Types Inference for Generic Haskell

Alexey Rodriguez, Johan Jeuring, and Andres Löh

Institute of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht, the Netherlands

Abstract. The more expressive a type system, the more type information has to be provided in a program. Having to provide a type is sometimes a pain, but lacking expressivity is often even worse. There is a continuous struggle between expressivity and (type-)verbosity. However, even very expressive type systems allow type inference for parts of a program. Generic Haskell is an extension of Haskell that supports defining generic functions. Generic Haskell assumes that the type of a generic function is explicitly specified. This is often no problem, but sometimes it is rather painful to have to specify a type – in particular for generic functions with many dependencies – and sometimes the specified type can be generalized. In this paper, we identify three type inference problems specific to generic functions, and present (partial) solutions to each of them.

1 Introduction

A generic function can be used on a large set of data types. Generic functions are useful, amongst others, for implementing functions that have to be written over and again for different data types. Examples are equality, reading and showing functions, traversal functions over large structures, and programs on data types that frequently change [12, 20].

There exist several approaches to generic programming [1, 15, 20, 9, 22, 11, 2]. Generic Haskell [9, 22, 21] is one of the more powerful (but also elaborate) approaches to generic programming. It is an extension of Haskell with amongst others type-indexed functions which have kind-indexed types, and type-indexed data types which have kind-indexed kinds.

A generic function in Generic Haskell is defined by induction on the structure of types. Generic Haskell translates a generic function into a number of components. The application of a generic function is then translated into an application of these components. During the translation process, the compiler makes use of type information for generic functions. Until now, work on Generic Haskell has assumed that this type information is provided explicitly by the programmer and merely checked by the compiler.

In some cases, providing the type of a generic function can be quite a pain. In other cases the provided type might be more specific than necessary. This

paper investigates which type information can be inferred instead of checked by the compiler. We introduce three different type inference problems typical for generic programming, and discuss (im)possibilities of type inference for these problems. The question of how much type information can be inferred for languages with advanced type systems has received quite some attention in the last decade: partial type-inference algorithms have been designed for System F [3], the lambda calculus with arbitrary-rank polymorphism [18, 25], Cyclone [16], intersection type systems [5], a system with generalized algebraic data types [27, 19, 24], to mention just a few. This paper investigates the same problem for a typed generic programming language.

The main results of this paper are:

- We distinguish three different type-inference problems for generic functions.
- We discuss (im)possibilities of type inference for generic functions, and give examples which show that complete type inference is impossible, and give algorithms for type inference for restricted classes of generic functions. We can infer the type of about half of the functions of Generic Haskell's library, and we argue why it is unlikely that the type of the other half of the library can be inferred.
- We introduce a special kind of qualified types to infer generic types.

We use the core language of Generic Haskell to illustrate the ideas, and Generic Haskell's type system to explain type inference, but the results also apply to similar approaches to generic programming, such as PolyP [15], derivable type classes [14], and Clean's generic-programming extension [1]. Furthermore, we expect that the qualified-types approach we use might have applications in other areas, but we have yet to investigate the relation between type inference for generic functions and areas such as generalized algebraic data types and intersection types.

Organization. This paper is organized as follows: in the next section, we introduce Generic Haskell, with the focus on the types of type-indexed functions. Then, in Section 3, we precisely formulate the three inference problems for "type arguments", "dependencies", and "base types", and describe how to solve the first two of these. Section 4 is devoted to base-type inference. In Section 5 we conclude and discuss related and future work.

2 Type-indexed functions and their types

A generic function can be used on a large set of data types. A generic function is a type-indexed function that is defined on a collection of types that can be used to represent the structure type of (almost) any data type. Since structure types only appear in the translation of generic functions, and not in the type system for Generic Haskell, we will use the term type-indexed function in the rest of this paper.

This section introduces type-indexed functions and their types. It discusses how the type of a type-indexed function is used in the translation of the type-indexed function.

2.1 Type-indexed functions

A type-indexed function consists of a family of function definitions, and each of the function definitions contains a type pattern. An example is the definition of equality:

$$
\begin{array}{lllll}
eq\ \langle \mathrm{Int} \rangle & x & y & = x == y & \text{-- use built-in equality} \\
eq\ \langle \mathrm{Bool} \rangle & True & True & = True & \\
eq\ \langle \mathrm{Bool} \rangle & False & False & = True & \\
eq\ \langle \mathrm{Bool} \rangle & \_ & \_ & = False & \\
eq\ \langle [\alpha] \rangle & xs & ys & & \\
\quad \mid eq\ \langle \mathrm{Int} \rangle\ (length\ xs)\ (length\ ys) & & & = and\ (zipWith\ (eq\ \langle \alpha \rangle)\ xs\ ys) & \\
\quad \mid otherwise & & & = False & \\
eq\ \langle (\alpha, \beta) \rangle\ (x_1, x_2)\ (y_1, y_2) & & & = eq\ \langle \alpha \rangle\ x_1\ y_1 \wedge eq\ \langle \beta \rangle\ x_2\ y_2 &
\end{array}
$$

This function is defined on integers, booleans, lists, and pairs, and defines the standard structural equality on these types. Function *eq* can be used on combinations of these three type constructors by supplying a type argument, for instance, $eq\ \langle ([\mathrm{Bool}], \mathrm{Bool}) \rangle$ can be used to test two pairs, each consisting of a list of booleans and a single boolean, for equality.

In the definition of *eq* we have used several syntactic extensions of the core language of Generic Haskell, which contains a typecase construct for defining type-indexed functions. The core language is presented in Figure 1.

The expression language is standard, and includes variables, constructors, applications, abstractions and let bindings. In addition to these constructs, we have type application (i.e., the application of a type-indexed function to a type) and definitions of type-indexed functions. A type is a type variable, a type constructor (decorated by its arity) or a type pattern variable (pattern variable for short). For simplicity, we do not introduce kinds in the language, and assume that all type variables have kind $\star$. It follows that we cannot handle parametrized type patterns with higher-kinded pattern types. However, the results of this paper can be applied in a straightforward way to a type system with kinds [23].

A type pattern is an application of a type constructor to (possibly zero) pattern variables. We assume that type patterns are linear, so that no pattern variable occurs more than once. A type argument is a pattern variable, or an application of a type constructor to type arguments. A type environment Γ binds possibly type-indexed functions to type schemes. It may also introduce type pattern variables explicitly (Γ, α) or in type-indexed function bindings $(\Gamma, x\ \langle \alpha \rangle :: \sigma)$, but only if not already bound in environment Γ. This condition appears in a well-formedness judgement for environments, which has been omitted.

We write $\forall \overline{a_i}\,.\,\sigma$ instead of the more verbose $\forall a_1\,.\,\forall a_2\,.\,\ldots\sigma$, for a type scheme abstracting over a number of type variables, and similarly for other expressions where multiple arguments are used. We use the standard definitions for function *ftv*, which returns the free type variables, and a similar function *fpv*, which returns the free pattern variables of a type. Function *dpv*, for defined pattern variables, returns the pattern variables explicitly introduced in an environment Γ. In type rules we write $A \# B$ to express that sets A and B are disjoint; for a set

of variables, we write $\{\overline{a_i}\}$. Generalization of types $Gen(X;t)$ is a shorthand for $\forall\overline{a_i}\,.\,t$ such that $\{\overline{a_i}\}\#ftv\ (X)$.

Expressions	e	$::= x \mid C \mid e_1\ e_2 \mid \lambda x \to e \mid \mathbf{let}\ x = e\ \mathbf{in}\ e' \mid x\ \langle A\rangle$ $\mid\ \mathbf{let}\ x\ \langle\alpha\rangle = \mathbf{typecase}\ \alpha\ \mathbf{of}\ R\ \mathbf{in}\ e'$
Arms	R	$::= \emptyset \mid \Omega \to e, R$
Types	t, u	$::= a, b, \ldots \mid T^n\ (\overline{t_i}) \mid \alpha, \beta, \ldots$
Type patterns	Ω	$::= T^n\ \overline{\alpha_i}$
Type arguments	A	$::= \alpha \mid T^n\ \overline{A_i}$
Type schemes	σ	$::= t \mid \forall a\ .\sigma$
Environments	Γ	$::= \emptyset \mid \Gamma, x :: \sigma \mid \Gamma, x\ \langle\alpha\rangle :: \sigma \mid \Gamma, \alpha$

Fig. 1. Expression and type syntax for core Generic Haskell

2.2 Translation of type-indexed functions

The Generic Haskell compiler translates a type-indexed function by generating a specialized function for each of the type patterns. For the Int and Bool cases, this is trivial. For example, the *component* of equality on integers is

$\mathsf{cp}(eq, \mathrm{Int})\ x\ y = x == y$ -- use built-in equality

The generation of components is interesting if applications of type-indexed functions appear on the right-hand side. The component for lists is

$$\begin{array}{ll}\mathsf{cp}(eq,[\,])\ \mathsf{cp}(eq,\alpha)\ xs\ ys & \\ \quad \mid \mathsf{cp}(eq,\mathrm{Int})\ (length\ xs)\ (length\ ys) & = and\ (zipWith\ (\mathsf{cp}(eq,\alpha))\ xs\ ys) \\ \quad \mid otherwise & = False\end{array}$$

As can be observed, the applications of type-indexed functions are translated into applications of components, but in addition, components of type-indexed functions on variables of the type pattern appear as extra arguments of a component. In this case, the component for equality on lists $\mathsf{cp}(eq,[\,])$ takes an equality function on the element type of the list $\mathsf{cp}(eq,\alpha)$ as argument. This argument is used on the right-hand side in the translation of the call $eq\ \langle\alpha\rangle$, but not in the translation of the call $eq\ \langle\mathrm{Int}\rangle$, which has a constant type argument. However, we would translate a call to $eq\ \langle[\mathrm{Int}]\rangle$ as an application of components, supplying $\mathsf{cp}(eq,\mathrm{Int})$ as the argument for $\mathsf{cp}(eq,[\,])$.

The function eq has a *dependency* on eq, because the component $\mathsf{cp}(eq,\alpha)$ is required to define $\mathsf{cp}(eq,[\,])$. A type-indexed function is usually recursive and therefore depends on itself, but a function can call other type-indexed functions as well, and thus have multiple dependencies [22, 21].

2.3 Generic type signatures

In Generic Haskell, type signatures of type-indexed functions are called *generic type signatures*. A generic type signature for a function f takes a special form, containing enough information to both

- indicate that f is indeed a type-indexed function,
- allow the compiler to instantiate the type of f for any type application $f\ \langle A \rangle$ of the function f to some type argument A; in particular, the type of $\mathsf{cp}(f, T)$ can be derived from the type signature for any named type T.

To continue our running example, the generic type signature of *eq* is

$$eq\ \underbrace{\langle \alpha \rangle}_{\text{type arguments}} :: \underbrace{(eq\ \langle \alpha \rangle)}_{\text{dependencies}} \Rightarrow \underbrace{\alpha \to \alpha \to \mathrm{Bool}}_{\text{base type}}$$

As indicated above, there are three important parts to the type signature:

- one or more *type arguments*, enclosed by $\langle\rangle$, indicate that *eq* is a type-indexed function,
- there are *dependencies* if the function calls itself or other type-indexed functions recursively,
- the *base type* gives the type of a variable component such as $\mathsf{cp}(eq, \alpha)$.

From this type signature, the following *kind-indexed type* [10] can be generated:

$$\begin{array}{ll} \mathsf{kit}(eq, \star) & a = a \to a \to \mathrm{Bool} \\ \mathsf{kit}(eq, \kappa_1 \to \kappa_2)\ a & = \forall a' \,.\, \mathsf{kit}(eq, \kappa_1)\ a' \to \mathsf{kit}(eq, \kappa_2)\ (a\ a') \end{array}$$

With this kind-indexed type, we have that $\mathsf{cp}(eq, T) :: \mathsf{kit}(eq, \kappa)\ T$ for each type constructor T of kind κ. In particular, we get the following types for the components of the *eq* function:

$$\begin{array}{ll} \mathsf{cp}(eq, \mathrm{Int}) & :: \mathrm{Int} \to \mathrm{Int} \to \mathrm{Bool} \\ \mathsf{cp}(eq, \mathrm{Bool}) & :: \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool} \\ \mathsf{cp}(eq, [\,]) & :: \forall a \,.\, (a \to a \to \mathrm{Bool}) \to [a] \to [a] \to \mathrm{Bool} \\ \mathsf{cp}(eq, (,)) & :: \forall a\ b \,.\, (a \to a \to \mathrm{Bool}) \to (b \to b \to \mathrm{Bool}) \to \\ & \qquad\qquad (a, b) \to (a, b) \to \mathrm{Bool} \end{array}$$

In general, the situation is slightly more complex than for *eq*. The kind-indexed type for *eq* takes a single type argument, and therefore the *arity* of *eq* is 1. For some type-indexed functions the kind-indexed type takes more than one type argument, examples are *map* (arity 2) and *zipWith* (arity 3). In Section 4 we argue that it is impossible to infer the arity of a type-indexed function, and that type inference is only possible for a fixed arity. Since the majority of type-indexed functions has arity 1, we will assume that type-indexed functions have arity 1 in the rest of this paper.

2.4 A type system for core Generic Haskell

This section presents a type system for core Generic Haskell, or GH for short. Since we are not going to use dependencies in the typing rules in the following sections, rules for dependencies are omitted. The type checking rules for core

$$\frac{x\ \langle\alpha\rangle :: \sigma \in \Gamma \qquad fpv\ (A) \subseteq dpv\ (\Gamma)}{\Gamma \vdash x\ \langle A\rangle :: \sigma[A/\alpha]}\ \text{(gapp)} \qquad \frac{\overline{\Gamma, x\ \langle\alpha\rangle :: t', dpv\ (\Omega_i) \vdash e_i :: t'[\Omega_i/\alpha]} \qquad \{\overline{a_j}\}\#ftv\ (\Gamma) \qquad \Gamma, x\ \langle\alpha\rangle :: \forall\overline{a_j}\,.\,t' \vdash e :: t}{\Gamma \vdash \textbf{let}\ x\ \langle\alpha\rangle = \textbf{typecase}\ \alpha\ \textbf{of}\ \overline{\Omega_i \to e_i}\ \textbf{in}\ e :: t}\ \text{(glet)}$$

Fig. 2. Type checking rules for Generic Haskell

Generic Haskell are presented in Figure 2, in which judgements of the form $\Gamma \vdash e :: \sigma$ express that expression e has type σ in environment Γ. We only present the non-standard rules; the rules for variables, λ-abstraction, application, $\forall$-introduction and $\forall$-elimination, and **let** are omitted.

