Kind Inference for Datatypes

NINGNING XIE, The University of Hong Kong, China
RICHARD A. EISENBERG, Bryn Mawr College, USA and Tweag I/O
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China

In recent years, languages like Haskell have seen a dramatic surge of new features that significantly extends the expressive power of their type systems. With these features, the challenge of *kind inference* for datatype declarations has presented itself and become a worthy research problem on its own.

This paper studies kind inference for datatypes. Inspired by previous research on type-inference, we offer declarative specifications for what datatype declarations should be accepted, both for Haskell98 and for a more advanced system we call PolyKinds, based on the extensions in modern Haskell, including a limited form of dependent types. We believe these formulations to be novel and without precedent, even for Haskell98. These specifications are complemented with implementable algorithmic versions. We study *soundness*, *completeness* and the existence of *principal kinds* in these systems, proving the properties where they hold. This work can serve as a guide both to language designers who wish to formalize their datatype declarations and also to implementors keen to have principled inference of principal types.

CCS Concepts: • **Theory of computation → Type theory**; • **Software and its engineering → Functional languages**; **Polymorphism**; **Data types and structures**.

Additional Key Words and Phrases: Haskell, Dependent Types

ACM Reference Format:
Ningning Xie, Richard A. Eisenberg, and Bruno C. d. S. Oliveira. 2020. Kind Inference for Datatypes. *Proc. ACM Program. Lang.* 4, POPL, Article 53 (January 2020), 28 pages. https://doi.org/10.1145/3371121

1 INTRODUCTION

Modern functional languages such as Haskell, ML, and OCaml come with powerful forms of type inference. The global type-inference algorithms employed in those languages are derived from the Hindley-Milner type system (HM) [Damas and Milner 1982; Hindley 1969], with multiple extensions. As the languages evolve, researchers also formalize the key aspects of type inference for the new extensions. Common extensions of HM include *higher-ranked polymorphism* [Odersky and Läufer 1996; Peyton Jones et al. 2007] and *type-inference for GADTs* [Peyton Jones et al. 2006], which have both been formally studied thoroughly.

Most research work for extensions of HM so far has focused on forms of polymorphism (such as support for impredicativity [Le Botlan and Rémy 2003; Leijen 2009; Rémy and Yakobowski 2008; Serrano et al. 2018; Vytiniotis et al. 2008]), where type variables all have the same kind. In these systems, the type variables introduced by universal quantifiers and/or type declarations all stand for proper types (i.e., they have kind $\star$). In such a simplified setting, datatype declarations such as

Authors' addresses: Ningning Xie, The University of Hong Kong, Department of Computer Science, Hong Kong, China, nnxie@cs.hku.hk; Richard A. Eisenberg, Bryn Mawr College, Department of Computer Science, Bryn Mawr, PA, USA, Tweag I/O, rae@richarde.dev; Bruno C. d. S. Oliveira, The University of Hong Kong, Department of Computer Science, Hong Kong, China, bruno@cs.hku.hk.

2475-1421/2020/1-ART53
https://doi.org/10.1145/3371121

data *Maybe a* = *Nothing* | *Just a* pose no problem at all for type inference: with only one possible kind for *a*, there is nothing to infer.

However, real-world implementations for languages like Haskell support a non-trivial kind language, including kinds other than $\star$. Haskell98 accepts *higher-kinded polymorphism* [Jones 1995], enabling datatype declarations such as **data** *AppInt f* = *Mk* (*f Int*). The type of constructor *Mk* applies the type variable *f* to an argument *Int*. Accordingly, *AppInt Bool* would not work, as the type *Bool Int* (in the instantiated type of *Mk*) is invalid. Instead, we must write something like *AppInt Maybe*: the argument to *AppInt* must be suitable for applying to *Int*. In Haskell98, *AppInt* has kind $(\star \to \star) \to \star$. For Haskell98-style higher-kinded polymorphism, Jones [1995] presents one of the few extensions of HM that deals with a non-trivial language of kinds. His work addresses the related problem of inference for *constructor type classes*, although he does not show directly how to do inference for datatype declarations.

Modern Haskell[1] has a much richer type and kind language compared to Haskell98. In recent years, Haskell has seen a dramatic surge of new features that extend the expressive power of algebraic datatypes. Such features include *GADTs*, *kind polymorphism* [Yorgey et al. 2012] with *implicit kind arguments*, and *dependent kinds* [Weirich et al. 2013], among others. With great power comes great responsibility: now we must be able to infer these kinds, too. For instance, consider these datatype declarations:

```
data App f a = MkApp (f a)        data T = MkT1 (App Maybe Int)
data Fix f   = In (f (Fix f))             | MkT2 (App Fix Maybe)   -- accept or reject?
```

Should the declaration for *T* be accepted or rejected? In a Haskell98 setting, the kind of *App* is $(\star \to \star) \to \star \to \star$. Therefore *T* should be rejected, because in *MkT2* the datatype *App* is applied to *Fix* :: $(\star \to \star) \to \star$ and *Maybe* :: $\star \to \star$, which do not match the expected kinds of *App*. However, with kind polymorphism, *T* is accepted, because *App* has the more general kind $\forall k.\, (k \to \star) \to k \to \star$. With this kind, both uses of *App* in *T* are valid.

The questions we ask in this paper are these: *Which datatype declarations should be accepted? What kinds do accepted datatypes have?* Surprisingly, the literature is essentially silent on these questions—we are unaware of any formal treatment of kind inference for datatype declarations.

Inspired by previous research on type inference, we offer declarative specifications for two languages: Haskell98, as standardized [Peyton Jones 2003] (Section 3); and PolyKinds, a significant fragment of modern Haskell (Section 6). These specifications are complemented with algorithmic versions that can guide implementations (Sections 4 and 7). To relate the declarative and algorithmic formulations we study various properties, including *soundness*, *completeness*, and the existence of *principal kinds* (Sections 4.7, 5, and 7.6).

We offer the following contributions:

- **Kind inference for Haskell98:** We formalize Haskell98's datatype declarations, providing both a declarative specification and syntax-driven algorithm for kind inference. We prove that the algorithm is sound and observe how Haskell98's technique of defaulting unconstrained kinds to $\star$ leads to incompleteness. We believe that ours is the first formalization of this aspect of Haskell98. Its inclusion in this paper both sheds light on this historically important language and also prepares us for the more challenging features of modern Haskell.
- **Completeness for Haskell98 kind inference:** To model the Haskell98 behavior of defaulting declaratively, and thus to achieve completeness, Section 5 proposes a variant of the declarative system that adapts the *type parameters* approach from Garcia and Cimini [2015].
- **Kind inference for modern Haskell:** We present a type and kind language that is unified and dependently typed, modeling the challenging features for kind inference in modern

[1]We consider the Glasgow Haskell Compiler's implementation of Haskell, in version 8.8.

Haskell. We include both a declarative specification (Section 6) and a syntax-driven algorithm (Section 7). The algorithm is proved sound, and we observe where and why completeness fails. In the design of our algorithm, we must choose between completeness and termination; we favor termination but conjecture that an alternative design would regain completeness (Section 9). Unlike other dependently typed languages, we retain the ability to *infer* top-level kinds instead of relying on compulsory annotations.

- **Technical advances:** This work introduces a number of technical innovations that appear important in the implementation of type-inference for a dependently typed language. We expect implementations to have developed these ideas independently, but this paper provides their first known formalization. These innovations include *promotion* (Sections 4.6 and 7.4), *local scopes* and *moving* (Section 7.4), and the *quantification check* (Section 7.2). In addition, our kind-directed unification appears to risk divergence, yet we provide a subtle proof that it is indeed terminating.

Our type systems are detailed, and many rules are elided to save space. The full judgments—and all proofs of stated lemmas and theorems—are provided in the technical supplement[2]. In addition, we have included there a detailed comparison of our work here to the GHC implementation. It is our belief that this study can help inform the design of principled inference algorithms for languages beyond Haskell, as well as to guide the continued evolution of GHC's kind inference algorithm.

2 OVERVIEW

This section gives an overview of our work. We start by contrasting kind inference with type inference, and then summarize the key aspects of the two systems of datatypes that we develop.

2.1 Kind Inference in Haskell98

Haskell98's kind language contains a constant (the kind $\star$) and kinds built from arrows ($k_1 \to k_2$). Kind inference for Haskell98 datatypes is thus closely related to type inference for the simply typed λ-calculus (STLC). For example, consider a term $+ :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ and a type constructor $\oplus :: \star \to \star \to \star$. At the term level, we infer that $\mathit{add}\ a\ b = a + b$ yields $\mathit{add} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}$. Similarly, we can create a datatype $\mathbf{data}\ \mathit{Add}\ a\ b = \mathit{Add}\ (a \oplus b)$ and infer $\mathit{Add} :: \star \to \star \to \star$.

No principal types. Consider now the function definition $k\ a = 1$. In the STLC, there are infinitely many (incomparable) types that can be assigned to k, including $k :: \mathit{Int} \to \mathit{Int}$ and $k :: (\mathit{Int} \to \mathit{Int}) \to \mathit{Int}$. Assuming there are no type variables, the STLC accordingly has no *principal types*. An analogous datatype declaration is $\mathbf{data}\ K\ a = K\ \mathit{Int}$. As with k, there are infinitely many (incomparable) kinds that can be assigned to K, including $K :: \star \to \star$ and $K :: (\star \to \star) \to \star$.

Defaulting. Definitions like k (in STLC) or K (in Haskell98) do not have a principal type/kind, which raises the immediate question of what type/kind to infer. Haskell98 solves this problem by using a *defaulting* strategy: *if the kind of a type variable cannot be inferred, then it is defaulted to* $\star$. Therefore the kind of K in Haskell98 is $\star \to \star$. From the perspective of type inference, such defaulting strategy may seem somewhat ad-hoc, but due to the role that $\star$ plays at the type level it seems a defensible design for kind inference. Defaulting brings complications in writing a declarative specification. We discuss this point further in Section 4.3.

2.2 Kind Inference in Modern GHC Haskell

The type and kind languages for modern GHC are *unified* (i.e., types and kinds are indistinguishable), *dependently typed*, and the kind system includes the $\star :: \star$ axiom [Cardelli 1986; Weirich et al. 2013]. We informally use the word *type* or *kind* where we find it appropriate. Unlike Haskell98's datatypes,

[2] http://arxiv.org/abs/1911.06153.

whose inference problem is quite closely related to the well-studied inference problem for STLC, type inference for various features in modern Haskell is not well-studied. While we are motivated concretely by Haskell, many of the challenges we face would be present in any dependently typed language seeking principled type inference. We use the term PolyKinds to refer to the fragment of modern Haskell we model.[3] We enumerate the key features of this fragment below.

Kind polymorphism and dependent types. Global type inference, in the style of Damas and Milner [1982], allows polymorphic kinds to be assigned to datatype definitions. For instance, reconsider **data** $K\ a = K\ Int$. In PolyKinds, K can be given the kind $K :: \forall\{k\}.\ k \to \star$. This example shows one of the interesting new features of PolyKinds over Haskell98: *kind polymorphism* [Yorgey et al. 2012]. The polymorphic kind is obtained via *generalization*, which is a standard feature in Damas-Milner algorithms. Polymorphic types are helpful for recovering principal types, since they generalize many otherwise incomparable monomorphic types.

System-F-based languages do not have dependent types. In contrast, PolyKinds supports dependent kinds such as **data** $D :: \forall(k :: \star)\ (a :: k).\ K\ a \to \star$. There are two noteworthy aspects about the kind of D. Firstly, kind and type variables are *typed*: different type variables may have different kinds. Secondly, the kinds of later variables can *depend* on earlier ones. In D, the kind of a depends on k. Both typed variables and dependent kinds bring technical complications that do not exist in many previous studies of type inference (e.g., [Peyton Jones et al. 2007; Vytiniotis et al. 2011]).

First-order unification with dependent kinds and typed variables. Although PolyKinds is dependently typed, its unification problem is remarkably *first-order*. This is in contrast to many other dependently typed languages, where unification is usually *higher-order* [Andrews 1971; Huet 1973]. Since unification plays a central role in inference algorithms this is a crucial difference. Higher-order unification is well-known to be undecidable in the general case [Goldfarb 1981]. As a consequence, type-inference algorithms for most dependently typed languages make various trade-offs.

A key reason why unification can be kept as a first-order problem in PolyKinds is because the type language *does not include lambdas*. Type-level lambdas have been avoided since the start in Haskell, since they bring major challenges for (term-level) type inference [Jones 1995].

The unification problem for PolyKinds is still challenging, compared to unification for System-F-like languages: unification must be *kind-directed*, as first observed at the term level by Jones [1995]. Consider the following (contrived) example:

```
data X :: ∀a (b :: ⋆ → ⋆). a b → ⋆        -- accepted
data Y :: ∀(c :: Maybe Bool). X c → ⋆     -- rejected
```

In X's kind, we discover $a :: (\star \to \star) \to \star$. When checking Y's kind, we must infer how to instantiate X: that is, we must choose a and b so that $a\ b$ unifies with $Maybe\ Bool$, which is c's kind. It is tempting to solve this with $a \mapsto Maybe$ and $b \mapsto Bool$, but doing so would be ill-kinded, as a and $Maybe$ have different kinds. Our unification thus features *heterogeneous constraints* [Gundry 2013]. When solving a unification variable, we need to first unify the kinds on both sides.

Because unification recurs into kinds, and because types are undifferentiated from kinds, it might seem that unification might not terminate. In Section 7.4 we show that the first-order unification with heterogeneous constraints employed in PolyKinds is guaranteed to terminate.

Mutual and polymorphic recursion. Recursion and mutual recursion are omnipresent in datatype declarations. In PolyKinds, mutually recursive definitions will be kinded together and then get generalized together. For example, both P and Q get kind $\forall(k :: \star).\ k \to \star$.

[3] Some of the features we model are slightly different in our presentation than they exist in GHC. The technical supplement outlines the differences. These minor differences do not affect the applicability of our work to improving the GHC implementations, but they may affect the ability to test our examples in GHC.

data *P a* = *MkP* (*Q a*)
data *Q a* = *MkQ* (*P a*)

The recursion is simple here: all recursive occurrences are at the same type. In existing type-inference algorithms, such recursive definitions are well understood and do not bring considerable complexity to type inference. However, we must also consider *polymorphic recursion* as in *Poly*:

data *Poly* :: ∀*k*. *k* → ⋆
data *Poly k* = *C1* (*Poly Int*) | *C2* (*Poly Maybe*)

This example includes a *kind signature*, meaning that we must *check* the kind of the datatype, not *infer* it. In the definition of *Poly*, the type *Poly Int* requires an instantiation $k \mapsto \star$, while the type *Poly Maybe* requires an instantiation of $k \mapsto (\star \to \star)$. These differing instantiations cause the declaration to be polymorphic recursive.

PolyKinds deals with such cases of polymorphic recursion, which also appear at the term level—for example, when writing recursive functions over GADTs or nested datatypes [Bird and Meertens 1998]. Polymorphic recursion is known to render type-inference undecidable [Henglein 1993]. Furthermore, most existing formalizations of type inference avoid the question entirely, either by not modeling recursion at all or not allowing polymorphic recursion. Our PolyKinds system has full support for polymorphic recursion, implemented directly without the use of a *fix* operator. Polymorphic recursion is allowed only on datatypes with a kind signature; other datatypes are treated as monomorphic during inference.

Visible kind application. PolyKinds lifts *visible type application (VTA)* [Eisenberg et al. 2016], whereby we can explicitly instantiate a function call, as in *id* @*Bool True*, to kinds, giving us *visible kind application (VKA)*. Following the design of VTA, we distinguish *specified variables* from *inferred variables*. As described by Eisenberg et al. [2016, Section 3.1], only specified variables can be instantiated via VKA. Instantiation of variables is inferred when no explicit kind application is given. To illustrate, consider **data** *T* :: ∀*a b*. *a b* → ⋆. Here, *a* and *b* are specified variables. Because their order is given, explicit instantiation of *a* must happen before *b*. For example, *T* @*Maybe* instantiates *a* to *Maybe*. On the other hand, the kind of *a* and *b* can be generalized to *a* :: *k* → ⋆ and *b* :: *k*. Elaborating the kind of *T*, we write *T* :: ∀{*k* :: ⋆} (*a* :: *k* → ⋆) (*b* :: *k*). *a b* → ⋆. The variable *k* is *inferred* and is not available for instantiation with VKA. This split between specified and inferred variables supports predictable type inference: if the variables invented by the compiler (e.g., *k*) were available for instantiation, then we have no way of knowing what order to instantiate them.

Open kind signatures and generalization order. Echoing the design of Haskell, PolyKinds supports *open kind signatures*. We say a signature is *closed* if it contains no free variables (e.g., **data** *T* :: ∀*a*. *a* → ⋆). Otherwise, it is *open* (e.g., **data** *Q* :: ∀(*a* :: (*f b*)) (*c* :: *k*). *f c* → ⋆). Free variables (in this case, *f*, *b*, *k*) will be generalized over. We have a decision to make: in which order do we generalize the free variables? This question is non-trivial, as there can be dependency between the variables. We infer *k* :: ⋆, *f* :: *k* → ⋆, *b* :: *k*. Even though *f* and *b* appear before *k*, their kinds end up depending on *k* and we must quantify *k* before *f* and *b*. Inferring this order is a challenge: we cannot know the correct order before completing inference. We thus introduce *local scopes*, which are sets of variables that may be reordered. Since the ordering is not fixed by the programmer, these variables are considered *inferred*, not *specified*, with respect to VKA.

