Certified Reasoning with Infinity

Asankhaya Sharma* Shengyi Wang* Andreea Costea*
Aquinas Hobor†,* Wei-Ngan Chin*

{asankhs, shengyi, andreeac, hobor, chinwn} @comp.nus.edu.sg
*School of Computing and †Yale-NUS College, National University of Singapore

Abstract. We demonstrate how infinities improve the expressivity, power, readability, conciseness, and compositionality of a program logic. We prove that adding infinities to Presburger arithmetic enables these improvements without sacrificing decidability. We develop Omega++, a Coq-certified decision procedure for Presburger arithmetic with infinity and benchmark its performance. Both the program and proof of Omega++ are parameterized over user-selected semantics for the indeterminate terms (such as $0 * \infty$).

1 Introduction

Formal software analysis and verification frameworks benefit from expressive, compositional, decidable, and readable specification mechanisms. Of course, these goals often conflict with each other: for example, it is easy to add expressivity if one is willing to give up decidability! Happily, we have found a free lunch: by adding the notion of "infinity" to the specification language we can usefully add to the expressivity, readability, and compositionality of our specifications while maintaining their decidability.

Specifically, we start from the well-established domains of separation logic [25] and Presburger arithmetic [24] and add two abstract/fictitious/ghost symbols ∞ and $-\infty$, for which we support a precise, well-defined semantics. Although a seeming-minor addition, these symbols add significantly to the expressivity and power of our logic.

In section 2.3, we use infinities to increase the compositionality of our logic by showing that "lists" and "bounded lists" are equivalent when the bound is ∞. Moreover, in section 2.4, we use ∞ to mix notions of partial and total correctness within a logic.

Infinities also add to our specification framework's readability and conciseness. For example, we will see in section 2.2 that ∞ allows us to drop disjuncts in the specification for code that manipulates a sorted linked list.

Finally, infinities enable some interesting applications. In section 2.5, we apply the notion of quantifier elimination in Presburger arithmetic with infinities to infer pure (non-heap) properties of programs.

All of the previous gains are worthy in their own right, but our major technical advance is the development of Omega++, a sound and complete decision procedure for Presburger arithmetic with infinities (including arbitrary quantifier use). In other words, we do not sacrifice any of the computational advantage normally gained by restricting ourselves to Presburger arithmetic, despite the addition of infinities. We call our tool "Omega++" both to acknowledge the importance of the underlying Presburger solver Omega [9] and because we believe we have modestly incremented its utility.

Omega++ is written in Gallina, the specification language of Coq [1], allowing us to formally certify it (modulo the correctness of Omega itself, which we utilize as our backend). We extract our performance-tuned Gallina into OCaml and package it as a library, which we have benchmarked using the HIP/SLEEK verification toolset [5].

One notable technical feature of Omega++ is that it can handle several semantic variants of Presburger arithmetic with infinity. For example, Presburger arithmetic usually admits multiplication by a constant as a notational convenience, *e.g.* $3 \cdot x \overset{\text{def}}{=} x + x + x$. This obvious-seeming convenience becomes a little less obvious when one adds infinities: what is $0 \cdot \infty$? Mathematical sophisticates can—*and do*—disagree: some prefer 0 as a convention in certain contexts (including, reasonably, ours) [19], while others prefer the result to be undefined due to the indeterminate status of the corresponding limit forms [10]. When possible, Omega++ takes an agnostic approach to such disagreements by allowing the user to specify the semantics of some subtle cases. Omega++ is thus a certified compiler from a *set* of related source languages (Presburger arithmetics with infinities) to a fixed, well-understood target (vanilla Presburger).

Omega++ is available for download and experimentation here:
`http://loris-7.ddns.comp.nus.edu.sg/~project/SLPAInf/`

2 Motivation

In this section, we highlight the benefits of augmenting a specification logic with infinities. For consistency we focus on separation logic [6,25] but other specification mechanisms which rely on Presburger arithmetic can enjoy similar benefits.

2.1 Orientation

Our flavor of separation logic has its grounds in the HIP/SLEEK system [5], thus offering the convenience to test and benchmark with a state-of-the-art verification toolchain. Methods are specified with a pair of pre- and postcondition (Φ_{pr}, Φ_{po}), with the keyword `res` consistently used in the Φ_{po} to refer to the return value. We have enhanced the logic to allow the symbols ∞ and $-\infty$ where it would normally require integers; we also allow quantification over infinities.

From a systems perspective, our setup is sketched in figure 1. First, entailment between separation logic formulae with infinities in HIP/SLEEK is reduced (*à la* Chin *et al.* [5]) to entailment between numeric formulae in Presburger arithmetic with infinities (PAInf). Next, we translate PAInf to vanilla Presburger arithmetic (PA). We emphasize on this phase as being our main contribution and detail it in section 4.

Finally, we discharge PA proof obligations with Omega. There are other combinations of separation logic with extensions of PA (such as sets/multisets) that can be used to enhance the specification. We discuss them in section 7 as related work.

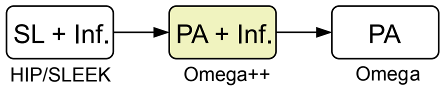

Fig. 1. Our setup: SL + Inf to PA

2.2 Infinities enable Concise Specifications

Let's start to see what infinities can buy us! Consider a simple program which inserts a new node into a sorted linked list, whose nodes are defined as follows:

$$\texttt{data node } \{\texttt{int val; node next;}\}$$

The data field `val` stores numerical information and the pointer field `next` points to the subsequent node in the structure. Consider the next two alternative inductive predicates which characterize sortedness using only a single numeric parameter[1] describing the list's minimum value:

Scenario 1 - no infinity enhancement:

$$\begin{aligned}&\texttt{sorted\_ll}\langle\texttt{root},\texttt{min}\rangle \equiv \texttt{root}{\mapsto}\texttt{node}\langle\texttt{min},\texttt{null}\rangle\\ &\quad\vee\ \exists\,\texttt{q},\ \texttt{mtail}\cdot(\texttt{root}{\mapsto}\texttt{node}\langle\texttt{min},\texttt{q}\rangle * \texttt{sorted\_ll}\langle\texttt{q},\texttt{mtail}\rangle \wedge \texttt{min}{\leq}\texttt{mtail})\end{aligned}$$

Scenario 2 - with infinity enhancement:

$$\begin{aligned}&\texttt{sorted\_ll}\langle\texttt{root},\texttt{min}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{min}{=}\infty)\\ &\quad\vee\ \exists\,\texttt{q},\ \texttt{mtail}\cdot(\texttt{root}{\mapsto}\texttt{node}\langle\texttt{min},\texttt{q}\rangle * \texttt{sorted\_ll}\langle\texttt{q},\texttt{mtail}\rangle \wedge \texttt{min}{\leq}\texttt{mtail})\end{aligned}$$

The base case of *Scenario 1* denotes a singleton, while its inductive case describes a linked list of length at least two. Though useable, this definition has a frustrating shortcoming: it cannot handle empty linked lists, since such lists do not have a finite minimum value. In contrast, *Scenario 2* handles the empty list gracefully since the minimum of an empty list can be defined to be just ∞! We could similarly use $-\infty$ to build a predicate which captures the maximum property of a linked list.

The code for `insert` is in figure 2. Parameter `x` points to a sorted linked list, while `y` is the data node we wish to insert (preserving sortedness). Notice that the pre/post specifications in Scenario 1 require disjunctions to separate the cases when `x` is empty and nonempty, whereas Scenario 2 handles both cases uniformly. Infinities thus enable more *concise* and *readable* (easy to maintain) specifications.

```
node insert(node x, node y){
  if (x == null) return y;
  else {
    if (y.val <= x.val){
      y.next = x;
      return y;
    } else {
      x.next = insert(x.next, y);
      return x;
} } }
```

Scenario 1 :

$$\begin{aligned}\Phi_{pr}:\ &\texttt{y}{\mapsto}\texttt{node}\langle\texttt{v},\texttt{null}\rangle \wedge \texttt{x}{=}\texttt{null}\\ &\vee\ \texttt{sorted\_ll}\langle\texttt{x},\texttt{a}\rangle * \texttt{y}{\mapsto}\texttt{node}\langle\texttt{v},\texttt{null}\rangle\\ \Phi_{po}:\ &\texttt{sorted\_ll}\langle\texttt{res},\texttt{b}\rangle \wedge \texttt{x}{=}\texttt{null} \wedge \texttt{b}{=}\texttt{v}\\ &\vee\ \texttt{sorted\_ll}\langle\texttt{res},\texttt{b}\rangle \wedge \texttt{b}{=}\texttt{min}(\texttt{a},\texttt{v})\end{aligned}$$

Scenario 2 :

$$\begin{aligned}\Phi_{pr}:\ &\texttt{sorted\_ll}\langle\texttt{x},\texttt{a}\rangle * \texttt{y}{\mapsto}\texttt{node}\langle\texttt{v},\texttt{null}\rangle\\ \Phi_{po}:\ &\texttt{sorted\_ll}\langle\texttt{res},\texttt{b}\rangle \wedge \texttt{b}{=}\texttt{min}(\texttt{a},\texttt{v})\end{aligned}$$

Fig. 2. Two pre-/post-specifications for insertion into a sorted linked list .