The type application rule (gapp), adapted from [21], returns the specialized type of a type-indexed function. The rule checks that all pattern variables occurring in a type argument are defined.

The type-indexed let rule (glet) first checks that each arm types with the base type specialized to the corresponding type pattern. The arms may use the function recursively, be it in a non-polymorphic way. Then it uses the generalized type to check the type of the expression e.

In his thesis [21], Löh shows how to translate type-indexed functions and types to plain Haskell functions and types, and that type-correct programs in Generic Haskell are translated to type-correct programs in Haskell.

3 Type inference problems

We can identify three type inference problems for type-indexed functions. Since type-indexed functions cannot be defined in plain Haskell, these problems cannot be formulated for Haskell.

The first type inference problem is about inferring type arguments. If we use the type-indexed function *eq* on expressions x_1 and x_2 of type Bool, we have to write $eq\ \langle\text{Bool}\rangle\ x_1\ x_2$. We would rather write $eq\ x_1\ x_2$, and infer from the types of *eq*, x_1 and x_2 that it is the instance of *eq* on Bool we mean.

Problem 1 (Type-argument inference). Given a type application $f\ \langle A\rangle$ of a type-indexed function f to a type argument A, infer the type argument A. In other words, use contextual information to determine A so that the programmer can simply write f instead of $f\ \langle A\rangle$.

As explained in Section 2.2, function *eq* depends on itself. This information has to be provided explicitly in the type of *eq*. This dependency can be inferred from the arm for $\langle[\alpha]\rangle$ in its definition.

Problem 2 (Dependency inference). Given the definition of a type-indexed function and a partial type signature without dependencies, infer the dependencies to complete the type signature.

Finally, in some circumstances even the base type for type-indexed functions can be inferred.

Problem 3 (Base-type inference). Given the definition of a type-indexed function and a partial type signature without base type, infer the base type to complete the type signature.

The first two problems are relatively easy to solve. Problem 1 can be reduced to type inference for overloaded functions in Haskell. If a type-indexed function f is applied to a constant type argument A, the problem of inferring the A is equivalent to finding the correct instance of a class C_f which has f as a method and one instance for each arm of the typecase. This reduction is described in more detail in the technical report [23] accompanying this paper.

Problem 2 can be solved by scanning each type-indexed function's body for calls to other type-indexed functions. This induces direct dependencies between type-indexed functions. Because the dependency relation has to be reflexive and transitive [21], the reflexive transitive closure has to be computed. For functions of arity 1, this technique completely solves the problem of dependency inference. A more detailed discussion is again included in the technical report.

Problem 3 is by far the most difficult and most interesting of the three problems. Therefore, the rest of this paper is devoted to the problem of base-type inference.

4 Base-type inference

Base-type inference amounts to computing a type t for a term e, such that under an environment Γ, $\Gamma \vdash e :: t$, where $\vdash$ is the typing relation for type-indexed functions defined in Section 2.4.

We would like type-indexed functions to have principal types [7]. If type-indexed functions have principal types, we can design a compositional type-inference algorithm that gives the best possible type for an expression, and we can use standard techniques to prove its correctness.

Unfortunately, there exist type-indexed functions that may be assigned two incomparable types [21]. An example of such a function is the function which equals the type-indexed identity function (*gid*) with one type and the type-indexed map function (*map*) with the other type. The type of the type-indexed map function cannot be used in all contexts where the type of the type-indexed identity is used, and conversely. Since we cannot give the function a single type that subsumes the two others, the Generic Haskell type system lacks the principal types property.

A natural approach to this problem is to infer types only for a fragment of the Generic Haskell type system. Based on the observation that the type-indexed identity and map functions have different arities, we could restrict type inference to infer types of type-indexed functions with a fixed arity, for example the arity 1.

Unfortunately, even in this restricted setting we can find functions with no principal type. Consider the definition of *fun*

$$\textbf{let } \mathit{fun}\ \langle\alpha\rangle = \textbf{typecase } \alpha \textbf{ of } \begin{array}{ll} \text{Bool} & \to \lambda x \to \mathit{not}\ x \\ \text{Int} & \to \lambda x \to x \end{array}$$

We can assign two different types to *fun*: $\mathit{fun}\ \langle\alpha\rangle :: \text{Bool} \to \text{Bool}$ and $\mathit{fun}\ \langle\alpha\rangle :: \alpha \to \alpha$. These types are not comparable, and there does not exist a type subsuming these two types in Generic Haskell's type system.

We solve the problem by adding qualified types [17] to the Generic Haskell type system. The above function's type now becomes $\mathit{fun}\ \langle\alpha\rangle :: \forall a\,.\,P(a) \Rightarrow a \to a$. This type represents the two types above through appropriate instantiations of type variable a such that they satisfy the predicate P.

Using qualified types we can assign a type to an expression that represents all its possible types. We shall now introduce the predicates used in the extension of the type system with qualified types.

4.1 Qualified types for type-indexed functions

Function *fun* lacks a principal type because the substitution in rule (glet), Figure 2, is a non-injective function from base types to arm types. It follows that several incomparable base types may type the arm of a type-indexed function, or even a type-indexed function.

Our type system represents all the types of a type-indexed function with qualified types. Types are qualified with disjunctions of type equations called choice predicates. For example, the type of function *fun* is: $\mathit{fun}\ \langle\alpha\rangle :: \forall a\,.\,a = \alpha \vee a = \text{Bool} \Rightarrow a \to a$. From it, we can derive the two types for *fun* using a generic instance [7, 17] relation. Specifically, we substitute either α or Bool for variable a; these are the only substitutions that satisfy the equations at either side of the disjunction.

Each arm in the definition of a type-indexed function gives rise to a qualified type. This process, which we call marking, may be seen as the inverse of (glet) substitution: we obtain the base type from the arm type. More precisely, marking obtains a qualified (base) type q from the arm type t replacing all occurrences of the type index Ω by constrained type variables; formally we write $\mathit{mark}^{\alpha}_{\Omega}\Gamma; t \equiv q$. A choice predicate constrains a type variable to be either a type argument (α) or the original type. Figure 3 gives a specification for marking.

Marking and generalizing $\mathit{fun}\ \langle\text{Bool}\rangle :: \text{Bool} \to \text{Bool}$ produces:

$$\mathit{fun}\ \langle\alpha\rangle :: \forall b\ c\,.\,(b = \alpha \vee b = \text{Bool}, c = \alpha \vee c = \text{Bool}) :: b \to c$$

where every Bool in the original type is replaced by potentially different constrained variables. Indeed, this type scheme subsumes all 4 possible base types including $\mathit{fun}\ \langle\alpha\rangle :: \alpha \to \alpha$ and $\mathit{fun}\ \langle\alpha\rangle :: \text{Bool} \to \text{Bool}$.

Generalized marked types have the property that all generic instances, when specialized to the type index, are also generic instances of the generalized type of the arm. That is, for all σ such that $\mathit{mark}^{\alpha}_{\Omega}\Gamma; t \equiv q$ and $\sigma \leqslant \mathit{Gen}(\Gamma; q)$, we have that $\sigma[\Omega/\alpha] \leqslant \mathit{Gen}(\Gamma; t)$. It follows that all specialized instance types can also type the arm.

$$\frac{a \notin \Gamma}{mark^{\alpha}_{T}\Gamma;\, T \equiv a = \alpha \vee a = T \Rightarrow a} \text{ (mark-const)}$$

$$\frac{a \notin \Gamma}{mark^{\alpha}_{T}\Gamma;\, b \equiv a = \alpha \wedge b = T \vee a = b \Rightarrow a} \text{ (mark-var)}$$

$$\frac{\overline{mark^{\alpha}_{T}\Gamma;\, t_i \equiv P_i \Rightarrow t'_i} \qquad T \neq T'}{mark^{\alpha}_{T}\Gamma;\, T'\ \overline{t_i} \equiv \overline{P_i} \Rightarrow T'\ \overline{t'_i}} \text{ (mark-app)}$$

Fig. 3. Marking: Introduction of predicates

Marking a type variable deserves special attention. The resulting choice predicate mentions the variable in an additional type equation; thus imposing a restriction when the type argument (left) choice is taken. The additional type equation ensures that the property just stated above is enforced for variable cases. To witness, we mark and generalize *fun* $\langle \text{Int} \rangle :: a \to a$:

$$\begin{aligned} \mathit{fun}\ \langle\alpha\rangle :: \forall b\ c\ a\,.\,(&b = \alpha \wedge a = \text{Int} \vee b = a \\ &, c = \alpha \wedge a = \text{Int} \vee c = a) \Rightarrow b \to c \end{aligned}$$

A generalized type without the additional equations would admit the generic instance *fun* $\langle\alpha\rangle :: \forall a\,.\,\alpha \to a$. The instance type, specialized, breaks the property *fun* $\langle \text{Int} \rangle :: \forall a\,.\,\text{Int} \to a \leqslant \mathit{fun}\ \langle \text{Int} \rangle :: \forall a\,.\,a \to a$, making it unsuitable to type the Int arm of the function. The additional equations prevent this typing.

Base types for a type-indexed function should be generic instances of all generalized marked types. Combining the generalized marked types for Bool and Int for *fun* we thus get

$$\begin{aligned} \mathit{fun}\ \langle\alpha\rangle :: \forall b\ c\ a\,.\,(&b = \alpha \vee b = \text{Bool}, c = \alpha \vee c = \text{Bool} \\ &, b = \alpha \wedge a = \text{Int} \vee b = a, c = \alpha \wedge a = \text{Int} \vee c = a) \\ &\Rightarrow b \to c \end{aligned}$$

which represents the same set of types as *fun* $\langle\alpha\rangle :: \forall a\,.\,a = \alpha \vee a = \text{Bool} \Rightarrow a \to a$. However, our type system cannot prove their equality. We provided the latter type at the start of this section for readability.

A well-formedness condition, appearing in our type rules, requires that type variables constrained by several predicates, such as b and c above, be constrained by a single predicate. Hence, the type of *fun* changes to:

$$\begin{aligned} \mathit{fun}\ \langle\alpha\rangle :: \forall b\ c\ a\,.\,(&b = \alpha \wedge a = \text{Int} \vee b = a \wedge b = \text{Bool} \\ &, c = \alpha \wedge a = \text{Int} \vee c = a \wedge c = \text{Bool}) \Rightarrow b \to c \end{aligned}$$

ensuring a consistent choice for the constrained variables.

Otherwise, some untypable programs in the Generic Haskell type system become typable in GQH. For instance, consider the following program

$$\begin{aligned} &\textbf{let}\ f\ x = \\ &\quad \textbf{let}\ \mathit{val}\ \langle\alpha\rangle = \textbf{typecase}\ \alpha\ \textbf{of} \end{aligned}$$

$$
\begin{array}{l}
\qquad\qquad\qquad\quad \text{Int} \quad \rightarrow x \\
\qquad\qquad\qquad\quad \text{Bool} \rightarrow \mathit{True} \\
\qquad\qquad\qquad\quad \text{Float} \rightarrow x \\
\quad \textbf{in } \mathit{val}\ \langle \text{Int} \rangle \\
\textbf{in } \mathit{plus}\ (f\ 2)\ 2
\end{array}
$$

The GH type system assigns the type Bool to $\mathit{val}\ \langle\alpha\rangle$. It follows that x has type Bool. The typing $\mathit{val}\ \langle\alpha\rangle :: \alpha$ is wrong because it makes one of the Int or Float arms ill-typed, depending on the type of x.

Consider the situation when *val* type checks with the following type, assuming that x has type b in the environment.

$$
\begin{array}{l}
\mathit{val}\ \langle\alpha\rangle :: \forall a\,.\,(a = \alpha \wedge b = \text{Int} \quad \vee a = b \\
\qquad\qquad\qquad\quad , a = \alpha \qquad\qquad\quad\ \ \vee a = \text{Bool} \\
\qquad\qquad\qquad\quad , a = \alpha \wedge b = \text{Float} \vee a = b) \Rightarrow a
\end{array}
$$

Substituting Int for α, this leads to the following type for f:

$$
\begin{array}{l}
f :: \forall a\ b\,.\,(a = \text{Int} \wedge b = \text{Int} \quad \vee a = b \\
\qquad\qquad\quad , a = \text{Int} \qquad\qquad\ \ \vee a = \text{Bool} \\
\qquad\qquad\quad , a = \text{Int} \wedge b = \text{Float} \vee a = b) \Rightarrow a
\end{array}
$$

The function f may be typed with the generic instance Int. However, $f :: \text{Int}$ is not valid in GH and hence GQH would not be sound with respect to the former.

That is why GQH ensures that the qualified type for f is as follows:

$$
\begin{array}{l}
f :: a = \text{Int} \wedge b = \text{Int} \wedge b = \text{Float} \vee \\
\quad a = b \wedge a = \text{Bool} \Rightarrow a
\end{array}
$$

Making GQH sound with respect to GH.

Completeness and recursive type-indexed functions Most type-indexed functions are recursive. Currently we do not know how to type recursive type-indexed functions with qualified types. Therefore our type system types recursive type-indexed functions with types that do not use predicates. As a result, some recursive type-indexed functions do not have a principal type, since that type might be qualified. In order to exploit the principal types property in the inference algorithm, it disallows recursive type-indexed functions that may type with a qualified type. It follows that the type inference algorithm is not complete with respect to the type system.