Existential quantification. PolyKinds supports existentially quantified variables on datatype constructors. This is useful, for example, to model GADTs. Given **data** *T1* = ∀*a*. *MkT1 a*, we get *MkT1* :: ∀(*a* :: ⋆). *a* → *T1*. The type of the data constructor declaration can also be generalized. Given **data** *P1* :: ∀(*a* :: ⋆). ⋆, from **data** *T2* = *MkT2 P1*, we infer *MkT2* :: ∀{*a* :: ⋆}. *P1* @*a* → *T2*, where *P1* is elaborated to *P1* @*a* with *a* generalized as an inferred variable.

program	pgm	::=	$\mathbf{rec}\ \overline{\mathcal{T}_i}^{\,i}\ ;\ pgm \mid e$	polytype	σ	::=	$\forall \overline{a_i : \kappa_i}^{\,i}.\tau$
datatype decl.	$\mathcal{T}$	::=	$\mathbf{data}\ T\ \overline{a_i}^{\,i} = \overline{\mathcal{D}_j}^{\,j}$	monotype	τ	::=	$\mathsf{Int} \mid a \mid T \mid \tau_1\ \tau_2 \mid \rightarrow$
data c'tor decl.	$\mathcal{D}$	::=	$D\ \overline{\tau_i}^{\,i}$	kind	κ	::=	$\star \mid \kappa_1 \rightarrow \kappa_2$
expression	e	::=	$\ldots$	term context	Ψ	::=	$\bullet \mid \Psi, D : \sigma$
				type context	Σ	::=	$\bullet \mid \Sigma, a : \kappa \mid \Sigma, T : \kappa$

$\boxed{\Sigma; \Psi \vdash^{\mathsf{pgm}} pgm : \sigma}$ *(Typing Program)*

PGM-EXPR
$$\frac{\Sigma; \Psi \vdash e : \sigma}{\Sigma; \Psi \vdash^{\mathsf{pgm}} e : \sigma}$$

PGM-DT
$$\frac{\Sigma' = \Sigma, \overline{T_i : \kappa_i}^{\,i} \qquad \overline{\Sigma' \vdash^{\mathsf{dt}} \mathcal{T}_i \rightsquigarrow \Psi_i}^{\,i} \qquad \Sigma'; \Psi, \overline{\Psi_i}^{\,i} \vdash^{\mathsf{pgm}} pgm : \sigma}{\Sigma; \Psi \vdash^{\mathsf{pgm}} \mathbf{rec}\ \overline{\mathcal{T}_i}^{\,i}\ ; pgm : \sigma}$$

$\boxed{\Sigma \vdash^{\mathsf{dt}} \mathcal{T} \rightsquigarrow \Psi}$ *(Typing Datatype Decl.)*

DT-DECL
$$\frac{(T : \overline{\kappa_i}^{\,i} \rightarrow \star) \in \Sigma \qquad \overline{\Sigma, \overline{a_i : \kappa_i}^{\,i} \vdash^{\mathsf{dc}}_{T\ \overline{a_i}^{\,i}} \mathcal{D}_j \rightsquigarrow \tau_j}^{\,j}}{\Sigma \vdash^{\mathsf{dt}} \mathbf{data}\ T\ \overline{a_i}^{\,i} = \overline{\mathcal{D}_j}^{\,j} \rightsquigarrow \overline{D_j : \forall \overline{a_i : \kappa_i}^{\,i}.\tau_j}^{\,j}}$$

$\boxed{\Sigma \vdash^{\mathsf{dc}}_{\tau} \mathcal{D} \rightsquigarrow \tau'}$ *(Typing Data Constructor Decl.)*

DC-DECL
$$\frac{\Sigma \vdash^{\mathsf{k}} \overline{\tau_i}^{\,i} \rightarrow \tau : \star}{\Sigma \vdash^{\mathsf{dc}}_{\tau} D\ \overline{\tau_i}^{\,i} \rightsquigarrow \overline{\tau_i}^{\,i} \rightarrow \tau}$$

$\boxed{\Sigma \vdash^{\mathsf{k}} \tau : \kappa}$ *(Kinding)*

K-VAR
$$\frac{(a : \kappa) \in \Sigma}{\Sigma \vdash^{\mathsf{k}} a : \kappa}$$

K-TCON
$$\frac{(T : \kappa) \in \Sigma}{\Sigma \vdash^{\mathsf{k}} T : \kappa}$$

K-NAT
$$\frac{}{\Sigma \vdash^{\mathsf{k}} \mathsf{Int} : \star}$$

K-ARROW
$$\frac{}{\Sigma \vdash^{\mathsf{k}} \rightarrow : \star \rightarrow \star \rightarrow \star}$$

K-APP
$$\frac{\Sigma \vdash^{\mathsf{k}} \tau_1 : \kappa_1 \rightarrow \kappa_2 \quad \Sigma \vdash^{\mathsf{k}} \tau_2 : \kappa_1}{\Sigma \vdash^{\mathsf{k}} \tau_1\ \tau_2 : \kappa_2}$$

Fig. 1. Declarative specification of Haskell98 datatype declarations

2.3 Desirable Properties for Kind Inference

One goal in writing this paper is to provide concrete, principled guidance to implementors of dependently typed languages, such as GHC/Haskell. It is thus important to be able to describe our inference algorithm as sound and complete against a *declarative specification*. This declarative specification is what we might imagine a programmer to have in her head as she programs. This system should be designed with a minimum of low-level detail and a minimum of surprises. It is then up to an algorithm to live up to the expectations set by the specification. The algorithm is sound when all programs it accepts are also accepted by the specification; it is complete when all programs accepted by the specification are accepted by the algorithm.

Why choose the particular set of features described here? Because they lead to interesting kind inference challenges. We have found that the features above are sufficient in exploring kind inference in modern Haskell. We consider unformalized extensions in Section 8.

3 DATATYPES IN HASKELL98

We begin our formal presentation with Haskell98. The fragment of the syntax of Haskell98 that concerns us appears at the top of Figure 1, including datatype declarations, types, kinds, and contexts. The metavariable e refers to expressions, but we do not elaborate the details of expressions' syntax or typing rules here. A program pgm is a sequence of groups (defined below) of datatype declarations $\mathcal{T}$, followed by an expression e. We write $\tau_1 \rightarrow \tau_2$ as an abbreviation for $(\rightarrow)\ \tau_1\ \tau_2$.

3.1 Groups and Dependency Analysis

Users are free to write declarations in any order: earlier declarations can depend on later ones in the same compilation unit. However, any kind-checking algorithm must process the declarations in dependency order. Complicating this is that type declarations may be mutually recursive. A formal analysis of this dependency analysis is not enlightening, so we consider it to be a preprocessing step that produces the grammar in Figure 1. This dependency analysis breaks up the (unordered) raw input into mutually recursive groups (potentially containing just one declaration), and puts these in dependency order. We use the term *group* to describe a set of mutually recursive declarations.

3.2 Declarative Typing Rules

The declarative typing rules are in Figure 1. There are no surprises here; we review these rules briefly. The top judgment is $\Sigma; \Psi \vdash^{\mathrm{pgm}} pgm : \sigma$. Its rule PGM-DT extends the input type context Σ with kinds for the datatype declarations to form Σ', which is used to check both the datatype declarations and the rest of the program. In rule PGM-DT, we implicitly extract the names $\overline{T}^{\,i}$ from the declarations $\overline{\mathcal{T}}^{\,i}$ (and use this abuse of notation throughout our work, relating T to $\mathcal{T}$ and D to $\mathcal{D}$). The kinds are *guessed* for an entire group all at once: they are added to the context *before* looking at the declarations. This is needed because the declarations in the group refer to one another. Guessing the right answer is typical of declarative type systems. The algorithmic system presented in Section 4 provides a mechanism for an implementation. Although there is no special judgment for typing a group of mutually recursive datatypes, we use $\Sigma \vdash^{\mathrm{grp}} \mathbf{rec}\, \overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\kappa_i}^{\,i}; \overline{\Psi_i}^{\,i}$ to denote that the kinding results of datatype declarations are $\overline{\kappa_i}^{\,i}$, and the output term contexts are $\overline{\Psi_i}^{\,i}$.

Declarations are checked with $\Sigma \vdash^{\mathrm{dt}} \mathcal{T} \rightsquigarrow \Psi$. This uses the guessed kinds to process the data constructors of a declaration, producing a term context Ψ with the data constructors and their types. The rule DT-DECL ensures that the datatype has an appropriate kind in the context and then checks data constructors using the $\vdash^{\mathrm{dc}}$ judgment. These checks are done in a type context extended with bindings for the type variables $\overline{a_i}^{\,i}$, where each a_i has a kind extracted from the guessed kind of the datatype T. The subscript on the $\vdash^{\mathrm{dc}}$ judgment is the return type of the constructors, whose types are easily checked by rule DC-DECL. The kinding judgment $\Sigma \vdash^{\mathrm{k}} \tau : \kappa$ is standard.

4 KIND INFERENCE FOR HASKELL98

We now present the algorithmic system for Haskell98. Of particular interest is the defaulting rule (Section 4.3), which means that these rules are not complete with respect to the declarative system.

4.1 Syntax

The top of Figure 2 describes the syntax of kinds and contexts in the algorithmic system for Haskell98. The differences from the declarative system are highlighted in gray. Kinds are extended with unification kind variables $\widehat{\alpha}$. Algorithmic contexts are also extended with unification kind variables, either unsolved ($\widehat{\alpha}$) or solved ($\widehat{\alpha} = \kappa$). Although the grammar for algorithmic term contexts Γ appears identical to that of declarative contexts, note that the grammar for κ has been extended; accordingly, algorithmic contexts Γ might include kinds with unification variables, while declarative contexts Ψ do not. This approach of recording unification variables and their solutions in the contexts is inspired by Gundry et al. [2010] and Dunfield and Krishnaswami [2013]. Importantly, an algorithmic context is an *ordered* list, which enforces that given $\Delta_1, \widehat{\alpha} = \kappa, \Delta_2$, the kind κ must be well-formed under Δ_1. This rules out solutions like $\widehat{\alpha} = \widehat{\alpha} \to \star$ or $\widehat{\alpha} = \widehat{\beta}, \widehat{\beta} = \widehat{\alpha}$. Complete contexts Ω are contexts with all unification variables solved.

kind	κ	::=	$\star \mid \kappa_1 \longrightarrow \kappa_2 \mid \widehat{\alpha}$
term context	Γ	::=	$\bullet \mid \Gamma, D : \sigma$
type context	Δ, Θ	::=	$\bullet \mid \Delta, a : \kappa \mid \Delta, T : \kappa \mid \Delta, \widehat{\alpha} \mid \Delta, \widehat{\alpha} = \kappa$
complete type context	Ω	::=	$\bullet \mid \Omega, a : \kappa \mid \Omega, T : \kappa \mid \Omega, \widehat{\alpha} = \kappa$

$\boxed{\Omega; \Gamma \Vdash^{\text{pgm}} pgm : \sigma}$ *(Typing Program)*

A-PGM-EXPR
$$\frac{[\Omega]\Omega; [\Omega]\Gamma \vdash e : \sigma}{\Omega; \Gamma \Vdash^{\text{pgm}} e : \sigma}$$

A-PGM-DT
$$\frac{\Theta_1 = \Omega, \overline{\widehat{\alpha}_i}^{\,i}, \overline{T_i : \widehat{\alpha}_i}^{\,i} \qquad \overline{\Theta_i \Vdash^{\text{dt}} \mathcal{T}_i \rightsquigarrow \Gamma_i \dashv \Theta_{i+1}}^{\,i} \qquad \Theta_{n+1} \longrightarrow\!\!\!\!\!\rightarrow \Omega' \qquad \Omega'; \Gamma, \overline{\Gamma_i}^{\,i} \Vdash^{\text{pgm}} pgm : \sigma}{\Omega; \Gamma \Vdash^{\text{pgm}} \mathbf{rec}\ \overline{\mathcal{T}_i}^{\,i \in 1..n}; pgm : \sigma}$$

$\boxed{\Delta \Vdash^{\text{dt}} \mathcal{T} \rightsquigarrow \Gamma \dashv \Theta}$ *(Typing Datatype Decl.)*

A-DT-DECL
$$\frac{(T : \kappa) \in \Delta \qquad \Delta, \overline{\widehat{\alpha}_i}^{\,i} \Vdash^{\text{u}} [\Delta]\kappa \approx (\overline{\widehat{\alpha}_i}^{\,i} \longrightarrow \star) \dashv \Theta_1, \overline{\widehat{\alpha}_i = \kappa_i}^{\,i} \qquad \overline{\Theta_j, \overline{a_i : \kappa_i}^{\,i} \Vdash^{\text{dc}}_{T\,\overline{a_i}^{\,i}} \mathcal{D}_j \rightsquigarrow \tau_j \dashv \Theta_{j+1}, \overline{a_i : \kappa_i}^{\,i}}^{\,j}}{\Delta \Vdash^{\text{dt}} \mathbf{data}\ T\,\overline{a_i}^{\,i} = \overline{\mathcal{D}_j}^{\,j \in 1..n} \rightsquigarrow \overline{D_j : \forall \overline{a_i : \kappa_i}^{\,i}.\tau_j}^{\,j} \dashv \Theta_{n+1}}$$

$\boxed{\Delta \Vdash^{\text{dc}}_{\tau} \mathcal{D} \rightsquigarrow \tau' \dashv \Theta}$ *(Typing Data Constructor Decl.)*

A-DC-DECL
$$\frac{\Delta \Vdash^{\text{k}} \overline{\tau_i}^{\,i} \longrightarrow \tau : \star \dashv \Theta}{\Delta \Vdash^{\text{dc}}_{\tau} D\,\overline{\tau_i}^{\,i} \rightsquigarrow \overline{\tau_i}^{\,i} \longrightarrow \tau \dashv \Theta}$$

Fig. 2. Algorithmic program typing in Haskell98

We use a hole notation for inserting or replacing declarations in the middle of a context. $\Delta[\Theta]$ means that Δ is of the form $\Delta_1, \Theta, \Delta_2$. To reduce clutter, when we have $\Delta[\widehat{\alpha}]$, we also use only Δ to refer to the same context. If we have $\Delta[\widehat{\alpha}] = \Delta_1, \widehat{\alpha}, \Delta_2$, then $\Delta[\widehat{\alpha} = \kappa] = \Delta_1, \widehat{\alpha} = \kappa, \Delta_2$. This notation allows multiple holes: $\Delta[\Theta_1][\Theta_2]$ means that Δ is of the form $\Delta_1, \Theta_1, \Delta_2, \Theta_2, \Delta_3$. For example, $\Delta[\widehat{\alpha}][\widehat{\beta}]$ is $\Delta_1, \widehat{\alpha}, \Delta_2, \widehat{\beta}, \Delta_3$. Critically, $\widehat{\alpha}$ appears before $\widehat{\beta}$.

Since type contexts carry solutions for unification variables, we use contexts as substitutions: $[\Delta]\kappa$ applies Δ to kind κ. Applying Δ substitutes all solved unification variables in its argument idempotently. If under a complete context Ω, a kind κ is well-formed, then $[\Omega]\kappa$ contains no unification variables and is thus a well-formed declarative kind. For term contexts, $[\Delta]\Gamma$ applies Δ to each kind in Γ. Similarly, if under Ω, a term context Γ is well-formed, then $[\Omega]\Gamma$ gives back a declarative term context. The notation $[\Omega]\Delta$ applies a complete context Ω to Δ. We apply Ω to the kind of type variables and type constructors in Δ and remove the binding of solved unification variables from Δ. The full definition of context substitution is in the technical supplement. As above, $[\Omega]\Delta$ is a declarative type context.

4.2 Algorithmic Typing Rules

Figure 2 presents the typing rules for programs, datatype declarations and data constructor declarations. As this work focuses on the problem of kind inference of datatypes, we reduce the expression typing to the declarative system (rule A-PGM-EXPR); note that the contexts used there are declarative, as explained above. For type-checking a group of mutually recursive datatypes (rule A-PGM-DT), we first assign each type constructor a unification variable $\widehat{\alpha}$, and then type-check ($\Vdash^{\text{dt}}$) each datatype definition (Section 4.4), producing the context Θ_{n+1}. Then we default (Section 4.3) all unsolved unification variables with $\star$ using $\Theta_{n+1} \longrightarrow\!\!\!\!\!\rightarrow \Omega$, and continue with the rest of the program. Defaulting here means that the constraints of one group do not propagate to the rest of the program; accordingly, the input context of $\Vdash^{\text{pgm}}$ is always a complete context. Echoing the

notation for the declarative system, we write $\Omega \Vdash^{\text{grp}} \mathbf{rec}\, \overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\kappa_i}^{\,i}; \overline{\Gamma_i}^{\,i} \dashv \Theta$ to denote that the results of type-checking a group of datatype declarations are the kinds $\overline{\kappa_i}^{\,i}$, the output term contexts $\overline{\Gamma_i}^{\,i}$, and the final output type context Θ.

