Type inference in context

Adam Gundry Conor McBride
University of Strathclyde, Glasgow
{adam.gundry,conor.mcbride} at cis.strath.ac.uk

James McKinna
Radboud University, Nijmegen
james.mckinna at cs.ru.nl

Abstract

We consider the problems of first-order unification and type inference from a very general perspective on problem-solving, namely that of information increase in the problem context. This leads to a powerful technique for implementing type inference algorithms. We describe a unification algorithm and illustrate the technique by applying it to the familiar Hindley-Milner type system, but it can be applied to more advanced type systems. The algorithms depend on a well-founded invariant on contexts in which type variable bindings and type-schemes for terms may depend only on bindings appearing earlier in the context. We ensure that unification produces a most general unifier, and that type inference produces principal types, by advancing definitions earlier in the context only when absolutely necessary.

1. Introduction

Algorithm $\mathcal{W}$ is a well-known type inference algorithm, based on Robinson's Unification Algorithm [1965], for the Hindley-Milner type system [Milner 1978], verified by Damas and Milner [1982].

Successive presentations and formalisations of Algorithm $\mathcal{W}$ have treated the underlying unification algorithm as a 'black box', but by considering both simultaneously we are able to give a more elegant type inference algorithm. In particular, the generalisation step (for inferring the type of a let-expression) becomes straightforward.

This paper is literate Haskell, with full source code available at `http://personal.cis.strath.ac.uk/~adam/type-inference/`.

1.1 Motivating Context

Why revisit Algorithm $\mathcal{W}$? This is a first step towards a longer term objective: explaining the elaboration of high-level *dependently typed* programs into fully explicit calculi. Elaboration involves inferring *implicit arguments* by solving constraints, just as $\mathcal{W}$ specialises polymorphic type schemes, but with fewer algorithmic guarantees. Dependently typed programs are often constructed incrementally, with pieces arriving in an unpredictable order. Unification problems involve computations as well as constructors, and may evolve towards tractability even if not apparently solvable at first. Type inference without annotation is out of the question, but we may still exploit most general solutions to constraints when they exist. Milner's insights still serve us well, if not completely.

The existing literature on 'implicit syntax' [Norell 2007; Pollack 1990] neither fully nor clearly accounts for the behaviour of the systems in use today, nor are any of these systems free from murky corners. We feel the need to step back and gain perspective. Pragmatically, we need to account for stepwise progress in problem solving from states of partial knowledge.

What are such states of partial knowledge? In this paper, we model them by *contexts* occurring in typing judgments, describing the known properties of all variables in scope. We present algorithms via systems of inference rules defining relationships between assertions of the form $\Gamma \preceq \Delta \vdash S$. Here Γ is the input context (before applying the rule), S the statement to be established, and Δ the output context (in which S holds). We revisit Algorithm $\mathcal{W}$ as a sanity check that our perspective is helpful, as a familiar example of problem solving presented anew, and because our context discipline yields a clearer account of generalisation in let-binding.

This idea of assertions producing a resulting context goes back at least to Pollack [1990]. An interesting point of comparison is with the work of Nipkow and co-workers [Naraschewski and Nipkow 1999; Nipkow and Prehofer 1995], but substitutions and new contexts are there kept separate. We define an ordering on contexts based on their information content, and show that Δ is minimal with respect to this ordering. If one thinks of a context as a set of atomic facts, then Δ is the least upper bound of Γ together with the facts required for S to hold. In each case, at most one rule matches the input context and condition, and we specify a termination order so the rules define algorithms. These are straightforward to implement by translating the rule systems into appropriately monadic code. We illustrate this with our Haskell implementation.

Contexts here are not simply sets of assumptions, but lists containing information about type and term variables. The unification problem thus becomes finding a 'more informative' context in which two expressions are equivalent up to definition. Order of entries in a context is important. They are subject to well-foundedness conditions: any definition or declaration must be in terms of variables earlier in the context, as in dependent type theories. We obtain most general unifiers and principal types *just* by keeping entries as far to the right as possible, moving them left only when necessary to satisfy a constraint. Imposing order restrictions on context entries is similar to the *ordered hypotheses* of deduction systems for non-commutative logic [Polakow and Pfenning 1999].

In contrast to other presentations of unification and Hindley-Milner type inference, our algorithm uses explicit definitions to avoid the need for a substitution operation. (We do use substitution in reasoning about the system.) Many implementations of (variations

on) the Robinson unification algorithm are incorrect because they do not handle substitutions correctly [Norvig 1991].

This paper has been brewing for a long time. Its origins lie in a long lost constraint engine which McBride cannibalised from components of an implementation of Miller's 'mixed prefix' unification [Miller 1992], mutating the quantifier prefix into a context. McBride's thesis [McBride 1999] gives an early account of typing contexts representing the state of an interactive construction system, with 'holes' in programs and proofs as specially designated variables. Contexts come equipped with an information order: increase of information preserves typing and equality judgments; proof tactics are admissible context validity rules which increase information; unification is specified as a tactic which increases information to make an equation hold, but its implementation is not discussed. This view of construction underpinned the implementation of Epigram [McBride and McKinna 2004] and informed Norell's implementation of Agda [Norell 2007]. It is high time we began to explain how it works and perhaps to understand it.

2. Types and type variables

2.1 Syntax

The syntax of Hindley-Milner types is

$$\tau ::= \alpha \mid \tau \triangleright \tau$$

where α ranges over some set of type variables $\mathcal{V}_{\mathrm{TY}}$. For simplicity, we only consider one type constructor. In the sequel, α and β are type variables and τ and υ are types. Let Type be the set of types.

We write $FTV(\tau)$ for the free type variables of τ, defined as follows. This will later be extended to other syntactic objects.

$$FTV(\alpha) = \{\alpha\}$$
$$FTV(\tau \triangleright \upsilon) = FTV(\tau) \cup FTV(\upsilon)$$

The foldable functor Ty defines types in our object language parameterised by the type of variable names, which will be useful later. Thanks to a language extension in GHC 6.12 [GHC Team 2009] we can simply derive the required typeclass instances. For simplicity, we use integers as names in the implementation.

```
data Ty a = V a | Ty a ▷ Ty a
  deriving (Functor, Foldable)

type TyName = Integer
type Type   = Ty TyName
```

We implement FTV via a typeclass with membership function $(\in)$. We get most of the required instances for free using $\mathit{Foldable}$.

```
class FTV a where
  (∈) :: TyName → a → Bool

instance FTV TyName where
  (∈) = (≡)

instance (Foldable t, FTV a) ⇒ FTV (t a) where
  α ∈ t = any (α ∈) t
```

2.2 Introducing contexts

Types contain variables, but we need some way of interpreting what the variables mean. Our ideology is that such information belongs in the context. We give an abstract description of contexts, which may contain type variables and other information.

Let $\mathcal{K}$ be a set of sorts, and for each $K \in \mathcal{K}$ let $\mathcal{V}_K$ be a set of variables and $\mathcal{D}_K$ a set of *declaration properties*. Our running example will be the sort $\mathbf{TY}$, where $\mathcal{V}_{\mathrm{TY}}$ is some set of type variables and $\mathcal{D}_{\mathrm{TY}}$ initially contains only the 'unbound variable' property $:=?$. Properties of variables play the same atomic role in *derivations* that variables themselves play in terms.

A *context* is a list of declarations vD, with $v \in \mathcal{V}_K$ and $D \in \mathcal{D}_K$. The empty context is written ε. We let Γ, Δ, Θ range over contexts.

We will gradually construct a set $\mathcal{S}$ of statements, which can be judged in a context: these are the 'sorts' of our syntax of derivations. We write the *normal judgment* $\Gamma \vdash S$ to mean that the declarations in Γ support the statement $S \in \mathcal{S}$. We write the *neutral judgment* $\Gamma \Vdash S$ to mean that S follows directly from applying a fact in Γ. Neutral judgments capture exactly the legitimate appeals to assumptions in the context, just the way 'neutral terms' in λ-calculus are applied variables. We embed neutral into normal:

$$\textsc{Neutral}\ \frac{\Gamma \Vdash S}{\Gamma \vdash S}$$

It is not enough for contexts to be lists of declarations: they must be well-founded, that is, each declaration should make sense in *its* context. A context is valid if it declares each variable at most once, and each declaration property is meaningful in the preceding context. We maintain a map $\mathbf{ok}_K : \mathcal{D}_K \to \mathcal{S}$ for every $K \in \mathcal{K}$. Let $\mathcal{V}_K(\Gamma)$ be the set of K-variables in Γ. We define the context validity statement $\mathbf{valid}$ as shown in Figure 1.

$$\boxed{\Gamma \vdash \mathbf{valid}}$$

$$\frac{}{\varepsilon \vdash \mathbf{valid}} \qquad \frac{\Gamma \vdash \mathbf{valid} \quad \Gamma \vdash \mathbf{ok}_K D}{\Gamma, vD \vdash \mathbf{valid}}\ v \in \mathcal{V}_K \setminus \mathcal{V}_K(\Gamma)$$

Figure 1. Rules for context validity

From now on, we consider only valid contexts. All future definitions implicitly assume the context is valid, and it is straightforward to verify that our algorithms preserve context validity.

In the example of type declarations, we let $\mathbf{ok}_{\mathrm{TY}}(:=?) = \mathbf{valid}$. That is, declaring our ignorance is always reasonable.

2.3 Making types meaningful

Now we can ask whether a type is meaningful with respect to a context. This requires us to determine whether a type variable is in scope. In general, each context entry forces some statement to hold.

We suppose that there is a map $[\![\cdot]\!]_K : \mathcal{V}_K \times \mathcal{D}_K \to \mathcal{S}$ for each $K \in \mathcal{K}$, from declarations to statements. (We typically omit the subscript when the sort is irrelevant or can be inferred.) The idea is that $[\![vD]\!]$ is the statement that holds by virtue of the declaration vD in the context. The basic rule of our inference system is

$$\textsc{Lookup}\ \frac{\Gamma \vdash \mathbf{valid} \quad vD \in \Gamma}{\Gamma \Vdash [\![vD]\!]}.$$

As promised, uses of LOOKUP act as 'variables' in derivations. Our $[\![\cdot]\!]_K$ associates to an 'expression atom' its 'derivation atom'. This is the only rule which interrogates the context, hence we propose dropping the shared context from the presentation of all other rules.

We define the statement $\tau\ \mathbf{type}$ by taking $[\![\alpha :=?]\!] = \alpha\ \mathbf{type}$ together with the structural rule

$$\frac{\tau\ \mathbf{type} \quad \upsilon\ \mathbf{type}}{\tau \triangleright \upsilon\ \mathbf{type}}.$$

Note that we omit the context from rules if it is constant throughout. We observe the *sanity* condition $\Gamma \vdash \tau\ \mathbf{type} \Rightarrow \Gamma \vdash \mathbf{valid}$.

