Partial Type Signatures for Haskell

Thomas Winant[1], Dominique Devriese[1], Frank Piessens[1], and Tom Schrijvers[2]

[1] KU Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium
firstname.lastname@cs.kuleuven.be
[2] Ghent University, Krijgslaan 281 S9 WE02, 9000 Gent, Belgium
tom.schrijvers@ugent.be

Abstract. Strong type systems can be used to increase the reliability and performance of programs. In combination with type inference the overhead for the programmer can be kept small. Nevertheless, explicit type signatures often remain needed or useful. In languages with standard Hindley-Milner-based type systems, programmers have a binary choice between omitting the type signature (and rely on type inference) or explicitly providing the type entirely; there are no intermediate options. Proposals for partial type signatures exist, but none support features like local constraints and GHC's non-generalisation of local bindings. Therefore we propose and motivate a practical form of partial type signatures for present-day Haskell. We formally describe our proposal as an extension of the OutsideIn(X) system and prove some of its properties. We have developed a (not yet complete) implementation for the GHC Haskell compiler. Our design fits naturally in both the OutsideIn(X) formalism and the compiler.

Keywords: Haskell, Hindley-Milner type inference, (partial) type signatures, wildcards

1 Introduction

Static type checking can help catch errors at compile-time and provide useful information for compiler optimisations. Through the use of type inference, programmers are not required to provide explicit type signatures for all values in a program. Nevertheless, explicit signatures can still be needed or useful: type signatures provide a form of machine-checked documentation, they can be used to make general inferred types more specific, and help to verify whether the program corresponds to the programmer's intentions.

Haskell's overloaded math operators exemplify the need for type signatures:

let *harmonic* $x\ y = \frac{2}{\frac{1}{x}+\frac{1}{y}}$ **in** *print* (*harmonic* 3 2)

Under Haskell's defaulting rule[3], x and y are interpreted as floating point numbers leading to the inexact output `2.4000000000000004`. The exact output $\frac{12}{5}$

[3] Haskell lets programmers manually specify to which type the ambiguous type variables satisfying the *Num* class should resolve.

is produced with the signature $harmonic :: Rational \to Rational \to Rational$. Without defaulting, an ambiguous type variable would make a type signature mandatory.

Additionally, type inference is fundamentally limited. It is impossible to infer types for all programs that are typeable in more complex type systems (see e.g. [10, §23.6]). Consider the following Haskell program.

$$foo\ x = (x\ [True, False], x\ [\texttt{'a'}, \texttt{'b'}])$$
$$test = foo\ reverse$$

This program is rejected by Haskell's type checker, because of the Damas-Milner rule that *a lambda-bound argument (like x) must have a monomorphic type.* x could be assigned the type $[Bool] \to [Bool]$, or $[Char] \to [Char]$, but not $\forall a.[a] \to [a]$, see e.g. [9]. With a correct signature, the program is accepted:

$$foo :: (\forall a.[a] \to [a]) \to ([Bool], [Char])$$
$$foo\ x = (x\ [True, False], x\ [\texttt{'a'}, \texttt{'b'}])$$

Haskell, like many other programming languages provides a binary, all-or-nothing choice when it comes to type signatures: either the programmer writes the whole signature or none at all. Nevertheless, in many of the situations where type signatures are needed or useful, it suffices to pin down certain parts of the type. Providing the full type is unneeded and sometimes tedious or distracting. For example, when types are intended to document the code or to make its inferred type more specific, this is often only needed for one argument of a function or for the monad in which a computation runs, but not its result type. For example, only the type of *foo*'s argument cannot be inferred, but its result type can. In cases where we want or need to specify only a part of a type, it can be beneficial to *not* specify the rest. That remainder can be boilerplate, tedious or obscure the intention of the type signature. Not providing this information can save the programmer some thought and work, especially if the uninteresting bits of the type are unknown or prone to frequent change during development.

For such cases, *partial type signatures* can specify a type only partially and leave the rest for the type inferencer to decide. For *foo*, we could use:

$$foo :: (\forall a.[a] \to [a]) \to _$$
$$foo\ x = (x\ [True, False], x\ [\texttt{'a'}, \texttt{'b'}])$$

This partial signature specifies that *foo* is a function and defines the polymorphic type of *foo*'s first argument. The result type is unspecified, as indicated by a type wildcard (written _). Similarly, for the *harmonic* example, it would suffice to write the shorter signature $harmonic :: Rational \to _$.

At this point, we should mention some partial workarounds for the lack of partial signatures in Haskell. *foo* could for example use a pattern type signature:

$$foo\ (x :: \forall a.[a] \to [a]) = ...$$

Expression type signatures similarly provide a partial solution. Another way to simulate partial type signatures uses a helper function, *isTypeFor*, which forces its second argument to have the same type as the first. Combined with an explicitly typed dummy value, we could write for example *foo* as follows:

```
isTypeFor :: a → a → a
_ `isTypeFor` x = x
foo = (⊥ :: (∀a.[a] → [a]) → b) `isTypeFor`
  (λx → (x [True, False], x ['a', 'b']))
```

The type variable b acts as a type wildcard and will be instantiated to the result type. A downside is that *foo*'s implementation is obscured with computationally insignificant code. A combinator library supports this technique [2]. Kiselyov also proposes a trick using *fake clauses* to partially annotate constraints [5]:

```
addOrd :: Ord x ⇒ x → a
addOrd = ⊥
foo2 x | False = addOrd x
foo2 x = x
```

The first clause of *foo2* is never executed but it does make the type inferencer produce an additional constraint $Ord\ x$, leading to the type $Ord\ a \Rightarrow a \to a$ instead of $a \to a$. Not every partial signature can be emulated though: we do not see a way to forbid the inference of additional constraints and there may be other limitations w.r.t. our work as well.

These workarounds are generally poorly legible, cumbersome to use (e.g. requiring lambda functions instead of left-hand-side patterns) and limited (e.g. only a lower bound on type constraints). Their existence does prove the need for actual partial type signatures.