4.3 Defaulting

One of the key properties of datatypes in Haskell98 is the *defaulting* rule. In a datatype definition, if a type parameter is not fully determined by the definitions in its mutually recursive group, it is defaulted to have kind $\star$.

Definition 4.1 (Defaulting, $\longrightarrow\!\!\!\!\!\rightarrow$). *An algorithmic context Δ is defaulted to a complete context Ω, written $\Delta \longrightarrow\!\!\!\!\!\rightarrow \Omega$ by replacing all unsolved unification variables $\widehat{\alpha}$ in Δ with $\widehat{\alpha} = \star$.*

To understand how this rule affects code in practice, consider the following definitions:

```
data Q1 a = MkQ1   -- Q1 :: (⋆ → ⋆)          data P1 a = MkP1 P2   -- P1 :: (⋆ → ⋆) → ⋆
data Q2   = MkQ2 (Q1 Maybe)   -- rejected     data P2   = MkP2 (P1 Maybe)   -- accepted
```

One might think that the result of checking *Q1* and *Q2* would be the same as checking *P1* and *P2*. However, this is not true. *Q1* and *Q2* are not mutually recursive: they will not be in the same group and are checked separately. In contrast, *P1* and *P2* are mutually recursive and are checked together. This difference leads to the rejection of *Q2*: after kinding *Q1*, the parameter *a* is defaulted to $\star$, and then *Q1 Maybe* fails to kind check. Our algorithm is a faithful model of datatypes in Haskell98, and this rejection is exactly what the step $\Theta_{n+1} \longrightarrow\!\!\!\!\!\rightarrow \Omega$ (in rule A-PGM-DT) brings.

Other design alternatives. One alternative design is to default in rule A-PGM-EXPR instead of rule A-PGM-DT, as shown in rule A-PGM-EXPR-ALT. This means constraints in one group propagate to other groups, but not to expressions. Then *Q2* above is accepted.

$$\frac{\Delta \longrightarrow\!\!\!\!\!\rightarrow \Omega \qquad [\Omega]\Omega; [\Omega]\Gamma \vdash e : \sigma}{\Delta; \Gamma \Vdash^{\text{pgm}} e : \sigma}\ \text{A-PGM-EXPR-ALT}$$

A second alternative is that defaulting happens at the very end of type-checking a compilation unit. In this scenario, we wait to commit to the kind of a datatype until checking expressions. Now we can accept the following program, which would otherwise be rejected. However, this strategy does not play along well with modular design, as it takes an extra action at a module boundary.

```
data Q1 a = MkQ1
mkQ1      = MkQ1 :: Q1 Maybe
```

In the rest of this section, we stay with the standard, doing defaulting as portrayed in Figure 2.

4.4 Checking Datatype Declarations

The judgment $\Delta \Vdash^{\text{dt}} \mathcal{T} \rightsquigarrow \Gamma \dashv \Theta$ checks the datatype declaration $\mathcal{T}$ under the input context Δ, returning a term context Γ and an output context Θ. Its rule A-DT-DECL first gets the kind κ of the the type constructor from the context. It then assigns a fresh unification variable $\widehat{\alpha}$ to each type parameter. The expected kind of the type constructor is $\overline{\widehat{\alpha}_i}^{\,i} \to \star$. The rule then unifies κ with $\overline{\widehat{\alpha}_i}^{\,i} \to \star$. Before unification, we apply the context to κ; unification (Section 4.6) requires its inputs to be inert with respect to the context substitution. Our implementation of unification guarantees that all the $\widehat{\alpha}_i$ will be solved, as reflected in the rule A-DT-DECL. The type parameters are added to the context to type check each data constructor. Checking the data constructor $\mathcal{D}_j$ returns its type τ_j and the context $\Theta_{j+1}, \overline{a_i : \widehat{\alpha}_i}^{\,i}$. Note that each output context must be of this form as no new entries are added to the end of the context during checking individual data constructors. We can then generalize the type τ_j over type parameters, returning Θ_{n+1} as the result context.

$\boxed{\Delta \Vdash^{\mathsf{k}} \tau : \kappa \dashv \Theta}$ *(Kinding)*

$$\textsc{a-k-arrow}\ \frac{}{\Delta \Vdash^{\mathsf{k}} (\longrightarrow) : \star \to \star \to \star \dashv \Delta} \qquad \textsc{a-k-tcon}\ \frac{(T : \kappa) \in \Delta}{\Delta \Vdash^{\mathsf{k}} T : \kappa \dashv \Delta} \qquad \textsc{a-k-nat}\ \frac{}{\Delta \Vdash^{\mathsf{k}} \mathsf{Int} : \star \dashv \Delta} \qquad \textsc{a-k-var}\ \frac{(a : \kappa) \in \Delta}{\Delta \Vdash^{\mathsf{k}} a : \kappa \dashv \Delta}$$

$$\textsc{a-k-app}\ \frac{\Delta \Vdash^{\mathsf{k}} \tau_1 : \kappa_1 \dashv \Theta_1 \qquad \Theta_1 \Vdash^{\mathsf{k}} \tau_2 : \kappa_2 \dashv \Theta_2 \qquad \Theta_2 \Vdash^{\mathsf{kapp}} [\Theta_2]\kappa_1 \bullet [\Theta_2]\kappa_2 : \kappa_3 \dashv \Theta}{\Delta \Vdash^{\mathsf{k}} \tau_1\ \tau_2 : \kappa_3 \dashv \Theta}$$

$\boxed{\Delta \Vdash^{\mathsf{kapp}} \kappa_1 \bullet \kappa_2 : \kappa \dashv \Theta}$ *(Application Kinding)*

$$\textsc{a-kapp-kuvar}\ \frac{\Delta[\widehat{\alpha}_1, \widehat{\alpha}_2, \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2] \Vdash^{\mathsf{u}} \widehat{\alpha}_1 \approx \kappa \dashv \Theta}{\Delta[\widehat{\alpha}] \Vdash^{\mathsf{kapp}} \widehat{\alpha} \bullet \kappa : \widehat{\alpha}_2 \dashv \Theta} \qquad \textsc{a-kapp-arrow}\ \frac{\Delta \Vdash^{\mathsf{u}} \kappa_1 \approx \kappa \dashv \Theta}{\Delta \Vdash^{\mathsf{kapp}} \kappa_1 \to \kappa_2 \bullet \kappa : \kappa_2 \dashv \Theta}$$

$\boxed{\Delta \Vdash^{\mathsf{u}} \kappa_1 \approx \kappa_2 \dashv \Theta}$ *(Kind Unification)*

$$\textsc{a-u-refl}\ \frac{}{\Delta \Vdash^{\mathsf{u}} \kappa \approx \kappa \dashv \Delta} \qquad \textsc{a-u-arrow}\ \frac{\Delta \Vdash^{\mathsf{u}} \kappa_1 \approx \kappa_3 \dashv \Theta_1 \qquad \Theta_1 \Vdash^{\mathsf{u}} [\Theta_1]\kappa_2 \approx [\Theta_1]\kappa_4 \dashv \Theta}{\Delta \Vdash^{\mathsf{u}} \kappa_1 \to \kappa_2 \approx \kappa_3 \to \kappa_4 \dashv \Theta}$$

$$\textsc{a-u-kvarL}\ \frac{\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \kappa \rightsquigarrow \kappa_2 \dashv \Theta[\widehat{\alpha}]}{\Delta[\widehat{\alpha}] \Vdash^{\mathsf{u}} \widehat{\alpha} \approx \kappa \dashv \Theta[\widehat{\alpha} = \kappa_2]} \qquad \textsc{a-u-kvarR}\ \frac{\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \kappa \rightsquigarrow \kappa_2 \dashv \Theta[\widehat{\alpha}]}{\Delta[\widehat{\alpha}] \Vdash^{\mathsf{u}} \kappa \approx \widehat{\alpha} \dashv \Theta[\widehat{\alpha} = \kappa_2]}$$

$\boxed{\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \kappa_1 \rightsquigarrow \kappa_2 \dashv \Theta}$ *(Promotion)*

$$\textsc{a-pr-star}\ \frac{}{\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \star \rightsquigarrow \star \dashv \Delta} \qquad \textsc{a-pr-arrow}\ \frac{\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \kappa_1 \rightsquigarrow \kappa_3 \dashv \Delta_1 \qquad \Delta_1 \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} [\Delta_1]\kappa_2 \rightsquigarrow \kappa_4 \dashv \Theta}{\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \kappa_1 \to \kappa_2 \rightsquigarrow \kappa_3 \to \kappa_4 \dashv \Theta}$$

$$\textsc{a-pr-kuvarL}\ \frac{}{\Delta[\widehat{\beta}][\widehat{\alpha}] \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\beta} \rightsquigarrow \widehat{\beta} \dashv \Delta[\widehat{\beta}][\widehat{\alpha}]} \qquad \textsc{a-pr-kuvarR}\ \frac{}{\Delta[\widehat{\alpha}][\widehat{\beta}] \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\beta} \rightsquigarrow \widehat{\beta}_1 \dashv \Delta[\widehat{\beta}_1, \widehat{\alpha}][\widehat{\beta} = \widehat{\beta}_1]}$$

Fig. 3. Algorithmic kinding, unification and promotion in Haskell98.

The data constructor declaration judgment $\Delta \Vdash^{\mathsf{dc}}_{\tau} \mathcal{D} \rightsquigarrow \tau' \dashv \Theta$ type-checks a data constructor, by simply checking that the expected type $\overline{\tau_i}^{\,i} \to \tau$ is well-kinded.

4.5 Kinding

The algorithmic kinding $\Delta \Vdash^{\mathsf{k}} \tau : \kappa \dashv \Theta$ is given in Figure 3. Most rules are self-explanatory. For applications (rule A-K-APP), we synthesize the type for an application $\tau_1\ \tau_2$, where τ_1 and τ_2 have kinds κ_1 and κ_2, respectively. The hard work is delegated to the *application kinding* judgment.

Application kinding $\Delta \Vdash^{\mathsf{kapp}} \kappa_1 \bullet \kappa_2 : \kappa \dashv \Theta$ says that, under the context Δ, applying an expression of kind κ_1 to an argument of kind κ_2 returns the result kind κ and an output context Θ. We require the invariants that $[\Delta]\kappa_1 = \kappa_1$ and $[\Delta]\kappa_2 = \kappa_2$. Therefore, if the kind is a unification variable $\widehat{\alpha}$ (rule A-KAPP-KUVAR), we know it must be an unsolved unification variable. Since we know κ_1 must be a function kind, we solve $\widehat{\alpha}$ using $\widehat{\alpha}_1 \to \widehat{\alpha}_2$, unify $\widehat{\alpha}_1$ with the argument kind κ, and return $\widehat{\alpha}_2$. Note that the unification variables $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$ are inserted in the *middle* of the context Δ; this allows us to remove the type variables from the end of the context in rule A-DT-DECL and also plays a critical role in maintaining unification variable scoping in the more complicated system we analyze later. If the kind of the function is not a unification variable, it must surely be a function kind $\kappa_1 \to \kappa_2$ (rule A-KAPP-ARROW), so we unify κ_1 with the known argument kind κ, returning κ_2.

4.6 Unification

The unification judgment $\Delta \Vdash^{\mu} \kappa_1 \approx \kappa_2 \dashv \Theta$ is given in Figure 3. The elaborate style of this judgment (and its helper judgment $\Vdash^{\mathsf{pr}}$) is overkill for Haskell98, but this design sets us up well to understand unification in the presence of our PolyKinds system, later. We require the preconditions that $[\Delta]\kappa_1 = \kappa_1$ and $[\Delta]\kappa_2 = \kappa_2$, so that every time we encounter a unification variable, we know it is unsolved. Rule A-U-REFL is our base case, and rule A-U-ARROW unifies the components of the arrow types. When unifying $\widehat{\alpha} \approx \kappa$ (rule A-U-KVARL), we cannot simply set $\widehat{\alpha}$ to κ, as κ might include variables bound to the *right* of $\widehat{\alpha}$. Instead, we need to *promote* ($\Vdash^{\mathsf{pr}}$) κ.

Promotion. The crucial observation of $\Vdash^{\mathsf{pr}}$ is that *the relative order between unification variables does not matter for solving a constraint.* Consider unifying $\widehat{\alpha}, \widehat{\beta} \Vdash^{\mu} \widehat{\alpha} \approx \widehat{\beta} \to \star$. We cannot set $\widehat{\alpha} = \widehat{\beta} \to \star$, as this is ill-scoped. However, the constraint is solvable, as one solution context can be $\widehat{\beta}_1, \widehat{\alpha} = \widehat{\beta}_1 \to \star, \widehat{\beta} = \widehat{\beta}_1$. In other words, although $\widehat{\beta} \to \star$ contains an out-of-scope variable $\widehat{\beta}$, we can solve the constraint by introducing a fresh in-scope variable $\widehat{\beta}_1$ and setting $\widehat{\beta} = \widehat{\beta}_1$.

The promotion judgment $\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \kappa_1 \rightsquigarrow \kappa_2 \dashv \Theta$ captures this observation. The judgment says that, under the context Δ, we promote the kind κ_1, yielding κ_2, so that κ_2 is well-formed in the prefix context of $\widehat{\alpha}$, while retaining $[\Theta]\kappa_1 = [\Theta]\kappa_2$. At a high-level, $\Vdash^{\mathsf{pr}}$ looks for free variables in κ_1. Kind constants are always well-formed (rule A-PR-STAR). Variables bound to the left of $\widehat{\alpha}$ in Δ are unaffected (rule A-PR-KUVARL), as they are already well-formed. In rule A-PR-KUVARR, a unification variable $\widehat{\beta}$ bound to the right of $\widehat{\alpha}$ in Δ is replaced by a fresh variable introduced to $\widehat{\alpha}$'s left. Promotion is a partial operation, as it requires $\widehat{\beta}$ either to be to the right or to the left of $\widehat{\alpha}$. There is yet another possibility: if $\widehat{\beta} = \widehat{\alpha}$, then no rule applies. This is a desired property, as the $\widehat{\beta} = \widehat{\alpha}$ case exactly corresponds to the "occurs-check" in a more typical presentation of unification. By preventing promoting $\widehat{\alpha}$ to the left of $\widehat{\alpha}$, we prevent the possibility of an infinite substitution when applying an algorithmic context. It is this promotion algorithm that guarantees that all the $\widehat{\alpha}_i$ will be solved in rule A-DT-DECL: those variables will appear to the right of the unification variable invented in rule A-PGM-DT and will be promoted (and thus solved).

Returning to the $\Vdash^{\mu}$ judgment, rule A-U-KVARL first promotes the kind κ, yielding κ_2, so that κ_2 is well-formed in the prefix context of $\widehat{\alpha}$. We can then set $\widehat{\alpha} = \kappa_2$ in the concluding context. Rule A-U-KVARR is symmetric to rule A-U-KVARL. Note that when unifying $\widehat{\alpha} \approx \widehat{\beta}$, either rule A-U-KVARL and rule A-U-KVARR could be tried; an implementation can arbitrarily choose between them.

4.7 Soundness and Completeness

The main theorem of soundness is for program typing:

Theorem 4.2 (Soundness of $\Vdash^{\mathsf{pgm}}$). *If* Ω ok, *and* $\Omega \Vdash^{\mathsf{ectx}} \Gamma$, *and* $\Omega; \Gamma \Vdash^{\mathsf{pgm}} pgm : \sigma$, *then* $[\Omega]\Omega; [\Omega]\Gamma \vdash^{\mathsf{pgm}} pgm : \sigma$.

This lemma statement refers to judgments Ω ok and $\Omega \Vdash^{\mathsf{ectx}} \Gamma$; these basic well-formedness checks are given in the technical supplement. Because the declarative judgment $\vdash^{\mathsf{pgm}}$ requires declarative contexts, we write $[\Omega]\Omega$ and $[\Omega]\Gamma$ in the conclusion, applying the complete algorithmic context Ω as a substitution to form a declarative context, free of unification variables.

The statement of completeness relies on the definition of *context extension* $\Delta \longrightarrow \Theta$. The judgment captures a process of *information increase*. The formal definition of context extension is given in the technical supplement, and its treatment is as in Dunfield and Krishnaswami [2013]. Intuitively, context extension preserves all information in Δ, and may increase the information by adding or solving more unification variables. In all the algorithmic judgments, the output context is an extension of the input context.

We prove that our system is complete only up to checking *a group of datatype declarations*.

Theorem 4.3 (Completeness of $\Vdash^{\text{grp}}$). *Given* Ω ok, *if* $[\Omega]\Omega \Vdash^{\text{grp}} \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\kappa_i}^{\,i}; \overline{\Psi_i}^{\,i}$, *then there exists* $\overline{\kappa_i'}^{\,i}, \overline{\Gamma_i}^{\,i}, \Theta$, *and* Ω', *such that* $\Omega \Vdash^{\text{grp}} \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\kappa_i'}^{\,i}; \overline{\Gamma_i}^{\,i} \dashv \Theta$, *where* $\Theta \longrightarrow \Omega'$, *and* $\overline{[\Omega']\kappa_i' = \kappa_i}^{\,i}$, *and* $\overline{\Psi_i = [\Omega']\Gamma_i}^{\,i}$.

