Handling Polymorphism in Automated Deduction

Jean-François Couchot[1,2] and Stéphane Lescuyer[1,2]

[1] INRIA Futurs, ProVal, Parc Orsay Université, F-91893
[2] LRI, Univ Paris-Sud, CNRS, Orsay, F-91405
{couchot,lescuyer}@lri.fr

Abstract. Polymorphism has become a common way of designing short and reusable programs by abstracting generic definitions from type-specific ones. Such a convenience is valuable in logic as well, because it unburdens the specifier from writing redundant declarations of logical symbols. However, top shelf automated theorem provers such as Simplify, Yices or other SMT-LIB ones do not handle polymorphism. To this end, we present efficient reductions of polymorphism in both unsorted and many-sorted first order logics. For each encoding, we show that the formulas and their encoded counterparts are logically equivalent in the context of automated theorem proving. The efficiency keynote is to disturb the prover as little as possible, especially the internal decision procedures used for special sorts, e.g. integer linear arithmetic, to which we apply a special treatment. The corresponding implementations are presented in the framework of the Why/Caduceus toolkit.

1 Introduction

Polymorphism allows a single definition to be used with different types of data: a polymorphic function's definition generalizes several type-specific ones. Obviously, polymorphism is not reduced to the foundation of basic arithmetic operators (e.g. + deals naturally with integers, reals, vectors, ...), but it is a common feature of object-oriented and functional languages. For instance, a function `append` which concatenates two lists of elements of the same type can be constructed without taking care of the elements' type: let α be any type and let α LIST denote the type of lists with elements of type α. `append` can then be typed α LIST $\times$ α LIST $\rightarrow$ α LIST.

In the context of program verification by analysis of assertions and generation of proof obligations, formalizing the features of a programming language involves a large collection of definitions and axioms, some of them being polymorphic. Even when the language, like C, does not have polymorphism, memory modelization makes use of polymorphic types and definitions. In these cases, polymorphism becomes critical since it allows specifications to be both typed and short: types can be used to guide the prover and avoid ill-sorted deductions, which often pollute the proof search of unsorted provers.

F. Pfenning (Ed.): CADE 2007, LNAI 4603, pp. 263–278, 2007.

Then, it is definitely a challenge to automatically discharge the verification conditions expressed in a polymorphic first order language: only interactive higher-order proof assistants like Isabelle/Hol [1] or Coq [2] are amenable to such polymorphic logic goals. Among automated provers, some deal with unsorted First Order Logic (FOL), e.g. Simplify [3] or haRVey [4] and others treat many-sorted FOL, e.g. SMT-LIB [5] provers and among them Yices [6] or CVC-lite [7], but none of them allows the definition of uninterpreted polymorphic symbols.

The main contribution of this work is a correct and complete reduction of polymorphic many-sorted FOL to unsorted FOL that preserves the sorting informations (Sect. 3). The second contribution is an equally powerful reduction targeting many-sorted FOL (Sect. 4), which takes advantage of sorted provers to provide an *ad hoc* translation for interpreted terms, e.g. arithmetic terms. The third contribution is an implementation of both approaches and their application on a full scale benchmark (Sect. 5). Section 2 presents some formal preliminaries about polymorphism, gives a running example and shows how inefficient some coarse techniques of polymorphism reduction are. Section 6 discusses works related to our study, concludes and presents future work. Note that proofs are sketched in this paper to fit within the space constraints. An extended version [8] details them.

2 Preliminaries

Our polymorphic logic is built on classical first order logic and its usual connectives $\vee, \neg, \exists, \ldots$ Atoms are built on a signature $\Sigma = (\mathcal{S}, \mathcal{F}_s, \mathcal{X}_s, \mathcal{X}, \mathcal{F})$ where $\mathcal{S}$ is the set of constant sorts, $\mathcal{F}_s$ is the set of sort functions (e.g. LIST or ARRAY), $\mathcal{X}_s$ is the set $\{\alpha, \beta, \ldots\}$ of type variables, $\mathcal{X}$ is the set of term variables and $\mathcal{F}$ is the set of all functional and predicative symbols (including constant symbols).

Each symbol in $\mathcal{F}_s$ comes with an integer number n of parameters and the set of all sorts Ω is the smallest set containing type variables and constant sorts such that if $f \in \mathcal{F}_s$ is a sort function of arity n and $s_1, \ldots, s_n \in \Omega$, then $f(s_1, \ldots, s_n) \in \Omega$.

Similarly, each functional symbol in $\mathcal{F}$ comes with an arity $s_1 \times \ldots \times s_n \rightarrow s$ where each s_i, $1 \leqslant i \leqslant n$, is in Ω and s is either in Ω or the special propositional sort o. When $n = 0$, the symbol is a constant of type s. Variables from $\mathcal{X}$ and functional symbols from $\mathcal{F}$ are then used to build terms in the usual manner. Among these terms, those of type o form the atoms of our propositional language.

Since we are only interested in well-typed atoms and well-typed formulas, we define a well-typing relation $\Gamma \vdash e : \tau$ where Γ is a typing context (i.e. a mapping from $\mathcal{X}$ to Ω) and τ is a sort. Inference rules for this typing system are given in Fig. 1. The rules for $\forall$, $\neg$ and $\wedge$ are similar to rules (EXISTS) and (OR) and are not shown. In the following, unless we specify otherwise, we only consider well-typed formulas ϕ, that is, such that $\vdash \phi : o$. Finally, type variables that appear in formulas are implicitly universally quantified.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad (\textsc{Var}) \qquad \frac{\begin{array}{c}\sigma \text{ substitution from } \mathcal{X}_s \text{ to } \Omega \\ f : s_1 \times \ldots \times s_n \to s \\ \text{for all } i, \Gamma \vdash e_i : s_i\sigma\end{array}}{\Gamma \vdash f(e_1, \ldots, e_n) : s\sigma} \quad (\textsc{App})$$

$$\frac{x \in \mathcal{X} \quad \Gamma, x : s \vdash e : o}{\Gamma \vdash \exists x : s.e : o} \quad (\textsc{Exists}) \qquad \frac{\Gamma \vdash \phi : o \qquad \Gamma \vdash \psi : o}{\Gamma \vdash \phi \vee \psi : o} \quad (\textsc{Or})$$

Fig. 1. Typing system

Let us now explain why the simplest encodings that come to mind for removing typing information are not adequate. First of all, instead of dealing with sorts, one may think that we can omit sorting information, but unfortunately, this method is unsound. For instance, consider the following formula: $\forall x : \mathit{unit} \,.\, x = \mathit{void}$. It states that sort *unit* is a singleton whose only member is *void*. Removing sorts in this formula gives $\forall x \,.\, x = \mathit{void}$ which states that *every* term is equal to *void*, and in turn easily yields things like $1 = 2$.