[1] Note that there are other ways of specifying sortedness, such as through the use of multi-set, that may also capture stronger properties, such as content preservation. However, they may require more complex provers in their reasoning.

2.3 Infinities increase Compositionality

Consider this definition for an n-node linked list whose values are bounded by b:

$$\mathtt{llB}\langle\mathtt{root},\mathtt{n},\mathtt{b}\rangle \equiv (\mathtt{root}{=}\mathtt{null}\ \wedge\ \mathtt{n}=0) \\ \vee(\exists\,\mathtt{q},\mathtt{v}\cdot\mathtt{root}{\mapsto}\mathtt{node}\langle\mathtt{v},\mathtt{q}\rangle * \mathtt{llB}\langle\mathtt{q},\mathtt{n}-1,\mathtt{b}\rangle\ \wedge\ \mathtt{v}\leq\mathtt{b})$$

Suppose we have a function f which uses this definition in its precondition:

$$\Phi_{pr}:\ \mathtt{llB}\langle\mathtt{x},\mathtt{n},\mathtt{m}\rangle * \ldots$$

where x points to a linked list bounded by m. Next, suppose we call f from a program point where the only available information involves the shape and length of a linked list x (that is, we have no information about its bound), *e.g.* we satisfy the predicate $\mathtt{ll}\langle\mathtt{x},\mathtt{n}\rangle$ as defined below:

$$\mathtt{ll}\langle\mathtt{root},\mathtt{n}\rangle \equiv (\mathtt{root}{=}\mathtt{null}\wedge\mathtt{n}{=}0)\vee\ \exists\,\mathtt{q}\cdot(\mathtt{root}{\mapsto}\mathtt{node}\langle\_,\mathtt{q}\rangle * \mathtt{ll}\langle\mathtt{q},\mathtt{n}-1\rangle)$$

With infinities this is easy: just instantiate m to ∞ since

$$\mathtt{ll}\langle\mathtt{x},\mathtt{n}\rangle \leftrightarrow \mathtt{llB}\langle\mathtt{x},\mathtt{n},\infty\rangle$$

Without infinities, however, this is not so easy since we must first determine an appropriate bound for x's values. Thus, infinities increase the *compositionality* of our logic, which in turn improves the reusability and conciseness of our specifications.

2.4 Infinities support Termination and Non-Termination Reasoning

Le *et al.* developed a technique to reason about termination and non-termination with a resource constraint $\mathtt{RC}\langle\mathtt{min},\mathtt{max}\rangle$ that tracks the minimum and maximum permitted execution steps [14]. Using Presburger arithmetic with infinity, terminating programs are modeled by $\mathtt{RC}\langle\_,\mathtt{max}\rangle\wedge\mathtt{max}{<}\infty$, while non-terminating programs are captured by $\mathtt{RC}\langle\infty,\infty\rangle$. Le *et al.* evaluated the semantics of non-termination reasoning with the help of Omega++.

```
int length(node x){
  if (x == null)
    return 0;
  else
    return (1 + length(x.next));
}
```

Termination Spec :
$\Phi_{pr}:\ \mathtt{ls}\langle\mathtt{x},\mathtt{null},\mathtt{n}\rangle * \mathtt{RC}\langle\_,\mathtt{M}\rangle\wedge\mathtt{n}{<}\mathtt{M}\wedge\mathtt{M}{<}\infty$
$\Phi_{po}:\ \mathtt{ls}\langle\mathtt{x},\mathtt{null},\mathtt{n}\rangle * \mathtt{RC}\langle\_,\mathtt{M}-(\mathtt{n}+1)\rangle\wedge\mathtt{res}{=}\mathtt{n}$

Non-Termination Spec :
$\Phi_{pr}:\ \mathtt{cll}\langle\mathtt{x},\mathtt{n}\rangle * \mathtt{RC}\langle\infty,\infty\rangle$
$\Phi_{po}:\ \mathtt{false}$

Fig. 3. Example 4: `length` terminates on proper lists and diverges on cyclic lists

Figure 3 demonstrates these resource constraints on a `length` function for linked lists. We show two specifications: the first shows that `length` terminates on finite lists `ls`, and the second shows that `length` diverges on circular lists `cll`, where `ls` and `cll` are defined as below:

$$\begin{aligned}
\mathtt{ls}\langle\mathtt{root},\mathtt{p},\mathtt{n}\rangle &\equiv (\mathtt{root}{=}\mathtt{p} \wedge \mathtt{n}{=}0)\\
&\quad \vee\ \exists\,\mathtt{q}\cdot(\mathtt{root}{\mapsto}\mathtt{node}\langle\_,\mathtt{q}\rangle * \mathtt{ls}\langle\mathtt{q},\mathtt{p},\mathtt{n}-1\rangle \wedge \mathtt{root}{\neq}\mathtt{p})\\
\mathtt{cll}\langle\mathtt{root},\mathtt{n}\rangle &\equiv \exists\,\mathtt{q}\cdot(\mathtt{root}{\mapsto}\mathtt{node}\langle\_,\mathtt{q}\rangle * \mathtt{ls}\langle\mathtt{q},\mathtt{root},\mathtt{n}-1\rangle)
\end{aligned}$$

2.5 Infinities support Analysis via Quantifier Elimination

Algorithmic quantifier elimination (QE) is a powerful technique for decision procedures in symbolic logic [8]. Kapur *et al.* highlight the importance of geometric QE heuristics for the case of generating program invariants [7]. While they exploit the structure of verification conditions generated from numerical programs, our PAInf-based QE allows us to generate inductive invariants (e.g. using octagonal constraints with infinity: $-\infty \leq \pm x \pm y \leq \infty$) for programs manipulating dynamically allocated data structures.

```
void append(node x, int a){
  if (x.next == null)
    x.next = new node(a, null);
  else
    insert(x.next, a);
}
```

Shape Spec :
Φ_{pr} : $\mathtt{ll}\langle\mathtt{x},\_\rangle \wedge \mathtt{x}{\neq}\mathtt{null}$
Φ_{po} : $\mathtt{ll}\langle\mathtt{x},\_\rangle \wedge \mathtt{x}{\neq}\mathtt{null}$

Spec with Inferred Pure :
Φ_{pr} : $\mathtt{ll}\langle\mathtt{x},\mathtt{n}\rangle \wedge \mathtt{n}{>}0$
Φ_{po} : $\mathtt{ll}\langle\mathtt{x},\mathtt{n}+1\rangle \wedge \mathtt{n}{>}0$

Fig. 4. Pure Specification Inferred from PAInf QE

Consider for example the code in figure 4, which appends a node to the end of an acyclic linked list. The Shape Spec does not express the strongest verifiable post-condition as it does not account for the newly inserted node. It would be thus useful to infer size properties as well. We can do so if the verification's relational obligations are discharged by QE over PAInf, leading to the specification with numeric properties.

3 Syntax and Parameterized Semantics

There are several benefits of adding the notion of infinity to a program logic. However, due to the presence of certain terms like $(\infty - \infty)$, it is an interesting problem to define the correct (or rather desired) semantics. We will now proceed to a formal discussion of Presburger arithmetic with infinity.

Our constraint language extends Presburger arithmetic with two abstract symbols designating positive (∞) infinity and negative $(-\infty)$ infinity. The language is detailed in figure 5. However, we would like to make some extra notes. First, we use a type based approach to distinguish between the domain of variables. The notation $w : \tau$ denotes that the variable w is of type τ; thus there is a clear distinction between the domain of variables. Second, for performance reasons that are explained in section 5 we do not aim for a minimal input constraint language. That is the reason why the input language also supports min and max constraints over expressions. The min and max constraints in the input language are translated to $\min_=$ and $\max_=$ (using $\pi \rightsquigarrow [v/\max(a_1,a_2)]\pi \wedge \max_=(v,a_1,a_2)$ and $\pi \rightsquigarrow [v/\min(a_1,a_2)]\pi \wedge \min_=(v,a_1,a_2)$).

$$
\begin{array}{rl}
\pi & ::= \beta \mid \neg\pi \mid \pi_1{\wedge}\pi_2 \mid \pi_1{\vee}\pi_2 \mid \pi_1{\rightarrow}\pi_2 \mid \exists(w:\tau)\cdot\pi \mid \forall(w:\tau)\cdot\pi \\
\beta & ::= \texttt{true} \mid \texttt{false} \mid a_1{<}a_2 \mid a_1{\leq}a_2 \mid a_1{=}a_2 \mid a_1{\neq}a_2 \\
 & \quad \mid a_1 \geq a_2 \mid a_1 > a_2 \\
a & ::= k \mid v \mid c{\times}a \mid a_1+a_2 \mid -a \mid a_1-a_2 \mid \max(a_1,a_2) \mid \min(a_1,a_2) \\
k & ::= c \mid \infty \mid -\infty
\end{array}
$$

where v, w are variable names; c is an integer constant

Fig. 5. PAInf: Input Constraint Language

Next, we present the parameterized semantic model for PAInf and establish theorems and lemmas that show the correctness of our decision procedure. All theorems and lemmas in this paper are machine checked in Coq. Parameters are introduced to adapt different possible ways of handling tricky parts of PAInf such as the terms $(\infty-\infty)$ and $(0\times\infty)$. Since our semantics is parameterized, all procedures, theorems and lemmas based on the semantics are also parameterized. We start by defining an *environment* to map variables to values.

