THE UNIVERSITY OF IOWA

Bidirectional Type Inference in Programming Languages

Submitted in Partial Fulfillment of the Requirements for the Qualifying Exam of the Ph.D. Program
University of Iowa, Department of Computer Science

Christopher Jenkins

2018 September 3

0 Abstract

Type inference in programming languages helps to maintain the balance between powerful features and user-friendliness. *Bidirectional* type inference is a simple, effective, and widely-used technique in approaches to *partial* type inference that *locally* propagates type information between nodes of the abstract syntax tree (AST) through two main judgment forms, called *synthesis* and *checking* mode. By itself, bidirectional type inference serves as a strong foundation for *local type inference* systems, an approach that benefits from simplicity in use and implementation while still being powerful enough to infer a good number of type annotations; in conjunction with other techniques for type inference, bidirectionality can be used to help relate the type inference implementation to an independent specification that can better communicate where type annotations are required, where they can be omitted, and what types are inferred, all without getting into implementation details. This report provides an introduction to the concept of bidirectional type inference and gives an in-depth analysis of how it is used in three published type systems to help provide a specification for sophisticated type inference techniques.

1 Introduction

In programming languages, types serve as a kind of machined-checked specification for programs and data, indicating their "shape" and what kinds of operations can be performed on them – `integer`s can be added, `string`s can be concatenated, etc. The two main approaches to type checking, *static* and *dynamic*, differ in when this specification is enforced, with the latter making violations of this specification run-time errors and the former introducing a separate type-checking phase before run-time. Proponents of static typing point out that it can guarantee software is free from certain classes of errors, whereas testing can only confirm the *existence* of errors. For languages such as C with simple, weakly-enforced static type systems, the kinds of specifications programmers can write – and thus, bugs they can rule out – are not very interesting. However, types in programming languages lie on a continuum of expressivity, and languages with richer type systems such as Haskell, ML, and Scala are popular in part because of their ability to statically enforce non-trivial behavioral guarantees for programs. At the extreme end, due to the *Curry-Howard correspondence* [Cur34] we can view types as propositions and terms as proofs, and languages such as Agda, Idris, and Coq use this to blur the lines between general-purpose functional programming language and interactive theorem prover!

This increase in expressivity comes at a cost, however. Detractors of static type checking are quick to point out that languages with static typing often require a great deal of help from the programmer in the form of *type annotations*. Type annotations are not necessarily a bad thing – at their best, they serve as formal documentation of the programmer's intent for an expression – but writing them quickly becomes tedious when it is "obvious" what type each sub-expression of a program should have. *Type inference*, which is the automatic reconstruction of typing information from programs, is the key technology which helps to reduce the programmer's annotation burden. However, there is an inherent conflict between advanced type-level features allowing programmers to impose precise specifications and, on the other hand, expecting type inference algorithms to *reconstruct* these advanced specifications. Additionally, with more sophisticated type inference algorithms, it becomes increasingly difficult for programmers to understand why their code has a type error, where they should provide additional annotations, or even why a specific type, and not another, was inferred for an expression!

This report is concerned with *bidirectional type inference*, an approach to type inference that helps reduce the number of redundant and silly type annotations while at the same time being amenable to providing a high-level, specificational description of the implementation suitable for communicating to programmers. The design space for type inference algorithms is just as large (if not more so) as that for type systems themselves, so to help orient the reader to where bidirectional type inference sits within it, I begin with categorizing approaches to type inference by asking the following two high-level questions: does the algorithm perform *complete* or *partial* type inference, and does it rely on *global* or purely *local* type information?

Complete vs Partial Methods Some programming languages have type systems simple enough to admit complete type inference, meaning that if a program is typeable at all, it is typeable *without* any typing annotations. Of such methods, the most popular are variants of Damas-Hindley-Milner [DM82] type inference, most often used in functional programming languages with (restricted) parametric polymorphism like ML and Haskell. For such languages, specifying the type annotation requirements is trivial – nowhere! – and inference of "principal" (i.e. most general) types answers *which* types will be inferred when multiple might work.

Complete type inference is possible only in languages where the structure of types is somehow restricted. In contrast to the predicative, prenex polymorphism of ML, it is well-known that complete type inference for the "gold-standard" of expressivity in functional languages, impredicative System F, is undecidable [Wel98]. Languages aspiring to approach or exceed the power of System F must therefore rely on *partial* methods of type inference; for such languages, some type annotations will be unavoidable, and the goal of type inference is to make the annotation requirements and the types that the algorithm infers *predictable* and *sensible*.

Global vs Local Methods Global methods of type inference work by first generating typing constraints for the whole program and then attempting to solve these constraints, usually using some form of *unification* algorithm. Because they have access to all available typing information, global methods do a very good job at minimizing annotation requirements. However, when type errors *do* occur they can be more difficult for programmers to interpret, as the root cause of the error can be distant from the location it is reported [Mca00]. *Local* methods of type inference, on the other hand, work by *only* propagating typing information between adjacent nodes of the AST and restricting the generation and solution of typing constraints to a well-defined locality (typically a node of the AST). What local methods lack in power they gain in comprehensibility and improved localization of error messages. Unlike the divide between complete and partial methods, type inference systems can combine local and global methods as desired.

Bidirectional Type Inference sits at the intersection of local methods used for partial type inference: *local* because it describes the flow of typing information between adjacent nodes of the language AST, and *partial* because complete methods usually need not concern themselves with this flow. Systems using bidirectional type inference use two main judgment forms, usually called *synthesizing* and *checking* mode. Synthesizing mode is used for "pulling" type information out of a term and making it available to the surrounding context, and checking mode is used for "pushing" typing information available from the context into a term. This may seem like a rather simple technique – and indeed, it is! – but it is surprisingly useful for designing and specifying type inference algorithms, partly because it can bring the specification a little closer to the implementation and make it easier to show the two are equivalent, and partly because the two modes significantly affect what and how types are inferred.

In this report I will consider several type inference systems, what the goals of these systems are, and how they use bidirectionallity to effectively specify their inference algorithm. Section 2 provides background on how to read type inference rules and some simple properties and advantages of bidirectional systems. Section 3 presents the type system of "Local Type Inference" [PT00], Pierce and Turner's seminal work which first formally introduced bidirectional type-checking, and explores the interplay between bidirectional inference, subtyping, and *type-argument* inference. Section 4 shows a refinement of the previous type system called "Colored Local Type Inference" [OZZ01] by Odersky et al. internalizing bidirectionality into the structure of types themselves, allowing for the local propagation of *partial* type information. Finally, Section 5 departs from local-only methods to examine the type system by Dunfield and Krishnaswami in "Complete and Easy Bidirectional Typechecking for Higher-rank Polymorphism" [DK13], which adds a third judgment form for type-argument inference to the usual bidirectional mix. All three systems have bidirectional *specificational* rules with respect to which the implementations are proven sound and complete.

2 Bidirectional Type Inference

2.1 System F-sub

In this section I give some necessary background for reading type inference rules. For the remainder of this report, a least some familiarity with System F is assumed[1]. Figure 1 gives the grammar, subtyping, and type inference rules for System $F_{\leq}$, an extension of System F that adds subtyping. Type systems are often given as a set of *inference rules*, where premises and assumptions go above the horizontal line and conclusions below. Typing derivations form bottom-up "proof trees" similar to those seen in natural deduction. The advantage of describing them in this way, rather than through pseudo-code, is that inference rules allow for a higher-level view of the system that users and implementors of the system alike can understand. Figure 1 contains only "uni-directional" (synthesizing-only) inference rules so that, at a first pass, we can focus only on how to read them. After giving a brief overview of this system I will discuss its limitations and motivate making the system bidirectional (Figure 2).

Grammar In $F_{\leq}$ constants $\top$ (pronounced "top") and $\bot$ ("bot") are added to the language of System F types, representing resp. the universal and uninhabited types. Intuitively, any typeable term can be used when a term of type $\top$ is needed, and if you have a term of type $\bot$ it can be used as if it were any type. Besides this, the most significant deviation from System F is the coupling of type abstractions with arrow types and the use of *uncurried* function types, though these are more of a stylistic rather than a fundamental departure. Terms are formed by variables, function abstractions $\texttt{fun}[\overline{X}](\overline{x} : \overline{S})t$ (with $\overline{X}$ the abstracted type variables, $\overline{x}$ the

[1] For a thorough introduction of the subject see Chapter 23 of [Pie02]

(a) Grammar for types, terms, and contexts

Types R, S, T, U, V	::=	X	Type variable
		$\top \mid \bot$	Maximal and minimal type
		$\forall \overline{X}.\, \overline{S} \to T$	Uncurried function type
Terms t, f	::=	x	Term variable
		$\texttt{fun}[\overline{X}](\overline{x}{:}\overline{S})t$	Abstraction
		$f[\overline{T}](\overline{t})$	Application
Contexts Γ	::=	$\bullet$	Empty context
		$\Gamma, x{:}T \mid \Gamma, X$	Term and type variable binding

(b) Subtyping rules: $\boxed{S \leq T}$

$$\frac{}{X \leq X}\ \texttt{S-Refl} \qquad \frac{}{T \leq \top}\ \texttt{S-Top} \qquad \frac{}{\bot \leq T}\ \texttt{S-Bot}$$

$$\frac{\overline{T} \leq \overline{R} \quad S \leq U}{\forall \overline{X}.\, \overline{R} \to S \leq \forall \overline{X}.\, \overline{T} \to U}\ \texttt{S-Fun}$$

(c) Type synthesis rules: $\boxed{\Gamma \vdash_{\Uparrow} t : T}$

$$\frac{}{\Gamma \vdash_{\Uparrow} x : \Gamma(x)}\ \texttt{S-Var} \qquad \frac{\Gamma, \overline{X}, \overline{x}{:}\overline{S} \vdash_{\Uparrow} t : T}{\Gamma \vdash_{\Uparrow} \texttt{fun}[\overline{X}](\overline{x}{:}\overline{S})t : \forall \overline{X}.\, \overline{S} \to T}\ \texttt{S-Abs}$$

$$\frac{\Gamma \vdash_{\Uparrow} f : \bot \quad \Gamma \vdash_{\Uparrow} \overline{t} : \overline{S}}{\Gamma \vdash_{\Uparrow} f[\overline{T}](\overline{t}) : \bot}\ \texttt{S-App-Bot}$$

$$\frac{\Gamma \vdash_{\Uparrow} f : \forall \overline{X}.\, \overline{S} \to R \quad \Gamma \vdash_{\Uparrow} \overline{t} : \overline{U} \quad \overline{U} \leq [\overline{T}/\overline{X}]\overline{S}}{\Gamma \vdash_{\Uparrow} f[\overline{T}](\overline{t}) : [\overline{T}/\overline{X}]\ R}\ \texttt{S-App}$$

Figure 1: System $\mathrm{F}_{\leq}$ (part 1)

term variables, and t the body of the function), and applications $f[\overline{T}](\overline{t})$ (with function f, type arguments $\overline{T}$, and term arguments $\overline{t}$), with these last two given in a fully-uncurried style – the over-bar notation indicates some sequence of the grammatical category underneath it.

