One Logic To Use Them All

Jean-Christophe Filliâtre

One Logic To Use Them All

Jean-Christophe Filliâtre

[1] CNRS
[2] LRI, Univ. Paris-Sud, Orsay, F-91405
[3] INRIA Saclay – Île-de-France, Orsay, F-91893

Abstract. Deductive program verification is making fast progress these days. One of the reasons is a tremendous improvement of theorem provers in the last two decades. This includes various kinds of automated theorem provers, such as ATP systems and SMT solvers, and interactive proof assistants. Yet most tools for program verification are built around a single theorem prover. Instead, we defend the idea that a collaborative use of several provers is a key to easier and faster verification.
This paper introduces a logic that is designed to target a wide set of theorem provers. It is an extension of first-order logic with polymorphism, algebraic data types, recursive definitions, and inductive predicates. It is implemented in the tool Why3, and has been successfully used in the verification of many non-trivial programs.

1 Introduction

The idea behind deductive program verification [20] is to break down the correctness of a program to a set of logical formulas and to prove them valid. A tremendous improvement of theorem provers in the last two decades now allows this idea to scale up. Projects such as CompCert [31] and seL4 [26] show how interactive proof assistants (resp. Coq and Isabelle) can be successfully used to tackle large program verifications. Even more impressive is the progress in automated provers[4], notably the so-called *SMT revolution*. Designed with program verification in mind, SMT solvers have led to powerful verification tools, such as VCC [39], Frama-C [23], Dafny [29], or VeriFast [25], to mention only a few. Yet we note that most of these tools are built on top of a single automated prover (*e.g.* Z3 in the case of VCC, Dafny, or VeriFast) or a single dedicated prover to handle a specific program logic (*e.g.* B method [1] or KIV [37]).

We rather defend the idea that program verification should be done using as many theorem provers as possible, including those that were not necessarily designed with program verification in mind (*e.g.* ATP systems). We are developing the tool Why3 [10,22] to implement this idea. Both the specification logic of Why3 and its programming language, WhyML, are used as intermediate languages in tools such as Frama-C [23], Krakatoa [33], Easycrypt [4], and GNATprove [14]. WhyML is also used directly to implement data structures and

[4] We use *automated provers* to denote both SMT solvers and ATP systems.

algorithms that can later be translated to executable OCaml code. Our gallery of verified programs (http://toccata.lri.fr/gallery/) currently contains more than 80 entries. These examples show the benefits of our approach. Indeed, it is often the case that several theorem provers are successfully used in proving all goals, but none can prove all of them by itself. In particular, an interactive theorem prover can be used to discharge a complex lemma (for instance one requiring induction), while the remaining goals can all be discharged automatically.

Designing a logic to target a wide set of theorem provers is not that easy. We have come up with a *logic of compromise*, that is not as rich as the logic of proof assistants such as Coq or PVS, yet richer than the usual logic of automated theorem provers. Our logic is an extension of first-order logic with rank-1 polymorphism, algebraic data types, recursive definitions, and inductive predicates. The purpose of this paper is to define this logic (Section 2), and to explain the processes by which Why3 translates it to the input format of fifteen theorem provers (Section 3). Finally, Section 4 presents experimental results obtained in the context of program verification. We conclude with related work and perspectives.

2 A Polymorphic First-Order Logic

This section describes the logic of Why3 as faithfully as possible. Earlier work [11] only describes the extension of first-order logic with polymorphism. Here we also consider algebraic data types, recursive definitions, and inductive predicates.

2.1 Syntax

A type symbol is simply a name $\mathtt{t}$ and a type arity $n \in \mathbb{N}$. We write it $\mathtt{t}\langle\alpha_1, \ldots, \alpha_n\rangle$ where the α_i are type variables. Names $\alpha_1, \ldots, \alpha_n$ are not relevant but we adopt an homogeneous syntax for all symbols. We note d_t such a declaration:

$$d_t \quad ::= \quad \mathtt{t}\langle\alpha, \ldots, \alpha\rangle \quad \text{type symbol declaration}$$

When $n = 0$, we say that $\mathtt{t}$ is a *monomorphic type symbol* and we simply write $\mathtt{t}$ instead of $\mathtt{t}\langle\rangle$. In the following, let Σ_T be a set of type symbols. We assume that Σ_T contains at least the two monomorphic type symbols int and real. Types are built from type variables and type symbols:

$$\begin{array}{rcll} \tau & ::= & \alpha & \text{type variable} \\ & | & \mathtt{t}\langle\tau, \ldots, \tau\rangle & \text{type symbol application} \end{array}$$

We note $tv(\tau)$ the set of type variables of type τ. When $tv(\tau) = \emptyset$, we say that τ is a *sort*.

Function and predicate symbols are declared, possibly with polymorphic types, as follows:

$$\begin{array}{rcll} d_f & ::= & \mathtt{f}\langle\alpha, \ldots, \alpha\rangle(\tau, \ldots, \tau) : \tau & \text{function symbol declaration} \\ d_p & ::= & \mathtt{p}\langle\alpha, \ldots, \alpha\rangle(\tau, \ldots, \tau) & \text{predicate symbol declaration} \end{array}$$

In the following, Σ_F denotes a set of function symbols and Σ_P a set of predicate symbols. We assume that Σ_P contains at least a polymorphic predicate $=\langle\alpha\rangle(\alpha,\alpha)$ that denotes equality. Terms and formulas are then built from function and predicate symbols according to the syntax given in Fig. 1–3. Note that syntax for terms and formulas are mutually recursive, since a conditional term expression `if` f `then` t_1 `else` t_2 involves a formula f.

We note $fv(t)$ (resp. $fv(f)$, $fv(p)$) the set of free variables of a term t (resp. a formula f, a pattern p). Definitions of $fv(t)$ and $fv(f)$ are standard, variables being bound by `let`, quantifiers, and patterns. A pattern p binds all variables in $fv(p)$, which is defined as follows:

$$\begin{aligned} fv(x_\tau) &= \{x_\tau\} \\ fv(\mathsf{f}\langle\tau_1,\ldots,\tau_m\rangle(p_1,\ldots,p_n)) &= fv(p_1)\cup\cdots\cup fv(p_n) \\ fv(\_) &= \emptyset \\ fv(p_1 \mid p_2) &= fv(p_1)\cup fv(p_2) \\ fv(p \texttt{ as } x_\tau) &= fv(p)\cup\{x_\tau\} \end{aligned}$$

The type checking rule for pattern $p_1 \mid p_2$, given in next section, imposes that $fv(p_1)$ and $fv(p_2)$ are equal. Thus the definition above actually simplifies to $fv(p_1 \mid p_2) = fv(p_1) = fv(p_2)$.

