Formal Verification of Algorithm $\mathcal{W}$: The Monomorphic Case

Dieter Nazareth* and Tobias Nipkow

Technische Universität München**

Abstract. A formal verification of the soundness and completeness of Milner's type inference algorithm $\mathcal{W}$ for simply typed lambda-terms is presented. Particular attention is paid to the notorious issue of "new" variables. The proofs are carried out in Isabelle/HOL, the HOL instantiation of the generic theorem prover Isabelle.

1 Introduction

Type systems for programming languages are usually defined by type inference rules which inductively define the set of well-typed programs. Functional languages of the ML-tradition also come with a type inference algorithm which computes the (most general) type of a program. The inference algorithm needs to be sound and complete w.r.t. the rules: the rules and the algorithm must determine the same set of type correct programs.

The idea of type inference goes back to Hindley [7]. Milner [14] gave the first account of type inference for a simply-typed lambda-calculus with `let`, the core of ML [15]. In particular, he presented a type inference algorithm $\mathcal{W}$ based on unification of types. Although Milner's original article only proves soundness of $\mathcal{W}$ w.r.t. the rules, completeness has been settled in the mean time [4, 25]. This polymorphic type system forms the basis of most modern functional languages, usually in some extended or generalized form. Each extension of the type system requires a corresponding modification of algorithm $\mathcal{W}$, which again has to be proved sound and complete, e.g. [9, 17, 16, 22]. However, all these proofs have been carried out in a mathematical or informal way. We are not aware of any completely machine checked proof.

This paper presents a formalization of the original type system restricted to the monomorphic case. The soundness as well as the completeness property of the type inference algorithm $\mathcal{W}$ is formally established within the Isabelle/HOL system. The paper provides the complete definition of all concepts, the key lemmas and theorems, but no proofs. The complete development is accessible via `http://www4.informatik.tu-muenchen.de/~nipkow/isabelle/HOL/MiniML/`.

The rest of the paper is organized as follows: after a brief introduction to Isabelle/HOL in Section 2, Section 3 deals with the formalization of the type level.

* Research supported by ESPRIT BRA 6453, *Types*.

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

It includes theories of substitution and unification and the treatment of "new" type variables. In Section 4 the type inference system is presented. Algorithm $\mathcal{W}$ is formalized in Section 5: we briefly show how to model exception handling in a functional style using monads and discuss the soundness and completeness proofs of $\mathcal{W}$. In Section 6 we compare our case study to related work. Section 7 concludes with some lessons learned and some future work.

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 [6]. In the remainder of the paper HOL is short for Isabelle/HOL.

We present no proofs but merely definitions and theorems. Hence it suffices to introduce HOL's surface syntax. A detailed introduction to Isabelle and HOL can be found elsewhere [18]. We have intentionally refrained from recasting HOL's ASCII syntax in ordinary mathematical symbols to give the reader a bit of an idea what interacting with HOL looks like.

Terms and formulae The following table summarizes the correspondence between ASCII and mathematical symbols:

`!`, `!!`	`?`	`%`	`-->`, `==>`	`&`	`|`	`=`, `==`	`~=`	`:`	`Un`	`UN`	`<=`
$\forall$	$\exists$	λ	$\Longrightarrow$	$\wedge$	$\vee$	$=$	$\neq$	$\in$	$\cup$	$\bigcup$	$\leq$, $\subseteq$

The two universal quantifiers, implications and equalities stem from the object and meta-logic, respectively. The distinction can be ignored while reading this paper. The notation `[|` $A_1; \ldots; A_n$ `|] ==>` A is short for the nested implication $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A$. The predicate `<=` is overloaded and applies to natural numbers ($\leq$) and to sets ($\subseteq$).

Types follow the syntax for ML-types, except that the function arrow is `=>` rather than `->`. The notation `[`τ_1`, ...,`τ_n`] => `τ abbreviates τ_1 `=> ... =>` τ_n `=>` τ. A term t is constrained to be of type τ by writing t`::`τ.
Isabelle also provides Haskell-like *type classes* [8], the details of which are explained as we go along. A type variable `'a` is restricted to be of class `c` by writing `'a::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 and Substitutions

This section describes the language of object-level types used in our case study. They should not be confused with Isabelle's built-in meta-level type system described in the previous section. To emphasize this distinction we sometimes call the object-level types *type terms*.

