Formalizing Type Theory in PVS: a case study

Technical report: ICIS-R08022

Sjaak Smetsers[1]

Institute for Computing and Information Sciences
Radboud University Nijmegen
The Netherlands

Erik Barendsen[2]

Institute for Computing and Information Sciences
Radboud University Nijmegen
The Netherlands

Abstract

In this case study we investigate the use of PVS for developing type theoretical concepts and verifying the correctness of a typing algorithm. PVS turns out to be very useful for efficient development of a sound basic theory about polymorphic typing. This research contributes to the PoplMark challenge on mechanizing metatheory.
The correctness of the typing algorithm is expressed as the so-called Contextual Principal Type Property, which is interesting in its own right.

Keywords: type systems, principal types, type inference, theorem proving, PVS

1 Introduction

This paper reports on a case study in computer aided verification of theories about syntactic objects.

Syntactic theories such as *type theories* play an important role in the (static) analysis of computer programs and construction of reliable implementations of programming languages. The usability and reliability of syntactic techniques could potentially be improved by using automated proof assistants, thus bridging the gap between theory and implementations.

[1] Email: S.Smetsers@cs.ru.nl
[2] Email: E.Barendsen@cs.ru.nl

9 December 2008

For example, subtle syntactical matters such as treatment of variables and bindings are crucial when implementing a typing algorithm as a compiler module. These syntactical details are usually not addressed in theoretical expositions and are therefore an embarrassing source of errors. These could have been avoided with a more detailed design and verification.

This need is recognized by many researchers. Most notably, the PoplMark Challenge [1] calls for experiments on verifications of metatheory of type systems using proof tools. The idea is to formalize existing proofs of properties of type systems with different proof assistants.

Our paper goes one step further: we propose to introduce the proof assistant already during the *development* of type theoretic concepts. With such a tool it is possible to verify the consistency of technical concepts while designing them. We do this by checking properties linking these concepts, such as substitution requirements. Tempting inaccuracies in the constructions are easily detected in this way. After successful formalization of the basic concepts, we use the proof tool to develop a complete correctness proof of the algorithms involved. We have done this in the case of a typing algorithm in a weakly polymorphic system.

The contribution of this paper is threefold.

First, it reports on a methodological experiment. We assess the usability of PVS for formalizing type theory. PVS has not been used for this kind of type theory yet, and there is no PVS-answer to the PoplMark challenge. Moreover we assess feasibility of the approach mentioned above: using PVS already during the development of basic theory and the algorithm. One could argue that the type system in our case study is simpler than the one in the PoplMark challenge. The technical complexity of the properties involved, however, is of the same level.

The second contribution is the machine verification of a typing algorithm. We prove the soundness and completeness of a specific refinement of the well-known Milner-Wand typing algorithm, the so-called Contextual Principal Type Property. This result is new and cannot be derived from existing results.

Finally, this paper proposes a labelling mechanism for dealing with various variable classes. With this method we avoid the tedious reasoning about α-conversion which is typical in formalizations of syntactic reasoning. We apply our method to distinguish between different roles of variables, connected to the specific universal quantification in our type theoretic setting. Our mechanism is flexible enough to be applied to similar systems with mixed free and bound variables such as rank-2 polymorphism and existential types [14].

2 Type theory

Typing is a powerful tool for static analysis of programs. Especially in the area of functional programming, numerous typing systems are used to capture properties varying from simple consistency of function applications to complex sharing requirements, see e.g. [3], [4].

This paper is about type systems with *weak polymorphism*. We focus on the

type reconstruction problem for these systems. It is well-known that typability of full polymorphism (*System-F* or $\lambda 2$ in typed lambda calculus) is undecidable, but there are many systems with some restricted ('weak') form of polymorphism, such as let-polymorphism and rank-2 polymorphism.

In this section we will introduce some basic type-theoretic notions.

First-order typing. Let us consider combinatory expressions built up from variables (from a given set V) and constants (C) using application and definition-abstraction:

$$\mathrm{E} ::= V \mid C \mid \mathrm{E}\,\mathrm{E} \mid \mathsf{let}\ V = \mathrm{E}\ \mathsf{in}\ \mathrm{E}.$$

Types are constructed from type variables ($\mathbb{V}$) with a function type constructor:

$$\mathbb{T} ::= \mathbb{V} \mid \mathbb{T}{\to}\mathbb{T}.$$

A type substitution is a function $* : \mathbb{V} \to \mathbb{T}$. The result of applying $*$ to σ is denoted by σ^*. In the sequel, we let $e, e_1, \ldots$ range over E, $\sigma, \tau, \ldots$ over $\mathbb{T}$.