Subtyping and Typing Figures 1b and 1c give the inference rules for forming resp. the subtyping and typing judgments of the system. We read such rules as follows: first we look at the judgment below the horizontal line for the *conclusion* of the rule, and then we look above the line for its *premises* (if there are no premises, the rule is an axiom). For example, rule `S-Fun` in Figure 1b can be read as "to conclude that $\forall \overline{X}.\, \overline{R} \to S$ is a subtype of $\forall \overline{X}.\, \overline{T} \to U$, we need to show that types $\overline{T}$ are subtypes of $\overline{R}$ and to show that S is a subtype of U". With this we can read all the other subtyping rules: rule `S-Refl` is reflexivity for type variables, and `S-Top` and `S-Bot` say resp. that every type is a subtype of $\top$ and supertype of $\bot$. Rule `S-Fun` allows for more involved subtyping to occur between polymorphic function types, so for example $\forall X.\, \bot \to X \leq \forall X.\, X \to \top$ is derivable. Note the contravariant twist for the domain type. Intuitively, the justification is that if $S \leq T$ and you need a function that consumes terms of type S, you can make do with a function that consumes the larger class of terms of type T.

As for the typing rules, rule `Var` is standard, saying that a term variable has whatever type associated to it in a typing context Γ (the notation $\Gamma(x)$ is meant to convey "looking up" the type for x in Γ). In `S-Abs` we can type a function $\texttt{fun}[\overline{X}](\overline{x} : \overline{S})t$ whose term parameters $\overline{x}$ have type annotations $\overline{S}$ (where type variables $\overline{X}$ may be free in $\overline{S}$) when, after adding the abstracted type and term variables to the typing context, we can type its body t. Rule `S-App` is more involved: to type an application, we first confirm that the function f really has an arrow type, then synthesize types $\overline{U}$ for term arguments $\overline{t}$, and lastly we check that terms $\overline{t}$ are legal arguments to function f by making sure $\overline{U}$ is a subtype of the expected type $[\overline{T}/\overline{X}]\overline{S}$

(representing the simultaneous and capture-avoiding substitution of types $\overline{T}$ for type variables $\overline{X}$ in types $\overline{S}$). Finally, a special exception must be carved out in rule `S-App-Bot` for using a term f of type $\bot$ as an applicand – after all, $\bot$ is a subtyping of all polymorphic function types, so it can be used as a function in an application.

Redundant Annotations As it stands now, programming in System $\mathrm{F}_{\leq}$ is tedious. Consider: assuming some base type `Int` and some function f of type[2] $(\texttt{Int} \to \texttt{Int}) \to \texttt{Int}$, in order to apply f to an anonymous function, we are *forced* to give type annotations to the argument, as in $f(\texttt{fun}(x\!:\!\texttt{Int})x)$. This is frustrating, because the type of f *tells us* that this annotation should be `Int` – the explicit annotation is redundant! To try to improve this situation, we would first add *bare-abstractions* to our term language, of the form $\texttt{fun}[\overline{X}](\overline{x})t$, (that is, sans annotations for variables $\overline{x}$). To add a suitable typing rule for bare abstractions, we might be tempted to propose the following:

$$\frac{\Gamma, \overline{X}, \overline{x}\!:\!\overline{S} \vdash_{\Uparrow} t : T}{\Gamma \vdash_{\Uparrow} \texttt{fun}[\overline{X}](\overline{x})t : \forall \overline{X}.\, \overline{S} \to T}\ \texttt{S-Abs-Bad}$$

but alas, if we wanted to implement a type-checking algorithm for this system, how are we supposed to generate types $\overline{S}$ from a bare-abstraction? How can we know that this is available from the surrounding context? The problem with rule `S-Abs-Bad` is that it is not *syntax-directed*.

Syntax-directedness The usual implementation of a type inference algorithm is through a function that uses case analysis on the subject of typing (and other inputs) to determine which rule applies, and uses recursion to generate or check the rule's premises [Chr13]. Only type systems that are syntax-directed can be easily translated to such an implementation, because it ensures that two conditions hold for the inference rules that very closely correspond the implementation approach: first, given any derivable judgment, *the form of the judgment unambiguously determines the single rule needed to introduce it*; second, in every inference rule, *the "inputs" to the conclusion uniquely determine the inputs to each premise*, and the "outputs" of the premises determine the outputs of the conclusion. The typing rules of Figure 1c *mostly*[3]satisfy this first condition, and adding rule `S-Abs-Bad` cannot cause overlap as it concerns a totally new grammatical form.

The second condition given is also called the *mode-correctness* condition. When viewed algorithmically, each judgment is implicitly moded to indicate the parts considered as input and output. For what I have presented of System $\mathrm{F}_{\leq}$ so far, our mode-annotated judgments are $\Gamma^{+} \vdash_{\Uparrow} t^{+} : T^{-}$ and $S^{+} \leq T^{+}$, where superscripts $+$ and $-$ indicate resp. inputs and outputs. To ensure a rule is mode-correct, it must pass the following reading [Pfe04]:

1. **Assume** the *inputs to the conclusion* are known

   In `S-Abs-Bad`, this is Γ and $\texttt{fun}[\overline{X}](\overline{x})t$

2. **Show** that the *inputs of the premise* are known.

   In `S-Abs-Bad`, we can show that Γ and t are known, but get stuck showing that the extended typing context $\Gamma, \overline{X}, \overline{x}\!:\!\overline{S}$ is known, as it is not clear how to provide types $\overline{S}$

3. **Assume** the *outputs of the premise* are known.

   In `S-Abs-Bad`, this it T

[2]When a function type does not quantify over any type variables, I omit the quantifier entirely, as well as [] from the application and abstraction.

[3] I say *mostly*, because technically rules `S-App` and `S-App-Bot` overlap; since the two rules are so similiar, they can easily be merged into one significantly more complex but unambiguous rule, but this does not seem particularly useful.

4. **Show** that the *outputs of the conclusion* is known.

   In `S-Abs-Bad`, this is $\forall \overline{X}.\, \overline{S} \to T$, where again its not clear how $\overline{S}$ is known.

Bidirectional Rules To remove unnecessary annotations on abstractions while preserving syntax-directedness, we need some way to "tweak" the mode of `S-Abs-Bad` so that the type is considered an *input* to the judgment. And lo! Bidirectionality give us just that by adding a new judgment $\Gamma^+ \vdash_{\Downarrow} t^+ : T^+$ indicating that the surrounding context of term t has provided an expected type T, and all we need to do is *check* that t can be ascribed type T. The additional typing rules induced by adding this new judgment are listed in Figure 2 – the previous rules remain a part of this new system (except for rule `S-App` – see below)

(a) Extended language of terms

Terms $f, t \quad ::= \quad ... \mid \texttt{fun}[\overline{X}](\overline{x})t$ Bare abstraction

(b) Type checking rules: $\boxed{\Gamma \vdash_{\Downarrow} t : T}$

$$\frac{\Gamma, \overline{X}, \overline{x}{:}\overline{S} \vdash_{\Downarrow} t : T}{\Gamma \vdash_{\Downarrow} \texttt{fun}[\overline{X}](\overline{x})t : \forall \overline{X}.\, \overline{S} \to T}\ \texttt{C-Abs-Inf} \qquad \frac{\Gamma(x) \leq T}{\Gamma \vdash_{\Downarrow} x : T}\ \texttt{C-Var} \qquad \frac{\overline{T} \leq \overline{S} \quad \Gamma, \overline{X}, \overline{x}{:}\overline{S} \vdash_{\Downarrow} t : R}{\Gamma \vdash_{\Downarrow} \texttt{fun}[\overline{X}](\overline{x}{:}\overline{S})t : \forall \overline{X}.\, \overline{T} \to R}\ \texttt{C-Abs}$$

$$\frac{\Gamma \vdash_{\Uparrow} f : \forall \overline{X}.\, \overline{S} \to R \quad [\overline{T}/\overline{X}]R \leq U \quad \Gamma \vdash_{\Downarrow} \overline{t} : [\overline{T}/\overline{X}]\overline{S}}{\Gamma \vdash_{\Downarrow} f[\overline{T}](\overline{t}) : U}\ \texttt{C-App} \qquad \frac{\Gamma \vdash_{\Uparrow} t : T}{\Gamma \vdash_{\Downarrow} t : \top}\ \texttt{C-Top} \qquad \frac{\Gamma \vdash_{\Uparrow} f : \bot \quad \Gamma \vdash_{\Uparrow} \overline{t} : \overline{S}}{\Gamma \vdash_{\Downarrow} f[\overline{T}](\overline{t}) : R}\ \texttt{C-App-Bot}$$

$$\frac{\Gamma \vdash_{\Uparrow} f : \forall \overline{X}.\, \overline{S} \to R \quad \Gamma \vdash_{\Downarrow} \overline{t} : [\overline{T}/\overline{X}]\overline{S}}{\Gamma \vdash_{\Uparrow} f[\overline{T}](\overline{t}) : [\overline{T}/\overline{X}]R}\ \texttt{S-App}$$

Figure 2: System $\mathrm{F}_{\leq}$ (part 2)

Adding the checking-mode judgment to our system requires we add several new inference rules covering the different term constructs of System $\mathrm{F}_{\leq}$ (whose definition now spans both Figures 1a and 2a). *The most important new addition is rule* `C-Abs-Inf`, our mode-correct version of `S-Abs-Bad`. In this rule, the types $\overline{S}$ for variables $\overline{x}$ are provided as input to the judgment by the surrounding context, which requires the whole expression have type $\forall \overline{X}.\, \overline{S} \to T$. The other significant development of adding bidirectionality is the change in rule `S-App`. If we see application $f[\overline{T}](\overline{t})$ then between the type of f and type arguments $\overline{T}$ we *fully know* the expected types for the arguments. So instead of requring that $\overline{t}$ *synthesize* their types independently, and *then* see if these agree (via subtyping) with the expected types, we *pass the expected types* $[\overline{T}/\overline{X}]\overline{S}$ *down*. Between just these two rules we can now type $f(\texttt{fun}(x)x)$, our original motivating example. (The other rules of the system are listed for the sake of completeness – some of them will be touched upon towards the end of the section).