We propose and study a form of partial type signatures in the context of a language with HM-based type inference. Our partial type signatures extend normal signatures with type wildcards (_). During type inference, such wildcards can be instantiated to arbitrary types, e.g. the type $_ \to _$ can be instantiated to $Int \to (Bool \to Int)$ or $(Int \to Bool) \to String$. They map nicely to the unification variables used internally by most type inferencers.

In the context of HM-based type inference, we take care to properly interact with the type generalisation that is performed to achieve let-polymorphism. If (part of) the type instantiating a wildcard is not restricted by type inference, a HM-style type inferencer will quantify over it. Consider the following program.

```
bar :: _ → _
bar _ = True
```

From the return value *True*, the type checker learns that the second wildcard in the partial signature of *bar* must be instantiated to *Bool*. However, the first wildcard remains open. In this case, type generalisation will infer *bar*'s principal type $\forall a.a \to Bool$, as when the type signature is omitted entirely.

A second, related challenge is dealing with constraints, for example type class constraints (e.g. $\forall a.Num\ a \Rightarrow a$) and equality constraints (e.g. $(Fun1\ a \sim (b \to b)) \Rightarrow a \to b$) supported by GHC. Our partial signatures allow the inference of additional constraints if and only if the type contains an *extra-constraints wildcard*, written as an underscore just before the double arrow: $_ \Rightarrow a \to b$. For example, the signature $_ \to b$ (without an extra-constraints wildcard) forbids

types with additional constraints like $Num\ b \Rightarrow Int \to b$. That type can be allowed explicitly with the signature $_ \Rightarrow _ \to b$. Only one extra-constraints wildcard can be present and allows any number of constraints to be added.

In a GHC ticket discussion, Peyton Jones has argued the usefulness of an extra-constraints wildcard based on the following example [13]. By placing a wildcard, the programmer tells the type checker to infer the context for him.

$f :: _ \Rightarrow [a] \to String$ -- Inferred constraints: $(Num\ a, Show\ a)$
$f\ xs = show\ (sum\ xs)$

We also allow multiple references to a wildcard within a signature using *named wildcards* (written as $_a$). They can be used to shorten tedious type signatures:

$isMeltdown :: NukeMonad\ param1\ param2\ Bool$
$unlessMeltdown :: _nm\ () \to _nm\ ()$
$unlessMeltdown\ c = \mathbf{do}\ m \leftarrow isMeltdown$
$\qquad\qquad\qquad\qquad \mathbf{if}\ m\ \mathbf{then}\ return\ ()\ \mathbf{else}\ c$

To make our proposal precise, we give a formal account based on Vytiniotis et al.'s OutsideIn(X) formalism [14]. We define natural and algorithmic typing rules and prove their correspondence. Additionally, we prove that our new rules generalise the old ones for signatures without wildcards and that a partial signature $f :: _ \Rightarrow _$ has the same effect as no signature at all. Such correspondences are important for consistency and to align with users' expectations.

We have an implementation of our proposal in the Glasgow Haskell Compiler, but it is not yet complete at the time of writing. Our current version correctly unifies wildcards and named wildcards with concrete types, but unifying with open types, generalisation, and the extra-constraints wildcard are not yet working as we intend. We hope to finish our modifications in the coming months.

Contributions The idea of partial type signatures is not novel. Several languages support them in some form or other [7,6,11] and they have been proposed for Haskell several times before [15,16]. Dijkstra [4] and Sulzmann and Wazny [12,17] have detailed proposals for Haskell-like languages. Still, we believe that ours is the first rigorous formalisation of partial type signatures for a HM-style inference that supports all the features of present-day Haskell. Specifically, we support local constraints (that arise e.g. from pattern matching on GADTs) and align with GHC's non-generalisation of local bindings. More specifically, our contributions are the following:

- A formalised proposal for partial type signatures, including generalisation, in a Hindley-Milner-style type inference system. Our work plugs into the constraint-based type inference approach OutsideIn(X) [14], currently employed by the de facto standard Haskell compiler GHC.
- We align our partial type signatures with the OutsideIn(X) policy that *let should not be generalised*.
- We formally show that the new typing rules generalise the existing rules for signatures without wildcards and for omitted signatures.
- A (not yet complete) implementation in the GHC Haskell compiler.

Term variables		$\in x, y, z, f, g, h$
Type variables		$\in a, b, c$
Named wildcards		$\in _a,\ _b,\ _c$
Data constructors		$\in K$
	ν	$::= K \mid x$
Programs	$prog$	$::= \epsilon \mid f = e,\ prog \mid$
		$f\ ::\ \underline{\sigma} = e,\ prog$
Expressions	e	$::= \nu \mid \lambda x\,.\,e \mid e_1\ e_2 \mid$
		$\texttt{case}\ e\ \texttt{of}\ \{\overline{K\ \overline{x} \to e}\}$
Type schemes	σ	$::= \forall \overline{a}\,.\,Q \Rightarrow \tau$
Type schemes with wildcards	$\underline{\sigma}$	$::= \forall \overline{a}\,.\,\underline{Q} \Rightarrow \underline{\tau}$
Constraints	Q	$::= \epsilon \mid Q_1 \wedge Q_2 \mid \tau_1 \sim \tau_2 \mid \mathtt{D}\ \overline{\tau} \mid \ldots$
Constraints with wildcards	$\underline{Q}^w$	$::= \underline{Q} \mid \underline{Q}_1^w \wedge \underline{Q}_2^w \mid \underline{\tau}_1 \sim \underline{\tau}_2 \mid \mathtt{D}\ \underline{\overline{\tau}} \mid \ldots$
Constraints with extra constraints wildcard	$\underline{Q}$	$::= \underline{Q}^w \mid \underline{Q}^w \wedge _$
Monotypes	τ, υ	$::= tv \mid \mathtt{Int} \mid \mathtt{Bool} \mid [\tau] \mid \mathtt{T}\ \overline{\tau} \mid \ldots$
Monotypes with wildcards	$\underline{\tau}, \underline{\upsilon}$	$::= \tau \mid _ \mid _a \mid [\underline{\tau}] \mid \mathtt{T}\ \underline{\overline{\tau}}$
Type environments	Γ	$::= \epsilon \mid (\nu{:}\sigma), \Gamma$
Free type variables	$ftv(\cdot)$	
Top-level axiom schemes	$\mathcal{Q}$	$::= \epsilon \mid \mathcal{Q} \wedge \mathcal{Q} \mid \forall \overline{a}\,.\,Q \Rightarrow Q$
Unification variables		$\in \alpha, \beta, \gamma, \omega, \ldots$
Unifiers	θ, φ	$::= [\overline{\alpha \mapsto \tau}]$
Unification or rigid (skolem) variables	tv	$::= \alpha \mid a$
Algorithm-generated constraints	C	$::= Q \mid C_1 \wedge C_2 \mid \exists \overline{\alpha}\,.\,(Q \supset C)$
Free unification variables	$fuv(\cdot)$	
Named wildcards	$nwc(\cdot)$	

