Partial Type Signatures for Haskell
Extended Version with Proofs of the Theorems

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

[1] KU Leuven firstname.lastname@cs.kuleuven.be
[2] UGent 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 of a value or function (and rely on type inference) or explicitly providing the type entirely; there are no intermediate options. Some 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 that can be used with present-day Haskell. We formally describe our proposal as an extension of the OutsideIn(X) system and we prove some of its properties. We have developed a (not yet complete) implementation for the GHC Haskell compiler. We find that our design fits naturally in both the OutsideIn(X) formalism and the compiler. Our wildcards map nicely to HM unification variables and our generalisation rules are carefully designed to align with existing behaviour.
This is the extended version of a paper submitted to the PADL symposium. It is identical to that paper except for two appendices containing proofs of our main theorems.

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 a programmer's intentions.

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

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

Under Haskell's defaulting, x and y are interpreted as floating point numbers leading to the inexact output `2.4000000000000004`. The exact output $\frac{12}{5}$ is produced with the signature $\mathit{harmonic} :: \mathit{Rational} \to \mathit{Rational} \to \mathit{Rational}$. 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. [1, §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. [2]. 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:

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

However, such a solution only applies if the parts of the type we want/do not want to specify happen to coincide with the type of one or more input variables and the rest. Expression type signatures similarly provide a partial solution. Another way to simulate partial type signatures uses explicitly typed identity functions in the implementation; we could write *foo* for example as follows:

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

A downside is that *foo*'s implementation is obscured with computationally insignificant code and Kiselyov has proposed to place such code in fake clauses [3]:

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

A combinator library supports this technique [4]. 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.

$$bar :: \_ \to \_$$
$$bar\ \_ = True$$

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 $\forall a.a \to Bool$, like 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 (where the wildcard would be instantiated to $(Num\ a, Show\ a)$) [5]:

$$f :: \_ \Rightarrow [a] \to String$$
$$f\ xs = show\ (sum\ xs)$$

We also allow multiple references to a wildcard within a signature using *named wildcards* (written as e.g. $\_a$). In the following example, we use them to shorten a tedious type signature.

$$isMeltdown :: NukeMonad\ param1\ param2\ Bool$$
$$unlessMeltdown :: \_nm\ () \to \_nm\ ()$$

$$\begin{aligned} &\mathit{unlessMeltdown}\ c = \mathbf{do}\ m \leftarrow \mathit{isMeltdown} \\ &\qquad\qquad\qquad\qquad \mathbf{if}\ m\ \mathbf{then}\ \mathit{return}\ ()\ \mathbf{else}\ c \end{aligned}$$

To make our proposal precise, we give a formal account based on Vytiniotis et al.'s OutsideIn(X) formalism [6]. 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 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.

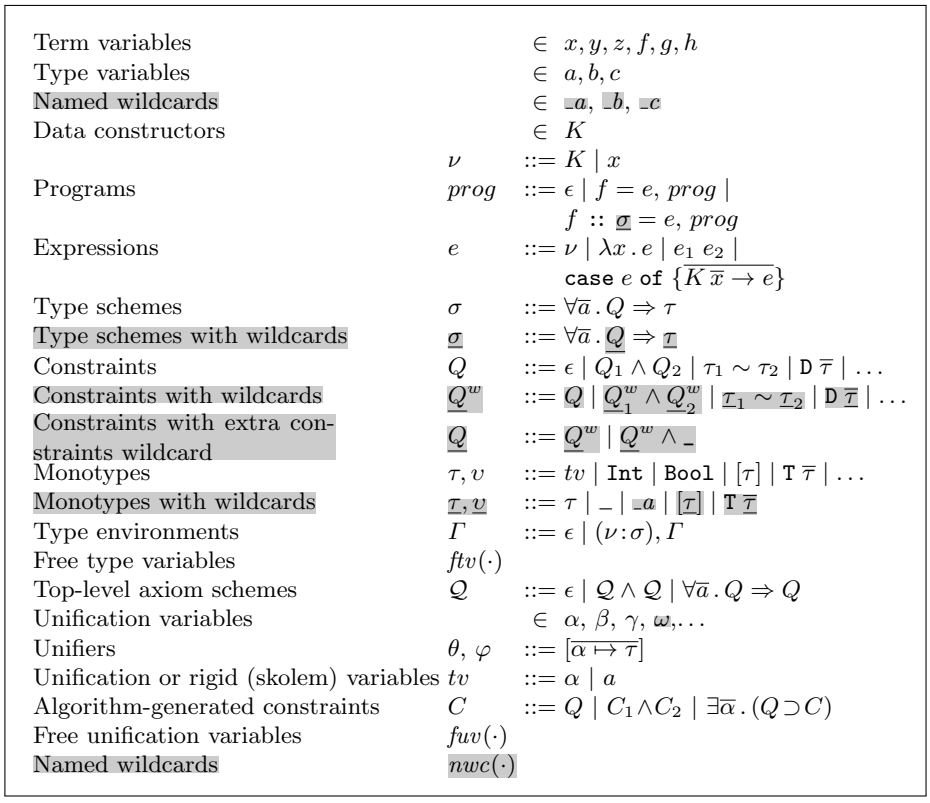
Contributions The idea of partial type signatures is not novel. Several languages support them in some form or other [7,8] and they have been proposed for Haskell several times before [9,10]. Dijkstra [11] and Sulzmann and Wazny [12,13] have detailed proposals for Haskell-like languages. Still, we believe 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) [6], 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.

Outline In Section 2, we describe our additional syntax, both informally and formally. Formal rules for handling wildcard syntax are listed in Section 3. We extend OutsideIn(X) typing rules to support wildcards in Section 4. Local bindings with partial type signatures are described in Section 5. We prove the correspondence of our rules to the standard ones for the uninformative signature $\_ \Rightarrow \_$ and for signatures without wildcards in section 6. We discuss our implementation in Section 7, related work in Section 8 and conclude in Section 9.

This extended version is identical to the submitted version except for the two appendices which contain the proofs of our formal results, which are omitted from the submitted version for space reasons.