2.4 Conjunctions

We shall sometimes need to package multiple facts about a single variable, so we introduce the composite statement $S \wedge S'$ for statements S and S', with normal introduction rule (pairing) and neutral elimination rules (projections):

$$\frac{S \quad S'}{S \wedge S'} \qquad \frac{\Vdash S \wedge S'}{\Vdash S} \qquad \frac{\Vdash S \wedge S'}{\Vdash S'}$$

This is but one instance of a general pattern: we add normal introduction forms for composite statements, but supply eliminators only for composite *hypotheses*, in effect forcing derivations to be cut-free. This facilitates reasoning by induction on derivations. We shall ensure that the corresponding elimination rules for *normal* judgments are in any case admissible.

2.5 Type variable declarations

At the moment, variables are rather useless, because they can do nothing more than exist. During unification we will solve constraints to discover the values of variables, so we could then substitute them out. However, finding a value for a variable does not render it meaningless, in fact the reverse is true. We will therefore extend declarations instead, allowing variables to retain their values and hence their meaning. We extend $\mathcal{D}_{\mathrm{TY}}$ with bindings $:= \tau$ for every type τ, and let $\mathbf{ok}_{\mathrm{TY}}(:= \tau) = \tau\ \mathbf{type}$.

If Ξ is a list of type variable declarations, we define its set of free type variables $FTV(\Xi)$ by

$$FTV(\Xi) = \bigcup\{FTV(\tau) \mid \alpha := \tau \in \Xi\}.$$

If $X_0, \ldots, X_n$ are types or lists of type variable declarations then

$$FTV(X_0, \ldots, X_n) = FTV(X_0) \cup \ldots \cup FTV(X_n).$$

2.6 Type equations

Previously we could only consider the syntactic equality of types, but type variable declarations now induce a more interesting equational theory. If τ and υ are types, we define the equivalence statement $\tau \equiv \upsilon$ by making declarations yield equations:

$$[\![\alpha := \tau]\!]_{\mathrm{TY}} = \alpha\ \mathbf{type} \wedge \alpha \equiv \tau$$

and taking structural and equivalence closure by the rules in Figure 2. We observe the sanity condition

$$\Gamma \vdash \tau \equiv \upsilon \Rightarrow \Gamma \vdash \tau\ \mathbf{type} \wedge \upsilon\ \mathbf{type}.$$

2.7 Implementing types and contexts

A type variable declaration is given by a *TyEntry*, where a variable is either bound to a type (written $!\,\tau$) or left unbound (written ?).

```
data TyDecl   = ! Type | ?
data TyEntry  = TyName := TyDecl
```

Thanks to the relevant *Foldable* instances, all we need to do is define FTV for entries.

$$\boxed{\Gamma \vdash \tau \equiv \upsilon}$$

$$\frac{\tau\ \mathbf{type}}{\tau \equiv \tau} \qquad \frac{\upsilon \equiv \tau}{\tau \equiv \upsilon}$$

$$\frac{\tau_0 \equiv \upsilon_0 \quad \tau_1 \equiv \upsilon_1}{\tau_0 \triangleright \tau_1 \equiv \upsilon_0 \triangleright \upsilon_1} \qquad \frac{\tau_0 \equiv \tau_1 \quad \tau_1 \equiv \tau_2}{\tau_0 \equiv \tau_2}$$

Figure 2. Rules for type equivalence

```
instance FTV TyEntry where
  α ∈ (_:=! τ) = α ∈ τ
  α ∈ (_:=?)   = False
```

A context is a (backwards) list of entries. At the moment we only have one kind of entry, but later we will add others, so this definition is incomplete. The context validity conditions will be maintained by the algorithm but are not enforced by the type system; this is possible in a language such as Agda. A context suffix is a (forwards) list containing only type variable declarations.

```
data Entry = TY TyEntry | ...

type Context = Bwd Entry
type Suffix  = Fwd TyEntry
```

The types *Bwd* and *Fwd* are backwards (snoc) and forwards (cons) lists, respectively. We overload ε for the empty list in both cases, and write $:<$ and $:>$ for the backwards and forwards list data constructors. Lists are monoids where $\oplus$ is the append operator, and the 'fish' operator $(<><) :: Context \to Suffix \to Context$ appends a suffix to a context.

We work in the *Contextual* monad (computations that can fail and mutate the context), defined as follows:

```
type Contextual = StateT (TyName, Context) Maybe
```

The *TyName* component is the next fresh type variable name to use; it is an implementation detail not mentioned in the typing rules. The *fresh* function generates a fresh variable name and appends a declaration to the context. Our choice of *TyName* means that it is easy to choose a name fresh with respect to a *Context*.

```
fresh :: TyDecl → Contextual TyName
fresh d = do (β, Γ) ← get
             put (freshen β Γ, Γ :< TY (β := d))
             return β
  where freshen α Γ = succ α
```

The *getContext*, *putContext* and *modifyContext* functions respectively retrieve, replace and update the stored context. They correspond to *get*, *put* and *modify* in the *State* monad, but ignore the first component of the state.

```
getContext :: Contextual Context
getContext = gets snd

putContext :: Context → Contextual ()
putContext Γ = do β ← gets fst
                  put (β, Γ)

modifyContext :: (Context → Context) → Contextual ()
modifyContext f = getContext >>= putContext ∘ f
```

3. Information and stable statements

3.1 Information order

Intuitively, defining a variable cannot make equations cease to hold. More generally, if we rely on the context to tell us what we may deduce about variables, then making contexts more informative must preserve deductions.

Let Γ and Δ be contexts. A *substitution from* Γ *to* Δ is a map from $\mathcal{V}_{\mathrm{TY}}(\Gamma)$ to $\{\tau \mid \Delta \vdash \tau\ \mathbf{type}\}$. Substitutions apply to types and statements in the obvious way. Composition of substitutions is given by $(\theta \cdot \delta)(\alpha) = \theta(\delta\alpha)$. We write $[\tau/\alpha]$ for the substitution that maps α to τ and other variables to themselves.

We write $\delta : \Gamma \preceq \Delta$ and say Δ *is more informative than* Γ if δ is a substitution from Γ to Δ such that, for every $vD \in \Gamma$, we have that $\Delta \vdash \delta[\![vD]\!]$.

We write $\delta \equiv \theta : \Gamma \preceq \Delta$ if $\delta : \Gamma \preceq \Delta$, $\theta : \Gamma \preceq \Delta$ and for all $\alpha \in \mathcal{V}_{\mathrm{TY}}(\Gamma)$, $\Delta \vdash \delta\alpha \equiv \theta\alpha$. We will sometimes just write $\delta \equiv \theta$ if the contexts involved are obvious. It is straightforward to verify that $\equiv$ is an equivalence relation for fixed contexts Γ and Δ, and that if $\delta \equiv \theta$ then $\Delta \vdash \delta\tau \equiv \theta\tau$ for any Γ-type τ.

We may omit δ and write $\Gamma \preceq \Delta$ if we are only interested in the existence of a suitable substitution. This relation between contexts captures the notion of *information increase*: Δ supports all the statements corresponding to declarations in Γ.

This definition of information increase is not quite complete, because it does not place any constraints on the order of context entries, other than the dependency order of variables in declarations. We will later see how to extend $\preceq$ to capture the order of entries at an appropriate level of precision.

3.2 Stability

We say a statement S is *stable* if it is preserved under information increase, that is, if

$$\Gamma \vdash S \quad \text{and} \quad \delta : \Gamma \preceq \Delta \quad \Rightarrow \quad \Delta \vdash \delta S.$$

This says that we can extend a simultaneous substitution on syntax to a simultaneous substitution on derivations.

Since we are only interested in valid contexts, the statement $\mathbf{valid}$ always holds, and is invariant under substitution, so is clearly stable.

We have a standard strategy for proving stability of most statements, which is effective by construction. In each case we proceed by induction on the structure of derivations. Where the LOOKUP rule is applied, stability holds by the definition of information increase. Otherwise, for rules that do not refer to the context, we can verify that non-recursive hypotheses are stable and that recursive hypotheses occur in strictly positive positions, so they are stable by induction. Applying this strategy shows that the statements $\tau\ \mathbf{type}$ and $\tau \equiv \upsilon$ are stable.

Lemma 1. *If $[\![vD]\!]$ is stable for every declaration vD, then the $\preceq$ relation is a preorder, with reflexivity demonstrated by the inclusion substitution $\iota : \Gamma \preceq \Gamma : v \mapsto v$, and transitivity by composition:*

$$\delta : \Gamma \preceq \Delta \text{ and } \theta : \Delta \preceq \Theta \quad \Rightarrow \quad \theta \cdot \delta : \Gamma \preceq \Theta.$$

Proof. Reflexivity follows immediately from the LOOKUP rule. For transitivity, suppose $vD \in \Gamma$, then $\Delta \vdash \delta[\![vD]\!]$ since $\delta : \Gamma \preceq \Delta$. Now by stability applied to $\delta[\![vD]\!]$ using θ, we have $\Theta \vdash \theta\delta[\![vD]\!]$ as required. □

Lemma 2. *If $\delta_0 \equiv \delta_1 : \Gamma \preceq \Delta$ and $\theta_0 \equiv \theta_1 : \Delta \preceq \Theta$ then $\theta_0 \cdot \delta_0 \equiv \theta_1 \cdot \delta_1 : \Gamma \preceq \Theta$.*

Proof. Fix $\alpha \in \mathcal{V}_{\mathrm{TY}}(\Gamma)$. By definition of $\equiv$, $\Delta \vdash \delta_0\alpha \equiv \delta_1\alpha$, so by stability, $\Theta \vdash \theta_0\delta_0\alpha \equiv \theta_0\delta_1\alpha$. Moreover $\Theta \vdash \theta_0\delta_1\alpha \equiv \theta_1\delta_1\alpha$, and hence $\Theta \vdash \theta_0\delta_0\alpha \equiv \theta_1\delta_1\alpha$ by transitivity. □

3.3 Composite statements

If S is a statement and vD is a declaration, then we define the composite statement $(vD \succ S)$ with the introduction rule

$$\frac{\Gamma \vdash \mathbf{ok}_K D \quad \Gamma, vD \vdash S}{\Gamma \vdash (vD \succ S)}\ v \in \mathcal{V}_K \setminus \mathcal{V}_K(\Gamma).$$

and neutral elimination rule

$$\frac{\Gamma \Vdash (\alpha D \succ S) \quad \Gamma \vdash [\tau/\alpha][\![\alpha D]\!]}{\Gamma \Vdash [\tau/\alpha]S}\ D \in \mathcal{D}_{\mathrm{TY}}$$

Lemma 3 (Composition preserves stability)**.** *If S and S' are stable then $S \wedge S'$ is stable. If vD is a declaration and both $\mathbf{ok}_K D$ and S are stable, then $(vD \succ S)$ is stable.*

Proof. Suppose $\delta : \Gamma \preceq \Delta$, the statements S and S' are stable and $\Gamma \vdash (S \wedge S')$. If the proof is by LOOKUP then $\Delta \vdash \delta(S \wedge S')$ by definition of information increase. Otherwise $\Gamma \vdash S$ and $\Gamma \vdash S'$, so by stability, $\Delta \vdash \delta S$ and $\Delta \vdash \delta S'$, so $\Delta \vdash \delta(S \wedge S')$.