Fig. 1. Wildcard syntax extension of [14, Fig. 1, page 12] and [14, Fig. 5, page 17]

Outline In Sect. 2, we describe our additional syntax, both informally and formally. Formal rules for handling wildcard syntax are listed in Sect. 3. We extend OutsideIn(X) typing rules to support wildcards in Sect. 4. Local bindings with partial type signatures are described in Sect. 5. We prove the correspondence of our rules to the standard ones for the uninformative signature $_ \Rightarrow _$ and for signatures without wildcards in Sect. 6. We discuss our implementation in Sect. 7, related work in Sect. 8 and conclude in Sect. 9. Proofs of our results can be found in an extended version of this paper on `http://people.cs.kuleuven.be/dominique.devriese/hpts-ext.pdf`.

Acknowledgments This research is partially funded by the Research Foundation - Flanders (FWO), and by the Research Fund KU Leuven. Dominique Devriese holds a Ph.D. fellowship of the Research Foundation - Flanders (FWO).

2 Wildcard Syntax

In the introduction we already gave an informal account of the wildcard syntax we support. We quickly reiterate and formalise the syntax of wildcards as an

extension of the syntax in OutsideIn(X) [14]. Figure 1 contains the formal definitions with additions and changes highlighted in grey.

First of all, *type wildcards* can take the place of monotypes, e.g. $f :: _ \to _$. For type inference, they are translated to unification variables (see Sect. 3.2). By convention, we write unification variables that arise from wildcards as $\omega_1, \omega_2, \cdots$.

A wildcard *in* a constraint is called a *constraint wildcard*, e.g. $Eq\ _ \Rightarrow a$. A wildcard occurring *as* a constraint is an *extra-constraints wildcard*, e.g. $_ \Rightarrow a$. When it is present, any number of constraints may be added to the type during inference. Because one extra-constraints wildcard can be instantiated to any number of constraints, more than one such wildcard would be pointless. For clarity, we allow only one and require that it comes last in the list of constraints.

Additionally, we support *named wildcards*, e.g. $_a \to _a$. All instances of a named wildcard within a partial type signature must unify with the same type. Named wildcards are particularly useful to express constraints on wildcard types, e.g. $Eq\ _a \Rightarrow _a$ or $(_a \sim b) \Rightarrow _a \to [b]$. Although syntactically similar, named wildcards should not be confused with type variables: they can unify with concrete types. Only when not unified with concrete types, they are generalised over and behave like type variables.

In Fig. 1 we provide variants of type schemes (σ), constraints (Q), and monotypes (τ) that *can* contain wildcards, respectively $\underline{\sigma}$, $\underline{Q}$, and $\underline{\tau}$. A distinction between *constraints with wildcards* ($\underline{Q}^w$) and *constraints with* [*an*] *extra-constraints wildcard* ($\underline{Q}$) is made to enforce that the extra-constraints wildcard can occur at most once and must come last.

3 Wildcard Instantiation and Desugaring

Before we introduce the adapted typing rules, we formalise the relation between wildcards and types. To this end, we define two judgments: the *wildcard instantiation judgment* and the *wildcard desugaring judgment*. They are employed in Sect. 4 by the natural and algorithmic typing rules respectively and the latter should be understood as an algorithmic version of the former.

3.1 Wildcard Instantiation

The wildcard instantiation judgment $\underline{Q}\,;\underline{\tau} \twoheadrightarrow Q\,;\tau$ can be read as "The wildcards in constraints $\underline{Q}$ and monotype $\underline{\tau}$ can be instantiated to obtain constraints Q and monotype τ". Each wildcard in $\underline{Q}$ and $\underline{\tau}$ corresponds to a concrete type or a type variable in Q and τ. Remember that $\underline{Q}$ and $\underline{\tau}$ can contain wildcards, whereas Q and τ cannot. This judgment will be used by the adapted typing rules to instantiate a partial type signature to a type signature without wildcards.

The rules of the judgment are shown in Fig. 2. The rule NamedWc requires monotypes $\overline{v}$ that are substituted by the named wildcards in $\underline{Q}$ and $\underline{\tau}$. We then delegate to two subjudgments that instantiate the unnamed wildcards in respectively $\underline{Q}^w$ and τ. The rule ExtraWc states that an extra-constraints wildcard can be instantiated to an arbitrary conjunction of constraints Q_{res},