Typing statements are of the form $\Gamma \vdash e : \tau$, where Γ is a set of declarations of the form $x{:}\sigma$. We suppose that the constants have some fixed type given by a type environment $\mathrm{env} : C \to \mathbb{T}$ called a *basis*. The typing rules are straightforward:

$$\Gamma, x{:}\sigma \vdash x : \sigma \qquad \Gamma \vdash c : \mathrm{env}(c)$$

$$\frac{\Gamma \vdash e_1 : \sigma{\to}\tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\, e_2 : \tau} \qquad \frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x{:}\sigma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}$$

The computation of a type for an expression is called *type inference*. There are several implementations of typing algorithms for functional languages. Most of these are based on Milner's algorithm, commonly denoted by $\mathcal{W}$ [13]. The approach by Wand ([23]) differs from Milner's approach in that type reconstruction is split into two phases. During the first phase, expressions are traversed and typing constraints (following from the rules for the respective syntactic constructions) are collected as type equations. Type variables are used to denote the unknowns in these equations. Solving the constraints (via unification) takes place during the second phase. In $\mathcal{W}$ the identified constraints are solved immediately. See also [2].

The *principal typing algorithm* decides for each e whether it is typable; in the positive case it computes a *principal pair* Γ, σ which is correct (*sound*), i.e.,

$$\Gamma \vdash e : \sigma$$

and moreover *complete*, i.e., each other typing can be obtained from Γ, σ by substitution:

$$\Gamma' \vdash e : \sigma' \quad \Rightarrow \quad \Gamma' \supseteq \Gamma^*,\ \sigma' = \sigma^* \text{ for some } * .$$

A stronger variant of the principal typing algorithm computes a *contextual principal type* σ given Γ and e. The substitution in the soundness property then only affects σ. The contextual version implies the principal typing property, but not vice versa. We will focus on this contextual principal type property in our formalization. Moreover, we will use Wand's style of deriving types.

Weak polymorphism. We will describe a system which allows types with universal quantification of variables at the outermost level, such as $\forall\alpha.\alpha{\to}\alpha$. The resulting set of *type schemes* is denoted by $\mathbb{T}^\forall$:

$$\mathbb{T}^\forall ::= \mathbb{T} \mid \forall\mathbb{V}.\mathbb{T}^\forall.$$

$S, T, \ldots$ range over $\mathbb{T}^\forall$. Type schemes are assigned to expression variables (by Γ), and to constants (by the type environment $\mathrm{env} : C \to \mathbb{T}^\forall$). These schemes can be *instantiated* by substituting types for the quantified variables. To this end, the first order system is extended with rules such as

$$\frac{\Gamma \vdash e : \forall\alpha.S}{\Gamma \vdash e : S[\alpha := \tau]} \qquad \frac{\Gamma \vdash e : S}{\Gamma \vdash e : \forall\alpha.S} \quad (\alpha \text{ not free in } \Gamma)$$

For this system one can prove principal typing results like for the first-order case. To allow for an inductive generation of type constraints one can transform the system into a 'syntax directed' one, in which each rule corresponds to exactly one syntactic construction. We will not go into the details.

3 Formalizing expressions and types

In this section we will prepare for the representation of types and show how to formalize the basic notions in PVS [18]. In our formalization we will consider a variant of the weakly polymorphic system introduced in the previous section. In our case study we wish to focus on the (sometimes subtle and error-prone) administration and manipulation of *types* and the various rôles of type variables. We therefore restrict the *expression* syntax to the simplest interesting example: the applicational fragment, so without let expressions. This is not a serious restriction, since [21] shows that let-polymorphism can be translated into a purely combinatoric system via substitutions.

Types with markings

In the typing algorithm for weak polymorphism one has to distinguish two substitution-like operations on types. *Instantiation* should affect the (quantified) scheme variables, but not the other (free) variables. The *solving substitutions* should be restricted to auxiliary type variables (denoting the unknowns in equations). For a transparent formalization we introduce the notion of *marked type variables* and two replacement operations on marked types. We will use this for a 'secure' formalization of the changing rôles of the variables in the subsequent steps of the algorithm.