2.2 Example

Figure 3: Example derivation in System $\mathrm{F}_{\leq}$

$$\dfrac{\dfrac{}{\Gamma \vdash f : \forall X.\, (X, X) \to \top}\ \texttt{S-Var} \qquad \dfrac{\dfrac{\dfrac{}{\bot \leq X}\ \texttt{S-Bot} \quad \dfrac{}{X \leq \top}\ \texttt{S-Top}}{\forall X.\, X \to X \leq \forall X.\, \bot \to \top}\ \texttt{S-Fun}}{\Gamma \vdash_{\Downarrow} y : \forall X.\, \bot \to \top}\ \texttt{C-Var} \qquad \dfrac{\dfrac{\dfrac{}{\bot \leq \top}\ \texttt{S-Bot}}{\Gamma, X, x{:}\bot \vdash_{\Downarrow} x : \top}\ \texttt{C-Var}}{\Gamma \vdash_{\Downarrow} \texttt{fun}[X](x)x : \forall X.\, \bot \to \top}\ \texttt{C-Abs-Inf}}{\Gamma \vdash_{\Uparrow} f[\forall X.\, \bot \to \top](y, \texttt{fun}[X](x)x) : \top}\ \texttt{S-App}$$

To see how these inference rules work in practice, it is useful to have an example of a full derivation for a concrete expression. This is given in Figure 3 which synthesizes a type for the expression $f[\forall X.\, \bot \to \top](y, \texttt{fun}[X](x)x)$, assuming typing context Γ binds f to $\forall X.\, (X, X) \to \top$ and y to $\forall X.\, X \to X$. To type this application, we first synthesize the type of function f by looking it up in the context Γ in `S-Var` (premise omitted for space), then *check* arguments $(y, \texttt{fun}[X](x)x)$ against their expected types $[\forall X.\, \bot \to \top / X]\ (X, X)$ (where the instantiation for X comes from the given type argument). The first argument y synthesizes type $\forall X.\, X \to X$, which is subsumed by the checked type; the second argument, $\texttt{fun}[X](x)x$ does *not* synthesize its type, but the checked type allows us to associate the local variable x with type $\bot$, allowing us to check that the body of the function (here just x itself) has type $\top$.

2.3 Subsumption

We conclude by remarking on simple but significant interaction between bidirectionality and subtyping, one which will be a re-occuring theme throughout this report: in System $\mathrm{F}_{\leq}$, and in bidirectional type systems in general, subsumption of the type of a term is governed entirely by the term's context. Referring back to Figure 2, this phenomenon is most obvious in rules `C-Var` and `C-Top`, where both can be read as saying the *synthesized* type of the subject is subsumed by the *checked* type of the context. More subtly, if you keep this in mind when comparing the two versions of `S-App`, you might wonder whether *checking* the arguments $\overline{t}$ against types $[\overline{T/X}]\ \overline{S}$ (second version) includes the cases where the arguments *synthesize* types *which are then subsumed by* the expected type (first version). The answer is *yes* – any term that synthesizes its type can be checked against a supertype[4]:

$$\text{If } \Gamma \vdash_{\Uparrow} t : T \text{ and } T \leq S \text{ then } \Gamma \vdash_{\Downarrow} t : S$$

Furthermore, it is important to limit type subsumption to switches from checking to synthesis mode to keep our rules syntax-directed. If we had a synthesizing rule like

$$\frac{\Gamma \vdash_{\Uparrow} t : T \quad T \leq S}{\Gamma \vdash_{\Uparrow} t : S}\ \texttt{S-Sub-Bad}$$

it would be mode-incorrect, as supertype S is not determined by the inputs to the rule. Adding bidirectionality to System $\mathrm{F}_{\leq}$ allows for more precise control of type subsumption and let us infer types for more terms, all without making translation to an implementation difficult!

3 Local Type Inference

3.1 System PT

Inferring Type Arguments In Section 2 we saw that adding bidirectionality to System $\mathrm{F}_{\leq}$ reduced the number of typing annotations required for some function abstractions. Unfortunately, there is another pain-point of using the system: inferring *type arguments* in function *applications*! Sometimes these are painfully obvious – for example, assuming some base type `Int`, a constant `0` of type `Int`, and id of type $\forall X.\, X \to X$, we would have to write $id[\texttt{Int}](\texttt{0})$ – but we would clearly rather write $id(0)$ and infer that X should be instantiated to `Int`!

Adding type-argument inference to our bidirectional System $\mathrm{F}_{\leq}$ requires some care. First, type-argument inference requires we synthesize types for term arguments *and then* find valid instantiations of the type arguments from this. This is in direct tension with our bidirectional approach of using the *known* types of the arguments given by the function to *check* term arguments.

[4] I proved this version of the theorem myself – the original (Theorem 4.4.1 of [PT00]) is a completeness theorem stated in terms of fully-annotated terms.

(a) Extended language of terms

$$\text{Terms } f,t \quad ::= \quad ... \mid f(\overline{e}) \quad \text{Bare application}$$

(b) Type-argument inference rules

$$\frac{\Gamma \vdash_{\Uparrow} f : \forall \overline{X}.\, \overline{T} \to R, |\overline{X}| > 0 \quad \Gamma \vdash_{\Uparrow} \overline{t} : \overline{S} \quad \overline{S} \leq [\overline{U}/\overline{X}]\overline{T} \quad \texttt{Minimizes}(\overline{X}, \overline{U}, \overline{T}, \overline{S})}{\Gamma \vdash_{\Uparrow} f(\overline{t}) : [\overline{U}/\overline{X}]R} \ \texttt{S-App-Inf}$$

$$\frac{\Gamma \vdash_{\Uparrow} f : \forall \overline{X}.\, \overline{T} \to R, |\overline{X}| > 0 \quad \Gamma \vdash_{\Uparrow} \overline{t} : \overline{S} \quad \overline{S} \leq [\overline{U}/\overline{X}]\overline{T} \quad [\overline{U}/\overline{X}]R \leq V}{\Gamma \vdash_{\Downarrow} f(\overline{t}) : V} \ \texttt{C-App-Inf}$$

where $\texttt{Minimizes}(\overline{X}, \overline{U}, \overline{T}, \overline{S})$ = for all $\overline{V}.(\overline{S} \leq [\overline{V}/\overline{X}]\overline{T}$ implies $[\overline{U}/\overline{X}]R \leq [\overline{V}/\overline{X}]R)$

Figure 4: System **PT**

Second, there may be multiple legal type argument instantiations, risking non-determinism: in the example above, what stops us from inferring that $id(\texttt{0})$ has type $\top$ by instantiating X to $\top$?

Pierce and Turner in "Local Type Inference" [PT00] show one approach to resolving the tension between bidirectionality and type-argument inference. The additional term construct $f(\overline{e})$ (bare applications) and its two new typing rules are given for it are listed in Figure 4, which should be seen as further extending the rules for System $\mathrm{F}_{\leq}$ given in Figures 1 and 2 to form System **PT**. To the first issue ("synthesize or check the arguments?"), we know which approach is called for when the function of a bare application expects a non-zero number of type arguments (written $|\overline{X}| > 0$) – we *must* infer $\overline{X}$ by synthesizing types from the term arguments. To the second issue, non-determinism in the result type is staved-off in rule `S-App-Inf` by requiring a *minimality* condition (fourth premise) for the instantiation, which will guarantee uniqueness.

Type-argument Inference Rules In both `S-App-Inf` and `C-App-Inf`, after synthesizing types $\forall \overline{X}.\, \overline{T} \to R$ for the function f and $\overline{S}$ for the arguments $\overline{t}$, for the application to be well-typed we must check that $\overline{S}$ is a subtype of *some instantiation* $[\overline{U}/\overline{X}]$ of the expected type $\overline{T}$. The tricky part of this is that multiple such instantiations might satisfy $\overline{S} \leq [\overline{U}/\overline{X}]\ \overline{T}$ and specifically in `S-App-Inf` the particular choice can affect the *result type* of the application $[\overline{U}/\overline{X}]R$. (`C-App-Inf`, meanwhile, has its result type imposed on it from the context, so the particular choice of instantiation matters less).

To clearly communicate to programmers what type is produced from using it, rule `S-App-Inf` requires a *minimality condition*, i.e. that the result type $[\overline{U}/\overline{X}]R$ is the "smallest" (wrt. the subtyping relation) possible. More precisely, any other instantiation $[\overline{V}/\overline{X}]$ that could type the application would produce a result type $[\overline{V}/\overline{X}]R$ more general than result type $[\overline{U}/\overline{X}]R$; in concrete terms, in the application $id(0)$ above we could not instantiate X to $\top$ because this would produce a non-minimal result type $\top$. Thanks to anti-symmetry of the subtyping relation, this guarantees uniqueness, as two such minimal types are subtypes of each other and thus identical.

A minimal instantiation for the return type may not always exist. For example, assuming h has type $\forall X.\, X \to (X \to X)$, bare application $h(\texttt{0})$ might be typed (prior to the minimality check) at $\texttt{Int} \to \texttt{Int}$ or $\top \to \top$, which are incomparable in the subtyping relation. Because type variable X occurs co- and contravariantly in the result type of h, no minimal instantiation is possible. In such cases, rule `C-App-Inf` performs significantly better at typing bare applications – if the expected type is $\top \to \top$, then X can only be instantiated to $\top$.

3.2 Discussion

Rules as specification The careful reader will have noticed that even though we have tamed non-determinism in the result types of rules `S-App-Inf` and `C-App-Inf`, these two rules are not

syntax-directed. In both, when we check that they are mode-correct we get stuck trying to show that types $\overline{U}$ (needed for our instantiation $[\overline{U}/\overline{X}]$) are known. Perhaps surprisingly, this is not a bug but a feature! Pierce and Turner provide alternative, syntax-directed versions of these rules along with a sophisticated constraint solving algorithm producing $\overline{U}$, and show that these rules are sound and complete with respect to the rules given in Figure 4b – that is, every type derivation in the algorithmic system has a corresponding derivation in the specificational system, and vice-versa. Therefore, `S-App-Inf` and `C-App-Inf` can be seen as giving a *declarative specification* for users of the language, providing a full account of type-argument inference without reference to implementation details. Furthermore, the different between the specification and implementation is a mere the swapping of just these two rules with their algorithmic versions.

Bidirectionality is not required for making a clean distinction between specification and implementation. Indeed, in [PT00] the authors begin with a uni-directional system featuring type-argument inference with a clear distinction between specification and implementation, and refine it to have bidirectionality afterwards. Instead, bidirectionality provided an easy way to increase the power of type inference, both by allowing bare function abstractions and by increasing the set of typeable applications (by relaxing the minimality requirement in checking mode), *without* causing any confusion about what result types are inferred for bare applications or overly complicating the presentation of the type system.

Design-choices of System PT We conclude this section with discussion of some design choices made for System **PT** (comprised of the rules in Figures 1 (sans `S-App`), 2, and 4). Pierce and Turner sought to develop a type inference system using the purely *local* techniques of bidirectional rules and local type-argument inference. For the latter of these techniques to be effective, a single AST node for a bare application must be given all arguments up front to maximize the typing information available for type-argument inference. This leads them to use a fully-uncurried style for functions. Uncurried applications are still supported, but type inference performs significantly worse for them – we can synthesize a type for $h'(0,0)$, where h' is an "uncurried" function h' of type $\forall X.\,(X,X) \to X$, where (as we saw earlier) $h(0)(0)$ can *not* with a curried h.

A type system based on purely local techniques has the advantage of requiring only local reasoning when dealing with type errors. In their discussion, Pierce and Turner noted that type errors tend to be reported "close to the point where more type annotations are required (or where an actual error is present in the program)". Furthermore, they established empirically that this does not have to come at the price of requiring too many "silly" annotations in programs by analyzing ˜160,000 lines of ML code, identifying the kinds of annotations different styles of functional programming would require in an explicitly typed language, and confirming that the majority of redundant type annotations this would introduce is handled well by their system.