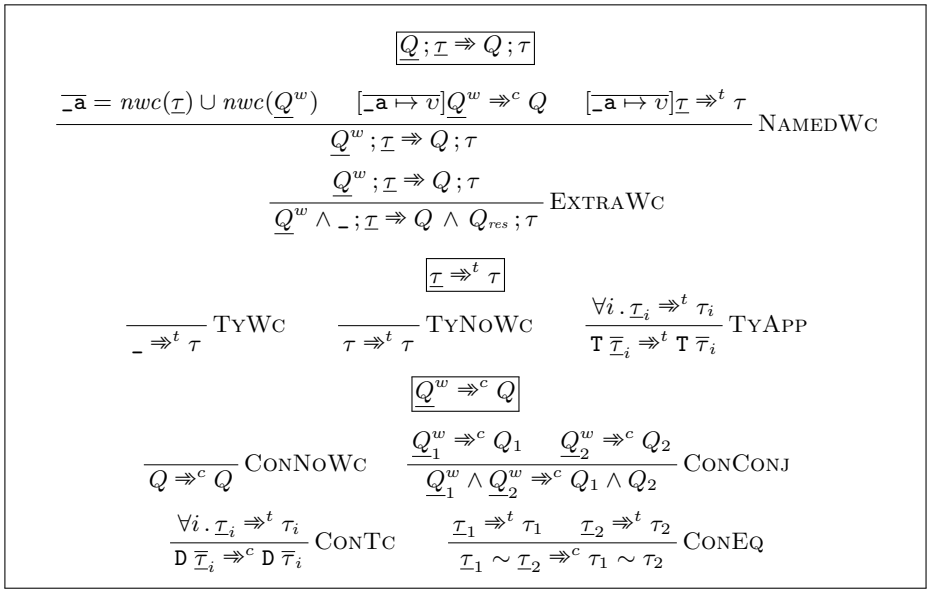


Fig. 2. Natural wildcard instantiation judgment rules

which can consist of zero or more constraints. Remember that $\underline{Q}$ can contain an extra-constraints wildcard and $\underline{Q}^w$ cannot.

The first subjudgment $\underline{\tau} \Rrightarrow^t \tau$ instantiates wildcards in a monotype to concrete types or type variables. The rule TyWc states that a type wildcard can be instantiated to any monotype τ. A monotype without wildcards is instantiated to itself (TyNoWc) and there is a congruence rule for type constructor applications (TyApp). Note that function types: $(\rightarrow)$, tuples: $(,)$, lists: $[]$, ... are all treated as type constructor applications.

The second subjudgment $\underline{Q}^w \Rrightarrow^c Q$ instantiates wildcards in constraints to concrete types or type variables. Constraints without wildcards need no further wildcard instantiation (ConNoWc). A conjunction of constraints is handled recursively in ConConj. A type-class constraint can also contain wildcards (ConTc), which will be instantiated using the previously described subjudgment. Type wildcards in equality constraints are handled in ConEq.

3.2 Wildcard Desugaring

We also define an algorithmic variant of the wildcard instantiation judgment, the *wildcard desugaring judgment*. Instead of instantiating wildcards to concrete types or type variables as the wildcard instantiation judgment does, the wildcard desugaring judgment replaces them by fresh unification variables in order to participate in OutsideIn(X)'s type inference.

The wildcard desugaring judgment $\underline{Q}\,;\underline{\tau} \Rrightarrow_a Q\,;\tau\,;\mathit{extra}$ can be read as: replacing all the wildcards in $\underline{Q}$ and $\underline{\tau}$ with fresh unification variables, gives us Q, τ, and *extra*. This last boolean output parameter indicates whether the con-

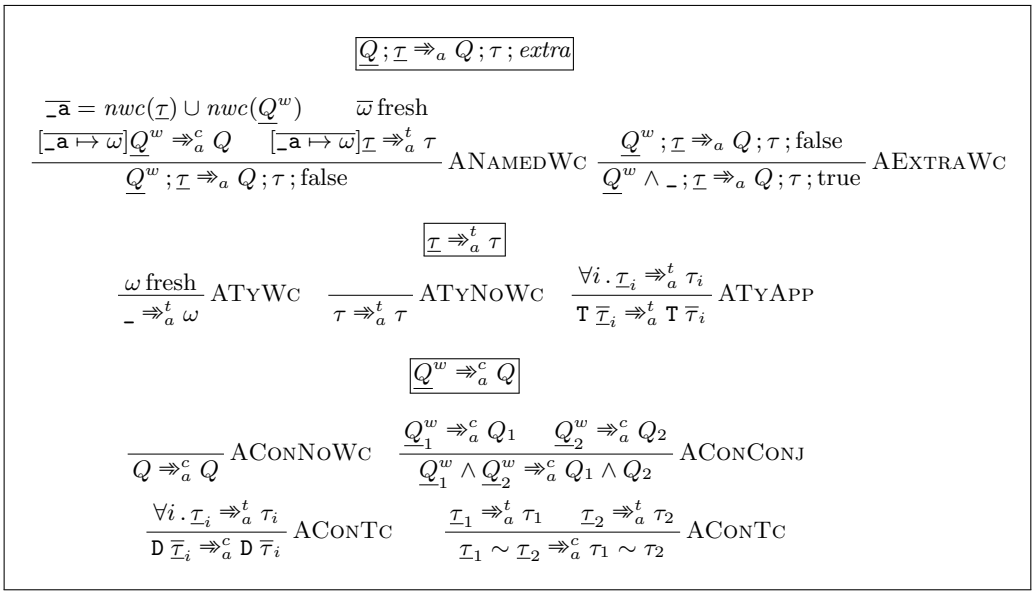


Fig. 3. Algorithmic wildcard desugaring judgment rules

straints contained an extra-constraints wildcard or not, e.g. the underscore in $_ \Rightarrow a$. If and only if *extra* = true, extra constraints can be generated.

The rules of this judgment are shown in Fig. 3. As they strongly resemble the corresponding natural rules, we shall only highlight the differences. If $\underline{Q}$ contains an extra-constraints wildcard, *extra* will be true (AExtraWc). Subsequently, or if it did not, the named wildcards in $\underline{Q}^w$ and $\underline{\tau}$ are replaced with fresh unification variables $\omega_1, \omega_2, \ldots$ (ANamedWc). Note that multiple occurrences of a named wildcard are replaced with the same unification variable. Unnamed wildcards in $\underline{\tau}$ and $\underline{Q}^w$ are desugared separately by two subjudgments $\underline{\tau} \twoheadrightarrow^t_a \tau$ and $\underline{Q}^w \twoheadrightarrow^c_a Q$ respectively. The only difference with the corresponding wildcard instantiation subjudgments is that in the rule ATyWc, a wildcard is replaced with a fresh unification variable instead of a monotype τ.

4 Typing Rules

When checking a partial type signature, the wildcards are unified with concrete types if necessary, otherwise they are replaced with fresh universally quantified type variables, i.e. the type is generalised. If an extra-constraints wildcard is present, additional constraints may be generated and added to the annotated constraints. We formalise this by adapting the OutsideIn(X) typing rules [14].