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	$::= \epsilon \mid Q_1 \wedge Q_2 \mid \tau_1 \sim \tau_2 \mid \texttt{D}\ \overline{\tau} \mid \ldots$
Constraints with wildcards	$\underline{Q}^w$	$::= \underline{Q} \mid \underline{Q}^w_1 \wedge \underline{Q}^w_2 \mid \underline{\tau}_1 \sim \underline{\tau}_2 \mid \texttt{D}\ \underline{\overline{\tau}} \mid \ldots$
Constraints with extra constraints wildcard	$\underline{Q}$	$::= \underline{Q}^w \mid \underline{Q}^w \wedge \_$
Monotypes	τ, υ	$::= tv \mid \texttt{Int} \mid \texttt{Bool} \mid [\tau] \mid \texttt{T}\ \overline{\tau} \mid \ldots$
Monotypes with wildcards	$\underline{\tau}, \underline{\upsilon}$	$::= \tau \mid \_ \mid \_a \mid [\underline{\tau}] \mid \texttt{T}\ \underline{\overline{\tau}}$
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 [6, Fig. 1, page 12 and Fig. 5, page 17]

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) [6]. 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 Section 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 wildcard occurring *as* a constraint is an *extra-constraints wildcard*, e.g. $\_ \Rightarrow \_$. 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 Figure 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 Section 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 Figure 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}, 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 $\underline{\tau} \twoheadrightarrow^t \tau$ 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: $(\to)$, tuples: $(,)$, lists: $[]$, ... are all treated as type constructor applications.

The second subjudgment $\underline{Q}^w \twoheadrightarrow^c Q$ 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.

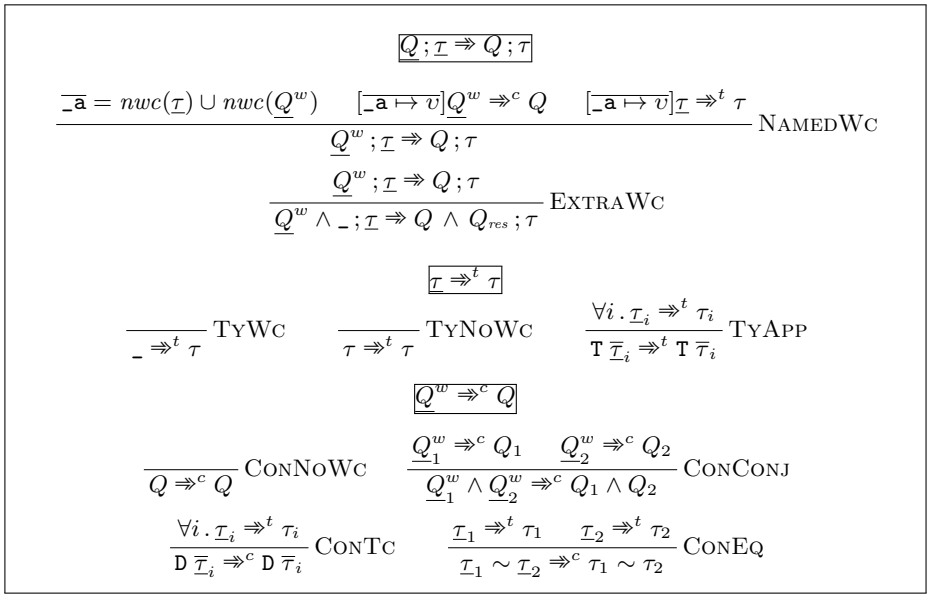


Fig. 2. Natural wildcard instantiation judgment rules

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 $\underline{Q}\,;\underline{\tau} \twoheadrightarrow_a Q\,;\tau\,;\mathit{extra}$ 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 constraints contained an extra-constraints wildcard or not, e.g. the first underscore in $\_ \Rightarrow \_$. If and only if $\mathit{extra} = \text{true}$, extra constraints can be generated.

The rules of this judgment are shown in Figure 3. As their structure resembles the structure of the rules of the wildcard instantiation judgment, 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 contain such a wildcard, the named wildcards in $\underline{Q}^w$ and $\underline{\tau}$ are collected and replaced with fresh unification variables $\omega_1, \omega_2, \ldots$ in the rule 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 $\underline{\tau} \twoheadrightarrow_a^t \tau$ and $\underline{Q}^w \twoheadrightarrow_a^c Q$ 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.

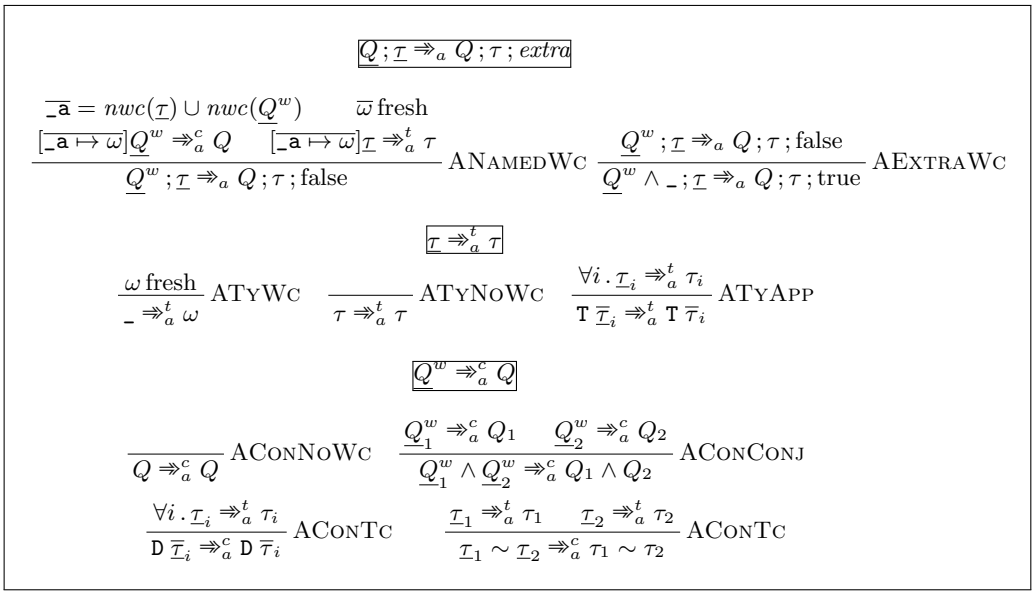


Fig. 3. Algorithmic wildcard desugaring judgment rules

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 [6].

