Nominal Verification of Algorithm W

Christian Urban and Tobias Nipkow

March 18, 2008

Abstract

The Milner-Damas typing algorithm *W* is one of the classic algorithms in Computer Science. In this paper we describe a formalised soundness and completeness proof for this algorithm. Our formalisation is based on names for both term and type variables, and is carried out in Isabelle/HOL using the Nominal Datatype Package. It turns out that in our formalisation we have to deal with a number of issues that are often overlooked in informal presentations of *W*.

> *"Alpha-conversion always bites you when you least expect it."*
>
> A remark made by Xavier Leroy when discussing with us the informal proof about *W* in his PhD-thesis [7].

1 Introduction

Milner's polymorphic type system for ML [8] is probably the most influential programming language type system. The second author learned about it from a paper by Clément, Despeyroux, Despeyroux, and Kahn [2]. He was immediately taken by their view that type inference can be viewed as Prolog execution, in particular because the Isabelle system, which he had started to work on, was based on a similar paradigm as the Typol language developed by Kahn and his coworkers [1]. Milner himself had provided the explicit type inference algorithm *W* and proved its correctness. Completeness was later shown by Damas and Milner [4]. Neither soundness nor completeness of *W* are trivial because of the presence of the Let-construct (which is not expanded during type inference). Two machine-checked proofs for soundness and completeness were implemented previously [5, 9]. Both of these proofs code type and term variables using de Bruijn indices. This leads to slick proofs which however require strange lemmas about arithmetic on de Bruijn terms, not present in typical proofs done with "pencil-and-paper" (for example [7, 13]).

Here we will describe a formalisation for soundness and completeness of *W* using the Nominal Datatype Package developed by Berghofer and the first author [14, 16]. This package is based on ideas pioneered by Pitts et al [11, 12] and aims to provide all necessary infrastructure for reasoning conveniently about languages with bound variables. For this it provides mechanisms to deal with named binders and allows one

to define datatypes modulo α-equivalence. For example, when defining the lambda-calculus with the terms

$$a \quad | \quad t_1\ t_2 \quad | \quad \lambda a.t$$

one defines term-constructors for variables, applications and lambdas and indicates in case of lambdas that a is bound in t. The Nominal Datatype Package constructs then a (nominal) datatype representing α-equivalence classes of those terms. Unlike constructions involving de Bruijn indices, however, the α-equivalence classes in the Nominal Datatype Package involve names. This is similar to the convention in "pencil-and-paper" proofs where one states that one identifies terms that differ only in the names of bound variables, but works with terms in a "naïve" way. However, dealing with α-equivalence classes has some subtle consequences: some functions cannot be defined anymore and α-equivalence classes do not immediately come equipped with a structural induction principle. Therefore the Nominal Datatype Package provides a recursion combinator that ensures functions respect α-equivalence classes and also provides two principles for performing proofs by structural induction over them. The first induction principle looks as follows:

$$\frac{\begin{array}{l}\forall a.\ P\ (a)\\ \forall t_1\ t_2.\ P\ t_1\ \wedge P\ t_2 \longrightarrow P\ (t_1\ t_2)\\ \forall a\ t.\ P\ t \longrightarrow P\ (\lambda a.t)\end{array}}{P\ t}$$

where a property P holds for all (α-equated) lambda-terms t provided the property holds for variables, applications and lambdas. In the latter two cases one can as usual assume the property holds for the immediate subterms. However this principle is quite inconvenient in practice, since it requires to prove the lambda-case for all binders, which often means one has to rename binders and establish auxiliary lemmas concerning such renamings. In informal reasoning this renaming is nearly always avoided by employing the variable convention for bound variables. Therefore the Nominal Datatype Package generates automatically the following stronger induction principle for α-equated lambda-terms

$$\frac{\begin{array}{l}\forall a\ C.\ P\ C\ (a)\\ \forall t_1\ t_2\ C.\ (\forall C.\ P\ C\ t_1) \wedge (\forall C.\ P\ C\ t_2) \longrightarrow P\ C\ (t_1\ t_2)\\ \forall a\ t\ C.\ a \#\ C \wedge (\forall C.\ P\ C\ t) \longrightarrow P\ C\ (\lambda a.t)\end{array}}{P\ C\ t}$$

where one only needs to prove the lambda-case for all fresh binders (w.r.t. some suitably chosen context C). With the stronger induction principle we can relatively easily formalise informal proofs employing the variable convention (for more details see [15, 16]). The reason is that the variable convention usually states which free variables the binder has to avoid. We can achieve the same with the stronger induction principle by instantiation C with what is informally avoided.

2 Terms, Types and Substitutions

2.1 Terms and Types

Our *terms* represent λ-terms enriched with Let-expressions:

$$trm = Var\ var \mid App\ trm\ trm \mid Lam\ var.\ trm \mid Let\ var\ be\ trm\ in\ trm$$

where *var* is some infinite type of *term variables*. This definition looks like an ordinary recursive datatype, but its definition in Isabelle includes the keyword **nominal_datatype**. This means that *trm* is really a type of equivalence classes of terms modulo α-conversion. The definition also includes the information that the term-variable *var* in *Lam var. trm* binds all free occurrences *var* in *trm*, and similarly for *Let var be trm in trm'* that all occurrences of *var* are bound in *trm'*. Thus we really work with α-equivalence classes, as we have for example the equation $Lam\ a.\ Var\ a = Lam\ b.\ Var\ b$, which does not hold had we defined terms as an ordinary datatype.

However, *types* do not contain any binders and therefore are defined as an ordinary datatype *ty*

$$ty = TVar\ tvar \mid ty \rightarrow ty$$

based on some infinite type *tvar* of *type variables*.