The theorem statement uses the notational convenience for checking groups, defined in Section 3.2 and Section 4.2. The theorem states that for every possible declarative typing for a group, the algorithmic typing results can be extended to support the declarative typing.

Unfortunately, the typing program judgment $\vdash^{\text{pgm}}$ is incomplete, as our algorithm models defaulting, while the declarative system does not. (For example, the *Q1*/*Q2* example of Section 4.3 is accepted by the declarative system but rejected by both GHC and our algorithmic system.) As straightforward as the defaulting rule may seem, it is surprisingly hard to model in a declarative system. We remedy this in the next section.

5 TYPE PARAMETERS, PRINCIPAL KINDS AND COMPLETENESS IN HASKELL98

We have seen that our judgments for checking programs $\vdash^{\text{pgm}}$ and $\Vdash^{\text{pgm}}$ do not support completeness, because the declarative system cannot easily model the defaulting rule given in Section 4.3. In this section, we introduce *kind parameters*, inspired by type parameters in Garcia and Cimini [2015], and relate the defaulting rule to principal kinds to recover completeness.

5.1 Type Parameters

Consider the datatype **data** *App f a* = *MkApp* (*f a*) again. The parameter *a* in this example can be of any kind, including $\star$, $\star \to \star$, or others. To express this polymorphism without introducing first-class polymorphism, we endow the declarative system with a set of *kind parameters*. Importantly, kind parameters live only in our reasoning; users are not allowed to write any kind parameters in the source. We amend the definition of kinds in Figure 1 as follows.

kind parameter	P	$\in$	KPARAM
kind	κ	$::=$	$\star \mid \kappa_1 \to \kappa_2 \mid P$

Kind parameters are uninterpreted kinds: there is no special treatment of kind parameters in the type system. Think of them as abstract, opaque kind constants. Kind parameters are eliminated by substitutions S, which map kind parameters to kinds, and homomorphically work on kinds themselves. For example, *App* can be assigned kind $(P \to \star) \to P \to \star$. By substituting for P, we can get, for example, $(\star \to \star) \to \star \to \star$. Indeed, from $(P \to \star) \to P \to \star$ we can get all other possible kinds of *App*. This leads to the definition of *principal kinds* for a group; and to the property that for every well-formed group, there exists a list of principal kinds.

Definition 5.1 (Principal Kind in Haskell98 with Kind Parameters). *Given a context* Σ, *a group* $\mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i}$, *and a list of kinds* $\overline{\kappa_i}^{\,i}$, *we say that the* $\overline{\kappa_i}^{\,i}$ *are* principal kinds *of* Σ *and* $\mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i}$, *denoted as* $\Sigma \vdash \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \rightsquigarrow^{\text{p}} \overline{\kappa_i}^{\,i}$, *if* $\Sigma \Vdash^{\text{grp}} \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\kappa_i}^{\,i}; \overline{\Psi_i}^{\,i}$, *and whenever* $\Sigma \Vdash^{\text{grp}} \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\kappa_i'}^{\,i}; \overline{\Psi_i'}^{\,i}$ *holds, there exists some substitution* S, *such that* $\overline{S(\kappa_i) = \kappa_i'}^{\,i}$ *and* $\overline{S(\Psi_i) = \Psi_i'}^{\,i}$.

Theorem 5.2 (Principality of Haskell98 with Kind Parameters). *If* $\Sigma \Vdash^{\text{grp}} \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\kappa_i}^{\,i}; \overline{\Psi_i}^{\,i}$, *then there exists some* $\overline{\kappa_i'}^{\,i}$ *such that* $\Sigma \vdash \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \rightsquigarrow^{\text{p}} \overline{\kappa_i'}^{\,i}$.

5.2 Principal Kinds and Defaulting

Using the notion of kind parameters, we can now incorporate defaulting into the declarative specification of Haskell98. To this end, we define the defaulting kind parameter substitution $S^\star$:

Definition 5.3 (Defaulting Kind Parameter Substitution)**.** *Let* $S^\star \in \mathit{KPARAM} \to \kappa$ *denote the substitution that substitutes all kind parameters to* $\star$*.*

Using $S^\star$, we can rewrite rule PGM-DT. Noteworthy is the fact that kind parameters only live in the middle of the derivation (in the κ_i), but never appear in the results $S^\star(\kappa_i)$.

$$\frac{\Sigma \vdash^{\mathsf{grp}} \mathbf{rec}\, \overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\kappa_i}^{\,i}; \overline{\Psi_i}^{\,i} \qquad \Sigma \vdash \mathbf{rec}\, \overline{\mathcal{T}_i}^{\,i} \rightsquigarrow^{\mathsf{p}} \overline{\kappa_i}^{\,i} \qquad \Sigma, \overline{T_i : S^\star(\kappa_i)}^{\,i}; \Psi, \overline{S^\star(\Psi_i)}^{\,i} \vdash^{\mathsf{pgm}} pgm : \sigma}{\Sigma; \Psi \vdash^{\mathsf{pgm}} \mathbf{rec}\, \overline{\mathcal{T}_i}^{\,i}; pgm : \sigma}\ \textsc{pgm-dtp}$$

5.3 Completeness

The two versions of defaulting (the one above and $\Delta \longrightarrow\!\!\!\!\!\rightarrow \Omega$ of Section 4.2) are equivalent. This fact is embodied in the following theorem, stating that the algorithmic system is complete with respect to the declarative system with kind parameters.

Theorem 5.4 (Completeness of $\Vdash^{\mathsf{pgm}}$ with Kind Parameters)**.** *Given algorithmic contexts* Ω*,* Γ*, and a program pgm, if* $[\Omega]\Omega; [\Omega]\Gamma \vdash^{\mathsf{pgm}} pgm : \sigma$*, then* $\Omega; \Gamma \Vdash^{\mathsf{pgm}} pgm : \sigma$*.*

6 DECLARATIVE SYNTAX AND SEMANTICS OF POLYKINDS

Having set the stage for kind inference for datatypes in Haskell98, we now present the declarative PolyKinds system. Our syntax is given at the top of Figure 4. Compared to Haskell98, programs *pgm* now include datatype signatures $\mathcal{S}$. Data constructor declarations $\mathcal{D}$ support existential quantification. Types and kinds are collapsed into one level; σ and K are now synonymous metavariables and allow prenex polymorphism, where variables in a kind binder ϕ can optionally have kind annotations. Monotypes τ and κ allow visible kind applications $\tau_1\ @\tau_2$. Elaborated types μ, η are the result of elaboration, which decorates source types to make them fully explicit. This is done so that checking equality of elaborated types is straightforward. The syntax for elaborated types contains inferred polymorphism $\forall\{\phi^{\mathsf{c}}\}.\mu$, where complete free kind binders ϕ^{c} have all variables annotated. Elaborated monotypes ρ and ω share the same syntax as monotypes. We informally use only ρ or ω for elaborated monotypes.

6.1 Groups and Dependency Analysis

Decomposition of signatures and definitions allows a more fine-grained control of dependency analysis. If *T* has a signature, and *S* depends on *T*, then we can kind-check *S* without inspecting the definition of *T*, because we know the kind of *T*. In other words, *S* only depends on the *signature* of *T*, not the *definition* of *T*. The complete dependency analysis rule, inspired by Jones [1999, Section 11.6.3], is:

Definition 6.1 (Dependency Analysis in PolyKinds)**.**

(i) If the signature/definition of T_1 *mentions* T_2*, then:*
  (a) if T_2 *has a signature, the signature/definition of* T_1 *depends on the signature of* T_2*;*
  (b) otherwise, the signature/definition of T_1 *depends on the definition of* T_2*.*
(ii) A definition depends on its signature.

To avoid a type that mentions itself in its own kind, we disallow self-dependency or mutual dependency involving signatures. For example, a group **data** *T1* :: *T2 a* → ⋆; **data** *T2* :: *T1* → ⋆ is rejected, lest *T1* be assigned type ∀(*a* :: *T1*). *T2 a* → ⋆. In other words, signatures do not form groups: they are always processed individually. Moreover, the definition of a datatype which has a signature does not join others in a group, as according to Definition 6.1, there will be no dependency from datatypes on it. This simplifies the kinding procedure, as we will see in the coming section.

program	pgm	::=	$\mathbf{sig}\,\mathcal{S}; pgm \mid \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i}; pgm \mid e$
datatype signature	$\mathcal{S}$	::=	$\mathbf{data}\ T : \sigma$
datatype decl.	$\mathcal{T}$	::=	$\mathbf{data}\ T\,\overline{a_i}^{\,i} = \overline{\mathcal{D}_j}^{\,j}$
data constructor decl.	$\mathcal{D}$	::=	$\forall\phi.D\,\overline{\tau_i}^{\,i}$
type, kind	σ, K	::=	$\forall\phi.\ \sigma \mid \tau$
monotype, monokind	$\tau, \kappa, \rho, \omega$	::=	$\star \mid \mathsf{Int} \mid a \mid T \mid \tau_1\,\tau_2 \mid \tau_1\,@\tau_2 \mid \rightarrow$
elaborated type, kind	μ, η	::=	$\forall\{\phi^{\mathsf{c}}\}.\mu \mid \forall\phi^{\mathsf{c}}.\ \mu \mid \rho$
term context	Ψ	::=	$\bullet \mid \Psi, D : \mu$
type context	Σ	::=	$\bullet \mid \Sigma, a : \rho \mid \Sigma, T : \eta$
kind binder list	ϕ	::=	$\bullet \mid \phi, a \mid \phi, a : \kappa$
complete kind binder list	ϕ^{c}	::=	$\bullet \mid \phi^{\mathsf{c}}, a : \rho$

$\boxed{\Sigma; \Psi \vdash^{\mathsf{pgm}} pgm : \sigma}$ *(Typing Program)*

PGM-SIG
$$\frac{\Sigma \vdash^{\mathsf{sig}} \mathcal{S} \rightsquigarrow T : \eta \quad \Sigma, T : \eta; \Psi \vdash^{\mathsf{pgm}} pgm : \mu}{\Sigma; \Psi \vdash^{\mathsf{pgm}} \mathbf{sig}\,\mathcal{S}; pgm : \mu}$$

PGM-DT-TTS
$$\frac{(T : \eta) \in \Sigma \quad \Sigma \vdash^{\mathsf{dt}} \mathcal{T} \rightsquigarrow \Psi_1 \quad \Sigma; \Psi, \Psi_1 \vdash^{\mathsf{pgm}} pgm : \mu}{\Sigma; \Psi \vdash^{\mathsf{pgm}} \mathbf{rec}\,\mathcal{T}; pgm : \mu}$$

PGM-DT-TT
$$\frac{\dfrac{\overline{\Sigma,\ \phi_i^{\mathsf{c}} \vdash^{\mathsf{ela}} \omega_i : \star}^{\,i} \quad \overline{\phi_i^{\mathsf{c}} \in \mathcal{Q}(\omega_i)}^{\,i}}{\Sigma,\ \cup\overline{\phi_i^{\mathsf{c}}}^{\,i},\ \overline{T_i : \omega_i}^{\,i} \vdash^{\mathsf{gen}}_{\phi_i^{\mathsf{c}}} \Psi_i \rightsquigarrow \Psi_i'}^{\,i} \qquad \Sigma,\ \overline{T_i : \forall\{\phi_i^{\mathsf{c}}\}.\omega_i}^{\,i}; \Psi,\ \Psi_i'[\,\overline{T_i \mapsto T_i\,@\phi_i^{\mathsf{c}}}^{\,i}\,] \vdash^{\mathsf{pgm}} pgm : \sigma}{\Sigma; \Psi \vdash^{\mathsf{pgm}} \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i}; pgm : \sigma}$$

$\boxed{\Sigma \vdash^{\mathsf{sig}} \mathcal{S} \rightsquigarrow T : \eta}$ *(Typing Signature)*

SIG-TT
$$\frac{\rceil\sigma\lceil \qquad \phi \in \mathcal{Q}(\sigma) \qquad \phi_1^{\mathsf{c}} \in \mathcal{Q}(\forall\phi^{\mathsf{c}}.\ \eta) \qquad \Sigma, \phi_1^{\mathsf{c}} \vdash^{\mathsf{k}} \forall\phi.\ \sigma : \star \rightsquigarrow \forall\phi^{\mathsf{c}}.\ \eta \qquad |\phi| = |\phi^{\mathsf{c}}|}{\Sigma \vdash^{\mathsf{sig}} \mathbf{data}\ T : \sigma \rightsquigarrow T : \forall\{\phi_1^{\mathsf{c}}\}.\forall\{\phi^{\mathsf{c}}\}.\eta}$$

$\boxed{\Sigma \vdash^{\mathsf{dt}} \mathcal{T} \rightsquigarrow \Psi}$ *(Typing Datatype Decl.)*

DT-TT
$$\frac{(T : \forall\{\phi_1^{\mathsf{c}}\}.\forall\phi_2^{\mathsf{c}}.\ \overline{\omega_i}^{\,i} \rightarrow \star) \in \Sigma \qquad \overline{\Sigma, \phi_1^{\mathsf{c}}, \phi_2^{\mathsf{c}}, \overline{a_i : \omega_i}^{\,i} \vdash^{\mathsf{dc}}_{(T\ @\phi_1^{\mathsf{c}}\ @\phi_2^{\mathsf{c}}\ \overline{a_i}^{\,i})} \mathcal{D}_j \rightsquigarrow \mu_j}^{\,j}}{\Sigma \vdash^{\mathsf{dt}} \mathbf{data}\ T\,\overline{a_i}^{\,i} = \overline{\mathcal{D}_j}^{\,j} \rightsquigarrow \overline{D_j : \forall\{\phi_1^{\mathsf{c}}\}.\forall\phi_2^{\mathsf{c}}.\ \forall\overline{a_i : \omega_i}^{\,i}.\mu_j}^{\,j}}$$

$\boxed{\Sigma \vdash^{\mathsf{dc}}_{\rho} \mathcal{D} \rightsquigarrow \mu}$ *(Typing Data Constructor Decl.)*

DC-TT
$$\frac{\phi^{\mathsf{c}} \in \mathcal{Q}(\mu\backslash_{\Sigma,\overline{\tau_i}^{\,i}}) \qquad \Sigma, \phi^{\mathsf{c}} \vdash^{\mathsf{k}} \forall\phi.\overline{\tau_i}^{\,i} \rightarrow \rho : \star \rightsquigarrow \mu}{\Sigma \vdash^{\mathsf{dc}}_{\rho} \forall\phi.D\,\overline{\tau_i}^{\,i} \rightsquigarrow \forall\{\phi^{\mathsf{c}}\}.\mu}$$

$\boxed{\Sigma \vdash^{\mathsf{gen}}_{\phi^{\mathsf{c}}} \Psi_1 \rightsquigarrow \Psi_2}$ *(Generalization)*

GEN
$$\frac{\overline{\phi^{\mathsf{c}}, \phi_i^{\mathsf{c}} \in \mathcal{Q}(\mu_i)}^{\,i}}{\Sigma \vdash^{\mathsf{gen}}_{\phi^{\mathsf{c}}} \overline{D_i : \mu_i}^{\,i} \rightsquigarrow \overline{D_i : \forall\{\phi^{\mathsf{c}}, \phi_i^{\mathsf{c}}\}.\mu_i}^{\,i}}$$

Fig. 4. Declarative specification of PolyKinds

6.2 Checking Programs

The declarative typing rules appear in Figure 4. The judgment $\Sigma; \Psi \vdash^{\mathsf{pgm}} pgm : \sigma$ checks the program. From now on we omit the typing rule for expressions in programs, which is essentially the same as in Haskell98. Rule PGM-SIG processes kind signatures by elaborating and generalizing the kind, then adding it to the context Σ. The helper judgment $\Sigma \vdash^{\mathsf{sig}} \mathcal{S} \rightsquigarrow T : \eta$ checks a kind

$\boxed{\Sigma \vdash^{\mathsf{inst}} \mu_1 : \eta \sqsubseteq \omega \rightsquigarrow \mu_2}$ *(Instantiation)*

$$\textsc{inst-refl}\ \frac{}{\Sigma \vdash^{\mathsf{inst}} \mu : \omega \sqsubseteq \omega \rightsquigarrow \mu} \qquad \textsc{inst-forall}\ \frac{\Sigma \vdash^{\mathsf{ela}} \rho : \omega_1 \qquad \Sigma \vdash^{\mathsf{inst}} \mu_1\ @\rho : \eta[a \mapsto \rho] \sqsubseteq \omega_2 \rightsquigarrow \mu_2}{\Sigma \vdash^{\mathsf{inst}} \mu_1 : \forall a : \omega_1.\eta \sqsubseteq \omega_2 \rightsquigarrow \mu_2}$$

$\boxed{\Sigma \vdash^{\mathsf{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu}$ *(Kind Checking)*

$$\textsc{kc-sub}\ \frac{\Sigma \vdash^{\mathsf{k}} \sigma : \eta \rightsquigarrow \mu_1 \qquad \Sigma \vdash^{\mathsf{inst}} \mu_1 : \eta \sqsubseteq \omega \rightsquigarrow \mu_2}{\Sigma \vdash^{\mathsf{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu_2}$$