Secondly, the reader who is familiar with polymorphism can think that a polymorphic theory can be reduced to a many-sorted one by instantiating the type variables in polymorphic symbols according to the contexts they are applied to, and generating as many instances as necessary to "monomorphize" the original theory.

However, the computation of these mandatory monomorphic instances can be an infinite process: for instance, consider the symbol $\mathit{cons} : \alpha \times \alpha\,\textsc{list} \to \alpha\,\textsc{list}$, which adds an element in front of a list, and $\mathit{nested} : \textsc{int} \to \alpha\,\textsc{list}$, which is axiomatized by:

$$\mathit{nested}(0) = \mathit{cons}(0, \mathit{nil})$$
$$\forall x : \textsc{int} \,.\, \mathit{nested}(x + 1) = \mathit{cons}(\mathit{nested}(x), \mathit{nil})$$

If *nested* is then applied to some arithmetic function f whose result is not statically predictable, one can see that there is no way to adequately monomorphize this theory since a potentially infinite number of instantiations of *cons* (on $\textsc{int}\,\textsc{list}$, $(\textsc{int}\,\textsc{list})\,\textsc{list}$, and so on) may be required. Even if this shortcoming does not appear often in practical cases, monomorphization always returns theories that are much bigger than the original ones, which dramatically slows down provers. Therefore, this approach is not satisfactory in our context.

Finally, a potentially good reduction of polymorphic logic to unsorted logic is suggested by a reduction from many-sorted logic to unsorted logic developed in [9, p. 277]: it suffices to take a language with the same symbols as in the many-sorted language being translated, augmented with a predicate symbol P used to ensure the type of variables. P has two parameters: the first one is the term being sorted and the second one is its sort, given as a term. For instance, the formula $\forall l : \alpha\,\textsc{list} \,.\, \mathit{append}(l, \mathit{nil}) = l$ would be translated to $\forall \alpha, l \,.\, P(l, \mathit{list}(\alpha)) \Rightarrow \mathit{append}(l, \mathit{nil}) = l$.

Similarly, sorts in existentially quantified formulae are replaced by a conjunction and axioms are added for all symbols so that $P(t, \tau)$ is provable iff t has type τ. For instance, the following axioms are added for lists:

$$\forall \alpha \,.\, P(nil, list(\alpha))$$
$$\forall \alpha, x, l \,.\, (P(x, \alpha) \wedge P(l, list(\alpha))) \Rightarrow P(cons(x, l), list(\alpha))$$

Altogether, this new typing predicate P and the axioms ensure soundness and completeness. However, this translation cannot be used in our context since it strongly modifies the boolean structure of formulas, especially by introducing useless disjunctions, that are often inefficiently handled by SMT solvers (see benchmarks in Sect. 5).

Notice that this encoding is not a solution for many-sorted logic with built-in types: we cannot give a type for P that is compliant both with terms of built-in sorts and user ones. Adding one predicate p_t for each sort t is not a solution either since it does not allow one to deal with polymorphism.

We now present the running example, used in the following to illustrate non-obvious steps. Let $acc : \alpha\,\text{ARRAY} \times \text{INT} \rightarrow \alpha$ and $upd : \alpha\,\text{ARRAY} \times \text{INT} \times \alpha \rightarrow \alpha\,\text{ARRAY}$ be the functional symbols of an array theory [10], axiomatized by:

$$\forall a : \alpha\,\text{ARRAY}, i : s\text{INT}, e : \alpha \,.\, acc(upd(a, i, e), i) = e \tag{1}$$
$$\forall a : \alpha\,\text{ARRAY}, i : \text{INT}, j : \text{INT}, e : \alpha \,.\, i \neq j \Rightarrow acc(upd(a, i, e), j) = acc(a, j). \tag{2}$$

Intuitively, a being an INT-indexed array and i an integer, $acc(a, i)$ represents the ith element of a while $upd(a, i, e)$ represents the array a where the ith element has been changed to an (α-sorted) element e.

Running example. We want to check whether the formula

$$acc(upd(k, 3, x), 3) + 2 = 7 \tag{3}$$

is satisfiable modulo the theory built as the union of $\{(1), (2)\}$ and a theory of linear arithmetic, where x (resp. k) has the sort INT (resp. INT ARRAY).

3 Reduction to Unsorted Logic

Unsorted FOL is a special case of polymorphic FOL where there is only one constant sort U (for Unsorted elements) and neither sort functions, built-in types nor type variables. We now present an encoding from a polymorphic logic with built-in types to an unsorted logic: the idea behind this encoding is to syntactically type expressions by adding the sort information directly in terms and rely on the fact that unification will guide the prover and ensure that instantiations remain well-typed. We also illustrate how to deal with interpreted symbols, as we consider the case of arithmetic terms.

3.1 Translation

In the polymorphic signature $\Sigma = (\mathcal{S}, \mathcal{F}_s, \mathcal{X}_s, \mathcal{X}, \mathcal{F})$ we translate, we assume $\mathcal{F}$ contains an *equality* predicate $=$ of arity $\alpha \times \alpha \to o$. We also consider a built-in sort $\text{INT} \in \mathcal{S}$ and $\mathcal{F}_\mathcal{A} \subset \mathcal{F}$ the set of arithmetic symbols (i.e. built-in operations and predicates on integers, but not $=$). The translation of signature Σ is the monosorted signature $\Sigma^* = (\{\text{U}\}, \emptyset, \emptyset, \mathcal{X}^*, \mathcal{F}^*)$ defined by:

- $\mathcal{X}^* = \mathcal{X} \cup \{t_\alpha \mid \alpha \in \mathcal{X}_s\}$ where the t_α are fresh term variables
- $\mathcal{F}^* = \mathcal{F} \cup \mathcal{S} \cup \mathcal{F}_s \cup \widehat{\mathcal{F}_\mathcal{A}} \cup \{\mathsf{sort}\}$ where this union is supposed disjoint and
  - symbols from $\mathcal{F}$ have the same arity as before where all sorts from Ω have been replaced by U (e.g. $cons : \text{U} \times \text{U} \to \text{U}$)
  - symbols from $\mathcal{S}$ have arity U
  - symbols from $\mathcal{F}_s$ with n parameters have arity $\underbrace{\text{U} \times \ldots \times \text{U}}_{n \text{ times}} \to \text{U}$
  - $\widehat{\mathcal{F}_\mathcal{A}} = \{\hat{f} \mid f \in \mathcal{F}_\mathcal{A}\}$ where each symbol $\hat{f}$ is fresh in $\mathcal{F}$, has the same arity as f and where INT has been replaced by U (e.g. $\hat{+} : \text{U} \times \text{U} \to \text{U}$)
  - the new special symbol sort has arity $\text{U} \times \text{U} \to \text{U}$

The idea behind this new function symbol sort is that it is used to associate a term with its type: the first argument is a syntactic sort and the second a term whose type is the first argument in the context. Unlike the P-predicate of Enderton ([9]), it is a function symbol and not a predicate symbol. Also, treating arithmetic symbols in a special way is mandatory to ensure completeness and is discussed below.