Type schemes are universally quantified types. This quantification is again modelled via a nominal datatype, namely

$$tyS = Ty\ ty \mid \forall\, tvar.tyS$$

where in the latter clause a type variable is bound in a type scheme. With this definition we fix the order of the binders, and also allow type schemes with multiple occurrences of the same bound type variable, for example $\forall X.\forall X.Ty\ (TVar\ X)$. This will require some care in the proofs we shall give later on. Ideally one would like to quantify over a whole set of variables in one go, as in $\forall \{X_1 \ldots X_n\}.\ ty$, however this is not yet supported in either the nominal or any other approach to datatypes with binders. We are not the first to chose the representation using a fixed order for binders: it has been used in the description of *W* given by Gunter [6] and also by Damas in parts of his thesis (see [3, Page 66]).

Our naming conventions for term variables, terms, type variables, types and types-schemes are

$$a : var, \quad t : trm, \quad X : tvar, \quad T : ty, \quad S : tyS.$$

We use the following list notation: $x::xs$ is the list with head x and tail xs, $xs\ @\ ys$ is the concatenation of two lists, and $x \in xs$ means that x occurs in the list xs. List inclusion is defined by

$$xs \subseteq ys \stackrel{\text{def}}{=} \forall x.\ x \in xs \longrightarrow x \in ys$$

and two lists of type variables are considered *equivalent* provided

$$xs \approx ys \stackrel{\text{def}}{=} xs \subseteq ys \wedge ys \subseteq xs.$$

2.2 Substitutions

We model substitutions as lists, namely $Subst = (tvar \times ty)\ list$, and reserve variables θ, σ and δ for them. Because lists are finite, one can always find a new type variable that does not occur in a substitution. We will use for this concept the notion of freshness, written $X \# \_$, from the nominal logic work [11, 16]. When modelling substitutions as functions, one has to require finiteness of their domain (the type variables not mapped to themselves) explicitly, which complicates matters. Since their is no free lunch, we have to define a number of concepts that would come for free with substitutions as functions.

- Application to a type
  $\theta(TVar\ X) = lookup\ \theta\ X$
  $\theta(T_1 \rightarrow T_2) = \theta(T_1) \rightarrow \theta(T_2)$
  is defined in terms of the auxiliary function *lookup*:
  $lookup\ []\ X = TVar\ X$
  $lookup\ ((Y, T)::\theta)\ X = (if\ X = Y\ then\ T\ else\ lookup\ \theta\ X)$

- Application to a type scheme:
  $\theta(Ty\ T) = Ty\ \theta(T)$
  $\theta(\forall X.S) = \forall X.\theta(S)$ provided $X \# \theta$

- Substitution composition:
  $\theta_1 \circ [] = \theta_1$
  $\theta_1 \circ ((X, T)::\theta_2) = (X, \theta_1(T))::\theta_1 \circ \theta_2$

- Extensional equivalence:
  $\theta_1 \approx \theta_2 \stackrel{\text{def}}{=} \forall X.\ \theta_1(TVar\ X) = \theta_2(TVar\ X)$

- Domain of a substitution:
  $dom\ [] = []$
  $dom\ ((X, T)::\theta) = X::dom\ \theta$

The only technically interesting point here is the application of a substitution to a type scheme. For a start, this definition is not by ordinary structural recursion since it operates on equivalence classes. Luckily the nominal datatype infrastructure provides a mechanism whereby one can specify the recursion equations as above and Isabelle generates verification conditions that imply that the definition is independent of the choice of representatives of the equivalence class. This is the case if X does not occur freely in θ. Note, however, that substitution application over type schemes is *not* a partial function, since type schemes are α-equivalence classes and one can always rename the X away from θ. We can easily show that substitution composition is associative, that is $\theta_1 \circ (\theta_2 \circ \theta_3) = (\theta_1 \circ \theta_2) \circ \theta_3$, and that $\theta_1 \circ \theta_2(\_) = \theta_1(\theta_2(\_))$ holds for types and type schemes. The substitution of a single type variable is defined as a special case:

$$(\_)[X := T] \stackrel{\text{def}}{=} [(X, T)](\_)$$

2.3 Free Type Variables

Free type variables, *ftv*, of types and type-schemes are defined as usual

$$\begin{array}{ll} ftv\ (TVar\ X) = [X] & ftv\ (Ty\ T) = ftv\ T \\ ftv\ (T_1 \rightarrow T_2) = ftv\ T_1\ @\ ftv\ T_2 & ftv\ (\forall X.S) = ftv\ S - [X] \end{array}$$

except that *ftv* returns a list, which may contain duplicates (in the last clause the difference stands for removing all elements of the second list from the first). The reason for lists rather than sets is the following: The typing of Let-expressions (see §3) requires to turn a type into a type scheme by quantifying over some free variables (see §2.4). If the free variables are given as a list, this is just recursion over the list. If they are given as a finite set, one faces the problem that recursion over a set is only well-defined if the order of elements does not matter [10]. But the order of quantifiers does matter in our representation of type schemes! Hence one would need to order the set artificially, for example via HOL's choice operator, which we prefer to avoid.

We shall also make use of the notion of free type variables for pairs and lists, defined by the clauses

$$ftv\ (x, y) = ftv\ x\ @\ ftv\ y \qquad ftv\ [] = [] \qquad ftv\ (x::xs) = ftv\ x\ @\ ftv\ xs.$$

For term and type variables we define $ftv\ a \stackrel{\text{def}}{=} []$ and $ftv\ X \stackrel{\text{def}}{=} [X]$. The free type variables for substitutions are therefore the free type variables in their domain and co-domain.