A *signature* Σ denotes a triple $(\Sigma_T, \Sigma_F, \Sigma_P)$. A *context* Γ extends a signature Σ with definitions for some of its symbols. A definition d introduces either algebraic data types, recursive definitions, or inductive predicates, as follows:

d	::=	`datatype` a `with` ... `with` a	algebraic data types
	|	`recursive` δ `with` ... `with` δ	recursive definitions
	|	`inductive` i `with` ... `with` i	inductive predicates
a	::=	d_t `=` $d_f \mid \ldots \mid d_f$	algebraic data type
δ	::=	`function` $\mathsf{f}\langle\alpha,\ldots,\alpha\rangle(x_\tau,\ldots,x_\tau) : \tau = t$	function definition
	|	`predicate` $\mathsf{p}\langle\alpha,\ldots,\alpha\rangle(x_\tau,\ldots,x_\tau) = f$	predicate definition
i	::=	d_p `=` $f \mid \ldots \mid f$	inductive predicate

Example. Let Σ_T be the following set of type symbols:

$$\Sigma_T = \{\mathsf{int}, \mathsf{real}, \mathsf{nat}, \mathsf{list}\langle\alpha\rangle, \mathsf{pair}\langle\alpha_1,\alpha_2\rangle\}.$$

Let Σ_F and Σ_P be the following sets of function and predicate symbols:

$$\begin{aligned} \Sigma_F = \{&\mathsf{O} : \mathsf{nat}, \mathsf{S}(\mathsf{nat}) : \mathsf{nat}, \\ &\mathsf{Nil}\langle\alpha\rangle : \mathsf{list}\langle\alpha\rangle, \mathsf{Cons}\langle\alpha\rangle(\alpha,\mathsf{list}\langle\alpha\rangle) : \mathsf{list}\langle\alpha\rangle, \mathsf{length}\langle\alpha\rangle(\mathsf{list}\langle\alpha\rangle) : \mathtt{nat}, \\ &\mathsf{Pair}\langle\alpha_1,\alpha_2\rangle(\alpha_1,\alpha_2) : \mathsf{pair}\langle\alpha_1,\alpha_2\rangle\,\} \\ \Sigma_P = \{&=\langle\alpha\rangle(\alpha,\alpha), \mathsf{even}(\mathsf{nat})\} \end{aligned}$$

t	::=	c_{int}	literal integer constant
	|	c_{real}	literal real constant
	|	x_τ	variable
	|	$\mathsf{f}\langle\tau,\ldots,\tau\rangle(t,\ldots,t)$	function symbol application
	|	$\mathtt{let}\ x_\tau = t\ \mathtt{in}\ t$	local binding
	|	$\mathtt{if}\ f\ \mathtt{then}\ t\ \mathtt{else}\ t$	conditional expression
	|	$\mathtt{match}\ t\ \mathtt{with}\ p \to t \mid \ldots \mid p \to t\ \mathtt{end}$	pattern matching

Fig. 1. Syntax for terms.

f	::=	$\mathsf{p}\langle\tau,\ldots,\tau\rangle(t,\ldots,t)$	predicate symbol application
	|	$\forall x_\tau.\, f$	universal quantification
	|	$\exists x_\tau.\, f$	existential quantification
	|	$f \wedge f$	conjunction
	|	$f \vee f$	disjunction
	|	$f \Rightarrow f$	implication
	|	$f \Leftrightarrow f$	equivalence
	|	$\mathtt{not}\ f$	negation
	|	$\mathtt{true}$	tautology
	|	$\mathtt{false}$	absurdity
	|	$\mathtt{let}\ x_\tau = t\ \mathtt{in}\ f$	local binding
	|	$\mathtt{if}\ f\ \mathtt{then}\ f\ \mathtt{else}\ f$	conditional expression
	|	$\mathtt{match}\ t\ \mathtt{with}\ p \to f \mid \ldots \mid p \to f\ \mathtt{end}$	pattern matching

Fig. 2. Syntax for formulas.

p	::=	x_τ	variable
	|	$\mathsf{f}\langle\tau,\ldots,\tau\rangle(p,\ldots,p)$	constructor application
	|	$\_$	catch all
	|	$p \mid p$	or pattern
	|	$p\ \mathtt{as}\ x_\tau$	binding

Fig. 3. Syntax for patterns.

Let Γ be the extension of the signature above with the following set of definitions:

$$\texttt{datatype } \mathsf{nat} = \mathsf{O} \mid \mathsf{S}(\mathsf{nat})$$
$$\texttt{datatype } \mathsf{list}\langle\alpha\rangle = \mathsf{Nil}\langle\alpha\rangle \mid \mathsf{Cons}\langle\alpha\rangle(\alpha, \mathsf{list}\langle\alpha\rangle)$$
$$\texttt{recursive function } \mathsf{length}\langle\alpha\rangle(l_{\mathsf{list}\langle\alpha\rangle}) : \mathsf{nat} =$$
$$\quad \texttt{match } l_{\mathsf{list}\langle\alpha\rangle} \texttt{ with } \mathsf{Nil}\langle\alpha\rangle \to \mathsf{O} \mid \mathsf{Cons}\langle\alpha\rangle(\_, r_{\mathsf{list}\langle\alpha\rangle}) \to \mathsf{S}(\mathsf{length}(r_{\mathsf{list}\langle\alpha\rangle})) \texttt{ end}$$
$$\texttt{inductive } \mathsf{even}(\mathsf{nat}) =$$
$$\quad \mathsf{even}(\mathsf{O}) \mid \forall n_{\mathsf{nat}}.\, \mathsf{even}(n_{\mathsf{nat}}) \Rightarrow \mathsf{even}(\mathsf{S}(\mathsf{S}(n_{\mathsf{nat}})))$$

Here and below, we omit type annotations when they are obvious from the context.

2.2 Type Checking

For a signature Σ to be well-formed, any type expression τ occurring in a declaration of Σ_F or Σ_P must be well-typed, that is $\Sigma \vdash \tau$. Additionally, the free type variables of τ must be included in the type variables $\langle\alpha_1, \ldots, \alpha_m\rangle$ of the declaration. In the following, we only consider signatures that are well-formed. Let Γ be a context extending a signature Σ with definitions. For Γ to be well-formed, any symbol from a definition must appear in Σ, and no symbol can be defined more that once.

Type checking of types, terms, formulas, and patterns is straightforward. Judgements are $\Sigma \vdash \tau$ (type τ is well-formed in Σ), $\Sigma \vdash t : \tau$ (term t is well-formed and of type τ), $\Sigma \vdash f$ (formula f is well-formed), and $\Sigma \vdash p : \tau$ (pattern p is well-formed and of type τ), respectively. Typing rules are given in Fig. 4–7. Note that, contrary to Hindley-Milner type system, the `let` binder does not generalize the type of the term that is bound. Thus polymorphism is introduced by symbols, but not by local definitions. In addition to these rules, we also check that, for any pattern matching expression $\texttt{match}\, t\, \texttt{with}\, p_1 \to . \mid \ldots \mid p_n \to .\, \texttt{end}$ (in terms or formulas), t has type $\mathsf{t}\langle\tau_1, \ldots, \tau_m\rangle$ where type t is defined as an algebraic data type and patterns $p_1, \ldots, p_n$ cover any possible value for term t.

Finally, we perform the following checks on definitions.

