

Qed. Computing what remains to be proved

Loïc Correnson

▶ **To cite this version:**

Loïc Correnson. Qed. Computing what remains to be proved. NFM 2014 - NASA Formal Methods, 6th International Symposium, Apr 2014, Houston, United States. pp.215-229, 10.1007/978-3-319-06200-6_17 . cea-01809013

HAL Id: cea-01809013
https://cea.hal.science/cea-01809013

Submitted on 8 Apr 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Qed.
Computing what Remains to be Proved

Loïc Correnson

CEA, LIST, Software Safety Laboratory
PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

Abstract. We propose a framework for manipulating in a efficient way terms and formulæ in classical logic modulo theories. Qed was initially designed for the generation of proof obligations of a weakest-precondition engine for C programs inside the Frama-C framework, but it has been implemented as an independent library. Key features of Qed include on-the-fly strong normalization with various theories and maximal sharing of terms in memory. Qed is also equipped with an extensible simplification engine. We illustrate the power of our framework by the implementation of non-trivial simplifications inside the Wp plug-in of Frama-C. These optimizations have been used to prove industrial, critical embedded softwares.

1 Introduction

In the context of formal verification of critical softwares, the recent fantastic improvement of automated theorem provers and SMT solvers[1] opens new routes. Inside the Frama-C [2] platform, we have developed the Wp plug-in to implement an efficient *weakest precondition calculus* to formally prove a C program against its specification. The specification is written in terms of the "ANSI-C Specification Language" (ACSL) [3], which is a first-order logic system with dedicated constructs to express C properties such as pointer validity and floating point operations.

The Wp plug-in actually compile C and ACSL constructs into an internal logic representation that is finally exported to SMT solvers and other theorem provers. Thus, we need an internal system to represent and manipulate first-order logical formulæ. This is exactly what Qed has been designed for.

Designing such a library is not difficult in itself. Some datatype is needed for expressing terms and properties, combined with pretty-printing facilities to export them into several languages. This is what we implemented in our early prototypes.

However, experimental results shown that a formula can not be naively build then translated and finally sent to an external back-end prover. We actually observed limitations of such a naive approach on real life examples from critical embedded software:

– SMT solvers are quite efficient, but they are sensitive to the amount of hypotheses they receive. Having a proof for $A \to B$ does not mean you will have a proof for $A \wedge A' \to B$.
– The generated formulæ are huge and deep. Without extra precautions, you often face an exponential blow-up when dumping them to disk.
– On the contrary, few transformations of the generated formulæ reduce their size and complexity in a dramatic way.

These reasons drive us in the direction of designing a dedicated system for representing and simplifying formulæ in an efficient way. We use classical techniques inspired by *preprocessing* optimizations found in various SMT solvers. However, in practice, it is not possible to rely on external preprocessors. One reason is that not all SMT solvers are equipped with such techniques. But most importantly, without on-the-fly preprocessing, the generation of proof obligations simply doesn't terminate in practice.

Moreover, applying these preprocessing facilities on-the-fly allows for non-trivial optimizations during the *weakest precondition calculus* [4], by pruning out useless branches for instance. Moreover, it allows for *domain specific preprocessing*: we designed Qed to be equipped with an extensible simplification engine, and we made it available to the end-user of Wp [5, §2.3.10].

This paper is first (§ 2) a tour and a formal presentation of the Qed framework, as a pure first-order logic system equipped with built-in theories for equality, arithmetic, arrays, records and unspecified functions. Second (§ 3), we illustrate how Qed improves in a very significant way the results of Wp plug-in inside Frama-C. We finally conclude with future research directions.

2 The Qed Engine

Our logical framework allows for defining and manipulating formulæ in first-order classical logic modulo theories. The key concept that drives the design of Qed is to implement only *fast* and *non-local* simplifications. This is of course incomplete, but more complex resolution techniques are left for back-end solvers.

The framework actually consists in three parts: a formally defined algebra of *term normal forms*, a collection of *smart constructors* to build *terms*, and an extensible *simplification engine*. The three components are tiedly coupled with each others.

The framework is implemented as an OCaml library, with additional features for exporting Qed formulæ to foreign systems, like Coq [6], Alt-Ergo [7] and Why-3 [8]. The efficiency of the framework relies (although not only) on a compact representation of *terms* into memory. Especially, *hash-consing* [9] is used to maximize memory sharing of equal terms. Hence, we benefit from constant-time equality and hashing over terms. Moreover, hash-consing allows for the identification of each term by an unique integer. This can be used to implement sets and maps of terms based on Patricia-trees [10], which provides the end-user of Qed with $O(n)$ unions, intersections and merges instead of the usual $O(n \log n)$ ones.

We do not present the implementation details in this article. The code is freely available under open source license together with the Frama-C distribution.[1]

In this paper, we present formally the three coupled components of the Qed framework and how they work with each others. We first introduce the internal representation of formulæ, the *term* algebra. Then, we define the *smart constructors* for building *terms*, with the associated normalization algorithms. Finally, we present the extensible *simplification engine*.

2.1 Terms Algebra

The internal representation of *terms* consists in an inductive datatype quotiented by normalization invariants. The Qed *smart constructors* are then especially designed to enforce those invariants.

The datatype of *terms* ($a \in \mathcal{L}$) is presented in Figure 1. It is parametrized by datatypes for the symbols identifying variable names ($x \in \mathcal{X}$), record's field names ($\texttt{f} \in \texttt{Fd}$) and user-defined or unspecified functions ($f \in \mathcal{F}$). The notation $\overline{a}$ stands for finite lists of terms, that is, $\overline{a} = a_1, \ldots, a_n$ for some $n \geq 0$.

