Type Reconstruction for Type Classes[1]

Tobias Nipkow[2] and Christian Prehofer[3]
TU München[4]

Abstract

We study the type inference problem for a system with type classes as in the functional programming language Haskell. Type classes are an extension of ML-style polymorphism with overloading. We generalize Milner's work on polymorphism by introducing a separate context constraining the type variables in a typing judgement. This leads to simple type inference systems and algorithms which closely resemble those for ML. In particular we present a new unification algorithm which is an extension of syntactic unification with constraint solving. The existence of principal types follows from an analysis of this unification algorithm.

1 Introduction

The extension of Hindley/Damas/Milner polymorphism with the notion of *type classes* in the functional programming language Haskell (HJW92) has attracted much attention. Type classes permit the systematic overloading of function names while retaining the advantages of the Hindley/Damas/Milner system: every typable expression has a most general type which can be inferred automatically. Although many extensions to Haskell's type system have already been proposed (and also implemented), we believe that the *essence* of Haskell's type inference algorithm has still not been presented in all its simplicity. The main purpose of this paper is to give a particularly simple algorithm, a contribution for implementors. At the same time we present a correspondingly simple type inference system, a contribution aimed at users of the language. Finally we give rigorous proofs of the soundness and completeness of the algorithm with respect to the inference system. Although both the algorithm and the inference system resemble their ML-counterparts very closely, the proofs are considerably more involved.

A type class in Haskell is essentially a set of types (which all happen to provide a certain set of functions). The classical example is equality. In the pre-standard versions of ML, the equality function = has the polymorphic type $\forall\alpha.\alpha \to \alpha \to bool$,

[1] This is an extended version of (NP93)
[2] Research supported by ESPRIT BRA 6453, *TYPES*.
[3] Research supported by the Deutsche Forschungsgemeinschaft (DFG) under grant Br 887/4, *Deduktive Programmentwicklung.*
[4] Address: Institut für Informatik, Technische Universität München, 80290 München, Germany. Email: `{nipkow,prehofer}@informatik.tu-muenchen.de`

where the type variable α ranges over all types. However, $=$ should not be applied to arguments of function type. To fix this problem, Standard ML (MTH90) introduces special type variables that range only over types where equality is defined. Equality differs from other polymorphic functions not just because of its restricted domain but also because of its mixture of polymorphism and overloading: equality on lists is implemented differently from equality on integers.

Type classes treat both issues in a systematic way: the type variable α is restricted to elements of a certain type class, say Eq, the class of all "equality types". Then for each type τ where $=$ should be defined, we have to declare that τ is of class Eq by providing an implementation of $=$ of type $\tau \to \tau \to bool$.

To express the fact that a type τ is in some class C we introduce the judgement $\tau : C$.[1] The idea of viewing Haskell as a three level system of expressions, types and classes, where classes classify types, goes back to Nipkow and Snelting (NS91). However, in their system it is impossible to express that a type belongs to more than one class. To overcome this difficulty we introduce *sorts* as finite sets of classes. The judgement $\tau : \{C_1, \ldots, C_n\}$ is a compact form of the conjunction $\tau : C_1 \wedge \ldots \wedge \tau : C_n$. Alternatively we may think of $\{C_1, \ldots, C_n\}$ as a notation for $C_1 \cap \ldots \cap C_n$, the intersection of the types belonging to the classes C_1 to C_n. This leads to a simple type inference system and algorithm. The former resembles that for Mini-ML (CDDK86), the latter is very similar to algorithm $\mathcal{I}$ by Milner (Mil78). The main difference is that in both cases we also compute a set of constraints of the form $\alpha : \{C_1, \ldots, C_n\}$ where α is a type variable.

2 Mini-Haskell

Since the aim of this paper is simplicity, we treat only the most essential features of Haskell relating to type classes. The resulting language is basically Mini-ML (CDDK86) plus *class* and *instance* declarations, *Mini-Haskell* for short. Its syntax is shown in Figure 1. Although the next paragraph provides a brief account of type classes, the reader should consult the Haskell Report (HJW92) or the original paper on type classes (WB89) for motivations and examples. Note that we do not follow the concrete names of classes etc. of Haskell in our examples.

Mini-Haskell extends ML by a restricted form of overloading. Ignoring subclasses for a moment, each class declaration introduces a new class C and a new overloaded function name x. Semantically, C represents the set of all types which support a function x. For instance

$$\textbf{class } \alpha : Eq \textbf{ where } eq : \alpha \to \alpha \to bool$$

introduces the class Eq of all those types τ which provide a function $eq : \tau \to \tau \to bool$. A class declaration is like a module interface: it separates declarations from implementations. In order to "prove" that a particular type, say int, is in Eq, a "witness" for the required function eq needs to be provided. This is the purpose of

[1] If classes are viewed as predicates on types, this leads to the Haskell notation $C(\tau)$.

Type classes	C		
Sorts	S	$=$	$\{C_1, \ldots, C_n\}$
Type variables	α		
Type constructors	t		
Types	τ	$=$	$\alpha \mid t(\tau_1, \ldots, \tau_n)$
Type schemes	σ	$=$	$\tau \mid \forall\alpha{:}S.\sigma$
Identifiers	x		
Expressions	e	$=$	x
		$\mid$	$(e_0\ e_1)$
		$\mid$	$\lambda x.e$
		$\mid$	$\textbf{let}\ x = e_0\ \textbf{in}\ e_1$
Declarations	d	$=$	$\textbf{class}\ \alpha : C \leq S\ \textbf{where}\ x : \sigma$
		$\mid$	$\textbf{inst}\ t : (S_1, \ldots, S_n)C\ \textbf{where}\ x = e$
Programs	p	$=$	$d; p \mid e$

Fig. 1. Syntax of Mini-Haskell types and expressions

instance declarations. In order to prove $int : Eq$ we instantiate eq by eq_int, some existing function of type $int \rightarrow int \rightarrow bool$:

$$\textbf{inst}\ int : Eq\ \textbf{where}\ eq = eq\_int$$

In general we can instantiate classes not just by ground types but also by type constructors. For example we may wish to express that a type $list(\tau)$ admits equality provided τ does:

$$\textbf{inst}\ list : (Eq)Eq\ \textbf{where}\ eq = \ldots$$

The declaration $list : (Eq)Eq$ expresses that $list$ maps types of class Eq to types of class Eq. The implementation of eq on lists is intentionally left blank: due to the absence of pattern matching and recursion in our language, the required code would be a nest of conditionals wrapped up in a fixpoint combinator.

Classes can be arranged in hierarchies. The general class declaration

$$\textbf{class}\ \alpha : C \leq S\ \textbf{where}\ x : \sigma$$

introduces the new class C as a subclass of all classes in S, which must have been defined already. Type α is in C only if it is in the intersection of all the classes in S and provides a function x of type σ. For example the class Ord of *ordered types* can be defined as a subclass of Eq which provides an additional function le:

$$\textbf{class}\ \alpha : Ord \leq Eq\ \textbf{where}\ le : \alpha \rightarrow \alpha \rightarrow bool$$

Subclasses are mere syntactic sugar (CHO92). In the above example Ord could be defined without reference to Eq as a completely separate class. The only difference is that without subclasses the judgement $\tau : Ord$ has to be expanded to become $\tau : Eq \wedge \tau : Ord$, i.e. $\tau : \{Eq, Ord\}$. However, it is almost easier to deal with subclasses directly than to eliminate them, as done in (NP93). To demonstrate this, and because subclasses are part of Haskell, we have included them in Mini-Haskell.

2.1 Sorts and Types

As motivated in the introduction, **sorts** are finite sets of classes. This representation is a key ingredient for the concise treatment of type inference. Yet semantically the sort $\{C_1, \ldots, C_n\}$ should be understood as $C_1 \cap \ldots \cap C_n$. Thus $\{C\}$ and C are equivalent, and the empty set $\{\}$ is the sort/set of all types. If S_1 is more specialized, i.e. represents fewer types than S_2, we write $S_1 \preceq S_2$. Given a partial order $\leq$ on classes, the induced quasi-order $\preceq$ on sorts is defined by

$$S_1 \preceq S_2 \quad \Leftrightarrow \quad \forall C_2 \in S_2. \exists C_1 \in S_1.\ C_1 \leq C_2$$

It follows directly that $S_1 \supseteq S_2$ implies $S_1 \preceq S_2$. In the context of a non-trivial ordering $\leq$ on classes, the reverse implication does not hold: for example $\{Ord\} \preceq \{Eq\}$ although $\{Ord\} \not\supseteq \{Eq\}$. It is easy to see that any two sorts S_1 and S_2 possess an infimum whose representation is their union $S_1 \cup S_2$.

Because $\preceq$ is in general only a quasi-order (i.e. it is not antisymmetric), it gives rise to an equivalence

$$S_1 \approx S_2 \quad \Leftrightarrow \quad S_1 \preceq S_2 \wedge S_2 \preceq S_1.$$

Sorts which are equivalent modulo $\approx$, for example $\{Ord\}$ and $\{Ord, Eq\}$, represent the same set of types. Although it would be mathematically more elegant to work with equivalence classes $[S]_\approx$, we prefer to stay closer to an implementation and work with sorts directly. Nevertheless it should be kept in mind that an implementation is free to choose an arbitrary representative from an equivalence class $[S]_\approx$, for example the one with fewest elements.

Types in Mini-Haskell are simply terms over variables and constructors of fixed arity. Note that $\rightarrow$ is just another type constructor, i.e. $\tau_1 \rightarrow \tau_2$ is short for $\rightarrow(\tau_1, \tau_2)$. The set of free variables in a type scheme is denoted by $\mathcal{FV}(\sigma)$. Bound variables in type schemes range only over certain subsets of types: $\forall\alpha{:}S.\sigma$ abbreviates all instances $\{\alpha \mapsto \tau\}\sigma$ where $\tau : S$, a judgment defined formally below.

In the sequel a list of syntactic objects $s_1, \ldots, s_n$ is abbreviated by $\overline{s_n}$. For instance, $\forall\overline{\alpha_n{:}S_n}.\sigma$ is equivalent with $\forall\alpha_1{:}S_1, \ldots, \alpha_n{:}S_n.\sigma$. Orderings extend to lists in the componentwise manner: $\overline{S_n} \preceq \overline{T_n} \ \Leftrightarrow\ \forall i.\ S_i \preceq T_i$.

2.2 Declarations and Programs

As shown in Figure 1, expressions are λ-terms extended with **let**-definitions. A program is a sequence of declarations followed by an expression.

A Mini-Haskell class declaration **class** $\alpha : C \leq \{C_1, \ldots, C_m\}$ **where** $x : \sigma$ corresponds to the Haskell declaration **class** $(C_1\alpha, \ldots, C_m\alpha) \Rightarrow C\alpha$ **where** $x :: \tau$, where τ is the body of σ. Note that σ must contain no free variables except α. The translation in the opposite direction is more involved because a Haskell class can declare any number of functions. This feature is clearly not essential and could, for instance, be modeled by representing a set of functions by a single tuple of functions. Strictly speaking, we could have dropped class names altogether since there is a one to one correspondence between class names and the single function declared in that class. This would have lead us to the language of Stefan Kaes (Kae88) but would have obscured the connection with Haskell.