2.4 Generalisation of Types and Unbinding of Type Schemes

Types can be turned into type schemes by generalising over a list of type variables. This is formalised by the function *gen* defined by

$$\begin{array}{l} gen\ T\ [] = Ty\ T \\ gen\ T\ (X::Xs) = \forall X.gen\ T\ Xs \end{array}$$

In the definitions and proofs that follow, we will also need an unbinding operation that for a type scheme, say S, provides a type T and a list of type variables Xs such that $S = gen\ T\ Xs$. Since type schemes are α-equated, we cannot expect this operation being a function from type schemes to lists of type variables and types: the reason is that such a function calculates with binders in a way that does not preserve α-equivalence classes. Indeed, the Nominal Datatype Package does not allow us to define

$$\begin{array}{l} unbind\ (Ty\ T) = [] \cdot T \\ unbind\ (\forall X.S) = X::(unbind\ S) \end{array}$$

as it would lead to an inconsistency (because $\forall X.Ty\ (TVar\ X)$ can unbind to both $[X] \cdot (TVar\ X)$ and $[Y] \cdot (TVar\ Y)$). However, we can define the unbinding operation of a type scheme as a three-place relation inductively defined by the rules

$$\frac{}{Ty\ T \hookrightarrow [] \cdot T} \qquad \frac{S \hookrightarrow Xs \cdot T}{\forall X.S \hookrightarrow (X::Xs) \cdot T}$$

One can easily establish the following three properties for the unbinding relations.

Lemma 1.
(i) $\exists\, Xs\ T.\ S \hookrightarrow Xs \cdot T$
(ii) $gen\ T\ Xs \hookrightarrow Xs \cdot T$
(iii) $If\ S \hookrightarrow Xs \cdot T\ then\ S = gen\ T\ Xs.$

Proof. The first is by induction on the type scheme S, the second is by induction on Xs and last is by induction over $S \hookrightarrow Xs \cdot T$. □

The property from Lemma 1(i) demonstrates that the unbinding relation is "total" if the last two parameters are viewed as results.

2.5 Instances of a Type Scheme

Types can be obtained as instances of type schemes by instantiating the bound variables. This we define inductively as follows

$$\frac{}{T \prec Ty\ T} \qquad \frac{X \# T' \qquad T \prec S}{T[X::=T'] \prec \forall X.S}$$

where $X \# T'$ stands for X not occurring in T'. The main reason for this slightly non-standard definition is that for $\prec$ we can easily show that it is preserved under substitutions, namely:

Lemma 2. *If $T \prec S$ then $\theta(T) \prec \theta(S)$.*

Proof. By induction on $\prec$; the only interesting case is the second rule. Since we added the side-condition $X \# T'$ in this rule, the Nominal Datatype Package provides us with a strengthened induction principle for $\prec$ that has the variable convention already built in [15]. As a result we can assume in this case that not only $X \# T'$ (which comes from the rule) but also that $X \# \theta$ (which comes from the variable convention). We need to show that $\theta(T[X::=T']) \prec \theta(\forall X.S)$ holds. By the freshness assumptions we have that the left-hand side is equal to $\theta(T)[X::=\theta(T')]$ and the right-hand side is equal to $\forall X.\theta(S)$. Moreover we have that $X \# \theta(T')$. Consequently we can apply the rule and are done. □

A more standard definition for T being an instance of a types scheme $\forall\{X_1..X_n\}.T'$ involves a substitution θ whose domain is $\{X_1..X_n\}$ and which makes $\theta(T')$ equal to T. This translates in our setting to the following definition:

$$T \prec' S \stackrel{\text{def}}{=} \exists\, Xs\ T'\ \theta.\ S \hookrightarrow Xs \cdot T' \wedge dom\ \theta \approx Xs \wedge \theta(T') = T.$$

However, it is much harder to show Lemma 2 for the $\prec'$-relation: for a start there is no convenient induction principle; also we have to analyse how $\theta(S)$ unbinds, which means we have to rename the binders appropriately. Nevertheless, it is relatively straightforward to show that both relations are equivalent.

Lemma 3. $T \prec S$ if and only if $T \prec' S$.

To prove this lemma, we shall first establish the two facts:

Lemma 4.
(i) *If* $T \prec S$ *then* $T \prec \forall X.S$.
(ii) *If* $S \hookrightarrow Xs \cdot T$ *then* $T \prec S$.

Proof. For the first part, we choose a fresh type variable Y such that $Y \# (S, T, X)$. From the assumption and Lemma 2 we obtain $T[X{:=}TVar\ Y] \prec S[X{:=}TVar\ Y]$. Using the fact $Y \# TVar\ X$, we can derive $T[X{:=}TVar\ Y][Y{:=}TVar\ X] \prec \forall Y.S[X{:=}TVar\ Y]$ using the rule. Because of how we have chosen Y, the left-hand side is equal to T and the right-hand side is (alpha-)equal to $\forall X.S$. Using (i) we can show by a routine induction over $S \hookrightarrow Xs \cdot T$ the second part of the lemma. □

Proof of Lemma 3. The left-to-right direction is by induction on $\prec$ and quite routine. For the other direction we have to show that $\theta(T') \prec S$ holds. We first use Lemma 4(ii) to infer $T' \prec S$ from $S \hookrightarrow Xs \cdot T'$. Appealing to Lemma 2, we can thus infer that $\theta(T') \prec \theta(S)$ holds. By a routine induction over $S \hookrightarrow Xs \cdot T'$, we can show that $\forall X{\in}Xs.\ X \# S$, which in turn implies that $\theta(S) = S$ since $dom\ \theta \approx Xs$. Consequently we can conclude with $\theta(T') \prec S$. □

2.6 Subsumption Relation for Type-Schemes

A type scheme S_1 is subsumed by another type scheme S_2 provided that

$$S_1 \lessdot S_2 \stackrel{\text{def}}{=} \forall T.\ T \prec S_1 \longrightarrow T \prec S_2.$$

Damas shows in [3] (slightly adapted to our setting) that:

Lemma 5. If $S_1 \hookrightarrow Xs_1 \cdot T_1$ and $S_2 \hookrightarrow Xs_2 \cdot T_2$ then

$$S_1 \lessdot S_2 \text{ if and only if } \exists\, \theta.\ dom\ \theta \approx Xs_2 \wedge T_1 = \theta(T_2) \wedge (\forall X{\in}Xs_1.\ X \# S_2).$$

Proof. The left-to-right directions is as follows: from the first assumption we know $T_1 \prec S_1$ by Lemma 4(ii) and thus $T_1 \prec S_2$. From this we obtain a θ such that $dom\ \theta \approx Xs_2$ and $T_1 = \theta(T_2)$ holds. The property $\forall X{\in}Xs_1.\ X \# S_2$ follows from the observation that all free type variables of S_2 are also free in T_1. The other direction follows from the fact that if $S_2 \hookrightarrow Xs_2 \cdot T_2$ and $dom\ \theta \approx Xs_2$ then $\theta(T_2) \prec S_2$. □

From Lemma 5 we can derive the following two properties of subsumption which will be useful later on.

Lemma 6.
(i) *If* $S_1 \lessdot S_2$ *then* $ftv\ S_2 \subseteq ftv\ S_1$.
(ii) *If* $Xs_1 \subseteq Xs_2$ *then* $gen\ T\ Xs_1 \lessdot gen\ T\ Xs_2$.

Proof. For the first part we obtain by Lemma 1(i) Xs_1, T_1, Xs_2 and T_2 such that $S_1 \hookrightarrow Xs_1 \cdot T_1$ and $S_2 \hookrightarrow Xs_2 \cdot T_2$. We have further that $(*)$ $ftv\ S_1 \approx ftv\ T_1 - Xs_1$ and $T_1 \prec S_1$. From Lemma 5 we can infer that $\forall X{\in}Xs_1.\ X \# S_2$, which in turn implies that $(**)$ $Xs_1 \cap ftv\ S_2 = \varnothing$. Using the assumption and $T_1 \prec S_1$ we obtain $T_1 \prec S_2$. By induction on $\prec$ we can show that $ftv\ S_1 \subseteq ftv\ T_1$. Using $(**)$ we can infer that $ftv\ S_2 \subseteq ftv\ T_1 - Xs_1$ and hence conclude appealing to $(*)$.

For the second part we have that $gen\ T\ Xs_1 \hookrightarrow Xs_1 \cdot T$ and $gen\ T\ Xs_2 \hookrightarrow Xs_2 \cdot T$ using Lemma 1(ii). By assumption it holds that $\forall X{\in}Xs_1.\ X \# gen\ T\ Xs_2$. Taking the identity substitution, written ε, which maps every $X \in Xs_2$ to $TVar\ X$, then $dom\ \varepsilon \approx Xs_2$ and $T = \varepsilon(T)$. Consequently we can conclude using Lemma 5. □

3 Typing

3.1 Typing Contexts

Typing contexts are lists of (term variable, type scheme)-pairs, i.e. $Ctxt = (var \times tyS)\ list$. A typing context Γ is *valid* when it includes only a single association for every term variable in Γ. We can define the notion of validity by the two rules:

$$\frac{}{valid\ []} \qquad \frac{valid\ \Gamma \qquad a \# \Gamma}{valid\ ((a, S)::\Gamma)}$$

where we attach in the second rule the side-condition that a must be fresh for Γ, which in case of our typing contexts is equivalent to x not occurring in Γ. The application of a substitution θ to a typing context is defined as usual by

$$\theta([]) = [] \qquad \theta((a, S)::\Gamma) = (a, \theta(S))::\theta(\Gamma).$$

The subsumption relation is extended to typing contexts as follows:

$$\frac{}{[] \lessdot []} \qquad \frac{S_1 \lessdot S_2 \qquad \Gamma_1 \lessdot \Gamma_2}{(a, S_1)::\Gamma_1 \lessdot (a, S_2)::\Gamma_2}$$

3.2 Typing Rules

Now we have all the notions in order to define the typing judgment $\Gamma \vdash t : T$:

$$\frac{valid\ \Gamma \qquad (a, S) \in \Gamma \qquad T \prec S}{\Gamma \vdash Var\ a : T}$$

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash App\ t_1\ t_2 : T_2}$$

$$\frac{a \# \Gamma \qquad (a, Ty\ T_1)::\Gamma \vdash t : T_2}{\Gamma \vdash Lam\ a.\ t : T_1 \rightarrow T_2}$$

$$\frac{a \# \Gamma \qquad \Gamma \vdash t_1 : T_1 \qquad (a, close\ \Gamma\ T_1)::\Gamma \vdash t_2 : T_2}{\Gamma \vdash Let\ a\ be\ t_1\ in\ t_2 : T_2}$$

The complexity of this system comes from *close* $\Gamma\ T_1$ in the Let-rule, which stands for the *closure* of T_1 w.r.t. Γ. This means that all free type variables in T_1 are universally quantified, except for those that are also free in Γ

$$\mathit{close}\ \Gamma\ T \stackrel{\text{def}}{=} \mathit{gen}\ T\ (\mathit{ftv}\ T - \mathit{ftv}\ \Gamma).$$

The first formulation of the above set of rules that we are familiar with appeared in the Mini-ML paper [2] where it was proved equivalent to the system by Damas and Milner. The Mini-ML rules are derived from the Damas-Milner rules by incorporating the quantifier rules into the other rules (via *close* and $\prec$), thus making the system syntax-directed.

The above system is much more faithful to standard presentations in the literature than the two previous formalisations based on de Bruijn indices [5, 9]. In fact, it is nearly identical to standard presentations. One exception is that we explicitly require that a pair $(a, \_)$ can only be added to the context Γ if a does not occur in it already. In the literature this corresponds roughly to those formalisations where the authors assume that the a is implicitly renamed beforehand.

We have that the typing relation is preserved under substitutions and that it is monotone under the subsumption relation:

Lemma 7. *If* $\Gamma \vdash t : T$ *then* $\theta(\Gamma) \vdash t : \theta(T)$.

