Algorithm-independent framework for verifying integer constraints[*]

David Teller[†]
dtelle@ens-lyon.fr

Zhong Shao
shao@cs.yale.edu

Abstract

Proof-carrying code (PCC), as pioneered by Necula and Lee, allows a code producer to provide a compiled program to a host, along with a formal proof of safety. The PCC-based systems often rely on solving integer constraints to prove the soundness of the index types and to control resource consumption. Unfortunately, existing approaches often require the inclusion of an oracle-like constraints solver into the trusted computing base (TCB) or at least lock the safety policy with one particular solver. This paper presents a feasibility study for dissociating the constraints solver from the TCB and the safety policy from the actual solver algorithm. To demonstrate this, we produce a simple framework, we show how to adapt the popular solvers such as the Omega test and the Simplex method into this framework and we study some of its properties.

keywords : certified code, certified array bounds checking elimination, certified parallelization, certified Omega test, certified formal calculus, axiomatization of $\mathbb{Z}$

1 Introduction

In proof-carrying code (PCC) [8, 10], a code producer and a code consumer (host) start by agreeing on a safety policy. This policy is specified as a set of axioms for reasoning about safety. The code producer will then ship a compiled program to the consumer, along with a formal proof of its safety. Of course, the formal proof must be expressed in term of those axioms. Common examples of such policies include memory soundness, security, CPU time bounds and other resource control. Someday, banking applets will presumably comply with money-transfer soundness policies.

PCC relies on the same formal methods as does program verification; but it has the significant advantage that safety properties are much easier to prove than program correctness. The producer's formal proof will not, in general, prove that the code produces a correct or meaningful result; but it guarantees that execution of the code can do no harm. Thus, it cannot replace other methods of program assurance. On the other hand, the proofs can be mechanically checked by the host; the producer need not be trusted at all, since a valid proof is incontrovertible evidence of safety.

Using PCC allows to remove many run-time checks without sacrificing safety. For example, the Touchstone compiler [10] can prove the memory safety of the compiled programs. In other words, Touchstone-compiled programs can be trusted to run on devices without memory protection. Recently, Xi [20, 18] introduced a dependent type system in which the costly process of array bounds checking can be removed—a method which has been adapted to do so in a provably safe way.

CPU time bounding [9, 5], memory soundness [8, 7, 1], and array bounds checking [20], all require some kind of integer constraints handling. So does automatic parallelization [14]. And so do presumably many unmentioned processes. To be more specific, they require solving a set of integer equality or inequality equations to prove the soundness of the index types and to control resource consumption. Unfortunately, existing approaches either include the constraints solver into the trusted computing base (e.g., Xi's DML [18] and DTAL [19]), or lock the safety policy with one particular solver logic (e.g., Necula's Simplex solver logic [10, 9]).

The goal of this work is to study the feasibility of producing a generic framework where the constraints solver does not have to be in the TCB and the safety policy is independent of the actual solver algorithm. This is important because it would allow the code producer to choose the constraints solver most appropriate to a particular application. The producer will have much more flexibility in the manner in which the mobile code is proved safe. Furthermore, by removing the solver algorithm or the specific solver logic inference rules from the TCB, we get a higher-assurance system: any assumption about a particular solver algorithm must be proved.

2 Background and motivation

2.1 Solving integer constraints

Trying to solve all integer constraints is a somewhat ambitious task. As a matter of fact, this would mean solving Hilbert's tenth problem " Determination of the solvability of a Diophantine equation. Given a diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers." A problem which has been showed undecidable.

Hence, no solver handles just any integer constraints. Most of them only handle linear integer constraints. Some of them contain extensions to simple families of polynomials. We will follow the wise precedents and only attack linear constraints, for now.

Three families of solvers are currently used :

- Fourier-Motzkin's variable elimination [6]
- SUP-INF [3]
- The Simplex algorithm [11]

The most commonly used algorithm is the Omega test [13], a variant on Fourier-Motzkin's variable elimination [6]. Suppos-

[*]This research was sponsored in part by the Defense Advanced Research Projects Agency ISO under the title "Scaling Proof-Carrying Code to Production Compilers and Security Policies," ARPA Order No. H559, issued under Contract No. F30602-99-1-0519 and in part by NSF Grants CCR-9901011. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

[†]This work was done while the first author was visiting the FLINT group in the Department of Computer Science at Yale University.

edly, it is the fastest and most complete algorithm currently in use. For now, suffice to say that it works by eliminating trivial equalities and inequalities, reducing non-trivial equalities into trivial equalities by applying a variant of the `mod` operator, reducing non-trivial inequalities into trivial equalities by projection in some n-dimensional space and checking exhaustively for solutions when the formal methods do not work. Since this is the most commonly used algorithm and this it is definitely non-trivial, we used it as a basis for our research.

The Simplex algorithm, noticeably used by Necula in [Necula 1998], is a much simpler algorithms which proceeds by some linear algebra transformations on the matrix representation of the constraints. Since this algorithm is fairly simple, completely different from the Omega test, and since Necula has already made some steps to make it produce proofs, we used it as a confirmation of the genericity of this work.

2.2 Verifying integer constraints

2.2.1 Dissociating collection and verification

The first approach one can think of for the verification of integer constraints-related policies is the definition of a source-level semantic/logic. In other words, a set of inference rules which would be applied directly to the source code, would check all integer-related operations and decide, for example, of the satisfiability of problematic cases. This verification could take, for example, the form of a type-checking algorithm.

A variant of this approach would imply the translation into a simpler language, say some form of λ-calculus, which would allow the use of a smaller set of rules.

The shortcoming of this method is in its definition : interleaved collection and resolution of integer constraints - both of which are undecidable problems - make this semantic bigger, more complex, harder to maintain. And, of course, if the security policy is to change, the whole semantic will be due for change.

Dissociating both aspects is not going to solve anything by itself, of course. But it allows to work on both aspects separately, and to find separate solutions. For example, using Xi's dependent type system [18] or an annotation system for the collection of integer constraints makes it possible to overcome the undecidability of constraints collection.

2.2.2 Dissociating the policy from the constraints checker

The logical successor of the former approach is a two-level architecture : one constraints collecter and one policy checker/prover. However, this is still no solution.

Some program require different security policies. For example, two different optimizations such as array bounds checking elimination and automatic parallelization may not be proved valid by the same set of criteria. A program taking advantage of both optimizations will hence be collected for constraints and then policy-checked/policy-proved twice. However, both policies rely on integer constraints verification. In other words, it might be that only one step of both policy solver/checker is different.

In other words, the size of the proof and the time required to produce and then check it can be reduced by just dissociating the policy checker and the constraints solver. For example, the Omega Test [13] is an “as generic as it gets” integer constraints solver.

2.2.3 Proving the result

The next step is that of solving the constraints, which can be handled by classical algorithms such as the Omega Test or the Simplex for given families of integer constraints.

This is, however, not the last step. For the receiving host still has to be convinced of the (un)satisfiability of the set of constraints.

The satisfiability is rather trivial to prove, since it only requires a numerical certificate. In other words, a numerical example. However, the unsatisfiability is a much harder problem. In practice, when the enforcement of the policy requires the unsatisfiability of a set of constraints, two distinct methods are used to check that the security policy is enforced.

Solver algorithm in TCB The first solution is to include the decision algorithm in the trusted computing base. In other words, the run-time proof checking system contains the Omega test, for example, and is submitted a set of linear equations and inequations, which it solves, just like the compiler did. The method seems to have been implicitly used by Xi [18].

This method is not without advantages :

- It is simple to implement.
- The included Omega test might be very low level and fast.
- Carried proofs can be very short.

However, it has serious drawbacks :

- Since the whole set of equations and inequations is solved again, it requires lots of possibly useless calculations.
- The algorithm used cannot be changed, which prevents, for example, from adding the capability of solving some simple kind of polynomial constraints, to the solver.
- The TCB must be expanded with a possibly complex algorithm, thus increasing the risks of bugs, leading to unsoundness.
- If the run-time system has to be ported to another platform, the included test makes it bigger, hence probably harder to port.
- Since no integer constraints solving algorithm is complete, some kinds of constraints cannot be checked at compile-time. In particular, if the type system requires these constraints, as is the case with dependent/indexed type systems [18], the compiler will flatly refuse perfectly sound programs.

Solver-logic in TCB Another simple idea, introduced by Necula [10], is to define a set of deduction/inference rules encompassing all the steps in the algorithm and proving the result using these rules. In fact, this is the same thing as introducing a new logic adapted to the solver.

Once again, this has advantages:

- The algorithm does not need to be included in the TCB as such.
- Some useless calculations can be removed from the proof, hence resulting in faster checks.