The collection $\underline{\mathbb{T}}$ of *marked first-order types* is built up from type variables that can appear either plain or marked (denoted by underlining):

$$\underline{\mathbb{T}} ::= \mathbb{V} \mid \underline{\mathbb{V}} \mid \underline{\mathbb{T}} \rightarrow \underline{\mathbb{T}}.$$

We will describe two replacement operations on marked types: instantiation of marked variables and substitution of unmarked variables. Let $\sigma \in \underline{\mathbb{T}}$ and let $* : \mathbb{V} \rightarrow \underline{\mathbb{T}}$ be a substitution. The *instantiation effect* of $*$ on σ, denoted by $[\sigma]^*$, is defined inductively by

$$\begin{aligned} [\alpha]^* &= \underline{\alpha}, \\ [\underline{\alpha}]^* &= *(\alpha), \\ [\sigma \rightarrow \tau]^* &= [\sigma]^* \rightarrow [\tau]^*. \end{aligned}$$

Observe that the free (i.e. unmarked) scheme variables of σ are marked in $[\sigma]^*$. The *substitution effect* of $*$ on σ, denoted by $(\sigma)^*$, is defined by

$$\begin{aligned} (\alpha)^* &= *(\alpha), \\ (\underline{\alpha})^* &= \underline{\alpha}, \\ (\sigma \rightarrow \tau)^* &= (\sigma)^* \rightarrow (\tau)^*. \end{aligned}$$

Typing in the weakly polymorphic system can be expressed using marked first-order types: quantified variables can be represented as marked variables. The instantiation mechanism becomes

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : [\sigma]^*}$$

or $\Gamma, x{:}\sigma \vdash x : [\sigma]^*$ and $\Gamma \vdash c : [\mathrm{env}(c)]^*$ in the syntax directed variant.

Representing syntax in PVS

We formalize the syntax in our proof tool. While explaining the formalization we will give a brief introduction to PVS.

PVS offers an interactive environment for the development and analysis of formal specifications. The system consists of a specification language and a theorem prover. The specification language of PVS is based on classical, typed higher-order logic. It resembles common functional programming languages, like Haskell, LISP or ML. PVS supports inductive definitions.

We use the following representation of expressions.

```
EXPR[V:TYPE, C:TYPE] : DATATYPE
BEGIN
    e_var   (v_id: V)              : e_var?
    e_const (c_id: C)              : e_const?
    e_appl  (e_fun, e_arg: EXPR)   : e_appl?
END EXPR
```

The basic syntactical categories V and C appear as the *parameters* `V` and `C` of the inductive data type `EXPR`. The data type itself has three *constructors*, `e_var`, `e_const`

and `e_appl` for representing values in `EXPR`. In addition, three *recognizers* `e_var?`, `e_const?` and `e_appl?` are defined (PVS allows question marks as constituents of identifiers), which can be used as predicates to test whether or not an `EXPR` object starts with the respective constructor. It will be more convenient, however, to define operations on inductive datatypes by pattern matching.

For each data type, PVS generates a collection of so-called *theories*. One of those theories contains the basic declarations and axioms formalizing the data type, including an induction scheme for proofs. Moreover, instantiations of some generic operations such as `map` (for lifting functions) and the recursor `reduce` are generated.

For example, the function `fvs` (giving the *free variables* of an expression) can be defined using the recursor by specifying the results for each case of the inductive data type (variable, constant, application):

```
fvs: [EXPR → PRED[V]] = reduce(singleton,λ(c:C):∅,∪)
```

When applied to an expression e, this function will return the subset of V (in PVS denoted as `PRED[V]`) consisting of the variables occurring in e. We have used the predefined set operations `singleton` and `union`.