Algebraic Data Types. Within a block of mutually-defined algebraic data types $\texttt{datatype}\ a_1\ \texttt{with}\ \ldots\ \texttt{with}\ a_k$, each algebraic data type definition a_i must have the form

$$\begin{aligned} \mathsf{t}_i\langle\alpha_1, \ldots, \alpha_m\rangle \;=\; & \mathsf{f}_1\langle\alpha_1, \ldots, \alpha_m\rangle(\ldots) : \mathsf{t}_i\langle\alpha_1, \ldots, \alpha_m\rangle \\ \mid\; & \ldots \\ \mid\; & \mathsf{f}_l\langle\alpha_1, \ldots, \alpha_m\rangle(\ldots) : \mathsf{t}_i\langle\alpha_1, \ldots, \alpha_m\rangle \end{aligned}$$

that is, all constructors f_j share the same type parameters as type t_i and all have return type $\mathsf{t}_i\langle\alpha_1, \ldots, \alpha_m\rangle$. Additionally, we require that all types t_i are *inhabited* according to the following definition: Type t_i is inhabited if it has at least one constructor $\mathsf{f}_j\langle\alpha_1, \ldots, \alpha_m\rangle(\tau_1, \ldots, \tau_{n_j})$ such that all types τ_l are inhabited. Since types τ_l may recursively involve types from data type definitions, such a definition is to be understood as a least fixed point. Types from Σ that are not part of data type definitions are assumed to be inhabited. As a consequence of this definition, an algebraic data type must have at least one constructor.

$$\frac{\mathsf{t}\langle\alpha_1,\ldots,\alpha_m\rangle \in \Sigma \quad \Sigma \vdash \tau_i}{\Sigma \vdash \mathsf{t}\langle\tau_1,\ldots,\tau_m\rangle}$$

Fig. 4. Typing rule for types.

$$\frac{}{\Sigma \vdash c_{\mathsf{int}} : \mathsf{int}} \qquad \frac{}{\Sigma \vdash c_{\mathsf{real}} : \mathsf{real}} \qquad \frac{\Sigma \vdash \tau}{\Sigma \vdash x_\tau : \tau}$$

$$\frac{\begin{array}{c}\mathsf{f}\langle\alpha_1,\ldots,\alpha_m\rangle(\tau'_1,\ldots,\tau'_n) : \tau \in \Sigma \quad \Sigma \vdash \tau_i \\ \sigma = \{\alpha_1 \mapsto \tau_1,\ldots,\alpha_m \mapsto \tau_m\} \quad \Sigma \vdash t_i : \sigma(\tau'_i)\end{array}}{\Sigma \vdash \mathsf{f}\langle\tau_1,\ldots,\tau_m\rangle(t_1,\ldots,t_n) : \sigma(\tau)}$$

$$\frac{\Sigma \vdash t_1 : \tau \quad \Sigma \vdash t_2 : \tau_2}{\Sigma \vdash \mathtt{let}\ x_\tau = t_1\ \mathtt{in}\ t_2 : \tau_2} \qquad \frac{\Sigma \vdash f \quad \Sigma \vdash t_1 : \tau \quad \Sigma \vdash t_2 : \tau}{\Sigma \vdash \mathtt{if}\ f\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 : \tau}$$

$$\frac{\Sigma \vdash t : \mathsf{t}\langle\tau_1,\ldots,\tau_m\rangle \quad \Sigma \vdash p_i : \mathsf{t}\langle\tau_1,\ldots,\tau_m\rangle \quad \Sigma \vdash t_i : \tau}{\Sigma \vdash \mathtt{match}\ t\ \mathtt{with}\ p_1 \to t_1 \mid \ldots \mid p_n \to t_n\ \mathtt{end} : \tau}$$

Fig. 5. Typing rules for terms.

$$\frac{\begin{array}{c}\mathsf{p}\langle\alpha_1,\ldots,\alpha_m\rangle(\tau'_1,\ldots,\tau'_n) \in \Sigma \quad \Sigma \vdash \tau_i \\ \sigma = \{\alpha_1 \mapsto \tau_1,\ldots,\alpha_m \mapsto \tau_m\} \quad \Sigma \vdash t_i : \sigma(\tau'_i)\end{array}}{\Sigma \vdash \mathsf{p}\langle\tau_1,\ldots,\tau_m\rangle(t_1,\ldots,t_n)}$$

$$\frac{}{\Sigma \vdash \mathtt{true}} \qquad \frac{}{\Sigma \vdash \mathtt{false}} \qquad \frac{\Sigma \vdash f_1 \quad \Sigma \vdash f_2 \quad \circ \in \{\wedge,\vee,\Rightarrow,\Leftrightarrow\}}{\Sigma \vdash f_1 \circ f_2}$$

$$\frac{\Sigma \vdash \tau \quad \Sigma \vdash f}{\Sigma \vdash \forall x_\tau.\, f} \qquad \frac{\Sigma \vdash \tau \quad \Sigma \vdash f}{\Sigma \vdash \exists x_\tau.\, f} \qquad \frac{\Sigma \vdash f}{\Sigma \vdash \mathtt{not}\ f}$$

$$\frac{\Sigma \vdash t : \tau \quad \Sigma \vdash f}{\Sigma \vdash \mathtt{let}\ x_\tau = t\ \mathtt{in}\ f} \qquad \frac{\Sigma \vdash f_1 \quad \Sigma \vdash f_2 \quad \Sigma \vdash f_3}{\Sigma \vdash \mathtt{if}\ f_1\ \mathtt{then}\ f_2\ \mathtt{else}\ f_3}$$

$$\frac{\Sigma \vdash t : \mathsf{t}\langle\tau_1,\ldots,\tau_m\rangle \quad \Sigma \vdash p_i : \mathsf{t}\langle\tau_1,\ldots,\tau_m\rangle \quad \Sigma \vdash f_i}{\Sigma \vdash \mathtt{match}\ t\ \mathtt{with}\ p_1 \to f_1 \mid \ldots \mid p_n \to f_n\ \mathtt{end}}$$

Fig. 6. Typing rules for formulas.

$$\frac{\Sigma \vdash \tau}{\Sigma \vdash x_\tau : \tau} \qquad \frac{\Sigma \vdash \tau}{\Sigma \vdash \_ : \tau}$$

$$\frac{\begin{array}{c}f\langle\alpha_1,\ldots,\alpha_m\rangle(\tau'_1,\ldots,\tau'_n) : \tau \in \Sigma \quad \Sigma \vdash \tau_i \quad \sigma = \{\alpha_1 \mapsto \tau_1,\ldots,\alpha_m \mapsto \tau_m\} \\ i \neq j \Rightarrow fv(p_i) \cap fv(p_j) = \emptyset \quad \Sigma \vdash p_i : \sigma(\tau'_i)\end{array}}{\Sigma \vdash f\langle\tau_1,\ldots,\tau_m\rangle(p_1,\ldots,p_n) : \sigma(\tau)}$$

$$\frac{fv(p_1) = fv(p_2) \quad \Sigma \vdash p_i : \tau}{\Sigma \vdash p_1 \mid p_2 : \tau} \qquad \frac{x_\tau \notin fv(p) \quad \Sigma \vdash p : \tau}{\Sigma \vdash p\ \mathtt{as}\ x_\tau : \tau}$$

Fig. 7. Typing rules for patterns.