However, similar drawbacks can be found:

- The TCB is still expanded, this time with a specific logical system, which is almost as hard to trust as the algorithm itself.
- The system is still locked with a particular algorithm and minor variations on this algorithm. This approach still prevents some optimizations and still refuses some sound programs.

3 Solver2FOL

3.1 Presentation

One of the next logical steps of modularization in the verification of integer constraints is that of rendering the algorithm independent from the proof. That of getting rid of the limits of the approaches previously exposed.

Solver2FOL is a small contribution to this task: a study of the feasibility of this dissociation. Although this work is no way related to the semantic model for PCC introduced by Appel and Felty [1], it can be seen as a complementary module: where Appel and Felty introduce a semantic model to describe types and machine instructions, in order to achieve language independence in PCC, we introduce a model to describe constraints solvers, in order to achieve algorithm independence in PCC.

Solver2FOL is a set of tools designed to allow the translation of integer constraints solving algorithms into algorithms building First Order Logic (FOL) proofs. It is composed of :

- A syntax to express constraints.
- A minimal semantics on the constraints.
- A minimal list of trusted operations.
- A collection of formal definitions to enrich the constraints using usual comparisons and operations.
- A collection of basic theorems, all of them formally proved, which prove the use of these enrichments valid.
- A collection of general-purpose lemmas, formally proved using these theorems.

In addition, we provide as an example Omega2FOL and Simplex2FOL, formally proved "Solver2FOL-izations" of the Omega test and of the Simplex.

FOL vs. set of rules First Order Logic was chosen instead of a custom set of rules since First Order Logic allows to express the kind of rules such a work would need as a set of axioms. Using a custom set of rules would provide shorter proofs but would not allow us to use the full power of First Order Logic. Unless, of course, we want to reexpress FOL into this set of rules.

And of course, using a custom set of rules is the first step toward defining and locking the algorithm by the mere description of the proof system - which is exactly what we want to avoid. FOL allows us to ignore this pitfall.

FOL vs. HOL First Order Logic was chosen because FOL rules and axioms tend to be simpler and more readable than true Higher Order Logics. However, we will show that this approach has several drawbacks.

Formalism Solver2FOL redefines most operations on $\mathbb{Z}$ using formal notations. This redefinition allows us to produce *purely formal proofs* of lemmas and theorems. In time, these purely formal proofs may be shipped and mechanically verified by a theorem checker.

3.2 Constructs

3.2.1 Grammar

Figure 1 gives the grammar for Solver2FOL. $Test$s are to be talen as *First Order Logic* terms.

This grammar is chosen as to make the basic set of axioms be as minimal as possible. In particular, the grammar does not contain inequalities or division—all these can be built on top of this skeleton.

This grammar is also designed to be fairly generic, so it can express the expressions as seen by the solver algorithm. For example, the Omega test mostly works with expressions of form "$Term \preceq 0$" along sometimes with "$a \otimes z \preceq \alpha$" and "$\beta \preceq b \otimes z$", whereas the Simplex only uses "$Term \succeq 0$" and "$Term \approx Term$".

$$\begin{array}{lcll}
Test & ::= & Term \approx Term & \text{(term comparison)} \\
 & & & \\
Term & ::= & Var \in \mathcal{V} & \text{(unknown variable)} \\
 & | & Int \in mathbbZ & \text{(integer constant)} \\
 & | & Term \oplus Term & \text{(term addition)} \\
 & | & Term \otimes Term & \text{(term multiplication)}
\end{array}$$

Where $\mathcal{V}$ is a countable infinite set.

Figure 1: Grammar for Solver2FOL

These design decisions allow the translated Solver2FOL algorithm to use the exact same set of conditions as the original algorithm as well as to reflect directly its inner workings. The other reasons for our choice are to serve as a place-holder for future extensions on polynomials and to pave the way to render the translation from an algorithm into Solver2FOL semi-automatic.

3.2.2 Semantics

Constants

Comparisons on constants Classical comparisons on $\mathbb{Z}$ such as $\leq$, $<$, $=$, $>$, $\geq$ and $|$ are supposed to be already implemented and trusted. This represents the fact that all those operations are already part of the CPU, hence already trusted in order to run the proof-checking system.

Operations on constants Classical operations on $\mathbb{Z}$ such as addition, multiplication, floor division ($\lfloor\dot{-}\rfloor$), ceil division ($\lceil\dot{-}\rceil$), modulo and absolute value are supposed to be already implemented and "mostly trusted". This represents the fact that all those operations are already part of the CPU, hence already trusted in order to run the proof-checking system - but that exceptions can occur.

"Mostly trusted" means that operations are trusted as long as no exceptions are thrown (division by zero, overflow, underflow). The semantics of exceptions in Solver2FOL is that if any exception should be raised, then the whole proof is considered false. Although this is not a fully satisfactory arrangement, it is not incoherent : if a proof contains such an exception, it probably means that the prover forgot to catch the same exception. However, this might not be true in case of cross-platform proof-building.

Variables

Operations on variables Although most of the times integer constants will not be effectively substituted to variables, the semantics of Solver2FOL is designed for "constant place-holders"-variables.

In other words, variables stand for unknown constants.

Terms

Definition Terms are members of set $\mathcal{T}$ defined by the grammar of Solver2FOL. Terms can be considered as an extension of integers with integer variables. For the sake of simplicity, we will consider $\mathcal{T}$ as a superset of $\mathbb{Z}$.

Comparisons on terms Classical comparisons on $\mathbb{Z}$ such as $\leq$, $<$, $=$, $>$, $\geq$ and $|$ can be extended to terms. However, the only needed comparison is the equality law $\approx$, defined by

$$\begin{cases} \approx \text{ is an equivalence law on } \mathcal{T} \\ \approx \text{ is compatible with } = \text{ on } \mathbb{Z} \end{cases}$$

Operations on terms Classical operations on $\mathbb{Z}$ such as addition, multiplication, division ($\lfloor \div \rfloor$) can be extended to terms. However, the only needed operations are addition $\oplus$ and multiplication $\otimes$, defined by

$$\begin{cases} \oplus \text{ is an internal composition law on } \mathcal{T} \\ \oplus \text{ is commutative, associative, distributive on } \otimes \\ 0 \text{ is neutral for } \oplus \\ \oplus_{|\mathbb{Z}} \equiv_{def} + \end{cases}$$

$$\begin{cases} \otimes \text{ is an internal composition law on } \mathcal{T} \\ \otimes \text{ is commutative, associative} \\ 0 \text{ is absorbant for } \otimes \\ 1 \text{ is neutral for } \otimes \\ \otimes_{|\mathbb{Z}} \equiv_{def} \cdot \end{cases}$$

3.3 Definitions as axioms

3.3.1 First generation

Syntactical conventions

- Operators on $\mathcal{T}$ have the same precedence as their counterparts in $\mathbb{Z}$.
- In case of ambiguity, all operations are supposed left-parenthesized.
- $X \ominus Y$ is a syntactical shortcut for $X \oplus ((-1) \otimes Y)$
- lowercase letters stand for integers constants
- uppercase letters stand for expressions

Presentation The basic axioms are presented in Figure 2. They are translations of the semantics of the basic constructs of Solver2FOL.

3.3.2 Second generation

Definition Operators $.\backslash.$ and $.\%.$ are the respective counterparts of $\lfloor \div \rfloor$ and `. mod .` on $\mathcal{T}$. Binary relations $\|$, $\preceq$, $\prec$, $\succ$, $\succeq$ are the respective counterparts of $|$, $\leq$, $<$, $>$, $\geq$. They are defined by :

$$\begin{cases} X\backslash a \approx Y \text{ iff for some b in } [0, a-1], X \approx a \otimes Y \oplus b \\ X\%a \approx b \text{ iff b is in } [0, a-1] \text{ and for some Y, } X \approx a \otimes Y \oplus b \\ a \parallel X \text{ iff } X\%a \approx 0 \\ X \preceq Y \text{ iff for some non-negative b, } Y \approx X \oplus b \\ X \prec Y \text{ iff for some strictly positive b, } Y \approx X \oplus b \\ X \succeq Y \text{ iff for some non-negative b, } X \approx Y \oplus b \\ X \succ Y \text{ iff for some strictly positive b, } X \approx Y \oplus b \end{cases}$$

Axiomatization The axiomatization of these operators and relations is immediate. It is presented on figure 3.