4.1 Natural typing rules

Figure 4 shows the three top-level natural typing rules in [6]: Empty, the base case, Bind, for definitions without a type signature, and BindA, for definitions 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 Section 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 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

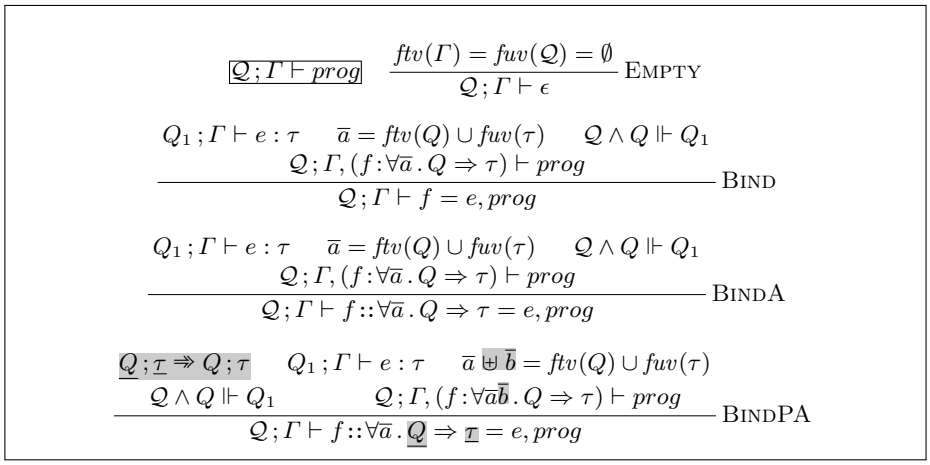


Fig. 4. Natural top-level typing rules, adapted from [6, Fig. 4, p. 15]

solver [6, Section 5.5, p. 41]. The OUTSIDEIN(X) type inference system is in fact 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 [6]), 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 (Figure 5) has the following signature.

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

The inputs in this signature are:

- $\mathcal{Q}$: the top-level axiom scheme. In a concrete setting, it will contain for example class instances or reduction rules of type functions, but we will leave it abstract. It does not change during type-checking.
- 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.

Formally, the choice to allow wildcards in constraints implies that we have to drop an invariant of the constraint solver. For the constraint entailment relation $\mathcal{Q}; Q_{given}; \overline{\alpha}_{tch} \overset{solv}{\mapsto} C_{wanted} \rightsquigarrow Q_{residual}; \theta$, Vytiniotis et al. mention two invariants that should hold: $\overline{\alpha}_{tch} \# fuv(Q_{given})$ and $dom(\theta) \# fuv(Q_{given})$, 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 [6] 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

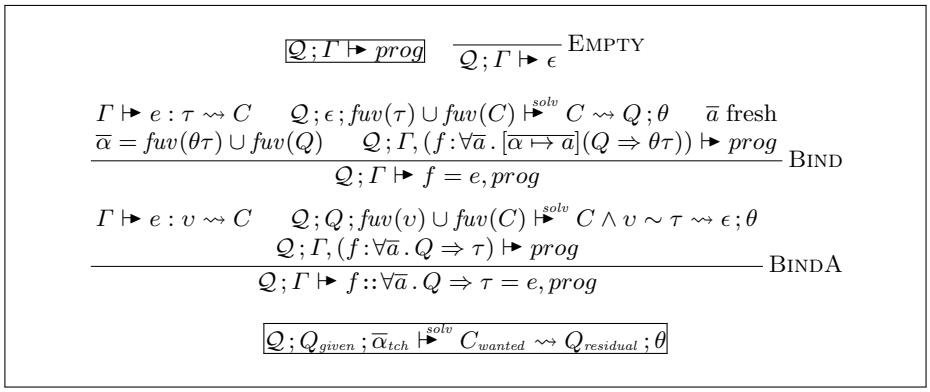


Fig. 5. Top-level algorithmic rules, taken from [6, Fig. 12, page 39]

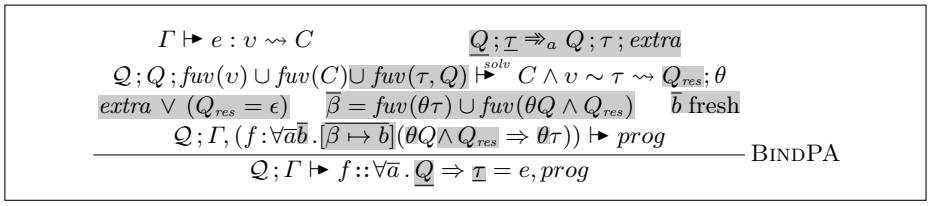


Fig. 6. New top-level algorithmic rule, adapted from Fig. 5

unexpected by programmers. Because of this, one might consider the possibility to disallow both named and unnamed type wildcards in constraints, but we have chosen not to do this. 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 new top-level natural typing rule BindPA, we also adapt the top-level algorithmic typing rules. The original top-level algorithmic typing rules are shown in Figure 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 Figure 6, with changes w.r.t. BindA highlighted in grey.