4.1 Natural Typing Rules

Figure 4 shows the three top-level natural typing rules in [14]: Empty, the base case, Bind, for definitions without a type signature, and BindA, for definitions

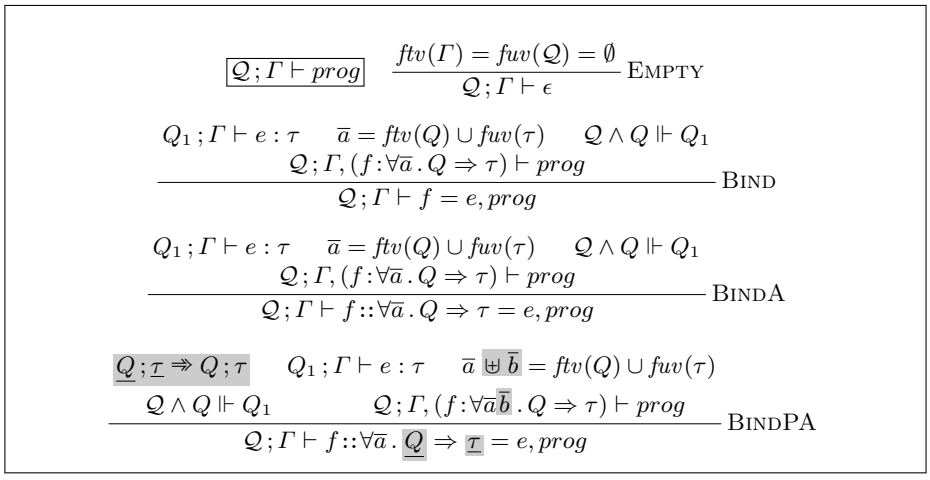


Fig. 4. Natural top-level typing rules, adapted from [14, Fig. 4, p. 15]

with a signature. It also shows the new rule BINDPA which replaces BINDA. Changes in BINDPA w.r.t. BINDA are greyed. The rules refer to the *constraint entailment* judgment $\mathcal{Q} \Vdash Q$, which should be read as: "the axioms $\mathcal{Q}$ imply Q".

Compared to BINDA, BINDPA supports partial type signatures. It is extended with the premise $\underline{Q}\,;\underline{\tau} \twoheadrightarrow Q\,;\tau$, i.e. $\underline{Q}$ and $\underline{\tau}$ are instantiated to Q and τ (see Sect. 3.1). Additional type variables that were not present in the partial type signature but arose from the generalisation of the type, are captured in $\overline{b}$, and are also universally quantified over in the final type of the top-level definition.

4.2 Constraint Solver

Before discussing the new top-level algorithmic typing rules, which make use of OUTSIDEIN(X)'s constraint solver, we shall briefly describe the constraint solver [14, Sect. 5.5]. The OUTSIDEIN(X) type inference system is parameterised by a constraint domain X. For present-day Haskell, X would be instantiated to a constraint domain that contains type-class and equality constraints (and Vytiniotis et al. present a concrete solver for this X [14]), but the OUTSIDEIN(X) typing rules and algorithms are designed to support alternative domains as well. In this text, we keep X abstract. We will only describe the form of the constraint solver, not the implementation, which is specific to X.

We have already seen the natural constraint entailment relation $\mathcal{Q} \Vdash Q$. On the algorithmic side, the constraint solver (Fig. 5) has the following signature.

$$\mathcal{Q}\,;Q_{given}\,;\overline{\alpha}_{tch} \overset{solv}{\blacktriangleright} C_{wanted} \rightsquigarrow Q_{residual}\,;\theta$$

The inputs in this signature are:

- $\mathcal{Q}$: the top-level axiom scheme. In a concrete setting, it will contain e.g. class instances or reduction rules of type functions, but we will leave it abstract.
- Q_{given}: the given constraints that arise from type annotations (or pattern matching),

– $\overline{\alpha}_{tch}$: the *touchable* unification variables that the solver is allowed to instantiate, and
– C_{wanted}: the constraints to be solved.

The outputs are:

– $Q_{residual}$: residual constraints that the solver has not been able to solve, and
– θ: a substitution mapping unification variables to types, with $dom(\theta) \subseteq \overline{\alpha}_{tch}$.

Vytiniotis et al. keep the constraint solver abstract, but require certain properties of it. It is required to be *sound* and yield *guess-free solutions*, two formal properties (specified in terms of the natural constraint entailment relation $\Vdash$) that we do not go into further. We will however require the solver to support a somewhat larger form of inputs. In the next section, we explain this further.

4.3 Wildcards in Constraints

We have chosen to allow both named and unnamed wildcards in constraints. Nevertheless, it is important to point out a limitation of such wildcards in our system. The OutsideIn(X) infrastructure will never apply unification to two constraints. Consider the following example

$h :: Eq\ _ \Rightarrow a \to a \to Bool$
$h = (\equiv)$

In this case, h's implementation generates the wanted constraint $Eq\ a$, which one might expect to be unified with $Eq\ _$, so that the wildcard is instantiated with type a, but this is not what happens. The OutsideIn(X) constraint solver does not unify the given constraint $Eq\ _$ with the wanted constraint $Eq\ a$. In general, it will never unify one constraint with another; the algorithm will only instantiate wildcards $_a$ in constraints C if

– $_a$ is a named wildcard also mentioned in the non-constraint part of the signature and it is instantiated during unification with the inferred type.
– The instantiation follows semantically from the constraint, i.e. $C \supset _a \sim ....$

In OutsideIn(X), unifying the non-constraint part of a signature with the inferred part happens through the generation of equality constraints, so in this sense the first case is comprised in the second. As a result, for h we get an error that the constraint $Eq\ a$ cannot be solved from given constraints $Eq\ _$.

Nevertheless, this limitation does not mean that wildcards in constraints are useless. Consider the following example:

$f :: Monad\ _m \Rightarrow _m\ Bool$

For this signature, $_m$ can either be unified with a concrete type constructor like *Maybe* for which there is a *Monad* instance or be generalised to a universally quantified monad m. Similarly, we can say something like:

$g :: (_a, _) \sim F\ _b \Rightarrow _b \to _a$

This signature states that g is a function whose domain type is mapped by type function F to a tuple whose first element is its range type.