Equality		
$\forall X$	$X \approx X$	(equ ref)
$\forall X, Y$	$X \approx Y \Rightarrow Y \approx X$	(equ sym)
$\forall X, Y, Z$	$X \approx Y {}_\wedge Y \approx Z \Rightarrow X \approx Z$	(equ trans)
$\forall a, b$	$a \approx b \Rightarrow a = b$	(equ on $\mathbb{Z}$)
$\forall X$	$\exists a, X \approx a$	(equ int)
Addition		
$\forall X$	$X \oplus 0 \approx X$	(add 0)
$\forall X, Y$	$X \oplus Y \approx Y \oplus X$	(add comm)
$\forall X, Y, Z$	$X \oplus (Y \oplus Z) \approx (X \oplus Y) \oplus Z$	(add assoc)
$\forall X, Y, Z$	$X \approx Y \Rightarrow X \oplus Z \approx Y \oplus Z$	(add eq)
$\forall a, b$	$a \oplus b \approx a + b$	(add on $\mathbb{Z}$)
Multiplication		
$\forall X$	$0 \otimes X \approx 0$	(mult 0)
$\forall X$	$1 \otimes X \approx X$	(mult 1)
$\forall X, Y$	$X \otimes Y \approx Y \otimes X$	(mult comm)
$\forall X, Y, Z$	$X \otimes (Y \otimes Z) \approx (X \otimes Y) \otimes Z$	(mult assoc)
$\forall X, Y, Z$	$(X \oplus Y) \otimes Z \approx (X \otimes Z) \oplus (Y \otimes Z)$	(add/mult dist)
$\forall X, Y, Z$	$X \approx Y \Rightarrow X \otimes Z \approx Y \otimes Z$	(mult eq)
$\forall a, b$	$a \otimes b \approx a \cdot b$	(mult on $\mathbb{Z}$)

Figure 2: Basic axioms for Solver2FOL

Definition of operators		
$\forall X, Y, a$	$X\backslash a \approx Y \equiv_{def} \exists b,$ $0 \leq b {}_\wedge b < a {}_\wedge X \approx a \otimes Y \oplus b$	(dodiv)
$\forall X, a, b$	$X\%a \approx b \equiv_{def} \exists Y,$ $0 \leq b {}_\wedge b < a {}_\wedge X \approx a \otimes Y \oplus b$	(mod)
Definition of relations		
$\forall X, a$	$a \parallel X \equiv_{def} X\%a \approx 0$	(isdiv)
$\forall X, Y$	$X \preceq Y \equiv_{def} \exists b, 0 \leq b {}_\wedge Y \approx X \oplus b$	(leq)
$\forall X, Y$	$X \prec Y \equiv_{def} \exists b, 1 \leq b {}_\wedge Y \approx X \oplus b$	(lt)
$\forall X, Y$	$X \succeq Y \equiv_{def} \exists b, 0 \leq b {}_\wedge X \approx Y \oplus b$	(geq)
$\forall X, Y$	$X \succ Y \equiv_{def} \exists b, 1 \leq b {}_\wedge X \approx Y \oplus b$	(gt)

Figure 3: The second wave of definitions

3.4 Basic theorems

The following set of theorems will prove that the newly introduced operators are what they seem - namely that they can be used as their integer counterparts. Purely formal proofs are given in appendix.

Basic theorem 1 *(dodiv on $\mathbb{Z}$) The definition of $.\backslash.$ is compatible with $\lfloor \div \rfloor$ on $\mathbb{Z}$. In other words, for any integers a, b, $a\backslash b \approx \lfloor \frac{a}{b} \rfloor$.*

Basic theorem 2 *(mod on $\mathbb{Z}$) The definition of $.\%.$ is compatible with `. mod .` on $\mathbb{Z}$. In other words, for any integers a, b, $a\%b \approx a$ `mod` b.*

Basic theorem 3 *(isdiv on $\mathbb{Z}$) The definition of $\parallel$ is compatible with $|$ on $\mathbb{Z}$. In other words, for any integers a, b, $a \parallel b \iff a|b$*

Basic theorem 4 *(leq on $\mathbb{Z}$) The definition of $\succeq$ is compatible with $\geq$ on $\mathbb{Z}$. In other words, for any integers a, b, $a \succeq b \iff a \geq b$*

Basic theorem 5 *$\succeq$ defines a partial order on $\mathcal{T}$. In other words,*

$$\begin{cases} \forall X & X \succeq X & (\textit{geq ref}) \\ \forall X, Y & X \succeq Y {}_\wedge Y \succeq X \Rightarrow X \approx Y & (\textit{geq antis}) \\ \forall X, Y, Z & X \succeq Y {}_\wedge Y \succeq Z \Rightarrow X \succeq Z & (\textit{geq trans}) \end{cases}$$

Basic theorem 6 *(gt on $\mathbb{Z}$) The definition of $\succ$ is compatible with $>$ on $\mathbb{Z}$. In other words, for any integers a, b, $a \succ b \iff a > b$*

Basic theorem 7 *(lt on $\mathbb{Z}$) The definition of $\prec$ is compatible with $<$ on $\mathbb{Z}$. In other words, for any integers a, b, $a \prec b \iff a < b$*

Basic theorem 8 *(leq on $\mathbb{Z}$) The definition of $\preceq$ is compatible with $\leq$ on $\mathbb{Z}$. In other words, for any integers a, b, $a \preceq b \iff a \leq b$*

Basic theorem 9 *$\preceq$ defines a partial order on $\mathcal{T}$. In other words,*

$$\begin{cases} \forall X & X \preceq X & \textit{(leq ref)} \\ \forall X, Y & X \preceq Y_\wedge Y \preceq X \Rightarrow X \approx Y & \textit{(leq antis)} \\ \forall X, Y, Z & X \preceq Y_\wedge Y \preceq Z \Rightarrow X \preceq Z & \textit{(leq trans)} \end{cases}$$

Basic theorem 10 *(leq equiv geq) $\preceq$ and $\succeq$ are each other's symmetric. In other words, $\vdash X \preceq Y \iff Y \succeq X$.*

Basic theorem 11 *(lt equiv gt) $\prec$ and $\succ$ are each other's symmetric. In other words, $\vdash X \prec Y \iff Y \succ X$.*

Basic theorem 12 *(lt as leq) $\preceq = \prec \cup \approx$. In other words, $\vdash X \preceq Y \iff (X \prec Y_\vee X \approx Y)$.*

Basic theorem 13 *(gt as geq) $\succeq = \succ \cup \approx$. In other words, $\vdash X \succeq Y \iff (X \succ Y_\vee X \approx Y)$.*

Basic theorem 14 *(eq as ineq) $\preceq \cap \succeq = \approx$. In other words, $X \preceq Y_\wedge X \succeq Y \vdash X \approx Y$.*

Basic theorem 15 *(absurd ineq) $\prec \cap \succ = \emptyset$. In other words, $X \prec Y_\wedge X \succ Y \vdash \perp$.*

3.5 General-purpose lemmas

Formal proofs are given in appendix.

Main lemma 1 *(opp add) Opposite addition is provable using Solver2FOL. In other words, $\vdash Y \ominus Y \approx 0$*

Main lemma 2 *(add eq2) A second formulation for addition on each side of an equality can be proved using Solver2FOL. In other words, $X \approx Y \vdash Z \oplus X \approx Z \oplus Y$*

Main lemma 3 *(mult eq2) A second formulation for addition on each side of an equality can be proved using Solver2FOL. In other words, $X \approx Y \vdash Z \otimes X \approx Z \otimes Y$*

Main lemma 4 *(add eq3) General addition on both side of the equality sign can be proved using Solver2FOL. In other words, $X \approx Y_\wedge Z \approx T \vdash X \oplus Z \approx Y \oplus T$*

Main lemma 5 *Substraction in equality (sub equ) is provable using Solver2FOL. In other words,*

$$\begin{cases} X \oplus Y \approx Z \vdash X \approx Z \ominus Y \\ Z \approx X \oplus Y \vdash Z \ominus Y \approx X \end{cases}$$

Main lemma 6 *(sub ineq) Substraction in an inequality is provable with Solver2FOL. In other words,*

$$X \oplus Y \preceq Z \vdash X \preceq Z \ominus Y$$

Main lemma 7 *(mult sum) The propagation of multiplication in a sum is provable with Solver2FOL. In other words,*

$$\vdash p \otimes \Sigma_{i \in I} X_i \approx \Sigma_{i \in I} p \otimes X_i$$

Main lemma 8 *(var isol) The isolation against a variable in a sum is provable with Solver2FOL. In other words, for any j in I,*

$$\vdash \Sigma_{i \in I} X_i \approx (\Sigma_{i \in I \setminus \{j\}} X_i) \oplus X_j$$

Main lemma 9 *(var resol) The resolution against a variable of an equality between a sum and a constant is provable in Solver2FOL. In other words, for any k in I,*

$$\Sigma_{i \in I} X_i \approx c \vdash X_k \approx c \oplus \Sigma_{i \in I \setminus \{k\}} \ominus X_i$$

Main lemma 10 *(0 add2) $\vdash X \approx X \oplus 0$*

Main lemma 11 *(div sum) The propagation of $\mathbf{.}\backslash p$ in a sum where all elements are divisible by p is provable using Solver2FOL. In other words, if $p|a_1, p|a_2, \ldots p|a_n$ then $\vdash (\Sigma_{i \in [1,n]} a_i \otimes X_i) \backslash p \approx \Sigma_{i \in [1,n]} \frac{a_i}{p} \otimes X_i$*

Main lemma 12 *(mult ineq) Multiplication in inequality is provable using Solver2FOL. In other words,*

$$X \preceq Y_\wedge 0 \preceq Z \vdash Z \otimes X \preceq Z \otimes Y$$

3.6 Summary

Once this set of theorems and lemmas is proved, we have a full syntax and its associated semantics to handle formal expressions. Note that as opposed to Xi's approach, for example, quantifiers are not handled by Solver2FOL - they don't need to, since they are already handled by First Order Logic.