Suppose $\delta : \Gamma \preceq \Delta$, the statement S is stable and $\Gamma \vdash (vD \succ S)$. If the proof is by LOOKUP then $\Delta \vdash \delta S$ by definition of information increase. Otherwise, $\Gamma \vdash \mathbf{ok}_K D$ and $\Gamma, vD \vdash S$, so by stability, $\Delta \vdash \delta\ \mathbf{ok}_K D$. Now $\delta : \Gamma, vD \preceq \Delta, v(\delta D)$ so by stability of S we have $\Delta, v(\delta D) \vdash \delta S$. Hence $\Delta \vdash (v(\delta D) \succ \delta S)$ and so $\Delta \vdash \delta(vD \succ S)$. □

Thanks to Lemma 3 and the foregoing, every statement we have introduced so far is stable. We will ensure stability for all statements in $\mathcal{S}$, so we can exploit it without qualification in the sequel.

4. Problems

4.1 What is a problem?

A problem represents a statement we wish to make hold by increasing information in the context. More generally, it is a statement with distinguished output positions for which we wish to find a witness in a more informative context. Unification is an example of the first kind of problem and type inference the second.

We are interested in creating algorithms to solve problems, preferably in as general a way as possible (that is, by making the smallest information increase necessary to find a solution). This corresponds to finding a most general unifier, in the case of unification, or a principal type in the case of type inference.

Distinguishing output positions with angle brackets $\langle\cdot\rangle$, formally, a *problem* P consists of

- sets In_P and Out_P of input and output parameters,
- a precondition map $Pre_P(\cdot) : In_P \to \mathcal{S}$,
- a postcondition map $P(\cdot)\langle\cdot\rangle : In_P \to Out_P \to \mathcal{S}$ and
- a relation map $R_P\langle\cdot\rangle\langle\cdot\rangle : Out_P \to Out_P \to \mathcal{S}$,

such that In_P and Out_P are closed under substitution and the maps respect substitution, for example, $Pre_P(\theta a) = \theta Pre_P(a)$. Moreover, for any Γ, $a \in In_P$ and $r, s, t \in Out_P$ such that

$$\Gamma \vdash Pre_P(a) \wedge P(a)\langle r\rangle \wedge P(a)\langle s\rangle \wedge P(a)\langle t\rangle,$$

we must have $\Gamma \vdash R_P\langle r\rangle\langle r\rangle$ and

$$\Gamma \vdash R_P\langle r\rangle\langle s\rangle \wedge R_P\langle s\rangle\langle t\rangle \Rightarrow \Gamma \vdash R_P\langle r\rangle\langle t\rangle.$$

The unification problem U is given by

$$\begin{aligned} In_U &= \mathit{Type} \times \mathit{Type} \\ Out_U &= \{1\} \\ Pre_U(\tau, \upsilon) &= \tau\ \mathbf{type} \wedge \upsilon\ \mathbf{type} \\ U(\tau, \upsilon)\langle\_\rangle &= \tau \equiv \upsilon \\ R_U\langle\_\rangle\langle\_\rangle &= \mathbf{valid} \end{aligned}$$

A *P-instance for a context* Γ is $a \in In_P$ such that $\Gamma \vdash Pre_P(a)$. The problem instance a has *solution* (r, δ, Δ) if $r \in Out_P$ and $\delta : \Gamma \preceq \Delta$ such that $\Delta \vdash P(\delta a)\langle r\rangle$. (Observe that $\Delta \vdash Pre_P(\delta a)$ by stability.)

The solution (r, δ, Δ) is *minimal* if for any solution (s, θ, Θ) there exists $\zeta : \Delta \preceq \Theta$ such that $\theta \equiv \zeta \cdot \delta$ and $\Theta \vdash R_P\langle \zeta r\rangle\langle s\rangle$.

We write $\delta : \Gamma \mathrel{\hat{\preceq}} \Delta \vdash P(a)\langle r\rangle$ to mean that (r, δ, Δ) is a minimal solution of the P-instance r.

4.2 The Optimist's Lemma

If P and Q are problems, then $P \wedge Q$ is a problem with

$$\begin{aligned} In_{P\wedge Q} &= In_P \times In_Q \\ Out_{P\wedge Q} &= Out_P \times Out_Q \\ Pre_{P\wedge Q}(a, b) &= Pre_P(a) \wedge Pre_Q(b) \\ (P \wedge Q)(a, b)\langle r, s\rangle &= P(a)\langle r\rangle \wedge Q(b)\langle s\rangle \\ R_{P\wedge Q}\langle r, s\rangle\langle t, u\rangle &= R_P\langle r\rangle\langle t\rangle \wedge R_Q\langle s\rangle\langle u\rangle \end{aligned}$$

The point of all this machinery is to be able to state and prove the following lemma, stating that the minimal solution to a conjunction of problems can be found by finding the minimal solution of the first problem, then (minimally) extending it to solve the second.

Lemma 4 (The Optimist's Lemma)**.** *The following inference rule is admissible:*

$$\frac{\delta : \Gamma \mathrel{\hat{\preceq}} \Delta \vdash P(a)\langle r\rangle \quad \theta : \Delta \mathrel{\hat{\preceq}} \Theta \vdash Q(b)\langle s\rangle}{\theta \cdot \delta : \Gamma \mathrel{\hat{\preceq}} \Theta \vdash P(a)\langle \theta r\rangle \wedge Q(b)\langle s\rangle}.$$

Sketch. The solutions of $P(a)$ arise exactly by extending δ, so if we seek also to solve $Q(b)$, it is necessary and sufficient to search amongst the extensions of δ. For details, see appendix. □

This sequential approach to problem solving is not the only decomposition justified by stability. The account of unification given by McAdam [1998] amounts to a concurrent, transactional decomposition of problems. The same context is extended via multiple different substitutions, then these are unified to produce a single substitution.

5. Deriving a unification algorithm

5.1 Transforming the rule system for equivalence

We wish to transform these rules into a unification algorithm. Starting with the rules in Figure 2, consider what happens if we remove each equivalence closure rule in turn and attempt to prove its admissibility. This will fail, but the proof obligations left over give us a more specific but equivalent system of algorithmic-looking rules for equivalence.

First, the reflexivity rule for types can be derived from the reflexivity rule for variables given by

$$\frac{\alpha\ \mathbf{type}}{\alpha \equiv \alpha}$$

by applying the structural rule until variables occur.

Next, transitivity can be derived from

$$\frac{\alpha \equiv \tau \quad \tau \equiv \upsilon}{\alpha \equiv \upsilon}\ \alpha \neq \tau, \alpha \neq \upsilon$$

as follows. Suppose $\chi \equiv \tau$ and $\tau \equiv \upsilon$ and seek to prove $\chi \equiv \upsilon$.

- If $\chi = \alpha$ is a variable distinct from τ and υ then we can use the restricted transitivity rule.
- If $\chi = \alpha = \upsilon$ then we can use reflexivity.
- If $\chi = \alpha = \tau$ then the result holds by hypothesis.
- If χ is not a variable but υ is then we can apply symmetry and one of the previous cases.
- If χ and υ are both not variables then we can apply the structural rule.

Finally, symmetry becomes admissible (but not derivable) if replaced by

$$\frac{\alpha \equiv \tau}{\tau \equiv \alpha}.$$

Note that the restricted symmetry rule covers the case we needed for deriving transitivity. Suppose $\upsilon \equiv \tau$ and seek to prove $\tau \equiv \upsilon$.

- If $\upsilon = \alpha$ is a variable then the rule applies.
- If υ is not a variable but $\tau = \beta$ is, then the proof of $\upsilon \equiv \beta$ must be by restricted symmetry, in which case its hypothesis says that $\beta \equiv \upsilon$.
- If τ and υ are both not variables then we can apply the structural rule.

5.2 Constructing a unification algorithm

Now we can see how to construct the algorithm. The structural rule says that whenever we have rigid $\triangleright$ symbols on each side, we decompose the problem into two subproblems, and thanks to the Optimist's Lemma we can solve these sequentially. Otherwise, we either have variables on both sides, or a variable on one side and a type on the other. In each case, we look at the head of the context to see what information it gives us, and use the transformed rules to see how to proceed. When solving a variable with a type, we need to accumulate the type's dependencies as we encounter them, performing the occur check to ensure a solution exists.

It is possible that a context entry may have no bearing on the unification problem being solved, and hence can be ignored. We define the orthogonality relation $\upsilon D \perp X$ (the set of type variables

X does not depend on the declaration vD) thus:

$$\alpha D \perp X \text{ if } \alpha \in \mathcal{V}_{\mathrm{TY}} \setminus X$$
$$vD \perp X \text{ if } v \in \mathcal{V}_K, D \in \mathcal{D}_K \text{ for } K \neq \mathbf{TY}$$

The rules in Figure 3 define our unification algorithm. The assertion $\Gamma \preceq \Delta \vdash \tau \equiv \upsilon$ means that given inputs Γ, τ and υ, where $\Gamma \vdash \tau\ \mathbf{type} \wedge \upsilon\ \mathbf{type}$, unification of τ with υ succeeds, producing output context Δ.

The assertion $\Gamma \mid \Xi \preceq \Delta \vdash \alpha \equiv \tau$ means that given inputs Γ, Ξ, α and τ, solving α with τ succeeds and produces output context Δ, subject to the conditions

- $\alpha \in \mathcal{V}_{\mathrm{TY}}(\Gamma)$,
- $\Gamma, \Xi \vdash \tau\ \mathbf{type}$,
- τ is not a variable,
- Ξ contains only type variable declarations and
- $\beta \in \mathcal{V}_{\mathrm{TY}}(\Xi) \Rightarrow \beta \in FTV(\tau, \Xi)$.

For clarity, we take a 'garbage-in, garbage-out' approach to the algorithm: we omit the above sanity conditions from the rules, and correspondingly do not check them in the implementation.

The rules DEFINE, EXPAND and IGNORE have symmetric counterparts that are identical apart from interchanging the equated terms in the conclusion. Usually we will ignore these without loss of generality.

Observe that we have no rule for the case

$$\Gamma_0, \alpha D \mid \Xi \preceq \Delta \vdash \alpha \equiv \tau \text{ with } \alpha \in FTV(\tau, \Xi)$$

so the algorithm fails if this situation arises. This is essentially an occur check failure: α and τ cannot be unified if α occurs in τ or in an entry that τ depends on, and τ is not a variable. Since we only have one type constructor symbol (the function arrow $\triangleright$), there are no failures due to rigid-rigid mismatch. Adding these would not significantly complicate matters, however.

By exposing the contextual structure underlying unification we make termination of the algorithm evident. Each recursive appeal to unification (directly or via the solving process) either shortens the context or preserves the context and decomposes types [McBride 2003]. We are correspondingly entitled to reason about the total correctness of unification by induction on the algorithmic rules.

5.3 Soundness and completeness

Lemma 5 (Soundness of unification)**.** *(a) If $\Gamma \preceq \Delta \vdash \tau \equiv \upsilon$, then $\mathcal{V}_{\mathrm{TY}}(\Gamma) = \mathcal{V}_{\mathrm{TY}}(\Delta)$, $\Delta \vdash \tau \equiv \upsilon$ and $\iota : \Gamma \preceq \Delta$ where ι is the inclusion substitution.*

(b) If $\Gamma \mid \Xi \preceq \Delta \vdash \alpha \equiv \tau$, then $\mathcal{V}_{\mathrm{TY}}(\Gamma, \Xi) = \mathcal{V}_{\mathrm{TY}}(\Delta)$, $\Delta \vdash \alpha \equiv \tau$ and $\iota : \Gamma, \Xi \preceq \Delta$.

