Type Inference Verified: Algorithm $\mathcal{W}$ in Isabelle/HOL *

Wolfgang Naraschewski and Tobias Nipkow

Technische Universität München**

1 Introduction

Most functional programming languages [13, 11, 2] have a common core: the simply typed λ-calculus enriched with `let`-expressions, i.e. local definitions of polymorphic values. This language is called Mini-ML [3]. The set of well-typed Mini-ML expressions is inductively defined by a set of inference rules. One of the key properties of Mini-ML is that every well-typed expression has a most general type. The computation of the most general type is called *type inference*. It was first studied by Hindley [10] in the context of combinatory logic and later independently by Milner [12] for Mini-ML. Milner's type inference algorithm is known as algorithm $\mathcal{W}$. Damas [4, 5] proved the completeness of $\mathcal{W}$.

This paper presents the first machine-checked proof of correctness and completeness of $\mathcal{W}$. It is an extension of the work by Nazareth and Nipkow [14], who treated the monomorphic case (no `let`-expressions). A partial verification of $\mathcal{W}$ in the proof checker Coq is reported by Dubois and Ménissier-Morain [7]. Our paper provides the definition of (due to lack of space: almost) all concepts, the key lemmas, but almost no proofs. The complete development is accessible via `http://www4.informatik.tu-muenchen.de/~nipkow/isabelle/HOL/MiniML/`.

In the sequel we employ the following terminology: an *informal* proof is one which has been developed on paper in the mathematical tradition, whereas a *formal* proof means a machine-checked one.

2 Isabelle/HOL

Isabelle is an interactive theorem prover which can be instantiated with different object logics. One particularly well-developed instantiation is Isabelle/HOL, which supports Church's formulation of Higher Order Logic and is very close to Gordon's HOL system [8]. In the remainder of the paper HOL is short for Isabelle/HOL. A detailed introduction to Isabelle and HOL can be found elsewhere [18].

Logical expressions are written as usually, except for the fact that we use two different implications ($\Longrightarrow$, $\Rightarrow$) and equalities ($\equiv$, $=$). This distinction stems

* Research supported by Esprit WG TYPES and DFG SPP Deduktion.

** Institut für Informatik, 80290 München, Germany.
`http://www4.informatik.tu-muenchen.de/~{narasche,nipkow}/`
`{narasche,nipkow}@informatik.tu-muenchen.de`

from the object and meta-logic, respectively, and can be ignored while reading this paper. The notation $[\![$ `A1; ...; An` $]\!] \Longrightarrow$ `A` is an abbreviation for the nested implication `A1` $\Longrightarrow$ `...` $\Longrightarrow$ `An` $\Longrightarrow$ `A`. The predicate $\leq$ is overloaded and applies to natural numbers and to sets.

Types follow the syntax for ML-types, except that the function arrow is $\Rightarrow$ rather than $\rightarrow$. A term `t` is constrained to be of type τ by writing `t ::` τ. Isabelle also provides Haskell-like *type classes* [11], the details of which are explained as we go along. A type variable α is restricted to be of class `c` by writing α `:: c`.

Theories introduce constants with the keyword `consts`, non-recursive definitions with `defs`, and primitive recursive definitions with the keyword `primrec`. For general axioms the keyword `rules` is used. Further constructs are explained as we encounter them.

Although we do not present any of the proofs, we usually indicate their complexity. If we do not state any complexity the proof is almost automatic. That means, it is either solved by rewriting or by the "classical reasoner", `fast_tac` in Isabelle parlance [19]. The latter provides a reasonable degree of automation for predicate calculus proofs. Note, however, that its success depends on the right selection of lemmas supplied as parameters.

3 Types, type variables and substitutions

This section describes the language of object-level types of Mini-ML. They should not be confused with Isabelle's built-in meta-level type system described in the previous section. To avoid ambiguities we sometimes call the latter *HOL types*.

This is not a section of exciting theorems but one of basic definitions. Although the corresponding Isabelle theory contains over 90 theorems, only a few are worth mentioning explicitly, many are proved automatically and none are particularly difficult.

3.1 Types and type variables

Types consist only of type variables and the function space constructor. They are immediately formalized as an inductive data type:

```
datatype typ = TVar nat | "→" typ typ (infixr 70)
```

Type variables are modeled by natural numbers.

In order to express ML-style polymorphism we need quantified types as well, so called *type schemes*. They are of the form $\forall\alpha_1,\ldots,\alpha_n.\tau$, where the α_i are type variables and τ is a type. A natural formalization would be a pair of a set or list of type variables and a type. However, anybody familiar with formalizations of terms with bound variables appreciates the advantage of being able to distinguish bound and free variables syntactically (see e.g. [15]). Therefore we decided on the following data type which has separate constructors for free and bound variables:

```
datatype type_scheme = FVar nat | BVar nat |
                       "=→" type_scheme type_scheme (infixr 70)
```

This is essentially the representation chosen by Dubois and Ménissier-Morain [7].