Definition 1 *An environment for a universe τ of concrete values is a function $\phi_\tau : V \rightarrow \tau$ from the set of variables V to τ. For such a ϕ_τ, we denote by $\phi_\tau[x \mapsto a]$ the function which maps x to a and any other variable y to $\phi_\tau(y)$.*

We define the semantics of arithmetic operations and relations for PAInf formally in figure 6, denoted by $[\![\beta]\!]_{\mathbb{Z}_\infty}$. The subscript of $[\![]\!]$ denotes the domain of constants. $\mathbb{Z}_\infty$ means $\mathbb{Z} \cup \{\infty, -\infty\}$. By analogy, $[\![\beta]\!]_{\mathbb{Z}}$ means the domain is $\mathbb{Z}$. With these definitions one can compute every atomic term into a truth value with respect to an environment ϕ_τ and domain of constants η as described in figure 7, and denoted by $\mathrm{E{\small VAL}}^{\eta}_{\phi_\tau}$.

We define the satisfaction relation $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi$ and dissatisfaction relation $\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi$ (in figure 8) for each logical formula π over the environment ϕ_τ and domain of constants η by structural induction on π. Sometimes, a formula π can neither be satisfied nor be dissatisfied. In that case, we say π is undetermined, which can be presented as $\phi_\tau \models^{\mathrm{udt}}_{\eta} \pi$. We define two distinct relations for satisfaction and dissatisfaction as we support both two-valued and three-valued logic. In case of three-valued logic a formula can be neither satisfied nor dissatisfied (undetermined).

Much of the semantics for PAInf is "as you might expect". For example, when all the values are finite, all of the operations and relations behave the same way they would in PA. On the other hand, any finite value plus ∞ equals ∞ and any finite value plus $-\infty$ equals $-\infty$. It is trickier to figure out what to do with the sum of ∞ and $-\infty$; we treat this as a meaningless value (much like the "value" of $\frac{0}{0}$ in the reals) denoted by "$\perp$". If ∞ and $-\infty$ were actually inverses, we would need to admit the following whopper:

$$0 \;=\; \infty + -\infty \;=\; \infty + (-\infty + 1) \;=\; (\infty + -\infty) + 1 \;=\; 1$$

In fact there is no perfect solution, since it is impossible to add a finite number of symbols to $\mathbb{Z}$ while remaining a group. Lasaruk and Sturm [13] propose dodging part of this problem by using only a single value for both positive and negative infinity, which is both greater than *and* less than all finite values. This approach ensures that every sum is defined, although ∞ still does not have an inverse and you lose antisymmetry for $\leq$. We find the notion of a single infinity to be too restrictive as it prohibits us from expressing some of the motivating examples from section 2.

[ADDITION]

$$\llbracket k_1 + k_2 \rrbracket_{\mathbb{Z}_\infty} \overset{\text{def}}{=} \begin{cases} \bot & k_1 \text{ or } k_2 \text{ is } \bot \\ \bot & k_1 = \infty, k_2 = -\infty \\ \bot & k_1 = -\infty, k_2 = \infty \\ \infty & k_1 \text{ or } k_2 \text{ is } \infty \text{, and neither is } -\infty \\ -\infty & k_2 \text{ or } k_2 \text{ is } -\infty \text{, and neither is } \infty \\ \llbracket k_1 + k_2 \rrbracket_{\mathbb{Z}} & k_1 \text{ and } k_2 \text{ are finite} \end{cases}$$

[LESS−THAN−EQ]

$$\llbracket k_1 \le k_2 \rrbracket_{\mathbb{Z}_\infty} \overset{\text{def}}{=} \begin{cases} \texttt{F/U} & k_1 \text{ or } k_2 \text{ is } \bot \\ \texttt{T} & k_2 = \infty \\ \texttt{T} & k_1 = -\infty \\ \texttt{T} & k_1 = k_2 = \infty \\ \texttt{T} & k_1 = k_2 = -\infty \\ \texttt{F} & k_1 = \infty, k_2 \neq \infty \\ \texttt{F} & k_1 \neq -\infty, k_2 = -\infty \\ \llbracket k_1 \le k_2 \rrbracket_{\mathbb{Z}} & k_1 \text{ and } k_2 \text{ are finite} \end{cases}$$

[IDENTITY]

$$\llbracket k \rrbracket_{\mathbb{Z}_\infty} \overset{\text{def}}{=} k$$

[NEGATION]

$$\llbracket -k \rrbracket_{\mathbb{Z}_\infty} \overset{\text{def}}{=} \begin{cases} \bot & k = \bot \\ \infty & k = -\infty \\ -\infty & k = \infty \\ \llbracket -k \rrbracket_{\mathbb{Z}} & k \text{ is finite} \end{cases}$$

[OTHER−OPERATIONS−AND−RELATIONS]

$$\llbracket 0 \times k \rrbracket_{\mathbb{Z}_\infty} \overset{\text{def}}{=} \begin{cases} 0 & k \text{ is finite} \\ 0/\bot/k & k \text{ is not finite} \end{cases} \qquad \llbracket c \times k \rrbracket_{\mathbb{Z}_\infty} \overset{\text{def}}{=} \begin{cases} \llbracket 0 \times k \rrbracket_{\mathbb{Z}_\infty} & c = 0 \\ k & c = 1 \\ \llbracket k + (c-1) \times k \rrbracket_{\mathbb{Z}_\infty} & c > 1 \\ \llbracket -((-c) \times k) \rrbracket_{\mathbb{Z}_\infty} & c < 0 \end{cases}$$

$$\llbracket k_1 \ge k_2 \rrbracket_{\mathbb{Z}_\infty} \overset{\text{def}}{=} \llbracket k_2 \le k_1 \rrbracket_{\mathbb{Z}_\infty} \qquad \llbracket k_1 > k_2 \rrbracket_{\mathbb{Z}_\infty} \overset{\text{def}}{=} \llbracket k_1 \ge k_2 \rrbracket_{\mathbb{Z}_\infty} \wedge \llbracket k_1 \neq k_2 \rrbracket_{\mathbb{Z}_\infty}$$

$$\llbracket k_1 \neq k_2 \rrbracket_{\mathbb{Z}_\infty} \overset{\text{def}}{=} \neg \llbracket k_1 = k_2 \rrbracket_{\mathbb{Z}_\infty} \qquad \llbracket k_1 = k_2 \rrbracket_{\mathbb{Z}_\infty} \overset{\text{def}}{=} \llbracket k_1 \le k_2 \rrbracket_{\mathbb{Z}_\infty} \wedge \llbracket k_2 \le k_1 \rrbracket_{\mathbb{Z}_\infty}$$

$$\llbracket k_1 - k_2 \rrbracket_{\mathbb{Z}_\infty} \overset{\text{def}}{=} \llbracket k_1 + (-k_2) \rrbracket_{\mathbb{Z}_\infty} \qquad \llbracket k_1 < k_2 \rrbracket_{\mathbb{Z}_\infty} \overset{\text{def}}{=} \llbracket k_1 \le k_2 \rrbracket_{\mathbb{Z}_\infty} \wedge \llbracket k_1 \neq k_2 \rrbracket_{\mathbb{Z}_\infty}$$

$$\llbracket \max_=(k_1, k_2, k_3) \rrbracket_{\mathbb{Z}_\infty} \overset{\text{def}}{=} (\llbracket k_1 = k_2 \rrbracket_{\mathbb{Z}_\infty} \wedge \llbracket k_3 \le k_2 \rrbracket_{\mathbb{Z}_\infty}) \vee (\llbracket k_1 = k_3 \rrbracket_{\mathbb{Z}_\infty} \wedge \llbracket k_2 \le k_3 \rrbracket_{\mathbb{Z}_\infty})$$

$$\llbracket \min_=(k_1, k_2, k_3) \rrbracket_{\mathbb{Z}_\infty} \overset{\text{def}}{=} (\llbracket k_1 = k_2 \rrbracket_{\mathbb{Z}_\infty} \wedge \llbracket k_2 \le k_3 \rrbracket_{\mathbb{Z}_\infty}) \vee (\llbracket k_1 = k_3 \rrbracket_{\mathbb{Z}_\infty} \wedge \llbracket k_3 \le k_2 \rrbracket_{\mathbb{Z}_\infty})$$

Fig. 6. Operations and Relations in $\mathbb{Z}_\infty$

In addition to the issues encountered while using a single infinity symbol, handling comparisons with $\bot$ is another challenge. A possible solution is treating all comparisons with $\bot$ as false. This is reasonable but not perfect. For example, in this context, it is not the case that $x > y$ is equivalent to $\neg(x \le y)$ when x or y are $\bot$. Interestingly, this is the choice made by IEEE floating point standard [2]. Another possibility is to use a three-valued logic and treat any comparison with $\bot$ as the "third unknown value". There are several three-valued logics studied in the literature [3]. We use Kleene's weak three-valued logic which interprets the unknown value as "Error" and propagates it to the entire formula. In three-valued logic, when x or y are $\bot$, $x > y$ and $\neg(x \le y)$ are equivalent. In Omega++, user can choose between a two-valued or three-valued logic, which is indicated in [LESS−THAN−EQ] of figure 6. Note that in three-valued logic, according to the relation definition in figure 8, formulae like $\bot < 0$ are neither satisfied nor dissatisfied.