The translation $(.)^*$ of sorts from Ω into terms of Σ^* is inductively defined on the structure of a sort:

$$\begin{array}{rcl} s \in \mathcal{S} & \rightsquigarrow & s^* = s \in \mathcal{F}^* \\ \alpha \in \mathcal{X}_s & \rightsquigarrow & \alpha^* = t_\alpha \in \mathcal{X}^* \\ f(s_1, \ldots, s_n) \in \Omega & \rightsquigarrow & f(s_1^*, \ldots, s_n^*) \text{ a term on } \Sigma^*. \end{array}$$

The propositional structure of formulas is not affected by our encoding: merely, sorts in quantifiers are all replaced by U since our signature is monosorted, and we will omit these sorts hereafter. Thus, we are left to define the translation of an atom t. Since a translation of a term depends on its type and since a term may have many different types in a polymorphic logic, we have to be careful as to the choice of the type. To translate a well-typed formula, we consider the most general typing derivation of this formula: such a derivation always exists and is unique up to a renaming of type variables (see [11, p. 33 – 35]). A subterm t of the formula is then translated with respect to its type τ: in other words, the translation is inductively defined on the most-general typing derivation of the formula.

Since the well-typing rules are syntax-directed, the different cases can be described by the following rules, where τ is the type on the right-hand side of the sequent:

1. If t is a variable x, then $t^* = \mathsf{sort}(\tau^*, x)$
2. If t is a functional term $f(t_1, \ldots, t_n)$ and $f \in \mathcal{F}_\mathcal{A}$, $t^* = \mathsf{sort}(\text{INT}, \hat{f}(t_1^*, \ldots, t_n^*))$

3. If t is a functional term $f(t_1, \ldots, t_n)$ and $f \notin \mathcal{F}_\mathcal{A}$, $t^* = \mathsf{sort}(\tau^*, f(t_1^*, \ldots, t_n^*))$
4. If t is a predicative term $p(t_1, \ldots, t_n)$ and $p \in \mathcal{F}_\mathcal{A}$, $t^* = \hat{p}(t_1^*, \ldots, t_n^*)$
5. Otherwise, t is an atom $p(t_1, \ldots, t_n)$ with $p \notin \mathcal{F}_\mathcal{A}$ and $t^* = p(t_1^*, \ldots, t_n^*)$

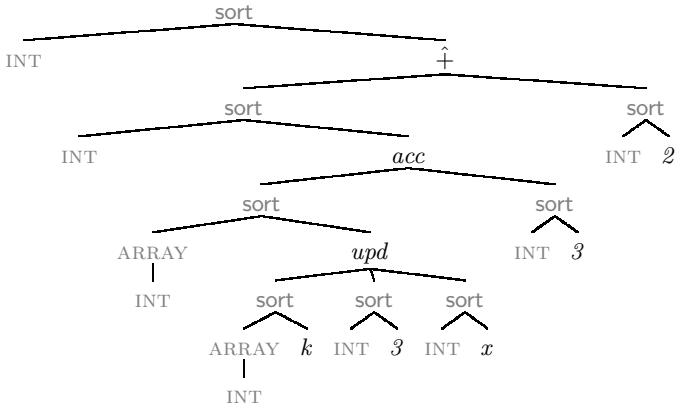

Fig. 2. Syntactic tree view of the translation of $acc(upd(k, 3, x), 3) + 2$

For instance, Fig. 2 represents the translation of $acc(upd(k, 3, x), 3) + 2$ as a syntactic tree. In that tree the term is globally unchanged, except that layers of sorts (in gray) have been introduced at each junction in the original syntactic tree, which is why we say this is a *stratified* encoding.

There are a couple of things to note about this translation. First, translated formulas are not necessarily closed because type variables that were implicitly universally quantified are now free term variables, and this can be accounted for by universally quantifying over these variables as well (which we will do in the next section when we describe the translation of a theory). Second, arithmetic expressions were encoded with new special symbols and are not interpreted expressions anymore, so we need axioms to define the meaning of these new symbols.

Each arithmetic symbol in $\mathcal{F}_\mathcal{A}$ is defined by one of the following axioms depending on whether it is a functional symbol f or a predicative symbol p:

$$\phi_f \equiv \forall x_1 \ldots x_n \,.\, \hat{f}(\mathsf{sort}(\text{INT}, x_1), \ldots, \mathsf{sort}(\text{INT}, x_n)) = f(x_1, \ldots, x_n)$$
$$\phi_p \equiv \forall x_1 \ldots x_n \,.\, \hat{p}(\mathsf{sort}(\text{INT}, x_1), \ldots, \mathsf{sort}(\text{INT}, x_n)) \Leftrightarrow p(x_1, \ldots, x_n).$$

They aim at stripping away type information from encoded arithmetic expressions. Simiarly, there is also a need for an instance of ϕ_p where p is the equality on integers, i.e.

$$\forall x_1, x_2 \,.\, \mathsf{sort}(\text{INT}, x_1) = \mathsf{sort}(\text{INT}, x_2) \Leftrightarrow x_1 = x_2 \tag{4}$$

This axiom corresponds to the injectivity of the function $x \mapsto \mathsf{sort}(\text{INT}, x)$. We write $\mathcal{P}$ (as *prelude*) for the set of all these axioms and we will later add this prelude to the solvers' context when trying to prove a translated formula.

The introduction of rules 2., 4. together with axioms ϕ_f and ϕ_p accounts for the completeness of the encoding: for arithmetic expressions, provers usually have dedicated decision procedure. Therefore, we need a logical link between encoded and original arithmetic expressions in order for a prover to be able to dispatch them to the decision procedure. This is not a solution specific to arithmetic : the same method could be used for other built-in types.