$a \in \mathcal{L}$ ::=

Equality	*Quantifiers*	*Functions*
$\mid a = a \mid a \neq a$	$\mid x \mid \forall x.a \mid \exists x.a$	$\mid f(\overline{a})$

Logic

$\mid \mathsf{true} \mid \mathsf{false}$	Booleans
$\mid \wedge\overline{a} \mid \vee\overline{a}$	Conjunction, Disjunction
$\mid \wedge\overline{a} \rightarrow a$	Implication
$\mid \neg a$	Negation
$\mid a \,?\, a : a$	If-then-else

Arithmetic

$\mid k \in \mathbf{Z} \mid q \in \mathbf{Q}$	Constants
$\mid a \leq a \mid a < a$	Inequalities
$\mid k.a \mid \Sigma\,\overline{a} \mid \Pi\,\overline{a}$	Factors, Sums & Products

Arrays & Records

$\mid a[a]$	Access
$\mid a[a \mapsto a]$	Updates
$\mid a.\texttt{f}$	Field Access
$\mid \{\texttt{f} \mapsto a\,; \ldots\}$	Records

Fig. 1. Qed Terms Algebra

[1] From http://frama-c.com/download/frama-c-Fluorine-20130601.tar.gz, Qed sources are provided in the self-contained sub-directory `src/wp/qed`.

In the flow of the text, we would write Qed formulæ within quotes, like $\ll a \leq b \gg$, to distinguish the *terms* from their *semantics*. For instance, we must read $0 < x$ as the usual math property that x is positive, and $\ll 0 < x \gg$ as a term in $\mathcal{L}$ where variable $\ll x \gg$ is compared to zero. Conversely, we denote by $[\![a]\!]$ the semantics in usual mathematics of formulæ $\ll a \gg$.

We assume all symbols to be equipped with total orders such that there is an induced total order $a \preceq b$. The constant-time structural equality $a \equiv b$ on terms is provided by hash-consing. To summarize our notations:

$\ll a \gg$	Term in $\mathcal{L}$	$[\![a]\!]$	Semantics of $a \in \mathcal{L}$
$a \preceq b$	Total order	$a \equiv b$	Structural (and physical) equality

The strict order $a \prec b$ defined by $(a \preceq b \wedge a \not\equiv b)$ is also used. For maintaining the normalization invariants of terms, we introduce $\mathsf{ac}(\overline{a})$ and $\mathsf{ac}^*(\overline{a})$ to denote non-empty sorted lists with or without repetitions:

$$\begin{array}{rcl} \mathsf{ac}(a_1 \dots a_n) & \Leftrightarrow & 0 < n \wedge \forall i, j \in 1..n,\ i < j \Rightarrow a_i \prec a_j \\ \mathsf{ac}^*(a_1 \dots a_n) & \Leftrightarrow & 0 < n \wedge \forall i, j \in 1..n,\ i < j \Rightarrow a_i \preceq a_j \end{array}$$

We now investigate the various normal forms of $a \in \mathcal{L}$ and the associated invariants.

Equality. Terms $\ll a = b \gg$ and $\ll a \neq b \gg$ are quotiented by $a \prec b$ and $[\![a]\!] \neq [\![b]\!]$ when the built-in theories of Qed applies. For instance, $\ll 1 = 2 \gg$ is *not* a Qed normal term.

Quantifiers. Terms $\ll \forall x.a \gg$ and $\ll \exists x.a \gg$ can only be formed if x appears free in a. Structural equality ($\equiv$) in $\mathcal{L}$ is *not* quotiented by α-conversion. This is a choice we made because in practice such equalities are rare and α-conversion can be costly [11]. For instance, using De-Bruijn indices requires lambda liftings [12] which are *not* local transformations.

Logic. Boolean connectives are n-ary operators quotiented with ac^* arguments. Moreover, there are never two-arguments a and b of logical connectives such that we can decide $[\![a]\!] \Leftrightarrow \neg [\![b]\!]$ with Qed. Moreover, there is no duplication of *boolean* term operators and logical connectives for *properties* as usual in first-order logic. Rather, we use a two-sorted typing system to recover this distinction when it is required, for instance, to send a Qed formula to a SMT-solver.

Arithmetic. We choose n-ary sums and products operators quotiented by ac arguments. Linear forms are maximally flattened and factorized. For instance, it is not possible to have formula $\ll 1 - x \leq x - y \gg$, but we would have $\ll y < 2.x \gg$ instead (provided x and y are integers). These operators apply to both integer and real values, which case can be disambiguated by typing when necessary.

Arrays. The theory of functional arrays [13] is built-in in Qed. Access-updates are reduced whenever equality can be decided with Qed. Hence, $\ll m[a \mapsto v][b] \gg$ is reduced into $\ll m[b] \gg$ or $\ll v \gg$ whenever $[\![a = b]\!]$ can be decided.

Records. The theory of records is built-in in Qed. We do not choose to represent field-update terms, since they can always be represented by extensive reconstruction of the record. This choice makes the computation of normal forms for records more local.

Unspecified Functions. We decided to never inline a *definition* of a function symbol $f \in \mathcal{F}$. Although, this can be done using the extensible simplification engine. However, function symbols $f \in \mathcal{F}$ can be attributed with *algebraic* properties, such as injectivity, commutativity, associativity, neutral elements and such. This leads to many normalizations and simplifications that will be discussed with the associated *smart constructors*.

2.2 Smart Constructors and Normalizations

To build formulæ with the Qed framework, one must use the provided *smart constructors* listed in Figure 2. Thus, it is not possible to forge arbitrary terms $a \in \mathcal{L}$ that would violate the expected invariants. Moreover, since all the simplifications in the framework are *local*, we always obtain fully normalized terms on-the-fly.