The BindPA rule works as follows. We start with the first expression: the type υ of e is inferred using the constraint generation judgment from [6] 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

$$\frac{\Gamma \vdash\!\blacktriangleright\ e_1 : \tau_1 \rightsquigarrow C_1 \qquad \Gamma, (x{:}\tau_1) \vdash\!\blacktriangleright\ e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \vdash\!\blacktriangleright\ \texttt{let}\ x\ = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2}\ \textsc{Let}$$

$$\frac{\Gamma \vdash\!\blacktriangleright\ e_1 : \tau \rightsquigarrow C_1 \qquad \Gamma, (x{:}\tau_1) \vdash\!\blacktriangleright\ e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \vdash\!\blacktriangleright\ \texttt{let}\ x :: \tau_1 = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2 \wedge \tau \sim \tau_1}\ \textsc{LetA}$$

$$\frac{\begin{array}{c}\sigma_1 = \forall \overline{a}\,.\,Q_1 \Rightarrow \tau_1 \qquad Q_1 \neq \epsilon \text{ or } \overline{a} \neq \epsilon \qquad \Gamma \vdash\!\blacktriangleright\ e_1 : \tau \rightsquigarrow C \\ \overline{\beta} = (\mathit{fuv}(\tau) \cup \mathit{fuv}(C)) - \mathit{fuv}(\Gamma) \qquad C_1 = \exists \overline{\beta}\,.\,(Q_1 \supset C \wedge \tau \sim \tau_1) \\ \Gamma, (x{:}\sigma_1) \vdash\!\blacktriangleright\ e_2 : \tau_2 \rightsquigarrow C_2\end{array}}{\Gamma \vdash\!\blacktriangleright\ \texttt{let}\ x :: \sigma_1 = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2}\ \textsc{GLetA}$$

$$\boxed{\Gamma \vdash\!\blacktriangleright\ e : \tau \rightsquigarrow C}$$

$$\frac{\begin{array}{c}\underline{\sigma_1} = \forall \overline{a}\,.\,\underline{Q_1} \Rightarrow \underline{\tau_1} \qquad \Gamma \vdash\!\blacktriangleright\ e_1 : \tau \rightsquigarrow C \qquad \underline{Q_1}\,;\underline{\tau_1} \twoheadrightarrow_a Q_1\,;\tau_1\,;\text{false} \\ Q_1 \neq \epsilon \text{ or } \overline{a} \neq \epsilon \text{ or } \mathit{fuv}(\tau_1) \cup \mathit{fuv}(Q_1) \neq \emptyset \\ \overline{\beta} = ((\mathit{fuv}(\tau) \cup \mathit{fuv}(C)) - \mathit{fuv}(\Gamma)) \cup \mathit{fuv}(\tau_1) \cup \mathit{fuv}(Q_1) \\ C_1 = \exists \overline{\beta}\,.\,(Q_1 \supset C \wedge \tau \sim \tau_1) \qquad \Gamma, (x{:}\forall \overline{a}\,.\,Q_1 \Rightarrow \tau_1) \vdash\!\blacktriangleright\ e_2 : \tau_2 \rightsquigarrow C_2\end{array}}{\Gamma \vdash\!\blacktriangleright\ \texttt{let}\ x :: \underline{\sigma_1} = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2}\ \textsc{GLetPA}$$

Fig. 7. Constraint generation for local `let`-bound definitions, taken and adapted from [6, Fig. 13, page 40]

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.

Exercise to the reader: apply the rule BindPA to the following example.

$f :: \_ \Rightarrow \_ \to \_ \to \mathit{Bool} \to \_$
$f\ x\ y\ b = x \equiv y \wedge b$

Theorem 1 (Algorithm soundness). *If $\mathcal{Q}\,;\Gamma \vdash\!\blacktriangleright\ prog$ 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 (see e.g. [6]). 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 Figure 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, because 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, i.e. the *touchables*, is extended with those resulting from the wildcards in the desugared τ_1 and Q_1. 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 in GLetPA.

$$
\begin{aligned}
\mathit{foo} = \textbf{let}\ & g :: \_ \to \_ \\
& g\ x = x \\
& h :: \mathit{Eq}\ \_a \Rightarrow \_a \to \_a \to \mathit{Bool} \\
& h\ x\ y = x \equiv y \\
\textbf{in}\ & (g\ \mathit{True}, g\ \texttt{'v'}, h\ \mathit{True}\ \mathit{True}, h\ \texttt{'a'}\ \texttt{'b'})
\end{aligned}
$$

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 $\mathit{Eq}\ \mathit{Bool} \Rightarrow \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}$ 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, definitions without a type signature are equivalent with definitions with a partial type signature of the form $\_ \Rightarrow \_$. More formally: the BindPA rule (Figure 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 following AltBind rule simply transforms definitions without a type signature into definitions 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 *(Figure 5), then* $\mathcal{Q}\,;\Gamma \vdash\!\!\blacktriangleright prog$*, using* AltBind*,* BindPA *(Figure 6), and* Empty *(Figure 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 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 [14] and ConstraintKinds [15,16] 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 [6], except to discuss aspects related to *partial* type signatures. Vytiniotis 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 [10,9]. 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 [5].

The programming language *Alice*, an extended version of Standard ML focused on concurrent and constraint programming, also features type wildcards. From the short paragraph in the online manual [8] on type wildcards, they appear similar to our partial type signatures (including w.r.t. the generalisation) but there are no constraint wildcards, as Alice does not have constraints. We have not found a formal description of Alice's type wildcards.

The *Agda* programming language [7] has a dependent type system, that does not make a strict distinction between types and values. The type system allows a powerful form of type-level computation, so that type inferencing becomes harder. On the other hand, terms may appear in types, so that 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 the value from the available type information using a conservative unification-based inference approach that performs well in practice. Agda's type inference does not perform any form of generalisation: if the type checker cannot infer the value of such a meta-variable, it just reports an error.

Our work was inspired by the partial signatures in Dijkstra and Swierstra's *Explicit Haskell* [11][17, Chapter 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 theorems 1 and 2.

For the *Chameleon* programming language, Sulzmann and Wazny describe a form of *existential type signatures*, supported in addition to standard *universal* signatures [12,13]. 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 in more or less like our named wildcards, so that the signature becomes equivalent to our $f :: \_a \to \_a$. A mixture of existential and universal annotations is not supported, but can be encoded by nesting an existential annotation in a universal one. Analogously to the *scoped type variables* extension, the variables in these signatures share the same scope.