In some settings, the user is not the only benefactor of purely-local reasoning for type inference. In notes sent to the `types` mailing list [Ode02], Odersky gives a simple example showing that in type systems with (rich) nominal subtyping and type-argument inference, "any constraint system which admits instantiatable type variables on both sides of a subtyping constraint does not have a single best solution", giving a strong incentive for such languages to use purely local methods (like bidirectionality) to keep type inference predictable and well-specified.

4 Colored Local Type Inference

4.1 System OZZ

Improving the Compromise In Section 2 I showed how adding bidirectionality to our type inference system allowed us to infer annotations on bare function abstractions, then showed in

Section 3 how to add type-argument inference for bare applications. Where these two features meet they are in apparent conflict – bidirectionality wants us to *check* the term arguments of applications using a fully-known type, whereas type-argument inference wants to *synthesize* types for term arguments to know how to fill in the missing *type* arguments of the function. The compromise struck by System **PT** is "all-or-nothing": all type arguments are provided and we perform type checking on the term arguments, or all type arguments are omitted and we proceed with type synthesis.

It is natural to ask whether we can improve upon this situation. While many silly type annotations can be removed when using System **PT**, it is easy give cases where some must still be provided that we expect to occur semi-frequently in functional programming. Consider: assuming g has type $\forall X.\,(\texttt{Int} \to X) \to X$, we could not type expression $g(\texttt{fun}(x)x)$, because 1) g is not given a type argument for X, meaning we must perform type-argument inference and synthesize a type for its term argument, but 2) argument $\texttt{fun}(x)x$ is missing a type annotation on x and so does not synthesize a type! This is irritating, because we *know* that the bound x must have type `Int` by looking at the type of g. Framed another way, the heart of this issue is that the expected type $\texttt{Int} \to X$ of the argument only has *partial* type information: we want to *check* the argument with known domain type `Int` but *synthesize* the codomain type to find the type instantiating X. The bidirectional rules of System **PT**, however, pass types down and up whole-cloth and do not offer a way to decompose them for different inference modes.

In "Colored Local Type Inference" [OZZ01][5], Odersky et al. refine System **PT** to allow for *partial* type information to be given by the context to an expression. The grammar, subtyping rules, and type inference rules for this System **OZZ** are given in Figure 5, with some auxiliary judgments given in Figure 6. The system is significantly more complex than System **PT** (and directly based on it), so due to space considerations I will not exhaustively cover each inference rule. Instead, in this section will carefully study only the significant differences of System **OZZ** from System **PT** – which means the grammar, subtyping rules, and a single typing rule – and how these address the identified short-comings of System **PT**.

Grammar Figure 5a lists the grammar for the new system. First, *type constructors* themselves, and not judgments, come with checking and synthesizing mode annotations. As detailed in the grammar, T^* indicates a type with an arbitrary mixture of parts in checking or synthesizing modes Each type constructor is decorated with either $\Uparrow$ or $\Downarrow$, which as before indicate resp. synthesizing or checking mode. Reading subtyping and typing rules that use "colored" (mode-annotated) types requires knowing a handful of notational conventions, which unlike the paper itself I will conveniently list in one place for your reference:

- Type meta-variable with fixed mode: $S^{\Uparrow}$, $S^{\Downarrow}$

  Every type constructor in S has the given mode

- Type meta-variable with no mode: S

  If it occurs within a moded type constructor, as in $(S_1 \overset{\overline{X}}{\to} S_2)^{\Downarrow}$ (i.e. checking mode for the arrow), then it inherits this mode, as in $(S_1^{\Downarrow} \overset{\overline{X}}{\to} S_2^{\Downarrow})^{\Downarrow}$; otherwise, it is the same as S^*.

- Multiple occurrences of "colored" type S^* in an inference rule

  Seen in for example rule `Sel` and `Abs` of Figure 5c. The interpretation is that each occurence of S^* has the same coloration or mode-mixture throughout the rule (though the moding of S^* is arbitrary, it is fixed for one reading of the rule).

[5] So called because the "official" version of the paper used font coloring instead of superscript annotations to indicate the different modes within types.

(a) Grammar for modes, types, and terms

$$
\begin{array}{lll}
\text{Modes } * & ::= & \Uparrow \mid \Downarrow \\
\text{Types } T^*, S^*, R^* & ::= & X^{\Uparrow} \mid X^{\Downarrow} \\
 & & \top^{\Uparrow} \mid \top^{\Downarrow} \mid \bot^{\Uparrow} \mid \bot^{\Downarrow} \\
 & & \{x_1 : T_1^*, ..., x_n : T_n^*\}^{\Uparrow} \mid \{x_1 : T_1^*, ..., x_n : T_n^*\}^{\Downarrow} \quad \text{Records} \\
 & & (T^* \xrightarrow{\overline{X}} S^*)^{\Uparrow} \mid (T^* \xrightarrow{\overline{X}} S^*)^{\Downarrow} \quad \text{Functions} \\
\text{Terms } E, F & ::= & x \\
 & & \texttt{fun}[\overline{X}](x{:}T)E \mid \texttt{fun}(x)E \\
 & & F[\overline{T}](E) \mid F(E) \\
 & & E.x \mid \{x_1 = E_1, ..., x_n = E_n\} \quad \text{Record selection and construction}
\end{array}
$$

(b) Subtyping: $\boxed{T \leq S,\ T \overline{<} S}$

$$
\frac{}{T \leq T} \quad \frac{}{X^{\Uparrow} \leq X^{\Downarrow}} \quad \frac{}{\bot^{\Uparrow} \leq \bot^{\Downarrow}} \quad \frac{}{\top^{\Uparrow} \leq \top^{\Downarrow}} \quad \frac{}{(T^* \xrightarrow{\overline{X}} S^*)^{\Uparrow} \leq (T^* \xrightarrow{\overline{X}} S^*)^{\Downarrow}} \quad \frac{T_1 \leq T_2 \quad T_2 \leq T_3}{T_1 \leq T_3}
$$

$$
\frac{}{\bot^{\Uparrow} \leq \{x_1 : \bot^{\Uparrow}, ..., x_n : \bot^{\Uparrow}\}^{\Downarrow}} \quad \frac{}{\bot^{\Uparrow} \leq X^{\Downarrow}} \quad \frac{}{\bot^{\Uparrow} \leq (\top^{\Uparrow} \xrightarrow{\overline{X}} \bot^{\Uparrow})^{\Downarrow}} \quad \frac{}{\bot^{\Uparrow} \leq \top^{\Downarrow}}
$$

$$
\frac{}{\{x_1 : \top^{\Downarrow}, ..., x_n : \top^{\Downarrow}\}^{\Uparrow} \leq \top^{\Downarrow}} \quad \frac{}{X^{\Uparrow} \leq \top^{\Downarrow}} \quad \frac{}{(\bot^{\Downarrow} \xrightarrow{\overline{X}} \top^{\Downarrow})^{\Uparrow} \leq \top^{\Downarrow}} \quad \frac{T_1 \overline{<} T_1' \quad T_2' \leq T_2}{(T_1' \xrightarrow{\overline{X}} T_2) \leq (T_1 \xrightarrow{\overline{X}} T_2')}
$$

$$
\frac{}{\{x_1 : T_1^*, ..., x_n : T_n^*, x_{n+1} : \top^{\Downarrow}, ..., x_m : \top^{\Downarrow}\}^{\Uparrow} \leq \{x_1 : T_1^*, ..., x_n : T_n^*\}^{\Downarrow}} \quad \frac{T_1 \leq T_1' \quad ... \quad T_n \leq T_n'}{\{x_1 : T_1, ..., x_n : T_n\} \leq \{x_1 : T_1', ..., x_n : T_n'\}}
$$

(c) Colored type inference rules: $\boxed{\Gamma \vdash_{\mathrm{C}} E : T}$

$$
\frac{}{\Gamma \vdash_{\mathrm{C}} x : \Gamma(x)^{\Uparrow}}\ \textsc{Var} \qquad \frac{\Gamma \vdash_{\mathrm{C}} E : T \quad T \leq T'}{\Gamma \vdash_{\mathrm{C}} E : T'}\ \textsc{Sub} \qquad \frac{\Gamma, \overline{X}, x{:}T^{\Uparrow} \vdash_{\mathrm{C}} E : S^*}{\Gamma \vdash_{\mathrm{C}} \texttt{fun}[\overline{X}](x{:}T)E : (T \xrightarrow{\overline{X}} S^*)^{\Uparrow}}\ \textsc{Abs}
$$

$$
\frac{\Gamma, \overline{X^{\Uparrow}}, x{:}T^{\Uparrow} \vdash_{\mathrm{C}} E : S^* \quad \overline{X} \notin tv(E)}{\Gamma \vdash_{\mathrm{C}} \texttt{fun}(x)E : (T^{\Downarrow} \xrightarrow{\overline{X}} S^*)^{\Downarrow}}\ \textsc{Abs-Inf} \qquad \frac{\Gamma \vdash_{\mathrm{C}} F : (S^{\Uparrow} \xrightarrow{\overline{X}} T^{\Uparrow})^{\Downarrow} \quad \Gamma \vdash_{\mathrm{C}} E : [\overline{R}/\overline{X}]S^{\Downarrow}}{\Gamma \vdash_{\mathrm{C}} F[\overline{R}](E) : [\overline{R}/\overline{X}]T^{\Uparrow}}\ \textsc{App}
$$

$$
\frac{\Gamma \vdash_{\mathrm{C}} F : (S^{\Uparrow} \xrightarrow{\overline{X}} T^{\Uparrow})^{\Downarrow} \quad \Gamma \vdash_{\mathrm{C}} E : S' \quad S' \lhd_{\overline{X}} S^{\Downarrow} \quad S' \leq [\overline{R}/\overline{X}]S^{\Downarrow} \quad [\overline{R}/\overline{X}]T^{\Uparrow} \leq T' \quad \texttt{Minimizes}(\overline{X}, \overline{R}, T, T', S, S')}{\Gamma \vdash_{\mathrm{C}} F(E) : T'}\ \textsc{App-Inf}
$$

$$
\frac{\Gamma \vdash_{\mathrm{C}} E : \{x{:}T^*\}^{\Downarrow}}{\Gamma \vdash_{\mathrm{C}} E.x : T^*}\ \textsc{Sel} \qquad \frac{\Gamma \vdash_{\mathrm{C}} E_1 : T_1^* \quad ... \quad \Gamma \vdash_{\mathrm{C}} E_n : T_n^*}{\Gamma \vdash_{\mathrm{C}} \{x_1 = E_1, ...x_n = E_n\} : \{x_1 : T_1^*, ..., x_n : T_n^*\}^{\Uparrow}}\ \textsc{Rec}
$$

Figure 5: System **OZZ**

- Unmoded (equivalently $*$-moded) type constructor: $(S \xrightarrow{\overline{X}} T)$

  Seen in the record and arrow subtyping rules (3rd and 4th row, last column). Used to abbreviate two rules, one each for when the constructor is annotated with $\Downarrow$ and $\Uparrow$. Type meta-variables occuring in the constructor (like S and T above) have mode $*$.