Equality

eq :	$a, a \to a$
neq :	$a, a \to a$

Variables

var :	$x \to a$
forall :	$x, a \to a$
exists :	$x, a \to a$

Functions

call :	$f, \overline{a} \to a$

Arithmetic

int :	$\mathbf{Z} \to a$
real :	$\mathbf{Q} \to a$
add :	$a, a \to a$
sub :	$a, a \to a$
times :	$\mathbf{Z}, a \to a$
mul :	$a, a \to a$
leq :	$a, a \to a$
lt :	$a, a \to a$

Arrays

get :	$a, a \to a$
set :	$a, a, a \to a$

Logic

true :	a
false :	a
not :	$a \to a$
and :	$a, a \to a$
or :	$a, a \to a$
imply :	$a, a \to a$
equiv :	$a, a \to a$
ite :	$a, a, a \to a$

Records

field :	$a, \mathtt{f} \to a$
record :	$(\mathtt{f}_i, a_i)_i \to a$

Fig. 2. Qed Smart Constructors (API)

In this section, we investigate the normalizations computed by the smart constructors of Qed framework. We first discuss boolean normalizations and arithmetic ones. Then, functions, arrays and records will be discussed in turn. Each theory $\mathcal{T}$ will define smart constructors $\mathsf{eq}_\mathcal{T}$ and $\mathsf{neq}_\mathcal{T}$ for equalities, which will be finally merged together into the smart constructor for equality on the entire algebra $\mathcal{L}$.

Logic. The normalization of logical connectives is based on list of literals packed with their negation, like $(a, \neg a)$. Equipped with a suitable order, such a representation allows for fast detection of a and $\neg a$ among arguments of logical connectives. This leads to frequent calls to the smart constructor $\mathsf{not}(a)$ and, in the OCaml implementation, we use a cache to amortize this cost.

We use recursive definitions to extract list of literals from terms. But thanks to invariants in the term algebra, it is always limited at 2-depth recursive calls. We also use an exception (denoted by $\bot_{\mathsf{Absorbing}}$) to handle absorbing elements. This leads to the following flattening accumulative functions (in Haskell flavor):

$$\begin{array}{l|l}
\mathsf{lit}_\vee \ \langle\!\langle \ \vee \overline{a} \ \rangle\!\rangle \ l = \mathsf{fold}\ \mathsf{lit}_\vee\ \overline{a}\ l & \mathsf{lit}_\wedge \ \langle\!\langle \ \wedge \overline{a} \ \rangle\!\rangle \ l = \mathsf{fold}\ \mathsf{lit}_\wedge\ \overline{a}\ l \\
\mathsf{lit}_\vee \ \langle\!\langle \ \mathsf{true} \ \rangle\!\rangle \ l = \bot_{\mathsf{Absorbing}} & \mathsf{lit}_\wedge \ \langle\!\langle \ \mathsf{false} \ \rangle\!\rangle \ l = \bot_{\mathsf{Absorbing}} \\
\mathsf{lit}_\vee \ \langle\!\langle \ \mathsf{false} \ \rangle\!\rangle \ l = l & \mathsf{lit}_\wedge \ \langle\!\langle \ \mathsf{true} \ \rangle\!\rangle \ l = l \\
\mathsf{lit}_\vee \quad a \quad l = (a, \mathsf{not}\ a) : l & \mathsf{lit}_\wedge \quad a \quad l = (a, \mathsf{not}\ a) : l
\end{array}$$

For instance, given the formula $a = \mathsf{and}(b, \mathsf{not}\, c)$, we obtain the list of and-literals $\mathsf{lit}_\wedge\ a\ [\,] = [b, \mathsf{not}\ b\,;\, \mathsf{not}\ c, \mathsf{not}(\mathsf{not}\ c)]$. Remark here that the double negation will be simplified on-the-fly by the not smart-constructor.

These lists of literals are then sorted in order for a and $(\neg a)$ to appear side by side. For this purpose, we use a tricky relation ($\mathcal{R}_{\mathsf{id}}$) based on the hash-consed unique identifiers of terms computed during hash-consing:

$$(a, a')\ \mathcal{R}_{\mathsf{id}}\ (b, b') \ \Leftrightarrow\ \min(a_{\mathsf{id}}, a'_{\mathsf{id}}) \leq \min(b_{\mathsf{id}}, b'_{\mathsf{id}})$$

The relation $\mathcal{R}_{\mathsf{id}}$ is clearly a total order on pairs of terms. Thus we can sort list of literals with it. Moreover, we have $(a, b)\ \mathcal{R}_{\mathsf{id}}\ (b, a)$ for all terms a and b, such that pairs $(a, \neg a)$ and $(\neg a, a)$ are equal modulo $\mathcal{R}_{\mathsf{id}}$. Thus, opposite literals will be placed side-by-side in the sorted list.

Reducing lists of literals is surprisingly the same algorithm for conjunctions and disjunctions. The normalizations are based on the fact that, for any boolean property φ, both $(\varphi \vee \neg\varphi)$ and $(\varphi \wedge \neg\varphi)$ simplify to their associated absorbing elements, respectively true and false. The dual normalization uses the simplification of both $(\varphi \vee \varphi)$ and $(\varphi \wedge \varphi)$ into φ. For this purpose, we define weak versions of $(a \Leftrightarrow b)$ and $(a \Leftrightarrow \neg b)$, respectively defined as follows:

$$\begin{array}{l}
\mathsf{eqv}_{\mathsf{lit}}\ (a, a')\ (b, b') = (a \equiv b) \\
\mathsf{neq}_{\mathsf{lit}}\ (a, a')\ (b, b') = (a \equiv b') \vee (a' \equiv b)
\end{array}$$

Then, provided $[\![\, a' \,]\!] = \neg[\![\, a \,]\!]$ and $[\![\, b' \,]\!] = \neg[\![\, b \,]\!]$ (which is the case for literals), the two following properties hold:

$$\begin{array}{lcl}
\mathsf{eqv}_{\mathsf{lit}}\ (a, a')\ (b, b') & \Rightarrow & [\![\, a \,]\!] \Leftrightarrow [\![\, b \,]\!] \\
\mathsf{neq}_{\mathsf{lit}}\ (a, a')\ (b, b') & \Rightarrow & [\![\, a \,]\!] \Leftrightarrow \neg[\![\, b \,]\!]
\end{array}$$