Without these axioms, a prover might for instance conclude that the formula $\mathsf{sort}(\text{INT}, 2) \geqslant \mathsf{sort}(\text{INT}, 3)$ is satisfiable whereas it is not. Another obvious alternative would be to add the axiom $\forall x \,.\, \mathsf{sort}(\text{INT}, x) = x$, but this results in a dramatic loss of efficiency for provers based on ground instantiation of quantifiers (which is currently the rule for SMT solvers), because they could instantiate this equality everytime an integer term is encountered, although there are many integers that do not require this because they are not applied on a symbol from $\mathcal{F}_{\mathcal{A}}$.

Running example. The translation of (3) is

$$\begin{array}{l} \mathsf{sort}(\text{INT}, \mathsf{sort}(\text{INT}, \\ \quad acc(\mathsf{sort}(\text{ARRAY}(\text{INT}), \\ \quad\quad upd(\mathsf{sort}(\text{ARRAY}(\text{INT}), k), \mathsf{sort}(\text{INT}, 3), \mathsf{sort}(\text{INT}, x)), \mathsf{sort}(\text{INT}, 3)))) \\ \quad \hat{+}\, \mathsf{sort}(\text{INT}, 2)) = \mathsf{sort}(\text{INT}, 7). \end{array} \tag{5}$$

By instantiating the counterpart of axiom (1), i.e.

$$\begin{array}{l} \forall \alpha, a, i, e \,. \\ \mathsf{sort}(\alpha, acc(\mathsf{sort}(\text{ARRAY}(\alpha), \\ \qquad\qquad upd(\mathsf{sort}(\text{ARRAY}(\alpha), a), \mathsf{sort}(\text{INT}, i), \mathsf{sort}(\alpha, e)), \mathsf{sort}(\text{INT}, i)))) \\ = \mathsf{sort}(\alpha, e) \end{array}$$

with $\alpha = \text{INT}$, $a = k$, $i = 3$ and $e = x$, the formula (5) simplifies to

$$\mathsf{sort}(\text{INT}, \mathsf{sort}(\text{INT}, x) \hat{+} \mathsf{sort}(\text{INT}, 2)) = \mathsf{sort}(\text{INT}, 7)$$

which is again reduced to $\mathsf{sort}(\text{INT}, x + 2) = \mathsf{sort}(\text{INT}, 7)$ thanks to the axiom ϕ_f and to $x + 2 = 7$ thanks to (4).

3.2 Translation Proofs

To prove results about our encoding, we use the natural deduction system for polymorphic FOL described in [13]. We slightly adapt this formalism by considering a theory $\mathcal{T} = (\Sigma, \mathcal{A}x, \Gamma)$ as the tuple of a signature Σ, a set of axioms $\mathcal{A}x$ and a typing context Γ. When ϕ is a monomorphic formula[1], the sequent $\mathcal{T} \models \phi$ expresses the fact that ϕ is a formula on Σ which is well-typed in Γ and valid given the axioms in $\mathcal{A}x$. Finally, axioms in $\mathcal{A}x$ are written $\forall \overline{\alpha}[P]$ where P is a formula and $\overline{\alpha}$ its polymorphic parameters. Figure 3 recalls rules of [13] which are used in the following proofs.

[1] Which is not a restriction, since a polymorphic goal can be monomorphized by replacing each type variable by a fresh uninterpreted sort, without loss of generality.

$$\frac{\mathcal{T} \vdash t : \tau}{\mathcal{T} \models t = t}(\mathsf{Eq}_1) \qquad \frac{\mathcal{T} \models x = y \quad \mathcal{T} \models Q[x/z]}{\mathcal{T} \models Q[y/z]}(\mathsf{Eq}_2) \quad \frac{\mathcal{T} \models P \wedge Q}{\mathcal{T} \models P}(\mathsf{And}_2)$$

$$\frac{\texttt{axiom } \forall \overline{\alpha}\,[P] \in \mathcal{T}}{\mathcal{T} \models \sigma(P)}(\mathsf{Ax}) \quad \frac{\mathcal{T} \models \forall x : \tau \,.\, P \quad \mathcal{T} \vdash t : \tau}{\mathcal{T} \models P[t/x]}(\mathsf{Forall}_2)$$

$$\frac{x_1, \ldots, x_n \models_A P}{\mathcal{T} \models \forall x_1 : \text{INT} \,.\, \ldots \forall x_n : \text{INT} \,.\, P}(\mathsf{Arith})$$

In rule (Ax), σ is a ground substitution from $\overline{\alpha}$ to sorts. In rule (Arith), the formula P is exclusively built with variables and constants of sort INT, symbols from $\mathcal{F}_{\mathcal{A}}$ and equality. If $x_1, \ldots, x_n$ are the free variables in P, we write $x_1, \ldots, x_n \models_A P$ as soon as P is valid (this is decidable, cf. for instance by the Omega test in [12]).

Fig. 3. Natural deduction system for polymorphic FOL (excerpt from [13])

The encoded version $\mathcal{T}^*$ of such a theory $\mathcal{T}$ is the tuple $(\Sigma^*, \mathcal{A}x^*, \Gamma^*)$ where:

- $\mathcal{A}x^* = \{[\forall t_{\alpha_1} : \text{U} \,.\, \ldots \forall t_{\alpha_k} : \text{U} \,.\, P^*] \,|\, \forall \alpha_1 \ldots \alpha_k [P] \in \mathcal{A}x\} \cup \mathcal{P}$
- Γ^* is Γ where all bindings are replaced by the unique sort U

Note that $\mathcal{A}x^*$ is made out of closed monomorphic encoded versions of the earlier axioms, and also contains the prelude we described in the previous section. Thanks to the axioms in this prelude, we can show the following results on arithmetic terms and their encoded versions:

Lemma 1 (Arithmetic terms). *Let t be an arithmetic term on Σ, that is, a term built exclusively with arithmetic symbols, integer constants and variables. Then $\mathcal{T}^* \models \mathsf{sort}(\text{INT}, t) = t^*$.*

Proof. The proof proceeds by structural induction on t. The case of a variable is obviously handled by the reflexivity rule (Eq_1). A functional term $f(t_1, \ldots, t_n)$ is treated by first applying the induction hypothesis to all the $t_i, 1 \leq i \leq n$, which transforms $\hat{f}(t_1^*, \ldots, t_n^*)$ to $\hat{f}(\mathsf{sort}(\text{INT}, t_1), \ldots, \mathsf{sort}(\text{INT}, t_n))$, and next by instantiating the additional axiom ϕ_f to transform this term to $f(t_1, \ldots, t_n)$, which is the desired result. □