Aside from modes, new to the language of types are *record types*. Records both make this system more similar to practical OOP languages with structural subtyping, *and* they obviate the need for uncurried applications, as passing multiple arguments can be simulated by passing a single record argument with multiple fields. The form of polymorphic function types has also been changed slightly, with the abstracted variables $\overline{X}$ located above the arrow (as in $(S \xrightarrow{\overline{X}} T)$) in order to make reading its three modalities easier (one each for the domain, codomain, and

the combination of an arrow and type quantifications). Finally, for the language of terms the only new additions are projection and construction of record types, and abstractions $\texttt{fun}(x)t$ that permit even type arguments to be omitted.

Subtyping The subtyping rules for System **OZZ** in Figure 5b are *far* more involved than the subtyping for System **PT**, but the details of how each rule works are less important than the high-level intuition of what purpose they serve. Recall the concluding remark in Section 2 about bidirectionality and subsumption: "subsumption of the type of a term is governed entirely by the term's context." System **OZZ** takes this insight even further than System **PT** by *tracking in the type itself the parts given by the context (⇓) and by the term (⇑)*; the subtyping relation carefully mediates the interaction of these two sources of information during subsumption, ensuring that synthesized information is never guessed and checked information is not arbitrarily discarded.

To get more specific, the crucial invariant of the subtyping rules is that, "*structural changes in the type always imply that the different (type) constructors differ also in color.*" In a derivation of $S \leq T$, a type constructor in S that gets changed in supertype T is **always** accompanied by a change in the corresponding part from $\Uparrow$ to $\Downarrow$. Reading from S to T, this means that during subsumption we cannot "guess" synthesized typing information that was supposed to come from the term itself; reading the other way, this means that we cannot "discard" checked typing information coming from the context unless the synthesized type of the term itself directs us to do so. (The subtyping relation $\overline{\leq}$ works in a similiar way but with flipped modes – it is needed because subtyping is contravariant in the domain of function types but modes are covariant. Its rules are omitted as it is a mirror image of $\leq$)

Example To continue to build an intuition for these rules, assume that some expression F synthesizes type $(\texttt{Int}^{\Uparrow} \to \texttt{Int}^{\Uparrow})^{\Uparrow}$. This is *not* a subtype of $(\texttt{Int}^{\Uparrow} \to \top^{\Uparrow})^{\Uparrow}$, but *is* a subtype of $(\texttt{Int}^{\Uparrow} \to \top^{\Downarrow})^{\Uparrow}$ – note the change in the mode of the codomain. This matches our expectation that we should never guess when doing type subsumption – in this example, the promotion of the codomain of F, $\texttt{Int}$, to $\top$ is valid only when directed by the context ($\top^{\Downarrow}$). Similarly, if the context of a term provides type $(\texttt{Int}^{\Downarrow} \to \top^{\Downarrow})^{\Downarrow}$ then this type can subsume $(\texttt{Int}^{\Downarrow} \to \texttt{Int}^{\Uparrow})^{\Downarrow}$ (where through synthesis of the term we learn the codomain was really $\texttt{Int}$) but not $(\texttt{Int}^{\Downarrow} \to \texttt{Int}^{\Downarrow})^{\Downarrow}$, because the context cannot both provide $\top^{\Downarrow}$ and $\texttt{Int}^{\Downarrow}$ as the expected codomain type.

Reading the rules The first row of subtyping rules are colorful variations of reflexivity and transitivity – note that the reflexivity rules mean that the mode annotations can also switch when the type constructors *agree*, not just when they differ. The second row says that a synthesized $\bot$ is a subtype of every type formed by a checked constructor, and says what to fill in for the types occurring in that constructor. For example, the smallest step to subsume $\bot^{\Uparrow}$ into a checked arrow type (with no information given from the context about the domain and codomain) is $(\top^{\Uparrow} \overset{\overline{X}}{\to} \bot^{\Uparrow})^{\Downarrow}$, the smallest arrow type possible. The rules in which $\top^{\Downarrow}$ occurs as a supertype are a mirror image of this. Last in the third row is the rule for functions with the usual contravariant twist in the domain type, complicated by the fact that modes are covariant and requiring the additional relation $\overline{\leq}$. Finally, the last row concerns records: a synthesized record type can be subsumed by a checked record type with fewer fields, and a record type is a subtype of another of the same mode if the corresponding field types are in the subtype relation.

Aside The reader may at this point be concerned about whether the subtyping relation is indeed syntax-directed. Clearly, it is crucial that it should be so – why do all this bidirectional book-keeping if the subtyping relation *itself* is non-deterministic? Still, it is not obvious that, for example, the transitivity rule is deterministic! Odersky et al. do not give give an explicit argument saying that this is so, however. To argue that they are syntax-directed, we would have to reason inductively (on the structure of types) about all possible subtyping derivations using the transitivity rule, showing in each case the rule does not overlap with another and that it is mode-correct[6]. As a brief example giving some intuition for why this should be provable,

(a) Matching rules $\boxed{S' \lhd_{\overline{X}} S}$

$$\frac{}{S^{\Downarrow} \lhd_{\overline{X}} S^{\Downarrow}} \quad \frac{X \in \overline{X}}{T^{\Uparrow} \lhd_{\overline{X}} X^{\Downarrow}} \quad \frac{T_1^* \lhd_{\overline{X}} T_2^* \quad S_1^* \lhd_{\overline{X}} S_2^*}{(T_1^* \xrightarrow{\overline{Y}} S_1^*)^{\Downarrow} \lhd_{\overline{X}} (T_2^* \xrightarrow{\overline{Y}} S_2^*)^{\Downarrow}} \quad \frac{T_1^* \lhd_{\overline{X}} S_1^* \quad \ldots \quad T_n^* \lhd_{\overline{X}} S_n^*}{\{x_1 : T_1^*, ..., x_n : T_n^*\}^{\Downarrow} \lhd_{\overline{X}} \{x_1 : S_1^*, ..., x_n : S_n^*\}^{\Downarrow}}$$

(b) $\boxed{\texttt{Minimizes}(\overline{X}, \overline{R}, T, T', S, S')}$

for all $\overline{R'}$,T''.$(S' \leq [\overline{R'}/\overline{X}]S^{\Downarrow} \wedge [\overline{R'}/\overline{X}]T^{\Uparrow} \leq T'' \sim T'$ implies $[\overline{R}/\overline{X}]T^{\Uparrow} \leq [\overline{R'}/\overline{X}]T^{\Downarrow})$

(c) Similarity of checked parts $\boxed{S \sim T}$

$$\frac{}{T^{\Uparrow} \sim S^{\Uparrow}} \quad \frac{}{T^{\Downarrow} \sim T^{\Downarrow}} \quad \frac{T_1^* \sim S_1^* \quad \ldots \quad T_n^* \sim S_n^*}{\{x_1 : T_1^*, ..., x_n : T_n^*\}^{\Downarrow} \sim \{x_1 : S_1^*, ..., x_n : S_n^*\}^{\Downarrow}} \quad \frac{S_1^* \sim S_2^* \quad T_1^* \sim T_2^*}{(S_1^* \xrightarrow{\overline{X}} T_1^*)^{\Downarrow} \sim (S_2^* \xrightarrow{\overline{X}} T_2^*)^{\Downarrow}}$$

Figure 6: System **OZZ** (auxiliary definitions)

consider the following derivation showing that $\bot^{\Uparrow}$ is a subtype of $(\texttt{Int} \to \{x : \bot^{\Uparrow}\})^{\Downarrow}$

$$\dfrac{\dfrac{}{\bot^{\Uparrow} \leq (\top^{\Uparrow} \to \bot^{\Uparrow})^{\Downarrow}}\ 1 \quad \dfrac{\texttt{Int}^{\Downarrow} \geqslant \top^{\Uparrow} \quad \bot^{\Uparrow} \leq \{x : \bot^{\Uparrow}\}^{\Downarrow}}{(\top^{\Uparrow} \to \bot^{\Uparrow})^{\Downarrow} \leq (\texttt{Int} \to \{x : \bot^{\Uparrow}\})^{\Downarrow}}\ 2}{\bot^{\Uparrow} \leq (\texttt{Int} \to \{x : \bot^{\Uparrow}\})^{\Downarrow}}\ 3$$

Only transitivity could conclude with this judgment (3); only one rule will bring $\bot^{\Uparrow}$ closer a super-type formed by a (checked) arrow (1); only one rule applies for two arrow types with differing components (2). No guessing is performed at any step.

4.2 Rule `App-Inf`

We shall only cover the most complex rule of the system, `App-Inf`, which is used for typing bare applications. This is the very rule that will allow us to type the motivating example from the beginning of the section, $g(\texttt{fun}(x)x)$ where g has type $((\texttt{Int} \to X) \xrightarrow{X} X)$, and a careful study of the rule yields a good understanding of the power of the system as a whole.

Typing F First, note that the mode of result type T' is arbitrary, meaning that as a special case the rule applies when T' is a purely synthesizing or checking result type. So `App-Inf` of System **OZZ** *combines* rules `S-App-Inf` and `C-App-Inf` of System **PT** (cf. Figure 4), as well as supporting a mixed-mode result type. To proceed, we check the function F against a checked arrow-type with synthesized domain and codomain types, $(S^{\Uparrow} \xrightarrow{\overline{X}} T^{\Uparrow})^{\Downarrow}$, indicating that through its *context* we only know that F should type as some function, but S and T we learn from inspecting F itself.

Typing E, matching, and result type T' After typing applicand F we next turn to argument E. Here it becomes apparent that, like `S-App-Inf` and `C-App-Inf` of System **PT**, `App-Inf` is also a specificational rule. Deriving $\Gamma \vdash_{\mathrm{C}} E : S'$ requires making an arbitrary choice for S' (both in its shape and mixture of modes). This choice is constrained by the third premise $S' \lhd_{\overline{X}} S^{\Downarrow}$, whose formation rules are given in Figure 6a. At a high level, judgment $S' \lhd_{\overline{X}} S$ is the mechanism by which the partial type information given by function F is passed down to argument E – it says that S' *must match the structure and color of* $S^{\Downarrow}$ *modulo the type variables* $\overline{X}$. For technical reasons[7] we need the fourth premise $S' \leq [\overline{R}/\overline{X}]S^{\Downarrow}$ to ensure E *really is*

[6]Subtley, mode-correctness for the subtyping relation is unrelated to the mode annotations of its subjects: the former is a meta-language convention, the latter is an object-language feature.

[7]Specifically, the matching judgment does not account for *solution sharing* for type variables $\overline{X}$ – the same X can match with two unrelatable types

a type-correct argument to F. The fifth premise says that the purely synthesized result type $[\overline{R}/\overline{X}]T^{\Uparrow}$ is subsumed by any contextually-provided parts of T'.