The reduction of both conjunction and disjunction of literals is then implemented by one single function group, as follows:

$$\begin{array}{l|l}
\mathsf{group}\ \varphi : \psi : l \mid \mathsf{eqv}_{\mathsf{lit}}\ \varphi\ \psi\ = \mathsf{group}\ (\psi : l) & \mathsf{group}\ \varphi : l = \varphi : \mathsf{group}\ l \\
\mathsf{group}\ \varphi : \psi : l \mid \mathsf{neq}_{\mathsf{lit}}\ \varphi\ \psi\ = \bot_{\mathsf{Absorbant}} & \mathsf{group}\ [\,] \quad = [\,]
\end{array}$$

Putting every ingredient together, we define the smart constructors and and or in terms of a generic function $\mathsf{connective}_{\otimes,\mathbf{1},\mathbf{0}}$ for connective $\otimes$ with neutral $\mathbf{1}$ and absorbing element $\mathbf{0}$:

$$\boxed{\mathsf{and} = \mathsf{connective}_{\wedge,\mathsf{true},\mathsf{false}} \qquad \mathsf{or} = \mathsf{connective}_{\vee,\mathsf{false},\mathsf{true}}}$$

The generic function $\mathsf{connective}_{\otimes,\mathbf{1},\mathbf{0}}$ is in turn defined by flattening, sorting and grouping the $\otimes$-literals as follows:

$$\begin{array}{l}\mathsf{connective}_{\otimes,\mathbf{1},\mathbf{0}}(a,b) = \\ \quad \mathsf{try\,let}\ p = \mathsf{group} \circ \mathsf{sort}\ \mathcal{R}_{\mathsf{id}}\ (\mathsf{lit}_{\otimes}\ a\ (\mathsf{lit}_{\otimes}\ b\ [\,])) \ \mathsf{in} \\ \quad\quad \mathsf{if}\ p = [\,]\ \mathsf{then}\ \mathbf{1}\ \mathsf{else} \\ \quad\quad\quad \mathsf{let}\ \overline{a} = \mathsf{sort}\ (\preceq) \circ \mathsf{map\ fst}\ p\ \mathsf{in}\ \mathrm{«}\otimes\ \overline{a}\ \mathrm{»} \\ \quad \mathsf{with}\ \bot_{\mathsf{Absorbing}} \to \mathbf{0}\end{array}$$

The smart constructor for implication is more direct. Recall that the normal form of implication is $\mathrm{«}\wedge\overline{a} \to a\mathrm{»}$. We only need to filter out the list of hypotheses on the left of $(\to)$ against the conclusion and its negation. Below are two examples of the reduction rules implemented for the imply smart constructor:

$$\boxed{\begin{array}{l}\mathsf{imply}\ \mathrm{«}\wedge\overline{a}\ \mathrm{»}\ b\ \ \mid (\exists i, a_i \equiv b) \ = \mathrm{«}\,\mathsf{true}\,\mathrm{»} \\ \mathsf{imply}\ \mathrm{«}\wedge\overline{a}\ \mathrm{»}\ b\ = \mathrm{«}\wedge[a_j | a_j \not\equiv \mathsf{not}\ b] \to b\,\mathrm{»}\end{array}}$$

Equivalence is the same than equality in the boolean theory. The contribution of boolean theory to equality smart constructors is defined below:

$$\boxed{\begin{array}{c}\mathsf{eq}_{\mathcal{B}}\ \mathrm{«}\,\mathsf{true}\,\mathrm{»}\ a = a \quad \mathsf{eq}_{\mathcal{B}}\ \mathrm{«}\,\mathsf{false}\,\mathrm{»}\ a = \mathsf{not}\,a \\ \mathsf{eq}_{\mathcal{B}}\ a\ b \mid (a \equiv \mathsf{not}\,b) \ = \mathrm{«}\,\mathsf{false}\,\mathrm{»}\end{array}}$$

Finally, we define the smart constructor for negation recursively with all the other connectives. We do not present all the rules here by lack of place. Let us just mention the transformation of $\mathsf{not}\,(a \neq b)$ into $(a = b)$, $\mathsf{not}\,(a \leq b)$ into $(b < a)$, among many other similar or dual patterns.

Arithmetic. The normalization of arithmetic terms relies on computing with linear forms of terms:

$$\mathsf{linear}(a) = c + \sum_{i=1}^{n} k_i.a_i \quad \text{with } c, k_i \in \mathbf{Z}$$

Maximal linear forms of terms are easy to compute in an efficient way with lists of monoms (k, a), in the same spirit than for logical connectives. For linear complexity, we use an accumulative variant of linear, denoted by lin, such that:

$$\mathsf{lin}\ k\ a\ L = k.\ \mathsf{linear}(a) + L$$

Conversely, it is easy to inject linear forms into well-formed terms as follows:

$$\mathsf{inj}_{\Sigma}\left(c + \sum_{i=1}^{n} k_i.a_i\right) = \mathrm{«}\,\Sigma\,\overline{s}\,\mathrm{»} \quad \text{where} \quad \begin{cases} s_0 = \mathrm{«}\,c\,\mathrm{»} \\ s_i = \mathrm{«}\,k.a_i\,\mathrm{»}, i \in 1..n \end{cases}$$

With list implementation, inj_Σ relies on sorting and compacting the list of monoms to obtain a normalized linear form. Smart constructors for arithmetic are then straightforward definitions:

$$\boxed{\begin{array}{r}\mathsf{add}(a,b) = \mathsf{inj}_\Sigma(\mathsf{lin}\ 1\ a\ (\mathsf{lin}\ 1\ b\ []))\\ \mathsf{sub}(a,b) = \mathsf{inj}_\Sigma(\mathsf{lin}\ 1\ a\ (\mathsf{lin}\ \text{-}1\ b\ []))\\ \mathsf{times}(k,a) = \mathsf{inj}_\Sigma(\mathsf{lin}\ k\ a)\end{array}}$$