Fortunately, the omitted functions are extremely contrived. By not allowing them we do not lose (to the best of our knowledge) useful type-indexed programs.

As an example, consider the following function:

$$
\begin{array}{l}
f\ \langle\alpha\rangle = \textbf{typecase } \alpha \textbf{ of} \\
\qquad\qquad [\beta] \rightarrow \mathit{const}\ [\bot]\ f\ \langle\beta\rangle
\end{array}
$$

This function can be typed with $f\ \langle\alpha\rangle :: \alpha$ and $f\ \langle\alpha\rangle :: \forall a\,.\,[a]$. This function does not have an unqualified principal type. Thus, it is not accepted by our type inference algorithm. More specifically, it is rejected by a condition in rule (w-gletrec) in Figure 8.

Type schemes
$\sigma ::= \ldots \mid q$

Type predicates
$\pi \quad ::= \varepsilon \vee \varepsilon$
$\varepsilon \quad ::= \mathit{True} \mid \varepsilon \wedge t_1 = t_2$

Predicate sets
$P, Q ::= \emptyset \mid P, \pi$

Qualified types
$q \ ::= P \Rightarrow t$

Environment
$\Gamma ::= \ldots \mid \Gamma, x\langle\alpha\rangle :: \cdot$

Generic instances environment
$\Delta ::= \emptyset \mid \Delta, x\ \langle A\rangle \sim t$

Fig. 4. Syntax for Generic Haskell with Qualified types and terms. Extends Figure 1

4.2 Type system with qualified types

This section presents the extension of of Generic Haskell's type system with qualified types.

Syntax and notation Figure 4 shows the extensions to Generic Haskell's type language to incorporate qualified types.

A qualified type has predicates that constrain type variables. Predicates are disjunctions of type equation conjunctions. For brevity we omit *True* when writing predicates. We regard conjunctions of type equations as sets: we assume no duplicate equations and their ordering does not matter; the same holds for sets of predicates. We write union of predicates with a comma P, Q.

A type environment Γ may now also contain incomplete bindings $x\langle\alpha\rangle :: \cdot$, which denote type-indexed functions with, as yet, unknown types. These bindings are used to infer the types of recursive type-indexed functions. An incomplete binding introduces, but does not define, a pattern variable α; behavior that is similar to type-indexed function bindings, discussed in Section 2.1.

The generic-instances environment (instances environment for short) Δ associates applications of type-indexed functions to type arguments with their inferred types. An instances environment is a set of such associations, we write the union operation with a comma: Δ, Δ'. We write Δ_x to express an instance environment Δ such that the instances for function x are removed. This environment and incomplete bindings are only used in type inference.

A substitution S is a finite mapping from type variables to types. We write Sa to denote the type corresponding to type variable a in the domain of S. The result of applying substitution S to type scheme σ is the type scheme obtained by replacing every free type variable in σ that is also in $\mathsf{dom}\ (S)$ by the type Sa. We generalize substitution application to environments Γ and instance environments Δ by applying substitutions to type schemes occurring in the environments. The extension of a substitution S for a type variable a is written $[\,a \mapsto t\,]S$ if $a \notin \mathsf{dom}\ (S)$.

A type scheme $\sigma \equiv \forall\overline{a_i}\,.\,P \Rightarrow t$, for bindings $x\ \langle\alpha\rangle :: \sigma$ and $x\ \langle\alpha\rangle :: \sigma'$, has a generic instance $\sigma' \equiv \forall\overline{b_i}\,.\,Q \Rightarrow t'$ if $t' \equiv t[\overline{t_i}/\overline{a_i}]$, $Q \Vdash P[\overline{t_i}/\overline{a_i}]$ for some types $\overline{t_i}$ such that variables $\overline{b_i}$ do not occur free in σ. We write this relation $\sigma' \leqslant_S \sigma$

where the substitution S is the witness of the generic instance: $S \equiv [\overline{a_i \mapsto t_i}]$. We may also choose to omit the substitution: $\sigma' \leqslant \sigma$.

Entail-TA	:		$\Vdash \mathit{True}$	$\vee\, \varepsilon$
Entail-NonTA	:		$\Vdash \varepsilon$	$\vee\, \mathit{True}$
I-TA	: ε_1	$\vee\, \varepsilon_2$	$\Vdash t = t \wedge \varepsilon_1$	$\vee\, \varepsilon_2$
I-NonTA	: ε_1	$\vee\, \varepsilon_2$	$\Vdash \varepsilon_1$	$\vee\, t = t \wedge \varepsilon_2$
E-TA	: $t_1 = t_2 \wedge \varepsilon_1$	$\vee\, \varepsilon_2$	$\Vdash \varepsilon_1$	$\vee\, \varepsilon_2$
E-NonTA	: ε_1	$\vee\, t_1 = t_2 \wedge \varepsilon_2$	$\Vdash \varepsilon_1$	$\vee\, \varepsilon_2$

Fig. 5. Entailment relation for predicates.

Type system Figure 6 shows the type checking rules for Generic Haskell with qualified types, which we call GQH. The judgement $P \mid \Gamma \vdash e :: \sigma$ assigns a type scheme σ to an expression e under environment Γ when the predicates P are satisfied.

The type inference rules, except for rule (glet) and rule (glet-rec), are the standard qualified types rules[17].

Rule (glet) types non-recursive type-indexed functions. The type scheme of the function should be a generic instance of all generalized marked types corresponding to the arms. Two more conditions on the type scheme ensure that its predicates are well-formed ($\vdash^{pred}$) and that occurrences of pattern variables are defined.

Rule (glet-rec) uses a more general marking relation that handles parametrized type patterns (Ω). The definition of the marking relation, in Figure 7, may introduce a predicate in a non-deterministic way. While the simple definition of Figure 3 introduces type variables, this definition constrains arbitrary types in the introduced predicates. Note that we write $\vdash \mathit{mark}^{\alpha}_{\Omega} t_1 \equiv q_2$ for the marking relation and $\mathit{mark}^{\alpha}_{\Omega} t_1 \equiv q_2$ for the algorithmic version. The algorithm differs only in that the introduced predicates constrain fresh variables. Figure 9 shows the introduction of predicates for algorithmic marking.

Rule (glet-rec) requires that the generic function types with an unqualified type scheme σ. Note that the rule allows recursion and that the environment is extended with the defined pattern variables of the type indices when typing arms.

The appendix defines a translation of GQH programs into the GH language to prove soundness.

Theorem 1 (Soundness of GQH). *The translation of a well-typed GQH program is a well-typed GH program.*

$$\frac{x :: \sigma \in \Gamma}{P \mid \Gamma \vdash x :: \sigma}\ (\text{var}) \qquad \frac{x\ \langle\alpha\rangle :: \sigma \in \Gamma \qquad \mathit{fdv}\ (A) \subseteq \mathit{dpv}\ (\Gamma)}{P \mid \Gamma \vdash x\ \langle A\rangle :: \sigma[A/\alpha]}\ (\text{gapp})$$

$$\frac{P \mid \Gamma, x :: t' \vdash e :: t \qquad \mathit{fpv}\ (t') \subseteq \mathit{dpv}\ (\Gamma)}{P \mid \Gamma \vdash \lambda x \to e :: t' \to t}\ (\to\text{I})$$

$$\frac{P \mid \Gamma \vdash e_1 :: t' \to t \qquad P \mid \Gamma \vdash e_2 :: t'}{P \mid \Gamma \vdash e_1\ e_2 :: t}\ (\to\text{E})$$

$$\frac{P, \pi \mid \Gamma \vdash e :: q}{P \mid \Gamma \vdash e :: \pi \Rightarrow q}\ (\Rightarrow\text{I}) \qquad \frac{P \mid \Gamma \vdash e :: \pi \Rightarrow q \qquad P \Vdash \pi}{P \mid \Gamma \vdash e :: q}\ (\Rightarrow\text{E})$$

$$\frac{P \mid \Gamma \vdash e :: \sigma \qquad a \notin \mathit{ftv}\ (\Gamma) \cup \mathit{ftv}\ (P)}{P \mid \Gamma \vdash e :: \forall a\,.\,\sigma}\ (\text{q-}\forall\text{I})$$

$$\frac{\begin{array}{l} P \mid \Gamma \vdash e :: \forall a\,.\,\sigma \\ \mathit{fpv}\ (t) \subseteq \mathit{dpv}\ (\Gamma) \end{array}}{P \mid \Gamma \vdash e :: \sigma[t/a]}\ (\text{q-}\forall\text{E}) \qquad \frac{\begin{array}{c} P \mid \Gamma \vdash e :: \sigma \\ Q \mid \Gamma, x :: \sigma \vdash e' :: t \end{array}}{P, Q \mid \Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' :: t}\ (\text{let})$$

$$\frac{\begin{array}{l} \overline{P_i \mid \Gamma \vdash e_i :: t_i} \qquad \overline{\vdash \mathit{mark}^{\alpha}_{T_i} t_i \equiv q_i} \qquad \overline{\sigma_i \equiv \mathit{Gen}(\Gamma; P_i \Rightarrow q_i)} \\ Q \mid \Gamma, x\ \langle\alpha\rangle :: \sigma \vdash e' :: t' \qquad \overline{\sigma \leqslant \sigma_i} \qquad \mathit{fpv}\ (\sigma) \subseteq \mathit{dpv}\ (\Gamma, \alpha) \qquad \vdash^{\mathit{pred}}_{\alpha} \sigma \end{array}}{Q \mid \Gamma \vdash \mathbf{let}\ x\ \langle\alpha\rangle = \mathbf{typecase}\ \alpha\ \mathbf{of}\ \overline{T_i \to e_i}\ \mathbf{in}\ e' :: t'}\ (\text{q-glet})$$

$$\frac{\begin{array}{l} \overline{P_i \mid \Gamma, x\ \langle\alpha\rangle :: t, \mathit{dpv}\ (\Omega_i) \vdash e_i :: t_i} \qquad \overline{\vdash \mathit{mark}^{\alpha}_{\Omega_i} t_i \equiv q_i} \\ \overline{\sigma \leqslant \sigma_i} \qquad \overline{\sigma_i \equiv \mathit{Gen}(\Gamma; P_i \Rightarrow q_i)} \qquad \sigma \equiv \forall \overline{a_i}\,.\,t \\ Q \mid \Gamma, x\ \langle\alpha\rangle :: \sigma \vdash e' :: t' \qquad \mathit{fpv}\ (\sigma) \subseteq \mathit{dpv}\ (\Gamma, \alpha) \end{array}}{Q \mid \Gamma \vdash \mathbf{letrec}\ x\ \langle\alpha\rangle = \mathbf{typecase}\ \alpha\ \mathbf{of}\ \overline{\Omega_i \to e_i}\ \mathbf{in}\ e' :: t'}\ (\text{q-gletrec})$$

Fig. 6. Type checking rules for Generic Haskell with Qualified types.

4.3 Type inference algorithm

This section describes the type-inference algorithm for type-indexed functions of arity 1. Figure 8 shows the most important rules of our algorithm. The inference judgement for the expression language takes the form $P \mid S\ \Gamma \vdash^{W} e :: t; \Delta$. The input parameters for the algorithm are an expression e and an environment Γ. The algorithm either produces a type t, a substitution S, a set of predicates P and constraints on instances of type-indexed functions Δ, or it fails.

The key idea of the algorithm is to collect the instance types of type-indexed functions in the instance environment Δ, to then use this information to infer the qualified base types of the functions.

The first type-application rule (w-gapp-1) specializes the type scheme of the type-indexed function with a given type argument, provided that all occurrences

$$\frac{\vdash trav^{\alpha}_{A}\, t \equiv t' \qquad \vdash intrP^{\alpha}_{A}\, t\ t' \equiv q}{\vdash mark^{\alpha}_{A}\, t \equiv q}\ \text{(mark)}$$

$$\frac{q \equiv t = \alpha \wedge t_1 = A \vee t = t_2 \Rightarrow t}{\vdash intrP^{\alpha}_{A} t_1\ t_2 \equiv q}\ \text{(intrP-1)} \qquad \frac{}{\vdash intrP^{\alpha}_{A} t_1\ t_2 \equiv t_2}\ \text{(intrP-2)}$$

$$\frac{\overline{\vdash mark^{\alpha}_{A} t_i \equiv P'_i \Rightarrow t'_i}}{\vdash trav^{\alpha}_{A}\ T\ \overline{t_i} \equiv \overline{P'_i} \Rightarrow\ T\ \overline{t'_i}}\ \text{(mark-app)} \qquad \frac{t \not\equiv T\ \overline{t_i}}{\vdash trav^{\alpha}_{A} t \equiv t}\ \text{(mark-rest)}$$

$$\frac{\forall \pi\,.\, \pi \in P \wedge \alpha \equiv \alpha_2\ \supset\ \{\,\overline{a_i}\,\} \mid (P - \pi) \vdash^{pred}_{\alpha_2} \pi}{\vdash^{pred}_{\alpha} \forall \overline{a_i}\,.\, P \Rightarrow t}\ \text{(chk-preds)}$$

$$\frac{\begin{array}{l} a \notin ftv\ (P) \cup ftv\ (\varepsilon_1) \\ \alpha \notin dpv\ (\varepsilon_1) \cup dpv\ (\varepsilon_2) \qquad a \in \{\,\overline{a_i}\,\} \end{array}}{\{\,\overline{a_i}\,\} \mid P \vdash^{pred}_{\alpha} a = \alpha \wedge \varepsilon_1 \vee \varepsilon_2}\ \text{(chk-pred)}$$

Fig. 7. Generalized marking relation and well-formedness of predicates.

of pattern variables are defined. Additionally, it instantiates the quantified type variables with fresh ones.

The second rule (w-gapp-2) deals with recursive occurrences. Since the type of function x is not yet known, it creates a fresh variable a and associates it in Δ with the type argument A. As type inference progresses, it provides more information about variable a, both in substitution S and environment Δ.