Recursive Definitions. A block of mutually recursive definitions `recursive` δ_1 `with` ... `with` δ_k is well-formed if each definition δ_i is well-typed. A function definition

$$\texttt{function}\ \mathsf{f}\langle\alpha,\ldots,\alpha\rangle(x_\tau,\ldots,x_\tau) : \tau = t$$

is well-typed if $\Sigma \vdash t : \tau$, and a predicate definition

$$\texttt{predicate}\ \mathsf{p}\langle\alpha,\ldots,\alpha\rangle(x_\tau,\ldots,x_\tau) = f$$

is well-typed if $\Sigma \vdash f$. Additionally, all free variables (including type variables) in t and f must belong to the symbol declaration. Finally, we require such definitions to *terminate*, according to the following criterion: there must exist a lexicographic order of arguments that guarantees a structural descent (over algebraic data types).

Inductive Predicates. An inductive definition

$$\mathsf{p}\langle\alpha_1,\ldots,\alpha_m\rangle(\tau_1,\ldots,\tau_n) = f_1 \mid \ldots \mid f_k$$

is well-formed if each clause f_i is a closed formula, well-typed *i.e.* $\Sigma \vdash f_i$, that belongs to the following grammar:

$$\begin{array}{rcl} f_0 & ::= & \mathsf{p}\langle\alpha_1,\ldots,\alpha_m\rangle(t_1,\ldots,t_n) \\ & \mid & f \Rightarrow f_0 \\ & \mid & \forall x_\tau.\, f_0 \\ & \mid & \texttt{let}\ x_\tau = t\ \texttt{in}\ f_0 \end{array}$$

Additionally, we require all inductive predicates from that block of inductive definitions to appear in strictly positive positions in the formulas on the left side of $\Rightarrow$, so as to ensure the existence of a least fixpoint.

In the following, we only consider contexts that are well-formed according to the typing rules we just introduced.

2.3 Semantics

We recall that a sort is a monomorphic type, that is, a type τ such that $tv(\tau) = \emptyset$. In the following, we use notation $\boldsymbol{\alpha}$ (resp. $\boldsymbol{s}$, $\boldsymbol{x}$, $\boldsymbol{t}$) for a vector of type variables (resp. sorts, variables, terms). Given a signature, a *pre-interpretation* $[\![.]\!]$ is defined as follows:

- Each sort s is interpreted as a non-empty domain $[\![s]\!]$. Sort int is interpreted as $\mathbb{Z}$ and sort real as $\mathbb{R}$.
- Given a function symbol $\mathsf{f}\langle\boldsymbol{\alpha}\rangle(\tau_1,\ldots,\tau_n) : \tau$ and sorts $\boldsymbol{s}$, we interpret the instance $\mathsf{f}\langle\boldsymbol{s}\rangle$ as a function $[\![\mathsf{f}\langle\boldsymbol{s}\rangle]\!]$ of type

  $$[\![\sigma(\tau_1)]\!] \times \cdots \times [\![\sigma(\tau_n)]\!] \to [\![\sigma(\tau)]\!]$$

  where σ maps the type variables $\boldsymbol{\alpha}$ to the sorts $\boldsymbol{s}$.

– Given a predicate symbol $\mathsf{p}\langle\boldsymbol{\alpha}\rangle(\tau_1,\ldots,\tau_n)$ and sorts $\boldsymbol{s}$, we interpret the instance $\mathsf{p}\langle\boldsymbol{s}\rangle$ as a function $[\![\mathsf{p}\langle\boldsymbol{s}\rangle]\!]$ of type

$$[\![\sigma(\tau_1)]\!]\times\cdots\times[\![\sigma(\tau_n)]\!]\to\{\bot,\top\}$$

where σ maps the type variables $\boldsymbol{\alpha}$ to the sorts $\boldsymbol{s}$. For each sort s, the predicate $=\langle s\rangle$ is interpreted as the equality over $[\![s]\!]$.
– For any algebraic data type $\mathsf{t}\langle\boldsymbol{\alpha}\rangle$, with constructors $\mathsf{f}_1\langle\boldsymbol{\alpha}\rangle,\ldots,\mathsf{f}_l\langle\boldsymbol{\alpha}\rangle$, we require $[\![\mathsf{t}\langle\boldsymbol{s}\rangle]\!]$ to be the free algebra generated by $[\![\mathsf{f}_1\langle\boldsymbol{s}\rangle]\!],\ldots,[\![\mathsf{f}_l\langle\boldsymbol{s}\rangle]\!]$, that is

$$\text{for } i\neq j, [\![\mathsf{f}_i\langle\boldsymbol{s}\rangle]\!](\boldsymbol{t})\neq[\![\mathsf{f}_j\langle\boldsymbol{s}\rangle]\!](\boldsymbol{u}) \tag{1}$$

$$[\![\mathsf{f}_i\langle\boldsymbol{s}\rangle]\!](\boldsymbol{t})=[\![\mathsf{f}_i\langle\boldsymbol{s}\rangle]\!](\boldsymbol{u})\Rightarrow\boldsymbol{t}=\boldsymbol{u} \tag{2}$$

$$\forall x\in[\![\mathsf{t}\langle\boldsymbol{s}\rangle]\!],\ \exists i,\exists\boldsymbol{t},\ x=[\![\mathsf{f}_i\langle\boldsymbol{s}\rangle]\!](\boldsymbol{t}) \tag{3}$$

In the following, isf_i denotes a predicate that identifies values in $[\![\mathsf{t}\langle\boldsymbol{s}\rangle]\!]$ that are applications of f_i, and $\mathsf{projf}_{i,j}$ returns the j-th argument of such an application.

Given a pre-interpretation, a *valuation* v maps each type variable α to a sort $v(\alpha)$, and each variable x_τ to some element of $[\![v(\tau)]\!]$. Given a pre-interpretation and a valuation v, a term t of type τ is interpreted as an element $[\![t]\!]_v\in[\![v(\tau)]\!]$ and a formula f is interpreted as a Boolean value $[\![f]\!]_v$, according to the definitions given in Fig. 8–9. Note that pattern matching is compiled away as show in Fig. 10. Operator M turns a `match` construct into elementary tests. More precisely, $M(t,p,b,h)$ filters value t against pattern p and returns b in case of success and h in case of failure. Since type checking ensures that any pattern matching is exhaustive, the *error* term cannot actually appear in the result of M. It is only used to simplify the definition and use of function M.