For example, `BVar i =→ FVar j` represents the type scheme $\forall\alpha.\alpha \to \beta$. Note that `i` and `j` need not be distinct. In particular, there is no de Bruijn index-like mechanism going on here that requires a particular numbering scheme for bound variables. This is because all quantifiers occur at the same (outermost) level and their order is irrelevant.

Having separated types and type schemes (in contrast to informal treatments, which regard types as a subset of type schemes), we need an embedding:

```
consts  make_scheme :: typ ⇒ type_scheme
primrec typ
  make_scheme (TVar n)   = FVar n
  make_scheme (t1 → t2) = (make_scheme t1) =→ (make_scheme t2)
```

which is easily proved injective.

There are two functions for collecting the free and the bound variables in a type. In addition we will need to determine the free variables of a type scheme and of a list of type schemes. To avoid cluttering the text with lots of special notation, we use Isabelle's overloading facility which resembles Haskell-style type classes [11]. We introduce a new class `type_struct`

```
classes type_struct < term
```

as a subclass of the predefined class `term` of all HOL types and declare `free_tv` and `bound_tv` for all types of class `type_struct`:

```
consts free_tv, bound_tv :: (α::type_struct) ⇒ nat set
```

There are three instances of class `type_struct`:

```
instance typ ::         type_struct
instance type_scheme :: type_struct
instance list ::        (type_struct)type_struct
```

The final one states that a list over a type of class `type_struct` is again of class `type_struct`. Values of a type of class `type_struct` are called *type structures*. We can now define the corresponding instances of `free_tv`

```
primrec typ
  free_tv (TVar m) = {m}
  free_tv (t1 → t2) = (free_tv t1) ∪ (free_tv t2)


primrec type_scheme
  free_tv (FVar m) = {m}
  free_tv (BVar m) = {}
  free_tv (S1 =→ S2) = (free_tv S1) ∪ (free_tv S2)
```

```
primrec list
  free_tv [] = {}
  free_tv (x#xs) = (free_tv x) ∪ (free_tv xs)
```

and `bound_tv`. The latter are omitted because they follow the same pattern.

3.2 Substitutions

The literature usually formalizes substitutions as "finite" functions from variables to types, i.e. functions which are distinct from the identity at finitely many points only. We have chosen to formalize substitutions as ordinary functions:

```
types subst = nat ⇒ typ
```

This is not just in contrast to the informal literature, but also to other formal verifications involving substitutions [17, 7], which use association lists. The two reasons for our choice are:

1. We believe that functions are easier to handle than association lists.
2. We wanted to find out if the restriction to finite functions is essential or not.

The identity substitution is

```
consts id_subst :: subst
defs   id_subst ≡ (λn.TVar n)
```

Strictly speaking there is no need to introduce `id_subst` because it is identical to `TVar` by η-contraction. It is only a concession to readability.

Substitutions can be extended to types, type schemes, etc. Thus we introduce an overloaded application operation

```
consts app_subst :: subst ⇒ (α::type_struct) ⇒ α  ("$")
```

which can be applied to arbitrary type structures. The "`$`" is syntactic sugar for `app_subst`. Because identifiers in Isabelle may not contain "`$`" we may write `$S` instead of `$ S`. The notation `$S` emphasizes that we regard "`$`" as a modifier acting on the substitution `S`. We define `app_subst` for types, type schemes and lists:

```
primrec typ
  $S (TVar n) = S n
  $S (t1 → t2) = ($S t1) → ($S t2)

primrec type_scheme
  $S (FVar n) = make_scheme (S n)
  $S (BVar n) = BVar n
  $S (sch1 =→ sch2) = ($S sch1) =→ ($S sch2)

defs $S ≡ map ($S)
```

Note that in the last case `$S` on the left has type α `list` $\Rightarrow$ α `list` and on the right type $\alpha \Rightarrow \alpha$, where α`::type_struct`.