The definition of multiplication in the presence of infinities ($0 \times \infty$) can also be selected by the user as shown in figure 6. There are three possible choices for defining

[**ARITH−EVAL**]

$$\textsc{Eval}^{\eta}_{\phi_\tau}(k) \overset{\text{def}}{=} [\![k]\!]_\eta \qquad \textsc{Eval}^{\eta}_{\phi_\tau}(v) \overset{\text{def}}{=} [\![\phi_\tau(v)]\!]_\eta$$
$$\textsc{Eval}^{\eta}_{\phi_\tau}(c \times a) \overset{\text{def}}{=} [\![\textsc{Eval}^{\eta}_{\phi_\tau}(c) \times \textsc{Eval}^{\eta}_{\phi_\tau}(a)]\!]_\eta$$
$$\textsc{Eval}^{\eta}_{\phi_\tau}(a_1 + a_2) \overset{\text{def}}{=} [\![\textsc{Eval}^{\eta}_{\phi_\tau}(a_1) + \textsc{Eval}^{\eta}_{\phi_\tau}(a_2)]\!]_\eta$$
$$\textsc{Eval}^{\eta}_{\phi_\tau}(a_1 - a_2) \overset{\text{def}}{=} [\![\textsc{Eval}^{\eta}_{\phi_\tau}(a_1) - \textsc{Eval}^{\eta}_{\phi_\tau}(a_2)]\!]_\eta$$
$$\textsc{Eval}^{\eta}_{\phi_\tau}(-a) \overset{\text{def}}{=} [\![-\textsc{Eval}^{\eta}_{\phi_\tau}(a)]\!]_\eta$$

[**BOOLEAN−EVAL**]

$$\textsc{Eval}^{\eta}_{\phi_\tau}(\texttt{true}) \overset{\text{def}}{=} \texttt{T} \qquad \textsc{Eval}^{\eta}_{\phi_\tau}(\texttt{false}) \overset{\text{def}}{=} \texttt{F} \qquad \textsc{Eval}^{\eta}_{\phi_\tau}(\texttt{undefined}) \overset{\text{def}}{=} \texttt{U}$$
$$\textsc{Eval}^{\eta}_{\phi_\tau}(a_1 \circ a_2) \overset{\text{def}}{=} [\![\textsc{Eval}^{\eta}_{\phi_\tau}(a_1) \circ \textsc{Eval}^{\eta}_{\phi_\tau}(a_2)]\!]_\eta$$
$$\textsc{Eval}^{\eta}_{\phi_\tau}(\max_=(a_1, a_2, a_3)) \overset{\text{def}}{=} [\![\max_=(\textsc{Eval}^{\eta}_{\phi_\tau}(a_1), \textsc{Eval}^{\eta}_{\phi_\tau}(a_2), \textsc{Eval}^{\eta}_{\phi_\tau}(a_2))]\!]_\eta$$
$$\textsc{Eval}^{\eta}_{\phi_\tau}(\min_=(a_1, a_2, a_3)) \overset{\text{def}}{=} [\![\min_=(\textsc{Eval}^{\eta}_{\phi_\tau}(a_1), \textsc{Eval}^{\eta}_{\phi_\tau}(a_2), \textsc{Eval}^{\eta}_{\phi_\tau}(a_2))]\!]_\eta$$

where $\circ$ above means one of $\leq, \geq, <, >, =, \neq$.

Fig. 7. Evaluations on atomic terms

$0 \times \infty$: 0, $\perp$ and ∞. For each of these options we can choose a two-valued or three-valued logic, thus Omega++ supports six different customized semantics in total. As described in section 6, for our experiments we use the semantics with three-valued logic and $0 \times \infty \overset{\text{def}}{=} 0$. However, in general any of the six customized semantics can be used as the decision procedure is parameterized over these choices and our certified proof guarantees that all choices are sound, complete and decidable.

In order to match the intuition of user, by design, most valid formulae in PA remain so in our semantics for PAInf, just as most invalid formulae in PA are still invalid in PAInf. Here are two short examples that are valid in both (if you drop the universe of quantification as you move from PAInf to PA):

$$\forall(x : \mathbb{Z}_\infty) \cdot \exists(y : \mathbb{Z}_\infty) \cdot x \leq y \qquad \forall(x : \mathbb{Z}_\infty) \cdot \forall(y : \mathbb{Z}_\infty) \cdot x + 1 = y + 1 \rightarrow x = y$$

However, there are differences. This formula is valid in PA but invalid in PAInf:

$$\forall(x : \mathbb{Z}_\infty) \cdot \exists(y : \mathbb{Z}_\infty) \cdot x + y = 0$$

The previous formula is false in PAInf when $x = \infty$. More generally, although $\mathbb{Z}_\infty$ is not a group, it still has many useful algebraic properties, such as the following.

Lemma 1. **$+$ is Associative** *$[\![(a + b) + c]\!]_{\mathbb{Z}_\infty}$ and $[\![a + (b + c)]\!]_{\mathbb{Z}_\infty}$ are equal or both undefined.*

Lemma 2. **$+$ is Commutative** *$[\![a + b]\!]_{\mathbb{Z}_\infty}$ and $[\![b + a]\!]_{\mathbb{Z}_\infty}$ are equal or both undefined.*

Lemma 3. **0 is the Additive Identity** *$[\![a + 0]\!]_{\mathbb{Z}_\infty}$ and a are equal for all defined a.*

Lemma 4. **$+$ is Monotonic** *If $[\![a \leq b]\!]_{\mathbb{Z}_\infty}$ is* T *and if both $[\![a + c]\!]_{\mathbb{Z}_\infty}$ and $[\![b + c]\!]_{\mathbb{Z}_\infty}$ are defined, then $[\![a + c \leq b + c]\!]_{\mathbb{Z}_\infty}$ is also* T.

$\phi_\tau \models^{\mathrm{sat}}_{\eta} \beta$	iff $\textsc{Eval}^{\eta}_{\phi_\tau}(\beta)$ is $\mathtt{T}$.
$\phi_\tau \models^{\mathrm{sat}}_{\eta} \neg\pi$	iff $\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi$ holds.
$\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_1 \wedge \pi_2$	iff both $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_1$ and $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_2$ holds.
$\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_1 \vee \pi_2$	iff both $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_1$ and $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_2$ holds, or both $\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi_1$ and $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_2$ holds, or both $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_1$ and $\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi_2$ holds.
$\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_1 \rightarrow \pi_2$	iff both $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_1$ and $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_2$ holds, or both $\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi_1$ and $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_2$ holds, or both $\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi_1$ and $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_2$ holds.
$\phi_\tau \models^{\mathrm{sat}}_{\eta} \exists(w:\tau)\cdot\pi$	iff $\phi_\tau[w \mapsto k] \models^{\mathrm{sat}}_{\eta} \pi$ holds for some $k \in \tau$, and forall all $k \in \tau$, either $\phi_\tau[w \mapsto k] \models^{\mathrm{sat}}_{\eta} \pi$ or $\phi_\tau[w \mapsto k] \models^{\mathrm{dst}}_{\eta} \pi$ holds.
$\phi_\tau \models^{\mathrm{sat}}_{\eta} \forall(w:\tau)\cdot\pi$	iff $\phi_\tau[w \mapsto k] \models^{\mathrm{sat}}_{\eta} \pi$ holds for all $k \in \tau$
$\phi_\tau \models^{\mathrm{dst}}_{\eta} \beta$	iff $\textsc{Eval}^{\eta}_{\phi_\tau}(\beta)$ is $\mathtt{F}$.
$\phi_\tau \models^{\mathrm{dst}}_{\eta} \neg\pi$	iff $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi$ holds.
$\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi_1 \wedge \pi_2$	iff both $\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi_1$ and $\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi_2$ holds, or both $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_1$ and $\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi_2$ holds, or both $\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi_1$ and $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_2$ holds.
$\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi_1 \vee \pi_2$	iff both $\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi_1$ and $\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi_2$ holds.
$\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi_1 \rightarrow \pi_2$	iff both $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi_1$ and $\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi_2$ holds.
$\phi_\tau \models^{\mathrm{dst}}_{\eta} \exists(w:\tau)\cdot\pi$	iff $\phi_\tau[w \mapsto k] \models^{\mathrm{dst}}_{\eta} \pi$ holds for all $k \in \tau$
$\phi_\tau \models^{\mathrm{dst}}_{\eta} \forall(w:\tau)\cdot\pi$	iff $\phi_\tau[w \mapsto k] \models^{\mathrm{dst}}_{\eta} \pi$ holds for some $k \in \tau$, and forall all $k \in \tau$, either $\phi_\tau[w \mapsto k] \models^{\mathrm{sat}}_{\eta} \pi$ or $\phi_\tau[w \mapsto k] \models^{\mathrm{dst}}_{\eta} \pi$ holds.
$\phi_\tau \models^{\mathrm{udt}}_{\eta} \pi$	iff neither $\phi_\tau \models^{\mathrm{sat}}_{\eta} \pi$ or $\phi_\tau \models^{\mathrm{dst}}_{\eta} \pi$ holds.