Comparisons, including equalities and inequalities, are also performed with linear forms using a generic comparison function $\mathsf{cmp}_\mathcal{R}$ for relation $\mathcal{R}$:

$$\boxed{\mathsf{leq} = \mathsf{cmp}_\leq \quad \mathsf{lt} = \mathsf{cmp}_< \quad \mathsf{eq}_\mathcal{A} = \mathsf{cmp}_= \quad \mathsf{neq}_\mathcal{A} = \mathsf{cmp}_\neq}$$

For the definition of this generic comparison function, we first introduce a dispatching function that takes a linear form L and separate positive from negative factors:

$$\mathsf{dispatch}\left(c + \sum_{i=1}^{n} k_i.a_i\right) = \left(\ c^\oplus + \sum_{i=1}^{n} k_i^\oplus.a_i\ ,\ c^\ominus + \sum_{i=1}^{n} k_i^\ominus.a_i\ \right)$$

where $c^\oplus = \max(c,0)$ and $c^\ominus = \max(-c,0)$. Then, we lift any arithmetic comparison $\mathcal{R}$ to linear forms with:

$$\mathsf{lift}_\mathcal{R}(L^\oplus, L^\ominus) \ = \ \text{«}\ L^\oplus\ \mathcal{R}\ L^\ominus\ \text{»} \quad \text{where typically} \quad L^\oplus, L^\ominus = \mathsf{dispatch}(L)$$

When linear forms are reduced to constants c and c', we compute the boolean result of $(c\mathcal{R}c')$ and turn it into «true» or «false». We also introduce few additional simplifications when both $L^\oplus$ and $L^\ominus$ are in $\mathbf{Z}$ (rather than in $\mathbf{R}$) in order to catch off-by-one comparisons ; typically $1 + a < b$ reduces to $a \leq b$.

Finally, the generic comparison operator $\mathsf{cmp}_\mathcal{R}$ is defined by:

$$\mathsf{cmp}_\mathcal{R}(a,b) = \mathsf{lift}_\mathcal{R} \circ \mathsf{dispatch} \circ \mathsf{inj}_\Sigma(\mathsf{lin}\ 1\ a\ (\mathsf{lin}\ \text{-}1\ b\ []))$$

Product are conducted in a similar, although simpler way. The simplification is here based on generalized products rather than linear forms:

$$\mathsf{product}(a) = k.\prod_{i=1}^{n} a_i \quad \text{and, conversely:} \quad \mathsf{inj}_\Pi\left(k.\prod_{i=1}^{n} a_i\right) = \text{«}\ k.\Pi\,\overline{a}\ \text{»}$$

Their implementation with lists are straightforward. We introduce an accumulative variant of $\mathsf{product}$, named prod such that:

$$\mathsf{prod}\ a\ (k,l) = k.\,\mathsf{product}(a) \times l$$

Finally, the smart constructor for multiplication is:

$$\boxed{\mathsf{mul}(a,b) = \mathsf{inj}_\Pi \circ \mathsf{sort}\ (\preceq)\ (\mathsf{prod}\ a\ (\mathsf{prod}\ b\ []))}$$

Arrays and Records The theories for arrays and records are similar and we present them together. For arrays, we need to decide whether two indices a and b are equal. Qed is not able to decide equality in all case, so we rely on a weak decision instead, *ie.* a sound but incomplete approximation of $[\![a = b]\!]$. Let us define:

$$a =_{\mathsf{true}} b \quad \Leftrightarrow \quad \mathsf{eq}(a,b) \equiv \mathop{\ll} \mathsf{true} \mathop{\gg}$$
$$a =_{\mathsf{false}} b \quad \Leftrightarrow \quad \mathsf{eq}(a,b) \equiv \mathop{\ll} \mathsf{false} \mathop{\gg}$$

The simplifications rules used by the smart constructors for arrays are then:

$$\begin{array}{l|l}
\mathsf{get} \mathop{\ll} a[b \mapsto c] \mathop{\gg} b' & (b =_{\mathsf{true}} b') = c \\
\mathsf{get} \mathop{\ll} a[b \mapsto c] \mathop{\gg} b' & (b =_{\mathsf{false}} b') = \mathsf{get}\ a\ b' \\
\mathsf{set} \mathop{\ll} a[b \mapsto c] \mathop{\gg} b'\ c' & (b =_{\mathsf{true}} b') = \mathop{\ll} a[b \mapsto c'] \mathop{\gg}
\end{array}$$

Records are more complete since we can always decide for field equality. But there is no mystery in them. We omit here the details of the normalization algorithms.

There is no special equalities for arrays. For records, we rely on the fact that two records are equal if and only they have equal field entries. More precisely, given $r = (\mathtt{f}_i, a_i)_{i \in 1..n}$ and $r' = (\mathtt{f}'_j, a'_j)_{j \in 1..m}$, we introduce:

$$\begin{array}{l}
\mathsf{eq}_{\mathtt{Fd}} \mathop{\ll} \{r\} \mathop{\gg} \mathop{\ll} \{r'\} \mathop{\gg} \\
\quad = \mathop{\ll} \mathsf{false} \mathop{\gg} \quad \textit{when } n \neq m \ \vee\ \exists k,\ \mathtt{f}_k \neq \mathtt{f}'_k \\
\quad = \mathsf{and}(\overline{e}) \quad \textit{otherwise, where } \forall k,\ e_k = \mathsf{eq}(a_i, a'_i)
\end{array}$$

Function Properties We enrich the standard theory of unspecified functions by attributing function symbols $f \in \mathcal{F}$ with algebraic properties. The structural equality ($\equiv$) over terms $a \in \mathcal{L}$ implements directly the general equality for unspecified functions. We enrich it with additional equalities when f is injective and when it is a constructor of an abstract datatype.