The *domain* and *codomain* of a substitution are defined as usual:

```
consts dom, cod :: subst ⇒ nat set
defs   dom S ≡ {n. S n ≠ TVar n}
       cod S ≡ UN m ∈ dom S. free_tv (S m)
```

We call the union of domain and codomain the set of free variables of a substitution. In order to reuse `free_tv` we would need to declare

```
instance subst :: type_struct
```

which is not possible in Isabelle (or Haskell) because `subst` is not an atomic type but abbreviates `nat` $\Rightarrow$ `typ`. Instead, we must state the propagation of class membership for the type constructor $\Rightarrow$. We already met this mechanism when adding the list types to class `type_struct`. In the same way we define a functionality for the type constructor $\Rightarrow$ which is the infix name for `fun`:

```
instance fun :: (term,type_struct)type_struct
```

Because `nat::term` and `typ::type_struct`, this implies `subst::type_struct`. Note, however, that it also implies further types to belong to class `type_struct`, for example `bool` $\Rightarrow$ `typ`. This may seem a bit permissive but does no harm. Now we can define the `subst` instance of `free_tv`:

```
defs free_tv S ≡ (dom S) ∪ (cod S)
```

3.3 New type variables

Algorithm $\mathcal{W}$ needs to generate new type variables. Informal descriptions of the algorithm almost invariably ignore this issue. It is simply assumed that there always exists some type variable never used before. The obvious formalization is to pass the set of already used type variables as an argument to $\mathcal{W}$, add all new variables generated during the execution to this set, and return the enlarged set upon successful termination (see e.g. [16]). Because our type variables are natural numbers and thus ordered linearly, it suffices to pass $\mathcal{W}$ a counter instead of a set of variables and to increment the counter every time a new variable is needed. Predicate `new_tv` formalizes this notion of a new type variable. It takes a type variable and a type structure and determines whether the given variable is greater than any type variable occurring in the structure. Such a type variable is called *new w.r.t. the structure.*

```
consts new_tv :: nat ⇒ (α::type_struct) ⇒ bool
defs   new_tv n ts ≡ ∀m. m ∈ free_tv ts ⟶ m < n
```

This predicate is a necessary precondition for most propositions about algorithm $\mathcal{W}$. A simple consequence is

```
⟦ n ≤ m; new_tv n ts ⟧ ⟹ new_tv m ts
```

which holds for arbitrary type structures `ts`.

Proofs about $\mathcal{W}$ frequently require that we can choose new variables. This is expressed by the following lemmas

```
∃n. new_tv n (t::typ)
∃n. new_tv n (sch::type_scheme)
∃n. new_tv n (A::type_scheme list)
```

which depend on the finite nature of the type structures involved. These lemmas are only required since we decided to represent substitutions as ordinary functions. This is the only price we had to pay for our added generality.

3.4 Unification

Type inference is based on unification of types. Of course the correctness of type inference does not depend on any particular implementation of unification but merely on general properties of unification. Therefore we introduce a function `mgu` (*m*ost *g*eneral *u*nifier), specify its characteristic properties, but provide no implementation. This is the only point in the whole development where we introduce new axioms as opposed to consistency preserving definitions. Of course we know that a function `mgu` which satisfies the axioms exists. Alternatively, we could have made `mgu` a parameter of all functions using it and the axioms about `mgu` preconditions of the theorems about those functions. However, that is overkill because `mgu` is not intended as a parameter of individual functions but of the whole development. It requires some form of parameterized theories to express this.

Unification may fail. To model the distinction between a successful computation and a failure situation we use the predefined HOL data type

```
datatype α option = None | Some α
```

Unification either terminates normally returning `Some(S)` for some substitution `S`, or indicates a failure situation by returning `None`:

```
consts mgu :: typ ⇒ typ ⇒ subst option
```

A most general unifier should satisfy the following three axioms:

```
rules mgu t1 t2 = Some U ⟹ $U t1 = $U t2
      ⟦ mgu t1 t2 = Some U; $S t1 = $S t2 ⟧ ⟹ ∃R. S = $R o U
      $S t1 = $S t2 ⟹ ∃U. mgu t1 t2 = Some U
```

The first axiom requires the result of `mgu` to be a unifier of the given types. The second one states that the computed unifier is a most general one. The third one requires `mgu` to return some result if its arguments are unifyable. This prevents trivial implementations which satisfy the first two axioms by always returning `None`.

Most general unifiers are only unique up to renaming of variables. Such a renaming may even introduce type variables not occurring in the types to be unified. Because we want to keep track of used variables we need one last axiom:

$$\texttt{mgu t1 t2 = Some U} \Longrightarrow \texttt{free\_tv U} \leq \texttt{free\_tv t1} \cup \texttt{free\_tv t2} \qquad (1)$$

This ensures that the algorithm does not introduce new type variables. We can then show that unification preserves the new type variable property:

$$[\![\texttt{ mgu t1 t2 = Some U; new\_tv n t1; new\_tv n t2 }]\!] \Longrightarrow \texttt{new\_tv n U}$$

4 Well-Typed Expressions

Lambda expressions are written in de Bruijn notation [6] as an inductive data type with constructors for variables, abstraction, application and let:

```
datatype expr = Var nat | Abs expr |
                App expr expr | LET expr expr
```

Note that by `Var` we denote variables of expressions, whereas `TVar` and `FVar` represent type variables. The representation of variables, abstraction and application is quite standard, whereas the representation of the `let`-construct is less familiar. Before illustrating the `let`-constructor, we shortly recapitulate the idea of de Bruijn indices: The index `i` in a subterm `Var i` indicates that, when moving upward, `i` abstractions must be traversed until the corresponding binder is found. If there are no more than `i` surrounding abstractions, the variable is free, e.g. `Var 0` on its own. For example take the expression `Abs (App (Var 0) (Abs (App (Var 0) (Var 1))))` which stands for $\lambda x.(x\ (\lambda y.y\ x))$. This example shows that different occurrences of the same bound variable may be represented by different indices, depending on the depth of abstractions.