Lemma 2 (Arithmetic predicates). *Let P be an arithmetic formula in a theory $\mathcal{T}$, that is, an atom built exclusively with arithmetic symbols, equality and arithmetic terms. Then, $\mathcal{T}^* \models P \Leftrightarrow P^*$.*

Proof. The proof is done by induction on the structure of a formula. The only non-trivial case is for an arithmetic atom $p(t_1, \ldots, t_n)$. By applying the previous lemma to all the $t_i, 1 \leq i \leq n$, we transform the encoded predicate $\hat{p}(t_1^*, \ldots, t_n^*)$ to $\hat{p}(\mathsf{sort}(\text{INT}, t_1), \ldots, \mathsf{sort}(\text{INT}, t_n))$, which is equivalent to $p(t_1, \ldots, t_n)$ by virtue of the added axiom ϕ_p. □

Theorem 1 (Completeness). *Let $\mathcal{T}$ be a theory and P a monomorphic formula on $\mathcal{T}$. Then, $\mathcal{T} \models P$ implies $\mathcal{T}^* \models P^*$.*

Proof. We proceed by structural induction on the proof derivation of $\mathcal{T} \models P$ and we only present non obvious cases.

Rule (Ax)*:* An application of the rule (Ax) can be translated in the following derivation where the application of the substitution σ is replaced by k applications of the rule (Forall_2), one for each type variable:

$$\dfrac{\dfrac{\texttt{axiom}\ \forall t_{\alpha_1} : \mathrm{U} \ldots \forall t_{\alpha_k} : \mathrm{U}\,.\, P^* \in \mathcal{T}^*}{\vdots}\,(\mathsf{Ax}) \qquad \mathcal{T}^* \vdash \sigma(\alpha_k)^* : \mathrm{U}}{\mathcal{T}^* \models P^*[\sigma(\alpha_1)^*/t_{\alpha_1}] \ldots [\sigma(\alpha_k)^*/t_{\alpha_k}]}\,(\mathsf{Forall}_2) \tag{6}$$

Rule (Forall_2)*:* By definition, we clearly have $(P[t/x])^* = P^*[e/x]$ where $t^* = \mathsf{sort}(\tau^*, e)$. This property makes the translation of the rules (Forall_2) straightforward:

$$\dfrac{\mathcal{T}^* \models \forall x : \mathrm{U}\,.\, P^* \qquad \mathcal{T}^* \vdash e : \mathrm{U} \quad \text{where} \quad t^* = \mathsf{sort}(\tau^*, e)}{\mathcal{T}^* \models P^*[e/x]}\,(\mathsf{Forall}_2)$$

Rule (Arith)*:* In this rule, P is only composed of arithmetic symbols (including integer constants) and integer variables. Therefore, P is true in the theory of arithmetic and is true in $\mathcal{T}^*$. The lemma 2 about the encoding of arithmetic atoms tells us that P is equivalent to P^* in $\mathcal{T}^*$, therefore it is obvious to build a proof for P^*. □

The soundness of the encoding is expressed by the fact that $\mathcal{T}^* \models P^*$ implies $\mathcal{T} \models P$.

Since we have erased type from quantifiers, a prover may *a priori* make ill-typed substitutions of the variables in translated formulae. Indeed, we emphasized in Sect. 2 that erasing types was not sound at all. Now, solvers do not instantiate a variable by *any* term but rather try to find an existing term that matches the context around the variable. Since our encoding ensures that the context around a variable contains its sort information, only terms with the same sort can be used for the instantiation. For instance, if a variable x was quantified over a sort τ in the original formula, every occurrence of x in the encoded formula is actually an occurrence of $\mathsf{sort}(\tau^*, x)$: when trying to intanstantiate x, solvers will *match* ground terms with that pattern and will only substitute x by a certain term e if the term $\mathsf{sort}(\tau^*, e)$ has already been encountered. If the ground terms have not been obtained by ill-typed instantiations (which is the case at first), this term e corresponds to a term t of type τ such that $t^* = \mathsf{sort}(\tau^*, e)$. The substitution of x by e in the encoded formula then corresponds to the substitution of x by t in the original formula, and this ensures that the property that all ground terms are well-formed is preserved when new instantiations are

made by the solver. A formalization of that *matching* process and a proof of that result are presented in [11, p. 53 – 58].

The conclusion of that discussion is that we can consider a restriction of the deduction system $\models$, later called $\models^*$, where instantiations of variables in rules for quantifiers are correct with respect to the types in the original formulas: if x appears as $\mathsf{sort}(\tau^*, x)$ in the formula, it can only be replaced with the translation of a ground term of sort τ, and otherwise it must be replaced with the translation τ^* of a ground sort τ. Therefore, our soundness theorem takes the following form:

Theorem 2 (Soundness). *Let $\mathcal{T}$ be a theory and P a monomorphic formula on $\mathcal{T}$. Then, $\mathcal{T}^* \models^* P^*$ implies $\mathcal{T} \models P$.*

Proof. We proceed by induction on a minimal proof derivation for $\mathcal{T}^* \models^* P^*$ but it is less straightforward than completeness. Indeed, consider elimination rules for logical connectives, like (And_2). When trying to reason by induction, we cannot apply the induction hypothesis to the second premise Q if it is not of the form Q_0^*. By taking a minimal proof, in particular without cuts, we ensure the connective $\wedge$ cannot come from an introduction rule, but only from an axiom in the theory, which means that the conjunction $P^* \wedge Q$ is a – possibly instantiated – subformula of an axiom in the theory. There are two cases to distinguish there: if the axiom comes from $\mathcal{P}$, the result is straightforward (either the connective does not appear in these axioms, or the proof of P is trivial); otherwise, the axiom comes from $\mathcal{A}x^*$ and since they are translations of axioms in $\mathcal{A}x$, $P^* \wedge Q$ is a subterm of an instantiation of a translated axiom, and that implies that Q has the expected form Q_0^*.