Minimization of $\overline{R}$ The fourth premise introduces another place where rule `App-Inf` is specificational, rather than syntax-directed: the choice of types $\overline{R}$ instantiating type variables $\overline{X}$. As with rule `S-App-Inf` in System **PT** (cf. Figure 4), multiple such $\overline{R}$ could make the application well-typed, so this non-determinism must be constrained via the `Minimizes` relation (given in Figure 6) to ensure that only one result type is possible when using this rule. The reading for $\texttt{Minimizes}(\overline{X}, \overline{R}, T, T', S, S')$ in plain language is: select an instantiation $\overline{R}$ for type variables $\overline{X}$ such that, for any *other* instantiation $\overline{R'}$ and result type T'' that carries exactly the same contextual information as T' (enforced by $T'' \sim T'$, discussed below), if $[\overline{R'}/\overline{X}]$ makes the application well-typed (by *subsuming* the argument type: $S' \leq [\overline{R'}/\overline{X}]S^{\Downarrow}$; and by *being subsumed by* the result type: $[\overline{R'}/\overline{X}]T^{\Uparrow} \leq T''$), then this result type would be a super-type of the (purely-synthesized) result type given by choosing $\overline{R}$.

The need for judgment $S \sim T$, indicating that types S and T *coincide precisely in their checked parts*, is a little subtle. To give an intuition for this, first note that instantiating types $\overline{R'}$ could produce different types in the synthesized parts of result $[\overline{R'}/\overline{X}]T$, so the relation $\sim$ must allow synthesizing parts to differ between its supertypes T' and T''. Second, recall the key difference between rules `C-App-Inf` and `S-App-Inf` in System **PT** was that the *former did not need a minimality statement*. The same is true of the checked parts of T', so $\sim$ enforces that our choice of $\overline{R}$ minimizes only the synthesized parts of T'.

4.3 Examples

We now illuminate the preceding discussion by giving two example derivations. The first is of the motivating example from the beginning of the section and shows how partial type propagation is carried out in System **OZZ**, and the second is an inference failure showing how the different sources of typing information are carefully managed.

$$\dfrac{\dfrac{\dfrac{\Gamma(g) = ((\texttt{Int} \to X) \xrightarrow{X} X)^{\Uparrow}}{\Gamma \vdash_{\mathrm{C}} g : ((\texttt{Int} \to X) \xrightarrow{X} X)^{\Uparrow}}\,\textsc{Var} \quad \mathcal{D}_1}{\Gamma \vdash_{\mathrm{C}} g : ((\texttt{Int} \to X)^{\Uparrow} \xrightarrow{X} X^{\Uparrow})^{\Downarrow}}\,\textsc{Sub} \quad \dfrac{\overline{\Gamma, x{:}\,\texttt{Int}^{\Uparrow} \vdash_{\mathrm{C}} x : \texttt{Int}^{\Uparrow}}}{\Gamma \vdash_{\mathrm{C}} \texttt{fun}(x)x : (\texttt{Int}^{\Downarrow} \to \texttt{Int}^{\Uparrow})^{\Downarrow}}\,\textsc{Abs-Inf} \quad \mathcal{D}_2 \quad \mathcal{D}_3 \quad \mathcal{D}_4 \quad \texttt{Minimizes}(...)}{\Gamma \vdash_{\mathrm{C}} g(\texttt{fun}(x)x) : \texttt{Int}^{\Uparrow}}\,\textsc{App-Inf}$$

$$\mathcal{D}_1 = \dfrac{}{((\texttt{Int} \to X)^{\Uparrow} \xrightarrow{X} X^{\Uparrow})^{\Uparrow} \leq ((\texttt{Int} \to X)^{\Uparrow} \xrightarrow{X} X^{\Uparrow})^{\Downarrow}} \qquad \mathcal{D}_2 = \dfrac{\texttt{Int}^{\Downarrow} \lhd_X \texttt{Int}^{\Downarrow} \quad \texttt{Int}^{\Uparrow} \lhd_X X^{\Downarrow}}{(\texttt{Int}^{\Downarrow} \to \texttt{Int}^{\Uparrow})^{\Downarrow} \lhd_X (\texttt{Int}^{\Downarrow} \to X^{\Downarrow})^{\Downarrow}}$$

$$\mathcal{D}_3 = \dfrac{\texttt{Int}^{\Downarrow} \geqslant \texttt{Int}^{\Downarrow} \quad \texttt{Int}^{\Uparrow} \leq \texttt{Int}^{\Downarrow}}{(\texttt{Int}^{\Downarrow} \to \texttt{Int}^{\Uparrow})^{\Downarrow} \leq (\texttt{Int}^{\Downarrow} \to \texttt{Int}^{\Downarrow})^{\Downarrow}} \qquad \mathcal{D}_4 = \overline{\texttt{Int}^{\Uparrow} \leq \texttt{Int}^{\Uparrow}}$$

Figure 7: System **OZZ** Ex. 1

Consider again the expression $g(\texttt{fun}(x)x + 1)$ where $g : ((\texttt{Int} \to X) \xrightarrow{X} X)^{\Uparrow}$. The full typing derivation for this expression is listed in Figure 7. Typing begins with the application of g to its argument, so to type g we pass down a *checked* arrow type with domain and codomain to-be-synthesized. Seeing that g is a variable, we look up the purely-synthesized type for it from the context and then promote this (via `Sub`) into the checked arrow type, resulting in type $((\texttt{Int} \to X)^{\Uparrow} \xrightarrow{X} X^{\Uparrow})^{\Downarrow}$.

Next, we need to find *some* type for the argument $\texttt{fun}(x)x$ that matches (via judgment $\lhd_X$) its partial expected type $(\texttt{Int} \to X)^{\Downarrow}$. Using rule `Abs-Inf` we can type argument $\texttt{fun}(x)x$

as $(\texttt{Int}^{\Downarrow} \to \texttt{Int}^{\Uparrow})^{\Downarrow}$, indicating we know the domain type from the outside, and synthesize the codomain from the inside of the term. Derivation $\mathcal{D}_2$ shows that this type does indeed match the expected type given from g. Derivations $\mathcal{D}_3$ and $\mathcal{D}_4$ show resp. that with instantiation $[\texttt{Int}/X]$ the application is well-typed in the argument and result type. Finally, it is easy to see that the only other possible instantiation for X is $\top$, which does not produce a smaller result type – so $[\texttt{Int}/X]$ is the minimal instantiation for this application.

For the second example, we need make only one small change to the expression to cause type inference to fail. Replace g with h of type $((X \to X) \xrightarrow{X} X)$, and consider typing $h(\texttt{fun}(x)x)$. Intuitively, the reason this fails to type is that there is not enough information provided by h to know what type x should be given. So, *how does this system enforce this?* We can still type the argument of h with $(\texttt{Int}^{\Downarrow} \to \texttt{Int}^{\Uparrow})^{\Downarrow}$, or any super-type of this of the form $(\texttt{Int} \to S)^{\Downarrow}$, but none of these can be matched (using $\lhd_X$) against the partial expected type $(X \to X)^{\Downarrow}$, because when we reach the domain types we find $\texttt{Int}^{\Downarrow} \lhd_X X^{\Downarrow}$ is not derivable, indicating that $\texttt{Int}^{\Downarrow}$ was an arbitrary choice. So our matching judgment, just like the other rules *really does* enforce that the unknown parts of the type are never guessed, but synthesized from the term.

4.4 Discussion

Implementation As was the case with System **PT**, System **OZZ** is a specificational type system that is *sound* and *complete* with respect to a type inference algorithm also presented in [OZZ01]. Mode annotations do not appear on types in the algorithmic system – instead, a new grammatical category of *prototypes* is introduced to explicitly represent partial type information. Prototypes introduce only one additional type constant "?", marking an unknown part of the type. For example, the "colored" judgment $\bullet \vdash_{\mathrm{C}} \texttt{fun}(x)x : (\texttt{Int}^{\Downarrow} \to \texttt{Int}^{\Uparrow})^{\Downarrow}$ translates to the algorithmic system as $(\texttt{Int} \to ?), \bullet \vdash_{\mathrm{W}} \texttt{fun}(x)x : (\texttt{Int} \to \texttt{Int})$, meaning that we start typing this expression knowing it should be a function and that the domain type should be `Int`, but without knowing what the codomain should be. When we finish typing it, this partial knowledge has been completed – incomplete information is never produced as output by the typing judgment.

Perhaps not surprisingly, the constraint generation and solution algorithm used by the implementation for System **OZZ** is almost exactly the same as the one used for the type inference algorithm for System **PT**, with the straightforward extension of decomposing constraints on record types to constraints on each record field. This is significant, as it means that the increased power of this system comes *solely from the more granular local propagation of type information.*

What's the point? In the next section, I will discuss a type system that combines the local propagation of type information with bidirectional rules *and* uses *global* methods of constraint generation and solution. The specificational rules are elegant and far simpler than the rules of System **OZZ**. Given this, the reader may well ask "Why go through all this trouble for a *local* type inference system?" Aside from the advantages of local inference methods mentioned in Section 1, for Odersky et al. [OZZ01] the answer was *robustness to language extensions.* The authors report that part of the motivation for this work came from designing a functional net language called Funnel [Ode00] based on System $F_{\leq}$. As remarked earlier, full type inference for (impredicative) System F is undecidable, and this is also true for its extension System $\mathrm{F}_{\leq}$ with its deep subtyping [TU96]. The authors found that placing careful restrictions on polymorphism and subtyping to maintain decidability produced a rather brittle system – "*every new language construct, or even just a slight change of an existing language construct is a possible threat to the decidability or tractability of the [system]*". Local inference methods allow for a decidable system in which typing each language construct can be considered separately, safely, because the *only* thing that matters for typing an expression is the typing information locally available.

5 Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism

5.1 System DK

The last type system we will look, System **DK** by Dunfield and Krishnaswami [DK13], is a departure from the preceding systems. First, it supports fewer type language features: it is *predicative*, meaning that type variables cannot be instantiated with polymorphic types. Second, it is based on System F (rather than $F_{\leq}$) and so lacks the type constants $\top$ and $\bot$. In particular, together these changes to the type system mean that the type-argument inference will no longer concern itself with finding some "minimizing instantiation", as usually at most one instantiation is guaranteed to be type-correct. Finally, it performs *non-local* type-argument inference: in the applications of polymorphic functions, the specification will not only abstract away *how* instantiations are selected, but even *which* instantiations will not be (locally) obvious. The trade-off for these changes is a type inference system with a refreshingly simple specification that gracefully handles *higher-rank polymorphism*.

"More polymorphic than" [8]In the type systems of the previous settings, polymorphic functions could be related via subtyping only if they had the same "polymorphic shape" – that is, both took the *same number of type arguments*, for example $\forall\overline{X}.\, S_1 \to T_1 \leq \forall\overline{X}.\, S_2 \to T_2$. But, consider: what if we wanted to type the expression $f\ id$, where f has type $(\texttt{Int} \to \texttt{Int}) \to \texttt{Int}$ and id has type $\forall X.\, X \to X$? It is reasonable to expect that id should work wherever a function of type $\texttt{Int} \to \texttt{Int}$ is expected, as we can infer the programmer really meant to give argument $id[\texttt{Int}]$. Unfortunately, the subtyping rules of Systems **PT** and **OZZ** do not allow us to derive $\forall X.\, X \to X \leq \texttt{Int} \to \texttt{Int}$, which we would need to type the application. We need a rule that allows us to say that if *some instantiation* of X (e.g. $[\texttt{Int}/X]$) allows us to derive $X \to X \leq \texttt{Int} \to \texttt{Int}$, then we have $\forall X.\, X \to X \leq \texttt{Int} \to \texttt{Int}$.