Lemma 8. *If* $\Gamma_1 \vdash t : T$ *and* $\Gamma_1 \lessdot \Gamma_2$ *then* $\Gamma_2 \vdash t : T$.

Proof of Lemma 7. For this lemma we derive a strong induction principle [15] for the typing relation that allows us in the Let-case to assume the bound type variables, that is $\mathit{ftv}\ T_1 - \mathit{ftv}\ \Gamma$, avoid θ. Then the variable case is as follows: We know that validity is preserved under substitution, which implies that *valid* $(\theta(\Gamma))$ holds; $(x, S) \in \Gamma$ implies $(x, \theta(S)) \in \theta(\Gamma)$; and by Lemma 2 we can infer $\theta(T) \prec \theta(S)$. In the Let-case we have that $\forall X \in \mathit{ftv}\ T_1 - \mathit{ftv}\ \Gamma.\ X \# \theta$, which implies that $\theta(\mathit{close}\ \Gamma\ T) = \mathit{close}\ (\theta(\Gamma))\ (\theta(T))$. Consequently all cases can be established by straightforward application of the inference rules. □

Proof of Lemma 8. We show first that $\Gamma_1 \lessdot \Gamma_2$ implies $\mathit{close}\ \Gamma_1\ T \lessdot \mathit{close}\ \Gamma_2\ T$: Using Lemma 6$(i)$ we show by induction that $\mathit{ftv}\ \Gamma_1 \subseteq \mathit{ftv}\ \Gamma_2$ holds; thus $\mathit{ftv}\ T - \mathit{ftv}\ \Gamma_2 \subseteq \mathit{ftv}\ T - \mathit{ftv}\ \Gamma_1$. By Lemma 6$(ii)$ we obtain $\mathit{close}\ \Gamma_1\ T \lessdot \mathit{close}\ \Gamma_2\ T$. Lemma 8 is now by a routine induction on $\Gamma_1 \vdash t : T$ using in the variable case the fact that if $(x, S_1) \in \Gamma_1$ and $\Gamma_1 \lessdot \Gamma_2$ then there exists an S_2 such that $S_1 \lessdot S_2$ and $(x, S_2) \in \Gamma_2$. □

The completeness proof in §5 needs the following lemma:

Lemma 9. $\mathit{close}\ (\theta(\Gamma))\ (\theta(T)) \lessdot \theta(\mathit{close}\ \Gamma\ T)$.

Proof. The proof is by a fiddly alpha-renaming for the type scheme on the right-hand side to move the substitution θ under the binders. Then we appeal to Lemma 5. □

4 Algorithm *W*

4.1 The Rules

At this point we depart from the standard formalisation: instead of defining W as a recursive function $W(\Gamma,t) = (\theta,T)$ which can possibly fail, we define W as an inductive relation $(V,\Gamma,t) \mapsto (V',\theta,T)$ where the lists V and V' contain the type variables that have been used so far by the algorithm. With this we rigorously treat the process of issuing new type variables and also avoid the vagueness present in some presentations of W, which state that a fresh variable is created, but omitting the information what it should be fresh for. Again, we are not the first one that thread through the algorithm a list of type variables that need to be avoided: for example Gunter [6] gives such rules, but does not give a soundness nor completeness proof for these rules; Leroy [7] gives both proofs but includes in the algorithm an infinite set of available type variables.

Our rules for W are as follows:

$$\frac{(a, S) \in \Gamma \qquad \mathit{freshen}\ V\ S\ T}{(V,\Gamma,\mathit{Var}\ a) \mapsto (V \mathbin{@} \mathit{ftv}\ T,[],T)}$$

$$\frac{\begin{array}{c}(V,\Gamma,t_1) \mapsto (V_1,\theta_1,T_1) \qquad (V_1,\theta_1(\Gamma),t_2) \mapsto (V_2,\theta_2,T_2) \\ \mathit{mgu}\ \theta_3\ (\theta_2(T_1))\ (T_2 \to \mathit{TVar}\ X) \qquad X \# V_2\end{array}}{(V,\Gamma,\mathit{App}\ t_1\ t_2) \mapsto (X::V_2,\theta_3 \circ \theta_2 \circ \theta_1,\theta_3(\mathit{TVar}\ X))}$$

$$\frac{\begin{array}{c}(X::V,(a,\ \mathit{Ty}\ (\mathit{TVar}\ X))::\Gamma,t) \mapsto (V',\theta_1,T_1) \\ a \# \Gamma \qquad X \# (\Gamma,\ V)\end{array}}{(V,\Gamma,\mathit{Lam}\ a.\ t) \mapsto (V',\theta_1,\theta_1(\mathit{TVar}\ X) \to T_1)}$$

$$\frac{\begin{array}{c}(V,\Gamma,t_1) \mapsto (V_1,\theta_1,T_1) \qquad a \# \Gamma \\ (V_1,(a,\ \mathit{close}\ (\theta_1(\Gamma))\ T_1)::\theta_1(\Gamma),t_2) \mapsto (V_2,\theta_2,T_2)\end{array}}{(V,\Gamma,\mathit{Let}\ a\ \mathit{be}\ t_1\ \mathit{in}\ t_2) \mapsto (V_2,\theta_2 \circ \theta_1,T_2)}$$

In the variable rule, in order to obtain the most general instance of a type schema, also known as its *generic instance*, all its bound variables are instantiated by *fresh* variables. This is formalised as a predicate $\mathit{freshen}\ V\ S\ T$ where V stands for the type variables to be avoided and T is the generic instance of S. Note that $\mathit{ftv}\ T$ contains the newly introduced type variables. The definition of *freshen* is inductive: type schemes are simply unpacked:

$$\frac{}{\mathit{freshen}\ V\ (\mathit{Ty}\ T)\ T}$$

Quantifiers are instantiated by new variables:

$$\frac{\mathit{freshen}\ V\ S\ T \qquad Y \# (V, X, S, T) \qquad X \# V}{\mathit{freshen}\ V\ (\forall X.S)\ (T[X:=\mathit{TVar}\ Y])}$$

We show that *freshen* is "total" if V and S are viewed as input, and that it produces an instance for S.