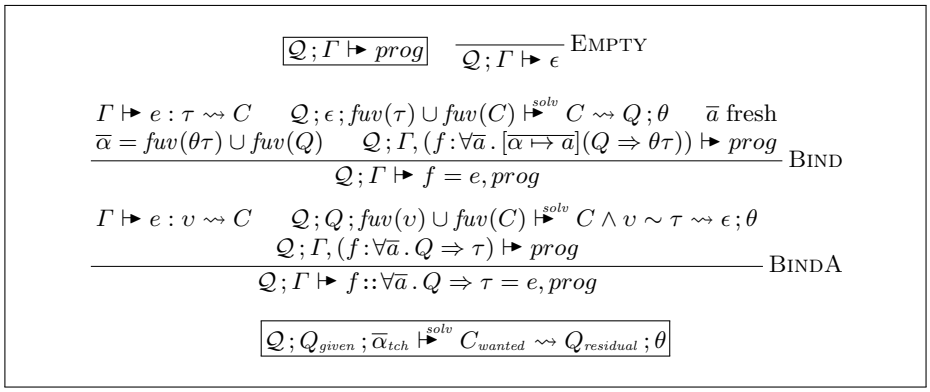


Fig. 5. Top-level algorithmic rules, taken from [14, Fig. 12, page 39]

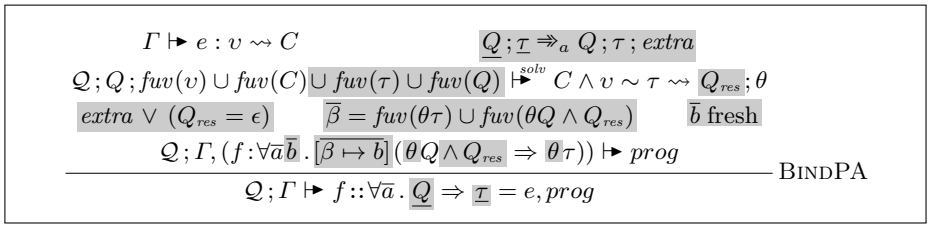


Fig. 6. New top-level algorithmic rule, adapted from Fig. 5

Formally, the choice to allow wildcards in constraints implies that we have to drop an invariant of the constraint solver. For the constraint solver (Sect. 4.2) Vytiniotis et al. mention two invariants that should hold: $\overline{\alpha}_{\mathit{tch}} \# \mathit{fuv}(Q_{\mathit{given}})$ and $\mathit{dom}(\theta) \# \mathit{fuv}(Q_{\mathit{given}})$, i.e. the free unification variables in Q_{given} should not be unified. This means that the given constraints Q_{given} are not allowed to contain unification variables that the solver can instantiate. In order to support wildcards in constraints, it is required to remove this restriction and we propose to do so. Although the proofs about the concrete constraint solver in the second part of Vytiniotis et al.'s paper [14] rely upon this restriction, we conjecture that this is a technical restriction that can be remedied.

Contrary to the behaviour of wildcards in the non-constraint part of a signature, some of the behaviour of wildcards in constraints we just discussed could be unexpected by programmers. Because of this, one might consider disallowing both named and unnamed type wildcards in constraints, but we have chosen not to do so. We think the limitations of wildcards in constraints can be explained to the user and our examples show that they can be useful despite the limitations.

4.4 Algorithmic Typing Rules

In addition to the top-level natural typing rules, we also adapt the top-level algorithmic typing rules. The original top-level algorithmic typing rules are shown in Fig. 5. As wildcards can only occur in a type signature, only the rule BindA

that handles declarations with a type annotation has to be adapted. The adapted rule is presented in Fig. 6, with changes w.r.t. BindA highlighted in grey.

The BindPA rule works as follows. First, the type v of e is inferred using the constraint generation judgment from [14] while generating the constraints C. The wildcards in $\underline{Q}$ and $\underline{\tau}$ are replaced with fresh unification variables with the wildcard desugaring judgment we defined earlier. The *extra* output parameter indicates whether we are allowed to infer extra constraints.

On the second line, the invocation of the constraint solver has been slightly modified. The free unification variables in τ and Q, introduced during the wildcard desugaring, are added to the set of touchable unification variables that the constraint solver is allowed to instantiate. We also capture the residual constraints, which were not allowed in the previous version of the rule, in Q_{res}. Now they are allowed, but only if *extra* is true.

In the next step, we collect the remaining free unification variables in $\theta\tau$ and $\theta Q \wedge Q_{res}$. These unification variables were not instantiated to concrete types while solving the constraints and so we generalise over them. They are replaced with fresh, universally quantified type variables, $\overline{b}$. The residual constraints, i.e. the extra constraints that have not been solved by the constraint solver, are added to the annotated constraints.

Theorem 1 (Algorithm soundness). *If* $\mathcal{Q}; \Gamma \mapsto$ *prog then* $\mathcal{Q}; \Gamma \vdash$ *prog in a closed top-level* Γ*.*

5 Typing of Local Definitions

Advanced type system features like GADTs have a profound impact on a type system. Crucially, the clean and simple principal typing property that the HM system satisfies is no longer valid [14]. This makes type inference a harder problem and Vytiniotis et al. present one possible way out. They advocate the policy that the types of local (unannotated) definitions should not be generalised, with the slogan "Let should not be generalised".

For partial type signatures of local definitions, we align with the policy to not generalise local definitions. Next, we present the adapted typing rules for local definitions, but we omit natural typing rules as the required changes are minimal. The existing algorithmic rules and our adapted rule are shown in Fig. 7.

The rule LetA applies to definitions with an annotated monomorphic type, GLetA for polymorphic type signatures and Let for definitions without a signature. The rule Let is remarkably simple, as it applies the NoGen policy of not generalising the inferred type at all. Our adapted typing rule GLetPA extends this policy to partial type signatures.