Also note that the trusted computing base is very small: it consists of classical integer operations and very few (and very basic) properties of $\otimes$ and $\oplus$. Thus, complex proofs can be traced back to a base of axioms which can easily be checked and trusted.

4 The Omega test

4.1 The original algorithm

The input of the Omega test is a set of linear equalities and linear inequalities involving only integer values. All the conditions are supposed to have form $\Sigma_{i=1}^{i=n} a_i \cdot x_i \leq c$ or $\Sigma_{i=1}^{i=n} a_i \cdot x_i = c$.

Removing equality constraints

- Equalities of the form $\Sigma_{i=1}^{i=n} a_i \cdot x_i = c$ are divided by the greatest common divisor of $a_1, \ldots, a_n$. If this leads to a formula with non-integer values, the test has failed and the system is unsatisfiable.
- If two equalities are visibly incompatible, the set is unsatisfiable.
- If one of the equalities contains a variable with coefficient 1 or -1, remove this equality and eliminate the variable.
- Otherwise, choose one equality $\Sigma_{i=1}^{i=n} a_i \cdot x_i = c$ and replace it with $-|a_k \cdot x_k| \cdot \sigma + \Sigma_{i=1}^{i=n, i \neq k} (\lfloor a_i/m + 1/2 \rfloor + a_i \mu m) \cdot x_i$ where $a \mu b \equiv_{def} a - b \cdot \lfloor a/b + 1/2 \rfloor$ and $m \cdot \sigma \equiv_{def} \Sigma_{i=1}^{i=n} (a_i \mu m) \cdot x_i = c$
- Proceed until there are no more equality constraints.

Removing inequality constraints

- Inequalities of the form $\Sigma_{i=1}^{i=n} a_i \cdot x_i \leq c$ are divided by the greatest common divisor of $a_1, \ldots, a_n$, rounding down c/g.
- If inequalities can be merged into equalities, merge them.
- If two inequalities are redundant, remove the weakest one.

- If inequalities are visibly incompatible, the set is unsatisfiable.
- If there are no more equalities and if there is a variable without upper bound or without lower bound, remove all the inequations involving this variable.
- Compute the *real shadow* of the set of inequalities along z by merging two constraints such as $a \cdot z \leq \alpha$ and $\beta \leq b \cdot z$ into constraint $a \cdot \beta \leq b \cdot \alpha$, where a and b are positive integers.
- Compute the *integer shadow* of the set of inequalities along z by merging the two same constraints into $b \cdot \alpha - a \cdot \beta \geq (a-1) \cdot (b-1)$.
- If both shadows are identical, the system is satisfiable iff there are integer solutions to the (common) shadow
- otherwise
  - if there are no integer solutions to the real shadow, the set is unsatisfiable
  - if there are integer solutions to the integer shadow, the set is satisfiable
  - otherwise
    - determine the largest coefficient a of z in any upper bound $\alpha \geq az$ on z and for each lower bound $bz \geq \beta$ on z, for each i in $[0, (ab - a - b)/a]$ do solve the problem combined with $bz = \beta + i$
  - Proceed until there is at most one inequality. If this happens the system is satisfiable. Otherwise, it is not.

4.2 Omega2FOL

Let us consider a reduced version Omega' of the known Omega test. This Omega' is a subset of the Omega test where two optimizations have been remove for the sake of simplicity. These optimizations are not required to run the Omega test :

Coefficients reduction We have not tried to prove this step yet. However, our work so far seems to indicate that this step will require a more complete work on the Omega test, including completeness studies.

Fusion of opposite inequalities This step is trivial but meaningless without coefficients reduction.

Construct 1 *There exists an algorithm Omega2FOL based on Solver2FOL which :*

- *can solve the same set of constraints as Omega'.*
- *instead of Yes/No, returns* **proofs of unsatisfiability** *whenever Omega' decides that the set is unsatisfiable.*

Steps All these steps have been formally proved using Solver2FOL. Proofs are presented in annex.

Algorithm step 1 *"Equality normalization" is provable using Solver2FOL. In other words, if $p|a_1, p|a_2, \dots p|a_n$ then* $\Sigma_{i\in[1,n]} a_i \otimes X_i \approx c \vdash \Sigma_{i\in[1,n]} \frac{a_i}{p} \otimes X_i \approx \lfloor \frac{c}{p} \rfloor$

Algorithm step 2 *"Unsatisfiable equality" is provable using Solver2FOL. In other words,*

$$p|a_1 {}_\wedge p|a_2 {}_\wedge \dots {}_\wedge p|a_n {}_\wedge \Sigma_{i=1}^{i=n} a_i \otimes X_i \approx c \vdash p|c$$

Algorithm step 3 *"Inequality tightening" is provable using Solver2FOL. In other words, if $p|a_1, p|a_2, \dots p|a_n$, then*

$$\Sigma_{i\in I} a_i \otimes X_i \preceq c \vdash \Sigma_{i\in I} \frac{a_i}{p} \otimes X_i \preceq \lfloor \frac{c}{p} \rfloor$$

Algorithm step 4 *"Real shadow" can be proved using Solver2FOL. In other words, if $1 \leq a$, $1 \leq b$ then* $a \otimes Z \preceq A {}_\wedge B \preceq b \otimes Z \vdash a \otimes B \preceq b \otimes A$.

Algorithm step 5 *"Exhaustive check" can be proved using Solver2FOL. In other words, if* $1 \leq a {}_\wedge 1 \leq b$, $(a-1) \cdot (b-1) \preceq b \otimes A \ominus a \otimes B {}_\wedge a \otimes \xi \leq A {}_\wedge B \leq b \otimes \xi \vdash \exists i, 0 \leq i {}_\wedge i \cdot a \leq (a \cdot b - a - b) {}_\wedge b \otimes \xi \approx B \oplus i$

Corollaries :

- Omega' was proved.
- We can produce Omega2FOL.

Informal listing of Omega2FOL For the sake of readability, this listing is an informal presentation of the decision procedure. For example, it does not detail the instantiation of each lemma, nor does it explicitly show that it tries to simplify the handled equations at each step, by turning multiplication by 0 into 0, by removing addition of 0, etc...

Also note that this algorithm returns a formal proof where Omega' would have returned No and $Maybe$ where Omega' would have returned Yes. This seemed more appropriate in absence of any completeness study on Omega'.