Lemma 10.

(i) $\exists T.$ *freshen V S T.*

(ii) *If freshen V S T then* $T \prec S$.

Proof. The first property is by induction on S and involving α-renamings; the second is by induction on *freshen V S T*. □

4.2 Unification

We treat unification as a black box and merely specify its behaviour via a predicate *mgu* θ T_1 T_2 which expresses that θ is a *most general unifier* of T_1 and T_2. We shall rely on the following four properties of *mgu*:

Proposition 1.

(i) *If mgu* θ T_1 T_2 *then* $\theta(T_1) = \theta(T_2)$.

(ii) *If mgu* θ T_1 T_2 *and* $\theta'(T_1) = \theta'(T_2)$ *then* $\exists \delta.\ \theta' \approx \delta \circ \theta$.

(iii) *If mgu* θ T_1 T_2 *then ftv* $\theta \subseteq$ *ftv* (T_1, T_2).

(iv) *If* $\theta(T_1) = \theta(T_2)$ *then* $\exists \theta'.$ *mgu* θ' T_1 T_2.

5 Soundness and Completeness Proofs

The soundness and completeness statements for *W* are as follows:

Theorem 1 (Soundness)**.** *If* $(V,\Gamma,t) \mapsto (V',\theta,T)$ *and valid* Γ *then* $\theta(\Gamma) \vdash t : T$.

Theorem 2 (Completeness)**.** *If* $[] \vdash t : T$ *then* $\exists V\ \theta\ T'.\ ([],[],t) \mapsto (V,\theta,T') \wedge (\exists \delta.\ T = \delta(T'))$.

The proof of the first theorem is by a strong induction over $(V,\Gamma,t) \mapsto (V',\theta,T)$ using Lemma 7. This induction is relatively straightforward and therefore we omit the details.

The proof of the completeness theorem is more interesting: it is by induction on the structure of *t*. However, the statement needs to be strengthened in order to succeed with the induction. The strengthened statement is:

Theorem 3. If $\sigma(\Gamma) \vdash t : T'$ and *ftv* $\Gamma \subseteq V$ then there exist θ, T, δ and V' such that

(i) $(V,\Gamma,t) \mapsto (V',\theta,T)$

(ii) $T' = \delta(T)$

(iii) $\sigma' \approx \delta \circ \theta$ *inside V.*

where we use the notion of two substitutions being *equal over a list of type variables*. This notion is defined as

$$\theta_1 \approx \theta_2 \textit{ inside } Xs \stackrel{\text{def}}{=} \forall X{\in}Xs.\ \theta_1(\textit{TVar } X) = \theta_2(\textit{TVar } X)$$

This relation is reflexive, symmetric and transitive in the arguments θ_1 and θ_2. We can also show that:

Lemma 11.

(i) If $\theta_1 \approx \theta_2$ *inside Xs* and *ftv* $\_ \subseteq$ *Xs* then $\theta_1(\_) = \theta_2(\_)$ where $\_$ stands for types, type schemes, typing contexts and substitutions.

(ii) *If* $X \# Xs$ *then* $\theta \approx (X, T)::\theta$ *inside Xs*.

(iii) *If* $\theta_1 \approx \theta_2$ *inside Xs and ftv* $\theta \subseteq Xs$ *and* $Xs' \subseteq Xs$ *then* $\theta_1 \circ \theta \approx \theta_2 \circ \theta$ *inside Xs'*.

Proof. The first property holds for types, type schemes, typing contexts and substitutions. In each case the property is shown by structural induction. The second property is by a simple calculation. The last is by induction on θ using (i). □

Next we show that the list of used variables increases in every "recursive call" in the algorithm and that the list of used variables indeed contains all used variables.

Lemma 12.

(i) *If* $(V,\Gamma,t) \mapsto (V',\theta,T)$ *then* $V \subseteq V'$.

(ii) *If* $(V,\Gamma,t) \mapsto (V',\theta,T)$ *and ftv* $\Gamma \subseteq V$ *then ftv* $(\Gamma, \theta, T) \subseteq V'$.

Proof. Both are by induction on the rules of *W*. □

Now we have everything in place to establish the completeness of *W*.

Proof of Theorem 3: The proof is by strong structural induction over *t* avoiding Γ and generalising over *V*, σ and *T*.

Variable-Case: By assumption we have $\sigma(\Gamma) \vdash Var\ x : T'$. From this we obtain an *S* such that $T' \prec S$ and (i) $(x, S) \in \sigma(\Gamma)$. From (i) we obtain an S' such that (ii) $(x, S') \in \Gamma$ and $S = \sigma(S')$. By Lemma 10(i) we have an *T* such that (iii) *freshen V S' T*. From this and (ii) we can derive $(V,\Gamma,Var\ x) \mapsto (V \mathbin{@} ftv\ T, [], T)$. Using (iii) and $T' \prec \sigma(S')$ we obtain a δ such that $T' = \delta(T)$ and $\sigma \approx \delta \circ [] $ *inside V*, which concludes this case.