3.1 Types

Type terms consist only of type variables and the function space constructor. They are expressed as an inductive data type with two constructors. Type variables are modeled by natural numbers.

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

We do not need quantified types because our term language does not contain `let`-expressions.

3.2 Substitution

A substitution is a function mapping type variables to types. In HOL, all functions are total. The identity substitution is denoted by `id_subst`.

```
types   subst = nat => typ

consts  id_subst :: subst

defs    id_subst == (%n.TVar n)
```

Substitutions can be extended to type terms, lists of type terms, etc. Type classes, i.e. overloading, allow us to use the same notation in all of these cases.

```
classes  type_struct < term
```

introduces a new class `type_struct` as a subclass of `term`, the predefined class of all HOL types. Class `type_struct` is meant to encompass all meta-level types which substitutions can be applied to. This is expressed by declaring

```
consts  app_subst :: [subst, 'a::type_struct] => 'a ("$")
```

The purpose of `app_subst` is to apply substitutions to values of types in class `type_struct`. `$` is syntactic sugar for `app_subst`. Because identifiers in Isabelle do not contain "$" we may write `$s` instead of `$ s`. The notation `$s` emphasizes that we regard `$` as a modifier acting on the substitution `s`.

So far there is no definition of `app_subst`, but there will be several, one for each instance of `'a` we are interested in. Hence `app_subst` will be overloaded. In Haskell, `app_subst` is a *member function* of class `type_struct`.

Now we want to turn `typ` into an element of class `type_struct` by extending substitutions from type variables to types in the usual fashion. This requires two steps. First we simply tell Isabelle that `typ` is an element of `type_struct`:

```
arities  typ :: type_struct
```

The general form of the `arities` declaration is $t :: (C_1, \ldots, C_n)C$, where t must be an n-ary type constructor. It expresses that $(\tau_1, \ldots, \tau_n)t$ is of class C provided each τ_i is of class C_i.

Then we define the appropriate instance of `app_subst` by primitive recursion over `typ`:

```
primrec app_subst typ
        $ s (TVar n) = s n
        $ s (t1 -> t2) = ($ s t1) -> ($ s t2)
```

In Haskell, both steps are combined into the `instance` construct.

In the same way we extend `app_subst` to lists:

```
arities  list :: (type_struct)type_struct
```

Hence (τ)`list` is of class `type_struct` provided the element type τ is. A substitution is applied to a list by mapping it over that list, where `map` is predefined:

```
defs  $ s == map ($ s)
```

Note that `$ s` on the left has type `'a list => 'a list` and on the right type `'a => 'a`, where `'a::type_struct`.

In the sequel, a *type structure* is a type in class `type_struct`.

Now we can prove that the extension of the identity substitution to type terms and list of type terms again yields identity functions:

```
$ id_subst = (%t::typ.t)
$ id_subst = (%ts::typ list.ts)
```

For the composition of substitutions the following propositions hold:

```
$ g ($ f t::typ) = $ (%x. $ g (f x) ) t
$ g ($ f ts::typ list) = $ (%x. $ g (f x)) ts
```

3.3 Free Type Variables

The set of type variables occurring in a type structure is denoted by `free_tv`. Again, we overload this function by using class `type_struct`. The definitions below describe the usual behaviour of the `typ` and `list` instances, respectively:

```
consts  free_tv :: 'a::type_struct => nat set

primrec free_tv typ
        free_tv (TVar m) = {m}
        free_tv (t1 -> t2) = (free_tv t1) Un (free_tv t2)

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

Note that infix `# :: ['a, 'a list] => 'a list` is the list constructor adding an element to the front of a list.

These definitions enable us to show some interesting properties:

```
[| $ s1 (t::typ) = $ s2 t; n : free_tv t |] ==> s1 n = s2 n
(!n. n: free_tv t --> s1 n = s2 n) ==> $ s1 (t::typ) = $ s2 t
(t::typ) mem ts ==> free_tv t <= free_tv ts
```

The first one states that if applying two different substitutions to the same type term yields the same result, then the substitutions coincide on the free type variables occurring in the type term. The second one reverses this implication. The third one states that if a type term is an element (infix `mem`) of some list, then the set of free type variables of the type term is a subset of the set of free type variables of the list. The first two propositions have also been proved for lists of type terms.

Domain and codomain of a substitution are defined in the usual way:

```
consts dom, cod :: subst => nat set

defs   dom s == {n. s n ~= TVar n}
       cod s == UN m:dom s. free_tv (s m)
```

The set of variables occurring either in the domain or the codomain of a substitution is called the set of free variables of a substitution. We want to use the identifier `free_tv` to denote this set. Hence, we must add type `subst` to class `type_struct`. Type `subst`, however, is only an abbreviation for the composed type `nat => typ`. Just like Haskell, Isabelle does not allow to add composed types to type classes directly. Instead, we must state the propagation of class membership for the type constructor `=>`. We already met this mechanism when adding the list types to class `type_struct`. In the same way we define an arity for the type constructor `=>` which is the infix name for `fun`:

```
arities  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 => 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) Un (cod s)
```

We do not define other possible instances of `free_tv`. Neither do we provide an instance for function `app_subst` at type `subst`.

The following lemmas capture important relationships between substitution application and free type variables:

```
free_tv ($ s (t::typ)) <= cod s Un free_tv t
[| v : free_tv(s n); v ~= n |] ==> v : cod s
free_tv (%n::nat. $ s1 (s2 n) :: typ) <= free_tv s1 Un free_tv s2
```

We need these propositions in the soundness and completeness proofs of the type inference algorithm.

3.4 New Type Variables

Algorithm $\mathcal{W}$ needs to generate new type variables. This mechanism is rarely formalized in the description of the algorithm. It is simply assumed that there always exists some type variable never used before. However, to perform formal proofs we have to completely formalize the algorithm. The obvious way of handling the generation of new type variables is as follows: the set of already used type variables is explicitly passed to $\mathcal{W}$, all new variables generated during the execution are added to this set, and the enlarged set is returned upon successful termination [17]. Because we use natural numbers for type variables we have a total ordering on these variables. Instead of passing all used type variables to $\mathcal{W}$, we only pass the successor of the greatest one used up to now. Each time algorithm $\mathcal{W}$ needs a new type variable it uses the counter and increments it by one. Predicate `new_tv` formalizes our 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,'a::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 $\mathcal{W}$. To prove these propositions we need some theorems about `new_tv`. The first one is quite simple: it states that all greater type variables are also new type variables. This holds not only for type terms, but also for lists of type terms and for substitutions.

```
[| n<=m; new_tv n (t::typ) |] ==> new_tv m t
```

This proposition is proved by induction on the structure of `t`. The next theorems show how substitutions and `new_tv` interact:

```
[| new_tv n s; new_tv n (t::typ) |] ==> new_tv n ($ s t)
[| new_tv n (s::subst); new_tv n r |] ==> new_tv n (($ r) o s)
new_tv n s = ((!m. n <= m --> s m = TVar m) &
              (! l. l < n --> new_tv n (s l) ))
```

The first two theorems tell us that the new type variable property is preserved by application and composition (`o`) of substitutions. The third one is more complex to express and prove: it requires about 15 proof steps.

3.5 Unification

The goal of unification is to unify two terms, i.e. to find a substitution of terms for variables which makes the two terms syntactically identical. A *unification algorithm* either computes a most general unifier of two terms or fails, if the two terms are not unifiable. The first in a long line of unification algorithms is due to Robinson [21]. 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 define

```
datatype  'a maybe =  Ok 'a | Fail
```

Unification either terminates normally returning `Ok(s)` for some substitution `s`, or indicates a failure situation by returning `Fail`:

```
consts  mgu :: [typ,typ] => subst maybe
```

A most general unifier should satisfy the following three axioms:

```
rules   mgu t1 t2 = Ok u ==> $u t1 = $u t2
        [| mgu t1 t2 = Ok u; $s t1 = $s t2 |] ==> ? r. s = $r o u
        $s t1 = $s t2 ==> ? u. mgu t1 t2 = Ok u
```

The first axiom requires the result of `mgu` to be a unifier of the given type terms. The second one states that the computed unifier is a most general one: each unifier can be obtained by composing the computed one with some substitution. The third one requires `mgu` to return an `Ok` result if the two types are unifiable. This prevents trivial implementations which satisfy the first two axioms by always returning `Fail`.

Most general unifiers are only unique up to consistent renaming of variables. Such a renaming may even introduce type variables not occurring in the type terms to unify. However, because we want to keep track of used variables we need one last axiom:

```
mgu t1 t2 = Ok u ==> free_tv u <= free_tv t1 Un free_tv t2          (0)
```

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

```
[| mgu t1 t2 = Ok u; new_tv n t1; new_tv n t2 |] ==> new_tv n u
```

4 Well-Typed Lambda Terms

Lambda terms are represented in de Bruijn notation [5] which can conveniently be expressed as an inductive data type with three constructors for variables, abstraction and application:

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

The index i in a subterm `Var` i indicates that, when moving upward, i abstractions must be traversed until the corresponding binder is found. For example, $\lambda x.x(\lambda y.y\ x)$ becomes `Abs(App(Var 0)(Abs(App(Var 0)(Var 1))))`. This shows that

- different occurrences of the same index may represent different bound variables, and
- different occurrences of the same bound variable may be represented by different indexes, depending on how far below the binding λ they are.

Free variables are those without enough enclosing applications, e.g. `Var 0` on its own.

Note that we have restricted our language to pure lambda terms without `let`-construct. Because `let`-bound identifiers are the only source of polymorphism in ML-like languages we do not need quantified type terms.

The datatype `expr` represents all untyped lambda-terms. Now we need to define the subset of *well-typed* lambda-terms. To keep track of the types of bound variables we use a context assigning type terms to variables. Because we use de Bruijn notation this context is simply a list of type terms. The type of variable i can be found at the i-th list position. Well-typedness is a relative notion because it depends on the context. Therefore we introduce a relation between contexts, lambda-terms and type terms:

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

The proposition `(a,e,t):has_type` should be read as "In context `a` expression `e` has type `t`". This allows a term to have more than one type in a given context. The following bit of syntactic sugar (which we will not explain in detail) allows us to read and write the more conventional `a |- e :: t`:

```
syntax "@has_type":: [typ list,expr,typ] => bool  ("_ |- _ :: _" 60)
translations  a |- e :: t == (a,e,t) : has_type
```

Warning: the delimiter `::` is now used for type annotations both in the logic and in the object level lambda-terms (`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. The following text defines `has_type` to be the least set closed under the given inference rules.

```
inductive has_type
    [| n < length a |] ==> a |- Var n :: nth n a
    [| t1#a |- e :: t2 |] ==> a |- Abs e :: t1 -> t2
    [| a |- e1 :: t2 -> t1; a |- e2 :: t2 |] ==> a |- App e1 e2 :: t1
```

Note that `nth :: [nat, 'a list] => 'a` selects the nth element of a list.

Modulo the fact that we use de Bruijn notation, these are the usual type inference rules for lambda terms. The reader not familiar with this type system is referred to [2]. Note that the simplicity of the `Abs`-rule is due to de Bruijn

notation: the extended context `t1#a` takes care of the fact that when descending into an abstraction all references to variables bound outside shift by 1.

The following theorem shows that `has_type` is closed w.r.t. substitution:

```
a |- e :: t ==> $ s a |- e :: $ s t
```

It is proved by induction on the derivation of `a |- e :: t`. This leads to three subgoals (corresponding to the three inference rules given above), each of which is proved almost automatically.

5 Type Inference

The purpose of *type inference* (or *type reconstruction*) is to find the most general type `t` for a given term `e` in a given context `a` such that `a |- e :: t`. Interpreting the type inference rules as a Prolog program yields such a type inference algorithm. Using a functional implementation language, the computation of a most general type requires a separate algorithm which was first presented by Milner [14] who called the algorithm $\mathcal{W}$ ($\mathcal{W}$ell-typing).

5.1 Programming with Monads

Given a context `a` and a term `e`, there may be no type `t` such that `a |- e :: t`. In this case $\mathcal{W}$ should "fail" with some meaningful error message. The easiest way of handling error messages is to use so-called *impure* features, like side effects or exceptions. Wadler [23] introduced the idea that *monads* could be used as a practical method for modeling such impure features in a purely functional way. In this paper we only give a brief introduction into programming with monads. A good presentation of this topic can be found in [24].

A *monad* is a data type consisting of a type constructor `M` and two operations `unit` and `bind` with the following functionalities:

```
unit :: 'a => 'a M
bind :: ['a M, 'a => 'b M] => 'b M
```

A value of type `'a M` represents a computation which is expected to produce a result of type `'a`. Function `unit` turns a value into the computation that returns that value and does nothing else. Function `bind` provides a means of combining computations. It applies a function of type `'a => 'b M` to a computation of type `'a M`.

We already met a monad in this paper, in connection with unification, where failure is also an issue. Together with appropriate `unit` and `bind` functions, type constructor `maybe` forms a monad. Values of type `'a maybe` represent computations that may raise an exception. Constructor `Ok` denotes the unit function. It turns a value `a` into `Ok a`. Because we only gave an abstract requirement specification for `mgu` we did not explain how to propagate failures. This is the task

of the `bind` function defined in the following way:

```
consts  bind :: ['a maybe, 'a => 'b maybe] => 'b maybe (infixl 60)

defs    m bind f == case m of Ok r => f r | Fail => Fail
```

The call `m bind f` examines the result of the computation `m`: if it is a failure, it is propagated; otherwise the function `f` is applied to the value of the computation. In most cases `f` is expressed as a λ-term:

```
m bind (%x.c)
```

This can be read as follows: perform computation `m`, bind the resulting value to variable `x`, and then perform computation `c`. In an imperative language this would be expressed in the following way:

```
x := m; c
```

The powerful translation mechanism of Isabelle allows us to use exactly this notation for the `bind` function:

```
syntax "@bind" :: [pttrns,'a maybe,'b maybe] => 'b ("_ := _; _" 0)
translations P := E; F  ==  E bind (%P.F)
```

Now we can define algorithm $\mathcal{W}$ in an imperative style without using any impure feature.

Warning: do not confuse the monad ";" with the separator for hypotheses.

5.2 Algorithm $\mathcal{W}$

If $\mathcal{W}$ succeeds, it returns a substitution `s` and a type `t` such that `$s a |- e::t`. At certain points $\mathcal{W}$ requires new type variables. As already explained in Section 3.4, we handle the generation of new type variables by passing the successor of the greatest type variable used up to now. Thus, we need an additional result component for this counter. Altogether, $\mathcal{W}$ has the following type:

```
consts  W :: [expr, typ list, nat] => (subst * typ * nat)maybe
```

$\mathcal{W}$ is recursively defined on the term structure. Because the type inference rules given in Section 4 are syntax-directed, each case of function $\mathcal{W}$ corresponds to exactly one rule.

```
primrec W expr
  W (Var i) a n = (if i < length a then Ok(id_subst, nth i a, n)
                                   else Fail)
  W (Abs e) a n = ( (s,t,m) := W e ((TVar n)#a) (Suc n);
                    Ok(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));
                        Ok($u o $s2 o s1, $u (TVar m2), Suc 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 our mixfix `bind` notation.

The main goal of this case study was to formally show the correctness of algorithm $\mathcal{W}$. Correctness is defined as soundness and completeness w.r.t. the type inference rules. We start with soundness of $\mathcal{W}$:

```
W e a n = Ok(s,t,m) ==> $s a |- e :: t
```

This proposition is shown by induction on the structure of `e`. The proof can be performed directly, without any additional auxiliary proposition. However, both the `Abs` and `App` case require explicit instantiation of variables. Therefore the user needs to be familiar with the details of the proof. An automatic proof which synthesizes these instantiations seems unlikely.

5.3 Completeness of $\mathcal{W}$

The proof of completeness of $\mathcal{W}$ w.r.t. the type inference rules is more complex. We have to prove some auxiliary lemmas first. All these lemmas deal with the problem of new type variables. In mathematical proofs about type inference algorithms this problem is simply ignored. The first lemma states that the counter for new type variables is never decreased:

```
W e a n  = Ok (s,t,m) ==> n<=m
```

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

```
[| new_tv n a; W e a n = Ok (s,t,m) |] ==> new_tv m s & new_tv m t
```

It says that the resulting type variable is new w.r.t. the computed substitution as well as 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`. The most difficult part of the proof is case `App e1 e2`. It takes about 30 proof steps; the degree of automation is quite low.

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:

```
[| W e a n = Ok (s,t,m); v : free_tv s | v : free_tv t; v<n |]
==> v : 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.

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 sucessfully and returns a type which is more general than `t'`, i.e. `t'` is an instance of that type.