The GLetPA rule applies to local bindings with a partial type signature, either polymorphic or monomorphic. It first desugars the partial type signature. The *extra* parameter must be false, i.e. we forbid an extra-constraints wildcard, since the NoGen policy forbids additional constraints. We verify that the type signature was indeed partial by requiring free unification variables in the desugared type and constraints. Next, the set of unification variables allowed to unify,

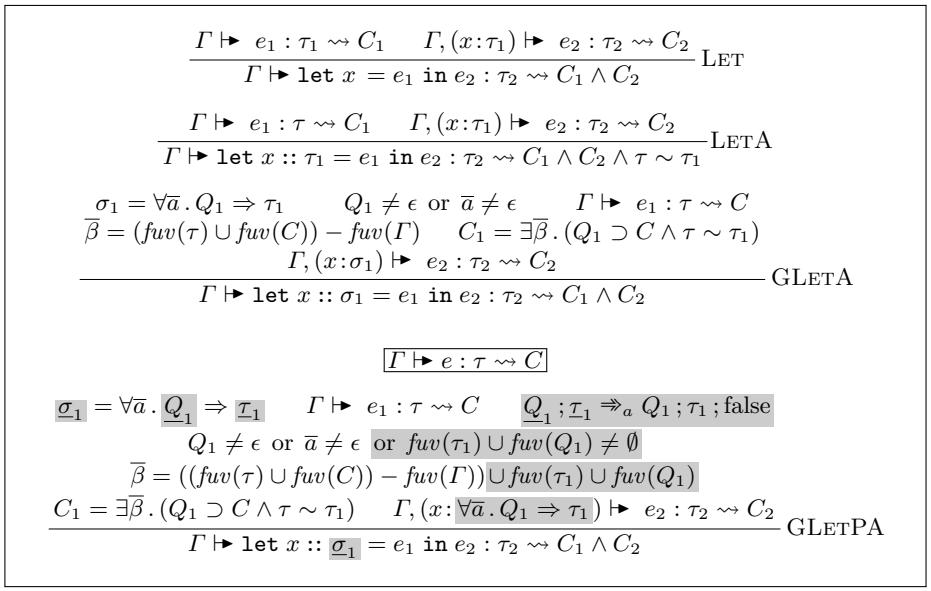


Fig. 7. Constraint generation for local `let`-bound definitions, taken and adapted from [14, Fig. 13, page 40]

i.e. the *touchables*, is extended with those resulting from the wildcard desugaring. Solving the implication constraint should unify them, fixing the definition's actual type. The local binding, annotated with the desugared type, is added to the environment to type check the body e_2. Following the NoGen policy, no generalisation is performed. The example *foo* shows the effect of not generalising:

```
foo = let g :: _ → _
          g x = x
          h :: Eq _a ⇒ _a → _a → Bool
          h x y = x ≡ y
      in (g True, g 'v', h True True, h 'a' 'b')
```

Instead of being quantified over, the free unification variables in the type of g unify with the *Bool* type at the first call of g. Thus, g's type is $\mathit{Bool} \to \mathit{Bool}$. As g is also called with a *Char* argument, the program will be rejected. Similarly, the unification variable for the named wildcard $_a$ in h's type is not generalised. Instead, it unifies with the *Bool* type, producing the type $\mathit{Eq\ Bool} \Rightarrow \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}$ for h.

6 Alignment with Existing Rules

Partial type signatures are a generalisation of the binary choice between a full signature or none at all. Using wildcards, partial type signatures can mix annotated and inferred types. To demonstrate that partial type signatures truly are a generalisation of the existing inference, we prove two properties.

First, partial type signatures are a conservative extension: the adapted typing rules are equivalent to the original rules for signatures without wildcards.

Second, (top-level) definitions without a type signature are equivalent to definitions with the partial type signature $_ \Rightarrow _$. More formally: the BindPA rule (Fig. 6) can be used to type check a definition $f = e$ without a type signature by treating it as if it had the partial type signature $f :: _ \Rightarrow _ = e$. The AltBind rule provides the definitions without type signature with the equivalent partial type signature.

$$\frac{\mathcal{Q}\,;\Gamma \vdash\!\blacktriangleright f :: _ \Rightarrow _ = e, prog}{\mathcal{Q}\,;\Gamma \vdash\!\blacktriangleright f = e, prog}\ \textsc{AltBind}$$

Theorem 2. *Given a program prog in which every definition f has either a type signature without wildcards, i.e. $f :: \forall \overline{a}\,.\,Q \Rightarrow \tau = e$, or no type signature at all, i.e. $f = e$. If $\mathcal{Q}\,;\Gamma \vdash\!\blacktriangleright prog$, using* Bind, BindA, *and* Empty *(Fig. 5), then $\mathcal{Q}\,;\Gamma \vdash\!\blacktriangleright prog$, using* AltBind, BindPA *(Fig. 6), and* Empty *(Fig. 5).*

These properties show that our proposal aligns well with the existing behaviour of type inference. This is not just theoretically important, but also shows that our proposal is natural and unsurprising for existing users.

7 Implementation and Extensions

We have developed an implementation of our proposal in the de facto standard Haskell compiler GHC. GHC's inferencer is based on the OutsideIn(X) type inference system. As a result, our proposal fits relatively nicely into the compiler's inference infrastructure. Nevertheless, GHC's actual inferencer is (unavoidably) more complex than Vytiniotis et al.'s elegant theory, notably when it comes to the inference and generalisation of mutually recursive blocks and higher-rank types. Hence, our prototype currently implements only part of our theoretical development. More specifically, it correctly unifies wildcards and named wildcards with closed types, but does not yet support unifying with open types, generalisation and extra-constraints wildcards. The prototype code is available for download at `http://github.com/mrBliss/ghc`. We still intend to check and ensure compatibility with the ScopedTypeVariables [8] and ConstraintKinds [1,18] extensions, but we expect no major problems there.

8 Related Work

Vytiniotis et al. provide a comprehensive overview of work on constraint-based type systems and type inference for advanced type system features that we do not repeat here [14], except to discuss aspects related to *partial* type signatures. Vytiniotis et al. claim that their presentation is the first one that deals with local assumptions introduced by type signatures and data constructors, and where those local assumptions may include type equalities.