Proof. By induction on the structure of derivations. □

Lemma 6 (Occur check)**.** *Let α be a variable and τ a non-variable type such that $\alpha \in FTV(\tau)$. For every context Γ and substitution θ, $\Gamma \nvdash \theta\alpha \equiv \theta\tau$ and $\Gamma \nvdash \theta\tau \equiv \theta\alpha$.*

Proof. It suffices to prove $\Gamma \nvdash \alpha \equiv \tau$ and $\Gamma \nvdash \tau \equiv \alpha$, because $\theta\alpha$ must contain a variable $\beta \in FTV(\theta\tau)$ and $\theta\tau$ is not a variable.

Since α is a variable but τ is not, neither reflexivity nor the structural rule apply. Symmetry and transitivity do not apply because their hypotheses cannot be satisifed.

$$\boxed{\Gamma \preceq \Delta \vdash \tau \equiv \upsilon}$$

$$\textsc{Idle}\ \frac{}{\Gamma_0, \alpha D \preceq \Gamma_0, \alpha D \vdash \alpha \equiv \alpha}$$

$$\textsc{Define}\ \frac{}{\Gamma_0, \alpha{:=}? \preceq \Gamma_0, \alpha{:=}\beta \vdash \alpha \equiv \beta}\ \alpha \neq \beta$$

$$\textsc{Expand}\ \frac{\Gamma_0 \preceq \Delta_0 \vdash \tau \equiv \beta}{\Gamma_0, \alpha{:=}\tau \preceq \Delta_0, \alpha{:=}\tau \vdash \alpha \equiv \beta}\ \alpha \neq \beta$$

$$\textsc{Ignore}\ \frac{\Gamma_0 \preceq \Delta_0 \vdash \alpha \equiv \beta}{\Gamma_0, vD \preceq \Delta_0, vD \vdash \alpha \equiv \beta}\ vD \perp \{\alpha, \beta\}$$

$$\textsc{Solve}\ \frac{\Gamma \mid \varepsilon \preceq \Delta \vdash \alpha \equiv \tau}{\Gamma \preceq \Delta \vdash \alpha \equiv \tau}\ \tau \text{ not variable}$$

$$\textsc{Decompose}\ \frac{\Gamma \preceq \Delta_0 \vdash \tau_0 \equiv \upsilon_0 \quad \Delta_0 \preceq \Delta \vdash \tau_1 \equiv \upsilon_1}{\Gamma \preceq \Delta \vdash \tau_0 \triangleright \tau_1 \equiv \upsilon_0 \triangleright \upsilon_1}$$

$$\boxed{\Gamma \mid \Xi \preceq \Delta \vdash \alpha \equiv \tau}$$

$$\textsc{DefineS}\ \frac{}{\Gamma_0, \alpha{:=}? \mid \Xi \preceq \Gamma_0, \Xi, \alpha{:=}\tau \vdash \alpha \equiv \tau}\ \alpha \notin FTV(\tau, \Xi)$$

$$\textsc{ExpandS}\ \frac{\Gamma_0, \Xi \preceq \Delta_0 \vdash \upsilon \equiv \tau}{\Gamma_0, \alpha{:=}\upsilon \mid \Xi \preceq \Delta_0, \alpha{:=}\upsilon \vdash \alpha \equiv \tau}\ \alpha \notin FTV(\tau, \Xi)$$

$$\textsc{DependS}\ \frac{\Gamma_0 \mid \beta D, \Xi \preceq \Delta \vdash \alpha \equiv \tau}{\Gamma_0, \beta D \mid \Xi \preceq \Delta \vdash \alpha \equiv \tau}\ \alpha \neq \beta, \beta \in FTV(\tau, \Xi)$$

$$\textsc{IgnoreS}\ \frac{\Gamma_0 \mid \Xi \preceq \Delta_0 \vdash \alpha \equiv \tau}{\Gamma_0, vD \mid \Xi \preceq \Delta_0, vD \vdash \alpha \equiv \tau}\ vD \perp FTV(\alpha, \tau, \Xi)$$

Figure 3. Algorithmic rules for unification

By the well-formedness conditions for contexts, if $\alpha{:=}\upsilon \in \Gamma$ then $\alpha \notin FTV(\upsilon)$, so the LOOKUP rule does not apply. □

Lemma 7 (Completeness and generality of unification)**.** *(a) If $\theta : \Gamma \preceq \Theta$, $\Gamma \vdash \upsilon\ \mathbf{type} \wedge \tau\ \mathbf{type}$ and $\Theta \vdash \theta\upsilon \equiv \theta\tau$, then there is some context Δ such that $\Gamma \hat{\preceq} \Delta \vdash \upsilon \equiv \tau$.*

(b) Moreover, if $\theta : \Gamma, \Xi \preceq \Theta$ is such that $\Theta \vdash \theta\alpha \equiv \theta\tau$ and

- *$\alpha \in \mathcal{V}_{\mathrm{TY}}(\Gamma)$,*
- *$\Gamma, \Xi \vdash \tau\ \mathbf{type}$,*
- *τ is not a variable,*
- *Ξ contains only type variable declarations and*
- *$\beta \in \mathcal{V}_{\mathrm{TY}}(\Xi) \Rightarrow \beta \in FTV(\tau, \Xi)$,*

then there is some context Δ such that $\Gamma \mid \Xi \hat{\preceq} \Delta \vdash \alpha \equiv \tau$.

Sketch. Each step preserves all solutions. The Optimist's Lemma justifies problem decomposition. The algorithm terminates, and the

only case not covered by the rules is the case where the occur check fails, indicating that no unifer exists. For details, see appendix. □

5.4 Implementing unification

First, we define some helpful machinery. The *onTop* operator applies its argument to the topmost type variable declaration in the context, skipping over any other kinds of entry. The argument function may *restore* the previous entry by returning *Nothing*, or it may return a context extension (that contains at least as much information as the entry that has been removed) with which to *replace* it.

```
onTop :: (TyEntry → Contextual (Maybe Suffix))
            → Contextual ()
onTop f = do
  Γ :< vD ← getContext
  putContext Γ
  case vD of
    TY αD → do m ← f αD
               case m of
                 Just Ξ   → modifyContext (<>< Ξ)
                 Nothing → modifyContext (:<vD)
    _       → onTop f ≫ modifyContext (:<vD)

restore :: Contextual (Maybe Suffix)
restore = return Nothing

replace :: Suffix → Contextual (Maybe Suffix)
replace = return ∘ Just
```

The *unify* function attempts to modify the context to produce a most general unifier for the two given types; it will fail if the types cannot be unified given the current state of the context.

```
unify :: Type → Type → Contextual ()
unify (V α) (V β) = onTop $ λ(γ := d) → case
  (γ ≡ α, γ ≡ β, d ) of
  (True,  True,  _ )    → restore
  (True,  False, ? )    → replace (α:=!(V β) :> ε)
  (False, True,  ? )    → replace (β:=!(V α) :> ε)
  (True,  False, !τ)    → unify (V β) τ      ≫ restore
  (False, True,  !τ)    → unify (V α) τ      ≫ restore
  (False, False, _ )    → unify (V α) (V β) ≫ restore
unify (V α)   τ         = solve α ε τ
unify τ       (V α)     = solve α ε τ
unify (τ0 ▷ τ1) (υ0 ▷ υ1) = unify τ0 υ0 ≫ unify τ1 υ1
```

The *solve* function attempts to unify a variable name with a (non-variable) type, given a list of entries that the type depends on, which must be placed into the context before it.

```
solve :: TyName → Suffix → Type → Contextual ()
solve α Ξ τ = onTop $ λ(γ := d) →
  let occurs = γ ∈ τ ∨ γ ∈ Ξ in case
  (γ ≡ α, occurs, d ) of
  (True,  True,  _ ) → fail "Occur check failed"
  (True,  False, ? ) → replace (Ξ ⊕ (α:=!τ :> ε))
  (True,  False, !υ) → modifyContext (<>< Ξ)
                       ≫ unify υ τ
                       ≫ restore
  (False, True,  _ ) → solve α (γ := d :> Ξ) τ
                       ≫ replace ε
  (False, False, _ ) → solve α Ξ τ
                       ≫ restore
```

6. The type inference problem

6.1 Introducing type schemes

Having implemented unification, we now turn to the problem of type inference for terms. We will reuse the abstract framework already introduced, defining a new sort TM for term variables. To handle polymorphism, these need to be associated with type schemes rather than monomorphic types.

A *type scheme* σ is a type wrapped in one or more $\forall$ quantifiers or let bindings, with the syntax

$$\sigma ::= .\tau \mid \forall\alpha\ \sigma \mid (!\alpha{:=}\tau \text{ in } \sigma).$$

We use explicit definitions in type schemes to avoid the need for substitution in the type inference algorithm.

We define a new statement σ **scheme** by the rules in Figure 4. We observe the sanity condition $\Gamma \vdash \sigma$ **scheme** $\Rightarrow \Gamma \vdash$ **valid**.

$$\boxed{\Gamma \vdash \sigma \textbf{ scheme}}$$

$$\frac{\tau \textbf{ type}}{.\tau \textbf{ scheme}} \qquad \frac{(\alpha{:=}? \succ \sigma \textbf{ scheme})}{\forall\alpha\ \sigma \textbf{ scheme}}$$

$$\frac{\upsilon \textbf{ type} \quad (\alpha{:=}\upsilon \succ \sigma \textbf{ scheme})}{(!\alpha{:=}\upsilon \text{ in } \sigma) \textbf{ scheme}}$$

Figure 4. Rules for scheme validity

The structure of these rules strongly suggests that schemes arise by discharging a list of type variable declarations over a type. In fact, any scheme can be viewed in this way. We write $(\Xi \Uparrow \sigma)$ for the generalisation of the type scheme σ over the list of type variable declarations Ξ, defined by

$$\begin{aligned} \varepsilon \Uparrow \sigma &= \sigma \\ \Xi, \alpha{:=}? \Uparrow \sigma &= \Xi \Uparrow \forall\alpha\ \sigma \\ \Xi, \alpha{:=}\nu \Uparrow \sigma &= \Xi \Uparrow (!\alpha{:=}\nu \text{ in } \sigma) \end{aligned}$$

We will usually be interested in the case $\sigma = .\tau$ for some type τ.

When we infer the specialised type of a variable, we rely on the ability to invert this operation, extending the contex with a *fresh* copy of a scheme's prefix. As shown above, we follow Naraschewski and Nipkow [1999] in achieving freshness with a simple counter, built into the *Contextual* monad.