Sometimes, the function f is just the n-ary notation for some unspecified operator ($\odot$), that is, $f(\overline{x}) = x_1 \odot \ldots \odot x_n$. In this case, f can be attributed with groupoid properties like associativity and such.

The available properties, for operators, injections and constructors, are listed in Figure 3. Each function can be attributed with zero, one or several properties, although you can not mix operator properties with non-operator ones.

Smart constructors for functions take into account those properties in two ways. Groupoid properties are used to flatten the list of arguments (associativity), to sort them with respect to ($\preceq$) (commutativity) and to filter out absorbing and neutral elements, whenever each case applies. The other properties are used to simplify equalities between terms $\mathop{\ll} f(\overline{a}) \mathop{\gg}$ and $\mathop{\ll} f'(\overline{a}') \mathop{\gg}$. Implementation is based on list manipulations similar to linear forms and logical connectives.

Equalities. The built-in theories of Qed define specific smart constructors for equality, that we need to merge into a single one. Moreover, equality as an equivalence relation also requires general normalizations to be applied. This is

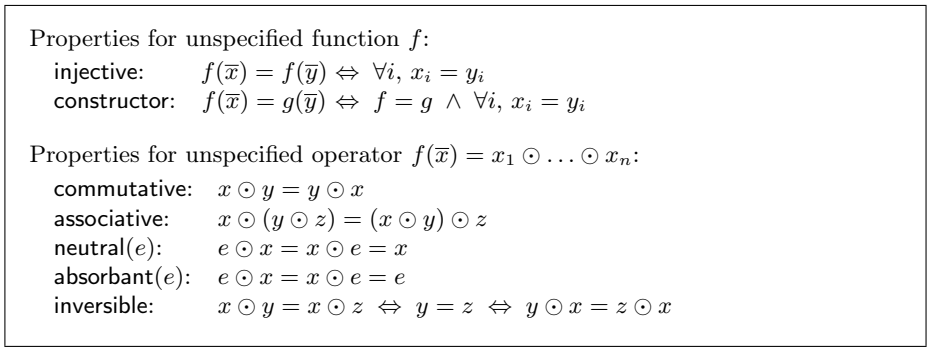


Fig. 3. Properties for unspecified functions

performed by smart constructor $\mathsf{eq}_{\mathcal{E}}$ which simplifies equal terms modulo ($\equiv$) and ensures that in $\ll a = b \gg$, we get $(a \prec b)$.

Combining equalities from all theories is achieved by applying each specific smart constructors in a staged way. Starting with the smart constructor of theory $\mathcal{T}$, if $\mathsf{eq}_{\mathcal{T}}(a, b) = \ll a' = b' \gg$, we pass the residual equality through the next theory $\mathsf{eq}_{\mathcal{T}'}(a', b')$, and so on.

In this process, several optimizations are performed to avoid unnecessary calls to dedicated smart constructors. The global stack is: first, use pure equality $\mathsf{e}_{\mathcal{E}}$; then, solve arithmetic with $\mathsf{eq}_{\mathcal{A}}$ or solve boolean equalities with $\mathsf{eq}_{\mathcal{B}}$; finally, depending on which theory applies, use $\mathsf{eq}_{\mathtt{Fd}}$ for records or $\mathsf{eq}_{\mathcal{F}}$ for functions.

2.3 Extensible Simplifier

One of the non-common features of Qed framework is its ability to be extended with user-supplied simplification routines. We have designed three possible entry points for additional normalizations, based on unspecified functions $f \in \mathcal{F}$:

- when applying a function $\ll f(\overline{a}) \gg$;
- for simplifying equalities $\ll a = f(\overline{a}) \gg$ and $\ll f(\overline{a}) = a \gg$;
- or inequalities $\ll a \leq f(\overline{a}) \gg$ and $\ll f(\overline{a}) \leq a \gg$.

Restricting these entry points to terms with a function symbol f at head is a design choice. It reduces the cost of finding routine, try to run them, and fallback to default implementation. In a similar way, we allow only *one* simplification routine per function symbol f and entry point. If several routines are desired, packing them with priorities and other features is left to the end-user of the framework, while keeping Qed simple and robust.

Regarding the implementation, calls to user-supplied simplification routines are staged *after* the default normalization routines and *before* hash-consing is performed. Although user-supplied simplification routines can be arbitrary OCaml code, there are some design rules to consider. We investigate them in turn.

Result. Simplification routines build terms using only the Qed smart constructors. A partial simplification routine may raise an exception $\bot_{\mathsf{Default}}$ to interrupt the simplification and makes Qed fallback to the default smart-constructor.

Recursion. To avoid infinite loops, Qed enforces a fallback to default smart constructors after a given depth of recursion with the same routine (2 in practice). This is consistent with the local complexity of all normalizers in the framework.

Decisions. Whenever a simplification needs to decide between several cases, it is recommended to build a Qed term instead, and decide upon its normalized form. For instance, to decide whether a sub-term a is positive, simply build the term $\mathsf{leq}\ \ll 0 \gg a$ and compare its normal form $\ll \mathsf{true} \gg$ and $\ll \mathsf{false} \gg$. This allows for several simplification routines to cooperate with each others.