$\boxed{\Sigma \vdash^{\mathsf{k}} \sigma : \eta \rightsquigarrow \mu}$ *(Kinding)*

$$\textsc{ktt-star}\ \frac{}{\Sigma \vdash^{\mathsf{k}} \star : \star \rightsquigarrow \star} \qquad \textsc{ktt-app}\ \frac{\Sigma \vdash^{\mathsf{k}} \tau_1 : \eta_1 \rightsquigarrow \rho_1 \qquad \Sigma \vdash^{\mathsf{inst}} \rho_1 : \eta_1 \sqsubseteq (\omega_1 \to \omega_2) \rightsquigarrow \rho_2 \qquad \Sigma \vdash^{\mathsf{kc}} \tau_2 \Leftarrow \omega_1 \rightsquigarrow \rho_3}{\Sigma \vdash^{\mathsf{k}} \tau_1\ \tau_2 : \omega_2 \rightsquigarrow \rho_2\ \rho_3}$$

$$\textsc{ktt-kapp}\ \frac{\Sigma \vdash^{\mathsf{k}} \kappa_1 : \forall a : \omega.\eta \rightsquigarrow \rho_1 \qquad \Sigma \vdash^{\mathsf{kc}} \kappa_2 \Leftarrow \omega \rightsquigarrow \rho_2}{\Sigma \vdash^{\mathsf{k}} \kappa_1\ @\kappa_2 : \eta[a \mapsto \rho_2] \rightsquigarrow \rho_1\ @\rho_2} \qquad \textsc{ktt-foralli}\ \frac{\Sigma \vdash^{\mathsf{ela}} \omega : \star \qquad \Sigma, a : \omega \vdash^{\mathsf{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu}{\Sigma \vdash^{\mathsf{k}} \forall a.\sigma : \star \rightsquigarrow \forall a : \omega.\mu}$$

$\boxed{\Sigma \vdash^{\mathsf{ela}} \mu : \eta}$ *(Elaborated Kinding)*

$$\textsc{ela-app}\ \frac{\Sigma \vdash^{\mathsf{ela}} \rho_1 : \omega_1 \to \omega_2 \qquad \Sigma \vdash^{\mathsf{ela}} \rho_2 : \omega_1}{\Sigma \vdash^{\mathsf{ela}} \rho_1\ \rho_2 : \omega_2} \qquad \textsc{ela-kapp}\ \frac{\Sigma \vdash^{\mathsf{ela}} \rho_1 : \forall a : \omega.\eta \qquad \Sigma \vdash^{\mathsf{ela}} \rho_2 : \omega}{\Sigma \vdash^{\mathsf{ela}} \rho_1\ @\rho_2 : \eta[a \mapsto \rho_2]}$$

Fig. 5. Selected rules for declarative kind-checking in PolyKinds

signature **data** $T : \sigma$. First, it uses $\rceil\sigma\lceil$ to ensure σ returns $\star$: $\rceil\sigma\lceil$ simply traverses over arrows and foralls, checking that the final kind of σ is $\star$. Then, as σ may be an open kind signature, it extracts the free kind variables $\phi \in Q(\sigma)$, where $Q(\sigma)$ is the set of all well-formed orderings of the free variables (transitively looking into variables' kinds) of σ; thus, ϕ is one such ordering. As discussed in Section 2.2, variables in ϕ are *inferred* so we accept any relative order, as long as it features the necessary dependency between the variables. Then the rule tries to elaborate ($\vdash^{\mathsf{k}}$) the kind $\forall\phi.\ \sigma$, where ϕ and ϕ^{c} have the same length ($|\phi| = |\phi^{\mathsf{c}}|$). As the elaborated result $\forall\phi^{\mathsf{c}}.\ \eta$ can be further generalized, we bring the free variables $\phi_1^{\mathsf{c}} \in Q(\forall\phi^{\mathsf{c}}.\ \eta)$ into scope when elaborating. The concluding output is $T : \forall\{\phi_1^{\mathsf{c}}\}.\forall\{\phi^{\mathsf{c}}\}.\eta$. As an example, consider a kind signature $\forall a.b \to \star$. We have $\phi = b, \phi^{\mathsf{c}} = b : \star$, and $\phi_1^{\mathsf{c}} = c : \star$, and the final kind is $\forall\{c : \star\}.\forall\{b : \star\}.\forall(a : c).b \to \star$. We see in this one example the three sources of quantified variables, always in this order: variables arising from generalization (*c*), from implicit quantification (*b*), and from explicit quantification (*a*).

Returning to the $\vdash^{\mathsf{pgm}}$ judgment, rule PGM-DT-TTS checks a datatype definition that has a kind signature. It ensures that the signature has already been checked, by fetching the kind information in the context using $(T : \eta) \in \Sigma$. Then it checks the datatype declaration, and gathers the output term context to check the rest of the program. Rule PGM-DT-TT, as in Haskell98, guesses kinds ω_i for each datatype T_i and puts $T_i : \omega_i$ in the context *before* looking at the declarations. The major difference from Haskell98 is that kinds can be generalized *after* the group is checked. We use ϕ_i^{c} to denote the free variables in each kind ω_i. After the recursive group is typed, we generalize the kind of each type constructor as well as the type of its data constructors. To generalize the type of data constructors, we use the $\vdash^{\mathsf{gen}}$ judgment. Rule GEN generalizes every data constructor in the context, where ϕ^{c} are free type variables of its corresponding type constructor, and ϕ_i^{c} are free type variables specific to the data constructor. Returning to rule PGM-DT-TT, note that since the kinds of type constructors are generalized, the occurrences of the type constructors now require more type

arguments. Therefore in Ψ'_i, we substitute T_i with $T_i\ @\phi^c_i$, where T_i is applied to all the variables bound in ϕ^c_i.

The judgment of checking datatype declarations $\Sigma \vdash^{dt} \mathcal{T} \rightsquigarrow \Psi$ has only rule DT-TT, which expands on the rule in Haskell98, to support top-level polymorphism for the kind of T.

Rule DC-TT supports existential variables ϕ. Moreover, the elaborated type μ of $\forall\phi.\overline{\tau_i}^{\,i} \to \rho$ can be further generalized over ϕ^c. Note that ϕ^c (via a small abuse of notation in the rule) excludes free variables in τ_i and Σ.

6.3 Checking Kinds

The kinding judgment $\vdash^k$ appears in Figure 5. For space reasons, we present only selected rules. Kinding $\Sigma \vdash^k \sigma : \eta \rightsquigarrow \mu$ infers the type σ to have kind η, and it elaborates σ to μ. The kinding rules are built upon the axiom $\Sigma \vdash^k \star : \star \rightsquigarrow \star$ (rule KTT-STAR). While this axiom is known to violate logical consistency, as Haskell is already logically inconsistent because of its general recursion, we do not consider it as an issue here. Rule KTT-APP concerns applications $\tau_1\ \tau_2$. It first infers the kind of τ_1 to be η_1. The kind η_1 can be a polymorphic kind headed by a $\forall$, though it is expected to be a function kind. Thus the rule uses $\vdash^{inst}$ to instantiate η_1 to $\omega_1 \to \omega_2$. The instantiation judgment $\Sigma \vdash^{inst} \mu_1 : \eta \sqsubseteq \omega \rightsquigarrow \mu_2$ instantiates a kind η to a monokind ω, where if μ_1 has kind η then μ_2 has kind ω. After instantiation, rule KTT-APP checks ($\vdash^{kc}$) the argument τ_2 against the expected argument kind ω_1. The kind checking judgment $\vdash^{kc}$ simply delegates the work to kinding and instantiation. Rule KTT-KAPP checks visible kind applications. Note in the return kind η, the variable a is substituted by the elaborated argument ρ_2. Rule KTT-FORALLI elaborates an unannotated type $\forall a.\sigma$ to $\forall a : \omega.\mu$, where ω is an *elaborated* kind ($\vdash^{ela}$) guessed for a.

The stand-alone elaborated kinding judgment $\vdash^{ela}$ type-checks elaborated types. As all necessary instantiation has been done, type-checking for elaborated types is easy. For example, rule ELA-APP concerns applications $\rho_1\ \rho_2$. Compared to rule KTT-APP, here ρ_1 has an arrow kind, and takes exactly the kind of ρ_2. All judgments output well-formed elaborated types, as the following lemma states:

Lemma 6.2 (Type Elaboration). *We have: (1) if* $\Sigma \vdash^k \sigma : \eta \rightsquigarrow \mu$, *then* $\Sigma \vdash^{ela} \mu : \eta$; *(2) if* $\Sigma \vdash^{kc} \sigma \Leftarrow \eta \rightsquigarrow \mu$, *then* $\Sigma \vdash^{ela} \mu : \eta$; *(3) if* $\Sigma \vdash^{ela} \mu_1 : \eta$, *and* $\Sigma \vdash^{inst} \mu_1 : \eta \sqsubseteq \omega \rightsquigarrow \mu_2$, *then* $\Sigma \vdash^{ela} \mu_2 : \omega$.

7 KIND INFERENCE FOR POLYKINDS

We now describe the *algorithmic* counterpart of the PolyKinds system. Figure 6 presents the syntax of kinds and contexts in the algorithmic system for PolyKinds. Elaborated monotypes are extended with unification variables $\widehat{\alpha}$. Echoing the algorithm for Haskell98, type contexts are extended with unification variables, which now have kinds ($\widehat{\alpha} : \omega$ and $\widehat{\alpha} : \omega = \rho$). Also added to contexts are local scopes $\{\Delta\}$. These are special type contexts, where *variables can be reordered*. Recall the kind $\forall(a :: (f\ b))\ (c :: k).f\ c \to \star$ in Section 2, where f and b appear before k, but end up depending on k. In which order should we put f, b and k in the algorithmic context to kind-check the signature? We cannot have a correct order before completing inference. Therefore, we put them into a local scope, knowing we can reorder the variables during kind-checking according to the dependency information. The well-formedness judgment for local scopes requires them to be well-scoped, leading to the fact that $\Delta, \{\Delta'\}$ is well-formed iff Δ, Δ' is. The marker $\blacktriangleright_D$, subscripted by the name of a data constructor, is used only in and explained with rule A-DC-TT.

7.1 Algorithmic Program Typing

The algorithmic typing rules appear in Figure 6. The judgment $\Omega; \Gamma \Vdash^{pgm} pgm : \mu$ checks the program. The rule A-PGM-SIG and rule A-PGM-DT-TTS correspond directly to the declarative rules. Note that as the datatype declaration in rule A-PGM-DT-TTS already has a signature, the output type

elaborated monotype	ρ, ω	::=	$\star \mid \mathsf{Int} \mid a \mid T \mid \rho_1\ \rho_2 \mid \rho_1\ @\rho_2 \mid \rightarrow \mid \widehat{\alpha}$
term context	Γ	::=	$\bullet \mid \Gamma, D : \mu$
type context	Δ, Θ	::=	$\bullet \mid \Delta, a : \omega \mid \Delta, T : \eta$
		$\mid$	$\Delta, \widehat{\alpha} : \omega \mid \Delta, \widehat{\alpha} : \omega = \rho \mid \Delta, \{\Delta'\} \mid \Delta, \blacktriangleright_D$
complete type context	Ω	::=	$\bullet \mid \Omega, a : \omega \mid \Omega, T : \eta \mid \Omega, \widehat{\alpha} : \omega = \rho \mid \Omega, \{\Omega'\} \mid \Omega, \blacktriangleright_D$
kind binder list	$\widehat{\phi}^{\mathsf{c}}$	::=	$\bullet \mid \widehat{\phi}^{\mathsf{c}}, \widehat{\alpha} : \kappa$

$\boxed{\Omega; \Gamma \Vdash^{\mathsf{pgm}} pgm : \mu}$ *(Typing Program)*

A-PGM-SIG
$$\frac{\Omega \Vdash^{\mathsf{sig}} \mathcal{S} \rightsquigarrow T : \eta \qquad \Omega, T : \eta; \Gamma \Vdash^{\mathsf{pgm}} pgm : \mu}{\Omega; \Gamma \Vdash^{\mathsf{pgm}} \mathbf{sig}\, \mathcal{S}; pgm : \mu}$$

A-PGM-DT-TTS
$$\frac{(T : \eta) \in \Omega \qquad \Omega \Vdash^{\mathsf{dt}} \mathcal{T} \rightsquigarrow \Gamma_1 \dashv \Omega \qquad \Omega; \Gamma, \Gamma_1 \Vdash^{\mathsf{pgm}} pgm : \mu}{\Omega; \Gamma \Vdash^{\mathsf{pgm}} \mathbf{rec}\, \mathcal{T}; pgm : \mu}$$

A-PGM-DT-TT
$$\frac{\Theta_1 = \Omega, \overline{\widehat{\alpha}_i : \star}^{\,i}, \overline{T_i : \widehat{\alpha}_i}^{\,i} \qquad \overline{\Theta_i \Vdash^{\mathsf{dt}} \mathcal{T}_i \rightsquigarrow \Gamma_i \dashv \Theta_{i+1}}^{\,i} \qquad \overline{\widehat{\phi}_i^{\mathsf{c}} = \mathsf{unsolved}([\Theta_{n+1}]\widehat{\alpha}_i)}^{\,i} \qquad \overline{\Theta_{n+1} \Vdash^{\mathsf{gen}}_{\phi_i^{\mathsf{c}}} ([\Theta_{n+1}](\Gamma_i[\overline{\widehat{\phi}_i^{\mathsf{c}}}^{\,i} \mapsto \phi_i^{\mathsf{c}}])) \rightsquigarrow \Gamma_i'}^{\,i} \qquad \overline{\Omega, T_i : \forall\{\phi_i^{\mathsf{c}}\}.(([\Theta_{n+1}]\widehat{\alpha}_i)[\overline{\widehat{\phi}_i^{\mathsf{c}}}^{\,i} \mapsto \phi_i^{\mathsf{c}}]) ; \Gamma, \Gamma_i'[\overline{T_i \mapsto T_i\ @\phi_i^{\mathsf{c}}}^{\,i}] \Vdash^{\mathsf{pgm}} pgm : \mu}^{\,i}}{\Omega; \Gamma \Vdash^{\mathsf{pgm}} \mathbf{rec}\, \overline{\mathcal{T}_i}^{\,i \in 1..n}; pgm : \mu}$$

$\boxed{\Omega \Vdash^{\mathsf{sig}} \mathcal{S} \rightsquigarrow T : \eta}$ *(Typing Signature)*

A-SIG-TT
$$\frac{\lceil \sigma \rceil \qquad \overline{a_i}^{\,i} = \mathsf{fkv}(\sigma) \qquad \Omega, \{\overline{\widehat{\alpha}_i : \star, a_i : \widehat{\alpha}_i}^{\,i}\} \Vdash^{\mathsf{k}} \sigma : \star \rightsquigarrow \eta \dashv \Delta \qquad \phi_1^{\mathsf{c}} = \mathsf{scoped\_sort}(\overline{a_i : [\Delta]\widehat{\alpha}_i}^{\,i}) \qquad \widehat{\phi}_2^{\mathsf{c}} = \mathsf{unsolved}(\Delta) \qquad \Delta \hookrightarrow \overline{a_i}^{\,i}}{\Omega \Vdash^{\mathsf{sig}} \mathbf{data}\, T : \sigma \rightsquigarrow T : \forall\{\phi_2^{\mathsf{c}}\}.((\forall\{\phi_1^{\mathsf{c}}\}.[\Delta]\eta)[\widehat{\phi}_2^{\mathsf{c}} \mapsto \phi_2^{\mathsf{c}}])}$$

$\boxed{\Delta \Vdash^{\mathsf{dt}} \mathcal{T} \rightsquigarrow \Gamma \dashv \Theta}$ *(Typing Datatype Decl.)*

A-DT-TT
$$\frac{(T : \forall\{\phi_1^{\mathsf{c}}\}.\forall\phi_2^{\mathsf{c}}.\ \omega) \in \Delta \qquad \Delta, \phi_1^{\mathsf{c}}, \phi_2^{\mathsf{c}}, \overline{\widehat{\alpha}_i : \star}^{\,i} \Vdash^{\mathsf{u}} [\Delta]\omega \approx (\overline{\widehat{\alpha}_i}^{\,i} \rightarrow \star) \dashv \Theta_1, \phi_1^{\mathsf{c}}, \phi_2^{\mathsf{c}}, \overline{\widehat{\alpha}_i : \star = \omega_i}^{\,i} \qquad \overline{\Theta_j, \phi_1^{\mathsf{c}}, \phi_2^{\mathsf{c}}, \overline{a_i : \omega_i}^{\,i} \Vdash^{\mathsf{dc}}_{(T\ @\phi_1^{\mathsf{c}}\ @\phi_2^{\mathsf{c}}\ \overline{a_i}^{\,i})} \mathcal{D}_j \rightsquigarrow \mu_j \dashv \Theta_{j+1}, \phi_1^{\mathsf{c}}, \phi_2^{\mathsf{c}}, \overline{a_i : \omega_i}^{\,i}}^{\,j}}{\Delta \Vdash^{\mathsf{dt}} \mathbf{data}\, T\, \overline{a_i}^{\,i} = \overline{\mathcal{D}_j}^{\,j \in 1..n} \rightsquigarrow \overline{D_j : \forall\{\phi_1^{\mathsf{c}}\}.\forall\phi_2^{\mathsf{c}}.\ \forall\overline{a_i : \omega_i}^{\,i}.\mu_j}^{\,j} \dashv \Theta_{n+1}}$$