Semantically the expression `LET e1 e2` is short for `App (Abs e2) e1`. To see why this representation of `LET` is meaningful, we give the following example. The expression `LET (Abs (Var 0)) (App (Var 0) (Var 0))` is equivalent to `App (Abs (App (Var 0) (Var 0))) (Abs (Var 0))` which in turn stands for the application $(\lambda x.x\ x)\ (\lambda x.x) \xrightarrow{\beta} \lambda x.x\ \lambda x.x$ or in short $let\ f\ =\ \lambda x.x\ in\ f\ f$.

The datatype `expr` represents all lambda-expressions, but we are only interested in *well-typed* expressions, which we identify by a relation. To ensure consistent typing of multiple occurrences of the same variable, we keep track of its type scheme in a context. Since we use de Bruijn notation, the context is simply a list of type schemes, with the type scheme of variable `i` located at the `i`-th position.

```
types ctxt = type_scheme list
```

Formally, we introduce a relation between contexts, lambda-expressions and types:

```
consts has_type :: (ctxt * expr * typ)set
```

The proposition `(A,e,t):has_type` should be read as "In context `A` expression `e` has type `t`". This relation allows an expression to have more than one type in a given context. To ease readability, Isabelle permits the more conventional notation `A |- e :: t` instead of `(A,e,t):has_type`. Note that the delimiter `::` is now used for type annotations both in the logic and in the object level lambda-expressions (`expr`). The latter is easily distinguished by its leading |−.

Relation `has_type` is defined inductively, i.e. by a set of inference rules. Proposition `A |- e :: t` holds iff it can be derived from the inference rules. HOL provides a package for defining inductive sets, which defines `has_type` to be the least set closed under the given inference rules.

```
inductive has_type
⟦ n < length A; t <| (nth n A) ⟧ ⟹ A |- Var n :: t
⟦ (make_scheme t1)#A |- e :: t2 ⟧ ⟹ A |- Abs e :: t1 → t2
⟦ A |- e1 :: t2 → t1; A |- e2 :: t2 ⟧ ⟹ A |- App e1 e2 :: t1
⟦ A |- e1 :: t1; (gen A t1)#A |- e2 :: t ⟧ ⟹ A |- LET e1 e2 :: t
```

In the inference rules we encounter some new definitions which we shortly explain before justifying the inference rules themselves.

The function `nth :: nat ⇒ α list ⇒ α` selects the nth element of a list. A tuple `(t,sch)` is related by `<| :: typ ⇒ type_scheme ⇒ bool` if and only if `t` is generated from `sch` by instantiating types for all bound variables. The function `gen :: ctxt ⇒ typ ⇒ type_scheme` is used to generalize the free variables of a type which do not occur free in the context.

Modulo the fact that we use de Bruijn notation, these are the type inference rules of the system DM' proposed in [3]. We have not based our case study on the original system of Damas [5] since in the equivalent system DM' there is exactly one inference rule for each syntactic construct, which eases the soundness and completeness proof.

In our representation of the system DM' we have modified the `Var`-rule slightly, which does no harm since we have proven this rule to be equivalent to the one in DM'. Note that the simplicity of the `Abs`-rule is due to de Bruijn notation: the extended context `(make_scheme t1)#A` takes care of the fact that when descending into an abstraction all references to variables bound outside shift by 1. A similar argument holds for the `LET`-rule, except for the fact that `t1` is generalized, which is the only source of polymorphism in the system.

Clearly the properties of <| and `gen` play a crucial role in the soundness and completeness proof. Hence we study them in more detail.