Fig. 8. Definition of satisfaction relation

4 Reasoning with Infinity

For the following discussion we assume the existence of a solver for Presburger arithmetic (such as Omega [9]). Our focus is to automate the reasoning of ghost infinities by leveraging on existing solvers. Note that $v \in \mathbb{Z}_\infty$, is the same as, $v \in \mathbb{Z} \vee v = \infty \vee v = -\infty$. This fact can be used to give a quantifier elimination procedure for PAInf as shown in figure 9. However, using this approach naively leads to an explosion in the size of formulae to be checked. As an example, consider the following formula,

$$\forall \mathtt{x}, \mathtt{y}, \mathtt{z} \cdot (\mathtt{z}{=}\infty \wedge \mathtt{y}{=}\mathtt{x} + \mathtt{z} \wedge \mathtt{x}{<}\infty)$$

Using the [FORALL−INF] rule to eliminate the three quantified variables (x, y and z), leads to 3^3 (= 27) constraints. To avoid this problem, we support both kinds of quantifiers ($\exists(w:\mathbb{Z})$ and $\exists(w:\mathbb{Z}_\infty)$) in the implementation. This allows for a more efficient quantifier elimination as variables with finite domain do not give rise to new disjunctions in formulae. Since, infinity is added as a ghost constant only in the specification logic, all program variables are still in finite domain. Supporting two kinds of quantifiers matches nicely with the distinction between the domain of specification variables ($\mathbb{Z}_\infty$)

and program variables ($\mathbb{Z}$). In section 6 we compare our system with an implementation of PAI from [13] and demonstrate the effectiveness of using our procedure.

$[\textbf{EXISTS}-\textbf{INF}]$	$[\textbf{FORALL}-\textbf{INF}]$
$\exists(w:\mathbb{Z}_\infty)\cdot\pi \leadsto \exists(w:\mathbb{Z})\cdot\pi$	$\forall(w:\mathbb{Z}_\infty)\cdot\pi \leadsto \forall(w:\mathbb{Z})\cdot\pi$
$\vee[\infty/w]\pi$	$\wedge[\infty/w]\pi$
$\vee[-\infty/w]\pi$	$\wedge[-\infty/w]\pi$

Fig. 9. PAInf: Quantifier Elimination (INF-TRANS)

For checking satisfiability in the PAInf we use the algorithm shown in figure 10. We denote the procedure for satisfiability checking as $SAT(\pi)$. The algorithm has four steps: (i) first we eliminate the quantifiers starting with the innermost quantifier, (ii) next we apply a normalization which detects tautologies and contradictions in constraints using infinity, (iii) then we eliminate min-max and constant constraints and (iv) finally we solve the resulting formula using an existing PA solver Omega.

	$SAT(\pi)$	$\pi_F = \text{INF-TRANS}(\pi)$	*(1) Quantifier Elimination*
$\Longrightarrow$	$SAT(\pi_F)$	$\pi_N = \text{INT-TRANS}(\pi_F)$	*(2) Normalization*
$\Longrightarrow$	$SAT(\pi_N)$	$\pi_G = \text{SIMP}(\pi_N)$	*(3) Simplification*
$\Longrightarrow$	$SAT(\pi_G)$		*(4) Omega*

Fig. 10. PAInf: SAT Checking

At a high level the intuition behind the SAT checking algorithm is as follows: after quantifier elimination, the π_F formula has quantifiers only on the finite domain variables. The normalization and simplification eliminate all the infinite constants from the formula. The resulting formula (π_G) is in PA and its satisfiability can be checked using Omega. Next we describe the steps in the SAT checking algorithm in detail.

$[\textbf{EVAL}-\textbf{FIN}]$	$[\textbf{EVAL}-\textbf{INF}]$	$[\textbf{EVAL}-\textbf{BOT}]$
$v \leadsto Z$	$\infty+\infty \leadsto \infty$	$\infty+(-\infty) \leadsto \bot$
$c \leadsto Z$	$-\infty+(-\infty) \leadsto -\infty$	$-\infty+\infty \leadsto \bot$
$-Z \leadsto Z$	$-\infty+Z \leadsto -\infty$	$\bot+Z \leadsto \bot$
$Z+Z \leadsto Z$	$Z+(-\infty) \leadsto -\infty$	$Z+\bot \leadsto \bot$
$Z-Z \leadsto Z$	$\infty+Z \leadsto \infty$	$\bot+\bot \leadsto \bot$
$c\times Z \leadsto Z$	$Z+\infty \leadsto \infty$	$-\bot \leadsto \bot$

Fig. 11. PAInf: Evaluation Check

4.1 Normalization and Simplification

We define a set of rewriting rules based on the semantics of formulae in PAInf. We work only with closed-form formulae, thus after applying the quantifier elimination given in figure 9, all the remaining variables are in the finite domain ($\mathbb{Z}$). It is possible to compare the variables with infinities by evaluating their values (as they are all finite) using the semantics given in the section 3. This is performed by the `Evaluation Check` function in figure 11 which reduces each expression to a finite value (denoted by Z). Thus, for the normalization rules in figure 12 we only need to consider the integer values (Z) and the infinity constants. Note that, the `Evaluation Check` is only applied for the purpose of checking the finiteness and eliminating infinity, the actual formula is not evaluated.

The normalization process uses the rewriting rules given in figure 12 (rules for $\neq$, $\geq$, $<$ and $\min_=$ are similar and omitted for brevity). These rules detect the tautologies and contradictions in the usage of ∞ and $-\infty$, and the constraints involving ∞ and $-\infty$ are eliminated. After the application of these rules the given formula is reduced to a form which can be solved by existing PA solvers like Omega.

[NORM−INF−EQ]	[NORM−INF−LEQ]	[NORM−INF−LT]
$\bot = \_ \leadsto \texttt{error}$	$\bot \leq \_ \leadsto \texttt{error}$	$\bot < \_ \leadsto \texttt{error}$
$\_ = \bot \leadsto \texttt{error}$	$\_ \leq \bot \leadsto \texttt{error}$	$\_ < \bot \leadsto \texttt{error}$
$Z = \infty \leadsto \texttt{false}$	$Z \leq \infty \leadsto \texttt{true}$	$Z < \infty \leadsto \texttt{true}$
$\infty = \infty \leadsto \texttt{true}$	$\infty \leq \infty \leadsto \texttt{true}$	$\infty < \infty \leadsto \texttt{false}$
$-\infty = \infty \leadsto \texttt{false}$	$-\infty \leq \infty \leadsto \texttt{true}$	$-\infty < \infty \leadsto \texttt{true}$
$-\infty = Z \leadsto \texttt{false}$	$-\infty \leq Z \leadsto \texttt{true}$	$-\infty < Z \leadsto \texttt{true}$
$-\infty = -\infty \leadsto \texttt{true}$	$-\infty \leq -\infty \leadsto \texttt{true}$	$-\infty < -\infty \leadsto \texttt{false}$
$\infty = Z \leadsto \texttt{false}$	$\infty \leq Z \leadsto \texttt{false}$	$\infty < Z \leadsto \texttt{false}$
$\infty = -\infty \leadsto \texttt{false}$	$\infty \leq -\infty \leadsto \texttt{false}$	$\infty < -\infty \leadsto \texttt{false}$
$Z = -\infty \leadsto \texttt{false}$	$Z \leq -\infty \leadsto \texttt{false}$	$Z < -\infty \leadsto \texttt{false}$

[NORM−EQ−MAX]