More involved inductive definitions can be given using a general pattern matching scheme (`CASES`). In particular, the function `reduce` itself is defined internally as follows.

```
reduce (vf:[V → ran],cf:[C → ran],af:[[ran, ran] → ran]): [EXPR → ran]
    = λ (e: EXPR): LET red:[EXPR → ran] = reduce(vf,cf,af)
                   IN CASES e OF
                        e_var(v):    vf(v),
                        e_const(c):  cf(c),
                        e_appl(f,a): af(red(f), red(a))
                      ENDCASES
```

¡¡¡¡¡¡¡ formalization.tex ======= ¿¿¿¿¿¿¿ 1.4

We will use the concept of marked types both for type schemes in environments (Γ and env) and for special variables in type assignments ($e : \sigma$). For convenience, however, we will use different names for these two occurrences of the collection of marked types: we represent them as two separate data types. This is shown in the following declarations.

```
SCHEME[V : TYPE]: DATATYPE
BEGIN
    s_bv  (bv:V): s_bv?
    s_fv  (fv:V): s_fv?
    s_arr (arg, res:SCHEME): s_arr?
END SCHEME
```

```
MTYPE[V:TYPE]: DATATYPE
BEGIN
    t_mv  (t_var:V): t_mv?
    t_fv  (t_var:V): t_fv?
    t_arr (t_arg, t_res:MTYPE): t_arr?
END MTYPE
```

```
discard(x:V) :PRED[V] = ∅

fvs:[SCHEME → PRED[V]] = reduce(discard,singleton,∪)
bvs:[SCHEME → PRED[V]] = reduce(singleton,discard,∪)

fvs:[MTYPE → PRED[V]]  = reduce(discard,singleton,∪)
mvs:[MTYPE → PRED[V]]  = reduce(singleton,discard,∪)
```

Contrary to our approach, in [15] and [16] expressions are typed with *monomorphic*

types (i.e. types with only one kind of type variables). A consequence is that type variables occurring free in the type scheme can, after instantiation, be altered during type reconstruction. The advantage of using two kinds of variables is that they remain distinguishable, even after unification. We will explain this in more detail in section 5.

The instantiation and substitution operations (denoted earlier by $[\cdot]^*$ and $(\cdot)^*$) can be defined easily using the recursors for `SCHEME` and `MTYPE`.

```
SUBST : TYPE = [V → MTYPE];

inst(s:SUBST) : [SCHEME → MTYPE] = reduce(s,t_mv,t_arr)
subst(s:SUBST): [MTYPE → MTYPE]  = reduce(t_mv,s,t_arr)
```

Observe that `inst` changes free scheme variables into marked type variables, implying that they cannot be further instantiated via substitution. In [21], an operation is introduced which converts a type back into a scheme. Usually this is called *generalization*. The result depends on the context in which the operation is performed, in particular on the type variables appearing in the used base. Generalization corresponds to the $\forall$-introduction rule in the weakly polymorphic type system. In PVS:

```
gen(p:pred[V]): [V → SCHEME]
    = λ(v:V): IF p(v) THEN s_fv(v) ELSE s_bv(v) ENDIF

generalize(p:pred[V]): [MTYPE → SCHEME]
    = reduce(gen(p), s_bv, s_arr)
```

Typically, the function `generalize` will be parameterized with the free variables of the present basis. Generalization (below indicated as $\mathcal{G}$) plays a crucial role in the proof of the following property concerning substitutions.

$$\Gamma \vdash e_1[x := e_2] : \sigma \;\Rightarrow\; \Gamma \vdash e_2 : \tau, \quad \Gamma, x{:}\mathcal{G}(\Gamma, \tau) \vdash e_1 : \sigma \text{ for some } \tau.$$

This property is used by [21] to justify the way let-polymorphism is translated into our combinatoric system.

For convenience, we prefer to use the infix operation `**` instead of `subst`. Moreover, the operator $\leq$ is used to express 'is an instance of'. We have two different versions, one for schemes and one for types.

For the definition of infix operations, PVS requires a specific syntax that does not allow parameter types to be included in the argument list.

```
s          : VAR SUBST
t, t1, t2  : VAR MTYPE
ts         : VAR SCHEME

**(t,s): MTYPE = subst(s)(t)
≤(t1, t2) : bool = ∃ (s) : t2 = t1 ** s;
≤(ts, t)  : bool = ∃ (s) : t = inst(s)(ts);
```