All introduction rules are straightforward and rely only on applying the induction hypothesis to all premises. We have also discussed elimination rules above. Among the remaining rules, we only detail the non obvious ones:

Rule (Forall_2)*:* We have $P^* = Q[t/x]$ and $\dfrac{\mathcal{T}^* \models^* \forall x : \mathrm{U}.Q \qquad \mathcal{T}^* \vdash t : \mathrm{U}}{\mathcal{T}^* \models^* Q[t/x]}(\mathsf{Forall}_2)$

As argued above, the universal quantifier comes from an axiom rule and corresponds either to a quantification on a type variable, or to a quantification on a variable x of type τ in the original formula. In the first case, P can be proved in $\mathcal{T}$ by replacing the instantiations of type variables by a type substitution in the premises of the rule (Ax). In the second case, $Q = Q_0^*$ and $P^* = (Q_0[u/x])^*$ for some u of type τ, hence the following derivation: $\dfrac{\mathcal{T} \models \forall x : \tau.Q_0 \qquad \mathcal{T} \vdash u : \tau}{\mathcal{T} \models Q_0[u/x]}$.

Rule (Eq_2)*:* Since the proof is minimal, the equality $x = y$ comes from an axiom and has the form $t^* = u^*$. Also, $Q[y/z]$ is a translated formula $P^* = (P_0[u/z])^*$, so $Q[x/z] = P_0[t/z]^*$ is also a translated formula and we can apply the induction hypothesis to create the following derivation $\dfrac{\mathcal{T} \models t = u \qquad \mathcal{T} \models P_0[t/z]}{\mathcal{T} \models P_0[u/z]}$.

4 Reduction to Many-Sorted Logic

In this section, we address the case of a prover that works in many-sorted logic. One may think that there is no need to do so since we have already presented an encoding to unsorted FOL and it is obvious that unsorted FOL is a special case of many-sorted FOL. Unfortunately, trying to use the previous encoding with a many-sorted prover raises the issue of how to type the new symbol sort and the built-in symbols $+$, $<$, etc. Indeed, built-in symbols and integer constants have the sort INT and cannot be considered as being of sort U. Taking $\text{U} = \text{INT}$ would be a solution, but this would not be easily adaptable to a case with several different built-in types and decision procedures (arithmetic + lists + ...).

Such a solution should not only translate polymorphic formulas to many-sorted logics in a generic way, but should also take advantage of the many-sorted logic to avoid to encode linear arithmetic as we did in the previous section, since it is a source of inefficiency. In the following, we present an encoding that solves this issue by only encoding integer expressions when they are used in a polymorphic context.

In addition to INT, there are three sorts S, T, U whose meaning is as follows: each term, except those of sort INT, is translated into a term $\mathsf{sort}(\_,\_)$ which has the sort S (for Syntactically sorted terms). The function sort has arity $\text{T} \times \text{U} \to \text{S}$ where T is the sort of Type and where U is the sort of terms not built with sort.

4.1 Translation

The polymorphic signature is translated into the monomorphic many-sorted signature $\Sigma^* = (\mathcal{S}^*, \emptyset, \emptyset, \mathcal{X}^*, \mathcal{F}^*)$ defined by:

- $\mathcal{S}^*$ is $\{\text{U}, \text{S}, \text{T}, \text{INT}\}$,
- $\mathcal{X}^*$ is $\mathcal{X} \cup \{t_\alpha \mid \alpha \in \mathcal{X}_s\}$ as in the unsorted case,
- $\mathcal{F}^*$ is $\mathcal{F} \cup \mathcal{F}_s \cup \{\mathsf{sort}\} \cup \mathcal{F}_{cast} \cup \mathcal{S} \setminus \{\text{INT}\}$ where
  - a symbol from $\mathcal{F}$ with arity $s_1 \times \ldots \times s_n \to s$ now has the arity $s'_1 \times \ldots \times s'_n \to s'$ where s' is INT if s is INT and U otherwise and where s'_i, $1 \leqslant i \leqslant n$, is INT if s_i is INT and S otherwise (e.g the function $upd : \alpha\ \text{ARRAY} \times \text{INT} \times \alpha \to \alpha\ \text{ARRAY}$ now has the arity $\text{S} \times \text{INT} \times \text{S} \to \text{U}$),
  - symbols from $\mathcal{F}_s$ with n parameters have arity $\underbrace{\text{T} \times \ldots \times \text{T}}_{n \text{ times}} \to \text{T}$,
  - $\mathcal{F}_{cast}$ is $\{int2u : \text{INT} \to \text{U}, s2int : \text{S} \to \text{INT}\}$,
  - each symbol of $\mathcal{S} \setminus \{\text{INT}\}$ is now a constant of sort T.

The translation t^* of a term t of type τ depends on the context.

1. If t is a predicate term $p(t_1, \ldots, t_n)$, $t^* = p(t_1^*, \ldots, t_n^*)$
2. if t is a functional term or a variable, then it is the ith parameter of some symbol g: $g(\ldots, t, \ldots)$. We call τ_i the type of the ith argument in g's arity and the translation depends on τ and τ_i being INT or not:
   (a) if neither τ nor τ_i are INT, then
   - if t is a variable x, $t^* = \mathsf{sort}(\tau^*, x)$

  - if t is a term $f(t_1, \ldots, t_n)$, $t^* = \mathsf{sort}(\tau^*, f(t_1^*, \ldots, t_n^*))$

(b) if τ and τ_i are INT,
  - if t is a variable x, $t^* = x$
  - if t is a term $f(t_1, \ldots, t_n)$, $t^* = f(t_1^*, \ldots, t_n^*)$

(c) if τ is not INT but τ_i is INT, then
  - if t is a variable x, $t^* = s2int(\mathsf{sort}(\tau^*, x))$
  - if t is a term $f(t_1, \ldots, t_n)$ then $t^* = s2int(\mathsf{sort}(\tau^*, f(t_1^*, \ldots, t_n^*)))$

(d) finally, if τ is INT but τ_i is not INT, then
  - if t is a variable x, $t^* = \mathsf{sort}(\text{INT}, int2u(\tau^*, x))$
  - if t is a term $f(t_1, \ldots, t_n)$ then $t^* = \mathsf{sort}(\text{INT}, int2u(f(t_1^*, \ldots, t_n^*))))$

It is straightforward to check that encoded expressions are still well-typed in Σ^*. Compared to the previous translation, exploiting sorts unburdens the terms sort INT from being encoded (rule 2b). However, rules 2c and 2d use casting functions $int2u$ and $s2int$ which have to be defined by the axioms

$$\forall x : \text{B}\,.\ s2int(\mathsf{sort}(\text{INT}, int2u(x))) = x \tag{7}$$

$$\forall x : \text{U}\,.\ int2u(s2int(\mathsf{sort}(\text{INT}, x))) = x. \tag{8}$$

In this section, the prelude is defined by $\mathcal{P} = \{(7), (8)\}$.