The non-recursive generic let collects the arms' types in the instance environment (Δ) with judgment $P \mid S\ \Gamma \vdash^{W}_{Arms(x)} \overline{T_i \to e_i}; \Delta$, shown in Figure 10. Judgement $\vdash^{x\ \langle\alpha\rangle} \Delta \rightsquigarrow q; S_2$ computes a qualified base type for function x using its corresponding instance types. The remaining judgments generalize the base type and use it to type expression e.

The recursive generic let extends the previous rule with the requirement that the *inst* algorithm only finds one instance for base-type scheme σ. In other words, the principal type should not need predicates; if it does, type inference fails. Note that arms are processed with an environment extended with an incomplete binding to allow recursive invocations. An additional check prevents pattern variables ($fpv\ (\{\,\overline{\Omega_i}\,\})$) from escaping.

Type inference for arms. The judgement $P \mid S\ \Gamma \vdash^{W}_{Arms(x)} \overline{A_i \to e_i}; \Delta$ collects the types of arms $\overline{e_i}$ in instance environment Δ. The case for no arms returns no information. The second case infers the type of arm e, which is stored in the instance environment together with results Δ and Δ_2. The pattern variables that are introduced can be used when processing e. Note that the substitution (S_2) resulting from the recursive call updates type information in Δ.

$$\frac{x :: \forall \overline{a_i} \,.\; P \Rightarrow t \in \Gamma \qquad \mathit{fresh}\ \overline{b_i}}{P[\overline{b_i}/\overline{a_i}] \mid \mathit{id}\ \Gamma \vdash^W x :: t[\overline{b_i}/\overline{a_i}]; \emptyset} \text{ (w-var)}$$

$$\frac{x\ \langle\alpha\rangle :: \forall \overline{a_i} \,.\; P \Rightarrow t \in \Gamma \qquad \mathit{fresh}\ \overline{b_i} \qquad \mathit{fdv}\ (A) \subseteq \mathit{dpv}\ (\Gamma)}{P[A/\alpha][\overline{b_i}/\overline{a_i}] \mid \mathit{id}\ \Gamma \vdash^W x\ \langle A\rangle :: t[A/\alpha][\overline{b_i}/\overline{a_i}]; \emptyset} \text{ (w-gapp-1)}$$

$$\frac{x\langle\alpha\rangle :: \cdot \in \Gamma \qquad \mathit{fresh}\ a \qquad \mathit{fdv}\ (A) \subseteq \mathit{dpv}\ (\Gamma)}{\emptyset \mid \mathit{id}\ \Gamma \vdash^W x\ \langle A\rangle :: a; x\ \langle A\rangle \sim a} \text{ (w-gapp-2)}$$

$$\frac{P \mid S\ \Gamma, x :: a \vdash^W e :: t; \Delta \qquad \mathit{fresh}\ a}{P \mid S\ \Gamma \vdash^W \lambda x \to e :: Sa \to t; \Delta} \text{ (w-}\to\text{I)}$$

$$\frac{\begin{array}{l} P \mid S_1\ \Gamma \vdash^W e_1 :: t_1; \Delta_1 \qquad Q \mid S_2\ S_1\Gamma \vdash^W e_2 :: t_2; \Delta_2 \\ S_2 t_1 \sim t_2 \to a \equiv S_3 \qquad \mathit{fresh}\ a \end{array}}{S_3S_2P, S_3Q \mid S_3S_2S_1\ \Gamma \vdash^W e_1\ e_2 :: S_3 a; S_3S_2\Delta_1, S_3\Delta_2} \text{ (w-}\to\text{E)}$$

$$\frac{\begin{array}{l} P \mid S_1\ \Gamma, x :: a \vdash^W e :: t; \Delta \qquad S_1 a \equiv t \qquad \mathit{fresh}\ a \\ \mathit{Gen}(S_1\Gamma, \Delta; P \Rightarrow t) \equiv \sigma \\ Q \mid S_2\ S_1\Gamma, x :: \sigma \vdash^W e' :: t'; \Delta' \end{array}}{Q \mid S_2S_1\ \Gamma \vdash^W \textbf{let}\ x = e\ \textbf{in}\ e' :: t'; S_2\Delta, \Delta'} \text{ (w-let)}$$

$$\frac{\begin{array}{l} P \mid S_1\ \Gamma \vdash^W_{Arms(x)} \overline{T_i \to e_i}; \Delta \qquad \vdash^{x\ \langle\alpha\rangle} \Delta \rightsquigarrow q; S_2 \\ \mathit{Gen}(S_2S_1\Gamma, S_2\Delta_x; S_2P \Rightarrow q) \equiv \sigma \\ Q \mid S_3\ S_2S_1\Gamma, x\ \langle\alpha\rangle :: \sigma \vdash^W e' :: t'; \Delta' \\ \Delta'' \equiv S_3S_2\Delta_x, \Delta' \end{array}}{S_3Q \mid S_3S_2S_1\ \Gamma \vdash^W \textbf{let}\ x\ \langle\alpha\rangle = \textbf{typecase}\ \alpha\ \textbf{of}\ \overline{T_i \to e_i}\ \textbf{in}\ e' :: t'; \Delta''} \text{ (w-glet)}$$

$$\frac{\begin{array}{l} P \mid S_1\ \Gamma, x\langle\alpha\rangle :: \cdot \vdash^W_{Arms(x)} \overline{\Omega_i \to e_i}; \Delta \qquad \vdash^{x\ \langle\alpha\rangle} \Delta \rightsquigarrow q; S_2 \\ \mathit{inst}\ (S_2P \Rightarrow q) \equiv \{(S_3, t)\} \qquad \mathit{Gen}(S_3S_2S_1\Gamma, S_3S_2\Delta_x; t) \equiv \sigma \\ \mathit{fpv}\ (\{\overline{\Omega_i}\})\#\mathit{dpv}\ (S_3S_2S_1\Gamma) \cup \mathit{fpv}\ (S_3S_2\Delta_x) \cup \mathit{fpv}\ (\sigma) \\ Q \mid S_4\ S_3S_2S_1\Gamma, x\ \langle\alpha\rangle :: \sigma \vdash^W e' :: t'; \Delta' \\ \Delta'' \equiv S_4S_3S_2\Delta_x, \Delta' \end{array}}{S_4Q \mid S_4S_3S_2S_1\ \Gamma \vdash^W \textbf{letrec}\ x\ \langle\alpha\rangle = \textbf{typecase}\ \alpha\ \textbf{of}\ \overline{\Omega_i \to e_i}\ \textbf{in}\ e' :: t'; \Delta''} \text{ (w-gletrec)}$$

Fig. 8. Type inference algorithm.

$$\frac{a \text{ fresh} \qquad mgu\ A\ t_1 \equiv S \qquad q \equiv a = \alpha \wedge t_1 = A \vee a = t_2 \Rightarrow a}{intrP^{\alpha}_{A} t_1\ t_2 \equiv q}\ \text{(W-intrP-1)}$$

$$\frac{mgu\ A\ t_1 \text{ is not defined}}{intrP^{\alpha}_{A} t_1\ t_2 \equiv t_2}\ \text{(W-intrP-2)}$$

Fig. 9. Introduction of predicates for marking in type inference.

$$\frac{}{P \mid id\ \Gamma \vdash^{W}_{Arms(x)} \emptyset; \emptyset}\ \text{(w-arm-1)} \qquad \frac{\begin{array}{l}\overline{mark^{\alpha}_{A_i} t_i \equiv P_i \Rightarrow t'_i} \\ mgu\ (\overline{t'_i}) \equiv S_1 \qquad t' \in \{\overline{t'_i}\} \\ compound_{\alpha}(S_1(\overline{P_i} \Rightarrow t')) \equiv q \\ simplify\ (q) \equiv (S_2, q') \\ \Delta \equiv \Delta_x \qquad S \equiv S_2 S_1\end{array}}{\vdash^{x\ \langle\alpha\rangle} (\Delta, \overline{x\ \langle A_i \rangle \sim t_i}) \rightsquigarrow q'; S}\ \text{(unify)}$$

$$\frac{\begin{array}{l}P \mid S\ \Gamma, fpv\ (\Omega) \vdash^{W}\ e :: t; \Delta \\ P_2 \mid S_2\ S\Gamma \vdash^{W}_{Arms(x)} R; \Delta_2 \\ \Delta_3 \equiv S_2\Delta, \Delta_2, x\ \langle\Omega\rangle \sim S_2 t\end{array}}{S_2 P, P_2 \mid S_2 S\ \Gamma \vdash^{W}_{Arms(x)} \Omega \rightarrow e, R; \Delta_3}\ \text{(w-arm-2)}$$

Fig. 10. Type inference algorithm for arms and unification of instance types.

Unification of instance types. The judgement $\vdash^{x\ \langle\alpha\rangle}\ \Delta \rightsquigarrow q; S$, shown in Figure 10, computes a qualified base type from the instance types of type-indexed function x. First, the rule marks all instance types of x in environment Δ. Next, it finds the most general unifier of all resulting types without considering predicates. The preliminary base type results from the application of the unifying substitution to all predicates P'_i and any type t'_i. Finally, the rule invokes algorithm *compound* to ensure consistent choices for constrained variables so that the resulting qualified type is well-formed. The remaining condition ensures that all type instances of x are processed.

Algorithm compound. The algorithm compound rewrites predicates that restrict a type variable to a single predicate. That ensures consistent choices for that type variable.

$$\begin{array}{l} compound_{\alpha}(a = \alpha \wedge \varepsilon_1 \vee \varepsilon_2, a = \alpha \wedge \varepsilon'_1 \vee \varepsilon'_2, P) \\ \quad \equiv compound_{\alpha}(a = \alpha \wedge \varepsilon_1 \wedge \varepsilon'_1 \vee \varepsilon_2 \wedge \varepsilon'_2) \\ compound_{\alpha}(a = \alpha \wedge \varepsilon_1 \vee \varepsilon_2, P) \\ \quad \equiv a = \alpha \wedge \varepsilon_1 \vee \varepsilon_2, compound_{\alpha}(P) \\ \quad \textbf{where } a = \alpha \text{ does not occur in } P \\ compound_{\alpha}(\emptyset) \\ \quad \equiv \emptyset \end{array}$$

Instances algorithm. This algorithm computes the set of instances of a qualified type.

$$
\begin{array}{l}
\mathit{inst}\ (q) \equiv \mathit{inst}'\ (\mathit{id}, q) \\
\mathit{inst}'\ (S, (\overline{t_i = t_i'} \vee \overline{t_j = t_j'}, P) \Rightarrow t) \\
\qquad = \quad \mathit{inst}\ (S_1 S, S_1 P \Rightarrow S_1 t) \\
\qquad \cup \quad \mathit{inst}\ (S_2 S, S_2 P \Rightarrow S_2 t) \\
\quad \textbf{where} \\
\qquad S_1 = \overline{\mathit{mgu}\ t_i\ t_i'} \\
\qquad S_2 = \overline{\mathit{mgu}\ t_j\ t_j'} \\
\qquad\quad \text{The failure of the } \mathit{mgu} \text{ algorithm in the computation} \\
\qquad\quad \text{of } S_1 \text{ or } S_2 \text{ makes the corresponding} \\
\qquad\quad \text{left or right union parameter empty.} \\
\mathit{inst}'\ (S, t) \\
\qquad = \quad \{(S, t)\}
\end{array}
$$

Its worst case complexity is exponential. However, most type-indexed functions generate type equations with enough information to make the average case good enough for practical purposes. The overline on the calls to the *mgu* function denotes the composition of the resulting substitutions.

Predicate simplification. Predicate simplification removes predicates that trivially contradict type equations:

$$
\begin{array}{l}
\mathit{simplify}\ (q) \\
\quad = \mathit{simplify}'\ (q; \emptyset; \mathit{id}) \\
\mathit{simplify}'\ (t_1 = t_2 \wedge \varepsilon \vee \overline{t_i = t_i'}, P \Rightarrow t; Q; S) \\
\quad = \mathit{simplify}'\ (S'(Q, P \Rightarrow t); \emptyset; S'S) \\
\quad \textbf{where } S' = \mathit{mgu}\ t_i\ t_i' \\
\qquad\quad \mathit{mgu}\ t_1\ t_2 \text{ is not defined} \\
\mathit{simplify}'\ (\overline{t_i = t_i'} \vee t_1 = t_2 \wedge \varepsilon, P \Rightarrow t; Q; S) \\
\quad = \mathit{simplify}'\ (S'(Q, P \Rightarrow t); \emptyset; S'S) \\
\quad \textbf{where } S' = \mathit{mgu}\ t_j\ t_j' \\
\qquad\quad \mathit{mgu}\ t_1\ t_2 \text{ is not defined} \\
\mathit{simplify}'\ (\pi, P \Rightarrow t; Q; S) \\
\quad = \mathit{simplify}'\ (P \Rightarrow t; \pi, Q; S) \\
\quad \textbf{where } \text{all equations in } \pi \text{ are unifyable} \\
\mathit{simplify}'\ (t; P; S) \\
\quad = (S, P \Rightarrow t)
\end{array}
$$

The appendix proves the following two theorems for type inference.

Theorem 2 (Soundness of type inference). *The type inference result of a program can be used to construct a GQH type derivation for it.*

5 Conclusions, related and future work

We have introduced several type inference problems typical for generic programming, we have discussed (partial) solutions to these problems, and we have argued why, in most cases, we cannot achieve something better.

Our main technical contribution is the introduction of an adapted type system for a restricted class of type-indexed functions, in which we use a special kind of qualified types for type-indexed functions. In the new type system, type-indexed functions have principal types, and we can give a sound type-inference algorithm for the base type of type-indexed functions. A prototype implementation is available at [23]. The algorithm is not complete, but the counterexamples we found are rather contrived generic functions. In the type system we assume that type-indexed functions have arity 1. This is a fundamental restriction. However, type-indexed functions of arity greater than 1 not in the library of Generic Haskell are rare, so we expect our type-inference algorithm to be useful for most generic programs. Our results also apply to similar approaches to generic programming, such as PolyP, Haskell's derivable type classes, and Clean's generic programming extension.