$\boxed{\Delta \Vdash^{\mathsf{dc}}_{\rho} \mathcal{D} \rightsquigarrow \mu \dashv \Theta}$ *(Typing Data Constructor Decl.)*

A-DC-TT
$$\frac{\Delta, \blacktriangleright_D \Vdash^{\mathsf{k}} \forall\phi.(\overline{\tau_i}^{\,i} \rightarrow \rho) : \star \rightsquigarrow \mu \dashv \Theta_1, \blacktriangleright_D, \Theta_2 \qquad \widehat{\phi}^{\mathsf{c}} = \mathsf{unsolved}(\Theta_2)}{\Delta \Vdash^{\mathsf{dc}}_{\rho} \forall\phi.D\, \overline{\tau_i}^{\,i} \rightsquigarrow \forall\{\phi^{\mathsf{c}}\}.(([\Theta_2]\mu)[\widehat{\phi}^{\mathsf{c}} \mapsto \phi^{\mathsf{c}}]) \dashv \Theta_1}$$

Fig. 6. Algorithmic program typing in PolyKinds

context remains unchanged. Rule A-PGM-DT-TT concerns a group (without kind signatures). Like in Haskell98, it first assigns a fresh unification variable $\widehat{\alpha}_i : \star$ as the kind of each type constructor, and then type-checks each datatype declaration, yielding the output context Θ_{n+1}. Unlike Haskell98 which then uses defaulting, here from each $\widehat{\alpha}_i$ we get their unsolved unification variables $\widehat{\phi}_i^{\mathsf{c}}$ and generalize the kind of each type constructor as well as the type of each data constructor. The unsolved(Δ) metafunction simply extracts a set of free unification variables in Δ, with their kinds substituted by Δ. Before generalization, we apply Θ_{n+1} to the results so all solved unification

variables get substituted away. We use the notation $\widehat{\phi_i^c} \mapsto \phi_i^c$ to mean that all unification variables in $\widehat{\phi_i^c}$ are replaced by fresh type variables in ϕ_i^c. The algorithmic generalization judgment $\Vdash^{\text{gen}}$ corresponds straightforwardly to the declarative rule, and thus is included only in the technical supplement. Though they appear daunting, the extended contexts used in the last premise to this rule are unsurprising: they just apply the relevant substitutions (the solved unification variables in Θ_{n+1}, the replacement of unification variables with fresh proper type variables $\widehat{\phi_i^c} \mapsto \phi_i^c$, and the generalization of the kinds of the group of datatypes $T_i \mapsto T_i\ @\phi_i^c$).

The judgment $\Omega \Vdash^{\text{sig}} \mathcal{S} \rightsquigarrow T : \eta$ type-checks a signature definition. We get all free variables in σ using $\text{fkv}(\sigma)$ and assign each variable a_i a kind $\widehat{\alpha_i} : \star$. Those variables are put into a local scope to kind-check σ. Then, we use scoped_sort—a standard topological sort—to return an ordering of the variables that respects dependencies. Finally, we substitute away solved unification variables in the result kind μ and generalize over the unsolved variables $\widehat{\phi_2^c}$ in Δ. As $\widehat{\phi_2^c}$ is generalized outside ϕ_1^c, we use the *quantification check* $\Delta \hookrightarrow \overline{a_i}^{\,i}$ (Section 7.2) to ensure the result kind is well-ordered.

Rule A-DT-TT is a straightforward generalization of rule A-DT-DECL to polymorphic kinds. Here T can have a polymorphic kind from kind signatures.

Rule A-DC-TT checks a data constructor declaration. It first puts a marker into the context before kinding. After kinding, it substitutes away all the solved unification variables to the right of the marker, and generalizes over all unsolved unification variables to the right of the marker. The fact that the context is ordered gives us precise control over variables that need generalization.

7.2 The Quantification Check

Ill-ordered kinds are rejected. Consider the following example:

data *Proxy* :: $\forall k.\ k \to \star$
data *Relate* :: $\forall a\ (b :: a).\ a \to \mathit{Proxy}\ b \to \star$
data T :: $\forall (a :: \star)\ (b :: a)\ (c :: a)\ d.\ \mathit{Relate}\ b\ d \to \star$

Proxy just gives us a way to write a type whose kind is not $\star$. The *Relate* $\tau_1\ \tau_2$ type forces the kind of τ_2 to depend on that of τ_1, giving rise to the unusual dependency in T. The definition of T then introduces a, b, c and d. The kinds of a, b and c are known, but the kind of d must be inferred; call it $\widehat{\alpha}$. We discover that $\widehat{\alpha} = \mathit{Proxy}\ \widehat{\beta}$, where $\widehat{\beta} :: a$. There are no further constraints on $\widehat{\beta}$. Naïvely, we would generalize over $\widehat{\beta}$, but that would be disastrous, as a is locally bound. Instead, we must reject this definition, as our declarative specification always puts inferred variables (such as the type variable $\widehat{\beta}$ would become if generalized) before other ones.

The quantification-checking metafunction $\Delta \hookrightarrow \phi$, defined as $\text{fkv}(\text{unsolved}(\Delta)) \sharp \phi$, ensures that free variables in $\text{unsolved}(\Delta)$ are disjoint ($\sharp$) with ϕ, so that we can safely generalize $\text{unsolved}(\Delta)$ outside ϕ.[4]

7.3 Kinding

Figure 7 presents the selected rules for kinding judgment $\Vdash^{\text{k}}$, along with the auxiliary judgments. Most rules correspond directly to their declarative counterparts. For applications $\tau_1\ \tau_2$, rule A-KTT-APP first synthesizes the kind of τ_1 to be η_1, then uses $\Vdash^{\text{kapp}}$ to type-check τ_2. The judgment $\Delta \Vdash^{\text{kapp}} (\rho_1 : \eta) \bullet \tau : \omega \rightsquigarrow \rho_2 \dashv \Theta$ is interpreted as, under context Δ, applying the type ρ_1 of kind η to the type τ returns kind ω, the elaboration result ρ_2, and an output context Θ. When η_1 is polymorphic (rule A-KAPP-TT-FORALL), we instantiate it with a fresh unification variable. Rule A-KTT-FORALLI checks a polymorphic type. We assign a unification variable as the kind of a, bring $\widehat{\alpha} : \star$, $a : \widehat{\alpha}$ into scope to check the body against $\star$, yielding the output context $\Delta_2, a : \widehat{\alpha}, \Delta_3$.

[4] See also the alternative design in the technical supplement.

$\boxed{\Delta \Vdash^{\mathsf{inst}} \mu_1 : \eta \sqsubseteq \omega \rightsquigarrow \mu_2 \dashv \Theta}$ *(Instantiation)*

$$\textsc{a-inst-refl}\ \frac{\Delta \Vdash^{\mu} \omega_1 \approx \omega_2 \dashv \Theta}{\Delta \Vdash^{\mathsf{inst}} \mu : \omega_1 \sqsubseteq \omega_2 \rightsquigarrow \mu \dashv \Theta} \qquad \textsc{a-inst-forall}\ \frac{\Delta, \widehat{\alpha} : \omega_1 \Vdash^{\mathsf{inst}} \mu_1\ @\widehat{\alpha} : \eta[a \mapsto \widehat{\alpha}] \sqsubseteq \omega_2 \rightsquigarrow \mu_2 \dashv \Theta}{\Delta \Vdash^{\mathsf{inst}} \mu_1 : \forall a : \omega_1 . \eta \sqsubseteq \omega_2 \rightsquigarrow \mu_2 \dashv \Theta}$$

$\boxed{\Delta \Vdash^{\mathsf{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu \dashv \Theta}$ *(Kind Checking)*

$$\textsc{a-kc-sub}\ \frac{\Delta \Vdash^{\mathsf{k}} \sigma : \eta \rightsquigarrow \mu_1 \dashv \Delta_1 \qquad \Delta_1 \Vdash^{\mathsf{inst}} \mu_1 : [\Delta_1]\eta \sqsubseteq [\Delta_1]\omega \rightsquigarrow \mu_2 \dashv \Delta_2}{\Delta \Vdash^{\mathsf{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu_2 \dashv \Delta_2}$$

$\boxed{\Delta \Vdash^{\mathsf{k}} \sigma : \eta \rightsquigarrow \mu \dashv \Theta}$ *(Kinding)*

$$\textsc{a-ktt-star}\ \frac{}{\Delta \Vdash^{\mathsf{k}} \star : \star \rightsquigarrow \star \dashv \Delta} \qquad \textsc{a-ktt-app}\ \frac{\Delta \Vdash^{\mathsf{k}} \tau_1 : \eta_1 \rightsquigarrow \rho_1 \dashv \Delta_1 \qquad \Delta_1 \Vdash^{\mathsf{kapp}} (\rho_1 : [\Delta_1]\eta_1) \bullet \tau_2 : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\mathsf{k}} \tau_1\ \tau_2 : \omega \rightsquigarrow \rho \dashv \Theta}$$

$$\textsc{a-ktt-foralli}\ \frac{\Delta, \widehat{\alpha} : \star, a : \widehat{\alpha} \Vdash^{\mathsf{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu \dashv \Delta_2, a : \widehat{\alpha}, \Delta_3 \qquad \Delta_3 \hookrightarrow a}{\Delta \Vdash^{\mathsf{k}} \forall a.\sigma : \star \rightsquigarrow \forall a : \widehat{\alpha}.[\Delta_3]\mu \dashv \Delta_2, \mathsf{unsolved}(\Delta_3)}$$

$\boxed{\Delta \Vdash^{\mathsf{kapp}} (\rho_1 : \eta) \bullet \tau : \omega \rightsquigarrow \rho_2 \dashv \Theta}$ *(Application Kinding)*

$$\textsc{a-kapp-tt-arrow}\ \frac{\Delta \Vdash^{\mathsf{kc}} \tau \Leftarrow \omega_1 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta \Vdash^{\mathsf{kapp}} (\rho_1 : \omega_1 \ \rightarrow\ \omega_2) \bullet \tau : \omega_2 \rightsquigarrow \rho_1\ \rho_2 \dashv \Theta} \qquad \textsc{a-kapp-tt-forall}\ \frac{\Delta, \widehat{\alpha} : \omega_1 \Vdash^{\mathsf{kapp}} (\rho_1\ @\widehat{\alpha} : \eta[a \mapsto \widehat{\alpha}]) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\mathsf{kapp}} (\rho_1 : \forall a : \omega_1 . \eta) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}$$

$$\textsc{a-kapp-tt-kuvar}\ \frac{\Delta_1, \widehat{\alpha}_1 : \star, \widehat{\alpha}_2 : \star, \widehat{\alpha} : \omega = (\widehat{\alpha}_1\ \rightarrow\ \widehat{\alpha}_2), \Delta_2 \Vdash^{\mathsf{kc}} \tau \Leftarrow \widehat{\alpha}_1 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta_1, \widehat{\alpha} : \omega, \Delta_2 \Vdash^{\mathsf{kapp}} (\rho_1 : \widehat{\alpha}) \bullet \tau : \widehat{\alpha}_2 \rightsquigarrow \rho_1\ \rho_2 \dashv \Theta}$$

$\boxed{\Delta \Vdash^{\mathsf{ela}} \mu : \eta}$ *(Elaborated Kinding)*

$$\textsc{a-ela-app}\ \frac{\Delta \Vdash^{\mathsf{ela}} \rho_1 : \omega_1 \ \rightarrow\ \omega_2 \qquad \Delta \Vdash^{\mathsf{ela}} \rho_2 : \omega_1}{\Delta \Vdash^{\mathsf{ela}} \rho_1\ \rho_2 : \omega_2} \qquad \textsc{a-ela-kapp}\ \frac{\Delta \Vdash^{\mathsf{ela}} \rho_1 : \forall a : \omega . \eta \qquad \Delta \Vdash^{\mathsf{ela}} \rho_2 : \omega}{\Delta \Vdash^{\mathsf{ela}} \rho_1\ @\rho_2 : \eta[a \mapsto [\Delta]\rho_2]}$$

Fig. 7. Selected rules for algorithmic kinding in PolyKinds

As a goes out of the scope in the conclusion, we need to drop a in the concluding context. To make sure that dropping a outputs a well-formed context, we substitute away all solved unification variables in Δ_3 for the return kind, and keep only $\mathsf{unsolved}(\Delta_3)$, which are ensured ($\Delta_3 \hookrightarrow a$) to have no dependency on a.

In the algorithmic elaborated kinding judgment $\Delta \Vdash^{\mathsf{ela}} \mu : \eta$, we keep the invariant: $[\Delta]\eta = \eta$. That is why in rule A-ELA-APP we substitute a with $[\Delta]\rho_2$.

Instantiation ($\Vdash^{\mathsf{inst}}$) contains the only entry to unification (rule A-INST-REFL).

7.4 Unification

The judgments of unification and promotion are excerpted in Figure 8. Most rules are natural extensions of those in Haskell98. Full rules are in the technical supplement.

Promotion. The promotion judgment $\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta$ is extended with kind annotations for unification variables. As our unification variables have kinds now, rule A-PR-KUVARR-TT must also promote the kind of $\widehat{\beta}$, so that $\widehat{\beta}_1 : \rho_1$ in the context is well-formed. Promotion now has a new

$\boxed{\Delta \Vdash^{\mu} \omega_1 \approx \omega_2 \dashv \Theta}$ *(Unification)*

$$\textsc{a-u-refl-tt}\ \frac{}{\Delta \Vdash^{\mu} \omega \approx \omega \dashv \Delta} \qquad \textsc{a-u-app}\ \frac{\Delta \Vdash^{\mu} \rho_1 \approx \rho_3 \dashv \Delta_1 \qquad \Delta_1 \Vdash^{\mu} [\Delta_1]\rho_2 \approx [\Delta_1]\rho_4 \dashv \Theta}{\Delta \Vdash^{\mu} \rho_1\ \rho_2 \approx \rho_3\ \rho_4 \dashv \Theta}$$

$$\textsc{a-u-kvarL-tt}\ \frac{\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \widehat{\alpha} : \omega_1, \Theta_2 \qquad \Theta_1 \Vdash^{\mathsf{ela}} \rho_2 : \omega_2 \qquad \Theta_1 \Vdash^{\mu} [\Theta_1]\omega_1 \approx \omega_2 \dashv \Theta_3}{\Delta \Vdash^{\mu} \widehat{\alpha} \approx \rho_1 \dashv \Theta_3, \widehat{\alpha} : \omega_1 = \rho_2, \Theta_2}$$

$$\textsc{a-u-kvarL-lo-tt}\ \frac{\begin{array}{c}\Delta_1, \Delta_2 +\!\!+^{\mathsf{mv}} \widehat{\alpha} : \omega_1 \rightsquigarrow \Theta \qquad \Delta[\{\Theta\}] \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \{\Theta_2, \widehat{\alpha} : \omega_1, \Theta_3\}, \Theta_4 \\ \Theta_1, \{\Theta_2\} \Vdash^{\mathsf{ela}} \rho_2 : \omega_2 \qquad \Theta_1, \{\Theta_2\} \Vdash^{\mu} [\Theta_1, \Theta_2]\omega_1 \approx \omega_2 \dashv \Theta_5, \{\Theta_6\}\end{array}}{\Delta[\{\Delta_1, \widehat{\alpha} : \omega_1, \Delta_2\}] \Vdash^{\mu} \widehat{\alpha} \approx \rho_1 \dashv \Theta_5, \{\Theta_6, \widehat{\alpha} : \omega_1 = \rho_2, \Theta_3\}, \Theta_4}$$

$\boxed{\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta}$ *(Promotion)*

$$\textsc{a-pr-tvar}\ \frac{}{\Delta[a][\widehat{\alpha}] \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} a \rightsquigarrow a \dashv \Delta[a][\widehat{\alpha}]} \qquad \textsc{a-pr-kuvarR-tt}\ \frac{\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} [\Delta]\rho \rightsquigarrow \rho_1 \dashv \Theta[\widehat{\alpha}][\widehat{\beta} : \rho]}{\Delta[\widehat{\alpha}][\widehat{\beta} : \rho] \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\beta} \rightsquigarrow \widehat{\beta}_1 \dashv \Theta[\widehat{\beta}_1 : \rho_1, \widehat{\alpha}][\widehat{\beta} : \rho = \widehat{\beta}_1]}$$

$\boxed{\Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2 \rightsquigarrow \Theta}$ *(Moving)*

$$\textsc{a-mv-empty}\ \frac{}{\bullet +\!\!+^{\mathsf{mv}} \Delta \rightsquigarrow \Delta} \qquad \textsc{a-mv-kuvar}\ \frac{\mathsf{vars}(\omega) \sharp \mathsf{dom}(\Delta_2) \qquad \Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2 \rightsquigarrow \Theta}{\widehat{\alpha} : \omega, \Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2 \rightsquigarrow \widehat{\alpha} : \omega, \Theta} \qquad \textsc{a-mv-kuvarM}\ \frac{\neg(\mathsf{vars}(\omega) \sharp \mathsf{dom}(\Delta_2)) \qquad \Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2, \widehat{\alpha} : \omega \rightsquigarrow \Theta}{\widehat{\alpha} : \omega, \Delta_1 +\!\!+^{\mathsf{mv}} \Delta \rightsquigarrow \Theta}$$

Fig. 8. Selected rules for unification, promotion, and moving in PolyKinds

failure mode: it cannot move proper quantified type variables. In rule A-PR-TVAR, the variable a must be to the left of $\widehat{\alpha}$.