Removing equality constraints

```
If there is at least one unnormalized equality left $\mathcal{E}$
    Normalize
      Call Omega2FOL with a system containing the normalized
          equality $\mathcal{F}$ instead of the unnormalized one.
      If the test returns $Maybe$, return $Maybe$
      Otherwise, if the test returns proof P
          Return the composition of proof $\vdash \mathcal{E} \Rightarrow \mathcal{F}$ (built by
              lemma "Equality normalization") and P
If we can find an unsatisfiable equality $\mathcal{E}$
      Return lemma "Unsatisfiable equality" applied to $\mathcal{E}$
If we can find two contradictory equalities A and B
      Return lemma "eq trans" applied to A and B
If there is one equality $\mathcal{E}$ which involves a variable $\xi$ with coefficient 1
    Remove one variable
      Let $(\mathcal{F}_i)_i$ be the set of equations and inequations. Call
          Omega2FOL with a system containing $(\mathcal{F}'_i)_i$, the set
          of all equalities/inequalities in which the formal value
          associated to $\xi$ has been substituted to $\xi$ itself.
      If the test returns $Maybe$, return $Maybe$.
      Otherwise, if the test returns proof P
          Return the composition of the proof of the formal
              value of $\xi$ (built by lemma "variable resolution"),
              the proofs $\vdash \mathcal{F}_i \Rightarrow \mathcal{F}'_i$ (built by lemma "variable
              isolation in a sum" and lemma "add eq2") and P.
If there is one equality $\mathcal{E}$ which involves a variable with coefficient -1
    Remove one variable
      Call Omega2FOL with a system containing the opposed
          equality $\mathcal{F}$ in which the -1 has been fully propagated.
      If the test returns $Maybe$, return $Maybe$
      Otherwise, if the test returns proof P
          Return the composition of proof $\vdash \mathcal{E} \Rightarrow \mathcal{F}$ (built by
              lemma "eq mult" and lemma "sum mult") and P
```

```
Otherwise, if there is still at least one equality, let $X \approx c$ be one of
    the equalities left
      Call Omega2FOL with a system containing $X \preceq c$ and $c \succeq X$ instead of $X \approx c$
      If the test returns $Maybe$, return $Maybe$
      Otherwise, if the test returns proof P
          Return the composition of proof $\vdash X \approx c \Rightarrow X \preceq c \wedge X \succeq c$ (built by lemma "eq as ineq") and P
```

Removing inequality constraints

```
If we can find two redundant constraints
      Call Omega2FOL with a system containing all the constraints minus the weakest of the two redundant constraints. Return the result.
If we can find two contradictory inequalities A and B
      Return lemma "leq trans" applied to A and B
If there are no more equalities and if we can find a variable without
    upper bound or without lower bound.
      Call Omega2FOL with a system containing only comparisons which do not involve this variable. Return the result.
If there is a variable $\zeta$, along with one lower bound condition
    $B \preceq b \otimes \zeta$ and one upper bound condition $a \otimes \zeta \leq A$ such
    as $a = 1$ and $1 \leq b$ or $b = 1$ and $1 \leq a$.
    Both shadows are identical
      Call Omega2FOL with a system containing $a \otimes B \preceq b \otimes A$ instead of both conditions
      If the test returns $Maybe$, return $Maybe$
      Otherwise, if the test returns proof P
          Return the composition of proof $\vdash B \preceq b \otimes \zeta \wedge a \otimes \zeta \leq A \Rightarrow a \otimes B \preceq b \otimes A$ (built by lemma "Real shadow") and P
If there is a variable $\zeta$, along with one lower bound condition $B \preceq$
    $b \otimes \zeta$ and one upper bound condition $a \otimes \zeta \preceq A$ $(a, b \in \mathbb{N}^*)$.
    Ideally, choose the largest $a$ and $b$ available.
    Shadows are not identical
      Call Omega2FOL with a system containing $a \otimes B \preceq b \otimes A$ instead of both conditions
      If the test returns $Maybe$
          Call Omega2FOL with a system containing $(a-1) \cdot (b-1) \preceq b \otimes A \ominus a \otimes B$ instead of both conditions
          If the test returns $Maybe$, return $Maybe$
          Otherwise, if the test returns proof P
             Try all possibilities
              for each i in $[0, \frac{ab-a-b}{a}]$ do
                  Call Omega2FOL with a system containing $b \otimes \zeta \approx B \oplus i$ instead of $B \preceq b \otimes \zeta$
                  If the test returns $Maybe$, return $Maybe$
                  Otherwise, let $P_i$ be the proof returned by the test
              Return the composition of $(P_i)_i$ and lemma "Exhaustive check" composed itself with P
              return the proof
      Otherwise, if the test returns proof P
          Return the composition of P and lemma "Real shadow"
If there is only one condition left, and if it is an inequality involv-
    ing a variable, return $Maybe$
```

```
    $X \ominus Y \approx 0$ is an equality involving variable X with coefficient 1
  Resolution of the equality against X
    $\mathcal{S} \vdash X \approx Y$ (proved by lemma "variable resolution")
  Introduction of the value of X into $X \oplus Y \approx 0$
    $\mathcal{S} \vdash Y \oplus X \approx 0$ (proved by lemma "variable isolation")
    $\mathcal{S} \vdash Y \oplus Y \approx 0$ (proved by axiom "add eq")
  Introduction of the value of X into $X \preceq -1$
    $\mathcal{S} \vdash Y \preceq -1$ (proved by "eq as ineq")
  Factorization of $Y \oplus Y \approx 0$
    $\mathcal{S} \vdash 1 \otimes Y \oplus Y \approx 0$ (proved by "add eq" and "mult 1")
    $\mathcal{S} \vdash 1 \otimes Y \oplus 1 \otimes Y \approx 0$ (proved by "add eq" and "mult 1")
    $\mathcal{S} \vdash 2 \otimes Y \approx 0$ (proved by "mult/add dist")
  Equality normalization of $2 \otimes Y \approx 0$
    $\mathcal{S} \vdash Y \approx 0$ (proved by "equality normalization")
    $Y \approx 0$ is an equality involving variable Y with coefficient 1
  Resolution of the equality against Y
    $\mathcal{S} \vdash Y \approx 0$ (already proved)
  Introduction of the value of Y into $Y \preceq -1$
    $\mathcal{S} \vdash 0 \preceq -1$ (proved by "eq as ineq")
    $\mathcal{S} \vdash 0 \preceq -1$ Is a constraint without any variables.
    $\mathcal{S} \vdash 0 \leq -1$ (proved by "leq on $\mathbb{Z}$")
    $\mathcal{S} \vdash \bot$
```

Figure 4: Example of a resolution using Omega2FOL (outline)

4.3 An example

Consider the system

$$\begin{cases} X \ominus Y & \approx & 0 \\ X \oplus Y & \approx & 0 \\ X & \preceq & -1 \\ \mathcal{S} \equiv_{def} X \ominus Y \approx 0 \wedge X \oplus Y \approx 0 \wedge X \preceq -1 \end{cases}$$

If we ignore the omnipresent "eq trans" axiom (transitivity of equality), the resolution will look like figure 4

5 Simplex

The Simplex algorithm is a well known integer constraints solving algorithm. Noticeably, it has been used by Necula and Lee [10] in order to produce proofs in the Touchstone compiler.

Using Necula's work, the Simplex can be seen as a decision procedure building proofs using 9 simple logical rules : the Simplex-logic rules, presented on figure 5. Of these rules, *geqgeq, gtgeq, leqgeq, ltgeq, eqgeq, geq0* are already axioms or lemma of Solver2FOL and the three other ones can be very easily translated to Solver2FOL.

Construct 2 *There exists an algorithm Simplex2FOL based on Solver2FOL which :*

- *can solve the same set of constraints as the Simplex.*
- *instead of Yes/No, returns* **proofs of unsatisfiability** *whenever the Simplex decides that the set is unsatisfiable.*

Corollary :

Construct 3 *Solver2FOL's logic is complete with respect to linear integer constraints.*

The proof is straightforward : Solver2FOL's logic can express Simplex2FOL's results, Simplex2FOL can produce results for the same

$$\frac{\vdash X \succeq y}{\vdash X \ominus Y \succeq 0}\ \textbf{geqgeq}\ \frac{\vdash X \succ y}{\vdash X \ominus Y \ominus 1 \succeq 0}\ \textbf{gtgeq}$$

$$\frac{\vdash X \preceq y}{\vdash Y \ominus X \succeq 0}\ \textbf{leqgeq}\ \frac{\vdash X \prec y}{\vdash Y \ominus X \ominus 1 \succeq 0}\ \textbf{ltgeq}$$

$$\frac{\vdash X \approx Y}{\vdash X \succeq Y}\ \textbf{eqgeq}\ \frac{}{\vdash 0 \succeq 0}\ \textbf{geq0}$$

$$\frac{\vdash X \succeq 0 \vdash Y \succeq 0 \vdash n \geq 0}{\vdash X \oplus n \otimes Y \succeq 0}\ \textbf{geqadd}$$

$$\frac{\vdash X \succeq 0 \quad \vdash Y \succeq 0 \quad m \otimes X \approx n \otimes Y \quad m \geq 0 - 1 - n \geq 0}{\vdash \bot}\ \textbf{falsei}$$

$$\frac{\begin{array}{ll} \vdash a \otimes (X \ominus Y) \approx \ominus Z & \vdash Z \succeq 0 \\ \vdash b \otimes (Y \ominus X) \approx \ominus T & \vdash T \succeq 0 \\ \vdash a \geq 0 & b \geq 0 \end{array}}{\vdash X \approx Y}\ \textbf{eqi}$$

Figure 5: Simplex-logic rules

Simplex-logic style : P, Q are proofs, r is the number of current row, $maxProof$ is an auxiliary function, `geqct(c)` is a construct meaning “c is non-negative”

```
mkEqProof(X,Y) =
        $r \leftarrow r + 1$
        fill row r with coefficients for $X \ominus Y$
        $(a, Z, P) \leftarrow$    $maxProof(r)$
        fill row r with coefficients for $Y \ominus X$
        $(b, T, Q) \leftarrow$    $maxProof(r)$
        $r \leftarrow r - 1$
        return eqi(   $arith(a \otimes (X \ominus Y), \ominus Z), P, geqct(a),$
                      $arith(b \otimes (Y \ominus X), \ominus T), Q, geqct(b)$ )
Solver2FOL style : the return statement changes and becomes
        return        lemma “equi”[“a” $\leftarrow a$, “b” $\leftarrow b$
                      “X” $\leftarrow X$, “Y” $\leftarrow Y$, “Z” $\leftarrow Z$
                      “T” $\leftarrow T$](P,Q)
```

Figure 6: Translation from Simplex-logic style to Solver2FOL (fragment)

set of constraints as the Simplex, Simplex is complete with respect to linear integer constraints.