Example. For instance, assume the symbol f_{abs} is specified to compute the absolute value of real and integral numbers. One may implement the following routine for simplifying f_{abs} expressions:

$$\begin{array}{l}\mathsf{call}_{\mathsf{abs}}\ a\ = \mathsf{match}\ (\mathsf{leq}\ \ll 0 \gg a)\ \mathsf{with}\\ \quad \mid\ \ll \mathsf{true} \gg\ \to a\\ \quad \mid\ \ll \mathsf{false} \gg\ \to \mathsf{times}\ \ll \text{-}\,1 \gg a\\ \quad \mid\ _\quad\ \to \bot_{\mathsf{Default}}\end{array}$$

To simplify notations, let us introduce $\mathsf{abs}(a) = \mathsf{call}(f_{\mathsf{abs}}, [a])$. This makes $\mathsf{abs}(\ll \text{-}\,1 \gg)$ to simplify into $\ll 1 \gg$ as expected. If we now add a routine for simplifying comparisons with symbols f_{abs}:

$$\begin{array}{l}\mathsf{leq}_{\mathsf{abs}}\ \ll 0 \gg\ \ll f_{\mathsf{abs}}(a) \gg\ = \mathsf{true}\\ \mathsf{leq}_{\mathsf{abs}}\ \ll f_{\mathsf{abs}}(a) \gg\ \ll 0 \gg\ = \mathsf{eq}\ a\ \ll 0 \gg\\ \mathsf{leq}_{\mathsf{abs}}\ a\ b = \bot_{\mathsf{Default}}\end{array}$$

Then we get the simplification of $\mathsf{abs}(\mathsf{abs}(a))$ into $\mathsf{abs}(a)$ for free by mutual interaction of the two simplification routines.

3 Experimental Results

In this section, we illustrate how Qed has been used to successfully empower the efficiency of the Wp plug-in of Frama-C. Recall from the introduction that Wp computes *weakest preconditions* on C programs annotated by ACSL contracts. The primary outcome of Wp is *proof obligations*, that are first-order logic formulæ. If one succeed in *proving* all those formulæ, then weakest precondition calculus entails that the C program is correct with respect to its specification.

The introduction of Qed as the internal implementation for building and managing the proof obligations has leverage the efficiency of Wp in many ways. First, it allows up to implement effectively a *linear* [14] weakest precondition calculus with on-the-fly *maximal memory sharing*. On programs with a lot of paths

in the control flow graph, like successive conditionals, this is absolutely necessary to avoid an exponential growth of proof obligations. Second, surprisingly, normalizations makes "not-so-few" proof obligations to simplifies into « true ». Hence, Qed became our primary back-end solver in practice.

	Without Qed			After Qed		
Case Study	Goals	Alt-Ergo	Coq	Goals	Qed	Alt-Ergo
A	13	13	-	11	11	-
B	35	14	17	22	18	3
C	54	24	30	25	25	-
D	Memory out			172	116	56

Fig. 4. Impact of Qed on Wp

An experiment conducted before and after the introduction of Qed is depicted in Figure 4. It depicts four simple case studies, that are small C routines from industrial embedded systems, and our attempts to discharge the generated proof obligations (goals). The figures show that introduction of Qed actually avoid exponential growth and demonstrate its capability to discharge proofs. Without Qed, hardly 50% of the goals must be discharged by hand with the Coq proof assistant. For case-study named 'D', Wp is not even capable of generating the proof obligations. Introducing Qed solves most of these issues, however one proof obligation is still not discharged in the 'B' case study.

We then conducted a much larger experiment on a full bench of real industrial codes from avionics and energy industries. These case studies can not be disclosed here because of industrial agreements. The bench consists of 15 case studies, cumulating 60,000 lines of code and specifications which generates up to 10,000 proof obligations to be discharged. Of course, on such a large-scale experiment, we encountered non-generated goals and non-discharged ones. This can be the consequence of bugs, inefficiencies and over-complicated goals.

The results of the experiment on different variants of Wp and Qed are depicted in Figure 5. The graphics shows the number of proof obligations actually generated, and those discharged by Qed and Alt-Ergo. The graphics also provides the number of goals where Alt-Ergo has been interrupted, and those where it returns without deciding the validity of the proof obligation.

The various versions we experimented with this bench illustrate the benefit from non-trivial optimizations implemented in Wp thanks to the Qed framework:

WP. The base version of Wp with Qed (beginning of the experiment).
VAR. Transformation of equalities introduced by Wp into substitutions.
CST. Addition of simplification routines for machine-integer computations.
LET. Correction of an inefficiency issue with in-memory sharing.

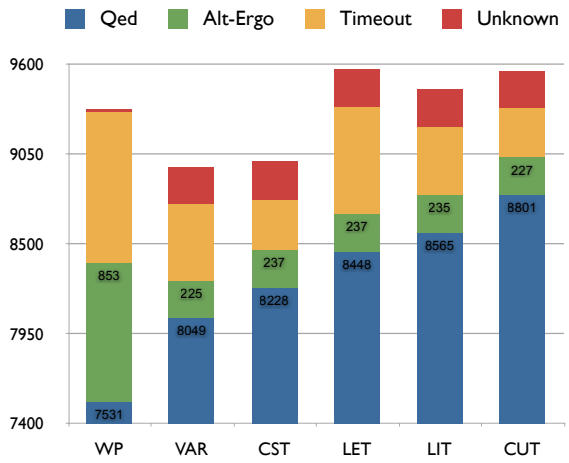


Fig. 5. Impact of Wp optimizations based on Qed

LIT. Propagation of literals by substitutions.
CUT. Pruning proof obligations by eliminating irrelevant chunks of code.

As illustrated by the results over the bench, each version improves the results in several ways. The number of generated proof obligations is lower when different control-flow paths can be merged thanks to simplifications during the weakest precondition calculus. On the other hand, inefficiency bugs may prevent Wp from generating proof obligations, leaving part of the specifications unproved. More goals are discharged by Qed after each optimization we introduced. And sometimes, residual goals are more efficiently discharged by Alt-Ergo, meaning that Qed has simplified them.

We now investigate in more details the experimented optimizations, and how they take benefit from the Qed framework.

Turning Equalities into Substitutions (VAR) During linear weakest precondition calculus, the side effects of the program are transformed into a kind of static single assignment form. This generates a huge number of intermediate variables, each receiving a small expression. This leads to many $(x = e)$ hypotheses in the formulæ to prove. But, formula $\forall x, x = e \rightarrow \varphi$ can be transformed into $\varphi[x := e]$ by substitution (provided x does not appear free in e). This is a well known transformation named variable elimination. But from the Qed point of view, this introduces many opportunities to perform aggressive normalization. For instance, $\forall x, x = 4 \rightarrow 0 \leq x$ does not simplifies locally in Qed, but simplifies into true after substitution.