4 Formalizing the typing system

To specify the type inference rules, we make use of PVS's facility to define *inductive predicates*. The type system is specified as a separate PVS theory `typingEXPR`. This theory has the environment `env` assigning types to constants as parameter.

```
typingEXPR [V, X, C:TYPE,
            (IMPORTING SCHEME[V]) env: [C → SCHEME[V]]]: THEORY
BEGIN
   BASE : TYPE     = [X → SCHEME]
   b  : VAR BASE
   st : VAR [EXPR,MTYPE]

   |-(b,st) : INDUCTIVE bool =
       CASES st`1 OF
           e_var(w)     : b(w)   ≤  st`2,
           e_const(c)   : env(c) ≤  st`2,
           e_appl(f, a) : ∃ (t:MTYPE): (b |- (f,t_arr(t,st`2)))
                                       ∧ (b |- (a, t))
       ENDCASES
END typingEXPR
```

Above, the variable `st` is declared as a pair consisting of an expression and its type. The notation `st`n` is used to select the $\mathtt{n}^{th}$ component of `st`.

An important property for our final theorem stating that type derivation is closed under substitution, is the following:

```
typable_subst : LEMMA
    ∀(e:EXPR, t:MTYPE, b:BASE, s:Substitution):
          (b |- (e,t)) ⇒ (b |- (e, t ** s))
```

The proof by induction on the structure of `e` is straightforward.

5 Formalizing the algorithm

Until now, we only considered the monomorphic subset of `MTYPE`. In the present section, the role of the free variables will become apparent: they serve as unknows in type equations. These type equations are represented as a list of pairs. Solving these equations is usually done via *unification*: the process of finding a substitution that is a *unifier* for all equations appearing in the list.

In PVS equations and solutions can be defined as follows:

```
EQS : TYPE   = list[[MTYPE, MTYPE]]
solves(s): pred[EQS]  = every(λ(t1, t2: MTYPE): t1 ** s = t2 ** s)
```

The predefined combinator `every` checks if all elements of a list satisfy a given predicate. In this case we verify whether the given substitution `s` is a unifier for each pair of types.

It is well-known that unification is decidable. However, correctness of type inference does not depend on a particular implementation of unification, but merely on some general properties. As is, we do not give a unification algorithm but specify its properties via *axioms*. A machine verified proof of these properties (also by using PVS) for the Robinson unification [20] is given in [9]. The second axiom

(mgu_complete) is based on the usual ordering on substitutions:

```
s, s1, s2: VAR SUBST
≤(s1, s2) : bool = ∃ s : s2 = (s o s1)
mgu: [EQS → lift[SUBST]]

mgu_sound : AXIOM
    ∀(eqs:EQS): mgu(eqs) = up(s) ⇒ solves(s)(eqs)
mgu_complete : AXIOM
    ∀(eqs:EQS): solves(s)(eqs) ⇒ up?(mgu(eqs)) ∧ down(mgu(eqs)) ≤ s
```

The predefined lift datatype adds a bottom element to a given base type, in this case SUBST. This is useful for defining partial functions, particularly to indicate the cases that unification fails.

The next step is to associate a set of type equations with each expression e, in such a way that typability of e can be expressed in terms solvability of those equations.

The generation of these equations is recursively defined on the structure of e. This algorithm needs to generate new free type variables. In hand written proofs this issue is often disposed of in a single remark stating that at certain points *fresh* variables are introduced. This merely means that these variables do not clash with variables used elsewhere. Obviously this solution will not work in a machine verified proof, which forces us to formalize such a notion of *freshness*. The easiest way to do this is by using natural numbers as type variables and by explicitly maintaining a counter indicating the next free variable number. The counter is incremented each time a fresh variable is required. This counter (below named heap) is returned as an additional component of the result of generate. Similar to [15], in our definition of generate we use two auxiliary functions called fresh and next_bv. The first function is used to create a fresh instance of a type scheme, i.e. a type in which bound scheme variables are substituted by fresh free variables. The second function computes the offset with which our heap must be increased such that uniqueness of fresh variables remains guaranteed.

```
fresh(heap:nat): [SCHEME → MTYPE] = inst(λ(n:nat):t_fv(n+heap))
next_bv: [SCHEME → nat] = reduce (λ(n:nat):n+1,λ(n:nat):0,maximum)
equa(t1,t2:MTYPE): EQS = cons((t1,t2),null)

generate(b:BASE)(e:EXPR,t:MTYPE)(h:nat): RECURSIVE [nat, EQS] =
  CASES e OF
    e_var(v):   (next_bv(b(v))+h,  equa(t,fresh(h)(b(v)))),
    e_const(c): (next_bv(en(c))+h, equa(t,fresh(h)(en(c)))),
    e_appl(f, a): LET (fh,feqs) = generate(b)(f,t_arr(t_fv(h),t))(h+1),
                      (ah,aeqs) = generate(b)(a,t_fv(h))(fh)
                  IN (ah, append(feqs,aeqs))
  ENDCASES
MEASURE e BY ≪
```