$\max_=(\infty,\infty,\infty) \leadsto \texttt{true}$	$\max_=(-\infty,Z,Z) \leadsto \texttt{false}$	$\max_=(\_,\_,\bot) \leadsto \texttt{error}$
$\max_=(-\infty,-\infty,-\infty) \leadsto \texttt{true}$	$\max_=(\infty,Z,-\infty) \leadsto \texttt{false}$	$\max_=(\infty,Z,Z) \leadsto \texttt{false}$
$\max_=(-\infty,Z,-\infty) \leadsto \texttt{false}$	$\max_=(\infty,-\infty,\infty) \leadsto \texttt{true}$	$\max_=(\infty,\infty,Z) \leadsto \texttt{true}$
$\max_=(-\infty,\infty,-\infty) \leadsto \texttt{false}$	$\max_=(-\infty,\infty,Z) \leadsto \texttt{false}$	$\max_=(Z,\infty,Z) \leadsto \texttt{false}$
$\max_=(\infty,-\infty,-\infty) \leadsto \texttt{false}$	$\max_=(\infty,-\infty,Z) \leadsto \texttt{false}$	$\max_=(\_,\bot,\_) \leadsto \texttt{error}$
$\max_=(-\infty,-\infty,Z) \leadsto \texttt{false}$	$\max_=(Z,\infty,-\infty) \leadsto \texttt{false}$	$\max_=(\infty,Z,\infty) \leadsto \texttt{true}$
$\max_=(\infty,\infty,-\infty) \leadsto \texttt{true}$	$\max_=(-\infty,Z,\infty) \leadsto \texttt{false}$	$\max_=(\bot,\_,\_) \leadsto \texttt{error}$
$\max_=(Z,-\infty,-\infty) \leadsto \texttt{false}$	$\max_=(Z,-\infty,\infty) \leadsto \texttt{false}$	$\max_=(Z,Z,\infty) \leadsto \texttt{false}$
$\max_=(-\infty,-\infty,\infty) \leadsto \texttt{false}$	$\max_=(-\infty,\infty,\infty) \leadsto \texttt{false}$	$\max_=(Z,\infty,\infty) \leadsto \texttt{false}$

[NORM−INF−ERR]

$\texttt{error} \leadsto \texttt{false}$	(two-valued logic)
$\texttt{error} \leadsto \texttt{undefined}$	(three-valued logic)

Fig. 12. PAInf: Normalization (INT-TRANS)

We also proved the following theorems and lemmas about quantifier elimination INF-TRANS and normalization INT-TRANS. These theorems and lemmas hold for both two-valued/three-valued logics and all choices of $(0 \times \infty)$. Hence, the Coq certified proof of these theorems and lemmas is also parameterized. Note that for quantifier elimination the universe of environment τ and the domain of constants η are both instantiated to $\mathbb{Z}_\infty$.

Lemma 5. Quantifier Elimination $\phi_{\mathbb{Z}_\infty} \models^{\mathrm{sat}}_{\mathbb{Z}_\infty} \pi$ *if and only if* $\phi_{\mathbb{Z}} \models^{\mathrm{sat}}_{\mathbb{Z}_\infty}$ INF-TRANS(π), $\phi_{\mathbb{Z}_\infty} \models^{\mathrm{dst}}_{\mathbb{Z}_\infty} \pi$ *if and only if* $\phi_{\mathbb{Z}} \models^{\mathrm{dst}}_{\mathbb{Z}_\infty}$ INF-TRANS(π),

For infinity elimination τ is $\mathbb{Z}_\infty$ and η is $\mathbb{Z}$. This is due to the fact that after quantifier elimination the domain of all the variables is finite.

Lemma 6. Infinity Elimination $\phi_{\mathbb{Z}} \models^{\text{sat}}_{\mathbb{Z}_\infty} \pi$ *if and only if* $\phi_{\mathbb{Z}} \models^{\text{sat}}_{\mathbb{Z}}$ $\textsc{Int-Trans}(\pi)$, $\phi_{\mathbb{Z}} \models^{\text{dst}}_{\mathbb{Z}_\infty} \pi$ *if and only if* $\phi_{\mathbb{Z}} \models^{\text{dst}}_{\mathbb{Z}}$ $\textsc{Int-Trans}(\pi)$.

[**ELIM**]

$$\max_{=}(a_1, a_2, a_3) \rightsquigarrow (a_1 = a_2 \wedge a_3 \leq a_2) \vee (a_1 = a_3 \wedge a_2 \leq a_3)$$
$$\min_{=}(a_1, a_2, a_3) \rightsquigarrow (a_1 = a_2 \wedge a_2 \leq a_3) \vee (a_1 = a_3 \wedge a_3 \leq a_2)$$

[**SIMP**]

$\beta \rightsquigarrow \textsc{Elim}(\beta)$	$\neg\texttt{undefined} \rightsquigarrow \texttt{undefined}$
$\texttt{undefined} \wedge \pi \rightsquigarrow \texttt{undefined}$	$\pi \wedge \texttt{undefined} \rightsquigarrow \texttt{undefined}$
$\texttt{true} \wedge \pi \rightsquigarrow \pi$	$\pi \wedge \texttt{true} \rightsquigarrow \pi$
$\texttt{false} \wedge \pi \rightsquigarrow \texttt{false}$	$\pi \wedge \texttt{false} \rightsquigarrow \texttt{false}$
$\texttt{undefined} \vee \pi \rightsquigarrow \texttt{undefined}$	$\pi \vee \texttt{undefined} \rightsquigarrow \texttt{undefined}$
$\texttt{true} \vee \pi \rightsquigarrow \texttt{true}$	$\pi \vee \texttt{true} \rightsquigarrow \texttt{true}$
$\texttt{false} \vee \pi \rightsquigarrow \pi$	$\pi \vee \texttt{false} \rightsquigarrow \pi$
$\texttt{undefined} \rightarrow \pi \rightsquigarrow \texttt{undefined}$	$\pi \rightarrow \texttt{undefined} \rightsquigarrow \texttt{undefined}$
$\texttt{false} \rightarrow \pi \rightsquigarrow \texttt{true}$	$\pi \rightarrow \texttt{true} \rightsquigarrow \texttt{true}$
$\texttt{true} \rightarrow \pi \rightsquigarrow \pi$	$\pi \rightarrow \texttt{false} \rightsquigarrow \neg\pi$
$\neg\texttt{true} \rightsquigarrow \texttt{false}$	$\neg\texttt{false} \rightsquigarrow \texttt{true}$
$\forall(w : \tau) \cdot \texttt{undefined} \rightsquigarrow \texttt{undefined}$	$\exists(w : \tau) \cdot \texttt{undefined} \rightsquigarrow \texttt{undefined}$
$\forall(w : \tau) \cdot \texttt{true} \rightsquigarrow \texttt{true}$	$\exists(w : \tau) \cdot \texttt{true} \rightsquigarrow \texttt{true}$
$\forall(w : \tau) \cdot \texttt{false} \rightsquigarrow \texttt{false}$	$\exists(w : \tau) \cdot \texttt{false} \rightsquigarrow \texttt{false}$

Fig. 13. Definition of Simplification

So for the total transformation $\textsc{Trans}(\pi) = \textsc{Int-Trans}(\textsc{Inf-Trans}(\pi))$ used in satisfiability checking, we have the following theorem:

Theorem 1. Satisfiability Checking $\phi_{\mathbb{Z}_\infty} \models^{\text{sat}}_{\mathbb{Z}_\infty} \pi$ *if and only if* $\phi_{\mathbb{Z}} \models^{\text{sat}}_{\mathbb{Z}}$ $\textsc{Trans}(\pi)$, $\phi_{\mathbb{Z}_\infty} \models^{\text{dst}}_{\mathbb{Z}_\infty} \pi$ *if and only if* $\phi_{\mathbb{Z}} \models^{\text{dst}}_{\mathbb{Z}}$ $\textsc{Trans}(\pi)$,

Gallina, the internal functional language of Coq is strongly normalizing. Thus, all functions written in Coq must terminate.

Theorem 2. Termination *Satisfiability checking in PAInf (figure 10) terminates.*

The quantifier elimination with infinity expands the logical formula π and the normalization introduces many logical constants. We introduce a simplification function $\textsc{Simp}$ which recursively eliminates logical constants according to the rules in figure 13 in order to reduce the length of a formula. As Omega doesn't support $\max_{=}$ or $\min_{=}$ we also include the elimination of $\max_{=}$ and $\min_{=}$ in $\textsc{Simp}$. Note that for three-valued logic, the logical constants contain a third value: `undefined` which is not supported by Omega. Our $\textsc{Simp}$ function propagates `undefined` to the whole formula such that we know if a formula is undetermined before calling Omega due of the following theorem:

Theorem 3. Decide Undetermined $\phi_{\mathbb{Z}} \models^{\text{udt}}_{\mathbb{Z}} \pi$ *if and only if* $\textsc{Simp}(\pi)$=`undefined`

Thus, we do not need to extend Omega to support `undefined`. $\textsc{Simp}$ also preserves the validity of formulae:

Theorem 4. Simplification $\phi_{\mathbb{Z}} \models^{\text{sat}}_{\mathbb{Z}} \pi$ *if and only if* $\phi_{\mathbb{Z}} \models^{\text{sat}}$ $\textsc{Simp}(\pi)$, $\phi_{\mathbb{Z}} \models^{\text{dst}}_{\mathbb{Z}} \pi$ *if and only if* $\phi_{\mathbb{Z}} \models^{\text{dst}}_{\mathbb{Z}}$ $\textsc{Simp}(\pi)$.