4.1 Instances of type schemes

The whole point of type schemes is to represent a set of types, namely all those that can be obtained by instantiating the quantified (i.e. bound) variables. This is formalized by

```
consts  bound_typ_inst :: subst ⇒ type_scheme ⇒ typ
primrec type_scheme
```

```
  bound_typ_inst S (FVar n) = TVar n
  bound_typ_inst S (BVar n) = S n
  bound_typ_inst S (sch1 =-> sch2) =
    (bound_typ_inst S sch1) -> (bound_typ_inst S sch2)
```

and induces a natural ordering between types and type schemes:

```
consts ◁| :: typ ⇒ type_scheme ⇒ bool (infixr 70)
defs   t ◁| sch ≡ ∃S. t = (bound_typ_inst S sch)
```

This in turn induces an ordering between type schemes:

```
sch' ≤ (sch::type_scheme) ≡ ∀t. t ◁| sch' ⟶ t ◁| sch
```

Both ◁| and ≤ are pronounced "is a generic instance of". The key property of ≤ is preservation under substitutions:

```
sch' ≤ sch ⟹ $S sch' ≤ $S sch                                    (2)
```

This proposition looks simple enough but its proof eluded us until we consulted Damas [5, Prop. 10, p. 51], where this very lemma is proved. Damas uses an alternative characterization of `sch' ≤ sch`: `sch'` can be obtained by instantiating the quantified variables in `sch` and quantifying once more, subject to certain freeness conditions [5, Prop. 9, p. 50]. In order to formalize this in our framework, we need an operation for replacing bound variables by type schemes:

```
consts bound_scheme_inst ::
         (nat ⇒ type_scheme) ⇒ type_scheme ⇒ type_scheme
primrec type_scheme
  bound_scheme_inst S (FVar n) = FVar n
  bound_scheme_inst S (BVar n) = S n
  bound_scheme_inst S (sch1 =-> sch2) =
    (bound_scheme_inst S sch1) =-> (bound_scheme_inst S sch2)
```

Now the equivalent of Prop. 9 by Damas simply becomes

```
(sch' ≤ sch) = (∃B. sch' = bound_scheme_inst B sch)             (3)
```

There are no freeness conditions attached because free and bound variables are syntactically distinct. Using this proposition we can reduce (2) to the goal

```
∃B'. $S (bound_scheme_inst B sch) = bound_scheme_inst B' ($S sch)
```

which becomes trivial when applying the following additional lemma:

```
$S (bound_scheme_inst B sch) = bound_scheme_inst ($S o B) ($S sch)
```

This lemma may look a bit mysterious at first, but it is simply a form of the well known substitution lemma $s[t/x][u/y] = s[u/y][t[u/y]/x]$ provided x is not free in u. The freeness condition again disappears because free and bound variables are kept apart.

It is interesting to note that the proof of (2) required two additional concepts: the function `bound_scheme_inst` and another function (hidden in the proof of (3)) which replaces type variables in types by type schemes. Both functions are purely local to the proof.

4.2 Generalization

As already mentioned, the function `gen` is used to generalize the free variables of a type which do not occur free in the context. Formally this is captured by the following definition:

```
consts gen :: ctxt ⇒ typ ⇒ type_scheme
primrec gen typ
  gen A (TVar n) = if n ∈ (free_tv A) then (FVar n) else (BVar n)
  gen A (t1 → t2) = (gen A t1) =→ (gen A t2)
```

In the soundness and completeness proof we use some simple propositions about `gen` like

```
free_tv t ≤ free_tv A ⟹ gen A t = make_scheme t
free_tv (gen ($S A) t # $S A) = free_tv ($S A)
gen ($S A) ($S t) ≤ $S (gen A t)                                    (4)
```

which can be proved almost automatically. The only lemma about `gen` that caused substantial problems during the development of the case study is the following:

```
(free_tv S) ∩ (free_tv t − free_tv A) = {}                          (5)
   ⟹ gen ($S A) ($S t) = $S (gen A t)
```

This lemma determines under which circumstances substitutions are compatible with `gen`. To see why in general substitutions are not compatible with `gen` consider the following counter example:

```
t ≡ TVar x   A ≡ [FVar y]   S x ≡ TVar y   S y ≡ TVar y   with x ≠ y
```

In this particular case the substitution `S` clearly is not compatible with `gen`, since

```
gen [FVar y] TVar y = FVar y ≠ BVar x = $S (gen [FVar y] TVar x)
```

Though the lemma itself is not difficult to prove, its application in the proofs appeared to be subtle. Each time we intended to use the lemma, its precondition did not hold. In all these cases, though, it was possible to replace the substitution involved by an equivalent one which fulfills the precondition.