Unfortunately, now we cannot easily tell whether promoting is terminating. In particular, the convergence of promotion in Haskell98 is built upon the obvious fact that the size of the kind being promoted always gets smaller from the conclusion to the hypothesis. However, rule A-PR-KUVARR-TT breaks this invariant, as the judgment recurs into the kinds of unification variables, and the size of the kinds may be larger than the unification variables. As shown in Section 7.5, we prove that promotion is terminating.

Unification. The unification judgment $\Delta \Vdash^{\mu} \omega_1 \approx \omega_2 \dashv \Theta$ for PolyKinds features *heterogeneous constraints*. Recall the definition of *X* and *Y* discussed in Section 2.2. When unifying $\widehat{\alpha}\,\widehat{\beta}$ with *Maybe Bool*, setting $\widehat{\alpha} =$*Maybe* and $\widehat{\beta} =$*Bool* results in ill-kinded results. This suggests that when solving a unification variable, we need to first unify the kinds of both sides, as shown in rule A-U-KVARL-TT. When unifying $\widehat{\alpha}$ with ρ_1, we first promote ρ_1, yielding ρ_2. Now ρ_2 must be well-formed under Θ_1, so we can get its kind ω_1. We then unify the kinds of both sides. If everything succeeds, we set $\widehat{\alpha} : \omega_1 = \rho_2$. Under this rule, the unification $\widehat{\alpha}\,\widehat{\beta} \approx$*Maybe Bool* would be rejected correctly.

Rule A-U-KVARL-LO-TT is essentially the same as rule A-U-KVARL-TT, but deals with unification variables in a local scope. We thus need an extra step to *move* $\widehat{\alpha}$ towards the end of the local scope.

Local scopes and moving. As we have mentioned, a local scope can be reordered as long as the context is well-formed. Consider unifying $\{\widehat{\alpha} : \star, a : \star, b : \widehat{\alpha}, c : \star\} \Vdash^{\mu} \widehat{\alpha} \approx a$. We see that a is not well-formed under the context before $\widehat{\alpha}$, and thus we cannot rewrite $\widehat{\alpha} : \star$ with $\widehat{\alpha} = a : \star$. However, we *can* reorder the context to put $\widehat{\alpha}$ to the right of a. In fact, to maximize the prefix context of $\widehat{\alpha}$, we can move $\widehat{\alpha}$ to the end of the context, yielding $\{a : \star, c : \star, \widehat{\alpha} : \star, b : \widehat{\alpha}\}$. As b depends on $\widehat{\alpha}$, b is also moved to the end of the context. The final context is now $\{a : \star, c : \star, \widehat{\alpha} : \star = a, b : \widehat{\alpha}\}$.

The *moving* judgment $\Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2 \rightsquigarrow \Theta$ reorders the context, by appending Δ_2 to the end of Δ_1, yielding Θ. As we have emphasized, reordering must preserve a well-formed context. Therefore, every term that depends on Δ_2 (rule A-MV-KUVARM) needs to be placed at the end, along with Δ_2.

In rule A-U-KVARL-LO-TT, we begin by reordering the local scope to put $\widehat{\alpha}$ as far to the right as possible. The rest of the rule is essentially the same as rule A-U-KVARL-TT: the added complication stems from the need to keep track of what bindings in the context are a part of the current local scope.

7.5 Termination

Now the challenge is to prove that our unification algorithm terminates, which relies on the convergence of the promotion algorithm. Next, we first discuss the termination of unification, and then move to the more complicated proof for promotion. Let $\langle\Delta\rangle$ denote the number of unsolved unification variables in Δ.

Lemma 7.1 (Promotion Preserves $\langle\Delta\rangle$)**.** *If* $\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta$*, then* $\langle\Delta\rangle = \langle\Theta\rangle$.

Lemma 7.2 (Unification Makes Progress)**.** *If* $\Delta \Vdash^{\mu} \omega_1 \approx \omega_2 \dashv \Theta$*, then either* $\Theta = \Delta$*, or* $\langle\Theta\rangle < \langle\Delta\rangle$.

Now we measure unification $\Delta \Vdash^{\mu} \omega_1 \approx \omega_2 \dashv \Theta$ using the lexicographic order of the pair $(\langle\Delta\rangle, |\omega_1|)$, where $|\omega_1|$ computes the standard size of ω_1. We prove the pair always gets smaller from the conclusion to the hypothesis. Formally, assuming promotion terminates, we have

Theorem 7.3 (Unification Terminates)**.** *Given a context* Δ ok*, and kinds* ρ_1 *and* ρ_2*, where* $[\Delta]\rho_1 = \rho_1$*, and* $[\Delta]\rho_2 = \rho_2$*, it is decidable whether there exists* Θ *such that* $\Delta \Vdash^{\mu} \rho_1 \approx \rho_2 \dashv \Theta$.

We are not yet done, since Theorem 7.3 depends on the convergence of promotion. As observed in rule A-PR-KUVARR, the size of the type being promoted increases from the conclusion to the hypothesis. Worse, the context never decreases. How do we prove promotion terminates? The crucial observation for rule A-PR-KUVARR is that, when we move from the conclusion to the hypothesis, we also move from a unification variable to its kind. Since the kind is well-formed under the prefix context of the variable, we are somehow moving leftward in the context.

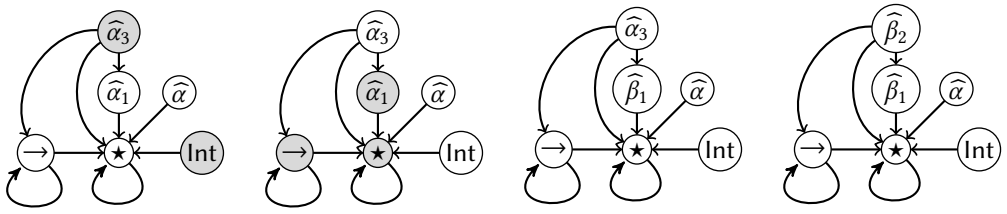
To formalize the observation, we define the *dependency graph* of a context.

Definition 7.4 (Dependency Graph)**.** *The dependency graph of a context* Δ *is a directed graph where:*

(1) Nodes are all type variables and unsolved unification variables of Δ*, and the terminal symbols* $\star$*,* $\rightarrow$ *and* Int.
(2) Edges indicate the dependency from a type to its substituted kind. For example, if $\widehat{\alpha} : \omega$*, then there is a directed edge from* $\widehat{\alpha}$ *to all the nodes appearing in* $[\Delta]\omega$.

As an illustration, consider the context $\Delta = \widehat{\alpha} : \star, \widehat{\alpha}_1 : \star, \widehat{\alpha}_2 : \star = \widehat{\alpha}_1, \widehat{\alpha}_3 : \star \rightarrow \widehat{\alpha}_2$, whose dependency graph is given in Figure 9a (the reader is advised to ignore the color for now). There are several notable properties. First, as long as the context is well-formed, the graph is *acyclic* except for the self-loop of $\star$ and $\rightarrow$. Second, solved unification variables never appear in the graph. The kind of $\widehat{\alpha}_3$ depends on $\widehat{\alpha}_2$, which is already solved by $\widehat{\alpha}_1$, so the dependency goes from $\widehat{\alpha}_3$ to $\widehat{\alpha}_1$.

Now let us consider how promotion works in terms of the dependency graph, by trying to unify $\Delta \Vdash^{\mu} \widehat{\alpha} \approx \widehat{\alpha}_3$ Int. We start by promoting $\widehat{\alpha}_3$ Int. The derivation of the promotion is given at the bottom of Figure 9. We omit some details via $(\cdots)$ as promoting constants ($\star$,$\rightarrow$ and Int) is trivial. At the top of Figure 9 we give the dependency graph at certain points in the derivation, where the part being promoted is highlighted in gray. At the beginning we are at Figure 9a. For $\widehat{\alpha}_3$, by rule A-PR-KUVARR, we first promote the kind of $\widehat{\alpha}_3$, which is (after context application) $\star \rightarrow \widehat{\alpha}_1$ (Figure 9b). As $\star$ and $\rightarrow$ are always well-formed, we then promote $\widehat{\alpha}_1$ whose kind is the



(a) Promote $\widehat{\alpha}_3$ Int (b) Promote $\star \to \widehat{\alpha}_1$ (c) Solve $\widehat{\alpha}_1 = \widehat{\beta}_1$ (d) Solve $\widehat{\alpha}_3 = \widehat{\beta}_2$

$\Theta_1 = \widehat{\beta}_1 : \star, \widehat{\alpha} : \star, \widehat{\alpha}_1 : \star = \widehat{\beta}_1, \widehat{\alpha}_2 : \star = \widehat{\alpha}_1, \widehat{\alpha}_3 : \star \to \widehat{\alpha}_2$

$\Theta_2 = \widehat{\beta}_1 : \star, \widehat{\beta}_2 : \star \to \widehat{\beta}_1, \widehat{\alpha} : \star, \widehat{\alpha}_1 : \star = \widehat{\beta}_1, \widehat{\alpha}_2 : \star = \widehat{\alpha}_1, \widehat{\alpha}_3 : \star \to \widehat{\alpha}_2 = \widehat{\beta}_2$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \star \rightsquigarrow \star \dashv \Delta}\ \text{A-PR-KUVARR}}{\cdots \quad \Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\alpha}_1 \rightsquigarrow \widehat{\beta}_1 \dashv \Theta_1 \quad \boxed{9c}}\ \text{A-PR-APP}}{\boxed{9b} \quad \Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \star \to \widehat{\alpha}_1 \rightsquigarrow \star \to \widehat{\beta}_1 \dashv \Theta_1}\ \text{A-PR-KUVARR}}{\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\alpha}_3 \rightsquigarrow \widehat{\beta}_2 \dashv \Theta_2 \quad \boxed{9d}} \quad \cdots}{\boxed{9a} \quad \Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\alpha}_3\ \mathsf{Int} \rightsquigarrow \widehat{\alpha}_2\ \mathsf{Int} \dashv \Theta_2}\ \text{A-PR-APP}$$

Fig. 9. Example of dependency graph

well-formed $\star$. Now we create a fresh variable $\widehat{\beta}_1 : \star$, and solve $\widehat{\alpha}_1$ with $\widehat{\beta}_1$ (Figure 9c). Note since $\widehat{\alpha}_1$ is solved, the dependency from $\widehat{\alpha}_3$ goes to $\widehat{\beta}_1$. Finally, we create a fresh variable $\widehat{\beta}_2$ with kind $\star \to \widehat{\beta}_1$, and solve $\widehat{\alpha}_3$ with $\widehat{\beta}_2$ (Figure 9d). Going back to unification, we solve $\widehat{\alpha} = \widehat{\beta}_2$ Int.

We have several key observations. First, when we move from Figure 9a to Figure 9b via rule A-PR-KUVARR, we are actually moving from the current node ($\widehat{\alpha}_3$) to its adjacent nodes ($\star$, $\to$, and $\widehat{\alpha}_1$). In other words, we are going down in this graph. Moreover, promotion terminates immediately at type constants, so we never fall into the trap of loop. Further, when we solve variables with fresh ones (Figure 9c and Figure 9d), the shape of the graph never changes.

With all those in mind, we conclude that *the promotion process goes top-down via rule* A-PR-KUVARR *in the dependency graph until it terminates at types that are already well-formed.* Based on this conclusion, we can formally prove that promotion terminates.

Theorem 7.5 (Promotion Terminates)**.** *Given a context* $\Delta[\widehat{\alpha}]$ ok*, and a kind* ρ_1 *with* $[\Delta]\omega_1 = \omega_1$*, it is decidable whether there exists* Θ *such that* $\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta$*.*

7.6 Soundness, Completeness and Principality

We prove our algorithm is sound:

Theorem 7.6 (Soundness of $\Vdash^{\mathsf{pgm}}$)**.** *If* $\Omega; \Gamma \Vdash^{\mathsf{pgm}} pgm : \mu$*, then* $[\Omega]\Omega; [\Omega]\Gamma \vdash^{\mathsf{pgm}} pgm : [\Omega]\mu$*.*

Unfortunately, we lose completeness. Recall the example in Section 7.2. This definition of T is rejected by the algorithmic quantification check as the kind of d cannot be determined. However, the declarative system can guess correctly, e.g., $Proxy\ b$ or $Proxy\ c$. Unfortunately, different choices lead to incomparable kinds for T. Thus we argue such programs must be rejected.

Nevertheless, if the user explicitly writes down $d :: Proxy\ b$ or $d :: Proxy\ c$, then the program will be accepted by the algorithm. Thus, we conjecture that if all local dependencies are annotated by the user, we can regain completeness. This, however, is a bit annoying to users, because it means that we cannot accept definitions like the one below, even though the dependency is clear.

data $Eq :: \forall k.\ k \to k \to \star$
data $P :: \forall k\ (a :: k)\ b.\ Eq\ a\ b \to \star$

We do not consider the incompleteness as a problematic issue in practice, as this scenario is quite contrived and (we expect) will rarely occur "in the wild". See more discussion of this point in Section 9.

Although the algorithm is incomplete, we offer the following guarantee: *if the algorithm accepts a definition, then that definition has a principal kind, and the algorithm infers the principal kind.*

Definition 7.7 (Kind Instantiation)**.** *Under context* Σ*, a kind* $\eta = \forall\{\phi_1\}.\forall\phi_2.\omega_1$*, where* ϕ*'s can be empty, instantiates to* ω*, denoted as* $\Sigma \vdash \eta \sqsubseteq \omega$*, if* $\omega_1[\phi_1 \mapsto \overline{\rho_1}][\phi_2 \mapsto \overline{\rho_2}] = \omega$ *for some* $\overline{\rho_1}$ *and* $\overline{\rho_2}$*.*

The relation is embedded in $\Sigma \vdash^{\text{inst}} \mu_1 : \eta \sqsubseteq \omega \rightsquigarrow \mu_2$ (Figure 5), where we ignore μ_1 and μ_2.

Definition 7.8 (Partial Order of Kinds in PolyKinds)**.** *Under context* Σ*, a kind* η_1 *is more general than* η_2*, denoted as* $\Sigma \vdash \eta_1 \preceq \eta_2$*, if for all* ω *such that* $\Sigma \vdash \eta_2 \sqsubseteq \omega$*, we have* $\Sigma \vdash \eta_1 \sqsubseteq \omega$*.*

To understand the definition, consider that if the program type-checks under $T : \eta_2$, then it must type-check under $T : \eta_1$, as $T : \eta_1$ can be instantiated to all monokinds that $T : \eta_2$ is used at.

Now we lift the definition of $\Vdash^{\text{grp}}$ to be the generalized result of kinds and contexts.

Theorem 7.9 (Principality of $\Vdash^{\text{grp}}$)**.** *If* $\Omega \Vdash^{\text{grp}} \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\eta_i}^{\,i}; \overline{\Gamma_i}^{\,i}$*, then whenever* $[\Omega]\Omega \Vdash^{\text{grp}} \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\eta'_i}^{\,i}; \overline{\Psi_i}^{\,i}$ *holds, we have* $[\Omega]\Omega \vdash [\Omega]\eta_i \preceq \eta'_i$*.*

This result echoes the result in the term-level type inference algorithm for Haskell ([Vytiniotis et al. 2011, Section 6.5]): our algorithm does not infer every kind acceptable by the declarative system, but the kinds it does infer are always the best (principal) ones.

8 LANGUAGE EXTENSIONS

We have seen that the PolyKinds system incorporates many features and enjoys desirable properties. In this section, we discuss how the PolyKinds system can be extended with more related language features. The technical supplement contains a few more, less impactful extensions.

8.1 Higher-Rank Polymorphism

The system can be extended naturally to support higher-rank polymorphism [Dunfield and Krishnaswami 2013; Peyton Jones et al. 2007]. With higher-rank polymorphism, every type can have a polymorphic kind. For example, data constructor declarations become $\forall\phi.D\,\overline{\sigma_i}^{\,i}$ instead of $\forall\phi.D\,\overline{\tau_i}^{\,i}$.