Lambda-Case: By the variable convention we have built into the induction principle for the typing relation, we know $x \# \Gamma$, from which we can infer that also $x \# \sigma(\Gamma)$ holds. By assumption we have $\sigma(\Gamma) \vdash Lam\ x.\ t : T'$. Taking the last two facts together we obtain an T_1 and T_2 such that

$$(i)\ (x, Ty\ T_1)::\sigma(\Gamma) \vdash t : T_2 \quad \text{and} \quad (ii)\ T' = T_1 \to T_2.$$

We now choose a new type-variable *X* such that $X \# (\sigma, \Gamma, \sigma(\Gamma), V)$ holds. By assumption we have *ftv* $\Gamma \subseteq V$, which implies *ftv* $((x, Ty\ (TVar\ X))::\Gamma) \subseteq X::V$. We also have that

$$(iii)\ (x, Ty\ T_1)::\sigma(\Gamma) = ((X, T_1)::\sigma)((x, Ty\ (TVar\ X))::\Gamma)$$

because by condition $X \# \sigma$ it holds that $((X, T_1)::\sigma)(\Gamma) = \sigma(\Gamma)[X:=T_1]$ and by condition $X \# \sigma(\Gamma)$ that $\sigma(\Gamma)[X:=T_1] = \sigma(\Gamma)$. Fact (iii) allows us to apply the induction hypothesis obtaining a θ, T_1, δ and V' such that

(iv) $(X::V,(x, Ty\ (TVar\ X))::\Gamma,t) \mapsto (V',\theta,T)$

(v) $T_2 = \delta(T)$

(vi) $(X, T_1)::\sigma \approx \delta \circ \theta$ *inside* $X::V$.

We can conclude the Lambda-Case with

$$(V,\Gamma,\mathit{Lam}\ x.\ t) \mapsto (V',\theta,\theta(\mathit{TVar}\ X)\rightarrow T).$$

which follows from $x \# \Gamma$ and $X \# (\Gamma, V)$. By the calculation

$$\begin{array}{rcll} T' & = & T_1\rightarrow T_2 & \text{by } (ii) \\ & = & T_1\rightarrow\delta(T) & \text{by } (v) \\ & = & \delta \circ \theta(\mathit{TVar}\ X)\rightarrow\delta(T) & \text{by } (vi) \end{array}$$

we have that $T' = \delta(\theta(\mathit{TVar}\ X)\rightarrow T)$. Finally we have that $\sigma \approx \delta \circ \theta$ *inside* V because

$$\begin{array}{rcll} \sigma & \approx & (X, T_1)::\sigma \quad \mathit{inside}\ V & \text{by } X \# V \text{ and Lemma 11}(ii) \\ & \approx & \delta \circ \theta \quad \mathit{inside}\ V & \text{by } (vi) \end{array}$$

Application-Case: By assumption we have $\sigma(\Gamma) \vdash \mathit{App}\ t_1\ t_2 : T'$, from which we obtain a T'' such that

$$(i)\ \ \sigma(\Gamma) \vdash t_1 : T''\rightarrow T' \quad \text{and} \quad (ii)\ \ \sigma(\Gamma) \vdash t_2 : T''$$

holds. Using (i) and the assumption *ftv* $\Gamma \subseteq V$, the induction hypothesis gives us a θ, T, δ and V' such that

$$(iii)\ (V,\Gamma,t_1) \mapsto (V',\theta,T) \qquad (iv)\ T''\rightarrow T' = \delta(T) \qquad (v)\ \sigma \approx \delta \circ \theta \ \mathit{inside}\ V\,.$$

From *ftv* $\Gamma \subseteq V$ and (v) we know that $\sigma(\Gamma) = \delta(\theta(\Gamma))$ holds. Using Lemma 12(ii) and (iii) we can infer *ftv* $(\theta, \Gamma, T) \subseteq V'$ which means *ftv* $(\theta(\Gamma)) \subseteq V'$ holds. In the induction hypothesis for t_2 we can set Γ to $\theta(\Gamma)$ and σ to δ. By using (ii) this gives us a θ', T''', δ' and V'' such that

$$\begin{array}{rl} (vi) & (V',\theta(\Gamma),t2) \mapsto (V'',\theta',T''') \\ (vii) & T'' = \delta'(T''') \\ (viii) & \delta \approx \delta' \circ \theta' \ \mathit{inside}\ V'\,. \end{array}$$

We now choose a new type-variable X such that $X \# (V'', T''', \theta'(T))$ holds. By calculation we can show that the type $((X, T')::\delta')(\theta'(T))$ is equal to $((X, T')::\delta')(T'''\rightarrow \mathit{TVar}\ X)$.

Let-Case: By the variable convention again, we know $x \# \Gamma$, from which we can infer that also $x \# \sigma(\Gamma)$ holds. By assumption we have $\sigma(\Gamma) \vdash \mathit{Let}\ x\ \mathit{be}\ t_1\ \mathit{in}\ t_2 : T'$. Taking the last two facts together we obtain a T'' such that

$$(i)\ \ \sigma(\Gamma) \vdash t_1 : T'' \qquad (ii)\ \ (x, \mathit{close}\ (\sigma(\Gamma))\ T'')::\sigma(\Gamma) \vdash t_2 : T'$$

hold. By assumption $(*)$ *ftv* $\Gamma \subseteq V$ and induction hypothesis for t_1, we obtain a θ, T_1, δ and V' such that

$$(iii)\ (V,\Gamma,t_1) \mapsto (V',\theta,T_1) \qquad (iv)\ T'' = \delta(T_1) \qquad (v)\ \sigma \approx \delta \circ \theta \ \mathit{inside}\ V\,.$$

By the assumption (*) and (v) we have $\sigma(\Gamma) = \delta(\theta(\Gamma))$. Using Lemma 9 and (iv) we can hence infer that

$$(x, \mathit{close}\ (\sigma(\Gamma))\ T'')::\sigma(\Gamma) \lessdot \delta((x, \mathit{close}\ (\theta(\Gamma))\ T_1)::\theta(\Gamma))\,.$$

Consequently we can use Lemma 8 and (ii) to infer

$$(vi)\quad \delta((x, \mathit{close}\ (\theta(\Gamma))\ T_1)::\theta(\Gamma)) \vdash t_2 : T'$$