Lemma 8. *If* $\Gamma \vdash \sigma$ **scheme** *then* $\sigma = \Xi \Uparrow .\tau$ *for some* Ξ *and* τ *such that* $\Gamma, \Xi \vdash \tau$ **type**

Proof. By induction on the structure of σ, given that it is possible to choose fresh variable names. □

6.2 Implementing type schemes

It is convenient to represent bound variables by de Brujin indices and free variables (i.e. those defined in the context) by names [McBride and McKinna 2004]. Moreover, we use the Haskell type system to prevent some incorrect manipulations of indices by defining a 'successor' type [Bellegarde and Hook 1994; Bird and Paterson 1999]

```
data Index a = Z | S a
  deriving (Functor, Foldable)
```

We can then represent schemes as

$$\begin{aligned}&\mathbf{data}\ \mathit{Schm}\ a = \mathit{Type}\ (\mathit{Ty}\ a)\\ &\qquad\qquad\ \mid \mathit{All}\ (\mathit{Schm}\ (\mathit{Index}\ a))\\ &\qquad\qquad\ \mid \mathit{LetS}\ (\mathit{Ty}\ a)\ (\mathit{Schm}\ (\mathit{Index}\ a))\\ &\quad \mathbf{deriving}\ (\mathit{Functor}, \mathit{Foldable})\end{aligned}$$

$$\mathbf{type}\ \mathit{Scheme} = \mathit{Schm}\ \mathit{TyName}$$

The outermost bound variable is represented by Z and the other variables are wrapped in the S constructor. For example, the type scheme $\forall\alpha\forall\beta.\beta \triangleright 2$ is represented as

$$\mathit{All}\ (\mathit{All}\ (\mathit{Type}\ (V\ (S\ Z) \triangleright V\ (S\ (S\ 2)))))$$

Note that the code forces us to distinguish a type τ and its corresponding type scheme (written $.\tau$), as the latter will be represented by $\mathit{Type}\ \tau :: \mathit{Scheme}$.

Implementing the generalisation function ($\Uparrow$) is straightforward:

$$\begin{aligned}&(\Uparrow) :: \mathit{Bwd}\ \mathit{TyEntry} \to \mathit{Scheme} \to \mathit{Scheme}\\ &\varepsilon \Uparrow \sigma = \sigma\\ &(\Xi :< \alpha := d) \Uparrow \sigma = \mathbf{case}\ d\ \mathbf{of}\\ &\qquad\qquad ?\ \ \to \Xi \Uparrow \mathit{All}\ \sigma'\\ &\qquad\qquad !\nu \to \Xi \Uparrow \mathit{LetS}\ \nu\ \sigma'\\ &\quad \mathbf{where}\\ &\qquad \sigma' = \mathit{fmap}\ \mathit{bind}\ \sigma\\ &\qquad \mathit{bind}\ \beta \mid \alpha \equiv \beta \quad\ = Z\\ &\qquad\qquad\quad \mid \mathit{otherwise} = S\ \beta\end{aligned}$$

Conversely, we can *specialise* a type scheme by extending the context with fresh variables to produce a type.

$$\begin{aligned}&\mathit{specialise} :: \mathit{Scheme} \to \mathit{Contextual}\ \mathit{Type}\\ &\mathit{specialise}\ (\mathit{Type}\ \tau) = \mathit{return}\ \tau\\ &\mathit{specialise}\ \sigma = \mathbf{do}\\ &\qquad \mathbf{let}\ (d, \sigma') = \mathit{unpack}\ \sigma\\ &\qquad \beta \leftarrow \mathit{fresh}\ d\\ &\qquad \mathit{specialise}\ (\mathit{fmap}\ (\mathit{fromS}\ \beta)\ \sigma')\\ &\quad \mathbf{where}\\ &\qquad \mathit{unpack}\ (\mathit{All}\ \sigma') = (?, \sigma')\\ &\qquad \mathit{unpack}\ (\mathit{LetS}\ \tau\ \sigma') = (!\tau, \sigma')\\ &\qquad \mathit{fromS}\ \beta\ Z \quad\ = \beta\\ &\qquad \mathit{fromS}\ \beta\ (S\ \alpha) = \alpha\end{aligned}$$

6.3 Type assignment system

Let $\mathcal{V}_{\mathrm{TM}}$ be some set of term variables and let x range over $\mathcal{V}_{\mathrm{TM}}$. Term variable declarations $\mathcal{D}_{\mathrm{TM}}$ are scheme assignments of the form $\hat{::}\,\sigma$, with $\mathbf{ok}_{\mathrm{TM}}(\hat{::}\,\sigma) = \sigma\ \mathbf{scheme}$.

The syntax of terms is

$$t ::= x \mid t\ t \mid \lambda x.t \mid \mathrm{let}\ x := t\ \mathrm{in}\ t.$$

Let Term be the set of terms.

We define the type assignability statement $t : \tau$ by the rules in Figure 5, and the scheme assignability statement $t :: \sigma$ for arbitrary terms t and schemes σ thus:

$$\begin{aligned}t :: .\tau &\mapsto t : \tau\\ t :: \forall\alpha\sigma &\mapsto (\alpha := ? \succ t :: \sigma)\\ t :: (!\alpha := \tau\ \mathrm{in}\ \sigma) &\mapsto (\alpha := \tau \succ t :: \sigma)\end{aligned}$$

We observe the sanity conditions $\Gamma \vdash x : \tau \Rightarrow \Gamma \vdash \tau\ \mathbf{type}$ and $\Gamma \vdash x :: \sigma \Rightarrow \Gamma \vdash \sigma\ \mathbf{scheme}$.

We define $[\![x \hat{::} \sigma]\!]_{\mathrm{TM}} = x :: \sigma$, so $\Gamma \preceq \Delta$ requires Δ to assign a term variable all the types that Γ assigns it, but allows x to become more polymorphic and acquire new types. This notion certainly retains stability: every variable lookup can be simulated in the more general context. However, it allows arbitrary generalisation of the schemes assigned to term variables which are incompatible with the known and intended value of those variables.

$$\boxed{\Delta \vdash t : \tau}$$

$$\frac{(x \hat{::} .\upsilon \succ t : \tau)}{\lambda x.t : \upsilon \triangleright \tau} \qquad \frac{f : \upsilon \triangleright \tau \quad a : \upsilon}{fa : \tau}$$

$$\frac{s :: \sigma \quad (x \hat{::} \sigma \succ w : \tau)}{\mathrm{let}\ x := s\ \mathrm{in}\ w : \tau} \qquad \frac{t : \tau \quad \tau \equiv \upsilon}{t : \upsilon}$$

Figure 5. Declarative rules for type assignment

As Wells [2002] points out, Hindley-Milner type inference is not in this respect compositional. He carefully distinguishes principal *typings*, given the right to demand more polymorphism, from Milner's principal *type schemes* and analyses how the language of types must be extended to express principal typings.

We, too, note this distinction. We cannot hope to find principal types with respect to $\preceq$, so we capture Milner's compromise by defining a sub-relation $\sqsubseteq$, by $\delta : \Gamma \sqsubseteq \Delta$ if $\delta : \Gamma \preceq \Delta$ and

$$x \hat{::} \sigma \in \Gamma \ \Rightarrow\ x \hat{::} \delta\sigma \in \Delta.$$

Thus, if $\Gamma \sqsubseteq \Delta$, then Δ assigns the *same* type schemes to term variables as Γ does (modulo substitution).

As with unification, we wish to translate these declarative rules into an algorithm for type inference. We define the type inference problem I by

$$\begin{aligned}\mathit{In}_I &= \mathit{Term}\\ \mathit{Out}_I &= \mathit{Type}\\ \mathit{Pre}_I(t) &= \mathbf{valid}\\ I(t)\langle\tau\rangle &= t : \langle\tau\rangle\\ R_I\langle\tau\rangle\langle\upsilon\rangle &= \langle\tau\rangle \equiv \langle\upsilon\rangle\end{aligned}$$

6.4 Implementing terms

We extend the *Entry* data type to include declarations of term variables. It is still not quite complete, however.

$$\begin{aligned}&\mathbf{type}\ \mathit{TmName} = \mathit{String}\\ &\mathbf{data}\ \mathit{TmEntry} = \mathit{TmName} \hat{::} \mathit{Scheme}\\ &\\ &\mathbf{data}\ \mathit{Entry} = \mathit{TY}\ \ \mathit{TyEntry} \mid \mathit{TM}\ \ \mathit{TmEntry} \mid \ldots\end{aligned}$$

A term t may be a variable (X), an application ($:\$$), an abstraction ($\lambda \cdot .\cdot$) or a let binding ($\mathrm{let}\ \cdot := \cdot\ \mathrm{in}\ \cdot$). As with Ty, we parameterise over the type of term variable names, so Tm is a foldable functor.

$$\begin{aligned}&\mathbf{data}\ \mathit{Tm}\ a = X\ a\\ &\qquad\qquad\ \mid \mathit{Tm}\ a :\$\ \mathit{Tm}\ a\\ &\qquad\qquad\ \mid \lambda a.\mathit{Tm}\ a\\ &\qquad\qquad\ \mid \mathrm{let}\ a := \mathit{Tm}\ a\ \mathrm{in}\ \mathit{Tm}\ a\\ &\quad \mathbf{deriving}\ (\mathit{Functor}, \mathit{Foldable})\\ &\\ &\mathbf{type}\ \mathit{Term} = \mathit{Tm}\ \mathit{TmName}\end{aligned}$$

7. Local contexts for local problems

7.1 Preserving order in the context

We have previously observed, but not yet exploited, the importance of declaration order in the context, and that we move declarations left as little as possible. Thus rightmost entries are those most local to the problem we are solving. This will be useful when we come to implement type inference for the $\mathrm{let}\ \cdot := \cdot\ \mathrm{in}\ \cdot$ construct, as we want to generalise over 'local' type variables but not 'global' variables.

In order to keep track of locality in the context, we need another kind of context entry: the $\fatsemi$ separator. We add a new validity rule

$$\frac{\Gamma \vdash \mathbf{valid}}{\Gamma \fatsemi \vdash \mathbf{valid}}$$

so the (finally) complete data type of context entries is:

$$\mathbf{data}\ \mathit{Entry} = \mathit{TY}\ \mathit{TyEntry} \mid \mathit{TM}\ \mathit{TmEntry} \mid \fatsemi$$

We also refine the $\preceq$ relation. Let $\lfloor$ be the partial function from contexts and natural numbers to contexts taking $\Gamma \lfloor n$ to Γ truncated after n $\fatsemi$ separators, provided Γ contains at least n such:

$$\begin{aligned} \Xi \lfloor 0 &= \Xi \\ \Xi \fatsemi \Gamma \lfloor 0 &= \Xi \\ \Xi \fatsemi \Gamma \lfloor n+1 &= \Xi \fatsemi (\Gamma \lfloor n) \\ \Xi \lfloor n+1 &\ \text{undefined} \end{aligned}$$

We write $\delta : \Gamma \preceq \Delta$ if δ is a substitution from Γ to Δ such that, for every $vD \in \Gamma \lfloor n$ and $S \in [\![vD]\!]$, we have that $\Delta \lfloor n$ is defined and $\Delta \vdash \delta S$.

This definition of $\Gamma \preceq \Delta$ is stronger than the previous definition, because it requires a correspondence between the $\fatsemi$-separated sections of Γ and Δ, such that declarations in the first n sections of Γ can be interpreted over the first n sections of Δ. However, it is mostly straightforward to verify that the previous results go through with the new definition.