What other subtypes should a polymorphic *supertype* have? Even if two polymorphic types do not have the same shape, we can make the case that some types are "more polymorphic than" (i.e. subtypes of) others. For example, let f have type $(\forall Z.\, Z \to Z \to Z) \to \texttt{Int}$ and let *const* have type $\forall X.\, \forall Y.\, X \to Y \to X$. Again, intuitively $f\ const$ should be typeable – for any type argument T instantiating Z, *const* can be provided T for *both* its type arguments, as in $const[T][T]$. *const* takes two type arguments, but it can always be used as if it took only one. For this case we need a rule saying that to derive $\forall X.\, \forall Y.\, X \to Y \to X \leq \forall Z.\, Z \to Z \to Z$ it suffices to derive $\forall X.\, \forall Y.\, X \to Y \to X \leq Z \to Z \to Z$, where Z is now a free type variable. Then, using the reasoning in preceding paragraph, we can instantiate X and Y with Z.

Performing the two kinds of reasoning given above to an arbitrary contravariant[9]depth in a type enables inference for *higher-rank* types. In the immediately preceding example, the function f has a rank 2 type because the quantification of Z is twice nested on the left of an arrow. Higher-rank types are not needed all too commonly, but "there are usually no workarounds; if you need [them], you *really* need them" ([PJVWS07], Section 2 – see examples). Unfortunately, combining these two rules and the usual subtyping rule for arrows with impredicativity results in an *undecidable* subtyping relation [TU96]. Because one goal of System **DK** is to support inference higher-rank types, it must compromise and use *predicative* polymorphism only.

Grammar The grammar of System **DK** is given in Figure 8a, and takes several departures from the grammars of the previous two systems we have seen. In the language of types, functions

[8]The examples given come from Petyton-Jones et al. [PJVWS07], an earlier work that influenced System **DK**.

[9]Why not covariantly? A type with aquantifier that occurs within the codomain of an arrow type, such as $\forall X.\, X \to (\forall Y.\, Y \to X)$, is equivalent to a type in *prenex* form (e.g. $\forall X, Y.\, X \to Y \to X$). The same cannot be said when the quantifier occurs contravariantly

(a) Grammar

$$
\begin{array}{lll}
\text{Types } S,T,U,V & ::= & \mathbb{1} \mid X \mid \forall X.\, T \mid S \to T \\
\text{Monotypes } \tau,\sigma & ::= & \mathbb{1} \mid X \mid \tau \to \sigma \\
\text{Terms } e & ::= & x \mid () \mid \lambda x.\, e \mid e\ e \mid (e{:}S) \\
\text{Contexts } \Psi & ::= & \bullet \mid \Psi, X \mid \Psi, x{:}S
\end{array}
$$

(b) Subtyping $\boxed{\Psi \vdash S \leq T}$

$$
\frac{X \in \Psi}{\Psi \vdash X \leq X}\ \leq\texttt{Var} \qquad \frac{}{\Psi \vdash \mathbb{1} \leq \mathbb{1}}\ \leq\texttt{Unit} \qquad \frac{\Psi \vdash T_1 \leq S_1 \quad \Psi \vdash S_2 \leq T_2}{\Psi \vdash S_1 \to S_2 \leq T_1 \to T_2}\ \leq\texttt{Arr}
$$

$$
\frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/X]S \leq T}{\Psi \vdash \forall X.\, S \leq T}\ \leq\forall\texttt{L} \qquad \frac{\Psi, X \vdash S \leq T}{\Psi \vdash S \leq \forall X.\, T}\ \leq\forall\texttt{R}
$$

(c) Type Inference Rules: Synthesis, Checking, and Application

$\boxed{\Psi \vdash_{\Uparrow} e : T}$ $\boxed{\Psi \vdash_{\Downarrow} e : T}$ $\boxed{\Psi \vdash_{\Rightarrow} S \cdot e : T}$

$$
\frac{}{\Psi \vdash_{\Uparrow} x : \Psi(x)}\ \texttt{Var} \qquad \frac{\Psi \vdash_{\Uparrow} e : T' \quad \Psi \vdash T' \leq T}{\Psi \vdash_{\Downarrow} e : T}\ \texttt{Sub} \qquad \frac{\Psi \vdash_{\Downarrow} e : T}{\Psi \vdash_{\Uparrow} (e{:}T) : T}\ \texttt{Ann} \qquad \frac{\Psi, X \vdash_{\Downarrow} e : T}{\Psi \vdash_{\Downarrow} e : \forall X.\, T}\ \forall\texttt{Intro}
$$

$$
\frac{\Psi, x{:}S \vdash_{\Downarrow} e : T}{\Psi \vdash_{\Downarrow} \lambda x.\, e : S \to T}\ \to\texttt{Chk} \qquad \frac{}{\Psi \vdash_{\Downarrow} () : \mathbb{1}}\ \mathbb{1}\texttt{Chk} \qquad \frac{\Psi \vdash_{\Uparrow} e_1 : S \quad \Psi \vdash_{\Rightarrow} S \cdot e_2 : T}{\Psi \vdash_{\Uparrow} e_1\ e_2 : T}\ \texttt{App} \qquad \frac{\Psi \vdash_{\Rightarrow} [\tau/X]S \cdot e : T}{\Psi \vdash_{\Rightarrow} \forall X.\, S \cdot e : T}\ \forall\texttt{App}
$$

$$
\frac{\Psi, x{:}\sigma \vdash_{\Downarrow} e : \tau}{\Psi \vdash_{\Uparrow} \lambda x.\, e : \sigma \to \tau}\ \to\texttt{Syn} \qquad \frac{}{\Psi \vdash_{\Uparrow} () : \mathbb{1}}\ \mathbb{1}\texttt{Syn} \qquad \frac{\Psi \vdash_{\Downarrow} e : S}{\Psi \vdash_{\Rightarrow} S \to T \cdot e : T}\ \to\texttt{App}
$$

Figure 8: System **DK**

and type quantifications are uncurried and de-coupled from each other – these are the traditional System F types. Second, an additional category has been carved out for types that are not polymorphic. Distinguishing *monotypes* from *polytypes* is important for a predicative system, as only the former can legally instantiate type variables in the latter. The last change in the language of types is the inclusion of the "unit type" $\mathbb{1}$, which classifies only the term $()$. For terms, we now no longer can directly annotate arguments to functions (here written $\lambda x.\, e$), nor indeed give explicit type abstractions! Instead, the roles of both of these are taken by the explicit type annotation $(e{:}S)$, which is discussed further below.

Subtyping Figure 8b lists the subtyping rules. Rule $\leq$`Var` and $\leq$`Unit` are reflexively rules and $\leq$`Arr` is the subtyping rule for arrows we have already seen, though now without mention of any quantified type variables. Instead, type variables are handled by the last two rules. Rule $\leq\forall$`L` says that, to determine that $\forall X.\, S$ is a subtype of some T, we only need to find *some* (monotype) instantiation τ of X that makes $[\tau/X]S$ a subtype of T. On the other hand, rule $\leq\forall$`R` says that trying to determine that S is a subtype of some polymorphic type $\forall X.\, T$, we extend the context of types with variable X and try to derive $S \leq T$.

The intuition for why $\leq\forall$`L` is sensible comes if we interpret types as sets of terms, and the subtyping relation as a *subset* relation. The set of terms classified by the type $\forall X.\, X \to X$ is, in a sense, the *intersection* of the set of terms classified by all types such as `Int` $\to$ `Int`, $\mathbb{1} \to \mathbb{1}$, etc. To show that some set $\mathcal{T}$ is a superset of an intersection of sets, it suffices to show that $\mathcal{T}$ is a superset of just *one* of the intersected sets – here, the ones given resp. by the instantiations $[\texttt{Int}/X]$ and $[\mathbb{1}/X]$. Giving an intuition for rule $\leq\forall$`R` requires using the same perspective – interpreting types as sets and $\forall$ as set intersection, to show that $\mathcal{S}$ is a subset of some intersection we must show that $\mathcal{S}$ is a subset of *every* set in the intersection, which in turn

means showing that $\mathcal{S}$ is a subset of some *arbitrary* set in the intersection. In the subtyping rule, this is done by adding X to the typing context, representing an arbitrary monotype.

Remark. Unlike the previous systems, the subtyping judgment for System **DK** is itself specificational – in rule $\leq\forall\mathtt{L}$ the type τ is guessed "out of thin air". This translates to the promise that if such an instantiation exists, *the implementation will find it.*

Typing rules The typing rules for System **DK** include the usual checking ($\vdash_{\Downarrow}$) and synthesizing ($\vdash_{\Uparrow}$) modes, and additionally have a *third* form ($\vdash_{\Rightarrow}$) for an *application* mode. The judgment $\Psi \vdash_{\Rightarrow} S \cdot e : T$ can be read as "a function of type S can be applied to term e, producing type T," and the rules forming this judgment will instantiate the leading quantified type variables of S as they check the application is well-typed. Recalling Section 2, the mode-annotation of this judgment is $\Gamma^+ \vdash_{\Rightarrow} S^+ \cdot t^+ : T^-$. Before discussing this judgment in more detail, let us consider some of the interesting rules of the more traditional bidirectional judgments.

Rule `Sub` is what we have come to expect in a subsumption rule in a bidirectional setting, with type subsumption mediating an expression's checked and synthesized type. Annotations, meanwhile, cause a switch in mode going the *other* direction: an annotated expression $(e : S)$ synthesizes type S if e *checks* against S. This rule is vitally important as System **DK** lacks explicit type abstractions or annotations on function arguments and so needs a reliable way to *contextually* provide this typing information. We see this directly in rule $\forall\mathtt{Intro}$, as nothing about the form of the subject of typing e tells us that it ought to have a polymorphic type, only its expected type $\forall X.\, T$, as well as more standard rule $\to\mathtt{Chk}$ for bare abstractions.

Now we consider the application judgment $\vdash_{\Rightarrow}$, begun by rule `App`. After synthesizing type S for the function e_1 we must have that *applying a function of this type to e_2 produces result type T*. Rule $\forall\mathtt{App}$ is the critical specificational typing rule, saying that if our function type is $\forall X.\, S$, and some instantiation τ for X will let us type the application, then the system will somehow find τ. Finally, rule $\to$ `App` says that once our application judgment reveals an arrow type $S \to T$, we check our argument e against the domain type S – even though S may contain "guessed" type instantiations that (algorithmically) could only have been learned from inspecting e itself!

A notable exception There is one more rule to discuss that is a bit of an outlier for the system: rule $\to\mathtt{Syn}$. This says that bare function abstractions *can* synthesize a type, but only a monotype. Furthermore, somehow the body of the function e is *checked* with type τ, instead of synthesizing it. If you read this rule algorithmically, it hardly makes any sense at all! Instead, it should be read as saying that if $\lambda x.\, e$ can be typed at some monotype, then the algorithm *will find* that monotype. Rather than being passed down to type the body of the function as part of some contextual (local) type propagation, checking e against type τ is a way to guarantee the implementation *need not worry about synthesizing a polymorphic type* for the expression – if it happened to, rule `Sub` would subsume it into the monotype τ. Rule $\to$ `Syn` was added by the authors to show that System **DK** can accommodate extensions that are not-so-strictly bidirectional, making the system closer to the Damas-Milner type inference style.

