it is not necessary that the programmer know the notably more complex details of prototype matching or type decoration to understand how some type arguments are inferred contextually. Indeed, the judgment $\vdash^{\cdot}$ provides more flexibility in reasoning about type inference than does $\Vdash^{\cdot}$, as in rule *PForall* we may freely decline to guess a contextual type argument even when this would be justified and instead try to learn it synthetically. In contrast, algorithmic rule ?*Forall* requires that we use any informative quantifier decoration. We use this flexibility when giving guidelines for the required annotations in Section 3.1 for typing external terms, as the required conditions for typeability in Theorem 3 would be further complicated if we could not restrict ourselves to using only synthetic type-argument inference.

5 DISCUSSION & RELATED WORK

5.1 Local Type Inference and System $\mathrm{F}_{\leq}$

Local Type Inference. Our work is most influenced by the seminal paper by Pierce and Turner[16] that lays out the approach of local type inference systems, including bidirectional typing rules, local type-argument inference, and the design-space restriction that polymorphic function applications be fully-uncurried to maximize the benefit of these techniques. In their system, either all term arguments to polymorphic functions must be synthesized or else all type arguments must be given – no compromise is available when only a *few* type arguments suffice to type an application, be they provided explicitly or inferred contextually. Our primary motivation in this work was addressing these issues – improving support for first-class currying and partial type application, and using the contextual type of an application for type-argument inference – while maintaining some of the desirable properties of local type inference and staying in the spirit of their approach.

Colored Local Type Inference. Odersky, Zenger, and Zenger[13] extend the type system of Pierce and Turner by allowing *partial* type information to be propagated when inferring types for term arguments. Their insight was to internalize the two modes of bidirectional type inference to the very structure of types, allowing different parts to be synthetic or contextual. In contrast, we use an "all or nothing" approach to type propagation, requring a term argument to fully synthesize its type when we have incomplete information. On the other hand, their system uses only the typing information provided by the application head, whereas we combine this with the contextual type of an application, allowing us to type some expressions their system cannot.

The syntax for prototypes in our algorithm was directly inspired by the prototypes used in the algorithmic inference rules for [13]. Our use of prototypes complements theirs; ours propagates the partial type information provided by contextual type of an application spine to its head, whereas theirs propagates the partial type information provided by an application head to its arguments. In future work, we hope to combine these two notions of prototype to propagate *partially* the type information coming from the application's contextual type *and* head to its arguments.

Subtyping. Local type inference is usually studied in the setting of System $\mathrm{F}_{\leq}$ which combines impredicative parametric polymorphism and subtyping. The reason for this is two-fold: first, a partial type inference technique is needed as complete type inference for $\mathrm{F}_{\leq}$ is undecidable[19]; second, global type inference systems fail to infer principal types in $\mathrm{F}_{\leq}$ [10, 12], whereas local type inference is able to promise that it infers the "locally best"[16] type arguments. The setting for our algorithm is System F, so the reader may ask whether our developments can be extended gracefully to handle subtyping. We believe the answer is yes, though with some modification on how synthetic type arguments are used.

In rule *PSyn* in Figure 2c, meta-variables $\overline{Y}$ are instantiated to types $\overline{U}$ immediately. In the presence of subtyping this would make our rules *greedy*[1, 4] and we would not be able to guarantee synthetic type-argument inference produced locally best types, possibly causing type inference to fail later in the application spine. To illustrate this, consider the expression $\texttt{rapp}\ x\ \texttt{neg}$, assuming $\texttt{rapp} : \forall X.\forall Y.X \to (X \to Y) \to Y$, $x : \mathbb{N}$, $\texttt{neg} : \mathbb{Z} \to \mathbb{Z}$, and some subtyping relation $\leq$ where $\mathbb{N} \leq \mathbb{Z}$. Greed causes us to instantiate X with $\mathbb{N}$, but in order to type the expression we would need to instantiate it to $\mathbb{Z}$ instead!

To correct this, we could instead collect these constraints and solve them only when the function is fully applied to its arguments (i.e., when we reach a stuck decoration). This mirrors the requirement in [16] that constraints are solved at fully uncurried applications, maintaining currying but losing a syntactically-obvious location for synthetic type-argument inference.

We would also need to justify our use of contextual type-argument inference for checking the types of term arguments. Happily, this does not appear to be an intractable problem like greed: unlike in synthesis mode, checking mode for applications in [16] does not require that the synthesized type arguments minimize the result type of the application, so there is greater freedom in choosing the instantiations for contextually-inferred type arguments. Hosoya and Pierce note in [7] that the optimal instantiations for these type arguments are ones that "maximize the expected type corresponding to the [argument]," as the type that the programmer meant for

the argument (if type correct) will be a subtype of this. Though the informal approach they proposed (and later dismissed) for inferring the types of hard-to-synthesize terms differs from ours in the use of a "slightly ad-hoc" analysis of arguments, it anticipated contextual type-argument inference and suggests the way forward for using it in the presence of subtyping.

5.2 Bidirectional Type Inference and System F

Predicative Polymorphism. Bidirectional type inference sees use outside of purely local systems. Dunfield and Krishnaswami[5] introduced a simple and elegant type inference system for predicative System F using a dedicated application judgment that instantiates type arguments at term applications. Their application judgment was the direct inspiration for our own, though there are some significant differences between the two. First, our rules distinguish between checking the argument of an application with a fully known expected type and synthesizing its argument when incomplete information is available to keep meta-variables *spine-local*, whereas in their approach meta-variables and typing constraints are passed downwards to check term arguments. Our system also contains the additional judgment form $\vdash^{\mathrm{P}}$ that theirs does not, again to contain meta-variables within an application spine.