Unfortunately, higher-rank polymorphism breaks principality. Consider:

data $Q1 :: \forall k_1\ k_2.\ k_1 \to \star$ **data** $Q2 :: (\forall(k_1 : \star)\ (k_2 : k_1).\ k_1 \to \star) \to \star$

First, we modify the definition of partial order of kinds (Definition 7.8) to state that one kind is more general than another if it can be instantiated to all *polykinds* that the other kind can be instantiated to. Now consider the kind of $Q1$, which under the algorithm is generalized to $\forall\{k3 : \star\}\ (k_1 : \star)\ (k_2 : k3).\ k_1 \to \star$. In Theorem 7.9, we guarantee that this kind is a principal kind as it can be instantiated to all monokinds that other possible kinds for $Q1$, e.g., $\forall(k_1 :: \star)\ (k_2 :: k_1).\ k_1 \to \star$, can be instantiated to. However, under the new definition, $\forall\{k3 :: \star\}\ (k_1 :: \star)\ (k_2 :: k3).\ k_1 \to \star$ is no longer more general than $\forall(k_1 :: \star)\ (k_2 :: k_1).\ k_1 \to \star$, as there is no way to instantiate the former to the latter. To see why we need to modify the definition at all, consider the rank-2 kind of $Q2$, which expects exactly an argument of kind $\forall(k_1 :: \star)\ (k_2 :: k_1).\ k_1 \to \star$.

We do not consider the absence of principality in the setting of higher-rank polymorphism to be a severe issue in practice, for two reasons: to our knowledge, higher-rank polymorphism for datatypes is not heavily used; and it may be possible to recover principality through the use

of a more generous type-subsumption relation. Currently, GHC (and our model of it) does not support first-class type-level abstraction (i.e., Λ in types) [Jones 1995]. This means that we cannot introduce new variables (also called *skolemization* [Peyton Jones et al. 2007, Section 4.6.2]) in an attempt to equate one type with another. Returning to the example above, we *could* massage $\forall\{k3 :: \star\}\ (k_1 :: \star)\ (k_2 :: k3).\ k_1 \to \star$ to $\forall(k_1 :: \star)\ (k_2 :: k_1).\ k_1 \to \star$ if we could abstract over the k_1 in the target type. Recent advances in type-level programming in Haskell [Kiss et al. 2019] suggest we may be able to add first-class abstraction, meaning that type-subsumption can use both instantiation *and* skolemization. We conjecture that this development would recover principal types.

8.2 Generalized Algebraic Datatypes (GADTs)

The focus of this work has been on uniform datatypes, where every constructor's type matches exactly the datatype head: this fact allows us to easily choose the subscript to the $\vdash^{dc}$ judgment in, e.g., rule DT-TT. Programmers in modern Haskell, however, often use *generalized* algebraic datatypes [Peyton Jones et al. 2006; Xi et al. 2003]. There are two impacts of adding these, both of which we found surprising.

Equality constraints. The power of GADTs arises from how they encode local equality constraints. Any GADT can be rewritten to a uniform datatype with equality constraints [Vytiniotis et al. 2011, Section 4.1]. For example, we can rewrite **data** *G a* **where** *MkG* :: *G Bool* to be **data** *G a* = $(a \sim Bool) \Rightarrow MkG$, where $\sim$ describes an equality constraint. For our purposes of doing kind inference, these equality constraints are uninteresting: the $\sim$ operator simply relates two types of the same kind and can be processed as any polykinded type constructor would be. Modeling constraints to the left of a $\Rightarrow$ similarly would add a little clutter to our rules, but would offer no real challenges.

The unexpected simplicity of adding GADTs to our system arises from a key fact: we do not ever allow *pattern-matching*. A GADT pattern-match brings a local equality assumption into scope, which would influence the unification algorithm. However, as pattern matching does not happen in the context of datatype declarations, we avoid this wrinkle here.

Syntax. The implementation of GADTs in GHC has an unusual syntax: **data** *G a* **where** *MkG* :: $a \to G\ Int$. The surprising aspect of this syntax is that the two *a*s above are *different*: the *a* in the header is unrelated to the *a* in the data constructor. This seemingly inconsequential design choice makes kind inference for GADTs very challenging, as constructors have no way to refer back to the datatype parameters. Given that this aspect of GADTs is a quirk of GHC's design—and is not repeated in other languages that support GADTs—we remark here that it is odd and perhaps should be remedied. For the details, please see the technical supplement.

8.3 Type Families

Type families [Chakravarty et al. 2005] are, effectively, type-level functions. Kind inference of type families thus can be designed much like type inference for ordinary functions. However, as they can have dependency, the complications we describe in this paper would arise here, too. In particular, unification would have to be kind-directed, as we have described. The current syntax for closed type families [Eisenberg et al. 2014] shares the same scoping problem as the syntax for GADTs, so our arguments above apply to closed type families equally.

The challenge with type families is that they indeed do pattern-matching, and thus (in concert with GADTs) can bring local equalities into scope. A full analysis of the ramifications here is beyond the scope of this paper, but we believe the literature on type inference in the presence of local equalities would be helpful. Principal among these is the work of Vytiniotis et al. [2011], but Gundry [2013] and Eisenberg [2016] also approach this problem in the context of dependent types.

9 RELATED WORK

The Glasgow Haskell Compiler. The systems we present here are inspired by the algorithms implemented in GHC. However, our goal in the design of these systems is to produce a sound and (nearly) complete pair of specification and implementation, not simply to faithfully record what is implemented. We have identified ways that the GHC implementation can improve in the future. For example, GHC quantifies over local scopes as *specified* where we believe they should be *inferred*; and the tight connection in our system between unification and promotion may improve upon GHC's approach, which separates the two. The details of the relationship between our work and GHC (including a myriad of ways our design choices differ in small ways from GHC's) appear in the technical supplement.

Unification with dependent types. While full higher-order unification is undecidable [Goldfarb 1981], the *pattern* fragment [Miller 1991] is a well-known decidable fragment. Much literature [Abel and Pientka 2011; Gundry and McBride 2013; Reed 2009] is built upon the pattern fragment.

Unification in a dependently typed language features *heterogeneous constraints*. To prove correctness, Reed [2009] used a weaker invariant on homogeneous equality, *typing modulo*, which states that two sides are well typed up to the equality of the constraint yet to be solved. Gundry and McBride [2013] observed the same problem, and use *twin variables* to explicitly represent the same variable at different types, where twin variables are eliminated once the heterogeneous constraint is solved. In both approaches the well-formedness of a constraint depends on other constraints. Cockx et al. [2016] proposed a proof-relevant unification that keeps track of the dependencies between equations. Different from their approaches, our algorithm unifies the kinds when solving unification variables. This guarantees that our unification always outputs well-formed solutions.

Ziliani and Sozeau [2015] present the higher-order unification algorithm for CIC, the base logic of Coq. They favor syntactic equality by trying first-order unification, as they argue the first-order solution gives the most *natural* solution. However, they omit a correctness proof for their algorithm. Coen [2004] also considers first-order unification, but only the soundness lemma is proved. Different from their systems, our system is based on the novel promotion judgment, and correctness including soundness and termination is proved.

The technique of *suspended substitutions* [Eisenberg 2016; Gundry and McBride 2013] is widely adopted in unification algorithms. Our system provides a design alternative, our *quantification check*. Choosing between suspended substitutions and the quantification check is a user-facing language design decision, as suspended substitutions can accept some more programs. The quantification check means that the kind of a locally quantified variable a must be fully determined in a's scope; it may *not* be influenced by usage sites of the construct that depends on a. Suspended substitutions relax this restriction. We conjecture that suspended substitutions can yield a complete algorithm. However, that mechanism is complex. Moreover, unification based on suspended substitutions is only decidable for the pattern fragment. Our system, in contrast, avoids all the complication introduced by suspended substitutions through its quantification check. Our unification terminates for all inputs, preserving backward compatibility to Hindley-Milner-style inference. Although we reject the definition of T (Section 7.2), we can solve more constraints outside the pattern fragment. We conjecture that those constraints are much more common than definitions like T. Suspended substitutions often come with a *pruning* process [Abel and Pientka 2011], which produces a valid solution before solving a unification variable. Our promotion process has a similar effect.

Homogeneous kind-preserving unification. Jones [1995] proposed a homogeneous kind-preserving unification between two types. Kinds κ are defined only as $\star$ or $\kappa_1 \to \kappa_2$. As the kind system is much simpler, kind-preserving unification $\approx_\kappa$ is simply subscripted by the kind, and working out the kinds is straightforward. Our unification subsumes Jones's algorithm.

Type inference in Haskell. Type inference in Haskell is inspired by Damas and Milner [1982] and Pottier and Rémy [2005], extended with various type features, including higher rank polymorphism [Peyton Jones et al. 2007] and local assumptions [Schrijvers et al. 2009; Simonet and Pottier 2007; Vytiniotis et al. 2011], among others. However, none of these works describe an inference algorithm for datatypes, nor do they formalize type variables of varying kinds or polymorphic recursion.

Dependent Haskell. Our PolyKinds system merges types and kinds, a key feature of *Dependent Haskell* (DH) [Eisenberg 2016; Gundry 2013; Weirich et al. 2013, 2017]. There is ongoing work dedicated to its implementation [Xie and Eisenberg 2018]. The most recent work by Weirich et al. [2019] integrates *roles* [Breitner et al. 2016] with dependent types. Our work is the first presentation of unification for DH, and our system may be useful in designing DH's term-level type inference.

Context extension. Our approach of recording unification variables and their solutions in the contexts is inspired by Gundry et al. [2010] and Dunfield and Krishnaswami [2013]. Gundry and McBride [2013] applied the approach to unification in dependent types, where the context also records constraints; constraints also appear in context in Eisenberg [2016]. Further, we extend the context extension approach with local scopes, supporting groups of order-insensitive variables.

Polymorphic recursion. Mycroft [1984] presented a semi-algorithm for polymorphic recursion. Jim [1996] and Damiani [2003] studied typing rules for recursive definitions based on rank-2 intersection types. Comini et al. [2008] studied recursive definitions in a type system that corresponds to the abstract interpreter in Gori and Levi [2002, 2003]. Our system does not infer polymorphic recursion; instead, we exploit kind annotations to guide the acceptance of polymorphic recursion.

Constraint-solving approaches. Many systems (e.g. [Pottier and Rémy 2005]) adopt a modular presentation of type inference, which consists of a constraint generator and a constraint solver. For simplicity, we have presented an eager unification algorithm instead of using a separate constraint solver. However, we believe changing to a constraint-solving approach should not change any of our main results. Section B.1 of the technical supplement considers this point further.

10 CONCLUSION

We have presented the first known, detailed account of kind inference for datatypes, codifying the inference both from the early days of Haskell and the Haskell of today. For the former, we can prove soundness and completeness using the technique of *kind parameters*. For the latter, we have described a sound algorithm for inferring types even in the presence of dependency, allowing users to infer datatype kinds instead of merely checking them. The algorithm is incomplete in obscure scenarios (Section 7.2), a conscious design decision in order to retain termination.

There are several ways we could extend this work. The most obvious is to include other constructs in our approach. However, even with only datatypes formalized, we see this work as sturdy enough to aim at implementation. A primary motivation for starting this work was shortcomings in GHC's current kind inference algorithm and results, yet we had no principled way of improving it. Having completed this research, we now feel encouraged to attack this practical problem afresh and apply what we have learned. We further believe that our approach to inference will be useful to designers and implementors of other dependently typed languages, as they share many of the same challenges as GHC, both in processing datatypes and in type-checking ordinary expressions.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This work has been sponsored by the Hong Kong Research Grant Council projects number 17210617 and 17209519. This material is based upon work supported by the National Science Foundation under Grant No. 1704041.

REFERENCES

Andreas Abel and Brigitte Pientka. 2011. Higher-order dynamic pattern unification for dependent types and records. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 10–26.

P. B. Andrews. 1971. Resolution in type Theory. *Journal of Symbolic Logic* 36 (1971), 414–432.

Richard S. Bird and Lambert Meertens. 1998. Nested datatypes. In *LNCS 1422: Proceedings of Mathematics of Program Construction*, Johan Jeuring (Ed.). Springer-Verlag, Marstrand, Sweden, 52–67. http://www.cs.ox.ac.uk/people/richard.bird/online/BirdMeertens98Nested.pdf

Joachim Breitner, Richard A Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2016. Safe zero-cost coercions for Haskell. *Journal of Functional Programming* 26 (2016).

L. Cardelli. 1986. *A polymorphic lambda-calculus with Type:Type.* Technical Report 10. SRC.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated type synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM, New York, NY, USA, 241–253. https://doi.org/10.1145/1086365.1086397

Jesper Cockx, Dominique Devriese, and Frank Piessens. 2016. Unifiers as equivalences: proof-relevant unification of dependently typed data. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 270–283. https://doi.org/10.1145/2951913.2951917

Claudio Sacerdoti Coen. 2004. *Mathematical knowledge management and interactive theorem proving.* Ph.D. Dissertation. University of Bologna, 2004. Technical Report UBLCS 2004-5.

Marco Comini, Ferruccio Damiani, and Samuel Vrech. 2008. On polymorphic recursion, type systems, and abstract interpretation. In *International Static Analysis Symposium*. Springer, 144–158.

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

Ferruccio Damiani. 2003. Rank 2 intersection types for local definitions and conditional expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25, 4 (2003), 401–451.

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/2500365.2500582

Richard A Eisenberg. 2016. *Dependent types in haskell: Theory and practice.* Ph.D. Dissertation. University of Pennsylvania.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 671–683. https://doi.org/10.1145/2535838.2535856

Richard A Eisenberg, Stephanie Weirich, and Hamidhasan G Ahmed. 2016. Visible type application. In *European Symposium on Programming*. Springer, 229–254.

Ronald Garcia and Matteo Cimini. 2015. Principal type schemes for gradual programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 303–315. https://doi.org/10.1145/2676726.2676992

Warren D Goldfarb. 1981. The undecidability of the second-order unification problem. *Theoretical Computer Science* 13, 2 (1981), 225–230.

Roberta Gori and Giorgio Levi. 2002. An experiment in type inference and verification by abstract interpretation. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 225–239.

Roberta Gori and Giorgio Levi. 2003. Properties of a type abstract interpreter. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 132–145.

Adam Gundry and Conor McBride. 2013. A tutorial implementation of dynamic pattern unification. *Unpublished draft* (2013).

Adam Gundry, Conor McBride, and James McKinna. 2010. Type inference in context. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*. ACM, 43–54.

Adam Michael Gundry. 2013. *Type inference, Haskell and dependent types.* Ph.D. Dissertation. University of Strathclyde.

Fritz Henglein. 1993. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.* 15, 2 (April 1993), 253–289. https://doi.org/10.1145/169701.169692

J. Roger Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60.

G. Huet. 1973. A unification algorithm for typed lambda calculus. *Theoretical Computer Science* 1, 1 (1973), 27–57.

Trevor Jim. 1996. What are principal typings and what are they good for?. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 42–53.

Mark P Jones. 1995. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of functional programming* 5, 1 (1995), 1–35.

Mark P. Jones. 1999. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop (Haskell '99)*, Erik Meijer (Ed.). Paris, France, pp. 9–22. University of Utrecht Technical Report UU-CS-1999-28.

Csongor Kiss, Susan Eisenbach, Tony Field, and Simon Peyton Jones. 2019. Higher-order type-level programming in Haskell. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming (ICFP 2019)*. ACM.

Didier Le Botlan and Didier Rémy. 2003. MLF: Raising ML to the Power of System F *(ICFP '03)*. 12.

Daan Leijen. 2009. Flexible types: robust type inference for first-class polymorphism. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 66–77. https://doi.org/10.1145/1480881.1480891

Dale Miller. 1991. Unification of simply typed lambda-terms as logic programming. (1991).

Alan Mycroft. 1984. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*. Springer, 217–228.

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

Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming* 17, 1 (2007), 1–82.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. ACM, New York, NY, USA, 50–61. https://doi.org/10.1145/1159803.1159811

François Pottier and Didier Rémy. 2005. The essence of ML type inference. *Advanced Topics in Types and Programming Languages* (2005).

Jason Reed. 2009. Higher-order constraint simplification in dependent type theory. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*. ACM, 49–56.

Didier Rémy and Boris Yakobowski. 2008. From ML to MLF: Graphic type constraints with efficient type inference. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1411204.1411216

Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 341–352. https://doi.org/10.1145/1596550.1596599

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

Vincent Simonet and François Pottier. 2007. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 1 (2007), 1.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn (X) Modular type inference with local assumptions. *Journal of functional programming* 21, 4-5 (2011), 333–412.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2008. FPH: First-class polymorphism for Haskell *(ICFP '08)*. 12.

Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A. Eisenberg. 2019. A Role for dependent types in Haskell. *Proc. ACM Program. Lang.* 3, ICFP, Article 101 (July 2019), 29 pages. https://doi.org/10.1145/3341705

Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind Equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 275–286. https://doi.org/10.1145/2500365.2500599

Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A Eisenberg. 2017. A specification for dependent types in Haskell. In *Proceedings of the 22th ACM SIGPLAN International Conference on Functional Programming (ICFP '17)*. ACM.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, New York, NY, USA, 224–235. https://doi.org/10.1145/604131.604150

Ningnign Xie and Richard A Eisenberg. 2018. Coercion Quantification. In *Haskell Implementors' Workshop*.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*. ACM, New York, NY, USA, 53–66. https://doi.org/10.1145/2103786.2103795

Beta Ziliani and Matthieu Sozeau. 2015. A Unification Algorithm for Coq Featuring Universe Polymorphism and Overloading. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 179–191. https://doi.org/10.1145/2784731.2784751