An *interpretation* is a pre-interpretation that is consistent with recursive and inductive definitions, that is:

– For any recursive definition $\texttt{function}\ \mathsf{f}\langle\boldsymbol{\alpha}\rangle(\boldsymbol{x}):\tau=t$ and any $\boldsymbol{s}$, we require $[\![\mathsf{f}\langle\boldsymbol{s}\rangle]\!]$ to be such that, for all $\boldsymbol{t}$, $[\![\mathsf{f}\langle\boldsymbol{s}\rangle]\!](\boldsymbol{t})=[\![t]\!]_v$ where v maps the $\boldsymbol{\alpha}$ to the $\boldsymbol{s}$ and the $\boldsymbol{x}$ to the $\boldsymbol{t}$ (and similarly for a predicate definition).
– For any inductive definition $\mathsf{p}\langle\boldsymbol{\alpha}\rangle(\boldsymbol{\tau})=f_1\mid\ldots\mid f_l$ and any $\boldsymbol{s}$, we require $[\![\mathsf{p}\langle\boldsymbol{s}\rangle]\!]$ to be the least predicate such that $[\![f_1]\!]_v,\ldots,[\![f_l]\!]_v$ hold where v maps the $\boldsymbol{\alpha}$ to the $\boldsymbol{s}$.

A set of closed formulas Δ is *satisfied* by $[\![.]\!]$ if and only if $[\![f]\!]_v=\top$ for every $f\in\Delta$ and every valuation v. (Note that only the mapping of type variables in v is relevant here, since formulas are closed.) A set of closed formulas Δ is *satisfiable* if and only if it is satisfied for some interpretation. Given a set of polymorphic closed axioms Δ and a closed formula f to be proved, we may assume that f is monomorphic (otherwise we simply replace type variables in f by fresh type constants). Then we say that formula f is a *logical consequence* of Δ if and only if the set $\Delta,\texttt{not}\ f$ is unsatisfiable.

$$
\begin{aligned}
[\![n_{\mathsf{int}}]\!]_v &= n \\
[\![r_{\mathsf{real}}]\!]_v &= r \\
[\![x_\tau]\!]_v &= v(x_\tau) \\
[\![\mathsf{f}\langle\tau_1,\ldots,\tau_m\rangle(t_1,\ldots,t_n)]\!]_v &= [\![\mathsf{f}\langle v(\tau_1),\ldots,v(\tau_m)\rangle]\!]([\![t_1]\!]_v,\ldots,[\![t_n]\!]_v) \\
[\![\texttt{let } x_\tau = t_1 \texttt{ in } t_2]\!]_v &= [\![t_2]\!]_{v[x_\tau \mapsto [\![t_1]\!]_v]} \\
[\![\texttt{if } f \texttt{ then } t_1 \texttt{ else } t_2]\!]_v &= [\![t_1]\!]_v \text{ if } [\![f]\!]_v = \top, \\
&= [\![t_2]\!]_v \text{ otherwise} \\
[\![\texttt{match } t \texttt{ with } p_1 \to t_1 \mid \ldots \mid p_n \to t_n \texttt{ end}]\!]_v &= [\![M(t,p_1,t_1,M(t,p_2,t_2, \\
&\qquad \ldots,M(t,p_n,t_n,\mathit{error})))]\!]_v
\end{aligned}
$$

Fig. 8. Interpretation of terms.

$$
\begin{aligned}
[\![\mathsf{p}\langle\tau_1,\ldots,\tau_m\rangle(t_1,\ldots,t_n)]\!]_v &= [\![\mathsf{p}\langle v(\tau_1),\ldots,v(\tau_m)\rangle]\!]([\![t_1]\!]_v,\ldots,[\![t_n]\!]_v) \\
[\![\forall x_\tau.\, f]\!]_v &= \\
[\![\exists x_\tau.\, f]\!]_v &= \\
[\![f_1 \circ f_2]\!]_v &= [\![f_1]\!]_v \circ [\![f_2]\!]_v \text{ where } \circ \in \{\wedge,\vee,\Rightarrow,\Leftrightarrow\} \\
[\![\texttt{not } f]\!]_v &= \neg[\![f]\!]_v \\
[\![\texttt{true}]\!]_v &= \top \\
[\![\texttt{false}]\!]_v &= \bot \\
[\![\texttt{let } x_\tau = t_1 \texttt{ in } f_2]\!]_v &= [\![f_2]\!]_{v[x_\tau \mapsto [\![t_1]\!]_v]} \\
[\![\texttt{if } f_1 \texttt{ then } f_2 \texttt{ else } f_3]\!]_v &= [\![f_2]\!]_v \text{ if } [\![f_1]\!]_v = \top, \\
&= [\![f_3]\!]_v \text{ otherwise} \\
[\![\texttt{match } t \texttt{ with } p_1 \to f_1 \mid \ldots \mid p_n \to f_n \texttt{ end}]\!]_v &= [\![M(t,p_1,f_1,M(t,p_2,f_2, \\
&\qquad \ldots,M(t,p_n,f_n,\mathit{error})))]\!]_v
\end{aligned}
$$

Fig. 9. Interpretation of formulas.

$$
\begin{aligned}
M(t,x_\tau,b,h) &= \texttt{let } x_\tau = t \texttt{ in } b \\
M(t,\mathsf{f}_i(),b,h) &= \texttt{if } \mathsf{isf}_i(t) \texttt{ then } b \texttt{ else } h \\
M(t,\mathsf{f}_i(p_1,\ldots,p_n),b,h) &= \texttt{if } \mathsf{isf}_i(t) \texttt{ then } M(\mathsf{projf}_{i,1}(t),p_1,M(\mathsf{projf}_{i,2}(t),p_2,\ldots,h),h) \texttt{ else } h \\
M(t,\_,b,h) &= b \\
M(t,p_1 \mid p_2,b,h) &= M(t,p_1,b,M(t,p_2,b,h)) \\
M(t,p \texttt{ as } x_\tau,b,h) &= \texttt{let } x_\tau = t \texttt{ in } M(t,p,b,h)
\end{aligned}
$$

Fig. 10. Interpretation of patterns.

Discussion. It is worth pointing out that our notion of interpretation does not prevent two distinct instances of a polymorphic symbol, say two types $\mathsf{t}\langle s_1\rangle$ and $\mathsf{t}\langle s_2\rangle$ or two functions $\mathsf{f}\langle s_1\rangle$ and $\mathsf{f}\langle s_2\rangle$, to be interpreted in two completely different ways. Even for an algebraic data type, say $\texttt{list}\langle\alpha\rangle$, we do not require $\mathsf{Nil}\langle s_1\rangle$ and $\mathsf{Nil}\langle s_2\rangle$ to be identical. We simply require $\texttt{list}\langle s_1\rangle$ to be the free algebra generated by $\mathsf{Nil}\langle s_1\rangle$ and $\mathsf{Cons}\langle s_1\rangle$, and similarly $\texttt{list}\langle s_2\rangle$ to be the free algebra generated by $\mathsf{Nil}\langle s_2\rangle$ and $\mathsf{Cons}\langle s_2\rangle$.

As a consequence, there is nothing wrong with an axiom defining a property of $\mathsf{f}\langle\mathsf{int}\rangle$, for some polymorphic function $\mathsf{f}\langle\alpha\rangle$, while all other instances are left uninterpreted. Even further, we could have two completely unrelated axioms for $\mathsf{f}\langle\mathsf{int}\rangle$ and $\mathsf{f}\langle\mathsf{real}\rangle$, *e.g.* that $\mathsf{f}\langle\mathsf{int}\rangle$ is the identity over type int while $\mathsf{f}\langle\mathsf{real}\rangle$ is the square root function.