In order to build Solver2FOL proofs using the Simplex, the only things one has to do is :

- keep the core of the Simplex decision procedure
- modify the proof-generating components so as to compose proofs using either *modus ponens* on Solver2FOL lemmas or lemma composition of Solver2FOL lemmas instead of rules composition of Simplex-logic rules. The example of such a modification is given on figure 6.

This transformation procedure is straightforward - and probably automatizable, once the lemmas have been proved. As a matter of fact, it does not require understanding the algorithm itself, only changing its return statements.

6 Properties of Solver2FOL

6.1 Representation power

6.1.1 Introduction

It is clear that any set of inequalities and equalities involving only integer linear relations can be represented using Solver2FOL. It is also clear that polynomial constraints can be represented using Solver2FOL.

A valid question is : can more complex conditions be represented using Solver2FOL ?

$$x \longmapsto \lfloor f(x) = 2^x \rfloor$$

or

$$\bigvee_{i=0}^{i=2^N} (x \approx i \wedge f(x) \approx 2^i) \vee (x \prec 0 \wedge f(x) \approx 0)$$

Figure 7: A represented non-representable function.

6.1.2 Expressing complex functions through brute force

A function such as $n \mapsto \lfloor 2^n \rfloor$ cannot be represented as a finite expression in Solver2FOL’s expression language on $\mathbb{N}$.

However, for any value of N, as figure 7 shows, it can be represented as one disjunction of $\mathcal{O}(2^N)$ conjunctions of equalities on $]-2^N, 2^N]$ plus one inequality. If we give to N a good value, say the size in bits of an memory word, we have, as far as the computer is concerned, represented our function.

As a matter of fact, all functions can be represented with Solver2FOL using such a trick, provided they stay into the integer domain of the computer. However, since this form potentially requires as many disjunctions as there are integer which can be represented by the machines, or as there are memory words which can be addressed by the hardware, this simple remark does not mean much about the expressiveness of the language : such representations are, computationally speaking, infinite.

6.1.3 Lessons from the brute-force approach

The brute force approach teaches us that the real limitation of the expressiveness using Solver2FOL is in fact a size limitation.

Disjunctive Normal Forms

Property 1 *Let us consider $t : k \mapsto 2^k$. Let $[0, B[$ be the range of addressable memory in a machine M. Let S be the memory size of an representation of t in Disjunctive Normal Form (DNF) using Solver2FOL for array bounds checking elimination or dynamic memory protection elimination on M.*

$$S > B$$

Corollary : As such, t cannot be represented in DNF using Solver2FOL.

Proof of property 1 *Let $]-R, R]$ be the range of representable integers on a given computer.*

The simple study conducted in annex B.1 proves that any representation of function t in Disjunctive Normal Form using Solver2FOL will require at least $R/4$ terms.

If we assume that Solver2FOL is used for dynamic array bounds checking elimination or for dynamic memory checking elimination, we need the range of computable integers to be able to represent the whole range of addressable memory. In other words, $R \geq B$.

$$T(x,y) \equiv_{def} \begin{cases} & x \prec 0 \wedge y \approx 0 \\ \vee & x \approx 0 \wedge y \approx 1 \\ \vee & T(x \ominus 1, z) \wedge z \approx y \backslash 2 \end{cases}$$

Figure 8: A possible definition of $t : n \mapsto 2^n$

If we assume that each term will require at least eight bytes of memory allocation, the total amount of memory required for representation of t *will be at least* $S = 2 \cdot R$.

Hence,

$$S > B$$

Recursivity However, as shown in figure 8 a logic with some notion of recursive definition would allow a simple definition of function t in DNF. Such a logic would allow a more complete or/and simpler expression language.

6.1.4 Summary

Solver2FOL's expression language is powerful enough to represent linear and polynomial expressions. Since most operations which require integer constraints solving, such as array bounds checking optimization of automatic loops parallelization, most often imply only linear constraints, seldom polynomials, and almost never more complicated expressions, this language can be considered complete enough.

However, more specialized security policies might require even more expressive languages. We handle this issue in section 7.2.

6.2 Expressiveness of proofs

As proved in section 5, Solver2FOL is complete with respect to linear integer constraints.

What can be said beyond that ?

Well, we can say that Solver2FOL knows no such thing as an induction proof. This means that theorems handling generalized sums, such as lemma 7, cannot be expressed entirely within Solver2FOL's logic. *They must rely on a meta-logic and, for each given sum met during a proof, they must be instantiated for the number of terms in the sum.*

This means that Solver2FOL will produce numerous large proofs where there could have been only one relatively simple proof in a proof library. This issue is also handled in section 7.2 .

7 From theory to implementation

7.1 Integrating Solver2FOL in a real system

So far, Solver2FOL has not been introduced as a part of a real system. We plan to incorporate this work into the FLINT system [15] by extending the typed intermediate language with dependent types. In the following, we briefly discuss how Solver2FOL may be integrated with some existing dependently typed languages.

Xi's dependent type system [18], which has been used for array-bounds checking elimination, relies on an external constraints solver, symbolized by $\models$ in Xi [21]. In practice, for integers, this constraints solver is some (simplified) variant of the Omega test.

Since we are replacing the Omega test by Omega2FOL, one of the next natural steps would be to insert Omega2FOL in the dependent type system. However, this is not as trivial as it sounds, since an implicit requirement on the external constraints solver seems to be completion. A lack of completion resulting in an incomplete type-checker and, as far as PCC is concerned, the rejection of perfectly sound programs for arbitrary reasons. Of course, the Omega test is not complete on the set of diophantine equations. No algorithm is.

So, how do we introduce Solver2FOL in here ? A convenient answer would be that if Solver2FOL can do everything the Omega test can, then of course it can serve as the external solver in a dependent type system. However, this is not true. Since the external solver is supposed complete, neither the Omega test nor Omega2FOL nor any other algorithm is powerful enough as to serve as such a solver.

Before introducing effectively Omega2FOL in the dependent type system, we will have to determine exactly what set of constraints can be solved with Omega2FOL or, for that matter, any *2FOL solver.

Or we can use another approach. After all, the whole point of Solver2FOL is not to be locked with any solver algorithm, even if it is a *2FOL solver. Turning the dependent type system from Omega test-dependent to Omega2FOL-dependent is only a small improvement.

The idea would be, simply put, to make the dependent type system solver-independent. In other words, replacing $\models$ by $\vdash$. Replacing "my algorithm can prove this" by "there is an algorithm which can prove this, here is how". And this algorithm may be Omega2FOL, Simplex2FOL, *2FOL. Or Twelf [12], or Coq [4], or the user himself. And the algorithm does not have to be trusted.

This is one of the next steps in the evolution of Solver2FOL.

7.2 Twelf

We have begun working on integration using the Edinburgh Logical Framework (LF) [2] through Twelf. In other words, we are trying to encode the whole Solver2FOL system into LF and to translate proofs into the Twelf system. This task has been undertaken with the support of Princeton's *PCC team*, using PCC team's meta-logic for Twelf.

Using some meta-language (such as LF) and some concrete implementation of it (like Twelf) to describe Solver2FOL allows to turn Solver2FOL into a really trustable and extensible part of the trusted computing base. Extensions to Solver2FOL are new definitions, such as new operators, and new hard-to-prove-but-fully-trusted "axioms". For example, Fermat's theorem might be needed some day or maybe some specialized theorems related to bank accounting.

Additionally, the Twelf meta-logic we are using allows us a reasoning of somewhat higher order. Axioms of induction on $\mathbb{N}$, for example, let us overcome the size problem of induction proofs (cf. section 6.2). New functions can be defined and seamlessly integrated, which is a solution to the potential expressiveness problem presented in section 6.1.4.