The MEASURE specification is a standard part in the definition of recursive functions such as generate. In PVS all functions are total. The measure is used to show that the function terminates. This is done by generating a proof obligation (a so-called Type Correctness Condition, *TCC*) indicating that the measure strictly decreases at each recursive call. In this case we can use the standard subtree-ordering on elements inductive data types <<. This ordering is part of the standard theory

generated with each inductive data type.

As can be deduced from the PVS code, `generate` uses free type variables as placeholders which are filled in later via unification. The advantage of separating these free variables from marked variables is that the substitution resulting from unification is restricted to placeholders only. This appears to crucial when formulating and proving the principal types property.

6 The correctness proof

In this section we show that type assignment has the *Contextual Principal Type Property*. The proof is divided into three steps. The first two steps concern correctness of our procedure, i.e. *soundness* and *completeness*. In the third step our main theorem is proven using both correctness and the properties of the unification algorithm.

Soundness. The soundness property can be formulated as follows:

```
generate_sound: PROPOSITION
    ∀(b:BASE,e:EXPR,t:MTYPE,n:nat,s:SUBST):
        solves(s)(generate(b)(e,t)(n)`2) ⇒ (b  |- (e,t ** s))
```

This proposition is proven by induction on the structure of `e`. In contrast to [15], we do not have any side-conditions with respect to the free variables occurring in `b`. The proof itself is actually not difficult and relatively short (approximately 50 proof steps). This size is slightly misleading because it depends on many intermediate results which were proved separately. The main lemma occurring in the proof (relating the instance of a scheme `ts` to a substitution on a fresh copy `ts`) is:

```
fresh_inst : LEMMA
    ∀(n:nat,ts:SCHEME,s:SUBST):
        inst(λ(m:nat):s(n+m))(ts) = fresh(n)(ts) ** s
```

This lemma can be proven in just 15 steps, using structural induction on `ts`.

Completeness. The formulation as well as the proof of the completeness property is more subtle.

```
generate_complete: PROPOSITION
  ∀(m:nat,n:(below?(m)),b:BASE,e:EXPR,t:(betweenT?(n,m)),s1:(bsubst?(n,m))):
      (b |- (e, t ** s1)) ⇒
          LET (nheap,eqs) = generate(b)(e,t)(m)
          IN ∃(s2:(bsubst?(n,nheap))): solves(s2)(eqs)
                 ∧ restrict(s2)(between?(n,m))= s1
```

To complete the proof we had to make some specific assumptions on the free type variables occurring in the input type `t` and the substitution `s1`. These assumptions are formulated using *dependent types*: types depending on values. E.g. the predicate `betweenT?(n,m)` states that any variable `v` occurring in `t` lies between `n` and `m`. For substitutions the predicate `bsubst?(n,m)` does something similar: both domain and range of a substitution should be bounded by `n` and `m`. The function `restrict` restricts `s2` to elements of the specified set, in this case `between?(n,m)`.

The complexity of the proof is significantly greater than the soundness proof: it requires some 1500 proof steps, not to mention the numerous sublemmas that are involved. Similar to the proof in [15], the `e_appl`-case is not only lengthy but also quite difficult.