The type-argument inference problem has also been solved in Clean [1] and PolyP [15], using similar techniques as we propose. Inferring base types for generic functions is similar to System CT [4] and, to a lesser extent, GCaml [8]. In the former, a variant of anti-unification generalizes a set of instance types for overloaded functions. In the latter, base types carry constraints that should be solved at every type application. No attempt is made to infer an unconstrained generalized type. In the 'Scrap your boilerplate' [20] approach the types of the basic generic combinators have to be supplied explicitly, because of the polymorphic arguments used in the combinators. Using Haskell's class system the types of the functions that use these combinators can then be inferred. The same largely holds for Hinze's lightweight approach [11]. Just as kind inference is a lot easier than type inference, inferring the kind of a type-indexed *type* [6, 13] is a lot easier than inferring the base type of a type-indexed function.

A type-inference problem we did not look at is to infer the type pattern in the definition of a type-indexed function. Given the type of the type-indexed function *eq*, we can infer from the arm $eq\ True\ True = True$ in the definition of *eq* that the type pattern for *eq* in this arm should be Bool. However, it will be difficult to infer the pattern in the arm $eq = (==)$. We have not yet looked in detail at this problem.

We expect that the qualified types approach we use might have applications in other areas. The problem of inferring the base type of a type-indexed function is the same as inferring the class declaration given the types of member functions on instance types of the class [26]. This might be useful in checking that given a class declaration and a number of instance declarations, the class declaration doesn't impose unnecessary restrictions on the types of member functions. In the future we hope to investigate the relation between type inference for generic functions and areas such as generalized algebraic data types and intersection types. Furthermore, type inference for generic functions is a first step towards generating generic functions from data type specific functions defined on several data types. We intend to investigate how the techniques described in this paper can be extended to be able to generate generic functions from examples.

Acknowledgements. Carlos Camarão challenged us to investigate the base-type inference question. Ralf Hinze suggested an improvement for our notation.

References

1. Artem Alimarine and Rinus Plasmijer. A generic programming extension for clean. In *IFL'01*, volume 2312 of *LNCS*, pages 168–186. Springer-Verlag, 2002.
2. Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Generic Programming*, pages 1–20. Kluwer Academic Publishers, 2003.
3. Didier Le Botlan and Didier Rémy. MLF: raising ML to the power of System F. In *ICFP'03*, pages 27–38, 2003.
4. Carlos Camarão, Lucília Figueiredo, and Cristiano Vasconcellos. Constraint-set satisfiability for overloading. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 67–77, New York, NY, USA, 2004. ACM Press.
5. Sébastien Carlier and J. B. Wells. Type inference with expansion variables and intersection types in system E and an exact correspondence with β-reduction. In *PPDP'04*, pages 132–143, 2004.
6. Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05*, pages 1–13. ACM Press, 2005.
7. L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL'82*, pages 207–212, 1982.
8. Jun Furuse. Generic polymorphism in ML. Journées Francophones des Langages Applicatifs 2001, 2001.
9. Ralf Hinze. A generic programming extension for Haskell. In *Haskell Workshop*, Technical report of Utrecht University, UU-CS-1999-28, 1999.
10. Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, 2002.
11. Ralf Hinze. Generics for the masses. In *ICFP '04*, pages 236–243, 2004.
12. Ralf Hinze and Johan Jeuring. Generic Haskell: applications. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 57–97. Springer-Verlag, 2003.
13. Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 51((1-2)):117–151, 2004.
14. Ralf Hinze and Simon Peyton Jones. Derivable type classes. In *Haskell Workshop*, 2000.
15. Patrik Jansson and Johan Jeuring. PolyP — a polytypic programming language extension. In *POPL'97*, pages 470–482, 1997.
16. Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, Monterey, CA*, 2002.
17. Mark P. Jones. *Qualified types: theory and practice*. Cambridge University Press, 1995.
18. Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types, 2004. Submitted for publication.
19. Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: Practical type inference for generalised algebraic dataypes. Available from `http://www.cis.upenn.edu/~sweirich/publications.html`, 2004.

20. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003. TLDI'03.
21. Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, September 2004.
22. Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In *ICFP'03*, pages 141–152. ACM Press, 2003.
23. Alexey Rodriguez, Johan Jeuring, and Andres Löh. Type Inference for Generic Haskell. Technical Report and Prototype available from `http://www.cs.uu.nl/wiki/Alexey/GHTypeInference`, 2005.
24. Vincent Simonet and François Pottier. Constraint-based type inference for guarded algebraic data types. Submitted for publication, July 2003.
25. Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy type inference for higher-rank types and impredicativity. Available from `http://www.cis.upenn.edu/~sweirich/publications.html`, 2005.
26. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM Press.
27. Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *POPL '03*, pages 224–235. ACM Press, 2003.

A Soundness proof for type system

A.1 Typed translation

This is a an extension of the GQH judgments from Figure 6. We use this to prove the soundness of the system.

A derivation $P \mid \Gamma \vdash e \overset{S}{\rightsquigarrow} e' :: \sigma$ translates a well-typed GQH program e and produces a GH program e'. The substitution S identifies a particular instance of σ, using relation *inst*: $inst\ (\sigma) \equiv (S, \sigma')$.

A GQH binding $x :: \sigma$ has a qualified type σ. Its translation to GH assigns it an instance type, which is unqualified type. We use a substitution subscript to identify one of the possible instances of x.

A.2 Soundness proof

Definition 1 (Instance type schemes[1]). *The inst relation holds when a type scheme in the QGH type system has a substitution S that satisfies the predicates and the same substitution obtains an instance GH type scheme.*

$$inst\ (\forall \overline{a_i}\ .\ P \Rightarrow t) \equiv (S, \forall \overline{b_i}\ .\ St)$$
$$\textbf{where } \Vdash SP$$
$$\{\overline{b_i}\} = \{\overline{a_i}\} - \mathsf{dom}\ (S)$$

The current section (for the soundness proof of GQH) uses this definition of *inst* instead of the algorithmic one presented in Section 4.3.

[1] We don't explicitly distinguish between GH and QGH type schemes with different syntactic categories. It is not a problem since it is always clear from the context.

$$\frac{x\ \langle\alpha\rangle :: \sigma \in \Gamma \qquad \mathit{fpv}\ (A) \subseteq \mathit{dpv}\ (\Gamma) \qquad \mathit{inst}\ (\sigma) \equiv (S, \sigma')}{P \mid \Gamma \vdash x\ \langle A\rangle \overset{S[A/\alpha]}{\rightsquigarrow} x_S\ \langle A\rangle :: \sigma[A/\alpha]}\ \text{(qt-gapp)}$$

$$\frac{P \mid \Gamma, x :: t' \vdash e \overset{\emptyset}{\rightsquigarrow} e' :: t \qquad \mathit{fpv}\ (t') \subseteq \mathit{dpv}\ (\Gamma)}{P \mid \Gamma \vdash \lambda x \to e \overset{\emptyset}{\rightsquigarrow} \lambda x_\emptyset \to e' :: t' \to t}\ \text{(qt-}\to\text{I)}$$

$$\frac{P \mid \Gamma \vdash e_1 \overset{\emptyset}{\rightsquigarrow} e_1' :: t' \to t \qquad P \mid \Gamma \vdash e_2 \overset{\emptyset}{\rightsquigarrow} e_2' :: t'}{P \mid \Gamma \vdash e_1\ e_2 \overset{\emptyset}{\rightsquigarrow} e_1'\ e_2' :: t}\ \text{(qt-}\to\text{E)}$$

$$\frac{P \mid \Gamma \vdash e \overset{S}{\rightsquigarrow} e' :: \sigma[t/a] \qquad a \notin \mathit{ftv}\ (\Gamma) \cup P}{P \mid \Gamma \vdash e \overset{[a \mapsto t]S}{\rightsquigarrow} e' :: \forall a\,.\,\sigma}\ \text{(qt-}\forall\text{I)}$$

$$\frac{P \mid \Gamma \vdash e \overset{[a \mapsto t]S}{\rightsquigarrow} e' :: \forall a\,.\,\sigma \qquad a \notin \mathsf{dom}\ (S)}{P \mid \Gamma \vdash e \overset{S}{\rightsquigarrow} e' :: \sigma[t/a]}\ \text{(qt-}\forall\text{E)}$$

$$\frac{P, \pi \mid \Gamma \vdash e \overset{\emptyset}{\rightsquigarrow} e' :: q}{P \mid \Gamma \vdash e \overset{\emptyset}{\rightsquigarrow} e' :: \pi \Rightarrow q}\ \text{(qt-}\Rightarrow\text{I)} \qquad \frac{P \mid \Gamma \vdash e \overset{\emptyset}{\rightsquigarrow} e' :: \pi \Rightarrow q \qquad P \Vdash \pi}{P \mid \Gamma \vdash e \overset{\emptyset}{\rightsquigarrow} e' :: q}\ \text{(qt-}\Rightarrow\text{E)}$$

Fig. 11. Type translation rules for QGH, Part 1. Extends Figure 6.

$$\overline{E_i} = \left\{ \textbf{let } x_S = e' \textbf{ in } [\,] \;/\; \begin{array}{l} P \mid \Gamma \vdash e \overset{S}{\rightsquigarrow} e''' :: \sigma \\ \mathit{inst}\ (\sigma) = (S, \sigma') \end{array} \right\}$$

$$\frac{Q \mid \Gamma, x :: \sigma \vdash e' \overset{\emptyset}{\rightsquigarrow} e'' :: t}{P, Q \mid \Gamma \vdash \textbf{let } x = e \textbf{ in } e' \overset{\emptyset}{\rightsquigarrow} \overline{E_i}\,[e''] :: t} \text{ (qt-let)}$$

$$\overline{E_j} = \left\{ \textbf{let } x_{S_j}\ \langle\alpha\rangle = \textbf{typecase } \alpha \textbf{ of } \overline{T_i \to e_{ij}} \textbf{ in } [\,] \;/\; \begin{array}{l} \overline{\overline{S_j S_i P_i \mid \Gamma \vdash e_i \overset{\emptyset}{\rightsquigarrow} e_{ij} :: S_j S_i t_i}} \\ \overline{\vdash \mathit{mark}^{\alpha}_{T_i} t_i \equiv q_i} \\ \overline{\sigma_i \equiv \mathit{Gen}(\Gamma; P_i \Rightarrow q_i)} \\ \overline{\sigma \leqslant_{S_i} \sigma_i} \\ \mathit{inst}\ (\sigma) \equiv (S_j, \sigma_j) \end{array} \right\}$$

$$\frac{Q \mid \Gamma, x\ \langle\alpha\rangle :: \sigma \vdash e' \overset{\emptyset}{\rightsquigarrow} e'' :: t' \qquad \mathit{fpv}\ (\sigma) \subseteq \mathit{dpv}\ (\Gamma, \alpha) \qquad \vdash^{\mathit{pred}}_{\alpha} \sigma}{Q \mid \Gamma \vdash \textbf{let } x\ \langle\alpha\rangle = \textbf{typecase } \alpha \textbf{ of } \overline{T_i \to e_i} \textbf{ in } e' \overset{\emptyset}{\rightsquigarrow} \overline{E_j}\,[e''] :: t'} \text{ (qt-glet)}$$

$$\frac{\begin{array}{l} \overline{\begin{array}{l} S_i P_i \mid \Gamma, x\ \langle\alpha\rangle :: t, \mathit{dpv}\ (\Omega_i) \vdash e_i \overset{\emptyset}{\rightsquigarrow} e_i' :: S_i t_i \qquad \overline{\vdash \mathit{mark}^{\alpha}_{\Omega_i} t_i \equiv q_i} \\ \overline{\sigma \leqslant \sigma_i} \qquad \overline{\sigma_i \equiv \mathit{Gen}(\Gamma; P_i \Rightarrow q_i)} \qquad \sigma \equiv \forall \overline{a_j}\,.\,t \\ \overline{\mathit{inst}\ (\sigma_i) \equiv (S_i, \sigma)} \end{array}} \\ Q \mid \Gamma, x\ \langle\alpha\rangle :: \sigma \vdash e' \overset{\emptyset}{\rightsquigarrow} e'' :: t' \qquad \mathit{fpv}\ (\sigma) \subseteq \mathit{dpv}\ (\Gamma, \alpha) \end{array}}{\begin{array}{l} Q \mid \Gamma \vdash \textbf{letrec } x\ \langle\alpha\rangle = \textbf{typecase } \alpha \textbf{ of } \overline{\Omega_i \to e_i} \textbf{ in } e' \\ \overset{\emptyset}{\rightsquigarrow}\ \textbf{let } x_\emptyset\ \langle\alpha\rangle = \textbf{typecase } \alpha \textbf{ of } \overline{\Omega_i \to e_i'} \textbf{ in } e'' :: t' \end{array}} \text{ (q-gletrec)}$$

Fig. 12. Type translation rules for QGH, Part 2. Extends Figure 6.

Lemma 1. *The inst relation is always defined on unqualified types: inst* $(t) \equiv (\emptyset, t)$.

Definition 2 (Instances environment). *The function inst transforms a QGH environment* Γ *into an environment valid in the GH type system. The transformation takes every binding from* Γ *and produces a set of bindings, each indexed by a substitution that witnesses the instance type of the original type scheme* σ.

$$\begin{aligned} inst_\Gamma\ (\Gamma) \equiv \{\, x_S\ \langle\alpha\rangle :: \sigma' \mid x\ \langle\alpha\rangle :: \sigma \in \Gamma \\ , inst\ (\sigma) \equiv (S, \sigma')\,\} \cup \\ \{\, x_S :: \sigma' \mid x :: \sigma \in \Gamma \\ , inst\ (\sigma) \equiv (S, \sigma')\,\} \end{aligned}$$

Lemma 2 (Preservation of instances under pattern variable substitution). *If inst* $(\sigma[A/\alpha]) \equiv (S', \sigma')$ *for a binding* $x\ \langle\alpha\rangle :: \sigma$ *with well-formed predicates, then there exist a substitution* S'' *and type scheme* σ'' *such that inst* $(\sigma) = (S'', \sigma'')$, $\sigma''[A/\alpha] \equiv \sigma'$ *and* $S''[A/\alpha] = S'$.