A Mini-Haskell instance declaration **inst** $t : (S_1, \ldots, S_n)C$ **where** $x = e$ expresses that $t(\tau_1, \ldots \tau_n)$ is in class C provided the τ_i are of sort S_i. It corresponds to the Haskell declaration **inst** $(con) \Rightarrow C(t\,\alpha_1 \ldots \alpha_n)$ **where** $x = e$ where con is a list consisting of assumptions $C'\,\alpha_i$ with $C' \in S_i$ for all $i = 1 \ldots n$.

2.3 Classifying Types

Before we embark on type inference, the simpler problem of sort inference has to be settled. In ML and many other languages we have the judgement $e : \tau$, expressing that e is of type τ. Similarly, we classify types by sorts with the judgement $\tau : S$, stating that type τ is in sort S. This judgement depends on

- the sorts of the type variables in τ. This is recorded in a **sort context** Γ, which is a total mapping from type variables to sorts such that $\mathcal{D}om(\Gamma) = \{\alpha \mid \Gamma\alpha \neq \{\}\}$ is finite. Sort contexts can be written as $[\alpha_1{:}S_1, \ldots, \alpha_n{:}S_n]$.
- the "functionality" of the type constructors. The behaviour of type constructors is specified by declarations of the form $t : (\overline{S_n})C$ which are lifted directly from instance declarations. In the sequel Δ always denotes a set of such declarations.
- the subclass ordering $\leq$.

The pair $\Delta, \leq$ is called a **(type) signature** and is denoted by Σ, i.e. $\Delta, \leq$ and Σ are used interchangeably.

Given Γ and Σ we can infer the sort of a type τ using the judgement $\Sigma, \Gamma \vdash \tau : S$. The rules are shown in Figure 2. Remember that the sort $\{C\}$ and the class C are equivalent.

The ordering $\preceq$ extends easily from sorts to contexts:

$$\Gamma \preceq \Gamma' \;\Leftrightarrow\; \forall\alpha.\, \Gamma\alpha \preceq \Gamma'\alpha$$

We say that Γ' is **more general** than Γ. It is easy to show that $\vdash$ is monotonic w.r.t. this ordering: $\Sigma, \Gamma' \vdash \tau : S$ implies $\Sigma, \Gamma \vdash \tau : S$. In the sequel this fact is often used implicitly.

Because every two sorts possess an infimum, every type τ has a most specific sort S, i.e. $\Sigma, \Gamma \vdash \tau : S$ and if $\Sigma, \Gamma \vdash \tau : S'$ then $S \preceq S'$. The computation of this most specific sort is straightforward and shall not concern us here because it is not relevant for our purposes.

Having seen sort inference for Mini-Haskell types we are prepared for our main goal, type inference and type reconstruction for Mini-Haskell programs.

3 Type Inference Systems

In this section we present two type inference systems for Mini-Haskell. We start with a set of inference rules which define the types of Mini-Haskell programs and

$$\frac{\begin{array}{c}[i=1\dots n]\\ \vdots \\ \Sigma,\Gamma\vdash\tau:C_i\end{array}}{\Sigma,\Gamma\vdash\tau:\{C_1,\dots,C_n\}}$$

$$\frac{\Sigma,\Gamma\vdash\tau:\{C_1,\dots,C_n\}}{\Sigma,\Gamma\vdash\tau:C_i}\qquad i=1\dots n$$

$$\frac{\Gamma(\alpha)=S}{\Sigma,\Gamma\vdash\alpha:S}$$

$$\frac{t:(\overline{S_n})C\in\Delta\qquad \begin{array}{c}[i=1\dots n]\\ \vdots \\ \Sigma,\Gamma\vdash\tau_i:S_i\end{array}}{\Sigma,\Gamma\vdash t(\overline{\tau_n}):C}$$

$$\frac{\Sigma,\Gamma\vdash\tau:C_1\quad C_1\le C_2}{\Sigma,\Gamma\vdash\tau:C_2}$$

Fig. 2. The judgement $\Sigma,\Gamma\vdash\tau:S$

$$\text{CLASS}\qquad\frac{\Delta,(\le\cup\{(C,D)\mid D\in S\})^*,\Gamma,E[x{:}\forall\alpha{:}C.\sigma]\vdash p:\sigma'}{\Delta,\le,\Gamma,E\vdash(\textbf{class}\ \alpha:C\le S\ \textbf{where}\ x:\sigma;p):\sigma'}$$

$$\text{INST}\qquad\frac{\begin{array}{c}\Delta\cup\{t:(\overline{S_n})C\},\le,\Gamma,E\vdash p:\sigma'\\ E(x)=\forall\alpha{:}C.\sigma\quad \Gamma[\overline{\alpha_n{:}S_n}],\Delta,\le,E\vdash e:\{\alpha\mapsto t(\overline{\alpha_n})\}\sigma\end{array}}{\Delta,\le,\Gamma,E\vdash(\textbf{inst}\ t:(\overline{S_n})C\ \textbf{where}\ x=e;p):\sigma'}$$

Fig. 3. The judgement $\Delta,\le,\Gamma,E\vdash p:\sigma$

expressions. Then we proceed to a more restricted, syntax-directed set of rules, which will be the basis for the type inference algorithm.

As usual in type inference for ML-like languages, an **environment** is a finite mapping $E=[x_1{:}\sigma_1,\dots,x_n{:}\sigma_n]$ from identifiers to types. The **domain** of E is $\mathcal{D}om(E)=\{x_1,\dots,x_n\}$. $E[x{:}\sigma]$ is a new map which maps x to σ and all other x_i to σ_i. The free type variables in E are $\mathcal{FV}(E)=\mathcal{FV}(E(x_1))\cup\dots\cup\mathcal{FV}(E(x_n))$. If V is a set of type variables the restriction of Γ to variables not in V is $\Gamma\backslash V=[\alpha{:}\Gamma\alpha\mid\alpha\in\mathcal{D}om(\Gamma)-V]$.

A **substitution** is a finite mapping from type variables to types, written as $\{\alpha_1\mapsto\tau_1,\dots\}$. Substitutions are denoted by θ and δ; $\{\}$ is the empty substitution. Define $\mathcal{D}om(\theta)=\{\alpha\mid\theta\alpha\ne\alpha\}$, $\mathcal{C}od(\theta)=\bigcup_{\alpha\in\mathcal{D}om(\theta)}\mathcal{FV}(\theta(\alpha))$ and $\mathcal{FV}(\theta)=\mathcal{D}om(\theta)\cup\mathcal{C}od(\theta)$.

There are two judgements which are defined in Figures 3 and 4: $\Delta,\le,\Gamma,E\vdash p:\sigma$ and $\Delta,\le,\Gamma,E\vdash e:\sigma$ express that program p and expression e are of type σ in the context of Δ, $\le$, Γ and E.

The rules for $\Delta,\le,\Gamma,E\vdash p:\sigma$, when applied backwards, simply traverse the declarations, building up Δ, $\leq$ and E. Class declarations extend E and $\leq$, instance declarations extend Δ. Notice that it is necessary to take the transitive closure $(\leq \cup \{(C, D) \mid D \in S\})^*$ of $\leq$ and the new subclass relations in rule CLASS.

Rule INST also type-checks the instantiation of x by e, making sure that e is of type $\{\alpha \mapsto t(\overline{\alpha_n})\}\sigma$, where σ is the generic type of x and $\{\alpha \mapsto t(\overline{\alpha_n})\}$ is a type substitution with new type variables $\overline{\alpha_n}$.

Note that there are two context conditions for declaration sequences we have chosen not to formalize:

1. **class** $\alpha : C \leq S$ must be preceded by a declaration for each superclass in S, but not by another declaration **class** $\alpha : C$;
2. **inst** $t : (\overline{S_n})C$ must be preceded, for each superclass D of C, by a declaration **inst** $t : (\overline{T_n})D$ such that $\overline{S_n} \preceq \overline{T_n}$, but not by another declaration **inst** $t(\ldots)C$.

These conditions are the result of translating the restrictions actually adopted in Haskell (HJW92, 4.3.2) to Mini-Haskell. Enforcing them is simple enough and has thus been ignored in this paper. Nevertheless we assume in the sequel that all declarations, and hence Δ and $\leq$, meet the above conditions.

$$\begin{array}{ll}
\text{ASM} & \overline{\Sigma, \Gamma, E \vdash x : E(x)} \\[2ex]
\forall\text{E} & \dfrac{\Sigma, \Gamma, E \vdash e : \forall\alpha{:}S.\sigma \quad \Sigma, \Gamma \vdash \tau : S}{\Sigma, \Gamma, E \vdash e : \{\alpha \mapsto \tau\}\sigma} \\[2ex]
\forall\text{I} & \dfrac{\Sigma, \Gamma[\alpha{:}S], E \vdash e : \sigma \quad \alpha \in \mathcal{FV}(\sigma) - \mathcal{FV}(E)}{\Sigma, \Gamma, E \vdash e : \forall\alpha{:}S.\sigma} \\[2ex]
\text{APP} & \dfrac{\Sigma, \Gamma, E \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Sigma, \Gamma, E \vdash e_2 : \tau_2}{\Sigma, \Gamma, E \vdash (e_1\ e_2) : \tau_1} \\[2ex]
\text{ABS} & \dfrac{\Sigma, \Gamma, E[x{:}\tau_1] \vdash e : \tau_2}{\Sigma, \Gamma, E \vdash \lambda x.e : \tau_1 \rightarrow \tau_2} \\[2ex]
\text{LET} & \dfrac{\Sigma, \Gamma, E \vdash e_1 : \sigma_1 \quad \Sigma, \Gamma, E[x{:}\sigma_1] \vdash e_2 : \sigma_2}{\Sigma, \Gamma, E \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \sigma_2}
\end{array}$$

Fig. 4. The judgement $\Sigma, \Gamma, E \vdash e : \sigma$

The rules for $\Sigma, \Gamma, E \vdash e : \sigma$ extend the classical system of Damas and Milner (DM82) by the notion of sorts, which are represented via Σ, Γ, and restricted quantification in type schemes. The assumption $\alpha \in \mathcal{FV}(\sigma)$ in $\forall$I is not really essential (for soundness). Its practical significance is discussed in Section 8. In contrast to the CLASS and INST rules, Σ remains fixed.

3.1 Syntax-directed Type Inference

The next step towards a type reconstruction algorithm is a more restricted set of rules. The application of these rules is determined by the syntax of the expression whose type is to be computed. To distinguish the syntax-directed system we use $\rhd$ instead of $\vdash$ and prime the names of its rules, e.g. ASM$'$.

Definition 3.1
The type scheme $\sigma' = \forall\overline{\alpha'_n{:}S'_n}.\tau'$ is a **generic instance** of $\sigma = \forall\overline{\alpha_m{:}S_m}.\tau$ under Σ and Γ, written $\Sigma,\Gamma \vdash \sigma \succeq \sigma'$, iff there exists a substitution θ such that

$$\begin{array}{rcl} \theta\tau &=& \tau', \\ \mathcal{D}om(\theta) &\subseteq& \{\overline{\alpha_m}\}, \\ \Sigma,\Gamma[\overline{\alpha'_n{:}S'_n}] &\vdash& \theta\alpha_i : S_i \quad [i = 1\ldots m], \\ \{\overline{\alpha'_n}\} \cap \mathcal{FV}(\sigma) &=& \{\}. \end{array}$$

With this relation on types we can now define the most general or principal type of an expression. We say E is **closed** if $\mathcal{FV}(E) = \{\}$.