Contextual Principal Types. We finally arrive at one of the main goals of our exercise: the contextual principal type property. A necessary technicality is that we show that the principal type itself is *clean*, meaning that there is no overlap between free and marked variables. The latter is important because the generalization (as defined in section 3) of a 'unhygienic' type may lead to undesired name clashes.

```
Clean? : PRED[MTYPE] = { t : MTYPE | disjoint?(fvs(t),mvs(t)) }

principal_types: THEOREM
    ∀(b:BASE, e:EXPR): ∀(t1, t2:MTYPE):
        (b |- (e,t1)) ∧ (b |- (e,t2)) ⇒
            ∃(t:(Clean?)): (b |- (e,t)) ∧ t ≤ t1 ∧ t ≤ t2
```

Roughly, the proof proceeds as follows. Create a fresh variable, say `v`, and two singleton substitutions assigning `t1` and `t2` respectively to `v`. Use `generate_complete` twice with `t_fv(v)` as `t` and the above singleton substitutions as `s1`. This results in two new substitutions, both solving the set of generated equations. By `mgu_complete` we obtain the most general solution for these equations. Then the combination of `mgu_sound` and `generate_sound` gives us a type `t` for `e`. Transforming this type into a clean variant and showing that this variant is smaller than or equal to both `t1` and `t2`. This is just a matter of simple case distinctions.

7 Conclusion

We have successfully formalized the syntax of expressions and types, as well as a typing system with weak polymorphism. We used the formalization to obtain an exciting new technical result. This was not done by using an existing 'paper proof', but by developing it from scratch within the proof tool.

Our formalization includes a variable administration to deal with mixed rôles of variables in systems with quantifiers (such as polymorphic and existential type systems). The representation allows for reasoning about these technical matters at a conveniently high level.

We have used PVS for two goals: consistent development of syntactical concepts (such as typing) and verification of an algorithm.

The use of the proof tool for conceptual development turned out to be very beneficial. Inaccuracies stemming from implicit assumptions in theoretical expositions were quickly discovered and repaired. The formalization of the correctness proof did not lead to any alterations in the basic theory: stability was achieved after checking some basic properties.

The formalization of the correctness proof[3] took about 2000 steps, comparable

[3] All the proofs presented in this paper can be downloaded from

to formalizations of similar syntactic theories. Our proofs could be optimized further by adding more lemmas, replacing repeating patterns.

After a short time, PVS turned out to be a helpful partner. In systems such as Coq [19] the focus is on 'backward reasoning', transforming proof goals into simpler ones. The ability of PVS to combine this with forward reasoning (reasoning from assumptions) was greatly appreciated. There is no general consensus about which proof tool is preferable, but our experience does not contradict the conclusions of the comparison in [7].

8 Related work

In the spirit of the PoplMark paper, it is interesting to compare our result with similar formalizations. The closest formalization is the verification of the 'classical' principal type property in Isabelle [17] reported in [16] and [15].

The complexity of the formalization (measured in definitions and proof steps) is roughly the same as ours, which is reasonable considering that the results deal with similar type theoretic concepts.

The structure of the Isabelle formalization looks different from the one in PVS. This is likely to be due to the different proof styles supported by Isabelle and PVS respectively. Isabelle focuses on goal-driven proving (so called 'backward reasoning'). Users typically adopt a bottom-up proof strategy, first proving many auxilary basic results before referring to these results in proofs of more complex properties. PVS supports both backward and forward reasoning (resasoning from assumptions), thus allowing a mixed (bottom-up and top-down) strategy. The main reason to single out an auxiliary result is that it occurs more often in a proof. These differences are not essential: a routine user will become comfortable with either style. It is merely interesting to see the difference being reflected in the structure of the proof code.

It would not be easy to adapt the proof in [15] of the Principal Type Property to a proof of the Contextual Principal Type Property. The 'dual' role of variables in bases and object types requires a less straightforward representation of these.

One of the PoplMark challenges is the treatment of variable binding. Several solutions to this challenge have been reported, e.g. [5]. Most of these are based on de Bruijn indices. Though [1] argues that this representation introduces too much overhead in formal proofs, and therefore should be avoided, this is not confirmed by any of the presented solutions.

Our approach can be regarded as a combination of De Bruijn indices and a labelling mechanism to avoid α-conversion reasoning. This works well in our case, allowing straightforward proofs of instantiation properties. Moreover, our method can easily be applied to other mechanisms of variable binding, such as rank-2 polymorphism and existential types [14].

The idea of explicitly distinguishing free from bound variables is also employed

`http://www.cs.ru.nl/S.Smetsers/files/principal.zip`

in [?]. The authors introduce two disjoint syntactic categories called *variables* and *parameters*. In our approach, the role of a variable is indicated by labels in one category, allowing the change of roles in the substitution and instantiation mechanisms to be expressed in a straightforward way.