Both F_{ML} *[18] and HMF [19]* combine the expressiveness of System F, including first-class polymorphism, 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 poly-

morphic types during type inference. These partial type annotations are similar to the ones described in our discussions of the *scoped type variables* Haskell extension, but without the scoping aspect. 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. The annotation should be read as "for some (monomorphic) 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, in the context of a HM-based type system. Neither Rémy nor Leijen consider 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. Pierce, B.: Types and programming languages. MIT Press (2002)
2. Peyton Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. J. Funct. Program. **17**(1) (2007) 1–82
3. Kiselyov, O.: Partial signatures (August 2004) Visited on 31/12/2012.
4. Claessen, K., Axelsson, E.: The patch-combinators package. Hackage (2012)
5. Various authors: Infer type context in a type signature. GHC Ticket (2011)
6. Vytiniotis, D., Peyton Jones, S., Schrijvers, T., Sulzmann, M.: OutsideIn(X): Modular type inference with local assumptions. J. Funct. Program. **21**(4-5) (2011) 333–412
7. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University and Göteborg University (2007)
8. Alice development team: Alice manual - extensions to the SML type language (March 2007) Visited on 30/12/2012.
9. Wallace, M.: Partial type signatures/annotations. Haskell Prime Wiki (Feb. 2006)
10. Wallace, M., et al.: Partial type signatures. Haskell Prime Wiki (Jan. 2006)
11. Dijkstra, A., Swierstra, D.S.: Making implicit parameters explicit. Technical Report UU-CS-2005-032, Universiteit Utrecht (2005)
12. Sulzmann, M., Wazny, J.: Lexically scoped type annotations. manuscript (2005)
13. Wazny, J.: Type inference and type error diagnosis for Hindley/Milner with extensions. PhD thesis, University of Melbourne (2006)
14. Peyton Jones, S., Shields, M.: Lexically-scoped type variables. (2004)
15. Bolingbroke, M.: Constraint kinds for GHC. online (2011)
16. Yorgey, B.A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., Magalhães, J.P.: Giving haskell a promotion. In: TLDI. (2012) 53–66

17. Dijkstra, A.: Stepping Through Haskell. PhD thesis, Universiteit Utrecht (2005)
18. Rémy, D.: Simple, partial type-inference for System F based on type-containment. In: ICFP, ACM (2005) 130–143
19. Leijen, D.: HMF: simple type inference for first-class polymorphism. In: ICFP, ACM (2008) 283–294

A Modified constraint solver lemma

Because of the changed invariants of the constraint solver judgment $\mathcal{Q}\,;Q_{given}\,;\overline{\alpha}_{tch} \overset{solv}{\mapsto} C_{wanted} \rightsquigarrow Q_{residual}\,;\theta$ that we discussed in section 4.3, we need to modify one of the lemmas employed in OutsideIn(X).

Lemma 1. *Assume that* $\Gamma \mapsto e : \tau \rightsquigarrow C$*. Then, for all* C_{ext}*, if* $\mathcal{Q}\,;Q_g\,;\overline{\beta} \overset{solv}{\mapsto} C \wedge C_{ext} \rightsquigarrow Q_r\,;\theta$ *then there exists* Q *such that* $Q\,;\theta\Gamma \vdash e : \theta\tau$ *and* $\mathcal{Q} \wedge \theta Q_g \wedge Q_r \vdash Q$*.*

This lemma replaces Vytiniotis et al.'s lemma 5.1. The only modification is that in the conclusion, we apply the produced substitution θ to the given constraints Q_g as well, since given constraints can now also contain unification variables, produced from wildcards. This change follows from the changes made in section 4.3 and more background can be found there.

B Proof for Theorem 1

Proof. To prove the soundness of the OutsideIn(X) algorithm, we need to prove that when a program is well-typed according to the algorithmic typing rules, it will also be well-typed according to the natural typing rules. The authors of OutsideIn(X) deemed this proof straightforward [6, Theorem 5.1, p. 44], but we shall explicitly formulate the proof for the adapted BindPA rule.

We need to prove from:

$$\Gamma \mapsto e : \upsilon \rightsquigarrow C \tag{1}$$

$$\underline{Q}\,;\underline{\tau} \twoheadrightarrow_a Q\,;\tau\,;\mathit{extra} \tag{2}$$

$$\mathcal{Q}\,;Q\,;\mathit{fuv}(\upsilon) \cup \mathit{fuv}(C) \cup\ \mathit{fuv}(\tau, Q) \overset{solv}{\mapsto} C \wedge \upsilon \sim \tau \rightsquigarrow Q_{res};\theta \tag{3}$$

$$\mathit{extra} \vee (Q_{res} = \epsilon) \tag{4}$$

$$\overline{b}\ \text{fresh} \tag{5}$$

$$\overline{\beta} = \mathit{fuv}(\theta\tau) \cup \mathit{fuv}(\theta Q \wedge Q_{res}) \tag{6}$$

$$\mathcal{Q}\,;\Gamma, (f : \forall\overline{a}\overline{b}\,.\,\overline{[\beta \mapsto b]}(\theta Q \wedge\ Q_{res} \Rightarrow \theta\tau)) \mapsto \mathit{prog} \tag{7}$$

$$\mathcal{Q}\,;\Gamma \mapsto f :: \forall\overline{a}\,.\,\underline{Q} \Rightarrow \underline{\tau} = e, \mathit{prog} \tag{8}$$

the following statements:

$$\underline{Q}\,;\underline{\tau} \twoheadrightarrow Q\,;\tau \tag{9}$$

$$Q_1\,;\Gamma \vdash e : \tau \tag{10}$$

$$\overline{a} \;\cup\; \overline{b} = \mathit{ftv}(Q) \cup \mathit{fuv}(\tau) \tag{11}$$

$$\mathcal{Q} \wedge Q \Vdash Q_1 \tag{12}$$

$$\mathcal{Q}\,;\Gamma, (f : \forall \overline{a}\overline{b}\,.\,Q \Rightarrow \tau) \vdash \mathit{prog} \tag{13}$$

$$\mathcal{Q}\,;\Gamma \vdash f :: \forall \overline{a}\,.\,\underline{Q} \Rightarrow \underline{\tau} = e, \mathit{prog} \tag{14}$$