Definition 3.2
The type scheme σ is a **principal type** of an expression e w.r.t. Σ and a closed environment E, if $\Sigma,[],E \vdash e : \sigma$ and for every σ' with $\Sigma,[],E \vdash e : \sigma'$, the type scheme σ' must be a generic instance of σ, i.e. $\Sigma,[] \vdash \sigma \succeq \sigma'$.

For the syntax-directed system, the rules APP and ABS remain unchanged, the quantifier rules are incorporated into ASM and LET, as shown in Figure 5.

$$\text{ASM}' \quad \frac{\Sigma,\Gamma \vdash E(x) \succeq \tau}{\Sigma,\Gamma,E \rhd x : \tau}$$

$$\text{LET}' \quad \frac{\Sigma,\Gamma[\overline{\alpha_k{:}S_k}],E \rhd e_1 : \tau_1 \quad \Sigma,\Gamma,E[x{:}\forall\overline{\alpha_k{:}S_k}.\tau_1] \rhd e_2 : \tau_2}{\Sigma,\Gamma,E \rhd \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}$$
$$\text{where } \{\overline{\alpha_k}\} = \mathcal{FV}(\tau_1) - \mathcal{FV}(E)$$

Fig. 5. The judgement $\Sigma,\Gamma,E \rhd e : \sigma$

There is a straightforward correspondence between the two systems. The syntax-directed derivations are sound

Theorem 3.3
If $\Sigma,\Gamma,E \rhd e : \tau$ then $\Sigma,\Gamma,E \vdash e : \tau$.

and in a certain sense complete w.r.t. the original system:

Theorem 3.4
If $\Sigma,\Gamma,E \vdash e : \forall\overline{\alpha_n{:}S_n}.\tau$ then $\Sigma,\Gamma[\overline{\alpha_n{:}S_n}],E \rhd e : \tau$.

The proof of the last theorem is standard, as for instance in (CDDK86, App. A.1). Theorem 3.4 clarifies in what sense $\rhd$ works differently from $\vdash$: by applying the primed rules backwards, the sort constraints for type variables are stored solely in Γ, and not in the type scheme of e. For instance, the LET$'$ rule explicitly extends Γ. The $\succeq$ operation, used in the ASM$'$ rule, may introduce new type variables, whose sorts must be constrained in Γ.

The syntax-directed system already has a very operational flavour. In order to make the transition from a type inference system to an algorithm we need one more ingredient: unification.

4 Unification of Types with Sort Constraints

This section deals with unification in the presence of sort constraints in the form of contexts. This problem can in principle be reduced to order-sorted unification, as done in (NS91) w.r.t. $\preceq$. However, we have refrained from doing so because it is contrary to our quest for simplicity: involving order-sorted unification makes the algorithm appear more complicated than it actually is. In addition, the standard theory of order-sorted unification would need to be reformulated anyway: it assumes that variables are tagged with their sort, rather than using contexts.

For the remainder of this paper we assume a fixed signature $\Sigma = (\Delta, \leq)$. This is simply a notational device which avoids excessive parameterization.

Since sort information is maintained in contexts, we frequently work with pairs of contexts and substitutions. A substitution θ obeys the sort constraints of Γ in the context of Γ', written $\Gamma' \vdash \theta : \Gamma$, iff $\Sigma, \Gamma' \vdash \theta\alpha : \Gamma\alpha$ for all α. Because $\Sigma, \Gamma' \vdash \theta\alpha : \Gamma\alpha$ is trivially fulfilled if $\Gamma\alpha = \{\}$ it suffices to require $\Sigma, \Gamma' \vdash \theta\alpha : \Gamma\alpha$ for all $\alpha \in \mathcal{D}om(\Gamma)$. For instance, let Eq and $list$ be defined as in the examples in Section 2. Then we have $[\beta{:}Eq] \vdash \{\alpha \mapsto list(\beta)\} : [\alpha{:}Eq]$.

We define an ordering on context-substitution pairs:

$$(\Gamma, \theta) \geq (\Gamma', \theta') \quad\Leftrightarrow\quad \exists\delta.\ \delta\theta = \theta' \ \wedge\ \Gamma' \vdash \delta : \Gamma$$

where $\delta\theta$ is defined as the composition: $(\delta\theta)(s) = \delta(\theta(s))$.

The set of unifiers of τ_1 and τ_2 w.r.t. Γ, written $\mathcal{U}(\Gamma, \tau_1{=}\tau_2)$, consists of the following context-substitution pairs:

$$\mathcal{U}(\Gamma, \tau_1{=}\tau_2) = \{(\Gamma', \theta) \mid \theta\tau_1 = \theta\tau_2 \ \wedge\ \Gamma' \vdash \theta : \Gamma\}$$

A unifier $(\Gamma_0, \theta_0) \in \mathcal{U}(\Gamma, \tau_1{=}\tau_2)$ is **most general** if $(\Gamma_0, \theta_0) \geq (\Gamma_1, \theta_1)$ for all $(\Gamma_1, \theta_1) \in \mathcal{U}(\Gamma, \tau_1{=}\tau_2)$. We say that unification modulo Σ is **unitary** if for all Γ and $\tau_1{=}\tau_2$ the set $\mathcal{U}(\Gamma, \tau_1{=}\tau_2)$ is empty or contains a most general unifier.

A signature Σ is called **coregular** if for all type constructors t and all classes C the set

$$D(t, C) = \{\overline{S_n} \mid \exists D \leq C.\ (t : (\overline{S_n})D) \in \Delta\}$$

is either empty or contains a greatest element w.r.t. $\preceq$. If Σ is coregular let $\mathcal{D}om(t, C)$ return the greatest element of $D(t, C)$ or fail if $D(t, C)$ is empty. For instance, $\mathcal{D}om(list, Eq) = Eq$ but $\mathcal{D}om(list, Ord)$ fails.

Sorted unification can be expressed as unsorted unification plus constraint solving. Given a coregular signature Σ, this has the following simple form:

```
$unify(\Gamma, \tau_1{=}\tau_2)$  =
  let $\theta$    =   $mgu(\tau_1{=}\tau_2)$
      $\Gamma_c$   =   $Constrain(\theta, \Gamma)$
  in $(\Gamma_c \cup (\Gamma \backslash \mathcal{D}om(\theta)), \theta)$
```

where

- mgu computes an unsorted mgu (in particular we assume that θ is idempotent and that $\mathcal{D}om(\theta) \cup \mathcal{C}od(\theta) \subseteq \mathcal{FV}(\tau_1{=}\tau_2)$) or fails if none exists,
- the union of two sort contexts is defined by

$$\Gamma_1 \cup \Gamma_2 \;=\; [\alpha : \Gamma_1\alpha \cup \Gamma_2\alpha \mid \alpha \in \mathcal{D}om(\Gamma_1) \cup \mathcal{D}om(\Gamma_2)]$$

- $Constrain(\theta, \Gamma)$ computes the most general context Γ_c such that $\Gamma_c \vdash \theta : \Gamma$:

$$Constrain(\theta,\Gamma) = \bigcup_{\alpha \in \mathcal{D}om(\theta)} constrain(\theta\alpha, \Gamma\alpha)$$

- $constrain(\tau, S)$ computes the most general context Γ such that $\Sigma, \Gamma \vdash \tau : S$:

$$\begin{aligned} constrain(\alpha, S) &= [\alpha{:}S] \\ constrain(t(\overline{\tau_n}), S) &= \bigcup_{C \in S} constrains(\overline{\tau_n}, \mathcal{D}om(t, C)) \\ constrains(\overline{\tau_n}, \overline{S_n}) &= \bigcup_{i=1\ldots n} constrain(\tau_i, S_i) \end{aligned}$$

Thus *unify* fails if mgu fails or if some $\mathcal{D}om(t, C)$ used in *Constrain* does not exist. By induction on the first argument of *constrain* it can be shown that

$$constrain(t(\overline{\tau_n}), S) = constrains(\overline{\tau_n}, \bigcup_{C \in S} \mathcal{D}om(t, C))$$

which provides an alternative definition of *constrain* which is also useful in the proofs below. To see how *constrain* works, assume Eq and $list$ again as in the examples in Section 2. Then $constrain(list(\beta), Eq) = constrains(\beta, \mathcal{D}om(list, Eq)) = [\beta{:}Eq]$.

Soundness and completeness of *Constrain* are captured by the following lemmas which assume coregularity of Σ and are proved by induction on the structure of τ:

Lemma 4.1
If $constrain(\tau, S)$ is defined then $\Sigma, constrain(\tau, S) \vdash \tau : S$.

Lemma 4.2
If $constrain(\theta\tau, S)$ is defined then $constrain(\tau, S)$ is defined as well and furthermore $constrain(\theta\tau, S) \vdash \theta : constrain(\tau, S)$ holds.

Lemma 4.3
If $\Sigma, \Gamma \vdash \tau : S$ then $constrain(\tau, S)$ is defined and more general than Γ.

Finally, the main theorems:

Theorem 4.4
If Σ is coregular, *unify* computes a most general unifier.

Proof To show soundness, let $unify(\Gamma, \tau_1{=}\tau_2)$ terminate with result (Γ_0, θ_0). It follows directly that $\theta_0\tau_1 = \theta_0\tau_2$. It remains to be seen that $\Gamma_0 \vdash \theta_0\alpha : \Gamma\alpha$ for all α. If $\alpha \notin \mathcal{D}om(\theta_0)$, then $\Gamma\alpha \subseteq \Gamma_0\alpha$ and the claim follows trivially. If $\alpha \in \mathcal{D}om(\theta_0)$ then $\Gamma_0 \preceq \Gamma_c = Constrain(\theta_0, \Gamma) \preceq constrain(\theta_0\alpha, \Gamma\alpha)$ and the claim follows from Lemma 4.1.

To show completeness let $(\Gamma_1, \theta_1) \in \mathcal{U}(\Gamma, \tau_1{=}\tau_2)$, i.e. $\theta_1\tau_1 = \theta_1\tau_2$ and $\Gamma_1 \vdash \theta_1 : \Gamma$. Since τ_1 and τ_2 have an unsorted unifier θ_1, $mgu(\tau_1{=}\tau_2)$ is defined and yields a substitution θ_0 such that $\theta_1 = \delta\theta_0$ for some δ. Definedness of $\mathit{unify}(\Gamma, \tau_1{=}\tau_2)$ also requires definedness of $constrain(\theta_0\alpha, \Gamma\alpha)$: since $\Gamma_1 \vdash \theta_1\alpha : \Gamma\alpha$, Lemma 4.3 implies definedness of $constrain(\theta_1\alpha, \Gamma\alpha)$ and Lemma 4.2 yields definedness of $constrain(\theta_0\alpha, \Gamma\alpha)$. Thus $\mathit{unify}(\Gamma, \tau_1{=}\tau_2)$ terminates with a result (Γ_0, θ_0).