Simplification Routines (CST) To model the semantics of C machine integers, the Wp introduces unspecified symbols with suitable properties in order for SMT

solvers to reason with. However, in many cases, these symbols are fed with constant integer values. Hence, we can compute on-the-fly the resulting values. For instance, when converting constants from one integer type into another. Together with variable elimination, this makes significant improvements.

Exploiting Memory Sharing (LET) When exporting a formula to an external solver, Qed takes benefit from the maximal sharing of equal sub-terms into memory. For instance, term « $f(a, a)$ » where a is a shared sub-term, is rendered by introducing a let-binding: « $\mathsf{let}\ x = a\ \mathsf{in}\ f(x, x)$ ». In early versions of Qed, there was an inefficiency bug in finding good candidates for let-binding introduction. This bug was responsible for combinatorial explosions during the export of proof obligations. This is an illustration of how maximal sharing is important in practice.

Propagation of Literals (LIT) Generalizing variable elimination, formula $(e = c) \rightarrow \varphi$ may sometimes be transformed into $\varphi[e := c]$. This is of particular interest when c is much simpler than e, say, a constant. Of course, recognizing a sub-expression e in φ can be costly. But with maximal in-memory sharing and hash-consing, this becomes feasible in reasonable time.

A special instance is the propagation of hypotheses: in formula $l \rightarrow \varphi$, we substitute l by true and $\mathsf{not}(l)$ by false in φ.

Moreover, we also propagate consequences inequalities: $a < b$ also propagates $a \leq b$ and $a \neq b$. Finally, we also detect both $a \leq b$ and $b \leq a$ and turn them into $a = b$. This combines well with the normalization of inequalities performed by Qed, since this makes variants of the same literal to be equal and substituted. For instance, it is often the case that at the end of a loop, the loop counter will be replaced with its final value, which introduces more opportunities for further variable eliminations.

However, in $\psi \rightarrow l \rightarrow \varphi$, we only propagate l from left-to-right, in φ only, because propagation in both directions is exponential.

Pruning Contradictory Branches (CUT) A typical program has many conditionals statements to detect error cases that shortcuts normal computations. When proving a property of such a program, we generally have a specification such as "*unless an error condition is raised, some property φ holds.*" This leads to formulæ with the following form:

$$(d\ ?\ \psi^+ : \psi^-) \rightarrow (e \rightarrow \varphi)$$

There are two opportunities for simplifications in this formula. First, we can put e in head of the goal, such that forward propagation of literals described above has a chance to filter out non relevant cases. Then, we may investigate whether $(e \wedge d \wedge \psi^+)$ or $(e \wedge \neg d \wedge \psi^-)$ leads to a contradiction by simplification with Qed. Whenever it is the case, the corresponding branch can be removed.

This is effective in practice, as shown by our experiments. However, it must be pointed out that this only occurs because Qed performs many normalizations in the background.

4 Conclusion

Our primary objective was to statically prove program properties with SMT solvers. For this purpose, we generate first-order logic formulæ relying on several domain specific theories. Naive approaches lead to generating huge formulæ that are tremendously difficult for SMT solvers to discharge. We have tackled this problem by introducing the Qed framework, an efficient library for managing formulæ modulo built-in and domain specific theories. This provides us with a mean of simplifying on-the-fly the generation of the formulæ to prove. Our rationale is that simplifications that are fast and local should be done in the early stage of the process, while only the difficult residual goals are sent to state-of-the art SMT solvers for deep exploration. Future research includes the simplification of terms by abstract interpretation and the usage of Qed in other tool chains.

References

1. Barrett, C.W., de Moura, L.M., Stump, A.: Smt-comp: Satisfiability modulo theories competition. In Etessami, K., Rajamani, S.K., eds.: CAV. Volume 3576 of Lecture Notes in Computer Science., Springer (2005) 20–23
2. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. In: Proceedings of the 10th International Conference on Software Engineering and Formal Methods. SEFM'12, Springer-Verlag (2012) 233–247
3. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL Specification Language. (2013) http://frama-c.com/acsl.html.
4. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. SIGSOFT Softw. Eng. Notes **31**(1) (September 2005) 82–87
5. Baudin, P., Correnson, L., Dargaye, Z.: WP User Manual, v0.7. (2013) http://frama-c.com/download/frama-c-wp-manual.pdf.
6. Coq Development Team: The Coq Proof Assistant. http://coq.inria.fr.
7. Conchon *et al*, S.: The Alt-Ergo Automated Theorem Prover. http://alt-ergo.lri.fr.
8. Bobot, F., Filliâtre, J.C., Marché, C., Melquiond, G., Paskevich, A.: The Why3 platform 0.81
9. Filliâtre, J.C., Conchon, S.: Type-safe modular hash-consing. In: Proceedings of the 2006 Workshop on ML. ML '06, ACM (2006) 12–19
10. Okasaki, C., Gill, A.: Fast mergeable integer maps. In: In Workshop on ML. (1998)
11. Gordon, A., Melham, T.: Five axioms of alpha-conversion. In: Theorem Proving in Higher Order Logics. Volume 1125 of LNCS. (1996) 173–190
12. Johnsson, T.: Lambda lifting: Transforming programs to recursive equations, Springer-Verlag (1985)
13. de Moura, L.M., Bjorner, N.: Generalized, efficient array decision procedures. In: FMCAD, IEEE (2009) 45–52
14. Leino, K.R.M.: Efficient weakest preconditions. Unpublished Manuscrit (2003) http://research.microsoft.com/en-us/um/people/leino/papers/krml114a.pdf.