We refine the $\sqsubseteq$ relation in a corresponding way, and further insist for $\Gamma \sqsubseteq \Delta$ that Γ and Δ have the same *shape* (list of $\fatsemi$ separators and term variables). Formally, we say that $\delta : \Gamma \sqsubseteq \Delta$ if $\delta : \Gamma \preceq \Delta$ and $\lfloor \Gamma \rfloor = \delta \lfloor \Delta \rfloor$, where $\lfloor \cdot \rfloor$ is the forgetful map from contexts to shapes that discards type variables:

$$\begin{aligned} \lfloor \varepsilon \rfloor &= \varepsilon \\ \lfloor \Gamma, \alpha D \rfloor &= \lfloor \Gamma \rfloor \\ \lfloor \Gamma, x \mathbin{\hat{::}} \sigma \rfloor &= \lfloor \Gamma \rfloor, x \mathbin{\hat{::}} \sigma \\ \lfloor \Gamma \fatsemi \rfloor &= \lfloor \Gamma \rfloor \fatsemi \end{aligned}$$

Substitution on shapes acts on type schemes only.

7.2 Fixing the unification algorithm

The only place where changing the $\preceq$ relation requires extra work is in the unification algorithm, because it acts structurally over the context, so we need to specify what happens when it finds a $\fatsemi$ separator. It turns out that these can simply be ignored, so we add the following algorithmic rules:

$$\textsc{Skip}\ \frac{\Gamma_0 \preceq \Delta_0 \vdash \alpha \equiv \beta}{\Gamma_0 \fatsemi \preceq \Delta_0 \fatsemi \vdash \alpha \equiv \beta}$$

$$\textsc{Repossess}\ \frac{\Gamma_0 \mid \Xi \preceq \Delta_0 \vdash \alpha \equiv \tau}{\Gamma_0 \fatsemi \mid \Xi \preceq \Delta_0 \fatsemi \vdash \alpha \equiv \tau}$$

Proving correctness of the SKIP rule is relatively straightforward, thanks to the following lemma.

Lemma 9. *If* $\delta : \Gamma \mathrel{\hat{\preceq}} \Delta \vdash P(a)\langle b \rangle$ *then* $\delta : \Gamma \fatsemi \mathrel{\hat{\preceq}} \Delta \fatsemi \vdash P(a)\langle b \rangle$.

The REPOSSESS rule is more complicated. It is so named because it moves the variable declarations in Ξ to the left of the $\fatsemi$ separator, thereby 'repossessing' them. Despite this, unification does still produce a most general solution:

Lemma 10 (Soundness and generality of REPOSSESS rule)**.** *If* $\Gamma \fatsemi \mid \Xi \preceq \Delta \fatsemi \vdash \alpha \equiv \tau$ *then* $\mathcal{V}_{\mathrm{TY}}(\Gamma \fatsemi \Xi) = \mathcal{V}_{\mathrm{TY}}(\Delta \fatsemi)$ *and* $\iota : \Gamma \fatsemi \Xi \mathrel{\hat{\preceq}} \Delta \fatsemi \vdash \alpha \equiv \tau$.

Proof. Suppose $\Gamma \fatsemi \mid \Xi \preceq \Delta \fatsemi \vdash \alpha \equiv \tau$, so $\Gamma \mid \Xi \preceq \Delta \vdash \alpha \equiv \tau$ as only the REPOSSESS rule applies. By induction and lemma 5, $\mathcal{V}_{\mathrm{TY}}(\Gamma, \Xi) = \mathcal{V}_{\mathrm{TY}}(\Delta)$ and $\iota : \Gamma, \Xi \mathrel{\hat{\preceq}} \Delta \vdash \alpha \equiv \tau$.

For the first part, we have

$$\mathcal{V}_{\mathrm{TY}}(\Gamma \fatsemi \Xi) = \mathcal{V}_{\mathrm{TY}}(\Gamma, \Xi) = \mathcal{V}_{\mathrm{TY}}(\Delta) = \mathcal{V}_{\mathrm{TY}}(\Delta \fatsemi).$$

For the second part, since $\iota : \Gamma, \Xi \preceq \Delta$ we have $\iota : \Gamma \fatsemi \Xi \preceq \Delta \fatsemi$, and $\Delta \vdash \alpha \equiv \tau$ so $\Delta \fatsemi \vdash \alpha \equiv \tau$.

For minimality, suppose $\theta : \Gamma \fatsemi \Xi \preceq \Theta \fatsemi \Phi$ and $\Theta \fatsemi \Phi \vdash \theta\alpha \equiv \theta\tau$. Observe that $\alpha \in \mathcal{V}_{\mathrm{TY}}(\Gamma)$ and $\beta \in \mathcal{V}_{\mathrm{TY}}(\Xi) \Rightarrow \beta \in FTV(\tau, \Xi)$ by the sanity conditions. Now $\theta\alpha$ is a Θ-type and $\theta\tau$ is equal to it, so the only declarations in Φ that $\theta\tau$ (hereditarily) depends on must be definitions over Θ. But all the variables declared in Ξ are used in τ, so there is a substitution $\psi : \Gamma \fatsemi \Xi \preceq \Theta \fatsemi$ that agrees with θ on Γ and maps variables in Ξ to their definitions in Θ. Note that $\psi \equiv \theta : \Gamma \fatsemi \Xi \preceq \Theta \fatsemi \Phi$.

Hence $\psi : \Gamma, \Xi \preceq \Theta$ and $\Theta \vdash \psi\alpha \equiv \psi\tau$, so by hypothesis there exists $\zeta : \Delta \preceq \Theta$ such that $\psi \equiv \zeta \cdot \iota : \Gamma, \Xi \preceq \Theta$. Then $\zeta : \Delta \fatsemi \preceq \Theta \fatsemi \Phi$ and $\psi \equiv \zeta \cdot \iota : \Gamma \fatsemi \Xi \preceq \Theta \fatsemi \Phi$, so $\theta \equiv \zeta \cdot \iota : \Gamma \fatsemi \Xi \preceq \Theta \fatsemi \Phi$. □

Observe that the unification algorithm makes no changes to the shape of the input context, so the corresponding results hold for the more restrictive $\sqsubseteq$ relation.

8. A type inference algorithm

8.1 Transforming the rule system for type assignment

To transform a rule into an algorithmic form, we proceed clockwise starting from the conclusion. For each hypothesis, we must ensure that the problem is fully specified, inserting variables to stand for unknown problem inputs. Moreover, we cannot pattern match on problem outputs, so we ensure there are schematic variables in output positions, fixing things up with appeals to unification.

Consider the rule for application, written to highlight problem inputs and outputs as

$$\frac{f : \langle v \triangleright \tau \rangle \quad a : \langle v \rangle}{fa : \langle \tau \rangle}.$$

Since we cannot match on the output of the first subproblem, we use a metavariable instead and add a unification constraint, giving

$$\frac{f : \langle \chi \rangle \quad a : \langle v \rangle \quad \chi \equiv v \triangleright \tau}{fa : \langle \tau \rangle}.$$

Furthermore, τ is an input to the unification problem, but it is not determined by the previous inputs or outputs, so we have to bind a

fresh variable β instead to give the algorithmic version

$$\frac{f : \langle \chi \rangle \quad a : \langle \upsilon \rangle \quad (\beta := \tau \succ \chi \equiv \upsilon \triangleright \beta)}{(\beta := \tau \succ f\,a : \langle \beta \rangle)}.$$

The rule for abstraction is

$$\frac{(x \hat{::} .\upsilon \succ t : \langle \tau \rangle)}{\lambda x.t : \langle \upsilon \triangleright \tau \rangle}$$

which has unknown input υ, so we bind a fresh variable β to give

$$\frac{(\beta := \upsilon \succ (x \hat{::} .\beta \succ t : \langle \tau \rangle))}{(\beta := \upsilon \succ \lambda x.t : \langle \beta \triangleright \tau \rangle)}.$$

The let rule is

$$\frac{s :: \sigma \quad (x \hat{::} \sigma \succ w : \tau)}{\text{let } x := s \text{ in } w : \langle \tau \rangle}.$$

Writing $\sigma = (\Xi \Uparrow .\upsilon)$ and expanding the definition of ::, we obtain

$$\frac{(\Xi \succ s : \upsilon) \quad (x \hat{::} (\Xi \Uparrow .\upsilon) \succ w : \tau)}{\text{let } x := s \text{ in } w : \langle \tau \rangle}.$$

where we let $(\varepsilon \succ S) = S$ and $((\Xi, \upsilon D) \succ S) = (\Xi \succ (\upsilon D \succ S))$.

But how can we find Ξ? This is where we use the $\fatsemi$ separator. Instead of an unknown list of type variables, we just add a $\fatsemi$ to the context, infer the type of s, then generalise its type by 'skimming off' type variables from the top of the context until the $\fatsemi$ is reached.

We define the type inference assertion $\Gamma \sqsubseteq \Delta \vdash t : \langle \tau \rangle$ by the rules in Figure 6. These rules are clearly structural on terms, so yield a terminating algorithm, leading naturally to an implementation, given in subsection 8.3.

We use Lemma 8 to ensure in rule VAR that we compute a suffix Ξ consisting of fresh names, such that the output Γ, Ξ is well-formed.

$$\boxed{\Gamma \sqsubseteq \Delta \vdash t : \langle \tau \rangle}$$

$$\textsc{Var}\ \frac{x \hat{::} (\Xi \Uparrow .\upsilon) \in \Gamma}{\Gamma \sqsubseteq \Gamma, \Xi \vdash x : \langle \upsilon \rangle}$$

$$\textsc{Abs}\ \frac{\Gamma, \alpha := ?, x \hat{::} .\alpha \sqsubseteq \Delta_0, x \hat{::} .\alpha, \Xi \vdash w : \langle \upsilon \rangle}{\Gamma \sqsubseteq \Delta_0, \Xi \vdash \lambda x.w : \langle \alpha \triangleright \upsilon \rangle}\ \alpha \notin \mathcal{V}_{\text{TY}}(\Gamma)$$

$$\textsc{App}\ \frac{\Gamma \sqsubseteq \Delta_0 \vdash f : \langle \chi \rangle \quad \Delta_0 \sqsubseteq \Delta_1 \vdash a : \langle \upsilon \rangle \quad \Delta_1, \beta := ? \preceq \Delta \vdash \chi \equiv \upsilon \triangleright \beta}{\Gamma \sqsubseteq \Delta \vdash f\,a : \langle \beta \rangle}\ \beta \notin \mathcal{V}_{\text{TY}}(\Delta_1)$$

$$\textsc{Let}\ \frac{\Gamma \fatsemi \sqsubseteq \Delta_0 \fatsemi \Xi_0 \vdash s : \langle \upsilon \rangle \quad \Delta_0, x \hat{::} (\Xi_0 \Uparrow .\upsilon) \sqsubseteq \Delta_1, x \hat{::} (\Xi_0 \Uparrow .\upsilon), \Xi_1 \vdash w : \langle \chi \rangle}{\Gamma \sqsubseteq \Delta_1, \Xi_1 \vdash \text{let } x := s \text{ in } w : \langle \chi \rangle}$$

Figure 6. Algorithmic rules for type inference

8.2 Soundness and completeness

Lemma 11 (Soundness of type inference)**.** *If* $\Gamma \sqsubseteq \Delta \vdash t : \langle \upsilon \rangle$ *then* $\iota : \Gamma \sqsubseteq \Delta$ *and* $\Delta \vdash t : \langle \upsilon \rangle$.