```
[] |- e :: t' ==> ? s t. (? m. W e [] n = Ok(s,t,m)) &
                         (? r. t' = $r t))
```

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

```
[| $s' a |- e :: t'; new_tv n a |]
==> ? s t. (? m. W e a n = Ok (s,t,m)) &
           (? r. $s' a = $r ($s a) & t' = $r t))
```

This theorem is the most difficult one to prove. Although the proof plan in [16] is quite detailed, translating it into Isabelle turned out to be hard work. The proof starts with an induction on the structure of term `e`. Again, case `App e1 e2` causes most of the problems. Proving this case requires about 90 proof steps. Isabelle is used mainly to keep track of the proof and to avoid foolish mistakes.

Let us have a brief look at the `App` case: the main problem is to show successful termination. The algorithm may fail during unification of `$s2 t1` and `t2 -> (TVar m2)`. Hence, we have to prove that these terms are indeed unifiable, i.e. that there exists a substitution `u` such that

```
$u ($s2 t1) = $u (t2 -> (TVar m2)).
```

In our proof we use the witness `u` given in [16], which differs slightly from the one used in much of the published literature (for example [9, 17]). We establish that this witness is indeed a unifier for the above type terms.

5.4 Algorithm $\mathcal{I}$

Milner [14] also presents a more efficient refinement of algorithm $\mathcal{W}$ called $\mathcal{I}$, where substitutions are extended incrementally instead of computing new substitutions and composing them later. We have also formalized $\mathcal{I}$ [1]

```
consts  I :: [expr, typ list, nat, subst] => (subst * typ * nat)maybe

primrec I expr
  I (Var i) a n s = (if i < length a then Ok(s,nth i a,n) else Fail)
  I (Abs e) a n s = ( (s,t,m) := I e ((TVar n)#a) (Suc n) s;
                      Ok(s, TVar n -> t, m) )
  I (App e1 e2) a n s =
                    ( (s1,t1,m1) := I e1 a n s;
                      (s2,t2,m2) := I e2 a m1 s1;
                      u := mgu ($s2 t1) ($s2 t2 -> TVar m2);
                      Ok($u o s2, TVar m2, Suc m2) )
```

and shown that it correctly implements $\mathcal{W}$:

```
[| new_tv m a;  new_tv m s;  I e a m s = Ok(s',t,n) |]
==>  ? r. W e ($s a) m = Ok(r, $s' t, n) & s' = ($r o s)