It remains to be shown that $\Gamma_1 \vdash \delta : \Gamma_0$. If $\beta \in \mathcal{D}om(\theta_0)$ then $\Gamma_0\beta = \{\}$ and hence $\Gamma_1 \vdash \delta\beta : \Gamma_0\beta$ holds trivially. Now assume $\beta \notin \mathcal{D}om(\theta_0)$. Thus $\Gamma_0\beta = \Gamma_c\beta \cup \Gamma\beta$. From $\Gamma_1 \vdash \theta_1 : \Gamma$ it follows that $\Gamma_1 \vdash \delta\beta : \Gamma\beta$. Proving $\Gamma_1 \vdash \delta : \Gamma_c$ is more involved. From Lemma 4.2 it follows that $constrain(\theta_1\alpha, \Gamma\alpha) \vdash \delta : constrain(\theta_0\alpha, \Gamma\alpha)$ for any α. Since $\Gamma_1 \vdash \theta_1\alpha : \Gamma\alpha$, Lemma 4.3 implies $\Gamma_1 \preceq constrain(\theta_1\alpha, \Gamma\alpha)$ and thus by monotonicity $\Gamma_1 \vdash \delta : constrain(\theta_0\alpha, \Gamma\alpha)$. This in turn easily yields $\Gamma_1 \vdash \delta : Constrain(\theta_0, \Gamma)$, i.e. $\Gamma_1 \vdash \delta : \Gamma_c$. □

Theorem 4.5
Unification modulo Σ is unitary iff Σ is coregular.

Proof The "if" direction is a consequence of Theorem 4.4. For the "only if" direction let Σ not be coregular. Thus there are classes $C, D \leq E$ and declarations $t : (\overline{S_n})C$ and $t : (\overline{T_n})D$, $\overline{S_n} \not\preceq \overline{T_n}$, and $\overline{T_n} \not\preceq \overline{S_n}$, such that there is no third declaration $t : (\overline{U_n})E'$, $E' \preceq E$, and $\overline{S_n}, \overline{T_n} \preceq \overline{U_n}$. Hence the unification problem $([\beta{:}E], t(\overline{\alpha_n}){=}\beta)$ does not have a most general unifier. Two maximal ones are $([\overline{\alpha_n{:}S_n}], \theta)$ and $([\overline{\alpha_n{:}T_n}], \theta)$ where $\theta = \{\beta \to t(\overline{\alpha_n})\}$. □

Thus we have a precise characterization of those signatures where principal types exist.

It remains to be seen if Mini-Haskell's CLASS and INST declarations yield coregular signatures. In fact they do if restricted by the unformalized context conditions set out in Section 3. The latter context conditions imply that every Δ and $\leq$ derived from valid class and instance declarations has the following strong property: $D(t, C)$ is either the singleton $\{\overline{S_n}\}$, where $t(\overline{S_n})C$ is the unique declaration for t with result C, or empty, if there is no such declaration. Therefore $\mathcal{D}om(t, C)$ can be computed using Δ alone, without reference to $\leq$. This leads to the observation that type unification, and hence, as we shall see in the next section, type inference, can ignore the subclass hierarchy completely.

It should be pointed out that ignoring the subclass hierarchy means giving up a degree of freedom afforded by the equivalence $\approx$ on sorts defined in Section 2. For example $\mathit{unify}([\alpha : \{Eq\}, \beta : \{Ord\}], \alpha = \beta)$ returns $([\alpha : \{Eq, Ord\}], \{\beta \mapsto \alpha\})$. Taking $\approx$ into account, we could just as well return $([\alpha : \{Ord\}], \{\beta \mapsto \alpha\})$. In order to show that the subsequent developments do not depend on which of these unifiers is computed, we assume in the sequel that *unify* is an arbitrary function which, provided Σ is coregular, returns a most general unifier: if $\mathcal{U}(\Gamma, \tau_1{=}\tau_2) \neq \{\}$ then

- $\mathit{unify}(\Gamma, \tau_1{=}\tau_2) \in \mathcal{U}(\Gamma, \tau_1{=}\tau_2)$ and
- $(\Gamma', \theta) \leq \mathit{unify}(\Gamma, \tau_1{=}\tau_2)$ for all $(\Gamma', \theta) \in \mathcal{U}(\Gamma, \tau_1{=}\tau_2)$.

This implies a number of simple properties:

Fact 4.6
If $\mathit{unify}(\Gamma, \tau_1{=}\tau_2) = (\Gamma', \theta)$ then

- θ is a most general unifier of τ_1 and τ_2,
- $\mathcal{D}om(\Gamma') \cup \mathcal{FV}(\theta) \subseteq \mathcal{D}om(\Gamma) \cup \mathcal{FV}(\tau_1) \cup \mathcal{FV}(\tau_2)$,
- $\mathcal{D}om(\Gamma') \cap \mathcal{D}om(\theta) = \{\}$.

The second fact states that *unify* does not introduce new variables, and the last expresses that Γ' does not constrain variables instantiated by θ. It is easy to see that the Γ' is determined only up to $\approx$. Hence the unification algorithm could always ensure that Γ' is "minimized" by removing redundant elements from each sort.

Finally one may wonder if the fact that coregularity is strictly weaker than Haskell's context conditions means the latter could be relaxed. We believe that there are no non-trivial relaxations but do not want to enlarge on this subject because it requires going beyond the type system to take semantics and pragmatics into account.

5 Algorithm $\mathcal{W}$

The syntax-directed rule system in Figure 5 is non-deterministic, since rule ASM$'$ can choose any instance of the type of x. To obtain a deterministic algorithm, we refine the syntax directed system such that it keeps types as general as possible. The result is algorithm $\mathcal{W}$ in Figure 6. In this section we assume that Σ is coregular — otherwise *unify* is not well-defined.

```
$\mathcal{W}(V, \Gamma, E, e)$ = case $e$ of
    $x$  ⇒  let $\forall\overline{\alpha_n{:}S_n}.\tau$ = $E(x)$
                $\beta_i \notin V$  $\overline{[i = 1 \ldots n]}$
            in $(V' \cup \{\overline{\beta_n}\}, \Gamma[\overline{\beta_n{:}S_n}], \{\}, \{\overline{\alpha_n \mapsto \beta_n}\}\tau)$
    $\lambda x.e$  ⇒  let $\alpha \notin V$
                $(V', \Gamma', \theta', \tau)$ = $\mathcal{W}(V \cup \{\alpha\}, \Gamma, E[x{:}\alpha], e)$
            in $(V', \Gamma', \theta', \alpha \to \tau)$
    $(e_1\ e_2)$  ⇒  let $(V_1, \Gamma_1, \theta_1, \tau_1)$ = $\mathcal{W}(V, \Gamma, E, e_1)$
                $(V_2, \Gamma_2, \theta_2, \tau_2)$ = $\mathcal{W}(V_1, \Gamma_1, \theta_1 E, e_2)$
                $\alpha \notin V_2$
                $(\Gamma', \theta')$ = $\mathit{unify}(\Gamma_2, \theta_2\theta_1\tau_1 = \theta_2\tau_2 \to \alpha)$
            in $(V_2 \cup \{\alpha\}, \Gamma', \theta'\theta_2\theta_1, \alpha)$
    **let** $x = e_1$ **in** $e_2$  ⇒  let $(V_1, \Gamma_1, \theta_1, \tau_1)$ = $\mathcal{W}(V, \Gamma, E, e_1)$
                $\{\overline{\alpha_n}\}$ = $\mathcal{FV}(\theta_1\tau_1) - \mathcal{FV}(\theta_1 E)$
                $(V_2, \Gamma_2, \theta_2, \tau_2)$ = $\mathcal{W}(V_1, \Gamma_1 \backslash \{\overline{\alpha_n}\},$
                        $(\theta_1 E)[x : \forall\overline{\alpha_n{:}\Gamma_1\alpha_n}.\theta_1\tau_1], e_2)$
            in $(V_2, \Gamma_2, \theta_2\theta_1, \tau_2)$
```

Fig. 6. Algorithm $\mathcal{W}$

Algorithm $\mathcal{W}$ follows the same pattern as Milner's original algorithm of the same name (Mil78): the type of an expression e is computed by traversing e in a top-down manner. $\mathcal{W}(V, \Gamma, E, e)$ returns a quadruple $(V', \Gamma', \theta, \tau)$, where $\theta\tau$ is the type of e

in the context of Γ' and θE. The top level call is $W(\{\}, [], E, e)$, where E is closed. Observe the different let-constructs: the one on the left hand side is in the object language, the ones on the right are part of the type inference algorithm.

The parameter V contains all "used" variables, i.e. variables that occur in Γ or in E. Thus a type variable $\alpha \notin V$ is a "new" variable. For our algorithm to be truly functional, a linear ordering on variables may be used, such that the "next" new variable $\alpha \notin V$ can be computed deterministically. We will assume in general that $\mathcal{W}$ is invoked with V, Γ and E such that $\mathcal{FV}(E) \cup \mathcal{D}om(\Gamma) \subseteq V$.

Algorithm $\mathcal{W}$ is not meant to be implemented directly but merely serves as a mathematically tractable stepping stone towards an efficient implementation. Its principal weakness is the fact that substitutions are computed from scratch and composed later on. This problem is addressed and solved with algorithm $\mathcal{I}$ in the next section. In contrast to substitutions, contexts are computed incrementally, i.e. the result context Γ' is an extension of the input context Γ.

A formal analysis of $\mathcal{W}$ requires some more notation. For an environment E and a substitution θ, define $\theta E = [x : \theta(E(x)) \mid x \in \mathcal{D}om(E)]$. Two **substitutions are equal on a set of variables** W, written as $\theta =_W \theta'$, if $\theta\alpha = \theta'\alpha$ for all $\alpha \in W$. The **restriction of a substitution to a set of variables** W is defined as $\theta|_W\alpha = \theta\alpha$ if $\alpha \in W$ and $\theta|_W\alpha = \alpha$ otherwise. Given a list of syntactic objects $\overline{C_n}$ we write $\mathcal{FV}(\overline{C_n})$ instead of $\mathcal{FV}(C_1) \cup \ldots \cup \mathcal{FV}(C_n)$.

We first show that the algorithm is invariant under α-conversion. The free variables of an expression e, i.e. $\mathcal{FV}(e)$, and the application of a substitution to e are defined as usually in λ-calculus.

Lemma 5.1
If $\mathcal{W}(V, \Gamma, E[x{:}\tau], e) = (V', \Gamma', \theta', \tau')$ and $y \notin \mathcal{D}om(E)$ then $\mathcal{W}(V, \Gamma, E[y{:}\tau], \{x \mapsto y\}e) = (V', \Gamma', \theta', \tau')$.

Proof by induction on e. □

With this lemma, we can easily show the desired theorem for α-conversion.

Theorem 5.2
Let e be $\lambda x.e_2$ or **let** $y = e_1$ **in** e_2. If $\mathcal{W}(V, \Gamma, E, e)$ is defined, $y \notin \mathcal{D}om(E)$, and $y \notin \mathcal{FV}(e)$, then $\mathcal{W}(V, \Gamma, E, e') = \mathcal{W}(V, \Gamma, E, e)$, where e' is $\lambda y.\{x \mapsto y\}e_2$ or **let** $y = e_1$ **in** $\{x \mapsto y\}e_2$ respectively.

Proof by induction, using Lemma 5.1. □

The following correctness and completeness results for $\mathcal{W}$ do not depend on the particular unification algorithm, as discussed towards the end of Section 4.

Theorem 5.3 (Correctness of $\mathcal{W}$)
If $\mathcal{W}(V, \Gamma, E, e) = (V', \Gamma', \theta, \tau)$ then $\Sigma, \Gamma', \theta E \rhd e : \theta\tau$.