First, we appeal to Lemma 2 with $[\overline{\beta \mapsto b}]\theta$ and Q_{res}, from (3) and (6) to attain (15) (note that $\theta Q_{res} = Q_{res}$, as $\mathit{dom}(\theta) \# \mathit{fuv}(Q_{res})$ [6, p. 20]). Lemma 2 imposes some conditions on the given Q_{res} and θ, namely $\mathit{dom}(\theta) \supseteq \mathit{fuv}(\tau) \cup \mathit{fuv}(Q)$ and $\mathit{extra} \vee (Q_{res} = \epsilon)$. The latter is satisfied by (4) and the former is satisfied because the solver will unify free unification variables from τ and Q, but not necessarily all free unification variables. The remaining free unification variables in $\theta\tau$ and θQ are handled by the additional substitution $[\overline{\beta \mapsto b}]$.

$$\underline{Q}\,;\underline{\tau} \twoheadrightarrow [\overline{\beta \mapsto b}]\theta(Q \wedge Q_{res})\,;[\overline{\beta \mapsto b}]\theta\tau \tag{15}$$

We then appeal to Lemma 1 as follows: $\Gamma \vdash\!\blacktriangleright e : \upsilon \rightsquigarrow C$ (1), then choose $C_{ext} = \upsilon \sim \tau$. Given (3), there exists Q_1 such that:

$$Q_1\,;\theta\Gamma \vdash e : \theta\upsilon \tag{16}$$

$$\mathcal{Q} \wedge \theta Q_g \wedge Q_r \vdash Q_1 \tag{17}$$

In the statements (9) to (14), choose for Q and τ respectively $[\overline{\beta \mapsto b}](\theta Q \wedge Q_{res})$ and $[\overline{\beta \mapsto b}]\theta\tau$ from the statements (1) to (8). Most statements from (9) to (14) are then directly proved. We now go over the remaining statements. (10) is proved because $\theta\Gamma = \Gamma$ as Γ is a closed top-level environment. For statement (13) we rely on induction.

Lemma 2. *Assume that* $\underline{Q}\,;\underline{\tau} \twoheadrightarrow_a Q\,;\tau\,;\mathrm{extra}$. *Then, for all* θ *and* Q_{res}, *if* $\mathit{dom}(\theta) \supseteq \mathit{fuv}(\tau) \cup \mathit{fuv}(Q)$, *and* $\mathrm{extra} \vee (Q_{res} = \epsilon)$, *then* $\underline{Q}\,;\underline{\tau} \twoheadrightarrow \theta Q \wedge Q_{res}\,;\theta\tau$.

Proof. By induction on the derivation of $\underline{Q}\,;\underline{\tau} \twoheadrightarrow_a Q\,;\tau\,;\mathit{extra}$. We consider cases corresponding to which rule was used.

- Case ANamedWc. We apply rule NamedWc. For proving $[\overline{\_a \mapsto \nu}]\underline{\tau} \twoheadrightarrow^t \theta\tau$ from $[\overline{\_a \mapsto \omega}]\underline{\tau} \twoheadrightarrow^t_a \tau$, we appeal to Lemma 3 as follows. We decorate the variable names corresponding with those from the lemma with a prime, the other variables were defined in this case. Let:
  - $\underline{\tau} = [\overline{\_a \mapsto \omega}]\underline{\tau}$.
  - $\tau' = \tau$
  - $\varphi' = [\overline{\omega \mapsto \nu}]$
  - $\theta' = \theta \setminus \mathit{dom}(\varphi')$.

  Lemma 3 then gives us $\varphi'\underline{\tau}' \twoheadrightarrow^t \theta'\varphi'\tau'$. $\varphi'\underline{\tau}' = [\overline{\omega \mapsto \nu}][\overline{\_a \mapsto \omega}]\underline{\tau} = [\overline{\_a \mapsto \nu}]\underline{\tau}$ and $\theta'\varphi'\tau' = \theta'[\overline{\omega \mapsto \nu}]\tau = \theta\tau$, as desired. Analogously for $[\overline{\_a \mapsto \nu}]\underline{Q}^w \twoheadrightarrow^c \theta Q$, appealing to Lemma 4, with $Q_{res} = \epsilon$.

– Case AExtraWc. Apply rule ExtraWc and use the induction hypothesis. Q_{res} need not be ϵ as $extra = \text{true}$.

Lemma 3. *Assume that* $\underline{\tau} \twoheadrightarrow_a^t \tau$. *Then, for all* θ *and* φ, *if* $dom(\theta) \supseteq fuv(\tau) \setminus fuv(\underline{\tau})$, $dom(\varphi) \supseteq fuv(\underline{\tau})$, *and* $dom(\theta) \cap dom(\varphi) = \emptyset$, *then* $\varphi\underline{\tau} \twoheadrightarrow^t \theta\varphi\tau$.

Proof. By induction on the derivation of $\underline{\tau} \twoheadrightarrow_a^t \tau$. We consider cases corresponding to which rule was used. The substitution φ has as domain the unification variables that take the place of named wildcards and substitution θ has as domain the unification variables that take the place of non-named wildcards.

– Case ATyWc. We have $\underline{\tau} = \_$, $\tau = \omega$, and want $\varphi\underline{\tau} \twoheadrightarrow^t \theta\varphi\tau$. $\varphi\underline{\tau} = \underline{\tau}$, as $fuv(\underline{\tau}) = \emptyset$, and $\theta\varphi\tau = \theta\varphi\omega$ results in the monotype τ, because $fuv(\tau) = \{\omega\} \subseteq dom(\theta)$. Thus, we can apply rule TyWc: $\underline{\tau} \twoheadrightarrow^t \tau$.
– Case ATyNoWc. We have that $\tau \twoheadrightarrow_a^t \tau$. As τ cannot contain any (free) unification variables, $\theta\varphi\tau = \varphi\tau$, in which case we can apply rule TyNoWc: $\varphi\tau \twoheadrightarrow^t \varphi\tau$.
– Case ATyApp. We have that $T\ \overline{\tau}_i \twoheadrightarrow_a^t T\ \overline{\tau}_i$. As substitution is distributive over the application of the type constructor T, we can use the induction hypothesis and apply rule TyApp.