We can now use the induction hypothesis for t_2 where we instantiate σ with δ, and Γ with $(x, \mathit{close}\ (\theta(\Gamma))\ T_1)::\theta(\Gamma)$. Since from (iii), $(*)$ and Lemma 12(ii) we have $(**)$ $\mathit{ftv}\ (\theta, \Gamma, T_1) \subseteq V'$ and hence $\mathit{ftv}\ (\theta(\Gamma)) \subseteq V'$, we can show that $\mathit{ftv}\ ((x, \mathit{close}\ (\theta(\Gamma))\ T_1)::\theta(\Gamma)) \subseteq V'$. This allows us to use the induction hypothesis for t_2 to obtain a θ', T_2, δ' and V'' such that

$$\begin{array}{rl} (vii) & (V',(x, \mathit{close}\ (\theta(\Gamma))\ T_1)::\theta(\Gamma),t_2) \mapsto (V'',\theta',T_2) \\ (viii) & T' = \delta'(T_2) \\ (ix) & \delta \approx \delta' \circ \theta' \ \mathit{inside}\ V' \,. \end{array}$$

Using $(viii)$ we can conclude provided we can further show that

$$(V,\Gamma,\mathit{Let}\ x\ \mathit{be}\ t_1\ \mathit{in}\ t_2) \mapsto (V'',\theta' \circ \theta,T_2) \quad \text{and} \quad \sigma \approx \delta' \circ \theta' \circ \theta\ \mathit{inside}\ V$$

hold. The first follows from (iii), (vii) and the variable convention $x \# \Gamma$. The second follows using $V \subseteq V'$ (from (iii)) and the calculation:

$$\begin{array}{lcll} \sigma & \approx & \delta \circ \theta & \mathit{inside}\ V \quad \text{by } (v) \\ & \approx & \delta' \circ \theta' \circ \theta & \mathit{inside}\ V \quad \text{by } (ix),\ (**) \text{ and Lem. } 11(iii) \end{array}$$

This concludes the proof. □

6 Conclusion

While the typing algorithm *W* is a classic algorithm implemented numerous times, there are surprisingly few careful descriptions of soundness and completeness proofs that can be readily used in a formalisation. For example in [13] a version of *W* is presented that leaves the choice of fresh variables implicit and only states our Theorems 1 and 2; [6] gives a rigorous description for how to chose fresh variables, but only presents the details for soundness and completeness where this choice is left implicit. Two slick machine-checked proofs for soundness and completeness were implemented previously [5, 9]. But both of these proofs code type and term variables using de Bruijn indices. As a result they were not easily adaptable to our setting, since they were streamlined for the representation of type schemes based on de Bruijn indices. For example, many proofs in [9] are by induction over the structure of type schemes, which we could not follow with our representation involving iterated binders. Most of the inspiration for our formalised proofs we have drawn from the treatment of soundness and completeness given by Leroy [7]. He encodes type schemes by quantifying over a whole set of variables in one go, and it took surprisingly a lot of work to adapt his proofs to our representation where we can only bind a single type variable in each quantification-step.

Although the Nominal Datatype Package provides a convenient reasoning infrastructure for α-equivalence classes, it did not provide as much help for this formalisation

as one might hope. One reason is that the package is not yet up to the task of representing general binding structures, and thus it is not yet possible to implement type schemes with a set of quantified type variables. Another reason is that the algorithm *W* contains many subtle "low level" operations involving type variables. This necessitates many α-renamings that had to be formalised explicitly without much help from the nominal infrastructure. However, we do not think that this can be avoided in any representation technique for binders (bar de Bruijn indices) that has been put forward in various theorem provers. Our formalisation is part of the Nominal Datatype Package, which can be downloaded at:

`http://isabelle.in.tum.de/nominal/`

References

[1] D. Clément, J. Despeyroux, T. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. In *K. Fuchi and M. Nivat, editors, proceedings of the France-Japan AI and CS Symposium, ICOT, Japan*, pages 49–89, 1986. Also Technical Memorandum PL-86-6 Information Processing Society of Japan and Rapport de recherche #0416, INRIA.

[2] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 13–27, 1986.

[3] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1984.

[4] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symp. Principles of Programming Languages*, pages 207–212, 1982.

[5] C. Dubois and V. Ménissier-Morain. Certification of a type inference tool for ML: Damas-Milner within Coq. *J. Automated Reasoning*, 23:319–346, 1999.

[6] C. A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.

[7] X. Leroy. *Polymorphic Typing of an Algorithmic Language*. PhD thesis, University Paris 7, 1992. INRIA Research Report, No 1778.

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

[9] W. Naraschewski and T. Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *J. Automated Reasoning*, 23:299–318, 1999.

[10] T. Nipkow and L. C. Paulson. Proof pearl: Defining functions over finite sets. In J. Hurd, editor, *Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *Lect. Notes in Comp. Sci.*, pages 385–396. Springer-Verlag, 2005.

[11] A. M. Pitts. Nominal logic, A first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[12] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *Proc. of the 5th International Conference on Mathematics of Program Construction (MPC)*, volume 1837 of *LNCS*, pages 230–255, 2000.

[13] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Edinburgh University, 1988.

[14] C. Urban and S. Berghofer. A Recursion Combinator for Nominal Datatypes Implemented in Isabelle/HOL. In *Proc. of the 3rd International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *LNAI*, pages 498–512, 2006.

[15] C. Urban, S. Berghofer, and M. Norrish. Barendregt's Variable Convention in Rule Inductions. In *Proc. of the 21th International Conference on Automated Deduction (CADE)*, volume 4603 of *LNAI*, pages 35–50, 2007.

[16] C. Urban and C. Tasson. Nominal Techniques in Isabelle/HOL. In *Proc. of the 20th International Conference on Automated Deduction (CADE)*, volume 3632 of *LNCS*, pages 38–53, 2005.