Before we can prove the correctness theorem, we need to supply a series of lemmas. The following lemma shows the basic relations between the variables of the objects used by $\mathcal{W}$. The first item states that all used variables are recorded in V'. Next, all new variables occuring in the computed objects are in V' but not in V, i.e. there is no "reuse" of names. The third item states that if some type variables of the computed type are not new, they must have been in the environment E. The last item requires the computed context to be free of assumptions about old variables (which are in $\mathcal{D}om(\theta')$), i.e. no "litter".

Lemma 5.4
Assume $\mathcal{W}(V, \Gamma, E, e) = (V', \Gamma', \theta', \tau)$ and $\mathcal{FV}(E) \cup \mathcal{D}om(\Gamma) \subseteq V$. Then

1. $V \subseteq V'$ and $\mathcal{D}om(\Gamma') \cup \mathcal{FV}(\theta', E, \tau) \subseteq V'$
2. $(\mathcal{D}om(\Gamma') \cup \mathcal{FV}(\theta', \tau)) - (\mathcal{D}om(\Gamma) \cup \mathcal{FV}(E)) \subseteq V' - V$.
3. $\mathcal{FV}(\theta', \tau) \cap V \subseteq \mathcal{FV}(E)$
4. $\mathcal{D}om(\Gamma') \cap \mathcal{D}om(\theta') = \{\}$

Proof The first claim follows easily since all new variables are recorded in V' and since the unification algorithm does not introduce new variables (see Fact 4.6). For the same reason and since all variables in $\mathcal{D}om(\Gamma') \cup \mathcal{FV}(\theta', \tau)$ are either new or in $\mathcal{D}om(\Gamma) \cup \mathcal{FV}(E)$, we obtain the second claim from $\mathcal{FV}(E) \cup \mathcal{D}om(\Gamma) \subseteq V$.

The remaining two items are shown by induction on the term structure:

x : trivial since all β_i are new variables, i.e. $\beta_i \notin V$.
$\lambda x.e$: by induction hypothesis and since α is a new variable.
$(e_1\ e_2)$: We first show

$$(\mathcal{FV}(\theta') \cup \{\alpha\}) \cap V \subseteq \mathcal{FV}(E).$$

Since the unification algorithm does not introduce new variables, it follows that $\mathcal{FV}(\theta') \subseteq \mathcal{FV}(\theta_2\theta_1, \tau_1, \tau_2) \cup \{\alpha\}$. Because $\alpha \notin V$, it suffices to show

$$\mathcal{FV}(\theta_2\theta_1, \tau_2, \tau_1) \cap V \subseteq \mathcal{FV}(E). \tag{1}$$

The induction hypothesis for e_2 yields $\mathcal{FV}(\theta_2, \tau_2) \cap V_1 \subseteq \mathcal{FV}(\theta_1 E)$. Using $\mathcal{FV}(\theta_1 E) \subseteq \mathcal{FV}(E, \theta_1)$ and $V \subseteq V_1$ we obtain $\mathcal{FV}(\theta_2, \tau_2) \cap V \subseteq \mathcal{FV}(E, \theta_1)$. By induction hypothesis for e_1, i.e. $\mathcal{FV}(\theta_1, \tau_1) \cap V \subseteq \mathcal{FV}(E)$, we easily get (1). Next we show that $\mathcal{D}om(\Gamma') \cap \mathcal{D}om(\theta'\theta_2\theta_1) = \{\}$. We obtain $\mathcal{D}om(\Gamma_1) \cap \mathcal{D}om(\theta_1) = \{\}$ from the induction hypothesis. Then from $\mathcal{D}om(\theta_1) \subseteq V_1$ and $\mathcal{D}om(\theta_1) \cap \mathcal{FV}(\theta_1 E) = \{\}$ (idempotence of θ_1) we obtain $\mathcal{D}om(\Gamma_2) \cap \mathcal{D}om(\theta_1) = \{\}$, as $\mathcal{D}om(\Gamma_2) - (\mathcal{D}om(\Gamma_1) \cup \mathcal{FV}(\theta_1 E)) \subseteq V_2 - V_1$ (item 2 of Lemma 5.4). Next, from $\mathcal{D}om(\Gamma_2) \cap \mathcal{D}om(\theta_2) = \{\}$ (induction hypothesis) and since $\mathcal{D}om(\theta_2\theta_1) = \mathcal{D}om(\theta_1) \cup \mathcal{D}om(\theta_2)$, $\mathcal{D}om(\Gamma_2) \cap \mathcal{D}om(\theta_2\theta_1) = \{\}$ follows. Then $\mathcal{D}om(\Gamma') \cap \mathcal{D}om(\theta'\theta_2\theta_1) = \{\}$ follows from the properties of the unification algorithm, i.e. it may not constrain variables from $\mathcal{D}om(\theta')$ (see Fact 4.6).

let $x = e_1$ **in** e_2**:** We first show

$$\mathcal{FV}(\theta_2\theta_1, \tau_2) \cap V \subseteq \mathcal{FV}(E).$$

The induction hypothesis for e_2 yields

$$\mathcal{FV}(\theta_2, \tau_2) \cap V_1 \subseteq \mathcal{FV}(\theta_1 E[x : \forall\overline{\alpha_n{:}\Gamma_1\alpha_n}.\theta_1\tau_1])$$

Since $\mathcal{FV}(\forall\overline{\alpha_n{:}\Gamma_1\alpha_n}.\theta_1\tau_1) \subseteq \mathcal{FV}(\theta_1 E)$, we get

$$\mathcal{FV}(\theta_2, \tau_2) \cap V_1 \subseteq \mathcal{FV}(\theta_1 E).$$

Then the rest of the proof proceeds as for $(e_1\ e_2)$.

The proof of $\mathcal{D}om(\Gamma_2) \cap \mathcal{D}om(\theta_2\theta_1) = \{\}$ also works as in the $(e_1\ e_2)$ case; the only difference (apart from the additional θ') is that we have $\mathcal{D}om(\Gamma_2) \cap \{\overline{\alpha_n}\} = \{\}$, which only simplifies the proof. □

The next lemma shows that the relation $\Gamma \vdash \theta : \Gamma'$ enjoys a kind of transitivity property w.r.t. substitutions.

Lemma 5.5

If $\Gamma_2 \vdash \theta_2 : \Gamma_1$ and $\Gamma_1 \vdash \theta_1 : \Gamma$ then $\Gamma_2 \vdash \theta_2\theta_1 : \Gamma$

Proof We have to show $\forall\alpha \in \mathcal{D}om(\Gamma).\Gamma_2 \vdash \theta_2\theta_1\alpha : \Gamma\alpha$. Consider the derivation of $\Gamma_1 \vdash \theta_1\alpha : \Gamma\alpha$ (by premise). It is easy to construct a derivation of $\Gamma_2 \vdash \theta_2\theta_1\alpha : \Gamma\alpha$, since $\forall\beta \in \mathcal{FV}(\theta_1\alpha).\Gamma_2 \vdash \theta_2\beta : \Gamma_1\beta$ and $\Gamma_1 \vdash \beta : \Gamma\beta$ (as $\theta_1\beta = \beta$ follows from the idempotence of θ_1). □

The fact that $\mathcal{W}$ specializes contexts is shown in the next result.

Lemma 5.6

If $\mathcal{W}(V, \Gamma, E, e) = (V', \Gamma', \theta', \tau)$ then $\Gamma' \vdash \theta' : \Gamma$.

Proof by induction on the structure of e:

x**:** $\Gamma[\beta_n{:}S_n] \vdash \{\} : \Gamma$ trivial.

$\lambda x.e$**:** Since the induction hypothesis holds for any E, including $E[x{:}\alpha]$, $\Gamma' \vdash \theta' : \Gamma$ follows directly.

$(e_1\ e_2)$**:** By induction hypothesis we get $\Gamma_1 \vdash \theta_1 : \Gamma$ and $\Gamma_2 \vdash \theta_2 : \Gamma_1$ and by transitivity (Lemma 5.5) and by correctness of the unification algorithm we get $\Gamma' \vdash \theta'\theta_2\theta_1 : \Gamma$.

let $x = e_1$ **in** e_2**:** By induction hypothesis we get

$$\Gamma_1 \vdash \theta_1 : \Gamma \tag{2}$$

$$\Gamma_2 \vdash \theta_2 : \Gamma_1 \backslash \{\alpha_n\} \tag{3}$$

Now we show

$$\Gamma_1 \backslash \{\alpha_n\} \vdash \theta_1 : \Gamma \tag{4}$$

That is, we have to show $\Gamma_1\backslash\{\overline{\alpha_n}\} \vdash \theta_1\beta : \Gamma\beta$ for all $\beta \in \mathcal{D}om(\Gamma)$. First we prove $\mathcal{FV}(\theta_1(\mathcal{D}om(\Gamma))) \cap \{\overline{\alpha_n}\} = \{\}$. From Lemma 5.4, item 3, it follows that $\mathcal{FV}(\theta_1\tau_1) \cap \mathcal{D}om(\Gamma) \subseteq \mathcal{FV}(E)$. Idempotence of θ_1 yields $\mathcal{FV}(\theta_1\tau_1) \cap \mathcal{FV}(\theta_1(\mathcal{D}om(\Gamma))) \subseteq \mathcal{FV}(\theta_1 E)$. Simple set theory yields $\mathcal{FV}(\theta_1(\mathcal{D}om(\Gamma))) \cap (\mathcal{FV}(\theta_1\tau_1) - \mathcal{FV}(\theta_1 E)) = \{\}$ as claimed above. Now (4) follows.

Combining (4) and (3) by transitivity (Lemma 5.5) yields $\Gamma_2 \vdash \theta_2\theta_1 : \Gamma$ □

The next lemma states that $\rhd$ is preserved under instantiation assuming a context that obeys the constraints.

Lemma 5.7

If $\Sigma, \Gamma, E \rhd e : \tau$ and $\Gamma' \vdash \theta' : \Gamma$, then $\Sigma, \Gamma', \theta' E \rhd e : \theta'\tau$.

Proof simple by adding proofs of the form $\Gamma' \vdash \theta'\alpha : \Gamma\alpha$ in the proof tree of $\Sigma, \Gamma, E \rhd e : \tau$ to obtain a proof of $\Sigma, \Gamma', \theta' E \rhd e : \theta'\tau$. □

At last we are able to prove the correctness theorem:

Proof of Theorem 5.3 by induction on the structure of e. We have the following cases:

x**:** Correctness follows easily from

$$\text{ASM}' \quad \frac{\Sigma, \Gamma[\overline{\beta_n{:}S_n}] \vdash E(x) \succeq \{\overline{\alpha_n \mapsto \beta_n}\}\tau}{\Sigma, \Gamma[\overline{\beta_n{:}S_n}], E \rhd x : \{\overline{\alpha_n \mapsto \beta_n}\}\tau}$$

$\lambda x.e$**:** By induction hypothesis we get $\Sigma, \Gamma', (\theta' E)[x{:}\theta'\alpha] \rhd e : \theta\tau$. Then ABS applies:

$$\text{ABS} \quad \frac{\Sigma, \Gamma, (\theta' E)[x{:}\theta'\alpha] \rhd e : \theta'\tau'}{\Sigma, \Gamma, \theta' E \rhd \lambda x.e : \theta'\alpha \to \theta\tau'}$$

$(e_1\ e_2)$**:** We get

$$\begin{aligned} \Sigma, \Gamma_1, \theta_1 E &\rhd e_1 : \theta_1\tau_1 \\ \Sigma, \Gamma_2, \theta_2\theta_1 E &\rhd e_2 : \theta_2\tau_2 \end{aligned}$$

from the induction hypotheses for e_1 and e_2. The correctness of the unification algorithm yields $\Gamma' \vdash \theta' : \Gamma_2$ and then with $\Gamma_2 \vdash \theta_2 : \Gamma_1$ (from Lemma 5.6) and Lemma 5.5 we obtain $\Gamma' \vdash \theta'\theta_2 : \Gamma_1$.