5.2 Example

Figure 9 gives an example typing derivation in System **DK** that helps give an intuition for some of its novel features. We start by trying to type $f\ id\ suc$, where f has type $\forall X.\, X \to X \to X$, id has type $\forall Y.\, Y \to Y$, and suc has type $\mathtt{Int} \to \mathtt{Int}$. Because id and suc have different types, and because f expects its two arguments to be the *same* type, it is up to the application judgment $\vdash_{\Rightarrow}$ and subtyping relation $\leq$ to find the instantiation $\mathtt{Int} \to \mathtt{Int}$ for X that will work for both.

After digging into the application and synthesizing a type for f, we kick off the application judgment to see whether a function of type $\forall X.\, X \to X \to X$ can be applied to argument id. The sub-derivation for this is labelled $\mathcal{D}$, and as its first step it (non-deterministically) picks

$$\dfrac{\dfrac{\dfrac{\Psi(f) = \forall X.\, X \to X \to X}{\Psi \vdash_{\Uparrow} f : \forall X.\, X \to X \to X}\,\textsc{Var} \quad \mathcal{D}}{\Psi \vdash_{\Uparrow} f\ id : (\texttt{Int} \to \texttt{Int}) \to \texttt{Int} \to \texttt{Int}}\,\textsc{App} \quad \dfrac{\dfrac{\overline{\Psi \vdash_{\Uparrow} suc : \texttt{Int} \to \texttt{Int}}\,\textsc{Var} \quad \dfrac{\cdots}{\texttt{Int} \to \texttt{Int} \leq \texttt{Int} \to \texttt{Int}}}{\Psi \vdash_{\Downarrow} suc : \texttt{Int} \to \texttt{Int}}\,\textsc{Sub}}{\Psi \vdash_{\Rightarrow} ((\texttt{Int} \to \texttt{Int}) \to \texttt{Int} \to \texttt{Int}) \cdot suc : \texttt{Int} \to \texttt{Int}}\,{\to}\textsc{App}}{\Psi \vdash_{\Uparrow} f\ id\ suc : \texttt{Int} \to \texttt{Int}}\,\textsc{App}$$

$$\mathcal{D} = \dfrac{\dfrac{\dfrac{\overline{\Psi \vdash_{\Uparrow} id : \forall Y.\, Y \to Y}\,\textsc{Var} \quad \dfrac{\dfrac{\cdots}{[\texttt{Int}/Y]Y \to Y \leq \texttt{Int} \to \texttt{Int}}}{\forall Y.\, Y \to Y \leq \texttt{Int} \to \texttt{Int}}\,{\leq}\forall_{\textsc{L}}}{\Psi \vdash_{\Downarrow} id : \texttt{Int} \to \texttt{Int}}\,\textsc{Sub}}{\Psi \vdash_{\Rightarrow} [\texttt{Int} \to \texttt{Int}/X](X \to X \to X) \cdot id : (\texttt{Int} \to \texttt{Int}) \to \texttt{Int} \to \texttt{Int}}\,{\to}\textsc{App}}{\Psi \vdash_{\Rightarrow} \forall X.\, X \to X \to X \cdot id : (\texttt{Int} \to \texttt{Int}) \to \texttt{Int} \to \texttt{Int}}\,\forall\textsc{App}$$

$$\Psi = f{:}\forall X.\, X \to X \to X, id{:}\forall Y.\, Y \to Y, suc{:}\texttt{Int} \to \texttt{Int}$$

Figure 9: System **DK** Ex. 1

instantiation $\texttt{Int} \to \texttt{Int}$ for X. This results in *id* being checked against monomorphic type $\texttt{Int} \to \texttt{Int}$. Of course, *id synthesizes* polymorphic type $\forall Y.\, Y \to Y$, so these two types are mediated in rule Sub, where the type variable Y is eventually instantiated with Int through use of $\leq\forall\texttt{L}$ in the subtyping rules. This results in the application $f\ id$ having type $(\texttt{Int} \to \texttt{Int}) \to \texttt{Int} \to \texttt{Int}$, which in turn can be applied to the second argument *suc* of type $\texttt{Int} \to \texttt{Int}$.

Implementation As mentioned earlier, the algorithm for System **DK** uses global methods of constraint collection, which was hinted at when discussing rules $\forall$App and $\to$Syn. In the implementation, each specificational "guess" instantiating a type variable X is replaced by a freshly-generated "existential variable" $\widehat{X}$ which is added to a typing context Γ that now contains such variables as well as their solutions. The algorithmic judgments also produced "updated" contexts, as in $\Gamma \vdash_{\Uparrow} e : T \dashv \Delta$ where existentials in Γ have been refined or solved in Δ based on the shape of e. For example, in the algorithmic version of $\to$Syn,

$$\dfrac{\Gamma, \widehat{X}, \widehat{Y}, x{:}\widehat{X} \vdash_{\Downarrow} e : \widehat{Y} \dashv \Delta, x{:}\widehat{X}, \Theta}{\Gamma \vdash_{\Uparrow} \lambda x.\, e : \widehat{X} \to \widehat{Y} \dashv \Delta}$$

we can tell only that the subject should have a function type, but cannot tell what the domain and codomain should be, so the rule generates $\widehat{X}$ and $\widehat{Y}$ and tries to learn more about these when checking e. This is somewhat similar in effect to how System **OZZ** propagates partial type information – except that an output context in System **DK** can have unsolved existentials, whereas System **OZZ** insists that a complete type for the expression is known when we leave its AST node. However, the two systems have this in common – the implementation details of partial type propagation and type-argument inference are not needed to give a full account of the type inference system.

5.3 Discussion

Refactoring Theorems Part of the elegance of System **DK** is that it is easy to reason about where type annotations are required: "only on bindings of polymorphic type." [DK13] To help make this intuitive account more precise, the authors provide a number of "refactoring" and "annotation removal" theorems. For example, the refactoring theorem

$$\text{If } \Psi \vdash_{\Uparrow} e : A \text{ and } \Psi \vdash_{\Downarrow} [(e{:}A)/x]e' : C \text{ then } \Psi, x{:}A \vdash_{\Uparrow} e' : C$$

reads as "any re-occurring annotated expression can be factored out of a program to a let binding", and the annotation-removal theorem

$$\text{If } \Psi \vdash_{\Uparrow} e_1\ (e_2{:}A) : C \text{ then } \Psi \vdash_{\Uparrow} e_1\ e_2 : C$$

reads as "function arguments *never* need annotations." The upshot of all this is that, using their specificational type systems, the authors can give a precise account of where annotations are required, where they may be removed, and the program transformations that preserve typeability – and they did *that* by using bidirectionality to carefully control how polytypes were inferred.

Checked applications The reader may wonder whether it is significant that applications in System **DK** can only synthesize their types, whereas in Systems **PT** and **OZZ** we saw that *checking* an application greatly affected type-argument inference by relaxing the minimality constraint on the application result type. System **DK** does not use any such side condition, so there is none to relax! But that begs the question – are multiple result types inferable for polymorphic function applications that a minimality condition could help make precise? The answer is no: *local* type inference *must* make a decision before leaving a sub-expression, whereas (the implementation of) System **DK** can propagate "under-specified" instantiations for type variables through the whole program until it learns the exact monotype needed. That said, *checking* such an under-specified synthesizing application with a monotype forces only one type instantiation be legal for the affected type variables, so in this sense checking mode plays a related role – helping the constraint solver infer type arguments in polymorphic function applications.

6 Conclusion

This report has examined *bidirectionality* and its role in type inference systems. The defining characteristic of bidirectional typing is the *local* propagation of typing information from two sources: the term itself ("synthesis") and from the context surrounding it ("checking"). This characterization of what bidirectionality *is* does not seem to be, at face value, related to one of its most important *uses*: to help provide a high-level, specificational account of type inference for both users and implementers. This seeming gap is closed when we introduce any form of *subtyping* into our type system, and throughout this report the mantra has been "subsumption of the type of a term is governed entirely by the term's context". The details for where and how subsumption occurs are effectively communicated by bidirectional inference rules, which give a greater degree of precision in describing the flow of typing information without reference to the implementation. Importantly, in all three works considered the specificational bidirectional type systems were shown *sound* and *complete* with respect to their algorithmic versions.

Bidirectionality can be combined with either *local* or *global* type inference methods. The first two systems covered in this report, System **PT** and **OZZ**, are forms of *local type inference* because they rely on bidirectionality and *local type-argument inference* only. Though languages that use local type inference require more type annotations than those using global inference, they also tend to be more easily extended to have richer type languages (e.g. impredicative System $F_{\leq}$), without threatening the decidability, or even comprehensibility, of the type inference system – after all, the presence of rich subtyping introduces some ambiguity in what types are inferred for some expressions, which bidirectionality helped to tame. When it is used in combination with more global methods of type inference, bidirectionality still helps to provide a full account of the system. In System **DK**, it was used to both describe and restrict a more advanced form of type inference that better supports *higher-rank polymorphic types*.

I close with a quote from [Dun09], a work proceeding and heavily influencing [DK13], that I myself have taken to heart: "*I attribute the virtues of my work to the essential simplicity of bidirectional typechecking... To designers of languages and type systems, consider bidirectional typechecking; as your type system becomes more powerful, you will likely outgrow Damas-Milner inference, and making it bidirectional from the beginning should lead to a cleaner and more logical system than what you get after retrofitting bidirectionality.*"

References

[Chr13] David Christiansen. Bidirectional typing rules: A tutorial. http://davidchristiansen.dk/tutorials/bidirectional.pdf, October 2013.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584–590, 1934.

[DK13] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. *SIGPLAN Not.*, 48(9):429–442, September 2013.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[Dun09] Joshua Dunfield. Greedy bidirectional polymorphism. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML*, ML '09, pages 15–26, New York, NY, USA, 2009. ACM.

[Mca00] Bruce J. Mcadam. Generalising techniques for type debugging. In *Trends in Functional Programming*, pages 49–57. Intellect, 2000.

[Ode00] Martin Odersky. Functional nets. In Gert Smolka, editor, *Programming Languages and Systems*, pages 1–25, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[Ode02] Martin Odersky. Inferred type instantiation for GJ. Note sent to the types mailing list, January 2002.

[OZZ01] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. *SIGPLAN Not.*, 36(3):41–53, January 2001.

[Pfe04] Frank Pfenning. Lecture notes on bidirectional type checking. https://www.cs.cmu.edu/~fp/courses/15312-f04/handouts/15-bidirectional.pdf, October 2004.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[PJVWS07] Simon Peyton-Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, January 2007.

[PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000.

[TU96] J. Tiuryn and P. Urzyczyn. The subtyping problem for second-order types is undecidable. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Jul 1996.

[Wel98] J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *ANNALS OF PURE AND APPLIED LOGIC*, 98:111–156, 1998.