Proof. By induction on the structure of derivations. □

Lemma 12 (Completeness and generality of type inference)**.** *If* $\theta : \Gamma \sqsubseteq \Theta$ *and* $\Theta \vdash t : \tau$ *then* $\Gamma \hat{\sqsubseteq} \Delta \vdash t : \langle \upsilon \rangle$ *for some type* υ *and context* Δ.

Sketch. The algorithm is structurally recursive over terms, failing only when unification fails. Each step locally preserves all possible solutions. For let-expressions, observe that any type specialising any scheme for s must certainly specialise the type we infer for s, and *ipso facto*, the principal type scheme we assign to x. For details, see appendix. □

8.3 Implementing type inference

The *infer* function attempts to infer the type of the given term, updating the context with the minimum necessary information.

```
infer :: Term → Contextual Type
```

To infer the type of a variable, we look up its type scheme in the context, and specialise this scheme with fresh variables.

```
infer (X x) = getContext >>= find >>= specialise
  where
    find :: Context → Contextual Scheme
    find (Γ :< TM (y ::̂ σ))
       | x ≡ y      = return σ
    find (Γ :< _)   = find Γ
    find ε          = fail "Missing variable"
```

To infer the type of a λ-abstraction, we recursively infer the type of its body w with variable x assigned type-scheme $.\alpha$, with α fresh.

```
infer (λx.w) = do
  α ← fresh ?
  υ ← x ::̂ Type (V α) ≻ infer w
  return (V α ▷ υ)
```

To infer the type of an application, we infer the type χ of the function f, then the type υ of the argument. Unifying χ with $\upsilon \triangleright \beta$, where β is a fresh variable, produces the result.

```
infer (f :$ a) = do
  χ ← infer f
  υ ← infer a
  β ← fresh ?
  unify χ (υ ▷ V β)
  return (V β)
```

Finally, to infer the type of a let construct, we infer the type of the definiens s and generalise it to yield a scheme σ. We then infer the type of the body w in the context where $x \hat{::} \sigma$.

```
infer (let x := s in w) = do
  σ ← generaliseOver (infer s)
  x ::̂ σ ≻ infer w
```

The *generaliseOver* operator adds a $\fatsemi$ to the context, evaluates its argument, then generalises over type variables to the right of the $\fatsemi$ marker.

```
generaliseOver :: Contextual Type → Contextual Scheme
generaliseOver mt = do
    modifyContext (:<⨾)
    τ ← mt
    Ξ ← skimContext
    return (Ξ ⇑ Type τ)
  where
    skimContext :: Contextual (Bwd TyEntry)
```

```
skimContext = do
  Γ :< vD ← getContext
  putContext Γ
  case vD of
    ∘          → return ε
    TY αD → (:<αD) <$> skimContext
    TM _   → error "Unexpected TM variable!"
```

The $(\succ)$ operator appends a term variable declaration to the context, evaluates its second argument, then removes the declaration.

```
(≻) :: TmEntry → Contextual a → Contextual a
x ∷̂ σ ≻ ma = do
    modifyContext (:< TM (x ∷̂ σ))
    a ← ma
    modifyContext extract
    return a
  where
    extract :: Context → Context
    extract (Γ :< TM (y ∷̂ _))
      | x ≡ y              = Γ
    extract (Γ :< TY xD) = (extract Γ) :< TY xD
    extract (Γ :< _) = error "Bad context entry!"
    extract ε        = error "Missing TM variable!"
```

9. Discussion

We have arrived at an implementation of Hindley-Milner type inference which involves all the same steps as Algorithm $\mathcal{W}$, but not necessarily in the same order. In particular, the dependency analysis which $\mathcal{W}$ performs all of a sudden in the let-rule is here pushed down to a requirement that the underlying unification algorithm maintain the well-foundedness of the context.

Our algorithm is presented as a system of problem transformation locally preserving all possibile solutions, hence finding a most general global solution if any at all. Accumulating solutions to decomposed problems is justified simply by the stability of solutions with respect to information increase. We have established a discipline of problem solving which happens to be complete for Hindley-Milner type inference but in any case maintains a coupling of soundness with generality.

Maintain context validity, make definitions anywhere and only where there is no choice, so the solutions you find will be general and generalisable locally: this is a key design principle for elaboration of high-level code in systems like Epigram and Agda; bugs arise from its transgression. By giving a disciplined account of 'current information' in terms of contexts and their information ordering, we provide a means to investigate these problems and justify the steps we take to repair them.

We are, however, missing yet more context. Our task was greatly simplified by studying a structural type inference process for 'finished' expressions in a setting where unification is complete. Each subproblem is either solved or rejected on first inspection—there is never a need for a 'later, perhaps' outcome. As a result, the conventional control discipline of 'direct style' recursive programming is adequate to the task. If problems could get stuck, how might we abandon them and return to them later? By storing their *context*, of course!

Here, we have combined the linguistic contexts for the various sorts of variable involved in problems; our next acquisition is the syntactic context of the target term, interspersing variable declarations with components of its *zipper* [Huet 1997]. We thus become free to abandon fixed recursion strategies and refocus wherever progress is to be made. The tree-like proof states of McBride's thesis evolved into exactly such 'zippers with binding' in the implementation of Epigram.

As we have seen, an information increase is nothing other than the extension of a simultaneous substitution from variables and terms to declarations and derivations. Our generic analysis of the role of declarations in derivations shows that stability is endemic, amounting to the action of hereditary substitution on 'cut-free' derivations. And that is exactly what it should be. We have rationalised Hindley-Milner type inference by adapting a discipline for interactively constructing inhabitants of dependent types as the means to manage unknowns when incrementally constructing solutions to problems. The analysis can only become clearer, the technology simpler, as we bring these two kinds of construction together, mediating *problems as types*.

References

Françoise Bellegarde and James Hook. Substitution: a formal methods case study using monads and transformations. *Science of Computer Programming*, 23(2-3):287–311, 1994.

Richard Bird and Ross Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '82)*, pages 207–212, Albuquerque, New Mexico, USA, 1982. ACM.

GHC Team. The Glorious Glasgow Haskell Compilation System user's guide, version 6.12.1. Section 7.5. Extensions to the "deriving" mechanism, 2009. URL `http://www.haskell.org/ghc/docs/latest/html/users_guide/deriving.html#deriving-typeable`.

Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

Bruce J. McAdam. On the unification of substitutions in type inference. In *Implementation of Functional Languages (IFL' 98)*, pages 139–154. Springer, 1998.

Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.

Conor McBride. First-Order Unification by Structural Recursion. *Journal of Functional Programming*, 13(6), 2003.

Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, January 2004.

Conor McBride and James McKinna. Functional pearl: I am not a number–I am a free variable. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 1–9, Snowbird, Utah, USA, 2004. ACM.

Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23 (3):299–318, November 1999.

Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Peter Norvig. Correcting a widespread error in unification algorithms. *Software: Practice and Experience*, 21(2):231–233, 1991.

Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In *Proceedings of the 4th international*

conference on Typed Lambda Calculi and Applications (TLCA '99), pages 295–309, 1999.

Robert Pollack. Implicit syntax. In *Informal Proceedings of First Workshop on Logical Frameworks*, May 1990.

J. Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

J. B. Wells. The essence of principal typings. In *ICALP '02: Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 913–925, London, UK, 2002. Springer.

A. Appendix

Proof of lemma 4 (Optimist's Lemma). We have that $\theta \cdot \delta : \Gamma \preceq \Theta$ by Lemma 1.

To show $\Theta \vdash P \wedge Q(a,b)\langle \theta r, s\rangle$, it suffices to show $\Theta \vdash P(a)\langle \theta r\rangle$ and $\Theta \vdash Q(b)\langle s\rangle$. The latter holds by assumption. For the former, note that $\Delta \vdash P(a)\langle r\rangle$ and hence $\Theta \vdash \theta(P(a)\langle r\rangle)$ by stability of $P(a)\langle r\rangle$. But $\theta(P(a)\langle r\rangle) = P(a)\langle \theta r\rangle$ by definition.

Finally, suppose there is some $\theta : \Gamma \preceq \Theta$ and outputs t, u such that $\Theta \vdash P \wedge Q(a,b)\langle t, u\rangle$, so $\Theta \vdash P(a)\langle t\rangle$ and $\Theta \vdash Q(b)\langle u\rangle$. Since $\delta : \Gamma \hat{\preceq} \Delta \vdash P(a)\langle r\rangle$, there exists $\zeta_1 : \Delta \preceq \Theta$ such that $\theta \equiv \zeta_1 \cdot \delta$ and $\Theta \vdash R_P\langle \zeta_1 r\rangle\langle t\rangle$. But then $\theta : \Delta \hat{\preceq} \Theta \vdash Q(b)\langle s\rangle$, so there exists $\zeta_2 : \Theta \preceq \Theta$ such that $\zeta_1 \equiv \zeta_2 \cdot \theta$ and $\Theta \vdash R_Q\langle \zeta_2 s\rangle\langle u\rangle$. Hence $\theta \equiv \zeta_2 \cdot (\theta \cdot \delta)$ and $\Theta \vdash R_{P\wedge Q}\langle \zeta_2(\theta r), \zeta_2 s\rangle\langle t, u\rangle$. □

Proof of lemma 7 (Completeness and generality of unification). It suffices to show that the algorithm succeeds and produces a solution that is below Θ. We proceed by induction on the call graph; since the algorithm terminates, this is well-founded.

(a) (i) Suppose $\upsilon = \alpha$ and $\tau = \beta$ are variables. Let $\Gamma = \Gamma_0, \upsilon D$ and examine υD:

- If $\upsilon = \alpha = \beta$ are all the same variable, then the IDLE rule applies, $\Delta = \Gamma$ and the result is trivial.
- If $\upsilon D = \alpha :=?$ then the DEFINE rule applies, $\Delta = \Gamma_0, \alpha := \beta$ and $\theta : \Delta \preceq \Theta$. The case $\upsilon D = \beta :=?$ is similar.
- If $\upsilon D = \alpha := \chi$ then $\Theta \vdash \theta\alpha \equiv \theta\chi$ by definition of $\preceq$, and $\Theta \vdash \theta\alpha \equiv \theta\beta$ by hypothesis, so $\Theta \vdash \theta\beta \equiv \theta\chi$ by transitivity and symmetry. But then $\theta_\alpha : \Gamma_0 \preceq \Theta$ where θ_α is θ with the type assigned to α removed, and $\Theta \vdash \theta_\alpha\beta \equiv \theta_\alpha\chi$, since β and χ cannot depend on α by the sanity conditions. Now, by induction, $\Gamma_0 \preceq \Delta_0 \vdash \beta \equiv \chi$ for some Δ_0. with $\theta_\alpha : \Delta_0 \preceq \Theta$. Hence the EXPAND rule applies, $\Delta = \Delta_0, \alpha := \upsilon$ and $\theta : \Delta \preceq \Theta$. The case $\upsilon D = \beta := \upsilon$ is similar.
- Otherwise, $\upsilon D \perp \{\alpha, \beta\}$ and the IGNORE rule applies by a similar argument.