From Lemma 5.7 we now get the two premises for the *APP* rule, since $\theta'\theta_2\theta_1\tau_1 = \theta'\theta_2\tau_2 \to \theta'\alpha$. Furthermore, $\theta_2\theta_1\alpha = \alpha$, since α is a new variable (i.e. $\alpha \notin V_2$ and $\mathcal{D}om(\theta_2) \cup \mathcal{D}om(\theta_1) \subseteq V_2$ by Lemma 5.4).

$$\text{APP} \quad \frac{\Sigma, \Gamma', \theta'\theta_2\theta_1 E \rhd e_1 : \theta'\theta_2\theta_1\tau_1 \quad \Sigma, \Gamma', \theta'\theta_2\theta_1 E \rhd e_2 : \theta'\theta_2\tau_2}{\Sigma, \Gamma', \theta'\theta_2\theta_1 E \rhd (e_1\ e_2) : \theta'\alpha}$$

let $x = e_1$ **in** e_2 **:** Using $\Gamma_1' = \Gamma_1 \backslash \{\overline{\alpha_n}\}$, $\overline{S_n = \Gamma_1\alpha_n}$, and $E' = E[x : \forall\overline{\alpha_n{:}S_n}.\theta_1\tau_1]$ the induction hypotheses are

$$\Sigma, \Gamma_1'[\overline{\alpha_n{:}S_n}], \theta_1 E \rhd e_1 : \theta_1\tau_1 \tag{5}$$

$$\Sigma, \Gamma_2, \theta_2\theta_1 E' \rhd e_2 : \theta_2\tau_2 \tag{6}$$

Notice that $\mathcal{FV}(\theta_1 E') \cap \{\overline{\alpha_n}\} = \{\}$. To apply LET′, we show

$$\Sigma, \Gamma_2[\overline{\alpha_n{:}S_n}], \theta_2\theta_1 E \rhd e_1 : \theta_2\theta_1\tau_1 \tag{7}$$

As we get $\Gamma_2 \vdash \theta_2 : \Gamma_1'$ from Lemma 5.6 and $\mathcal{FV}(\theta_2) \cap \{\overline{\alpha_n}\} = \{\}$ from Lemma 5.4 (recall that $\{\overline{\alpha_n}\} \subseteq V_1$ and $\{\overline{\alpha_n}\} \cap \mathcal{FV}(\theta_1 E') = \{\}$), we obtain

$$\Gamma_2[\overline{\alpha_n{:}S_n}] \vdash \theta_2 : \Gamma_1'[\overline{\alpha_n{:}S_n}].$$

Then (7) follows from Lemma 5.7 and (5).

As $\mathcal{D}om(\theta_1) \cap \mathcal{FV}(\tau_2) = \{\}$ is a consequence of Lemma 5.4 (as above, $\mathcal{D}om(\theta_1) \subseteq V_1$ and $\mathcal{D}om(\theta_1) \cap \mathcal{FV}(\theta_1 E') = \{\}$ as θ_1 is idempotent), we obtain

$$\Sigma, \Gamma_2, \theta_2\theta_1 E' \rhd e_2 : \theta_2\theta_1\tau_2 \tag{8}$$

Now LET$'$ applies to

$$\frac{(7) \quad (8) \quad \{\overline{\alpha_n}\} = \mathcal{FV}(\theta_1\tau_1) - \mathcal{FV}(\theta_1 E)}{\Gamma_2, \Sigma, \theta_2\theta_1 E \rhd \textbf{let } x = e_1 \textbf{ in } e_2 : \theta_2\theta_1\tau_2}$$

□

The following lemma is crucial for establishing the principal type theorem.

Lemma 5.8 (Completeness of $\mathcal{W}$)
If $\Sigma, \Gamma^*, \theta^* E \rhd e : \tau^*$, $\mathcal{D}om(\Gamma) \cup \mathcal{FV}(E) \subseteq V$, and $\Gamma^* \vdash \theta^* : \Gamma$ then there exists a substitution δ such that

$$\begin{array}{rcl} \mathcal{W}(V, \Gamma, E, e) & = & (V_1, \Gamma_1, \theta_1, \tau_1), \\ \theta^* E & = & \delta\theta_1 E, \\ \tau^* & = & \delta\theta_1\tau_1, \\ \Gamma^* & \vdash & \delta : \Gamma_1. \end{array}$$

Proof by induction on the structure of e. We assume w.l.o.g. a derivation for $\Sigma, \Gamma^*, \theta^* E \rhd e : \tau^*$ that has no variable overlap with the new variables $V_1 - V$ used by algorithm $\mathcal{W}$.

x**:** We have

$$\text{ASM}' \quad \frac{\Sigma, \Gamma^* \vdash \theta^* E(x) \succeq \tau^*}{\Sigma, \Gamma^*, \theta^* E \rhd x : \tau^*}$$

Observe that we can write $\theta E(x) = \theta^*\forall\overline{\alpha_n{:}S_n}.\tau$ as $\forall\overline{\alpha_n{:}S_n}.\hat{\theta}^*\tau$, where $\hat{\theta}^* = \theta^*|_{\mathcal{D}om(\theta^*)-\{\overline{\alpha_n}\}}$, possibly by renaming some $\overline{\alpha_n}$. Assuming $\Sigma, \Gamma^* \vdash \theta^* E(x) \succeq \tau^*$, let θ be the corresponding substitution as in Definition 3.1 with $\mathcal{D}om(\theta) \subseteq \{\overline{\alpha_n}\}$, and $\tau^* = \theta\hat{\theta}^*\tau$. Let $\delta = \theta^* \cup \{\overline{\beta_n \mapsto \theta\alpha_n}\}$. As $\overline{\beta_n}$ are new variables, $\theta^* E = \delta E$ holds. Next, $\tau^* = \delta(\{\overline{\alpha_n \mapsto \beta_n}\}\tau) = \theta\hat{\theta}^*\tau$ follows easily. Finally, $\Gamma^* \vdash \delta : \Gamma[\overline{\beta_n{:}S_n}]$ follows from $\Gamma^* \vdash \theta^* : \Gamma$ (by premise) and from $\Sigma, \Gamma^* \vdash \theta'\alpha_i : S_i, \quad [i = 1, \ldots, n]$ (see Definition 3.1).

$\lambda x.e$**:** The derivation ends with

$$\text{ABS} \quad \frac{\Sigma, \Gamma^*, (\theta^* E)[x{:}\tau_1^*] \rhd e : \tau_2^*}{\Sigma, \Gamma^*, \theta^* E \rhd \lambda x.e : \tau_1^* \to \tau_2^*}$$

As the algorithm is invariant under α-conversion (Lemma 5.2), we can safely assume that $x \notin \mathcal{D}om(E)$. To apply the induction hypotheses, we define $\theta_0 = \theta^* \cup \{\alpha \mapsto \tau_1^*\}$. Then $\Sigma, \Gamma^*, \theta_0(E[x{:}\alpha]) \rhd e : \tau_2^*$ and $\Gamma^* \vdash \theta_0 : \Gamma$ are easy to verify. By induction hypothesis there exists δ_1 such that

$$\begin{array}{rcll} \mathcal{W}(V \cup \{\alpha\}, \Gamma, E[x{:}\alpha], e) & = & (V', \Gamma', \theta', \tau'), & (9) \\ (\theta_0 E)[x{:}\tau_1^*] & = & \delta_1\theta' E[x{:}\alpha], & (10) \\ \tau_2^* & = & \delta_1\theta'\tau', & (11) \\ \Gamma^* & \vdash & \delta_1 : \Gamma' & (12) \end{array}$$

and hence $\mathcal{W}(V, \Gamma, E, \lambda x.e) = (V', \Gamma', \theta', \alpha \to \tau')$. Now $\theta_0 E = \delta_1\theta' E$ follows from (10). Furthermore, from (10) we obtain $\tau_1^* = \delta_1\theta'\alpha$ and hence $\tau_1^* \to \tau_2^* = \delta_1\theta'(\alpha \to \tau')$ from (11).

$(e_1\ e_2)$**:** We assume

$$\text{APP}\quad \frac{\Sigma, \Gamma^*, \theta^* E \rhd e_1 : \tau_2^* \rightarrow \tau_1^* \quad \Sigma, \Gamma^*, \theta^* E \rhd e_2 : \tau_2^*}{\Sigma, \Gamma^*, \theta^* E \rhd (e_1\ e_2) : \tau_1^*}$$

Applying the induction hypothesis to e_1 yields δ_1 such that

$$\begin{aligned} \mathcal{W}(V, \Gamma, E, e) &= (V_1, \Gamma_1, \theta_1, \tau_1), & (13)\\ \theta^* E &= \delta_1 \theta_1 E, & (14)\\ \tau_2^* \rightarrow \tau_1^* &= \delta_1 \theta_1 \tau_1, & (15)\\ \Gamma^* &\vdash \delta_1 : \Gamma_1. & (16) \end{aligned}$$

The induction hypothesis with e_2 with Γ_1, where $\Gamma^* \vdash \delta_1 : \Gamma_1$, and $\theta_1 E$ yields δ_2 such that

$$\begin{aligned} \mathcal{W}(V_1, \Gamma_1, \theta_1 E, e_2) &= (V_2, \Gamma_2, \theta_2, \tau_2), & (17)\\ \theta^* E &= \delta_2 \theta_2 \theta_1 E, & (18)\\ \tau_2^* &= \delta_2 \theta_2 \tau_2, & (19)\\ \Gamma^* &\vdash \delta_2 : \Gamma_2. & (20) \end{aligned}$$

Let δ^* be defined as

$$\delta^* \beta = \begin{cases} \delta_1 \beta & \text{if } \beta \in \mathcal{FV}(\theta_1 \tau_1) - \mathcal{C}od(\theta_2) \\ \tau_1^* & \text{if } \beta = \alpha \\ \delta_2 \beta & \text{otherwise} \end{cases}$$