Lemma 4. *Assume that* $Q^w \twoheadrightarrow_a^c Q$. *Then, for all* θ *and* φ, *if* $dom(\theta) \supseteq fuv(Q) \setminus fuv(\underline{Q}^w)$, $dom(\varphi) \supseteq fuv(\underline{Q}^w)$, *and* $dom(\theta) \cap dom(\varphi) = \emptyset$, *then* $\varphi\underline{Q}^w \twoheadrightarrow^c \theta\varphi Q$.

Proof. By induction on the derivation of $\underline{Q}^w \twoheadrightarrow_a^c Q$. We consider cases corresponding to which rule was used. The substitution φ has as domain the unification variables that take the place of named wildcards and substitution θ has as domain the unification variables that take the place of non-named wildcards.

– Case AConNoWc. We have that $Q \twoheadrightarrow_a^c Q$. As Q cannot contain any (free) unification variables, $\theta\varphi Q = \varphi Q$, in which case we can apply rule ConNoWc: $\varphi Q \twoheadrightarrow^c \varphi Q$.
– Case AConConj. We can use the induction hypothesis and apply rule ConConj.
– Case AConTc. Substitution is distributive over the application of the typeclass constraint D. We can apply rule ConTc using Lemma 3.
– Case AConEq. Substitution is distributive over an equality constraint. We can apply rule ConEq using Lemma 3.

C Proof for Theorem 2

Proof. Induction on the size of *prog*, using Lemma 5 for definitions with type signatures without wildcards, and Lemma 6 for definitions without a type signature.

Lemma 5. *For a definition* f *with a type signature without wildcards,* BindPA *(Figure 6) is equivalent with* BindA *(Figure 5).*

Proof. We shall prove that when f has a type signature without wildcards, the premise as well as the conclusion of BindPA are equivalent with those of BindA. This implies that the rules are equivalent in the case of a type signature without wildcards.

The statements in the premise of BindA are:

$$\Gamma \vdash\!\blacktriangleright e : \upsilon \rightsquigarrow C \tag{1}$$

$$\mathcal{Q}\,;Q\,;\mathit{fuv}(\upsilon) \cup \mathit{fuv}(C) \overset{solv}{\vdash\!\blacktriangleright} C \wedge \upsilon \sim \tau \rightsquigarrow \epsilon\,;\theta \tag{2}$$

$$\mathcal{Q}\,;\Gamma, (f : \forall \overline{a}\,.\,Q \Rightarrow \tau) \vdash\!\blacktriangleright \mathit{prog} \tag{3}$$

From these statements, the conclusion $\mathcal{Q}\,;\Gamma \vdash\!\blacktriangleright f :: \forall \overline{a}\,.\,Q \Rightarrow \tau = e, \mathit{prog}$ can be proved.

We will begin by proving the equivalence of BindA's premise with BindPA's premise.

BindA and BindPA will both infer the same υ and C in the expression $\Gamma \vdash\!\blacktriangleright e : \upsilon \rightsquigarrow C$, as the judgment is applied with identical input parameters. Next, the wildcard desugaring judgment is applied:

$$Q\,;\tau \twoheadrightarrow_a Q\,;\tau\,;\text{false} \tag{4}$$

Given that Q nor τ contain wildcards, the output parameters of the judgment will be the same as the input parameters, and *extra* will be false.

It is impossible for Q and τ to have free unification variables, as they do not contain wildcards. Thus:

$$\mathit{fuv}(\tau) \cup \mathit{fuv}(Q) = \emptyset \tag{5}$$

The call to the constraint solver happens with the same input parameters as the call in (2), resulting in the same output, except for the residual constraints, which are captured in Q_{res} instead of forced to be ϵ. However, as $\mathit{extra} = \text{false}$ (4), $Q_{res} = \epsilon$ must be true, which makes the output parameters of both calls to the solver identical after all.

As there are no free unification variables in τ and Q (5), the substitution θ will affect Q (6) nor τ, (7).

$$\theta Q = Q \tag{6}$$

$$\theta \tau = \tau \tag{7}$$

$$\overline{\beta} = \mathit{fuv}(\theta\tau) \cup \mathit{fuv}(\theta Q \wedge Q_{res}) = \mathit{fuv}(\tau) \cup \mathit{fuv}(Q \wedge \epsilon) = \emptyset \tag{8}$$

Both $\overline{\beta}$ and $\overline{b}$ will be empty, combined with $Q_{res} = \epsilon$, (6), and (7) results in the following type for f:

$$f : \forall \overline{a}\overline{b}\,.\,[\overline{\beta \mapsto b}](\theta Q \wedge Q_{res} \Rightarrow \theta\tau) \tag{9}$$

$$f : \forall \overline{a}\,.\,Q \Rightarrow \tau \tag{10}$$

This is the same type as the one in (3), thus, we have now proved the equivalence of the premises.

The conclusions of both rules are also equivalent as the difference between the rules lies in the fact that BindPA's conclusion allows for wildcards, of which we have said that there are none. Thus, the $\underline{Q}$ and $\underline{\tau}$ in BindPA's conclusion

can simply be replaced with Q and τ, which makes it equivalent with BINDA's conclusion.

Lemma 6. *For a definition f without a type signature,* ALTBIND *(Figure 6) is equivalent with* BIND *(Figure 5).*

Proof. When f has no type signature, the rule ALTBIND applies, which requires a proof of its premise $\mathcal{Q}\,;\Gamma \vdash_{\blacktriangleright} f :: \_ \to \_ = e, prog$. To prove this statement, we apply the BINDPA rule, of which the premise consists of the following statements:

$$\Gamma \vdash_{\blacktriangleright} e : v \rightsquigarrow C \tag{1}$$

$$\underline{Q}\,;\underline{\tau} \twoheadrightarrow_a Q\,;\tau\,;\mathit{extra} \tag{2}$$

$$\mathcal{Q}\,;Q\,;\mathit{fuv}(v) \cup \mathit{fuv}(C) \cup \mathit{fuv}(\tau) \cup \mathit{fuv}(Q) \vdash^{solv}_{\blacktriangleright} C \wedge v \sim \tau \rightsquigarrow Q_{res};\theta \tag{3}$$

$$\mathit{extra} \vee Q_{res} = \epsilon \tag{4}$$

$$\overline{b} \text{ fresh} \tag{5}$$

$$\overline{\beta} = \mathit{fuv}(\theta\tau) \cup \mathit{fuv}(\theta Q \wedge Q_{res}) \tag{6}$$

$$\mathcal{Q}\,;\Gamma, (f : \forall \overline{a}\overline{b}\,.\,[\overline{\beta \mapsto b}](\theta Q \wedge Q_{res} \Rightarrow \theta\tau)) \vdash_{\blacktriangleright} prog \tag{7}$$

We shall now prove that for $\underline{Q} = \_$ and $\underline{\tau} = \_$ the premise of BIND is equivalent with the premise of BINDPA. By consequence, BIND will be equivalent with ALTBIND for the same values of $\underline{Q}$ and $\underline{\tau}$.