5 Implementation

We gain a number of benefits in exchange for implementing Omega++ in Coq. We get proof of termination for free since Gallina (the extractable pure functional language of Coq) is strongly normalizing. More importantly, we get full machine-checked formal correctness proofs for our source code with respect to a well defined semantics for Presburger arithmetic with infinity. Coq's extraction facility then transforms the Gallina program into OCaml (or Haskell or Scheme), which we then compile and run as normal.

The following table presents some statistics for our Coq development of Omega++. The first column shows the file name, while the second and third columns are the number of lines in the file taken by the program and its soundness proof, respectively. Our total development is a modest 3,988 lines and the ratio of proof to program is 2.35. The fourth column gives the time taken by Coq to verify the file (*i.e.*, proof/type checking), using a 2.6 GHz Intel Core i7 with 16 GB of DDR3 RAM.

Coq File	*Program*	*Proof*	*Time (s)*	*Description*
`Theory.v`	585	737	20.68	*Syntax and Semantics;* SIMP
`Transformation.v`	350	1,203	31.07	INF-TRANS, INT-TRANS
`Simplification.v`	0	856	338.96	*Tactics/lemmas for* SIMP
`Extraction.v`	257	0	1.27	*Module to extract OCaml code*
	1,192	2,796	391.98	*Total Coq*

Note that type checking times have very little to do with file length. For example `Transformation.v` has 1,553 lines (combined program and proof), but takes less than 32 seconds to verify. On the other hand, verifying the 44 lines of the SIMP procedure, whose code is contained in `Theory.v`, takes more than five minutes! We also used one engineering trick to boost the performance of the extracted code. The code uses strings to represent both variables and (arbitrary-sized) integers, but Coq's encoding of strings is less efficient than OCaml's. We therefore usually treat strings as an abstract type within Coq and manipulate them via an interface to OCaml's string functions, passed in using a functor.

We will next highlight the key optimizations we used to get good performance and discuss how the program affected the proof—and vice versa. In the implementation we directly handle all of the logical operators and min-max constraints of the constraint language (figure 5), even though the "obvious" strategy would be to desugar aggressively. Unfortunately, sugar-free formulae are actually quite a bit larger than their svelter sugared cousins, resulting in a significant performance penalty. Working with fully-sugared formulae has a significant impact on the proofs because we must handle more cases.

Similarly, we allow the input formulae to specify, for each quantifier, whether the domain of quantification is over $\mathbb{Z}$ or over $\mathbb{Z}_\infty$. Quantifier elimination is expensive, and our "user"—the HIP/SLEEK verification toolset—often knows when a variable must be finite: in particular, program variables must be finite, whereas specification variables need not be. Communicating this fact to Omega++ resulted in significant performance gains, but again increased the proof effort due to the necessity of handling more cases.

To enable min/max, reduce the length of the output, eliminate redundant clauses, and propagate the `undefined` value, we implemented some basic simplifications (figure 13). The SIMP procedure was easy to implement but very painful to verify due to the vast number of cases we need to consider. In the end we wrote some custom proof tactics in Ltac (Coq's proof tactic language) which crunched through the tedium.

The previous examples all trade one-time verification effort for a better-performing algorithm. On the other hand, sometimes the proof improves the program. Before we started on our Coq implementation, we did a OCaml prototype for the quantifier-free fragment of the problem. That prototype's version of normalization did additional case analysis. Due to our careful treatment of quantifier elimination we were able to prove that much of this case analysis was unnecessary in our Coq tool. Moreover, the Coq development identified a soundness bug in the OCaml prototype, which allowed the invalid transformation $x{\geq}y \leadsto x{+}1{>}y$, which is false when $x = y = \infty$.

Overall, Omega++ is far better than our previous OCaml prototype. Consider:

Tool	*Sound*	*Complete*	*Termination*	*Semantics*	*Verified*
OCaml Prototype	*No*	*No*	*Unclear*	*Unclear*	*No*
Omega++	*Yes*	*Yes*	*Guaranteed*	*Precise*	*in Coq*

Of course, our OCaml prototype is a bit of a straw man, but we have been quite convinced that the substantial effort that it took to write Omega++ in Coq was well-rewarded. Moreover, as we will soon see, Omega++ has comparable performance to our OCaml prototype, despite solving a trickier problem in a much more through way.

6 Experiments

To benchmark Omega++ we integrated it into the HIP/SLEEK verification toolset [5] and developed a suite of tests (mostly searching and sorting programs) whose specifications use ∞ in interesting ways. The source code for each of these programs can be investigated in detail and tested with Omega++ [29] on our web site. In all the experiments we selected three-valued logic in Omega++ and used $0 \cdot \infty \overset{\text{def}}{=} 0$ as these are the appropriate choices for program verification. We used a 3.20GHz Intel Core i7-960 processor with 16GB memory running Ubuntu Linux 10.04 for our benchmarks, the first set of which are detailed in the table below.

Benchmark	*LOC*	*Disjuncts* ($\mathbb{Z}$)	*Time* (Ω)	*Disjuncts* ($\mathbb{Z}_\infty$)	*Time* (Ω++)
Insertion Sort	30	4	0.14	2	0.15
Selection Sort	69	14	0.36	7	0.35
Binary Search Tree	105	12	0.43	6	0.35
Bubble Sort	110	12	0.29	9	0.50
Merge Sort	91	6	0.32	4	1.81
Priority Queue	207	16	0.84	10	2.73
Total Correctness	21	_	_	2	0.21
Sorting with Min and Max	79	_	_	7	1.82

The first column lists the test name and the second gives its lines of code. The third and fifth columns show that $\mathbb{Z}_\infty$ enables more readable and concise specifications. Specifically, the third column gives the number of disjunctions required to express the test's specifications using $\mathbb{Z}$, whereas the fifth column expresses the same properties using $\mathbb{Z}_\infty$. For each test in the first group (top six), $\mathbb{Z}_\infty$ requires fewer disjunctions.

We do need to be a bit careful: although the specifications are informally for the same property (*e.g.*, “sortedness”), typically the specifications in $\mathbb{Z}_\infty$ are formally stronger since the embedded quantification occurs over larger sets. Note that we do not claim that Omega++ eliminates the disjunctions from reasoning since the quantifiers over infinities hide the disjunctions inside them. However, using infinities provides a useful abstraction to express the same property as the given specification is more concise. The difference in formal strength is the fundamental reason why the times given in columns four and six differ. Column four gives the time (including all of HIP/SLEEK) using Omega, whereas column six gives the time using Omega++. For the first four examples Omega++ is comparable to Omega, but in the final two of the first group of tests we believe the difference in the domain of quantification results in a significantly harder theorem in $\mathbb{Z}_\infty$, and thus, a noticeably longer runtime.

Comparison with similar tools. Lasaruk and Sturm [13] also propose extending Presburger arithmetic with infinity. Their work differs from ours in several respects. First, they only add a single infinity value, thus dodging any thorny—but in our view, important—semantic issues involving $\infty - \infty$. More importantly, Lasaruk and Sturm describe an algorithm but do not provide an implementation. For benchmarking purposes, we implemented their algorithm and tested it using the constraints generated from our test suite. We also compared our previous OCaml prototype as shown below:

Benchmark	*Calls*	*Time (PAI)*	*Time (Proto)*	*Time (Ω++)*
Insertion Sort	100	4.58	0.78	0.39
Selection Sort	245	>600.00	0.62	0.78
Binary Search Tree	116	150.00	0.48	0.50
Bubble Sort	336	>600.00	1.25	1.34
Merge Sort	155	>600.00	1.05	1.92
Priority Queue	778	>600.00	FAIL	1.20
Total Correctness	120	>600.00	0.31	0.16
Sorting with Min and Max	376	>600.00	0.29	0.19
Entailment Examples	124	1.89	FAIL	1.42
Lemma Examples	35	1.88	1.27	1.65
Total (except PQ and EE)	$1,824$	$>3,862.14$	7.21	8.11

The second column gives the number of times the associated decision procedure was called for each test. The third column gives the times for Lasaruk and Sturm’s “PAI” algorithm; many of the tests timed out after 10 minutes. The fourth column gives the times for our OCaml prototype “Proto”; notice that for two of the tests Proto failed (completeness holes). The fifth column gives the times for Omega++.

It is obvious that PAI, at least when implemented directly as given by Lasaruk and Sturm [13], is uncompetitive. Thus, Omega++ is always faster than PAI. When comparing Proto to Omega++, recall that Proto is only trying to solve the simpler problem of quantifier-free formulae. Despite this, for many of our tests the tools perform similarly. For a few tests, some of Proto’s heuristics result in appreciably better times; we plan to study these tests in more detail in the future to try to improve Omega++. Overall, Omega++’s performance is competitive.

Inference. As described in section 2.5, quantifier elimination in Presburger arithmetic with infinity can help with invariant generation of octagonal constraints. The table below benchmarks using Omega++ for this analysis technique.