Simply speaking, everything works as if we were using many-sorted logic with a possibly infinite set of simple sorts (int, $\mathsf{list}\langle\mathsf{int}\rangle$, $\mathsf{list}\langle\mathsf{real}\rangle$, $\mathsf{list}\langle\mathsf{list}\langle\mathsf{int}\rangle\rangle$, etc.), and a possibly infinite set of functions and predicates with simple types ($\mathsf{Nil}\langle\mathsf{int}\rangle$, $\mathsf{Nil}\langle\mathsf{real}\rangle$, $\mathsf{Cons}\langle\mathsf{list}\langle\mathsf{int}\rangle\rangle$, etc.), with suitable extensions for algebraic data types, recursive definitions, and inductive predicates.

2.4 Implementation

The logic we just described is implemented in Why3 at two different levels: an OCaml API and a surface language.

The OCaml API allows the user to build terms, patterns, formulas, declarations, and goals. The API is defensive: only well-typed values can be built, according to the typing rules of Sec. 2.2. Following the presentation above, one first builds symbols and then, later, possible definitions for these symbols. This way, terms, patterns, and formulas can be built as soon as symbols are available, without the need of passing around a context containing definitions.

On top of this API, Why3 provides a surface language. Fig. 11 contains an example of input file. Contrary to the API, when a definition is provided, we do not separate the signature and the definition. For instance, we simply write

```
type list 'a = Nil | Cons 'a (list 'a)
```

to simultaneously introduce the type symbol `list 'a`, the two function symbols `Nil` and `Cons` (its constructors), and the algebraic data type definition.

In the surface language, there are no angle brackets $\langle.\rangle$ anymore. First, type variables are not explicitly bound in symbol declarations. They are rather gathered, as the set of all type variables appearing in the symbol's type. For instance, the declaration

```
function length (l: list 'a) : nat = ...
```

introduces a function symbol `length` with one type variable. Second, type variables are not explicitly instantiated when a symbol is used. Instantiation is inferred from the arguments whenever possible. For instance, one simply writes

```
length (Cons 1 (Cons 2 Nil))
```

```
theory Example

  type nat = O | S nat

  type list 'a = Nil | Cons 'a (list 'a)

  function length (l: list 'a) : nat =
    match l with
    | Nil → O
    | Cons _ r → S (length r)
    end

  inductive even (nat) =
    | EvenO: even O
    | EvenS: forall m: nat. even m → even (S (S m))

  goal G: even (length (Cons O (Cons O Nil)))

end
```

Fig. 11. Example of Why3 input file.

without having to pass ⟨int⟩ to `Nil`, `Cons`, and `length`. When the instantiation cannot be computed from the arguments, it must be provided. A typical example is the following:

```
  lemma L: length (Nil: list 'a) = O
```

Without the type cast, this would result in an `undefined type variable` error. In both the API and the surface language, there is no type inference, only type checking. In particular, quantified variables are given types (such as `forall m: nat` in Fig. 11). This is a deliberate choice: we could implement type inference, but we think that specifications are easier to read when types are given.

Among other features of the surface language of Why3 are type aliases (such as `type t = list int`) and tuple types. Type aliases are inlined systematically. Tuple types are particular cases of algebraic data types, and are generated on the fly. There is also a syntax for record types (algebraic data types with a single constructor and named projection functions). Finally, logic declarations are organized in units called *theories*, as shown in Fig. 11. Theories can refer to and instantiate other theories. We do not discuss theories here; see [10] for details.

3 Targeting Multiple Provers

We now explain how Why3 translates the logic we just described into the native input format of external theorem provers. Currently, Why3 supports the following theorem provers:

- Proof assistants: Coq 8.4 [41] and PVS 6.0 [36];
- SMT solvers: Alt-Ergo 0.95.1 [7], CVC3 2.4.1 [3], CVC4 1.0 [2], Simplify 1.5.4 [19], Yices 1.0.38 [18] and Yices2 2.0.4, Z3 4.3.1 [17];
- ATP systems: E 1.6 [40], iProver 0.8.1 [27], SPASS 3.7 [42], Vampire 0.6 [38], Zenon 0.7.1 [13];
- Dedicated provers: Gappa 0.15.1 [16], Mathematica 8.0.

We only list the most recent versions that are supported; some older versions may be supported as well.

3.1 Tasks and Transformations

A central notion in Why3 is that of *task*. A task is a context Γ (symbols, possibly with definitions), a set of axioms Δ, and a formula to be proved. Tasks are massaged using *transformations* until they reach a subset of Why3's logic that coincides with the input format of a theorem prover. A transformation turns a task into a set of new tasks. Key transformations are the following:

- elimination of recursion, inductive predicates, algebraic data types and pattern matching, `if-then-else` construct, `let` binding;
- encoding of types, to target many-sorted or untyped logic [15,11].

In the process of proving a task, other transformations may be used, such as inlining, splitting, various kinds of simplification, or even induction (following Leino [30]). Why3 can be extended with new transformations via OCaml plugins. The OCaml API allows the user to build tasks and to apply transformations efficiently (thanks to memoization).

3.2 Drivers

The way a particular prover is handled in Why3 is controlled by a text file called a *driver*. Such a file lists the transformations that must be applied to a task prior to its transmission to the prover, defines the pretty-printer that must be used when we are done with transformations, lists symbols and axioms that are built-in in the prover, and provides regular expressions to interpret the output of calls to the prover.

For instance, the driver for SPASS refers to the printer registered under the name `tptp-fof`, includes transformations such as `eliminate_algebraic` (to get rid of algebraic data types and pattern matching) and `encoding_tptp` (to encode types), and states that equality is built-in with syntax `(`x`=`y`)`, among other things.

Users may define new drivers, to add support for a new prover or to experiment with alternative ways of using a prover (*e.g.* not making use of a built-in theory, or using an alternate input format). For that purpose, Why3 can be extended with new pretty-printers via OCaml plug-ins. The OCaml API provides ways to load a driver and then to use it to call the corresponding theorem prover on a task. Drivers are described in more details in our earlier work [10].

Prover Specificities. There are several places where we have to be careful to avoid introducing inconsistencies in the process of translating from Why3 to theorem provers.

An example is integer division. Why3 standard library contains two theories: one for Euclidean division, and another where division rounds towards zero as in most programming languages. When it comes to using a built-in notion of division provided by some prover, we have to identify which one it is, if any, or to provide workarounds otherwise, if possible. For instance, Z3 provides Euclidean division and modulo, but CVC4 provides a division that appears to be none of the two divisions from Why3 standard library.

Another example is the fact that all types in Why3 are inhabited. Automated theorem provers typically make that assumption already. When it comes to proof assistants Coq and PVS, however, it is not granted for free. PVS provides a built-in mechanism for non-empty types, that we use readily. Coq, on the contrary, does not provide any such facility. We resort to type classes to ensure that all types we manipulate are inhabited. For instance, function `length` is translated into

```
Fixpoint length {a:Type} {a_WT:WhyType a} (l:(list a)) ...
```

where `WhyType a` is a type class that states that type `a` is inhabited and has decidable equality.