Proof. We prove the lemma by constructing the required S'' and σ''. Being well-formed, predicates in σ with an occurring α have the form $a = \alpha \wedge \varepsilon_1 \vee \varepsilon_2$. From the definition of *inst* $(\sigma[A/\alpha]) \equiv (S', \sigma')$, we have $\Vdash S'(a = A \wedge \varepsilon_1 \vee \varepsilon_2)$ for every predicate in $\sigma[A/\alpha]$. Now, we construct S'' by making $S'a \equiv S''a$ for type variables a that are not equated to the type argument α in σ. Otherwise, we consider the predicate where a is equated to α; we have two possibilities: S' satisfies $a = A \wedge \varepsilon_1$ or ε_2, both in $\sigma[A/\alpha]$. In the former case, we make $S''a \equiv \alpha$ which satisfies $a = \alpha \wedge \varepsilon_1$ and in the latter case we choose $S''a \equiv S'a$ which satisfies ε_2.

We justify that $S''a \equiv \alpha$ satisfies equations ε_1 and the other predicates in σ. Suppose it doesn't, then there is an equation in σ along the lines of $a = A$. Such an equation does not occur in well-formed predicates and is thus a contradiction. Well-formedness also forbids equations such as $A = \alpha$ in σ, which would not be satifiable in σ but would in $\sigma[A/\alpha]$.

Repeating the procedure for all variables we have $S''[A/\alpha] = S'$ and by def. of *inst* we obtain *inst* $(\sigma) \equiv (S'', \sigma'')$. Finally, it follows that $\sigma''[A/\alpha] \equiv \sigma'$.

Lemma 3. *For all* a, σ, σ', S *and* t *such that* $a \notin \mathsf{dom}\ (S)$ *we have that inst* $(\forall a\,.\,\sigma) \equiv ([a \mapsto t]\ S, \sigma')$ *if and only if inst* $(\sigma[t/a]) \equiv (S, \sigma')$.

Lemma 4. *For all bindings* $x\ \langle\alpha\rangle :: \sigma$, A, S', S'', σ', σ'' *such that inst* $(\sigma) \equiv (S', \sigma')$ *and inst* $(\sigma[A/\alpha]) \equiv (S'', \sigma'')$; *we have* $\sigma'' \equiv \sigma'[A/\alpha]$ *if* $S'' \equiv S'[A/\alpha]$.

Lemma 5. *For all* π, t, S *and* q *such that* $\Vdash \pi$ *we have that inst* $(\pi \Rightarrow q) \equiv (\emptyset, t)$ *if and only if inst* $(q) \equiv (\emptyset, \sigma)$.

Lemma 6 (Instances of a generic instance). *If* $\sigma \leqslant_S \sigma'$ *then inst* $(\sigma) \equiv (S'', \sigma'')$ *implies inst* $(\sigma') \equiv (S''S, \sigma'')$.

Lemma 7 (Mark has solutions). *If* $\vdash mark^\alpha_A \Gamma; t_1 \equiv P_2 \Rightarrow t_2$ *and* $\Vdash SP_2$ *then* $St_1 \equiv St_2[A/\alpha]$.

Proof. We prove the following: If $\vdash \mathit{mark}^{\alpha}_{A}\Gamma; t_1 \equiv P_2 \Rightarrow t_2$ or $\vdash \mathit{trav}^{\alpha}_{A}\Gamma; t_1 \equiv P_2 \Rightarrow t_2$ and $\Vdash SP_2$, then $St_1 \equiv St_2[A/\alpha]$.

- Case (makeQ): We use the ind. hyp.: if $\vdash \mathit{trav}^{\alpha}_{A}\Gamma; t_1 \equiv P_2 \Rightarrow t_2$ and $\Vdash SP_2$, then $St_1 \equiv St_2[A/\alpha]$.
  From $\Vdash S(P, a = \alpha \wedge t = A \vee a = t')$ it follows that $\{Sa \equiv \alpha, St \equiv A\}$ or $\{Sa = St'\}$. We need to prove $St \equiv Sa[A/\alpha]$. Under the first set of conditions we can establish $A \equiv \alpha[A/\alpha]$ and then $A \equiv A$. Under the second set we have $St \equiv St'[A/\alpha]$ which is exactly what the ind. hyp. tells us.
- Case (mark-rest): This case is trivial.
- Case (mark-app): By the ind. hyp. we have: if $\overline{\vdash \mathit{mark}^{\alpha}_{A}\Gamma; t_i \equiv P'_i \Rightarrow t'_i}$ and $\Vdash SP'_i$, then $St_i \equiv St'_i[A/\alpha]$. The premises are easily satisfied, so using congruence we obtain: $T\ \overline{St_i} \equiv T\ \overline{St'_i[A/\alpha]}$. The substitutions do not affect T, thus, it follows that $S(T\ \overline{t_i}) \equiv S(T\ \overline{t'_i})[A/\alpha]$.

Theorem 3 (Soundness). *If* $P \mid \Gamma \vdash e \overset{S}{\rightsquigarrow} e' :: \sigma$ *and* $\Vdash P$ *and* $\mathit{inst}\ (\sigma) \equiv (S, \sigma')$ *and* $\mathit{inst}_{\Gamma}\ (\Gamma) \equiv \Gamma'$ *then* $\Gamma' \vdash e' :: \sigma'$.

Note that relation *inst* may not hold when the type scheme has a predicate where a free variable occurs. It follows that this theorem does not hold for such predicates. It is easy to handle such cases moving the predicate from the scheme to predicates P. Otherwise that variable can be quantified.

Proof. We use an induction argument on the type derivation.

- Case (qt-var): From $x :: \sigma \in \Gamma$, $\mathit{inst}\ (\sigma) \equiv S; \sigma'$ and the definition of $\mathit{inst}_{\Gamma}\ (\Gamma) \equiv \Gamma'$ we have $x_S :: \sigma' \in \Gamma'$, which is the required premise to obtain $\Gamma' \vdash x_S :: \sigma'$.
- Case (qt-gapp): From $x\ \langle\alpha\rangle :: \sigma \in \Gamma$ and $\mathit{inst}\ (\sigma) \equiv (S, \sigma')$ we have $x_S\ \langle\alpha\rangle :: \sigma' \in \Gamma'$ such that $\mathit{inst}_{\Gamma}\ (\Gamma) \equiv \Gamma'$. It follows that $\Gamma' \vdash x_S\ \langle A\rangle :: \sigma'[A/\alpha]$. From $\mathit{inst}\ (\sigma[A/\alpha]) \equiv (S[A/\alpha], \sigma'')$ and Lemma 4 we have $\sigma'[A/\alpha] \equiv \sigma''$ and thus $\Gamma' \vdash x\ \langle a\rangle :: \sigma''$, as required.
- Case (qt-→I): We have $\mathit{inst}\ (t' \to t) \equiv (\emptyset, t' \to t)$ and $\mathit{inst}\ (t) \equiv (\emptyset, t)$ which we use to satisfy the ind. hyp. premise and obtain $\Gamma'' \vdash e' :: t$ such that $\mathit{inst}_{\Gamma}\ (\Gamma, x :: t') \equiv \Gamma''$. By the definition of inst_{Γ} we also have that $\Gamma'' \equiv \Gamma', x_{\emptyset} :: t'$ and thus $\Gamma' \vdash \lambda x_{\emptyset} \to e' :: t' \to t$.
- Case (qt-→E): Using Lemma 1 we satisfy the needed premises of the induction cases to obtain $\Gamma' \vdash e'_1 :: t' \to t$ and $\Gamma' \vdash e'_2 :: t'$. It follows that $\Gamma' \vdash e'_1\ e'_2 :: t$.
- Case (qt-∀I): We have $\mathit{inst}\ (\forall a\,.\,\sigma) \equiv [a \mapsto t]\ S; \sigma'$ which implies $\mathit{inst}\ (\sigma[t/a]) \equiv (S, \sigma')$ by Lemma 3. The induction hypothesis states that if $\mathit{inst}\ (\sigma[t/a]) \equiv (S, \sigma'')$ for all σ'' then $\Gamma' \vdash e :: \sigma''$. It follows that $\Gamma' \vdash e :: \sigma'$.
- Case (qt-∀E): We have $\mathit{inst}\ (\sigma[t/a]) \equiv (S, \sigma')$ which implies $\mathit{inst}\ (\forall a\,.\,\sigma) \equiv ([a \mapsto t]\ S, \sigma')$ by Lemma 3. The induction hypothesis states that if $\mathit{inst}\ (\forall a\,.\,\sigma) \equiv ([a \mapsto t]\ S, \sigma'')$ then $\Gamma' \vdash e :: \sigma''$ for all σ''. It follows that $\Gamma' \vdash e :: \sigma'$.
- Case (qt-⇒I): From $\mathit{inst}\ (\pi \Rightarrow q) \equiv (\emptyset, t')$ and Lemma 5 it follows that $\mathit{inst}\ (q) \equiv (\emptyset, t')$ which we use to obtain $\Gamma' \vdash e' :: t'$ from the induction hypothesis.

– Case (qt-⇒E): From $inst\ (q) \equiv (\emptyset, t')$, $\Vdash \pi$ and Lemma 5 it follows that $inst\ (\pi \Rightarrow q) \equiv (\emptyset, t')$ which we use to obtain $\Gamma' \vdash e' :: t'$ from the induction hypothesis.
– Case (qt-let): Assertion $inst\ (\sigma) \equiv (S, \sigma')$ satisfies the induction hypothesis premise to obtain $\Gamma' \vdash e''' :: \sigma'$ for all e''' in $\overline{E_i}$. Note that we may weaken the typing of instance σ_i with an assumption $x_{S_j} :: \sigma_j$ of another instance j as follows $\Gamma', x_{S_j} :: \sigma_j \vdash e_i :: \sigma_i$ since $x \notin fv\ (e)$. From the definition of $inst_\Gamma$ we have $inst_\Gamma\ (\Gamma, x :: \sigma) \equiv \Gamma', \overline{x_{S_i} :: \sigma_i}$ for all S_i and σ_i such that $inst\ (\sigma) \equiv (S_i, \sigma_i)$. It follows that we can obtain $\Gamma', \overline{x_{S_i} :: \sigma_i} \vdash e'' :: t$ from the induction hypothesis. We now show that by nesting the definitions for the instances of x we build the required environment $\Gamma', \overline{x_{S_i} :: \sigma_i}$ starting from Γ'. Indeed, taking $E_0\ [E_1\ [e'']]$ as a translation (assuming two instances), we start with the environment Γ', extended to $\Gamma', x_{S_0} :: \sigma_0$, and then $\Gamma', x_{S_0} :: \sigma_0, x_{S_1} :: \sigma_1$ when it reaches e''. Since we can weaken the typing of an instance of x by inserting the bindings of previous instances, we are able to build a typed translation for let.
– Case (qt-glet): From $inst\ (\sigma) \equiv (S_j, \sigma_j)$, Lemma 6 and the def. of *inst*, we have $\Vdash S_j S_i P_i$, which satisfies a condition from the ind. hyp. to give $\Gamma' \vdash e_{ij} :: S_j S_i t_i$. Using assumption $q_i \equiv P_i' \Rightarrow t_i'$ and Lemma 7 we obtain $S_j S_i t_i \equiv S_j S_i t_i'[\,T_i/\alpha]$. From assumption $\forall \overline{a_k}\,.\,t_j \equiv \sigma_j$ and the def. of *inst* it follows that $t_j \equiv S_j S_i t_i'$ and so $\Gamma' \vdash e_{ij} :: t_j[\,T_i/\alpha]$, which we may weaken as above with an additional assumption: $\Gamma', x_{S_{j'}}\ \langle\alpha\rangle :: \sigma_{j'} \vdash e_{ij} :: t_j$. From the ind. hyp. $\Gamma, \overline{x_{S_j}\ \langle\alpha\rangle :: \sigma_j} \vdash e'' :: t'$ and the previous result we may nest instance definitions to obtain the desired typing derivation for the translation. Note that the definition of *inst* ensures To obtain $\overline{a_k} \# ftv\ (\Gamma')$, we choose variables not occurring in Γ for the quantified variables of σ, then by the definition of *inst* it follows that the same holds for σ_j.
– Case (qt-gletrec): From $inst\ (\sigma_i) \equiv (S_i, \sigma)$ and assumming $q_i \equiv P_i' \Rightarrow t_i'$ we have $\Vdash S_i P_i, S_i P_i'$. Then we obtain $\Gamma', x_\emptyset\ \langle\alpha\rangle :: t, dpv\ (\Omega_i) \vdash e_i :: S_i t_i$ such that $inst\ (\Gamma) \equiv \Gamma'$. From Lemma 7 we have $S_i t_i \equiv S_i t_i'[\Omega_i/\alpha]$ and by the def. of *inst*: $S_i t_i' \equiv t$; it follows that $\Gamma', x_\emptyset\ \langle\alpha\rangle :: t \vdash e_i' :: t[\Omega_i/\alpha]$. From the ind. hyp. we have $\Gamma', x_\emptyset\ \langle\alpha\rangle :: \sigma \vdash e'' :: \sigma$. Finally, to complete the proof, we obtain $\overline{a_j} \# ftv\ (\Gamma')$ following the same argument as in the previous case.

The typed translation rules in Figure 6 are different than the ones in Figure 11. It remains for us to show that whenever a term is typable under GQH, we can translate it to GH.