[| new_tv m a;  new_tv m s;  I e a m s = Fail |]
==>  W e ($s a) m = Fail
```

For lack of space, the details cannot be presented here.

[1] Ideally, `mgu` should take the substitution `s2` as a separate argument. For simplicity we have applied `s2` explicitly to the two type arguments of `mgu`.

6 Comparison

The literature contains many accounts of type systems and type inference algorithms for lambda-terms which boil down to the rules and algorithm we used [14, 3, 4, 2, 25, 1]. However, ours seems to be the first formal verification of $\mathcal{W}$. There are three key differences between the existing literature and our formal proof:

1. We do not treat `let`. This is a regrettable omission but is justified by the complexity of the formal proof for the `let`-free system.
2. We treat the issue of "new" variables, which is almost universally ignored (an exception is [17]). Ironically, it is this very issue which really complicates the proof for us.
3. In the literature, completeness of $\mathcal{W}$ is always proved under the assumption that the unification algorithm returns idempotent substitutions. In contrast, we require axiom (0) (see the end of Section 3.5), which turns out to be weaker than idempotence.

We would like to concentrate on the last point for a moment, employing the usual mathematical notation from unification theory [10].

A substitution σ is *idempotent* if $\sigma \circ \sigma = \sigma$. Axiom (0) requires $V(\sigma) \subseteq V(s,t)$ for the `mgu` σ of two terms s and t, where $V(\sigma) = dom(\sigma) \cup cod(\sigma)$ and $V(s,t)$ is the set of variables in s and t.

Theorem 1. *Let σ be a most general unifier of s and t. If σ is idempotent, then $V(\sigma) \subseteq V(s,t)$.*

Proof. It is known that $V(\sigma_1) = V(\sigma_2)$ holds for any two idempotent mgus of $s = t$ [11, Prop. 4.11]. If s and t are unifiable, there exists an idempotent mgu σ_0 such that $V(\sigma_0) \subseteq V(s,t)$ (use any of the standard unification algorithms). Thus the claim follows for every idempotent mgu σ.

Idempotence is strictly stronger than the variable condition as the following example [11] shows: $\sigma = \{x \mapsto f(x), y \mapsto x\}$ is a most general unifier of $x = f(y)$ which satisfies the variable condition but is not idempotent.

Thus we have verified $\mathcal{W}$ under weaker assumptions than usual. However, this is merely a theoretical curiosity: practical unification algorithms do return idempotent substitutions. In fact, it is hard to imagine a unification algorithm computing σ in the example above.

The only other published formal verification of a type checking algorithm we are aware of is by Pollack [20]. His object-language (a subset of "Pure Type Systems") is much more powerful than ours but his terms already contain types. Hence he does not need a separate theory of substitutions and unification to support type inference. On the other hand his proofs involve a substantial amount of lambda-calculus theory, and he also faces the issue of new (term) variables [13]. A final difference is that he does not verify an explicit algorithm but performs a constructive proof which embodies the algorithm.

7 Conclusion and Future Work

The results of this case study can be summarized as follows:

Specification: Isabelle/HOL offers a mature specification environment with a flexible syntax (mixfix) and type system (classes).

Proof: Isabelle/HOL provides some automation on the predicate calculus level, but not nearly enough for our case study. Especially reasoning by transitivity and monotonicity needs to be improved. Decidable subtheories (e.g. fragments of `nat` and `set`) and better predicate calculus support would help, but they need to be interleaved with user interactions for providing key instantiations.

$\mathcal{W}$**:** The proof has confirmed our suspicion that the issue of "new" variables is nontrivial. Although it is true that the formalization of this aspect has not given us any deeper insight into the algorithm, it has helped us to elucidate some finer points like the non-requirement of idempotence. It also helps to avoid mistakes. Although we are not aware of any incorrect published proofs of $\mathcal{W}$, they do occur in closely related areas: for example, the completeness statement and proof of SLD-resolution in [12] are incorrect because they ignore the "new" variable issue.

Despite all this, the proof has left us with the feeling that there should be a simpler way to treat variables and substitution. However, this is only brought to a head by the requirement for complete formalization and is a sentiment known to most people who have worked with substitutions.

The most important next step is to extend the object-language with a `let`-construct and polymorphic types. Although we do not expect any major new problems, it is likely to be a substantial piece of work.

Acknowledgements We thank Thomas Stauner for his help with the Isabelle proofs, Franz Baader for an email discussion on the intricacies of mgus, and two anonymous referees for their constructive comments.

References

1. L. Cardelli. Basic polymorphic typechecking. *Sci. Comp. Programming*, 8:147–172, 1987.
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 and R. Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symp. Principles of Programming Languages*, pages 207–212, 1982.
4. L. M. M. Damas. *Type Assignment in Programming Languages.* PhD thesis, Department of Computer Science, University of Edinburgh, 1985.
5. 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.

6. M. Gordon and T. Melham. *Introduction to HOL: a theorem-proving environment for higher-order logic*. Cambridge University Press, 1993.
7. J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, 1969.
8. 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.
9. M. P. Jones. Qualified Types: Theory and Practice. Technical Monograph PRG-106, Oxford University Computing Laboratory, Programming Research Group, July 1992.
10. J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 257–321. MIT Press, 1991.
11. J.-L. Lassez, M. Maher, and K. Mariott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufman, 1987.
12. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
13. J. McKinna and R. Pollack. Pure type systems formalized. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lect. Notes in Comp. Sci.*, pages 289–305. Springer-Verlag, 1993.
14. R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
15. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
16. D. Nazareth. *A Polymorphic Sort System for Axiomatic Specification Languages*. PhD thesis, Technische Universität München, 1995. Technical Report TUM-I9515.
17. T. Nipkow and C. Prehofer. Type reconstruction for type classes. *J. Functional Programming*, 5(2):201–224, 1995.
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. R. Pollack. A verified typechecker. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1995.
21. J. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.
22. M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.
23. P. Wadler. Comprehending monads. In *Conference on Lisp and Functional Programming*, pages 61–78, June 1990.
24. P. Wadler. The essence of functional programming. In *Proc. 19th ACM Symp. Principles of Programming Languages*, 1992.
25. M. Wand. A simple algorithm and proof for type inference. *Fundementa Informaticae*, 10:115–122, 1987.

This article was processed using the LaTeX macro package with LLNCS style