3.3 Proof Sessions

When a verification tool is built on top of a single automated prover, it is straightforward to replay a proof: one simply reruns the tool on the input files. The same applies for a proof that has been built interactively, say with Coq or KIV, and stored into files: it can be rechecked easily, in batch mode.

With Why3, the situation is somewhat different. Using the graphical user interface `why3ide`, the user interactively applies transformations and calls external theorem provers, until all goals are discharged. Calls to provers record the maximal amount of CPU time and memory that is allocated to the prover. Calls to proof assistants record user-edited proof scripts. All that information is stored into a set of files called a *proof session*. This way, it can be reloaded on a subsequent call to `why3ide` or replayed in a batch mode with another tool, `why3replayer`. This is of particular interest when the user upgrades the version of one or several provers. A run of `why3replayer` then tells whether the proof session is still valid or must be updated. Proof sessions are the subject of another publication [9].

3.4 Examples

We now give some examples of the way goals are translated to be passed to theorem provers. Let us consider our running example in Fig. 11 and let us assume that we are targeting the SMT solver Alt-Ergo.

Algebraic Data Types. We must get rid of algebraic data types. Thus, `list 'a` is simply turned into some uninterpreted data type, together with uninterpreted function symbols for constructors

```
type 'a list
logic Nil : 'a list
logic Cons : 'a, 'a list → 'a list
```

and axioms (not shown here) to state that `Nil` and `Cons` are distinct and that `Cons` is injective. To get rid of pattern matching on type `list`, we also introduce a function symbol

```
logic match_list : 'a list, 'a1, 'a1 → 'a1
```

together with axioms stating that $\texttt{match\_list}(\texttt{Nil}, a, b) = a$ and that $\texttt{match\_list}(\texttt{Cons}(x, y), a, b) = b$. We proceed similarly for type `nat`.

Recursive Definitions. We must also get rid of the recursive definition of function `length`. It is turned into some uninterpreted function symbol together with two axioms:

```
logic length : 'a list → nat
axiom length_def : length(Nil : 'a list) = 0
axiom length_def1 : forall x:'a. forall x1:'a list.
  length(Cons(x, x1)) = S(length(x1))
```

Note that we are not making use of function `match_list` here, since it immediately reduces to simpler axioms.

Inductive Definitions. Finally, we must get rid of the inductive definition of predicate `even`. It is axiomatized as follows:

```
logic even : nat → prop
axiom Even0 : even(0)
axiom EvenS : forall m:nat. even(m) → even(S(S(m)))
axiom even_inversion :
  forall z:nat. even(z) →
    (z = 0 or exists m:nat. even(m) and z = S(S(m)))
```

This is incomplete, since we cannot capture the minimality of `even` in first-order logic. For instance, Alt-Ergo cannot prove the goal `forall x: nat. even x → not (even (S x))`, while we could use Coq to prove it. Anyway, Alt-Ergo easily manages to prove the goal

```
goal G: even(length(Cons(0, Cons(0, (Nil : nat list)))))
```

with suitable instantiations of lemmas `length_def`, `length_def1`, `Even0`, and `EvenS`.

prover	proved	max. time	avg. time
CVC3 (2.4.1)	2203	21.00	0.17
Alt-Ergo (0.95.1)	2202	29.73	0.16
CVC4 (1.0)	2071	12.00	0.09
Z3 (4.3.1)	1869	45.52	0.11
Yices (1.0.38)	1634	4.30	0.05
Vampire (0.6)	1375	27.72	0.51
E (1.6)	1303	19.73	0.41
Spass (3.7)	1185	23.78	0.52

Table 1. Comparing eight automated provers using Why3.

Types. As we see on the example above, we did not have to encode types. Indeed, Alt-Ergo supports polymorphic types [8]. If we are instead targeting a prover that only supports simple types (*e.g.* Z3) or some untyped logic (*e.g.* SPASS), we have to encode the polymorphic types of Why3 in some way or another. For instance, on the following Why3 input

```
type list 'a
constant nil: list 'a
function length (list 'a) : int
axiom length_nil: length (nil: list 'a) = 0
goal G: length (nil: list int) = 0
```

the file that is passed to SPASS is the following:

```
fof(length_nil, axiom,
![A]: sort(int, length(sort(list(A),nil))) = sort(int,const_0)).

fof(g, conjecture,
sort(int, length(sort(list(int), nil))) = sort(int,const_0)).
```

Here `sort` is a binary function symbol that wraps terms with types. Types themselves are represented as regular terms, such as constant `int` or variable `A` above. The case of an SMT solver is more subtle, as we need to protect built-in types — such as integers, arrays, or reals — if we want to use built-in decision procedures [11].

4 Experimental Results

In this section, we use Why3 to run a small benchmark of 8 different automated provers. The experiment uses 83 proof sessions from our gallery, corresponding to logical theories or programs that were all successfully proved. This includes our solutions to several recent competitions in program verification (VSTTE 2012, FoVeOOS 2011, VSTTE 2012, FM 2012). The total number of subgoals is 2849.

For each subgoal that was discharged by at least one prover, we run the other 7 provers on that subgoal, with the same limit of CPU time that was given to the

first one. Provers are run on an 8-cores 3.20GHz Intel Xeon with 24Gb of RAM. Each prover runs on a separate core, with a limit of 1Gb of memory. Results are given in Table 1. For each prover, we give the total number of goals proved, and the maximum/average running time per goal.

The purpose of that table is not really to compare provers, but rather to show the benefits of a collaborative use of several provers: if we were using CVC3 only, we would be left with 646 ($= 2849 - 2203$) unproved subgoals. Besides, it is worth pointing out that most of these goals involve arithmetic; yet provers with no support for arithmetic (E, SPASS, and Vampire, in that case) are able to discharge a large subset of the goals. This was rather unexpected and encourages us to increase our daily use of these provers and to improve the way we use them even further.

5 Related Work

There is actually very little in the literature regarding the extension of first-order logic with polymorphism. For instance, a classical textbook such as Manzano's [32] does not contain a single occurrence of the word 'polymorphism'. One the other side of the logical spectrum, rich logics such as the Calculus of Inductive Constructions or HOL do have polymorphism (even beyond rank-1) but are seldom interested in identifying their first-order fragments. Among the recent work on this subject, we can mention the work by Leino and Rümmer on Boogie 2's type system [28] and the definition of the TPTP TFF1 format by Blanchette and Paskevich [6].

There is more related work regarding the second part of this paper, as a lot has been done in the context of Isabelle's Sledgehammer tactic [34,12]. It translates Isabelle's logic to several external provers, using type encodings different from ours, ranging from mere type erasure (which is unsound, but Sledgehammer uses proof reconstruction[5]) to partial monomorphisation [5] (which is proved incomplete [11] but seems efficient in practice anyway). Earlier, Hurd had already investigated encodings from higher-order to first-order logic [24].

6 Conclusion and Perspectives

We have presented a logic whose purpose is to provide a unified front-end to many existing theorem provers, either interactive or automated, and its implementation in Why3. We have designed this logic with the idea that specification must be as natural as possible and that tools should adapt themselves to proof obligations (and not the opposite). Using a wide range of theorem provers encourages this attitude.