The premise of BIND consists of the following statements:

$$\Gamma \vdash_{\blacktriangleright} e : \tau \rightsquigarrow C \tag{8}$$

$$\mathcal{Q};\epsilon;\mathit{fuv}(\tau) \cup \mathit{fuv}(C) \vdash^{solv}_{\blacktriangleright} C \rightsquigarrow Q\,;\theta \tag{9}$$

$$\overline{a} \text{ fresh} \tag{10}$$

$$\overline{\alpha} = \mathit{fuv}(\theta\tau) \cup \mathit{fuv}(Q) \tag{11}$$

$$\mathcal{Q}\,;\Gamma, (f : \forall \overline{a}\,.\,[\overline{\alpha \mapsto a}](Q \Rightarrow \theta\tau)) \vdash_{\blacktriangleright} prog \tag{12}$$

It is clear that (8) and (1) are equivalent. The rule (2) will be applied with $\underline{Q} = \_$ and $\underline{\tau} = \_$. The following rules will be applied by the wildcard desugaring judgment: AEXTRAWC ($\underline{Q} = \_ = \epsilon \wedge \_$), which results in $\mathit{extra} = \text{true}$, ANAMEDWC, which only applies ATYWC and ACONNOWC, as there are no named wildcards present in $\underline{Q}^w$ and $\underline{\tau}$. ATYWC will replace the type wildcard with a fresh unification variable ω. ACONNOWC is applied because $\underline{Q}^w = \epsilon$. This results in the following expression:

$$\underline{Q}\,;\underline{\tau} \twoheadrightarrow_a \epsilon\,;\omega\,;\text{true} \tag{13}$$

In the next step, the constraint solver is called (3). We shall prove that the call (3) happens with input parameters equivalent with those from the call in (9), which will result in equivalent output parameters. The first parameter, $\mathcal{Q}$, will be identical for both calls. The second parameter in (9) is ϵ, just like in (3), see (13). The third parameter differs, the $\mathit{fuv}(\tau) \cup \mathit{fuv}(C)$ from (9) will be identical to the $\mathit{fuv}(v) \cup \mathit{fuv}(C)$ from (3), but to the latter $\mathit{fuv}(\tau) \cup \mathit{fuv}(Q)$ will be added, namely $\{\omega\}$. The fourth parameters also differ slightly; in (3) there is an extra constraint: $v \sim \tau (= \omega)$. These two differing input parameters will only result

in an extra substitution, namely $\omega \mapsto \upsilon$. Thus, the substitution θ from (9) is equivalent to the one from (3). Rule (4) will not require $Q_{res} = \epsilon$ given that $extra = \text{true}$ (13).

We shall now prove that the $\overline{\alpha}$ (11) are identical to the $\overline{\beta}$ from (6). To avoid confusion between variables from (11) and (6), we shall suffix variables from (11) with an α-subscript. Given that $\tau = \omega$, $\theta = [\omega \mapsto \upsilon, \theta_\alpha]$, and $\upsilon = \tau_\alpha$, (14) is true. (15) is true because both Q_{res} and Q_α are output parameters of equivalent invocation of the constraints solver. From (14) and (15) follows (16).

$$\theta\tau = \theta\omega = [\omega \mapsto \upsilon, \theta_\alpha]\omega = \theta_\alpha\upsilon = \theta_\alpha\tau_\alpha \tag{14}$$

$$Q_{res} = Q_\alpha \tag{15}$$

$$\overline{\alpha} = fuv(\theta_\alpha\tau_\alpha) \cup fuv(Q_\alpha) = fuv(\theta\tau) \cup fuv(\epsilon \wedge Q_{res}) = \overline{\beta} \tag{16}$$

The final type in (7) will be the following:

$$f : \forall \overline{a}\overline{b} \,.\, [\overline{\beta \mapsto b}](\theta Q \wedge Q_{res} \Rightarrow \theta\tau) \tag{17}$$

$$f : \forall \overline{b} \,.\, [\overline{\beta \mapsto b}](\theta Q \wedge Q_{res} \Rightarrow \theta\tau) \tag{18}$$

$$f : \forall \overline{b} \,.\, [\overline{\beta \mapsto b}](Q_{res} \Rightarrow \theta\tau) \tag{19}$$

$$f : \forall \overline{b} \,.\, [\overline{\beta \mapsto b}](Q_\alpha \Rightarrow \theta\tau) \tag{20}$$

$$f : \forall \overline{b} \,.\, [\overline{\beta \mapsto b}](Q_\alpha \Rightarrow \theta_\alpha\tau_\alpha) \tag{21}$$

$$f : \forall \overline{a} \,.\, [\overline{\alpha \mapsto a}](Q_\alpha \Rightarrow \theta_\alpha\tau_\alpha) \tag{22}$$

In (18), the $\overline{a}$ disappear because the annotated type $(\_ \Rightarrow \_)$ did not contain any type variables. In the next line (19), θQ disappears because it is empty (13). In (20) Q_{res} is replaced by Q_α because of (15). After this step, $\theta\tau$ is replaced by $\theta_\alpha\tau_\alpha$ because of (14). As $\overline{\alpha} = \overline{\beta}$ (16), we can replace $\overline{\beta}$ in (22) by $\overline{\alpha}$, where $\overline{a}$ and $\overline{b}$ are fresh type variables. This is the same type as in (12).

We have now proved that for a definition f without a type signature, Alt-Bind, which uses BindPA, is equivalent with Bind.