8 Future work

8.1 Implementation

Our implementation of Solver2FOL in Twelf is not complete yet, since the PCC team's libraries we are basing our work on are still in early development. For example, since the libraries do not include full support for lists yet, and since Σ sums rely on lists, we have had to pause development to contribute to this non-trivial part of the libraries.

8.2 Extensions to non-linear constraints

Recent versions of the Omega test and of other tests handle some limited kinds of non-linear constraints, using several techniques such as polynomials factoring or linear approximation on segments. We are planning to prove as many as possible of these extensions using Solver2FOL and to add them to Omega2FOL.

8.3 Exceptions handling

As explained earlier, the semantics of Solver2FOL do not handle integer arithmetic exceptions in a fully satisfactory way. One satisfactory way would be to introduce some "non-bounded" integers, using either either unary integers or some fancy Big Integer construct. The first possibility would be simpler, hence easier to trust.

Another way would be to create proof templates instead of proofs. In this case, the templates would be instantiated at check-time with platform-dependent information. We have not pursued this trail any further yet. It promises to be very complicated as well as very rewarding.

9 Related work

The idea of PCC was first proposed by Necula and Lee [10]. To handle integer constraints, they introduce the Simplex-Logic, a form of minimalistic logic tailored for proving results obtained by the Simplex algorithm. However, since this logic is very specific, they can neither express proofs using any other algorithm or extend their algorithm to more generic constraints. Xi [21, 19], on the other hand, proposes a dependent type system, which totally abstracts the the solver algorithm, considering it as a prerequisite. Several other works require integer constraints solving algorithms in PCC and would benefit from a generic framework. Among them, TAL [7], Crary and Weirich [5]'s resource bound certifitation, Wang and Appel [17]'s safe garbage collection and an attempt at making the whole PCC system more generic proposed by Appel and Felty [1].

10 Conclusion

We presented a feasibility study for the elaboration of an algorithm independent framework for verifying and proving integer constraints. Our framework, Solver2FOL, is simple and can represent all commonly used integer constraints. Moreover, although its proof power is necessarily limited, it can be used at least on the whole range of linear integer constraints, which are by far the most common constraints encountered at compilation-time. We also presented our solution for overcoming expressiveness limitations and overgrown proofs size for its implementation in Twelf. We plan to implement this framework into the FLINT system [15].

References

[1] A. Appel and A. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 243–253. ACM Press, 2000.

[2] A. Avron, F. Honsell, and I. Mason. An overview of the edinburgh logical framework, 1989.

[3] W. W. Bledsoe. The Sup-Inf method in praesburger arithmetic. Technical report, University of Texas Math Dept., December 1974.

[4] C. Cornes. Compilation du filtrage de motifs avec types dépendants dans le système coq. In *Actes des Journées du GDR Programmation 1996*, Orleans, France, Novembre 1996.

[5] K. Crary and S. Weirich. Resource bound certification. In *Proc. 27th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. ACM Press, 2000.

[6] G. B. Dantzig and B. C. Eaves. Fourier-Motzkin elimination and its dual. *J. Combin. Theo. A*, 14:288–297, 1973.

[7] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. 25rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 85–97. ACM Press, 1998.

[8] G. Necula. Proof-carrying code. In *Twenty-Fourth Annual ACM Symp. on Principles of Prog. Languages*, pages 106–119, New York, Jan 1997. ACM Press.

[9] G. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In *Special Issue on Mobile Agent Security: LNCS Vol 1419*. Springer-Verlag, October 1997.

[10] G. C. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Sept. 1998.

[11] G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, 1981.

[12] F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[13] W. Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–115, August 1992.

[14] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the Sixth Annual Workshop on rogramming Languages and Compilers for Parallel Computing*, Dec 93.

[15] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*. Published as Boston College Computer Science Dept. Technical Report BCCS-97-03, June 1997.

[16] D. Teller. Algorithm-independent framework for verifying integer constraints. Technical report, Dept. of Computer Science, Yale University, New Haven, CT, March 2000. Available at URL `flint.cs.yale.edu/flint/publications`.

[17] D. Wang and A. Appel. Safe garbage collection = regions + intentional type analysis. Technical report, Dept. of Computer Science, Princeton University, July 1999.

[18] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Sept. 1998.

[19] H. Xi and R. Harper. A dependently typed assembly language. Technical Report CMU-CS-99-xxx, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1999.

[20] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proc. ACM SIGPLAN '98 Conf. on Prog. Lang. Design and Implementation*, pages 249–257, New York, 1998. ACM Press.

[21] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.

A Proofs built using Solver2FOL

A.1 Foreword

This appendix contains purely formal proofs built using Solver2FOL. However, since the formal proofs are extremely large, we could only present a handful of then in the article version of this presentation. The full proofs can be found in the companion technical report [16] of this paper, available at the FLINT web site: `http://flint.cs.yale.edu/flint/publications`.

A.2 Syntactical conventions

- A name or a number alone in a formal proof is a reference to the corresponding lemma, theorem, axiom, definition. Only sub-lemmas are referred to by their number. Other lemmas, theorems, axioms, definitions are referred to by their name.
- Notation $\dfrac{\{\text{somelemma}\}}{H \vdash \blacksquare}$ is a shortcut for the utilization of the *modus ponens* on an instantiation of lemma (or theorem, or axiom, ...) *sommelemma*.

- $\Sigma_{i=1}^{i=n} X_i$ is a shortcut for $(((X_1 \oplus X_2)\oplus)\cdots X_n)$
- $\Sigma_{i\in I} X_i$ is a shortcut for some left-parenthesized sum (using $\oplus$) of all terms of $(X_i)_{i\in I}$ [1]

A.3 Basic theorems

Proof for basic theorem 1 *Let* $H \equiv_{def} 0 \le e_\wedge e < b_\wedge a \approx b\otimes c\oplus e$ *and* $K \equiv_{def} 0 \le d_\wedge d < b_\wedge a = b\cdot c + d$.

Sub lemma 1 $\vdash b\otimes c\oplus e\approx b\cdot c\oplus e$

Proof of sub lemma 1

$$\dfrac{\dfrac{\{\text{“mult on }\mathbb{Z}\text{”}\}}{\vdash b\otimes c\approx b\cdot c} \quad \dfrac{\{\text{“add eq”}\}}{\vdash \blacksquare}}{\vdash b\otimes c\oplus e\approx b\cdot c\oplus e}$$

Sub lemma 2 $H \vdash a\approx b\cdot c\oplus e$

Proof of sub lemma 2

$$\dfrac{\dfrac{H\vdash a\approx b\otimes c\oplus e \quad \dfrac{\{1\}}{\vdash b\otimes c\oplus e\approx b\cdot c\oplus e}}{H\vdash a\approx b\otimes c\oplus e_\wedge b\otimes c\oplus e\approx b\cdot c\oplus e} \quad \dfrac{\{\text{“eq trans”}\}}{H\vdash\blacksquare}}{H\vdash a\approx b\cdot c\oplus e}$$

Sub lemma 3 $H\vdash a = b\cdot c + e$

Proof of sub lemma 3

$$\dfrac{\dfrac{\dfrac{\dfrac{\{2\}}{H\vdash a\approx b\cdot c\oplus e}\quad \dfrac{\{\text{“add on }\mathbb{Z}\text{”}\}}{H\vdash b\cdot c\oplus e\approx b\cdot c+e}}{H\vdash a\approx b\cdot c\oplus e_\wedge b\cdot c\oplus e\approx b\cdot c+e}\quad\dfrac{\{\text{“eq trans”}\}}{H\vdash\blacksquare}}{H\vdash a\approx b\cdot c+e}\quad \dfrac{\{\text{“eq on }\mathbb{Z}\text{”}\}}{H\vdash\blacksquare}}{H\vdash a=b\cdot c+e}$$

Sub lemma 4 $a\backslash b\approx c\vdash \lfloor\frac{a}{b}\rfloor = c$

Proof of sub lemma 4

$$\dfrac{\dfrac{\dfrac{H\vdash 0\le e\quad H\vdash e<b\quad\dfrac{\{3\}}{H\vdash a=b\cdot c+e}}{H\vdash 0\le e_\wedge e<b_\wedge a=b\cdot c+e}}{\exists e, 0\le e_\wedge e<b_\wedge a\approx b\otimes c\oplus e\vdash \exists e,0\le e_\wedge e<b_\wedge a=b\cdot c+e}}{a\backslash b\approx c\vdash\lfloor\frac{a}{b}\rfloor=c}$$