Definition 3 (Well-formed environments). *As pointed out in Section 2.1, a well-formed environment Γ binds possibly type-indexed functions to type schemes. The type schemes of type-indexed functions should be well-formed. The environment Γ may also introduce type pattern variables explicitly (Γ, α) or in type-indexed function bindings $(\Gamma, x\ \langle\alpha\rangle :: \sigma)$, but only if not already bound in environment Γ.*

Lemma 8 (Preservation of typing under substitution). *If $P \mid \Gamma \vdash e :: \sigma$ then for all S we have $SP \mid S\Gamma \vdash e :: S\sigma$.*

Lemma 9 (Preservation of typing under predicate extension). *If* $P \mid \Gamma \vdash e :: \sigma$ *then for all* Q *we have* $P, Q \mid \Gamma \vdash e :: \sigma$.

Lemma 10 (Specification of unification). *If* $t_1 \sim t_2 \equiv S$ *then* $St_1 \equiv St_2$

Lemma 11 (Preservation of type equality under substitution). *If* $t_1 \equiv t_2$ *then* $St_1 \equiv St_2$ *for all* S.

Theorem 4. *If* $P \mid \Gamma \vdash e :: \sigma$, $\Vdash P$ *and inst* $(\sigma) \equiv (S, \sigma')$ *for a well-formed environment* Γ *then* $P \mid \Gamma \vdash e \overset{S}{\rightsquigarrow} e' :: \sigma$.

Proof. We do the proof only for the most interesting cases.

- Case (gapp): From *inst* $(\sigma[A/\alpha]) \equiv (S, \sigma')$ and Lemmma 2 we have that *inst* $(\sigma) \equiv (S', \sigma'')$ such that $\sigma''[A/\alpha] \equiv \sigma'$ and $S'[A/\alpha] \equiv S$. It follows that $P \mid \Gamma \vdash e \overset{S}{\rightsquigarrow} e' :: \sigma[A/\alpha]$.
- Case (qt-let): We proceed to build the E_i expressions with holes for every σ' such that *inst* $(\sigma) \equiv (S, \sigma')$. We use the ind. hyp. with substitution S to obtain $P \mid \Gamma \vdash e \overset{S}{\rightsquigarrow} e''' :: \sigma$. The next ind. hyp. gives us $Q \mid \Gamma, x :: \sigma \vdash e' \overset{\emptyset}{\rightsquigarrow} e'' :: t$.
- Case (qt-glet): We proceed to build the E_j expressions with holes for every σ_j such that *inst* $(\sigma) \equiv (S_j, \sigma_j)$. First, we make explicit the witness substitution for every arm $\sigma \leqslant \sigma_i$ as follows: $\sigma \leqslant_{S_i} \sigma_i$. From Lemma 8 we obtain $S_j S_i P_i \mid S_j S_i \Gamma \vdash e_i :: S_j S_i t_i$. We choose σ such that the quantified variables do not occur in Γ, the same holds for σ_i and thus $S_j S_i \Gamma \equiv \Gamma$; it follows that $S_j S_i P_i \mid \Gamma \vdash e_i :: S_j S_i t_i$. Now applying the ind. hyp. we obtain $S_j S_i P_i \mid \Gamma \vdash e_i \overset{\emptyset}{\rightsquigarrow} e_{ij} :: S_j S_i t_i$ and $Q \mid \Gamma, x \langle \alpha \rangle :: \sigma \vdash e' \overset{\emptyset}{\rightsquigarrow} e'' :: t'$ while the remaining conditions are already available. Note that the ind. hyp. requires a well-formed environment. We enforce well-formedness on the new binding with a well-formedness check on its predicates.
- Case (qt-gletrec): Similar to the previous case.

B Type Inference

Lemma 12 (Definedness of pattern variables in type inference). *If for a well-formed environment* Γ, *expression* e, *type inference* $P \mid S\ \Gamma \vdash^W e :: t; \Delta$ *succeeds, then we have that* $fpv\ (P) \subseteq dpv\ (\Gamma)$, $fpv\ (t) \subseteq dpv\ (\Gamma)$, $fpv\ (\Delta) \subseteq dpv\ (\Gamma)$ *and* $fpv\ (Sa) \subseteq dpv\ (\Gamma)$ *for every* a *such that* $a \in \mathsf{dom}\ (S)$.

Lemma 13 (Definedness of pattern variables in type inference for arms). *If for a well-formed environment* Γ, *arms* $\overline{\Omega_i \to e_i}$, *type inference* $P \mid S\ \Gamma \vdash^W_{Arms(x)} \overline{\Omega_i \to e_i}; \Delta$ *succeeds, then we have that* $fpv\ (P) \subseteq dpv\ (\Gamma)$, $fpv\ (\Delta) \subseteq dpv\ (\Gamma)$ *and* $fpv\ (Sa) \subseteq dpv\ (\Gamma)$ *for every* a *such that* $a \in \mathsf{dom}\ (S)$.

Proof. The proof for most cases of these two theorems is trivial. We outline the proof for the two most interesting cases.

– Case (w-glet): Unification of instance types only introduce the pattern variable α, together with the ind. hyp. we have that $fpv\ (q) \in dpv\ (\Gamma, \alpha)$ and $fpv\ (S_2 a) \in dpv\ (\Gamma, \alpha)$ for $a \in \mathsf{dom}\ (S_2)$. Combined with the definition of generalization, this gives us $fpv\ (\sigma) \in dpv\ (\Gamma, \alpha)$. It is easy to show that the second ind. hyp. gives us the remaining necessary conditions to prove this case.
– Case (w-gletrec): A difference with the previous case is that instance unification introduces pattern variables from type patterns. A result is that pattern variables in q, t and the ranges of S_2 and S_3 do not occur in $dpv\ (\Gamma, \alpha)$. However, the check in the third line ensures that those pattern variables do not escape. It follows that the second ind. hyp. is satisfied and this case holds.

Theorem 5 (Soundness of type inference). *If* $P \mid S\ \Gamma, \overline{x_i \langle \alpha \rangle :: \cdot} \vdash^W e :: t; \Delta$ *and* $inst_\Delta\ (\Delta, \overline{x_i}) \equiv (S', \overline{t_i})$ *then* $S'P \mid S'S\Gamma, \overline{x_i\ \langle \alpha_i \rangle :: t_i} \vdash e_1 :: S't$.

Proof. We do a case analysis for the most interesting cases only:

– Case (w-gapp-1): We have that $x\ \langle \alpha \rangle :: \forall \overline{a_i}\,.\, S'P \Rightarrow S't \in S'\Gamma, \overline{x_i\ \langle \alpha_i \rangle :: t_i}$ since quantified variables are not in the domain of S', it follows by rule (q-gapp) that $S'P[A/\alpha][\overline{b_i}/\overline{a_i}] \mid \Gamma, \overline{x_i\ \langle \alpha_i \rangle :: t_i} \vdash x\ \langle A \rangle :: \forall \overline{a_i}\,.\, S'P[A/\alpha] \Rightarrow S't[A/\alpha]$. We obtain the expected $S'P[A/\alpha][\overline{b_i}/\overline{a_i}] \mid \Gamma, \overline{x_i\ \langle \alpha_i \rangle :: t_i} \vdash x\ \langle A \rangle :: S't[A/\alpha][\overline{b_i}/\overline{a_i}]$ using rules (q-$\forall$E) and (q-$\Rightarrow$E).
– Case (w-gapp-2): We have that $inst_\Delta\ (\Delta; \overline{x_i\ \langle \alpha_i \rangle}, x\ \langle \alpha \rangle) \equiv (S', (\overline{t_i}, t))$ and thus $x\ \langle \alpha \rangle :: t \in S'\Gamma, \overline{x_i\ \langle \alpha_i \rangle :: t_i}, x\ \langle \alpha \rangle :: t$. By rule (q-gapp) it follows that $\emptyset \mid S'\Gamma, \overline{x_i\ \langle \alpha_i \rangle :: t_i}, x\ \langle \alpha \rangle :: t \vdash x\ \langle A \rangle :: t[A/\alpha]$. Finally by the definition of $inst_\Delta$ we have $\emptyset \mid S'\Gamma, \overline{x_i\ \langle \alpha_i \rangle :: t_i}, x\ \langle \alpha \rangle :: t \vdash x\ \langle A \rangle :: S'a$, as desired.
– Case (w-$\rightarrow$E): We need to obtain $S_3'S_3(S_2P, Q) \mid S_3'S_3S_2S_1\Gamma, \overline{x_i \langle \alpha \rangle :: t_i} \vdash e_1\ e_2 :: S_3'S_3 a$ using $S_3S_2P, S_3Q \mid S_3S_2S_1\ \Gamma, \overline{x_i \langle \alpha_i \rangle :: \cdot} \vdash^W e_1\ e_2 :: S_3 a; S_3S_2\Delta_1, S_3\Delta_2$ and $inst_\Delta\ (S_3S_2\Delta_1, S_3\Delta_2; \overline{x_i\ (\alpha_i)}) \equiv (S_3', \overline{t_i})$.
From the first ind. hyp. we have that if $inst_\Delta\ (\Delta_1; \overline{x_i\ (\alpha_i)}) \equiv (S_1', \overline{t_i'})$ then $S_1'P \mid S_1'S_1\Gamma, \overline{x_i\ \langle \alpha_i \rangle :: t_i'} \vdash e_1 :: S_1't_1$ for all S_1'. From Lemma 15, if $inst_\Delta\ (S_3S_2\Delta_1; \overline{x_i\ \langle \alpha_i \rangle}) \equiv (S_1'', \overline{t_i'})$ holds for any S_1'' then we may choose $S_1' \equiv S_1''S_3S_2$ to satisfy the premise of the first ind. hyp. Next, we use Lemma 14 so that satisfaction of $inst_\Delta\ (S_3S_2\Delta_1, S_3\Delta_2; \overline{x_i\ (\alpha_i)}) \equiv (S_3', \overline{t_i})$ implies $inst_\Delta\ (S_3S_2\Delta_1; \overline{x_i\ \langle \alpha_i \rangle}) \equiv (S_1'', \overline{t_i'})$ such that $S_3' \equiv S_1''$ and $\overline{t_i} \equiv \overline{t_i'}$. It follows that we obtain the ind. hyp. premise $inst_\Delta\ (\Delta_1; \overline{x_i\ (\alpha_i)}) \equiv (S_3'S_3S_2, \overline{t_i})$ and thus, $S_3'S_3S_2P \mid S_3'S_3S_2S_1\Gamma, \overline{x_i\ \langle \alpha_i \rangle :: t_i} \vdash e_1 :: S_3'S_3S_2t_1$. Applying Lemma 9, we obtain the desired $S_3'S_3S_2P, S_3'S_3Q \mid S_3'S_3S_2S_1\Gamma, \overline{x_i\ \langle \alpha_i \rangle :: t_i} \vdash e_1 :: S_3'S_3S_2t_1$. Following a similar procedure for the second ind. hyp. we obtain $S_3'S_3S_2P, S_3'S_3Q \mid S_3'S_3S_2S_1\Gamma, \overline{x_i\ \langle \alpha_i \rangle :: t_i} \vdash e_2 :: S_3'S_3t_2$. Using Lemma 10 and Lemma 11 we know that $S_3'S_3S_2t_1 \equiv S_3'S_3t_2 \rightarrow S_3'S_3a$ and thus $S_3'S_3S_2P, S_3'S_3Q \mid S_3'S_3S_2S_1\Gamma, \overline{x_i\ \langle \alpha_i \rangle :: t_i} \vdash e_1 :: S_3'S_3t_2 \rightarrow S_3'S_3a$; which gives the required $S_3'S_3S_2P, S_3'S_3Q \mid S_3'S_3S_2S_1\Gamma, \overline{x_i\ \langle \alpha_i \rangle :: t_i} \vdash e_1\ e_2 :: S_3'S_3a$.
– Case (w-glet): We need to establish $S''Q \mid \Gamma' \vdash \mathbf{let} \ldots e' :: S''t'$ using $inst_\Delta\ (\Delta'', \overline{x_j\ \langle \alpha_j \rangle}) \equiv (S'', \overline{t_j})$ and $Q \mid S\ \Gamma, \overline{x_j \langle \alpha_j \rangle :: \cdot} \vdash^W \mathbf{let} \ldots e' :: t'; \Delta''$ where $S \equiv S_3S_2S_1$ and $\Gamma' \equiv S''S\Gamma, \overline{x_j\ \langle \alpha_j \rangle :: t_j}$.

By Theorem 6 if $inst_\Delta\ (\Delta; \overline{x_j\ \langle\alpha_j\rangle}) \equiv (S_1', \overline{t_j'})$ then $S_1'S_1P_i \mid S_1'S_1\Gamma, \overline{x_j\ \langle\alpha_j\rangle :: t_j'} \vdash e_i :: S_1'S_1t_i$ such that $P \equiv S_1P_i$ and $x\ \langle T_i\rangle \sim S_1t_i \in \Delta$. Using Lemmas 15 and 14, as in the application case, we obtain the required $S''SP_i \mid \Gamma' \vdash e_i :: S''St_i$ and $S''S_3S_2P \equiv S''SP_i$.

Now, type inference for arms gives us $x\ \langle T_i\rangle \sim S_1t_i \in \Delta$, which combining with Lemma 18 gives $\vdash mark^{\alpha}_{T_i}S_1t_i \equiv P_i' \Rightarrow t_i'$ for every T_i and S_1t_i for some P_i' and t_i' such that $t_2 \equiv S_2t_i'$, $P_2 \Vdash S_2P_i'$ and $q \equiv P_2 \Rightarrow t_2$. We use Lemma 19 and preservation of type equality under substitution to infer $\vdash mark^{\alpha}_{T_i}S''St_i \equiv S''S_3S_2(P_i' \Rightarrow t_i')$ (required by q-glet), $S''S_3t_2 \equiv S''S_3S_2t_i'$, $S''S_3P_2 \Vdash S''S_3S_2P_i'$ and $q \equiv (P_2 \Rightarrow t_2)$. We assume $\sigma' \equiv Gen(\Gamma'; S''S_3S_2P \Rightarrow S''S_3q)$ and $\sigma_i \equiv Gen(\Gamma'; S''S_3S_2(S_1P_i, P_i' \Rightarrow t_i'))$. The needed statement $\sigma' \leqslant \sigma_i$ follows trivially from $S''S_3S_2P, S''S_3P_2 \Vdash S''S_3S_2(S_1P_i, P_i')$, $S''S_3t_2 \equiv S''S_3S_2t_i'$ and the fact that free variables of σ_i are also free in σ'.