An alternative solution is proposed in [?], introducing the theory and applications of *nominal sets*, a mathematical model of names and binding based on permutations. The structure of α-equated terms can be defined as a nominal datatype. The main advantage of the construction is that one automatically obtains a structural induction principle. This induction principle is available as a package for Isabelle/HOL, and experiments have showed that mechanized proofs are almost identical to hand-written proofs in which α-conversion is done implicitly. It would be interesting to add this structural induction principle to PVS, and to compare our approach with that of [22].

In a broader sense, our work can be considered as a contribution to fully formalized (both functional and imperative) languages and fully verified compilers, see e.g. [12], [10]. Most theorem proving in this area has been done in Coq or LF. We prefer the more flexible proof style of PVS.

The present research is part of a larger project using PVS for both the verification of existing software [8] and the development of new software [9]. For instance, the correctness of a scheduling protocol for a smart-card personalization machine has been proven in [11]. This protocol was used as a case study to test the power of model checkers. Due to their nature, model checkers were only capable of verifying correctness for a machine with a limited amount of personalization units. Using PVS it is shown that the correctness holds for any number of units.

We have applied the present formalization to prove a conjecture in an analysis of program transformations [21]. It turned out to be easy to connect our formalization with existing work of the program transformation. It is crucial that our result allows the typing basis to be fixed in a proof. The classical Principal Type Property (about principal *pairs*) would not be usable. The combination of these results shows that in a core functional language like Mini-ML [6] principal types can be computed effectively.

References

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLMARK challenge. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

[2] H.P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbai, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford Univ. Press, 1992.

[3] E. Barendsen and J.E.W. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. 6:579–612, 1996.

[4] E. Barendsen and J.E.W. Smetsers. Graph rewriting aspects of functional programming. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, pages 63–102. 1999.

[5] Stefan Berghofer. A solution to the POPLMARK challenge in Isabelle/Hol, 2006. report.

[6] Dominique Clément, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. A simple applicative language: mini-ml. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 13–27, New York, NY, USA, 1986. ACM Press.

[7] David Griffioen and Marieke Huisman. A comparison of PVS and Isabelle/HOL. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs '98*, volume 1479, pages 123–142, Canberra, Australia, 1998. Springer-Verlag.

[8] M. Hohmuth and H. Tews. The semantics of C++ data types: Towards verifying low-level system components. In D. Basin and B. Wolff, editors, *TPHOLs 2003, Emerging Trends Proceedings*, pages 127–144. 2003. Technical Report No. 187 Institut für Informatik Universität Freiburg.

[9] Bart Jacobs, Sjaak Smetsers, and Ronny Wichers Schreur. Code-carrying theories. *Form. Asp. Comput.*, 19(2):191–203, 2007.

[10] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ml. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 173–184. ACM, 2007.

[11] Leonard Lensink, Sjaak Smetsers, and Marko van Eekelen. Machine checked formal proof of a scheduling protocol for smartcard personalization. In *12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2007)*, Berlin, Germany, 2007. LNCS. To appear.

[12] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54, New York, NY, USA, 2006. ACM Press.

[13] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[14] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[15] Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm $\mathcal{W}$ in Isabelle/HOL. *Journal of Automated Reasoning*, 23:299–318, 1999.

[16] Dieter Nazareth and Tobias Nipkow. Formal verification of algorithm $\mathcal{W}$: The monomorphic case. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *LNCS*, pages 331–346. Springer, 1996.

[17] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[18] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS language reference (version 2.4). Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001.

[19] The Coq Development Team LogiCal Project. The Coq proof assistant reference manual. Technical Report Version 8.1, 2006.

[20] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[21] Sjaak Smetsers and Arjen van Weelden. Bracket-abstraction preserves typability: A formal proof of Diller-algorithm-C in PVS. In Jordi Levy, editor, *20th International Workshop on Unification, UNIF 2006. Preliminary proceedings*, pages 29–43, Seattle, USA, August 2006.

[22] Christian Urban and Christine Tasson. Nominal techniques in Isabelle/Hol. In Robert Nieuwenhuis, editor, *CADE*, volume 3632 of *Lecture Notes in Computer Science*, pages 38–53. Springer, 2005.

[23] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Infomaticae*, X:115–122, 1987.