Running example. By encoding (3) we obtain

$$s2int(sort(\text{INT}, acc(sort(\text{ARRAY}(\text{INT}), upd(sort(\text{ARRAY}(\text{INT}), k), 3, sort(\text{INT}, int2u(x)))), 3))) + 2 = 7$$

which reduces to $s2int(\mathsf{sort}(\text{INT}, int2u(x))) + 2 = 7$, by applying the translation of the axiom (1) and to $x + 2 = 7$ thanks to the axiom (7).

4.2 Translation Proofs

We prove in this section the same theorems with the same notations as in Sect. 3.

Proof of Theorem 1. With this encoding, the proof is again achieved by structural induction on the proof derivation. The only non-obvious cases are the axiom rule and rules which infer a substitution: a term that appears in two hypotheses of an inference rule can generate two distinct translations according to its sort and its position in a logical symbol. Notice that unlike the proof detailed in Sect. 3.2, the arithmetic rule is obviously obtained since this encoding preserves pure arithmetic terms by definition.

An application of the rule (Ax) is replaced by a derivation similar to (6) where sort U is replaced by T.

For rules (Forall_2) and (Exists_1), if τ is not INT or if τ is INT and t^* has the sort INT, then the result is straightforward. Otherwise, τ is INT, $t^* = \mathsf{sort}(\text{INT}, e)$, and we have the following inference:

$$\dfrac{\dfrac{\vdots}{\mathcal{T}^* \models \forall x : \text{INT} \,.\, P^*} \qquad \dfrac{\dfrac{\vdots}{\Sigma^* \vdash \mathsf{sort}(\text{INT}, e) : \text{S}} \qquad s2int : \text{S} \to \text{INT}}{\Sigma^* \vdash s2int(t^*) : \text{INT}}\,(\mathsf{T}_2)}{\mathcal{T}^* \models P^*[s2int(\mathsf{sort}(\text{INT}, e))/x]}\,(\mathsf{Forall}_2)$$

However, if the generated predicate $P^*[s2int(\mathsf{sort}(\text{INT}, e))/x]$ contains some $int2u(s2int(\mathsf{sort}(\text{INT}, e)))$, it can be simplified by applying axiom (8) through rules (Forall_2) and (Eq_2).

For the rule (Eq_2), the proof is similar. If τ is not INT or if τ is INT and x^* has the sort INT, then the result is direct. Otherwise, τ is INT and the counterpart of $x = y$ is $\mathsf{sort}(\text{INT}, int2u(x^*)) = y^*$; it suffices to deduce the equality $s2int(\mathsf{sort}(\text{INT}, int2u(x))) = s2int(y^*)$ by applying (Eq_2) which is then simplified into $x^* = s2int(y^*)$ by applying axiom (7) through (Forall_2) and (Eq_2). □

Proof of Theorem 2. The proof, which relies on the same arguments than those of the proof of soundness in Sect. 3.2, is omitted. □

5 Experiments

All the encodings presented in this work are implemented as modules in the Why [14] tool. The benchmark we exploit has 1272 valid proof obligations (PO). They are built by the Caduceus tool [15] from 55 annotated C programs. They aim at certifying that each function implementation is consistent with its specification and that the program is free of null pointer dereferencing and out-of-bounds array access. To do so, it widely relies on linear arithmetic and polymorphism coming from the memory model axiomatization.

Results of benchmarks for several provers and several different encodings are summarized in the tables represented in Fig. 4. More precisely, Figure 4(a) shows experiments with the unsorted prover Simplify [3] whereas figure 4(b) shows experiments with many-sorted ones: Yices [6] and CVC-lite [7]. Rows labeled "Without encoding", "With typing predicates" and "With monomorphization" are the coarse ones detailed in Sect. 2 and "Stratified encoding" of the Fig. 4(a) and of the Fig. 4(b) correspond to encodings given in section 3 and 4 respectively.

Each prover got 10 seconds for each obligation, on an Intel Xeon 3.20GHz with 2Gb of Memory. The results represent the percentage of obligations proved valid within these 10 seconds. Also, the average time only takes proved obligations into account and not timeouts.

The first observation for unsorted provers is that the best results are obtained without any encoding (by just removing types), but we have seen that this method is unsound. Among the other encodings, differences are pretty important: the encoding with typing predicates is pretty poor (as we discussed in section 2) while our stratified encoding proves almost as many obligations as the prover without encoding.

Another remark is that the difference between the average times in benchmarks with and without encoding is really significant, whereas only a very small

		Simplify
Without encoding (no types at all)	Percentage	99.6%
	Avg Time	60ms
With typing predicates	Percentage	66.2%
	Avg Time	171ms
Stratified encoding	Percentage	98.8%
	Avg Time	243ms

(a) Unsorted Encodings

		Yices	CVCLite
With mono- -morphization	Percentage	49. %	36.9 %
	Avg Time	772ms	752ms
Stratified encoding	Percentage	99.1%	69.8%
	Avg Time	430ms	1312ms

(b) Many-Sorted Encodings

Fig. 4. Benchmarks of different encodings

percentage of the obligations cannot actually be proved within the 10 seconds. Of course, encoded formulas are slightly longer to prove because they are bigger and arithmetic requires more steps of instantiation than without encoding. But another possible and less obvious source of inefficiency (and even incompleteness) is that the encoding can disrupt other specificities of the inner mechanism of a prover, namely *triggers* (see for instance the description of Simplify triggers in [3]). Triggers are patterns used to trigger the instantiation of axioms in the theory, by matching them to existing ground terms. They can be declared by the user (e.g. Simplify) or automatically determined by some heuristics (in Simplify or Yices). In the latter case, nothing guarantees that the inferred triggers will be the same before and after encoding. For instance, an axiom like $\forall x : \mathrm{T} \,.\, pred(x)$ would have $pred(x)$ as unique trigger; the encoded version $\forall x \,.\, pred(\mathsf{sort}(\mathrm{T}, x))$ may have the trigger $\mathsf{sort}(\mathrm{T}, x)$ which would be instantiated much more often than the original since it is smaller and matches more terms than the bigger one, resulting in inefficiencies.

Concerning sorted provers, the monomorphization, does not terminate for 6 problems (corresponding to 625 POs). The percentage of proven obligations and the corresponding average time show that the stratified encoding allows Yices to discharge about twice as many proof obligations with the monomorphization. Furthermore, each of them is faster discharged than when encoded by the monomorphization. Note, though, that this encoding suffers from the same inherent inefficiencies than the first one (bigger theories, slower arithmetics, problems with triggers) but this is not visible on the benchmark because the monomorphized theories are already many times bigger than the original theories.