We show that δ^* is a unifier of $\theta_2 \theta_1 \tau_1 = \theta_2 \tau_2 \rightarrow \alpha$. Notice that $\mathcal{FV}(\theta_2, \tau_2) \cap \mathcal{FV}(\theta_1 \tau_1) \subseteq \mathcal{FV}(\theta_1 E)$ follows from Lemma 5.4 and that the two substitutions $\delta_2 \theta_2$ and δ_1 coincide on $\mathcal{FV}(\theta_1 E)$: combining (14) with (18) yields

$$\delta_2 \theta_2 =_{\mathcal{FV}(\theta_1 E)} \delta_1 \qquad (21)$$

This overlap simplifies the following proofs by case analysis, in which the case $\beta = \alpha$ is immaterial. First, to show $\delta^* \theta_2 \tau_2 = \delta_2 \theta_2 \tau_2 = \tau_2^*$, assume $\beta \in \mathcal{FV}(\tau_2)$. Then in case $\beta \notin \mathcal{FV}(\theta_2)$, $\delta^* \tau_2 = \delta_2 \tau_2$ follows from (21) if $\beta \in \mathcal{FV}(\theta_1 \tau_1)$ as $\mathcal{FV}(\theta_2, \tau_2) \cap \mathcal{FV}(\theta_1 \tau_1) \subseteq \mathcal{FV}(\theta_1 E)$, and is trivial otherwise. If $\beta \in \mathcal{FV}(\theta_2)$, then $\delta^* \theta_2 \tau_2 = \delta_2 \theta_2 \tau_2$ is trivial.
To show $\delta^* \theta_2 \theta_1 \tau_1 = \delta_1 \theta_1 \tau_1$, assume $\beta \in \mathcal{FV}(\theta_1 \tau_1)$. If $\beta \in \mathcal{D}om(\theta_2)$, (21) gives the desired result as $\mathcal{D}om(\theta_2) \cap \mathcal{FV}(\theta_1 \tau_1) \subseteq \mathcal{FV}(\theta_1 E)$. In case $\beta \notin \mathcal{D}om(\theta_2)$, $\delta^* \theta_1 \tau_1 = \delta_1 \tau_1$ follows easily. Hence δ^* is the desired unifier:

$$\delta^* \theta_2 \theta_1 \tau_1 = \tau_2^* \rightarrow \tau_1^* = \delta^* (\theta_2 \tau_2 \rightarrow \alpha).$$

We obtain $\Gamma^* \vdash \delta^* : \Gamma_2$ from (16) and (20) by case analysis: we have to show $\Sigma, \Gamma^* \vdash \delta^* \beta : \Gamma_2 \beta$ for all $\beta \in \mathcal{D}om(\Gamma_2)$. First recall that $\mathcal{D}om(\Gamma_2) \cap \mathcal{D}om(\theta_2) = \{\}$ by Lemma 5.4. If $\beta \in \mathcal{FV}(\theta_1 \tau_1) - \mathcal{C}od(\theta_2)$, then we have another case distinction: if $\beta \in \mathcal{D}om(\theta_2)$, then $\Gamma_2 \beta = \{\}$, otherwise $\theta_2 \beta = \beta$ and from $\Gamma_2 \vdash \theta_2 : \Gamma_1$ (Lemma 5.6), we have $\Gamma_1 \beta \subseteq \Gamma_2 \beta$ and the claim follows from (16). The case $\beta = \alpha$ is trivial because $\Gamma_2 \alpha = \{\}$. The remaining case, $\Sigma, \Gamma^* \vdash \delta_2 \theta_2 \beta : \Gamma_2 \beta$, follows easily from (20) since $\mathcal{D}om(\Gamma_2) \cap \mathcal{D}om(\theta_2) = \{\}$.

Then by completeness of unification, θ' is a most general unifier computed in *unify*, and there exists δ' such that $\delta^* = \delta'\theta'$. Hence we get

$$\begin{array}{rcl} \mathcal{W}(V, \Gamma, E, (e_1\ e_2)) & = & (V_2 \cup \alpha, \Gamma', \theta'\theta_2\theta_1, \alpha), \\ \theta^* E & = & \delta'\theta'\theta_2\theta_1 E, \\ \tau^* & = & \delta'\theta'\theta_2\tau_2, \\ \Gamma^* & \vdash & \delta' : \Gamma', \end{array}$$

where the last statement follows from the completeness of unification.

let $x = e_1$ **in** e_2**:** We assume $\Gamma^* \vdash \theta^* : \Gamma$ and, by LET$'$,

$$\frac{\Sigma, \Gamma^*[\overline{\alpha_k^*{:}S_k^*}], \theta^* E \rhd e_1 : \tau_1^* \quad \Sigma, \Gamma^*, (\theta^* E)[x{:}\forall\overline{\alpha_k^*{:}S_k^*}.\tau_1^*] \rhd e_2 : \tau_2^*}{\Sigma, \Gamma^*, \theta^* E \rhd \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2^*}$$

where $\{\overline{\alpha_k^*}\} = \mathcal{FV}(\tau_1^*) - \mathcal{FV}(\theta^* E)$. As the algorithm is invariant under α-conversion (Lemma 5.2), we can safely assume that $x \notin \mathcal{D}om(E)$.
As $\{\overline{\alpha_k^*}\} \cap \mathcal{FV}(\theta^* E) = \{\}$ we can w.l.o.g. rename $\overline{\alpha_k^*}$ in the premises of the above rule (not in Γ') in order to assume that $\{\overline{\alpha_k^*}\} \cap \mathcal{D}om(\Gamma^*) = \{\}$. Formally, this can be done by Lemma 5.5. Then we can apply the induction hypothesis to e_1 with $\Gamma^*[\overline{\alpha_k^*{:}S_k^*}] \vdash \theta^* : \Gamma$ and obtain δ_1 such that

$$\begin{array}{rcll} \mathcal{W}(V, \Gamma, E, e_1) & = & (V_1, \Gamma_1, \theta_1, \tau_1), & (22) \\ \theta^* E & = & \delta_1\theta_1 E, & (23) \\ \tau_1^* & = & \delta_1\theta_1\tau_1, & (24) \\ \Gamma^*[\overline{\alpha_k^*{:}S_k^*}] & \vdash & \delta_1 : \Gamma_1'[\overline{\alpha_n{:}S_n}], & (25) \end{array}$$

where $\Gamma_1' = \Gamma_1 \backslash \{\overline{\alpha_n}\}$ and $\overline{S_n} = \overline{\Gamma_1 \alpha_n}$.
From $\{\overline{\alpha_n}\} = \mathcal{FV}(\theta_1\tau_1) - \mathcal{FV}(\theta_1 E)$ we infer $\mathcal{FV}(\{\overline{\delta_1\alpha_n}\}) = \mathcal{FV}(\delta_1\theta_1\tau_1) - \mathcal{FV}(\delta_1\theta_1 E) = \{\overline{\alpha_k^*}\}$. Hence $\Gamma^* \vdash \delta_1 : \Gamma_1'$ follows from (25). Notice that $\delta_1\forall\overline{\alpha_n{:}S_n}.\theta_1\tau_1 = \forall\overline{\alpha_n{:}S_n}.\hat{\delta}_1\theta_1\tau_1$, where $\hat{\delta}_1 = \delta_1|_{\mathcal{D}om(\delta_1) - \{\overline{\alpha_n}\}}$, follows from the assumption that new variables used by $\mathcal{W}$ do not occur in the chosen derivation, i.e. in $\mathcal{C}od(\delta_1)$. Then we obtain $\Sigma, \Gamma^* \vdash \delta_1\forall\overline{\alpha_n{:}S_n}.\theta_1\tau_1 \succeq \forall\overline{\alpha_k^*{:}S_k^*}.\tau^*$ by using $\delta_1|_{\{\overline{\alpha_n}\}}$ as the substitution in Definition 3.1 and because $\Sigma, \Gamma^*[\overline{\alpha_k^*{:}S_k^*}] \vdash \delta_1\alpha_i : S_i, [i = 1 \ldots n]$ follows from (25).
Now the problem is that the induction hypothesis cannot be applied directly to e_2 with $\Gamma^* \vdash \delta_1 : \Gamma_1'$ and $\theta_1 E[\ldots]$, since in general $(\theta^* E)[x{:}\forall\overline{\alpha_k^*{:}S_k^*}.\tau_1^*] \neq (\delta_1\theta_1 E)[x{:}\delta_1\forall\overline{\alpha_n{:}S_n}.\theta_1\tau_1]$. Thus we have to find a different basis in order to apply the induction hypothesis for e_2.
From

$$\Sigma, \Gamma^*(\theta^* E)[x{:}\forall\overline{\alpha_k^*{:}S_k^*}.\tau_1^*] \rhd e_2 : \tau_2^* \qquad (26)$$

we can infer

$$\Sigma, \Gamma^*, (\theta^* E)[x{:}\delta_1\forall\overline{\alpha_n{:}S_n}.\theta_1\tau_1] \rhd e_2 : \tau_2^*,$$

since at each application of ASM$'$ to x in the proof of (26), we can use the more general $[x{:}\delta_1\forall\overline{\alpha_n{:}S_n}.\theta_1\tau_1]$ instead of $[x{:}\forall\overline{\alpha_k^*{:}S_k^*}.\tau_1^*]$.
Then the induction hypothesis applies to e_2 with $\Gamma^* \vdash \delta_1 : \Gamma_1'$ and the envi-

ronment $\theta_1 E[x{:}\forall\overline{\alpha_n{:}S_n}.\theta_1\tau_1]$. We get δ_2 such that

$$\begin{aligned}
\mathcal{W}(V_1,\Gamma_1',E,e_2) &= (V_2,\Gamma_2,\theta_2,\tau_2), && (27)\\
(\theta^* E)[x{:}\delta_1\forall\overline{\alpha_n{:}S_n}.\theta_1\tau_1] &= \delta_2\theta_2(\theta_1 E[x{:}\forall\overline{\alpha_n{:}S_n}.\theta_1\tau_1]), && (28)\\
\tau_2^* &= \delta_2\theta_2\tau_2, && (29)\\
\Gamma^* &\vdash \delta_2 : \Gamma_2. && (30)
\end{aligned}$$

Hence $\mathcal{W}(V,\Gamma,E,\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) = (V_2,\Gamma_2,\theta_2,\tau_2)$. We obtain $\tau_2^* = \delta_2\theta_2\theta_1\tau_2$ from $\tau_2^* = \delta_2\theta_2\tau_2$ and $\mathcal{D}om(\theta_1)\cap\mathcal{FV}(\tau_2) = \{\}$ (as in Theorem 5.3). It only remains to show $\theta^* E = \delta_2\theta_2\theta_1 E$, which is a consequence of (28), as $x \notin \mathcal{D}om(E)$. □

Now we can finally show the desired principal type theorem.

Theorem 5.9
If e has type σ_0 under a closed environment E, i.e. $\Sigma,[],E\vdash e:\sigma_0$ and $\mathcal{FV}(E)\subseteq V_0$, then $\mathcal{W}(V_0,[],\{\},E,e) = (V,\Gamma,\theta,\tau)$ and $\forall\overline{\alpha_n{:}\Gamma\alpha_n}.\theta\tau$ is a principal type of e w.r.t. Σ and E, where $\{\overline{\alpha_n}\} = \mathcal{FV}(\theta\tau)$.