The ind. hyp. for e' combined with Lemma 8 gives $S''S_3Q \mid \Gamma', x\ \langle\alpha\rangle :: S''S_3\sigma \vdash e' :: S''t'$, as needed. Now, it remains for us to show that $\sigma' \equiv S''S_3\sigma$. We note that S_3 may substitute only free variables of σ, thus $S_3Gen(S_2S_1\Gamma, S_2\Delta_x; S_2P \Rightarrow q)) \equiv Gen(S_3S_2S_1\Gamma, S_3S_2\Delta_x; S_3(S_2P \Rightarrow q)))$. The same argument applies to S'' so that $S''S_3\sigma \equiv Gen(S''S_3S_2S_1\Gamma, S''S_3S_2\Delta_x; S''S_3(S_2P \Rightarrow q)))$. From the definition of $inst_\Delta$ we have that $ftv\ (S''\Delta'') \equiv ftv\ (\overline{t_j})$, we may now write $S''S_3\sigma \equiv Gen(\Gamma'; S''SP \Rightarrow S''q)$ because the free variables of $S''\Delta'$ don't overlap with the quantified variables of $S''S_3\sigma'$. It follows that $S''S_3Q \mid \Gamma', x\ \langle\alpha\rangle :: \sigma' \vdash e' :: S''t'$. The remaining condition for the definedness of pattern variables follows from Lemmas 12 and 13.

– Case (w-gletrec): We need to establish $S''Q \mid \Gamma' \vdash \mathbf{let} \ldots e' :: S''t'$ using $inst_\Delta\ (\Delta''; \overline{x_j\ \langle\alpha_j\rangle}) \equiv (S'', \overline{t_j})$ and $Q \mid S\ \Gamma, \overline{x_j\langle\alpha_j\rangle :: \cdot} \vdash^W \mathbf{let} \ldots e' :: t'; \Delta''$ where $S \equiv S_4S_3S_2S_1$ and $\Gamma' \equiv S''S\Gamma, \overline{x_j\ \langle\alpha_j\rangle :: t_j}$.

By Theorem 6 if $inst_\Delta\ (\Delta; \overline{x_j\ \langle\alpha_j\rangle}, x\ \langle\alpha\rangle) \equiv (S_1', (\overline{t_j'}, t_x))$ then $S_1'S_1P_i \mid S_1'S_1\Gamma, \overline{x_j\ \langle\alpha_j\rangle :: t_j'}, x\ \langle\alpha\rangle :: t_x \vdash e_i :: S_1'S_1t_i$ such that $P \equiv S_1P_i$ and $x\ \langle\Omega_i\rangle \sim S_1t_i \in \Delta$. First, we use Lemma 18 to obtain $\vdash mark^{\alpha}_{A_i}t_k \equiv P_k' \Rightarrow t_k'$ for every $x\ \langle A_k\rangle \sim t_k \in \Delta$ such that $P'' \Vdash S_2P_k'$, $t'' \equiv S_2t_k'$ and $q \equiv P'' \Rightarrow t''$. Next, by the definition of algorithm $inst$, we have that $\Vdash S_3S_2P, S_3P''$ and $S_3t'' \equiv t$. Applying substitutions where necessary and using Lemma 7, we have $S_4S_3S_2t_k \equiv S_4t[A_k\,/\,\alpha]$, giving $inst_\Delta\ (S_4S_3S_2\Delta; x\ \overline{\langle\alpha\rangle}) \equiv (id, S_4t)$ (2). By the definition of $inst_\Delta$, we can write $inst_\Delta\ (\Delta''; \overline{x_j\ \langle\alpha_j\rangle}, x\ \langle\alpha\rangle) \equiv (S'', (\overline{t_j}, S''S_4t))$. Now, applying Lemmas 15 and 14 we obtain $S''SP_i \mid \Gamma', x\ \langle\alpha\rangle :: S''S_4t \vdash e_i :: S''St_i$. Since $x\ (\Omega_i) \sim St_i \in \Delta$, then $\vdash mark^{\alpha}_{A_i}St_i \equiv SP_i' \Rightarrow St_i'$, $\Vdash S''SP_i'$ and $S''St_i \equiv S''S_4t$. We take $\sigma' = Gen(\Gamma'; S''S_4t)$ and $\sigma_i = Gen(\Gamma'; S''S(P_i' \Rightarrow t_i'))$, it follows that $\sigma' \leqslant \sigma_i$ for all σ_i.

We combine the ind.hyp. with instance lemmas as above to obtain $S''Q \mid \Gamma', x\ \langle\alpha\rangle :: S''S_4\sigma \vdash e' :: S''t'$. It remains to show $\sigma' \equiv S''S_4\sigma$. We follow the reasoning of the previous case:

$$
\begin{array}{ll}
S''S_4Gen(S_3S_2S_1\Gamma, S_3S_2\Delta_x; t) & \equiv \\
S''Gen(S\Gamma, S_4S_3S_2\Delta_x; S_4t) & \equiv \\
Gen(S''S\Gamma, S''S_4S_3S_2\Delta_x; S''S_4t) & \equiv
\end{array}
$$

$Gen(\Gamma'; S''S_4 t)$

Next, we use Lemmas 12 and 13 to show that pattern variables in the type scheme are defined. This concludes our proof.

Theorem 6 (Soundness of type inference for arms). *If* $P \mid S\ \Gamma, \overline{x_j\langle\alpha_j\rangle :: \cdot} \vdash^W_{Arms(x)} \overline{\Omega_i \to e_i}; \Delta$ *and* $inst_\Delta\ (\Delta, \overline{x_j\ \langle\alpha_j\rangle}) \equiv (S', \overline{t_j})$ *then for every* e_i *and* Ω_i *there are* t_i *and* P_i *such that* $x\ \langle\Omega_i\rangle \sim t_i \in \Delta$, $S'SP_i \mid S'S\Gamma, \overline{x_j\ \langle\alpha_j\rangle :: t_j} \vdash e_i :: S't_i$ *and* $S'P \equiv \overline{S'SP_i}$ *for some* $\overline{P_i}$.

Proof. By case analysis.

- Case (w-arm-1): Holds trivially.
- Case (w-arm-2): Given $S_2P, P_2 \mid S_2S\ \Gamma, \overline{x_j\langle\alpha_j\rangle :: \cdot} \vdash^W_{Arms(x)} \Omega \to e, \overline{\Omega_i \to e_i}; \Delta_3$ and $inst_\Delta\ (\Delta_3, \overline{x_j\ (\alpha_j)}) \equiv (S', \overline{t_j})$ we want to prove that for every e_i and Ω_i (also e and Ω) there are t_i and P_i such that $x\ \langle\Omega_i\rangle \sim t_i \in \Delta_3$, t and P such that $x\ \langle\Omega\rangle \sim t \in \Delta_3$, $S'P_i \mid S'S_2S\Gamma, \overline{x_j\ \langle\alpha_j\rangle :: t_j} \vdash e_i :: S't_i$, $S'S_2P \mid S'S_2S\Gamma, \overline{x_j\ \langle\alpha_j\rangle :: t_j} \vdash e :: S'S_2t$ and $S_2P, P_2 \equiv S_2P, \overline{P_i}$.
  Using Lemma 5 we obtain that if $P \mid S\ \Gamma, \overline{x_j\langle\alpha\rangle :: \cdot} \vdash^W e :: t; \Delta$ and $inst_\Delta\ (\Delta, \overline{x_j\ \langle\alpha_j\rangle}) \equiv (S'', \overline{t'_j})$ then $S''P \mid S''S\Gamma, \overline{x_j\ \langle\alpha_j\rangle :: t'_j} \vdash e :: S''t$. Using Lemmas 15 and 14 we obtain $S'S_2P \mid S''S_2S\Gamma, \overline{x_j\ \langle\alpha_j\rangle :: t_j} \vdash e :: \overline{S''S_2t}$. Using a similar procedure for the ind. hyp. we obtain $S'P_i \mid S''S_2S\Gamma, \overline{x_j\ \langle\alpha_j\rangle :: t_j} \vdash e_i :: S't_i$ and furthermore $x_i\ \langle\alpha_i\rangle \sim t_i \in \Delta_2$ and $P_2 \equiv \overline{P_i}$ for all arms i.
  We have all the required conditions for the ind. hyp. part of the theorem. It is easy to see that $S_2P, P_2 \equiv S_2P, \overline{P_i}$, $x\ \langle\alpha\rangle \sim S_2t \in \Delta_3$ and $x_i\ \langle\alpha_i\rangle \sim t_i \in \Delta_3$, thereby proving the theorem.

B.1 Properties of unification of instance types

We add the following case to process instance environments that lack a binding for x.

$$\frac{\begin{array}{c}\Delta_x \equiv \Delta \\ a \text{ fresh}\end{array}}{\vdash^x \Delta \rightsquigarrow a; \emptyset}\ \text{(unify-empty)}$$

This case is not strictly needed since the language forbids definitions of generic functions with no cases. Nevertheless, this case is necessary to prove the correctness of the algorithm.

Definition 4 (Instance of an instances environment). *The instance of an instances environment* Δ *is written* $inst_\Delta\ (\Delta; \overline{x_i\ \langle\alpha_i\rangle}) \equiv (S, \overline{t_i})$. *It holds if for all* $x_i\ \langle A_j\rangle \sim t_j \in \Delta$ *we have that* $St_j \equiv t_i[A_j/\alpha_i]$.

Lemma 14. *If* $inst_\Delta\ (\Delta, \Delta'; \overline{x_i\ \langle\alpha_i\rangle}) \equiv (S, \overline{t_i})$ *then* $inst_\Delta\ (\Delta; \overline{x_i\ \langle\alpha_i\rangle} \equiv (S, \overline{t'_i})$.

Lemma 15. *If* $inst_\Delta\ (S\Delta; \overline{x_i\ \langle\alpha\rangle}) \equiv (S', \overline{t'_i})$ *then* $inst_\Delta\ (\Delta; \overline{x_i\ \langle\alpha\rangle}) \equiv (S'S, \overline{t'_i})$.

The two previous lemmas are easily proved using the definition of $inst_\Delta$.

Lemma 16. *If (S', t') is a solution for algorithm inst $(P \Rightarrow t)$ then $\Vdash S'P$ and $S't \equiv t'$.*

Lemma 17. *If $\vdash mark^{\alpha}_{A}\Gamma; St \equiv P' \Rightarrow t'$ and $\Vdash S'P$ then $\vdash mark^{\alpha}_{A}\Gamma; t \equiv P'' \Rightarrow t''$ such that $S'SP''$ and $S't' \equiv S'St''$.*

Lemma 18 (Properties of instance unification). *If $\vdash^{x\ \langle\alpha\rangle} \Delta \rightsquigarrow P \Rightarrow t; S$ and $x\ \langle A_i\rangle \sim t_i \in \Delta$ then $\vdash mark^{\alpha}_{A_i} t_i \equiv P_i \Rightarrow t'_i$ for every A_i and t_i such that $t \equiv St'_i$, $P \equiv SP_i$, $\sigma \equiv Gen(\Gamma; P \Rightarrow t)$ and $\vdash^{pred}_{\alpha} \sigma$ for all Γ and some $\overline{P_i}$ and $\overline{t_i}$.*

Proof. Initially, all qualified types $P_i \Rightarrow t'_i$ are well formed if generalized. That is because the introduced variables are fresh and the types t_i do not have an occurrence of α. After unification, we have three sets of predicates, well-formed predicates, predicates with the fresh variable replaced by a type constant and predicates sharing the fresh variable. The compound algorithm eliminates the last set of predicates by replacing each group of sharing predicates by a single one that entails the group. Note that at this point predicates with consistent equations are well-formed.

We continue the proof by showing that simplification gets rid of non-well-formed predicates, keeping the well-formed ones. The main algorithm is, in fact, *simplify′*. We maintain two invariants. First, the second parameter holds well-formed predicates. The other invariant for the first parameter is that predicates with consistent equations are well-formed. We do a case analysis:

- One of the left equations is inconsistent. We must apply the substitutions obtained by the unifier to the rest of the predicates. Remember that the fresh variables (if any) occur in only one predicate. Thus only variables occuring in the original type are substituted in the other predicates. It follows that well-formed predicates are still well-formed.
- One of the right equations is inconsistent. By the same argument as above, fresh variables in the other predicates are left untouched. Moreover, since the type argument α is equated to a fresh variable, no α is added to the other predicates. It follows that well-formedness of the other predicates is preserved.
- Predicate with consistent equations. Having consistent equations, the current predicate is well-formed.
- No more predicates. This case returns a qualified type with well-formed predicates.

The remaining properties of the algorithm follow easily from the definitions of the used algorithms.

Lemma 19 (Preservation of marking under substitution). *If $\vdash mark^{\alpha}_{A} t \equiv q$ then for all substitutions S we have that $\vdash mark^{\alpha}_{A} St \equiv Sq$.*

B.2 Most general unifier

The standard most general unifier algorithm:

$$
\begin{array}{ll}
mgu\ a\ a & = id \\
mgu\ a\ t & = \{\, a \mapsto t \,\} \\
\quad \textbf{where}\ a \text{ does not occur in } t & \\
mgu\ t\ a & = \{\, a \mapsto t \,\} \\
\quad \textbf{where}\ a \text{ does not occur in } t & \\
mgu\ \alpha\ \alpha & = id \\
mgu\ (T\ \overline{t_i})\ (T\ \overline{t'_i}) & = \overline{mgu\ t_i\ t'_i}
\end{array}
$$