4.3 Typing is closed under substitution

The following theorem shows that `has_type` is closed under substitution:

$\texttt{A |- e :: t} \Longrightarrow \forall\texttt{S. \$S A |- e :: \$S t}$

For the monomorphic case it was stated in [14] that the theorem can be proved almost automatically by induction on the derivation of `A |- e :: t`. When dealing with `let`-polymorphism, the induction also includes the `let`-case which is non-trivial and therefore presented in more detail. In this case the goal

`$S A |- LET e1 e2 :: $S t`

has to be solved relying on the assumptions

$\forall$`S. $S A |- e1 :: $S t1`
$\forall$`S. $S (gen A t1)#A |- e2 :: $S t`

Applying the `LET`-rule, the goal splits into the following two subgoals

`$S A |- e1 :: ?ta` (6)
`(gen ($S A) ?ta)#($S A) |- e2 :: $S t` (7)

where `?ta` is an unknown to be instantiated arbitrarily. For substitutions `S` which are compatible with `gen` the subgoals trivially hold, choosing `?ta` to be `$S t1`. Unfortunately the substitution `S` need not meet the precondition of lemma (5) (see section 4.2). To circumvent this problem, we replace the substitution `S` in (6) and (7) by the substitution `S' = (`λ`x. if x` $\in$ `free_tv A` $\cup$ `free_tv t then S x else (TVar x))`, which is equivalent in this context. Furthermore, we choose for the unknown `?ta` the type `$(S' o `α`) t1` where α is defined as follows: α = λ`x. if x` $\in$ `free_tv A then x else x + n` with `n` new for `A`, `t`, `$S A` and `$S t`. The existence of such a variable `n` is guaranteed by the lemma

$\exists$`n. new_tv n A` $\wedge$ `new_tv n t` $\wedge$ `new_tv n ($S A)` $\wedge$ `new_tv n ($S t)`

Note that (after restating) the first subgoal is trivial because by definition of α we have `$S A = $(S o `α`) A`. We now can apply the lemma (5) to subgoal (7) since `free_tv S'` $\leq$ `free_tv A` $\cup$ `free_tv t` $\cup$ `free_tv $S A` $\cup$ `free_tv $S t` and `free_tv ($(TVar o `α`) t1) - free_tv A` $\leq$ `{x.` $\exists$`y. x = n + y}`. Subgoal (7) then changes to `$S' (gen A ($(TVar o `α`) t1))#A |- e2 :: $S' t` which again is not immediate. To complete the proof, we use the relation $\leq$ which we explained in Section 4.1. The remaining proof relies on some (non-trivial) lemmas:

$[\![$ `A |- e :: t; A` $\leq$ `B` $]\!]$ $\Longrightarrow$ `B |- e :: t` (8)
`sch1` $\leq$ `sch2` $\Longrightarrow$ `$S sch1` $\leq$ `$S sch2`
`new n A` $\Longrightarrow$ `gen A t` $\leq$ `gen A ($(TVar o `α`) t)`

An alternative formulation of lemma (5) avoids the introduction of the relation $\leq$ at the prize of having to deal with α-conversion explicitely. This includes proving that `has_type` is invariant under α-conversion. Compared to this solution, the presented solution is more elegant and shorter; in particular since we have to deal with $\leq$ in the completeness proof anyway.

5 Algorithm $\mathcal{W}$

The aim of the algorithm $\mathcal{W}$ is to compute the most general type of a given λ-term. If $\mathcal{W}$ succeeds, it returns a substitution `S` together with a type `t` such that `$S A |- e :: t`. At certain points $\mathcal{W}$ requires fresh type variables. As already explained in Section 3.3, we handle this problem by passing a counter, for which we require an additional result component. Altogether, $\mathcal{W}$ has the following type:

```
consts  W :: expr ⇒ ctxt ⇒ nat ⇒ (subst * typ * nat)option
```

$\mathcal{W}$ is recursively defined on the term structure.

```
primrec W expr
  W (Var i) A n =
      (if i < length A then
          Some(id_subst,
               bound_typ_inst (λb. TVar(b + n)) (nth i A),
               n + (min_new_bound_tv (nth i A)))
          else None)

  W (Abs e) A n = ((S,t,m) := W e ((FVar n)#A) (Suc n);
                   Some(S, (S n) → t, m))

  W (App e1 e2) A n = ((S1,t1,m1) := W e1 A n;
                       (S2,t2,m2) := W e2 ($S1 A) m1;
                       U := mgu ($S2 t1) (t2 → (TVar m2));
                       Some($U o $S2 o S1, U m2, Suc m2))

  W (LET e1 e2) A n =
      ((S1,t1,m1) := W e1 A n;
       (S2,t2,m2) := W e2 ((gen ($S1 A) t1)#($S1 A)) m1;
       Some($S2 o S1, t2, m2))
```

A call `W e A n` fails if `A` contains no entry for some free variable in `e`, or if a call of the unification algorithm `mgu` fails. The failure propagation is invisibly handled by the monadic `option_bind`

```
constdefs
  option_bind :: (α option) ⇒ (α ⇒ (β option)) ⇒ β option
  option_bind m f ≡ case m of None ⇒ None ∨ Some r ⇒ f r
```

for which Isabelle permits the notation `P := E; F` instead of `option_bind E (λP.F)`. Compared with the algorithm in [14], we do not only have to deal with the `let`-case, but also with a slightly modified `var`-case: In the polymorphic algorithm a context comprises type schemes rather than types. To assign a variable `n` its corresponding "type" we have to convert the type scheme `nth n A` into a type by turning bound variables into fresh type variables. This is achieved by the function `bound_typ_inst :: subst ⇒ type_scheme ⇒ typ`. Accordingly, we increase the counter by the smallest bound variable which is new for `nth n A`. The function `min_new_bound_tv :: type_scheme ⇒ nat` computes this offset. The definition of the `let`-case is a straightforward implementation of the algorithm given in [5].