Proof Assume some typing $\Sigma,[],E\vdash e:\forall\overline{\alpha_m'{:}S_m'}.\tau'$. We infer $\Sigma,[\alpha_m'{:}S_m'],E\vdash e:\tau'$ by $\forall E$ and then obtain a syntax-directed derivation $\Sigma,[\alpha_m'{:}S_m'],E\rhd e:\tau'$ by Theorem 3.4. Then Lemma 5.8 applies with $\Gamma^* = [\alpha_m'{:}S_m']$ and $\theta^* = \{\}$. We thus get δ such that

$$\begin{aligned}
E &= \delta E,\\
\tau' &= \delta\theta\tau,\\
\Gamma^* &\vdash \delta : \Gamma.
\end{aligned}$$

Then $\forall\overline{\alpha_n{:}\Gamma\alpha_n}.\theta\tau$ is a principal type of e w.r.t. E, since $\forall\overline{\alpha_n{:}\Gamma\alpha_n}.\theta\tau \succeq \forall\overline{\alpha_m'{:}S_m'}.\tau'$ follows from $\tau' = \delta\theta\tau$, $\{\overline{\alpha_n}\} = \mathcal{FV}(\theta\tau)$, and $\Gamma^*\vdash\delta:\Gamma$. □

6 Algorithm $\mathcal{I}$

As in the original work by Milner (Mil78), we now present a more efficient refinment of algorithm $\mathcal{W}$. Compared to $\mathcal{W}$, algorithm $\mathcal{I}$[2] takes an extra argument, the substitution computed so far. This substitution is extended incrementally instead of computing new subtitutions and composing them later.

The equivalence of $\mathcal{W}$ and $\mathcal{I}$ is an easy matter. A renaming is an injective substitution that maps variables to variables only.

Theorem 6.1 (Equivalence of $\mathcal{W}$ and $\mathcal{I}$)
Assume θ_0 is an idempotent substitution such that $\theta_0 E_0 = E$. If $\mathcal{W}(V,\Gamma,E,e) = (V',\Gamma',\theta',\tau')$ then $\mathcal{I}(V,\Gamma,\theta_0,E_0,e) = (V'',\Gamma'',\theta'',\tau'')$ and there exists a renaming σ such that $V'' = \sigma V'$, $\forall\alpha.\Gamma''\alpha = \Gamma'(\sigma\alpha)$, $\theta''\tau'' = \sigma\theta'\tau'$ and $\theta'' E = \sigma\theta' E$.

Proof by simple induction on the structure of e. □

[2] Although the typography in (Mil78) is ambiguous, Milner has confirmed by email that he intended it to be $\mathcal{I}$, not $\mathcal{J}$: it is an imperative implementation of $\mathcal{W}$. Milner's $\mathcal{I}$ is imperative because he maintains a single global copy of θ which is updated by side-effects. In a purely functional style this requires an additional argument and result.

```
$\mathcal{I}(V,\Gamma,\theta,E,e) =$ case $e$ of
    $x$ $\Rightarrow$ let $\forall\overline{\alpha_n{:}S_n}.\tau = E(x)$
                          $\beta_i \notin V \ [i = 1 \ldots n]$
                      in $(V \cup \{\overline{\beta_n}\}, \Gamma[\overline{\beta_n{:}S_n}], \theta, \{\overline{\alpha_n \mapsto \beta_n}\}\tau)$
    $\lambda x.e$ $\Rightarrow$ let $\alpha \notin V$
                          $(V',\Gamma',\theta',\tau) = \mathcal{I}(V \cup \{\alpha\}, \Gamma, \theta, E[x{:}\alpha], e)$
                      in $(V',\Gamma',\theta',\alpha \to \tau)$
    $(e_1\ e_2)$ $\Rightarrow$ let $(V_1,\Gamma_1,\theta_1,\tau_1) = \mathcal{I}(V,\Gamma,\theta,E,e_1)$
                          $(V_2,\Gamma_2,\theta_2,\tau_2) = \mathcal{I}(V_1,\Gamma_1,\theta_1,E,e_2)$
                          $\alpha \notin V_2$
                          $(\Gamma',\theta') = \mathit{unify}(\Gamma_2, \theta_2\tau_1 = \theta_2\tau_2 \to \alpha)$
                      in $(V_2 \cup \{\alpha\}, \Gamma', \theta'\theta_2, \alpha)$
    $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ $\Rightarrow$ let $(V_1,\Gamma_1,\theta_1,\tau_1) = \mathcal{I}(\Gamma,\theta,E,e_1)$
                          $\{\overline{\alpha_n}\} = \mathcal{FV}(\theta_1\tau_1) - \mathcal{FV}(\theta_1 E)$
                      in $\mathcal{I}(V_1, \Gamma_1\backslash\{\overline{\alpha_n}\}, \theta_1, E[x : \forall\overline{\alpha_n{:}\Gamma_1\alpha_n}.\theta_1\tau_1], e_2)$
```

Fig. 7. Algorithm $\mathcal{I}$

7 Related Work

The structure of algorithms $\mathcal{W}$ and $\mathcal{I}$ is very close to that of Milner's algorithms of the same name (Mil78). Apart from the fact that our version of $\mathcal{I}$ is purely applicative (hence we carry the substitution and the set of used variables around explicitly), the main difference is that we also have to maintain a set of constraints Γ. In fact, this is the only real difference to Milner's algorithms.

Probably the first combination of ML-style polymorphism and parametric overloading (as opposed to finite overloading as in Hope (BMS80)) was presented by Kaes (Kae88). His language is in fact very close to our Mini-Haskell, except that he does not introduce classes explicitly. More importantly, he does not use contexts to record information about type variables but tags type variables directly.

The original version of type classes as presented by Wadler and Blott (WB89) was significantly more powerful than what went into Haskell, the reason being that the original system was undecidable, as shown later by Volpano and Smith (VS91). The relationship to Haskell proper is discussed in Section 2.

Nipkow and Snelting (NS91) realized that type inference for type classes can be formulated as an extension of ordinary ML-style type inference with order-sorted unification, i.e. simply by changing the algebra of types and the corresponding unification algorithm. Although this was an interesting theoretical insight, it only lead to a simple algorithm for a restricted version of Haskell where each type variable is constrained by exactly one class. In addition it was not possible to identify ambiguous typings like $\Sigma, [\alpha{:}C], E \vdash e : \mathit{int}$ because there was no notion of contexts and type variables were tagged with their sort. Both problems have been eliminated in the present paper.

An interesting extension of Haskell using the notion of "qualified types" was designed and implemented by Mark Jones (Jon92b). The main difference is that he allows arbitrary predicates $P(\tau_1,\ldots,\tau_n)$ over types as opposed to our membership constraints $\alpha : S$. On the other hand he does not solve constraints of the form $\tau : S$ to obtain atomic constraints of the form $\alpha : S'$ as is done in our function *constrain*. Instead he accumulates the unsolved constraints.

Independently of our own work Chen, Hudak and Odersky (CHO92) developed an extension of type classes using similar techniques and arriving at a similar type reconstruction algorithm. Since their type system is more general, they use different and more involved formalisms, in particular for unification. In contrast, we reduce unification to its essence by splitting it into standard unification plus constraint solving. This enables us to give a sufficient and necessary criterion for unitary unification, which is required for principal types. As discussed in Section 4, the restrictions in Haskell guarantee unitary unification.

Kaes (Kae92) presents an extension of Hindley/Milner polymorphism with overloading, subtypes and recursive types. Due to the overall complexity of the resulting system, the simplicity of the pure system for overloading is lost.

The pragmatics of implementing type classes are discussed by Peterson and Jones (PJ83). In particular they give hints on how to implement a truly imperative version of algorithm $\mathcal{I}$ using mutable variables. This is of significant importance because a naïve functional implementation of algorithm $\mathcal{I}$, in particular one representing substitutions as association lists, performs quite poorly.

8 Ambiguity

We would like to conclude this paper with a discussion of the ambiguity problem which affects most type systems with overloading. It is caused by the fact that although a program may have a unique type, its semantics is not well-defined. According to our rules, the program

$$\begin{array}{l}\textbf{class}\ \alpha : C\ \textbf{where}\ f : \alpha \rightarrow int;\\ \textbf{class}\ \alpha : D\ \textbf{where}\ c : \alpha;\\ (f\ c)\end{array}$$

has type *int* in any context containing an assumption $\alpha : \{C, D\}$. Yet the program has no semantics because there are no instances of f and c at all. If there were multiple instances of both C and D, it would be impossible to determine which one to use in the expression $(f\ c)$.

Motivated by such examples, a typing $\Sigma, \Gamma, E \vdash e : \sigma$ is usually defined to be **ambiguous** if there is a type variable in Γ which does not occur free in σ or E.

Ideally one would like to have that every well-typed expression has a well-defined semantics. However, ambiguous terms may have more than one semantics, as the above example suggests. Fortunately, Blott (Blo92) and Jones (Jon92a) have shown that in type systems closely related to the one studied in this paper, the semantics of unambiguous terms is indeed well-defined.

As we have not provided a semantics for our language, we have not introduced ambiguity formally. Nevertheless there is one place in our inference system where we anticipate a particular treatment of ambiguity. In rule $\forall$I, the proviso $\alpha \in \mathcal{FV}(\sigma)$ is intended to propagate ambiguity problems: with this restriction, the expression

let $x = (f\ c)$ **in** 5 (preceded by classes C and D as declared above) has type *int* only in a context containing an assumption $\alpha : \{C, D\}$. If the proviso is dropped, the expression also has type *int* in the empty context, thus disguising the local ambiguity. The reason is that x can be given the ambiguous type $\forall\alpha{:}\{C, D\}.int$, but since x does not occur in 5, this does not matter. Although in a lazy language x need not be evaluated and hence the semantics of the whole **let** is indeed unambiguous, we would argue that for pragmatic reasons it is advisable to flag ambiguities whenever they arise.

From this discussion it is obvious that a semantics and a coherence proof for the type system w.r.t. a semantics are urgently needed.

Acknowledgements. The authors wish to thank the anonymous referees for their critical reading and their helpful comments.

References

Stephen Blott. *An approach to overloading with polymorphism.* PhD thesis, Dept. of Computing Science, University of Glasgow, 1992.

Rod Burstall, Dave MacQueen, and Don Sannella. Hope: an experimental applicative language. In *Proc. 1980 LISP Conference*, pages 136–143, 1980.

Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 13–27, 1986.

Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *Proc. ACM Conf. on LISP and Functional Programming*, pages 170–181. ACM Press, June 1992.

Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symp. Principles of Programming Languages*, pages 207–212, 1982.

Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

Mark P. Jones. Qualified types: Theory and practice. D.Phil. Thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992.

Mark P. Jones. A theory of qualified types. In Bernd Krieg-Brückner, editor, *Proc. European Symposium on Programming*, pages 287–306. LNCS 582, 1992.

Stefan Kaes. Parametric overloading in polymorphic programming languages. In *Proc. 2nd European Symposium on Programming*, pages 131–144. LNCS 300, 1988.

Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proc. ACM Conf. LISP and Functional Programming*, pages 193–204. ACM Press, June 1992.

Robin Milner. A theory of type polymorphism in programming. *J. Comp. Sys. Sci.*, 17:348–375, 1978.

Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 409–418. ACM Press, 1993.

Tobias Nipkow and Gregor Snelting. Type classes and overloading resolution via order-sorted unification. In *Proc. 5th ACM Conf. Functional Programming Languages and Computer Architecture*, pages 1–14. LNCS 523, 1991.

John Peterson and Mark Jones. Implementing type classes. In *Proc. SIGPLAN '93 Symp. Programming Language Design and Implementation*, pages 227–236. ACM Press, 1983.

Dennis M. Volpano and Geoffrey S. Smith. On the complexity of ML typability with overloading. In *Proc. 5th ACM Conf. Functional Programming Languages and Computer Architecture*, pages 15–28. LNCS 523, 1991.

Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proc. 16th ACM Symp. Principles of Programming Languages*, pages 60–76, 1989.