Approaches to type inference for System F (impredicative and predicative alike) often make use of some form of subsumption rule to decrease the required type annotations in terms. A popular basis for such rules is the "more polymorphic than" subtyping relation introduced by Odersky and Läufer in [11] which stratifies polymorphic and monomorphic types and is able to perform deeply nested monomorphic type instantiation. This line of work also includes [5] above as well as work by Peyton Jones et. al. [14], both of which are able to infer arbitrary-rank types in the setting of predicative System F. In contrast our type inference algorithm supports more powerful impredicative polymorphism at the cost of significant increase in required type annotations.

Impredicative Polymorphism. The "more-polymorphic-than" subtyping relation for impredicative System F is undecidable[19], so type inference systems wishing to use a subsumption rule in this setting must make some compromises. With $\mathbf{ML}^{\mathbf{F}}$ [8] Le Botlan and Rémy develop a type language with bounded type quantification and an inference system using type *instantiation* (a covariant restriction of subtyping). *Boxy type inference*[21] by Vytiniotis et al. uses an idea similar to [13] of propagating partial type information (though with a very different implementation) to allow inference for polymorphic types only in checking mode; its later development in **FPH**[20] both simplifies the specification for type inference and extends boxy types to synthesis mode to allow "boxy monotypes" to be inferred for polymorphic functions. These inference systems add additional constructs (resp. bounded quantifications and boxy types) to System F types in their specification, whereas we reserve our new constructs (decorated types and prototypes) for the algorithmic rules only. Our use of first-order matching when typing applications of and arguments to polymorphic functions can be viewed as a crude form of subtyping via shallow type instantiation – it is significantly easier for programmers to understand but at the same time significantly less powerful than the subtyping used in the type inference systems above.

ACKNOWLEDGMENTS

We thank Larry Diehl, Anthony Cantor, and Ernesto Copello for their feedback on earlier versions of this paper which helped us clarify some points of terminology and improve the readability of the more technical sections of the paper. We gratefully acknowledge NSF support under award 1524519, and DoD support under award FA9550-16-1-0082 (MURI program).

REFERENCES

[1] Luca Cardelli. 1997. An implementation of F<:. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.6158
[2] Iliano Cervesato and Frank Pfenning. 2003. A Linear Spine Calculus. *J. Log. Comput.* 13 (2003), 639–688.
[3] Luis Damas and Robin Milner. 1982. Principal Type-schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*. ACM, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176
[4] Joshua Dunfield. 2009. Greedy Bidirectional Polymorphism. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML (ML '09)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/1596627.1596631
[5] Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-rank Polymorphism. *SIGPLAN Not.* 48, 9 (Sept. 2013), 429–442. https://doi.org/10.1145/2544174.2500582
[6] Jean-Yves Girard. 1986. The system F of variable types, fifteen years later. *Theoretical Computer Science* 45 (1986), 159 – 192. https://doi.org/10.1016/0304-3975(86)90044-7
[7] Haruo Hosoya and Benjamin Pierce. 1999. How Good is Local Type Inference? (07 1999).
[8] Didier Le Botlan and Didier Rémy. 2003. MLF: Raising ML to the Power of System F. *SIGPLAN Not.* 38, 9 (Aug. 2003), 27–38. https://doi.org/10.1145/944746.944709
[9] Bruce McAdam. 2002. Trends in Functional Programming. Intellect Books, Exeter, UK, UK, Chapter How to Repair Type Errors Automatically, 87–98. http://dl.acm.org/citation.cfm?id=644403.644412
[10] Martin Odersky. 2002. Inferred Type Instantiation for GJ. Note sent to the types mailing list.
[11] Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, New York, NY, USA, 54–67. https://doi.org/10.1145/237721.237729
[12] Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *Theor. Pract. Object Syst.* 5, 1 (Jan. 1999), 35–55. https://doi.org/10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4
[13] Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored Local Type Inference. *SIGPLAN Not.* 36, 3 (Jan. 2001), 41–53. https://doi.org/10.1145/373243.360207
[14] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2005. Practical type inference for arbitrary-rank types. 17 (January 2005), 1–82. https://www.microsoft.com/en-us/research/publication/practical-type-inference-for-arbitrary-rank-types/
[15] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.
[16] Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44. https://doi.org/10.1145/345099.345100
[17] Hubert Plociniczak. 2016. *Decrypting Local Type Inference.* Ph.D. Dissertation. École polytechnique fédérale de Lausanne. http://dx.doi.org/10.5075/epfl-thesis-6741
[18] Aaron Stump. 2017. The calculus of dependent lambda eliminations. *J. Funct. Program.* 27 (2017), e14. https://doi.org/10.1017/S0956796817000053
[19] J. Tiuryn and P. Urzyczyn. 1996. The subtyping problem for second-order types is undecidable. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. 74–85. https://doi.org/10.1109/LICS.1996.561306
[20] Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2008. FPH: First-class polymorphism for Haskell, In Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. https://www.microsoft.com/en-us/research/publication/fph-first-class-polymorphism-for-haskell/
[21] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2006. Boxy Types: Inference for Higher-rank Types and Impredicativity. *SIGPLAN Not.* 41, 9 (Sept. 2006), 251–262. https://doi.org/10.1145/1160074.1159838
[22] J. B. Wells. 1998. Typability and Type Checking in System F Are Equivalent and Undecidable. *ANNALS OF PURE AND APPLIED LOGIC* 98 (1998), 111–156.
[23] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560