The idea of partial type signatures is not new. The topic regularly comes up on the Haskell community mailing lists. In two 2006 tickets on the Haskell Prime wiki (where the Haskell community proposes and tracks future language changes), Malcolm Wallace proposes a form of partial type signatures [15,16]. His proposal seems similar to ours, but it does not contain a lot of detail. A GHC feature request has also been logged to request a form of constraint wildcards [13].

The *Agda* programming language [7] has a dependent type system, which allows terms in types and vice versa. The type system allows more powerful type-level computations, so that type inferencing becomes harder. On the other hand, the inferencer can sometimes infer terms as well. In Agda, any value or type can be replaced by an underscore, in which case Agda will try to infer it. Agda's inference does not perform generalisation: if the type checker cannot infer the value of such a meta-variable, it reports an error.

Our work was inspired by the partial signatures in Dijkstra and Swierstra's *Explicit Haskell* [4][3, Chapt. 10]. They also use wildcards and allow predicate wildcards very similar to our extra-constraints wildcards. However, where we follow Vytiniotis et al. in using a rather standard form of HM style type generalisation, Dijkstra and Swierstra use *quantifier location inference* rules that differ significantly, both for normal and partial type signatures. They argue that depending on the structure of the type in which a type variable appears, it should either be existentially or universally quantified to align with user expectations. For example, the type $a \to a$ is interpreted as $\forall a.a \to a$ but $(a \to a) \to Int$ is interpreted as $(\forall a.a \to a) \to Int$, unlike Haskell. In a product type, the variables are quantified existentially instead of universally, e.g. (a, a) is interpreted as $\exists a.(a, a)$ and $(a, a) \to Int$ as $(\exists a.(a, a)) \to Int$. Dijkstra and Swierstra formalise Explicit Haskell, but do not prove results like our Theorem 1 and Theorem 2.

For the *Chameleon* programming language, Sulzmann and Wazny describe a form of *existential type signatures*, supported in addition to standard *universal* signatures [12,17]. Type variables in a universal signature $f :: a \to a$ are interpreted in the same way as Haskell, i.e. as $f ::\forall a.a \to a$. However, in an existential type signature $f ::: a \to a$ (note: three colons) the variables are interpreted more or less like our named wildcards, so that it becomes equivalent to our $f ::_a \to _a$. A mixture of existential and universal annotations is not supported, but can be encoded by nesting existential in universal annotations.

Both F_{ML} [11] and *HMF* [6] combine the expressiveness of System F with the convenience of Hindley-Milner type inference, while remaining a conservative extension of ML and HM respectively. Both solutions employ *partial type annotations* to avoid the *guessing* of polymorphic types during type inference. These partial type annotations are similar to the ones in the introduction, which use the ScopedTypeVariables extension. Furthermore, they support partial type annotations of the following form: $e :: \exists \overline{\alpha}\,.\,\sigma$, where the free variables $\overline{\alpha}$ in σ are locally bound. This should be read as "for some types $\overline{\alpha}$, the expression e has type σ" and the $\overline{\alpha}$ correspond to our named wildcards. The authors formalised these partial type annotations, including generalisation, for a HM-based type system, but without considering GADTs or local type assumptions.

9 Conclusion

Partial type signatures are a useful feature that has often been requested and proposed for Haskell. They bridge the gap between complete type annotations and none at all. Our proposal pins down the precise behaviour and we formally prove its well-behavedness. The result fits naturally in both the existing formal description of GHC's type inferencer (OutsideIn(X)) and the implementation. The idea of partial type signatures is not novel, but we believe our proposal is the first that supports all the features necessary for present-day GHC Haskell, esp. local constraint assumptions.

References

1. Bolingbroke, M.: Constraint kinds for GHC. online (2011), `http://blog.omega-prime.co.uk/?p=127`
2. Claessen, K., Axelsson, E.: The patch-combinators package. Hackage (2012), `http://hackage.haskell.org/package/patch-combinators`
3. Dijkstra, A.: Stepping Through Haskell. Ph.D. thesis, Universiteit Utrecht (2005)
4. Dijkstra, A., Swierstra, D.S.: Making implicit parameters explicit. Tech. Rep. UU-CS-2005-032, Universiteit Utrecht (2005)
5. Kiselyov, O.: Partial signatures (August 2004), `http://okmij.org/ftp/Haskell/types.html#partial-sigs`, visited on 31/12/2012
6. Leijen, D.: HMF: simple type inference for first-class polymorphism. In: ICFP. pp. 283–294. ACM (2008)
7. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University and Göteborg University (2007)
8. Peyton Jones, S., Shields, M.: Lexically-scoped type variables (2004)
9. Peyton Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. J. Funct. Program. 17(1), 1–82 (2007)
10. Pierce, B.: Types and programming languages. MIT Press (2002)
11. Rémy, D.: Simple, partial type-inference for System F based on type-containment. In: ICFP. pp. 130–143. ACM (2005)
12. Sulzmann, M., Wazny, J.: Lexically scoped type annotations (2005), `http://www.cs.mu.oz.au/~sulzmann/manuscript/lexical-annot.ps`, manuscript
13. Various authors: Infer type context in a type signature. GHC Ticket (2011), `http://hackage.haskell.org/trac/ghc/ticket/5248`
14. Vytiniotis, D., Peyton Jones, S., Schrijvers, T., Sulzmann, M.: OutsideIn(X): Modular type inference with local assumptions. J. Funct. Program. 21(4-5), 333–412 (2011)
15. Wallace, M.: Partial type signatures/annotations. Haskell Prime Wiki (Feb 2006), `http://ghc.haskell.org/trac/haskell-prime/wiki/PartialTypeAnnotations`
16. Wallace, M., et al.: Partial type signatures. Haskell Prime Wiki (Jan 2006), `http://ghc.haskell.org/trac/haskell-prime/wiki/PartialTypeSigs`
17. Wazny, J.: Type inference and type error diagnosis for Hindley/Milner with extensions. Ph.D. thesis, University of Melbourne (2006)
18. Yorgey, B.A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., Magalhães, J.P.: Giving haskell a promotion. In: TLDI. pp. 53–66 (2012)