6 Related Works and Conclusion

Translating polymorphic logic to first order logic has been previously presented, but as far as we know, without considering dedicated cases for interpreted functional symbols as we do.

Encoding with predicates has been studied in [16,17]. A generic translation from Mizar Mathematical Library [18] into first order logic is studied in the former. In the latter, the authors show how they integrate a first-order intuitionistic prover into the Nuprl [19] proof assistant.

In [20], Joe Hurd proposes the `Metis` tactic to discharge subgoals from higher-order logic (HOL4) into first order proof procedures. This task is achieved by firstly omitting sort information and trying to find an untyped proof. Such a

proof is then checked to be typable by HOL. In the case of a failure, a complete translation is done, which corresponds to our unsorted stratified encoding without any optimization. Similarly, the authors of [21,22] address the task of proving higher-order formulas using first order resolution provers [23,24] and rely on Isabelle to check the proof's soundness.

In [25], L.C. Paulson introduces the Isabelle `Blast` tactic which integrates a tableau-based procedure. This procedure internally stores types of some symbols to avoid ill-sorted deductions whereas the approach we developped is applicable to a broad range of SMT solvers. In [26], the considered proof obligations are expressed in a specific many-sorted first order language, reduced to contain only two disjoint sorts. In such a context, the authors prove that sorts can be omitted without loosing soundness. This reduced context is not directly applicable here since we claim to present a generic approach.

To conclude, we have presented in this paper two close efficient reductions of polymorphism in unsorted logic and many-sorted logic. In both cases, the main idea is to syntactically type expressions by adding the sorts information directly into terms. Soundness is ensured by the way solvers produce ground instantiations of quantified variables, as argued in Sect. 3.2. The fact that our method depends on the intricacies of the solver's procedure is definitely a shortcoming. However, judging by the number of provers it applies to, we believe that the advantages of our method over other encodings, both performance-wise and flexibility-wise, sufficiently outweigh its shortcoming for this encoding to be of practical interest to others in the community.

In order to exploit as much as possible the decision procedures dedicated to linear arithmetic, the encodings also have special cases for arithmetic terms. Our large experiments show that the method is very efficient in the context of a representative panel of POs issued from software verification. An extension to several constant sorts should be straightforward, because we were as generic as possible. It is planned as a future work. That way, we could also benefit from decision procedures dedicated to other constant sorts (e.g. for reals, arrays, WS1S).

Acknowledgements

We are grateful to S. Conchon, E. Contejean, J.-C. Filliâtre, A. Giorgetti, C. Marché for stimulating discussions and suggestions on this topic, and to the anonymous referees for their comments which greatly helped to improve the final version of this paper.

References

1. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. In: Coq'Art: the Calculus of Inductive Constructions, Springer, Heidelberg (2004)
3. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a Theorem Prover for Program Checking. J. ACM 52(3), 365–473 (2005)
4. Déharbe, D., Ranise, S.: BDD-driven First-Oder Satisfiability Procedures (extended version). Technical Report 4630, LORIA (2002)

5. Ranise, S., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2006) http://www.SMT-LIB.org
6. Dutertre, B., de Moura, L.: The YICES SMT Solver (2006) avaliable at http://yices.csl.sri.com/tool-paper.pdf
7. Barrett, C.W., Berezin, S.: CVC Lite: A New Implementation of the Cooperating Validity Checker Category B. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 515–518. Springer, Heidelberg (2004)
8. Couchot, J.F., Lescuyer, S.: Handling Polymorphism in Automated Deduction (2007) Available at http://lri.fr/~couchot/ftp/publis/CL07t.ps
9. Enderton, H.B.: A Mathematical Introduction to Logic. Ac. Press, Inc. (1972)
10. Armando, A., Ranise, S., Rusinowitch, M.: A Rewriting Approach to Satisfiability Procedures. Journal of Information and computation 183, 140–164 (2003)
11. Lescuyer, S.: Codage de la logique du premier ordre polymorphe multi-sortée dans la logique sans sortes. Master's thesis (in english) pp. 28–58 (2006)
12. Pugh, W.: The Omega Test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM 35(8), 102–114 (1992)
13. Ayache, N., Filliâtre, J.C.: Combining the Coq Proof Assistant with First-Order Decision Procedures. Unpublished (March 2006)
14. Filliâtre, J.C.: Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud (March 2003)
15. Filliâtre, J.C., Marché, C.: Multi-Prover Verification of C Programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
16. Dahn, I.: Interpretation of a mizar-like logic in first-order logic. In: Selected Papers from Automated Deduction in Classical and Non-Classical Logics, pp. 137–151. Springer, London (2000)
17. Schmitt, S., Lorigo, L., Kreitz, C., Nogin, A.: Jprover: Integrating connection-based theorem proving into interactive proof assistants. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 421–426. Springer, Heidelberg (2001)
18. Rudnicki, P.: An overview of the Mizar project. In: Workshop on Types for Proofs and Programs, pp. 311–330 (1992)
19. Allen, S.F., Constable, R.L., Eaton, R., Kreitz, C., Lorigo, L.: The nuprl open logical environment. In: McAllester, D. (ed.) Automated Deduction - CADE-17. LNCS, vol. 1831, Springer, Heidelberg (2000)
20. Hurd, J.: First-Order Proof Tactics in Higher-Order Logic Theorem Provers. Technical Report NASA/CP-2003-212448, NASA (2003)
21. Meng, J., Paulson, L.C.: Translating Higher-Order Problems to First-Order Clauses. In: ESCoR (CEUR Workshop Proceedings), vol. 192, pp. 70–80 (2006)
22. Meng, J., Quigley, C., Paulson, L.C.: Automation for interactive proof: first prototype. Inf. Comput. 204(10), 1575–1596 (2006)
23. Schulz, S.: System Description: E 0.81. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 223–228. Springer, Heidelberg (2004)
24. Riazanov, A., Voronkov, A.: The design and Implementation of VAMPIRE. AI Commun. 15(2-3), 91–110 (2002)
25. Paulson, L.C.: A generic tableau prover and its integration with Isabelle. J.UCS: Journal of Universal Computer Science 5(3), 73 (1999)
26. Bouillaguet, C., Kuncak, V., Wies, T., Zee, K., Rinard, M.: Using first-order theorem provers in the Jahob data structure verification system. In: Verification, Model Checking and Abstract Interpretation. LNCS, pp. 74–88. Springer, Heidelberg (2007)