So far we have ignored the fact that the function `gen` tests whether a variable is an element of a set of variables, which entails the function `gen` to be non-computable. We have chosen this definition of the function `gen` because in the soundness and completeness proofs it is easier to deal with sets of variables than with lists of variables. To ensure the computability of algorithm $\mathcal{W}$ we show that `gen` is equivalent to a computable function `gen_ML`:

```
gen A t = gen_ML A t
```

The function `gen_ML` is implemented straightforwardly, relying on an auxiliary function `aux`:

```
consts aux :: nat list ⇒ typ ⇒ type_scheme
primrec aux typ
  aux A (TVar n) = (if (n mem A) then (FVar n) else (BVar n))
  aux A (t1 → t2) = ((aux A t1) =→ (aux A t2))

consts gen_ML :: ctxt ⇒ typ ⇒ type_scheme
defs gen_ML A t ≡ aux (free_tv_ML A) t
```

The function `free_tv_ML :: type_scheme ⇒ nat list` in this definition is a computable counterpart of the function `free_tv`.

Having introduced the basic definitions and lemmas, we now turn to the main goal of this case study, namely the proof of correctness of algorithm $\mathcal{W}$. Correctness is defined as soundness and completeness w.r.t. the type inference rules.

5.1 Soundness of $\mathcal{W}$

In contrast to the monomorphic case, we use in the soundness of $\mathcal{W}$ already the same lemmas we need for completeness – which was quite surprising to us. Furthermore we have shown soundness only for new counters. This is no real limitation since we can always compute such a new counter (see Section 3.3). We suspect that proving soundness without this auxiliary condition could be tricky. Soundness is stated as follows:

```
⟦ new_tv n A; W e A n = Some (S,t,m) ⟧ ⟹ $S A |- e :: t
```

This proposition is shown by induction on the structure of `e` using some auxiliary lemmas, all of which deal with the problem of new type variables. In mathematical proofs about type inference algorithms (including Damas' own) the problem of new type variables is simply ignored. The first lemma states that the counter for new type variables is never decreased:

$\texttt{W e A n = Some (S,t,m)} \Longrightarrow \texttt{n} \leq \texttt{m}$

It is proved by induction on the structure of `e` and in turn helps us to prove the following lemma:

$$[\![\ \texttt{new\_tv n A; W e A n = Some (S,t,m)}\ ]\!] \Longrightarrow \texttt{new\_tv m A} \wedge \texttt{new\_tv m S} \wedge \texttt{new\_tv m t} \quad (9)$$

It expresses that the resulting type variable is new w.r.t. the context, the computed substitution and the computed type term. This fact ensures that we can safely use the returned type variable as new type variable in subsequent computations. Again, we use induction on the structure of `e`. In the `let`-case many explicit instantiations are required whereas further automation seems unlikely.

Now we can prove a proposition that seems to be quite obvious. Roughly speaking, it says that type variables free in either the computed substitution or the computed type term did not materialize out of the blue: they either occur in the given context or were taken from the set of new type variables. Formally, the proposition is expressed as follows:

$$[\![\ \texttt{W e A n = Some (S,t,m); v} \in \texttt{free\_tv S} \vee \texttt{v} \in \texttt{free\_tv t; v} < \texttt{n}\ ]\!] \Longrightarrow \texttt{v} \in \texttt{free\_tv A}$$

The proof is performed by induction on the structure of term `e`. The complexity of this proof is roughly the same as the complexity of the last proof.