One of the key features of our logic is polymorphism. It is defined and handled roughly the same way it is in tools Boogie, Sledgehammer, and Alt-Ergo. There

[5] A significant difference between Sledgehammer and our work is that we do not perform any kind of proof reconstruction. Thus we have to keep to sound encodings.

is no competition, but rather a nice convergence. A contribution of this paper is to add the formalization of algebraic data types, recursive definitions, and inductive predicates on top of that.

Currently, Why3 is successfully used as a sub-component in various projects [23,33,4,14], as well as the vehicle for many non-trivial case studies in program verification (see for instance [21]). We also envision that verification environments that are currently built on top of a dedicated prover (*e.g.* B, KIV, SmallFoot) could also benefit from additional, external theorem provers. For instance, we are currently experimenting with the use of Why3 to discharge goals obtained from Atelier B [35], and the first results are promising.

We could still improve the way we are using theorem provers. For instance, we can observe that some tools that are based on a single automated prover — *e.g.* VCC, Dafny, or VeriFast — are able to carry out impressive case studies. It is clear that these tools are achieving a level of intimacy with the prover that is beyond that of Why3. We should learn from these tools and transpose relevant techniques to Why3. In particular, we our support of built-in theories.

Of course, it would be much simpler if we had native support for polymorphic types in provers. Alt-Ergo demonstrates that this is possible, and even simple to implement [8], yet this is currently the only automated prover with such a feature. We hope that TFF1 [6] will encourage some ATP developers to take the plunge.

Acknowledgments. I would like to thank Andrei Paskevich and Sylvain Conchon for fruitful discussions during the preparation of this paper. Some ideas behind the formalization in Sec. 2 already appear in Bobot and Paskevich's work [11]. The development of Why3 is joint work with François Bobot, Claude Marché, Guillaume Melquiond, and Andrei Paskevich.

References

1. J.-R. Abrial. *The B-Book, assigning programs to meaning.* Cambridge University Press, 1996.
2. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.
3. C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302, Berlin, Germany, July 2007. Springer.
4. G. Barthe, B. Grégoire, S. Heraud, and S. Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.
5. J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. In N. Piterman and S. A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 493–507. Springer, 2013.

6. J. C. Blanchette and A. Paskevich. TFF1: The TPTP typed first-order form with rank-1 polymorphism. In *24th International Conference on Automated Deduction (CADE-24)*, Lake Placid, USA, June 2013. Springer.
7. F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The Alt-Ergo automated theorem prover, 2008. http://alt-ergo.lri.fr/.
8. F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing Polymorphism in SMT solvers. In C. Barrett and L. de Moura, editors, *SMT 2008: 6th International Workshop on Satisfiability Modulo*, volume 367 of *ACM International Conference Proceedings Series*, pages 1–5, 2008.
9. F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. Preserving user proofs across specification changes. In E. Cohen and A. Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments (5th International Conference VSTTE)*, Lecture Notes in Computer Science, Atherton, USA, May 2013. Springer.
10. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.
11. F. Bobot and A. Paskevich. Expressing Polymorphic Types in a Many-Sorted Language. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 87–102, Saarbrücken, Germany, Oct. 2011.
12. S. Böhme and T. Nipkow. Sledgehammer: Judgement day. In J. Giesl and R. Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010.
13. R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *LPAR*, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007.
14. C. Comar, J. Kanig, and Y. Moy. Integrating formal program verification with testing. In *Proceedings of the Embedded Real Time Software and Systems conference*, ERTS^2 2012, Feb. 2012.
15. J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In *21th International Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNCS (LNAI)*, pages 263–278, Bremen, Germany, July 2007.
16. M. Daumas and G. Melquiond. Certification of bounds on expressions involving rounded operators. *Transactions on Mathematical Software*, 37(1):1–20, 2010.
17. L. de Moura and N. Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
18. L. de Moura and B. Dutertre. Yices: An SMT Solver. http://yices.csl.sri.com/.
19. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52:365–473, May 2005.
20. J.-C. Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(5):397–403, Aug. 2011.
21. J.-C. Filliâtre. Verifying two lines of C with Why3: an exercise in program verification. In R. Joshi, P. Müller, and A. Podelski, editors, *Verified Software: Theories, Tools, Experiments (4th International Conference VSTTE)*, volume 7152 of *Lecture Notes in Computer Science*, pages 83–97, Philadelphia, USA, Jan. 2012. Springer.
22. J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium*

on Programming, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.
23. The Frama-C platform for static analysis of C programs, 2008. http://www.frama-c.cea.fr/.
24. J. Hurd. An lcf-style interface between hol and first-order logic. In A. Voronkov, editor, *CADE 2002*, number 2392 in LNCS, pages 134–138. Springer, 2002.
25. B. Jacobs and F. Piessens. The VeriFast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven, August 2008.
26. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, June 2010.
27. K. Korovin. iProver – an instantiation-based theorem prover for first-order logic (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning, (IJCAR 2008)*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008.
28. K. Leino and P. P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In J. Esparza and R. Majumdar, editors, *TACAS 2010*, volume 6015 of *LNCS*, pages 312–327. Springer, 2010.
29. K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Springer, editor, *LPAR-16*, volume 6355, pages 348–370, 2010.
30. K. R. M. Leino. Automating induction with an SMT solver. In *Proc. 13th Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, Philadelphia, PA, 2012.
31. X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
32. M. Manzano. *Extensions of first order logic*. Cambridge University Press, New York, NY, USA, 1996.
33. C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. http://krakatoa.lri.fr.
34. J. Meng and L. C. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40:35–60, 2008.
35. D. Mentré, C. Marché, J.-C. Filliâtre, and M. Asuka. Discharging proof obligations from Atelier B using multiple automated provers. In S. Reeves and E. Riccobene, editors, *ABZ'2012 - 3rd International Conference on Abstract State Machines, Alloy, B and Z*, volume 7316 of *Lecture Notes in Computer Science*, pages 238–251, Pisa, Italy, June 2012. Springer. http://hal.inria.fr/hal-00681781/en/.
36. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga Springs, NY, June 1992. Springer.
37. W. Reif, G. Schnellhorn, and K. Stenzel. Proving system correctness with KIV 3.0. In W. McCune, editor, *14th International Conference on Automated Deduction*, Lecture Notes in Computer Science, pages 69–72, Townsville, North Queensland, Australia, july 1997. Springer.
38. A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 292–296, Trento, Italy, July 1999. Springer.

39. W. Schulte, S. Xia, J. Smans, and F. Piessens. A glimpse of a verifying C compiler. http://www.cs.ru.nl/~tews/cv07/cv07-smans.pdf.
40. S. Schulz. System description: E 0.81. In D. A. Basin and M. Rusinowitch, editors, *Second International Joint Conference on Automated Reasoning*, volume 3097 of *Lecture Notes in Computer Science*, pages 223–228. Springer, 2004.
41. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.3*, 2010. http://coq.inria.fr.
42. C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In R. A. Schmidt, editor, *22nd International Conference on Automated Deduction*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.