Sub lemma 5 $\vdash b\cdot c+d\approx b\otimes c\oplus d$

Proof of sub lemma 5

$$\dfrac{\dfrac{\dfrac{\dfrac{\{1\}}{\vdash b\otimes c\oplus d\approx b\cdot c\oplus d}\quad\dfrac{\{\text{“add in }\mathbb{Z}\text{”}\}}{\vdash b\cdot c\oplus d\approx b\cdot c+d}}{\vdash b\otimes c\oplus d\approx b\cdot c\oplus d_\wedge b\cdot c\oplus d\approx b\cdot c+d}}{\vdash b\otimes c\oplus d\approx b\cdot c+d}\quad\dfrac{\{\text{“eq ref”}\}}{\vdash\blacksquare}}{\vdash b\cdot c+d\approx b\otimes c\oplus d}$$

Sub lemma 6 $K\vdash a\approx b\otimes c\oplus d$

Proof of sub lemma 6

$$\dfrac{\dfrac{\dfrac{K\vdash a=b\cdot c+d}{K\vdash a\approx b\cdot c+d}\quad\dfrac{\{5\}}{K\vdash b\cdot c+d\approx b\otimes c\oplus d}}{K\vdash a\approx b\cdot c+d_\wedge b\cdot c+d\approx b\otimes c\oplus d}\quad\dfrac{\{\text{“eq trans”}\}}{K\vdash\blacksquare}}{K\vdash a\approx b\otimes c\oplus d}$$

Sub lemma 7 $\lfloor\frac{a}{b}\rfloor = c\vdash a\backslash b\approx c$

Proof of sub lemma 7

$$\dfrac{\dfrac{\dfrac{K\vdash 0\le d\quad K\vdash d<b\quad\dfrac{\{6\}}{K\vdash a\approx b\otimes c\oplus d}}{K\vdash 0\le d_\wedge d<b_\wedge a\approx b\otimes c\oplus d}}{\exists d,0\le d_\wedge d<b_\wedge a=b\cdot c+d\vdash\exists d,0\le d_\wedge d<b_\wedge a\approx b\otimes c\oplus d}}{\lfloor\frac{a}{b}\rfloor=c\vdash a\backslash b\approx c}$$

Hence,

$$\dfrac{\dfrac{\dfrac{\dfrac{\{7\}}{\lfloor\frac{a}{b}\rfloor=c\vdash a\backslash b\approx c}}{\vdash\lfloor\frac{a}{b}\rfloor=c\Rightarrow a\backslash b\approx c}\quad\dfrac{\dfrac{\{4\}}{a\backslash b\approx c\vdash\lfloor\frac{a}{b}\rfloor=c}}{\vdash a\backslash b\approx c\Rightarrow\lfloor\frac{a}{b}\rfloor=c}}{\vdash a\backslash b\approx c\Rightarrow\lfloor\frac{a}{b}\rfloor=c_\wedge\lfloor\frac{a}{b}\rfloor=c\Rightarrow a\backslash b\approx c}}{\vdash a\backslash b\approx c\iff\lfloor\frac{a}{b}\rfloor=c}$$

Proof for basic theorem 2 *Let* $H\equiv_{def} 0\le e_\wedge e<b_\wedge a\approx b\otimes c\oplus e$ *and* $K\equiv_{def} 0\le e_\wedge e<b_\wedge a=b\cdot c+e$.

Sub lemma 8 $a\%b\approx e\vdash a\ \mathtt{mod}\ b = e$

Proof of sub lemma 8

$$\dfrac{\dfrac{\dfrac{H\vdash 0\le e\quad H\vdash e<b\quad\dfrac{\{3\}}{H\vdash a=b\cdot c+e}}{H\vdash 0\le e_\wedge e<b_\wedge a=b\cdot c+e}}{\exists c,0\le e_\wedge e<b_\wedge a\approx b\otimes c\oplus e\vdash\exists d,0\le e_\wedge e<b_\wedge a=b\cdot d+e}}{a\%b\approx e\vdash a\ \mathtt{mod}\ b=e}$$

Sub lemma 9 $a\ \mathtt{mod}\ b = e\vdash a\%b\approx e$

Proof of sub lemma 9

$$\dfrac{\dfrac{\dfrac{K\vdash 0\le e\quad K\vdash e<b\quad\dfrac{\{6\}}{K\vdash a\approx b\otimes d\oplus e}}{K\vdash 0\le e_\wedge e<b_\wedge a\approx b\otimes d\oplus e}}{\exists c,0\le e_\wedge e<b_\wedge a=b\cdot d+e\vdash\exists c,0\le e_\wedge e<b_\wedge a\approx b\otimes c\oplus e}}{a\ \mathtt{mod}\ b=e\vdash a\%b\approx e}$$

Hence,

$$\dfrac{\dfrac{\dfrac{\dfrac{\{8\}}{a\%b\approx e\vdash a\ \mathtt{mod}\ b=e}}{\vdash a\%b\approx e\Rightarrow a\ \mathtt{mod}\ b=e}\quad\dfrac{\dfrac{\{9\}}{a\ \mathtt{mod}\ b=e\vdash a\%b\approx e}}{\vdash a\ \mathtt{mod}\ b=e\Rightarrow a\%b\approx e}}{\vdash a\%b\approx e\Rightarrow a\ \mathtt{mod}\ b=e_\wedge a\ \mathtt{mod}\ b=e\Rightarrow a\%b\approx e}}{\vdash a\%b\approx e\iff a\ \mathtt{mod}\ b=e}$$

[1]The left-parenthesized convention has been adopted for the sake of simplicity of proofs. However, there is no obligation of expressing sums like this.

A.4 Main lemmas

Proof for main lemma 1

$$\dfrac{\dfrac{\{\textit{“eq mult”}\}}{\vdash Y \ominus Y \approx 0 \otimes Y} \quad \dfrac{\{\textit{“0 mult”}\}}{\vdash 0 \otimes Y \approx 0} \quad \dfrac{\{\textit{“eq trans”}\}}{\vdash \blacksquare}}{\vdash Y \ominus Y \approx 0}$$

Proof for main lemma 2 ***Sub lemma 10*** $X \approx Y \vdash X \oplus Z \approx Z \oplus Y$

Proof of sub lemma 10

$$\dfrac{\dfrac{\dfrac{\{\textit{“add eq”}\}}{X \approx Y \vdash X \oplus Z \approx Y \oplus Z} \quad \dfrac{\{\textit{“add comm”}\}}{\vdash Z \oplus Y \approx Y \oplus Z}}{X \approx Y \vdash X \oplus Z \approx Y \oplus Z \wedge Z \oplus Y \approx Y \oplus Z} \quad \dfrac{\{\textit{“eq trans”}\}}{\vdash \blacksquare}}{X \approx Y \vdash X \oplus Z \approx Z \oplus Y}$$

Hence,

$$\dfrac{\dfrac{\dfrac{\{\textit{10}\}}{\vdash Z \oplus X \approx X \oplus Z} \quad \dfrac{\{\textit{“add comm”}\}}{\vdash Z \oplus X \approx X \oplus Z}}{X \approx Y \vdash X \oplus Z \approx Z \oplus Y \wedge Z \oplus X \approx X \oplus Z} \quad \dfrac{\{\textit{“eq trans”}\}}{\vdash \blacksquare}}{X \approx Y \vdash Z \oplus X \approx Z \oplus Y}$$

B Proofs on Solver2FOL

B.1 Expression through Disjunctive Normal Form

Property 2 *Let $]-R, R]$ be the range of representable integers for a given computer.*

Let us consider function $t : x \mapsto 2^x$

In Solver2FOL's language, any expression of function f as a Disjunctive Normal Form (DNF) which is exact on $[0, R]$ will require at least $R/4$ elementary conjunctions.

Proof of property 2 *Suppose that t can be represented on $[0, R]$ as*

$$T(x, y) \equiv_{def} \bigvee_{i=1}^{i=n} C_i(x, y)$$

where

$$T(x, y) \equiv y = t(x)$$

and $C_i(x, y)$ is a conjunction of elementary (linear) constraints.

Suppose $n < R/4$.

This means that at least one $C_i(x, y)$ will hold a constraint which will be valid for at least 3 values of x. Additionally, since $C_i(x, y)$ is in normal form, this means that these constraints define an integer interval and that the value of y will be defined by the same formula.

In other words,

$$\left\{ \begin{array}{lcl} y & = & a \cdot x + b \\ 2 \cdot y & = & a \cdot (x + 1) + b \\ 4 \cdot y & = & a \cdot (x + 2) + b \end{array} \right.$$

A resolution of this system gives :

$$a = 0, b = 0, y = 0$$

Since all values of x considered are in $[0, R]$, we necessarily have $y > 0$. Hence, contradiction.

Since $n > R/4$, there are at least $R/4$ elementary conjunctions.