5.2 Completeness of $\mathcal{W}$

With the help of these propositions we are able to show completeness of $\mathcal{W}$ w.r.t. the type inference rules: if a closed term `e` has type `t'` then $\mathcal{W}$ terminates successfully and returns a type which is more general than `t'`, i.e. `t'` is an instance of that type.

$$\texttt{[] |- e :: t'} \Longrightarrow \exists \texttt{S t. (}\exists\texttt{m. W e [] n = Some(S,t,m))} \wedge \texttt{(}\exists\texttt{R. t' = \$R t))}$$

This proposition needs to be generalized considerably before it is amenable to induction:

$$[\![\ \texttt{\$S' A |- e :: t'; new\_tv n A}\ ]\!] \Longrightarrow \exists\texttt{S t. (}\exists\texttt{m. W e A n = Some (S,t,m))} \wedge \texttt{(}\exists\texttt{R. \$S' A = \$R (\$S A)} \wedge \texttt{t' = \$R t))}$$

The proofs for abstraction and application are quite analogous to the ones in [14]. Proving the `var`-case is more tedious but not really difficult. To show the `let`-case we used the lemmas (4), (8) and (9). Surprisingly, the `let`-case is much more straightforward than application.

6 Conclusion

This verification is more than just another case study. Potentially it is the basis for

1. verified variations of $\mathcal{W}$, of which there exist many. For example, Nipkow and Prehofer [16] give a very detailed but (as we now noticed) still incomplete informal proof of a type inference algorithm for Haskell's type classes.
2. a verified Mini-ML compiler. We are aware of two formal proofs of the back end of a Mini-ML compiler [9, 1], unfortunately conducted in different systems (Elf and Coq). The time is ripe to combine these individual efforts.

How does our proof compare to Damas'?

1. Despite the fact that we consulted his proof only in the few cases where we were stuck, we ended up with an overall proof structure which is remarkably close to his. The same seems to be true for the proof by Dubois and Ménissier-Morain [7]. The latter, however, is harder to judge because they present only a selection of their definitions and their development was not complete at the time.
2. We did not compare the two proofs line by line, but in the cases where we did check, Damas' proof is remarkably detailed and free of flaws. This is certainly a tribute to Damas' thoroughness (which, unfortunately, seems to have failed him in his work on reference types [20, p. 2]).
3. The major complication, which we already encountered in the monomorphic case, is the treatment of new variables, which Damas, as most other authors, simply takes for granted. In the presence of `let` the situation is worse than before, because we now need to think very carefully about new variables already for the soundness proof. Hence our work is more than a mere formalization of Damas' proof.

Concerning the theorem proving process itself, we found that it rarely was the bottleneck. Finding the right formalization of our informal ideas and finding the right lemmas was much more crucial and time consuming than convincing Isabelle that we were right.

Finally, there are two interesting technical points. We have shown that the dogmas of idempotence and finiteness of substitutions can be relaxed quite easily:

1. instead of idempotence, a slightly weaker property (1) of mgus suffices. For details see the discussion in [14].
2. instead of finite functions, substitutions may be arbitrary total functions.

Acknowledgements We sincerely thank Dieter Nazareth for his collaboration in the earlier verification which this paper relies and builds on.

References

1. S. Boutin. Preuve de correction de la compilation de mini-ml en code cam dans le système d'aide à la démonstration coq. Technical Report 2536, INRIA, Apr. 1995.
2. R. Burstall, D. MacQueen, and D. Sannella. Hope: an experimental applicative language. In *Proc. 1980 LISP Conference*, pages 136–143, 1980.
3. 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.
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. L. M. M. Damas. *Type Assignment in Programming Languages.* PhD thesis, Department of Computer Science, University of Edinburgh, 1985.
6. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.
7. C. Dubois and V. Ménissier-Morain. A proved type inference tool for ML: Damas-Milner within Coq (work in progress). In J. von Wright, J. Grundy, and J. Harrison, editors, *Supplementary Proc. 9th Int. Conf. Theorem Proving in Higher Order Logics*, pages 15–30. Turku Centre for Comp. Sci., 1996.
8. M. Gordon and T. Melham. *Introduction to HOL: a theorem-proving environment for higher order logic.* Cambridge University Press, 1993.
9. J. Hannan and F. Pfenning. Compiler verification in lf. In *7th IEEE Symp. Logic in Computer Science*, pages 407–418. IEEE Computer Society Press, 1992.
10. J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, 1969.
11. P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.
12. R. Milner. A theory of type polymorphism in programming. *J. Comp. Sys. Sci.*, 17:348–375, 1978.
13. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
14. D. Nazareth and T. Nipkow. Formal verification of algorithm W: The monomorphic case. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lect. Notes in Comp. Sci.*, pages 331–346. Springer-Verlag, 1996.
15. T. Nipkow. Functional unification of higher-order patterns. In *8th IEEE Symp. Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, 1993.
16. T. Nipkow and C. Prehofer. Type reconstruction for type classes. *J. Functional Programming*, 5(2):201–224, 1995.
17. L. C. Paulson. Verifying the unification algorithm in LCF. *Sci. Comp. Programming*, 5:143–169, 1985.
18. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.
19. L. C. Paulson. Generic automatic proof tools. Technical Report 396, University of Cambridge, Computer Laboratory, 1996.
20. M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.