(ii) Now suppose wlog that $\upsilon = \alpha$ is a variable and τ is not a variable. The conditions for part (b) hold, so by induction, $\Gamma \mid \varepsilon \preceq \Delta \vdash \alpha \equiv \tau$ and the SOLVE rule applies.

(iii) Otherwise, we have $\tau = \tau_0 \triangleright \tau_1$ and $\upsilon = \upsilon_0 \triangleright \upsilon_1$. Then $\Theta \vdash \theta\tau_0 \equiv \theta\upsilon_0$ and $\Theta \vdash \theta\tau_1 \equiv \theta\upsilon_1$, so by induction there exist contexts Δ_0, Δ such that $\Gamma \preceq \Delta_0 \vdash \tau_0 \equiv \upsilon_0$ and $\Delta_0 \preceq \Delta \vdash \tau_1 \equiv \upsilon_1$. Hence the DECOMPOSE rule applies, and $\theta : \Delta \preceq \Theta$ by the Optimist's lemma.

(b) Let $\Gamma = \Gamma_0, \upsilon D$.

(i) If $\upsilon = \alpha \in FTV(\tau, \Xi)$, then there is some non-variable type χ such that $\Theta \vdash \theta\alpha \equiv \theta\chi$ and $\alpha \in FTV(\chi)$. But this cannot occur, by lemma 6.

(ii) If $\upsilon D = \alpha :=?$ and $\alpha \notin FTV(\tau, \Xi)$, then the DEFINES rule applies, $\Delta = \Gamma_0, \Xi, \alpha := \tau$ and $\theta : \Delta \preceq \Theta$.

(iii) If $\upsilon D = \alpha := \chi$ and $\alpha \notin FTV(\tau, \Xi)$, then $\Theta \vdash \theta\alpha \equiv \theta\chi$, so $\Theta \vdash \theta\chi \equiv \theta\tau$ by symmetry and transitivity. Moreover, $\Gamma_0, \Xi \vdash \tau$ **type** since $\alpha \notin FTV(\tau, \Xi)$, and $\theta_\alpha : \Gamma_0, \Xi \preceq \Theta$ so by induction $\Gamma_0, \Xi \preceq \Delta_0 \vdash \chi \equiv \tau$ for some Δ_0 with $\theta_\alpha : \Delta_0 \preceq \Theta$. Hence the EXPANDS rule applies with $\Delta = \Delta_0, \alpha := \chi$ and $\theta : \Delta \preceq \Theta$.

(iv) If $\upsilon = \beta$ for $\alpha \neq \beta$ and $\beta \in FTV(\upsilon, \Xi)$ then $\Gamma_0 \mid \beta D, \Xi \preceq \Delta \vdash \alpha \equiv \tau$ is well-posed, so it has a solution by induction and the DEPENDS rule applies.

(v) Otherwise $\upsilon D \perp FTV(\alpha, \tau, \Xi)$ and $\Gamma_0 \mid \Xi \preceq \Delta_0 \vdash \alpha \equiv \tau$ is well-posed, so it has a solution by induction and the IGNORES rule applies with $\Delta = \Delta_0, \upsilon D$. □

Proof of lemma 12 (Completeness and generality of type inference). It suffices to show that the algorithm succeeds and delivers a result that is below (τ, θ, Θ). We proceed by structural induction.

Variables. If $t = x$ is a variable, then by inversion $x \hat{::} \sigma \in \Theta$. Now by definition of $\sqsubseteq$, $x \hat{::} (\Xi \Uparrow \upsilon) \in \Gamma$ where $\sigma = \theta(\Xi \Uparrow \upsilon)$. Hence the VAR rule applies giving $\Gamma \sqsubseteq \Gamma, \Xi \vdash x : \langle \upsilon\rangle$.

The proof of $\Theta \vdash x : \tau$ must consist of applying $\Theta \Vdash x :: (\theta\Xi \Uparrow \theta\upsilon)$ to some Θ-types, so it determines a map from the unbound type variables of Ξ to types over Θ, and hence a substitution $\zeta : \Gamma, \Xi \sqsubseteq \Theta$ that agrees with θ on Γ and maps type variables in Ξ to their definitions in Θ. Thus $\theta \equiv \zeta \cdot \iota : \Gamma \sqsubseteq \Theta$.

Let bindings. If $t = (\text{let } x := s \text{ in } w)$, then by inversion there is some scheme $\sigma = (\Psi \Uparrow .\tau_s)$ such that $\Theta \vdash s :: \sigma$ and $\Theta, x \hat{::} \sigma \vdash w : \tau$. Then $\Theta ⨟ \vdash s :: (\Psi \Uparrow .\tau_s)$ so $\Theta ⨟ \Psi \vdash s : \tau_s$.

Moreover $\theta : \Gamma ⨟ \preceq \Theta ⨟ \Psi$, so by induction $\Gamma ⨟ \sqsubseteq \Delta_0 ⨟ \Xi_0 \vdash s : \langle \upsilon\rangle$ and there exists $\zeta_0 : \Delta_0 ⨟ \Xi_0 \sqsubseteq \Theta ⨟ \Psi$ such that $\theta \equiv \zeta_0 \cdot \iota$ and $\Theta ⨟ \Psi \vdash \zeta_0\upsilon \equiv \tau_s$.

Now $\zeta_0|_{\Delta_0} : \Delta_0 \sqsubseteq \Theta$, so

$$\zeta_0|_{\Delta_0} : \Delta_0, x \hat{::} (\Xi_0 \Uparrow .\upsilon) \sqsubseteq \Theta, x \hat{::} \zeta_0(\Xi_0 \Uparrow .\upsilon).$$

and (note the $\preceq$ relation since we are generalising the type scheme)

$$\iota : \Theta, x \hat{::} (\Psi \Uparrow .\tau_s) \preceq \Theta, x \hat{::} \zeta_0(\Xi_0 \Uparrow .\upsilon)$$

so by stability of type assignment under the $\preceq$ relation,

$$\Theta, x \hat{::} \zeta_0(\Xi_0 \Uparrow .\upsilon) \vdash w : \tau.$$

Hence, by induction,

$$\Delta_0, x \hat{::} (\Xi_0 \Uparrow .\upsilon) \sqsubseteq \Delta_1, x \hat{::} (\Xi_0 \Uparrow .\upsilon), \Xi_1 \vdash w : \langle \chi\rangle$$

and there is some

$$\zeta_1 : \Delta_1, x \hat{::} (\Xi_0 \Uparrow .\upsilon), \Xi_1 \sqsubseteq \Theta, x \hat{::} \zeta_0(\Xi_0 \Uparrow .\upsilon)$$

such that $\zeta_0|_{\Delta_0} \equiv \zeta_1 \cdot \iota$ and $\Theta, x \hat{::} \zeta_0(\Xi_0 \Uparrow .\upsilon) \vdash \zeta_1\chi \equiv \tau$.

Now the LET rule applies to give

$$\Gamma \sqsubseteq \Delta_1, \Xi_1 \vdash \text{let } x := s \text{ in } w : \langle \chi\rangle$$

and we have $\zeta_1 : \Delta_1, \Xi_1 \sqsubseteq \Theta$, $\Theta \vdash \zeta_1\chi \equiv \tau$ and $\theta \equiv \zeta_1 \cdot \iota$.

λ-abstractions. If $t = \lambda x.w$ is an abstraction, then by inversion $\Theta \vdash \tau \equiv \tau_0 \triangleright \tau_1$ for some types τ_0 and τ_1, and $\Theta, x \hat{::} .\tau_0 \vdash w : \tau_1$. Taking $\theta' = [\tau_0/\alpha]\theta$, we have that

$$\theta' : \Gamma, \alpha{:=}?, x \hat{::} .\alpha \sqsubseteq \Theta, x \hat{::} .\tau_0$$

and hence, by induction,

$$\Gamma, \alpha{:=}?, x \hat{::} .\alpha \sqsubseteq \Delta_0, x \hat{::} .\alpha, \Xi \vdash w : \langle \upsilon \rangle$$

with $\zeta : \Delta_0, x \hat{::} .\alpha, \Xi \sqsubseteq \Theta, x \hat{::} .\tau_0$ such that $\theta' \equiv \zeta \cdot \iota$ and $\Theta, x \hat{::} .\tau_0 \vdash \zeta\upsilon \equiv \tau_1$.

Thus the ABS rule applies, so we have

$$\Gamma \sqsubseteq \Delta_0, \Xi \vdash \lambda x.w : \langle \alpha \triangleright \upsilon \rangle,$$

$\zeta : \Delta_0, \Xi \sqsubseteq \Theta, \theta \equiv \zeta \cdot \iota$ and $\Theta \vdash \zeta(\alpha \triangleright \upsilon) \equiv \tau_0 \triangleright \tau_1$.

Applications. If $t = fa$ is an application, then $\Theta \vdash f : \tau_0 \triangleright \tau$, so by induction $\Gamma \sqsubseteq \Delta_0 \vdash f : \langle \chi \rangle$ and there exists $\zeta_0 : \Delta_0 \sqsubseteq \Theta$ such that $\theta \equiv \zeta_0 \cdot \iota$ and $\Theta \vdash \zeta_0\chi \equiv \tau_0 \triangleright \tau$.

Now $\Theta \vdash a : \tau_0$, so by induction $\Delta_0 \sqsubseteq \Delta_1 \vdash a : \langle \upsilon \rangle$ and there exists $\zeta_1 : \Delta_1 \sqsubseteq \Theta$ such that $\zeta_0 \equiv \zeta_1 \cdot \iota$ and $\Theta \vdash \zeta_1\upsilon \equiv \tau_0$.

Let $\zeta_2 = [\tau/\beta]\zeta_1$, then $\zeta_2 : \Delta_1, \beta{:=}? \sqsubseteq \Theta$. Now $\Theta \vdash \zeta_2\chi \equiv \tau_0 \triangleright \tau$ since χ is a Δ_0 type so $\Theta \vdash \zeta_2\chi \equiv \zeta_0\chi$. Similarly, and since ζ_2 maps β to τ, we have $\Theta \vdash \zeta_2(\upsilon \triangleright \beta) \equiv \tau_0 \triangleright \tau$. Hence $\Theta \vdash \zeta_2\chi \equiv \zeta_2(\upsilon \triangleright \beta)$ so we have $\Delta_1, \beta{:=}? \sqsubseteq \Delta \vdash \chi \equiv \upsilon \triangleright \beta$ with $\zeta : \Delta \sqsubseteq \Theta$ such that $\zeta_2 \equiv \zeta \cdot \iota$ by completeness of unification. Hence the APP rule applies, so $\Gamma \sqsubseteq \Delta \vdash fa : \langle \beta \rangle$, and $\theta \equiv \zeta \cdot \iota$.

□