Method	*Pre*	*Post*	*Inferred*	*Time (Omega++)*
Create	$\mathtt{true}$	$\mathtt{ll}\langle\mathtt{res},\mathtt{m}\rangle$	$\mathtt{m}{=}\mathtt{n}$	0.13
Delete	$\mathtt{ll}\langle\mathtt{x},\mathtt{n}\rangle$	$\mathtt{ll}\langle\mathtt{res},\mathtt{m}\rangle$	$\mathtt{n}{-}1{\leq}\mathtt{m}$	0.17
Insert	$\mathtt{ll}\langle\mathtt{x},\mathtt{n}\rangle\wedge\mathtt{x}{\neq}\mathtt{null}$	$\mathtt{ll}\langle\mathtt{x},\mathtt{m}\rangle$	$\mathtt{n}{=}\mathtt{m}{-}1$	0.13
Copy	$\mathtt{ll}\langle\mathtt{x},\mathtt{n}\rangle * \mathtt{ll}\langle\mathtt{res},\mathtt{m}\rangle$	$\mathtt{ll}\langle\mathtt{x},\mathtt{m}\rangle$	$\mathtt{m}{=}\mathtt{n}$	0.16
Remove	$\mathtt{ll}\langle\mathtt{x},\mathtt{n}\rangle\wedge\mathtt{x}{\neq}\mathtt{null}$	$\mathtt{ll}\langle\mathtt{x},\mathtt{m}\rangle$	$\mathtt{n}{-}1{\leq}\mathtt{m}\wedge\mathtt{m}{\leq}\mathtt{n}$	0.19
Return	$\mathtt{ll}\langle\mathtt{x},\mathtt{n}\rangle$	$\mathtt{ll}\langle\mathtt{x},\mathtt{m}\rangle$	$\mathtt{m}{=}\mathtt{n}\wedge 0{\leq}\mathtt{m}$	0.07
Traverse	$\mathtt{ll}\langle\mathtt{x},\mathtt{n}\rangle$	$\mathtt{ll}\langle\mathtt{x},\mathtt{m}\rangle$	$\mathtt{m}{=}\mathtt{n}$	0.12
Get	$\mathtt{ll}\langle\mathtt{x},\mathtt{n}\rangle\wedge\mathtt{x}{\neq}\mathtt{null}$	$\mathtt{ll}\langle\mathtt{res},\mathtt{m}\rangle$	$\mathtt{m}{=}\mathtt{n}{-}2\wedge 2{\leq}\mathtt{n}$	0.11
Head	$\mathtt{ll}\langle\mathtt{x},\mathtt{n}\rangle{*}\mathtt{ll}\langle\mathtt{y},\mathtt{m}\rangle$	$\mathtt{ll}\langle\mathtt{res},\mathtt{n}{+}\mathtt{m}{-}1\rangle$	$1{=}\mathtt{min}(\mathtt{n},\mathtt{m})$	0.21

The first column gives the test name. The second and third columns give the user-provided spatial pre- and postconditions in separation logic. The fourth column gives the inferred pure specification, while the last column gives the time used by Omega++. The final test is noteworthy because the inferred invariant uses min/max constraints.

7 Related Work and Conclusion

Reynolds demonstrated that *ghost variables* [26] were useful for verifying sequential programs. Their importance is highlighted when proving program, object or loop invariants [18], refining between two transition systems [17] or when considering program's security aspects [16]. Our work enriches specifications by extending the domain of ghost values with the mathematical concepts of positive and negative infinity.

Presburger arithmetic [24] is one of the canonical examples of an important decidable problem. Kuncak *et al.* [11,12] presented a decision procedure for a quantifier-free fragment of Boolean Algebra with Presburger arithmetic which can be used to prove a mixed set-based constraint with symbolic cardinality and linear arithmetic. QFBAPA was later extended to the more challenging case of multisets [22] and proved to be NP-complete [23]. The VCDryad [21] framework combines separation logic with decision procedures for sets and multi sets to verify programs with natural proofs. The combination of set/multi-sets with separation logic even though quite useful requires complex provers that can reason over the domain of sets/muti-sets.

Lasaruk and Sturm [13] were the first to tackle the problem of extending PA with infinity, proving completeness and decidability. Our work differs from theirs as we allow two distinct values for positive and negative infinities and provide a implementation. Our decision procedure is built on top of Omega calculator [9], and certified in Coq [1]. The general problem of adding infinities to the set of reals was addressed by Weispfenning [27]. This was later extended to mixed real and integer quantifier elimination in [28]. Another interesting extension of decision procedures for real arithmetic is the addition of infinitesimals. The proof assistant Isabelle/HOL [20] has support for infinitesimals. Loos and Weispfenning [15] first proposed a virtual substitution approach for quantifier elimination of infinitesimals. We also use a similar virtual substitution to eliminate infinities as part of the decision procedure. Chaieb and Nipkow [4] present a reflective implementation of Cooper's algorithm for quantifier elimination in PA. Their work complements our approach as we reduce from PA extended with infinities to PA.

Conclusion. We presented Omega++, a decision procedure for Presburger arithmetic with infinity $\mathbb{Z}_\infty$. Infinity is a useful abstraction, increasing a program logic's ability to reason about termination and compose more elegantly. Moreover, specifications with infinity are often more concise. Omega++ has been Coq-certified to respect a precise formal semantics for $\mathbb{Z}_\infty$. We integrated Omega++ into an existing verifier and evaluated it on a benchmark of small programs, demonstrating that it can perform well in practice. Omega++ demonstrates that we can develop useful, efficient, and certified programs for program verification and analysis.

Acknowlegement. This work is supported by MoE Tier-1 NUS research project R-252-000-525-112 and Yale-NUS College R-607-265-045-121.

References

1. The Coq Proof Assistant. `http://coq.inria.fr/`.
2. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.
3. Merrie Bergmann. *An introduction to many-valued and fuzzy logic: semantics, algebras, and derivation systems*. Cambridge University Press, 2008.
4. Amine Chaieb and Tobias Nipkow. Verifying and reflecting quantifier elimination for presburger arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*, pages 367–380, 2005.
5. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.
6. S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *ACM POPL*, January 2001.
7. D. Kapur, Z. Zhang, M. Horbach, H. Zhao, Q. Lu, and T.V. Nguyen. Automated reasoning and mathematics. chapter Geometric Quantifier Elimination Heuristics for Automatically Generating Octagonal and Max-plus Invariants, pages 189–228. Springer-Verlag, 2013.
8. Deepak Kapur. Automatically generating loop invariants using quantifier elimination. In *In Deduction and Applications*, 2005.
9. P. Kelly, V. Maslov, and W. Pugh. *The Omega Library Version 1.1.0 Interface Guide*, 1996.
10. N.A. Kolmogorov. *"Infinity." Encyclopaedia of Mathematics: An Updated and Annotated Translation of the Soviet "Mathematical Encyclopaedia,"*, volume 3. Reidel, 1995.
11. V. Kuncak, H. H. Nguyen, and M. Rinard. An algorithm for deciding BAPA: Boolean algebra with Presburger arithmetic. In *CADE*, 2005.
12. Viktor Kuncak and Martin Rinard. Towards efficient satisfiability checking for boolean algebra with Presburger arithmetic. In *CADE*. 2007.
13. A. Lasaruk and T. Sturm. Effective quantifier elimination for Presburger arithmetic with infinity. In *CASC*. 2009.
14. Ton Chanh Le, Cristian Gherghina, Aquinas Hobor, and Wei-Ngan Chin. A Resource-Based Logic for Termination and Non-Termination Proofs. In *ICFEM*, 2014.

15. Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *Comput. J.*, 36(5):450–462, 1993.
16. Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in expressos. In *ASPLOS*, 2013.
17. M. Marcus and A. Pnueli. Using ghost variables to prove refinement. In *AMST*. 1996.
18. S. McPeak and G. Necula. Data structure specifications via local equality axioms. In *CAV*, 2005.
19. Edward James McShane. *Unified integration*, volume 107. Academic Press, 1983.
20. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
21. Edgar Pek, Xiaokang Qiu, and P Madhusudan. Natural proofs for data structure manipulation in c using separation logic. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 46. ACM, 2014.
22. Ruzica Piskac and Viktor Kuncak. Decision procedures for multisets with cardinality constraints. In *VMCAI*. 2008.
23. Ruzica Piskac and Viktor Kuncak. Linear arithmetic with stars. In *CAV*. 2008.
24. Mojzesz Presburger. *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchen die Addition als einzige Operation hervortritt*. 1929.
25. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, 2002.
26. John C. Reynolds. *The craft of programming*. Prentice Hall International series in computer science. Prentice Hall, 1981.
27. Volker Weispfenning. Quantifier elimination for real algebra - the quadratic case and beyond. *Appl. Algebra Eng. Commun. Comput.*, 8(2):85–101, 1997.
28. Volker Weispfenning. Mixed real-integer linear quantifier elimination. In *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation, ISSAC '99, Vancouver, B.C., Canada, July 29-31, 1999*, pages 129–136, 1999.
29. Omega++ with HIP/SLEEK. Source and binaries available at `http://loris-7.ddns.comp.nus.edu.sg/~project/SLPAInf/`. October 2014.