Program Verification via Craig Interpolation for Presburger Arithmetic with Arrays*

Angelo Brillout ETH Zurich, Switzerland
Daniel Kroening Philipp Rümmer Thomas Wahl Oxford University Computing Laboratory, United Kingdom

Abstract

Craig interpolation has become a versatile tool in formal verification, in particular for generating intermediate assertions in safety analysis and model checking. In this paper, we present a novel interpolation procedure for the theory of arrays, extending an interpolating calculus for the full theory of quantifier-free Presburger arithmetic, which will be presented at IJCAR this year. We investigate the use of this procedure in a software model checker for C programs. A distinguishing feature of the model checker is its ability to faithfully model machine arithmetic with an encoding into Presburger arithmetic with uninterpreted predicates. The interpolation procedure allows the synthesis of quantified invariants about arrays. This paper presents work in progress; we include initial experiments to demonstrate the potential of our method.

1 Introduction

Craig interpolation for first-order formulas [1] has emerged as a practical approximation method in computing and has found many uses in formal verification. Given two formulae A and C such that A implies C, written $A \Rightarrow C$, an interpolant is a formula I such that the implications $A \Rightarrow I$ and $I \Rightarrow C$ hold and I contains only non-logical symbols occurring in both A and C. Interpolants exist for any two first-order formulae A and C such that $A \Rightarrow C$. As common in formal verification, we consider unsatisfiable conjunctions $A \wedge B$, which corresponds to $C = \neg B$ in the above formulation.

In software verification, interpolation is applied to formulae encoding the transition relation of a model underlying the program. In order to support a wide variety of programming language constructs, much effort has been invested in the design of algorithms that compute interpolants for formulae of various theories. For example, interpolating integer arithmetic solvers have so far been reported for fragments such as difference-bound logic, linear equalities, and constant-divisibility predicates.

In a recent paper to be published at IJCAR [2],[1] we present the first efficient interpolation procedure for the full range of quantifier-free linear integer arithmetic, commonly known as *Presburger arithmetic*, QFPA. This logic is instrumental in modeling the behavior of infinite-state programs [3] and of hardware designs [4]. Our procedure extracts an interpolant directly from an unsatisfiability proof for $A \wedge B$. Starting from a sound and complete proof system for QFPA based on a sequent calculus, the proof rules are augmented with labeled formulae and *partial interpolants* — proof annotations that reduce, at the root of a closed proof, to interpolants. The interpolating proof system has been shown to be sound and complete for QFPA [2].

In this paper, we extend the calculus to QFPA with *uninterpreted predicates* (QFPAUP). The significance of this extension is two-fold. From a theoretical point of view, it turns out that an analogous completeness result as stated above for QFPA no longer holds: there are formulae in QFPAUP that cannot be interpolated without introducing quantifiers. Our solution to this problem is to add proof rules to the calculus that explicitly handle and introduce quantifiers in interpolants.

*This research is supported by the EPSRC project EP/G026254/1, by the EU FP7 STREP MOGENTES, and by the EU ARTEMIS CESAR project.

[1] `http://www.philipp.ruemmer.org/publications/iprincess10.pdf`

From a practical point of view, uninterpreted predicates allow us to apply our interpolating theorem prover to verification problems for programs with array-like data structures. To realize this end, we follow a layered approach. We first introduce uninterpreted functions to the logic, by a relational encoding using uninterpreted predicates and axioms expressing functional consistency. To formalize array operations, we then add two uninterpreted function symbols, *select* and *store*, as well as axioms that characterise the first-order theory of (non-extensional) arrays.

In order to illustrate how to use our interpolating solver in model checking, we provide an encoding of fixed-width bitvector arithmetic in QFPAUP. This encoding is implemented in the model checker Wolverine [5] to convert the (unfolded) transition relation of a C program into a QFPAUP formula.

In summary, we extend in this paper the interpolation procedure for QFPA presented in [2] by uninterpreted predicates, uninterpreted functions, and the theory of arrays. We demonstrate, using a set of initial experiments, how to use this interpolation procedure to support software model checking of C programs with arrays. To the best of our knowledge, ours is the first complete interpolation procedure for the theory of quantifier-free Presburger arithmetic with arrays.

Related Work

Interpolation in integer arithmetic. McMillan considers the logic of difference-bound constraints [6]. This logic, a fragment of QFPA, is decidable by reduction to rational arithmetic. As an extension, Cimatti et al. [7] present an interpolation procedure for the *UTVPI* fragment of linear integer arithmetic. Both fragments allow efficient reasoning and interpolation, but are not sufficient to express many typical program constructs, such as integer division. In [8], separate interpolation procedures for two theories are presented, namely (i) QFPA restricted to conjunctions of integer linear (dis)equalities and (ii) QFPA restricted to conjunctions of stride constraints. The combination of both fragments with integer linear inequalities is not supported, however. Our work closes this gap, as it permits predicates involving all types of constraints.

Kapur et al. [9] prove that QFPA is closed under interpolation (as an instance of a more general result about recursively enumerable theories), but their proof does not directly give rise to an efficient interpolation procedure.

Interpolation in the theory of arrays. McMillan provides a complete interpolation procedure for arrays [10] on top of a saturation prover for first-order logic by means of explicit array axioms. Our interpolation method resembles McMillan's in that explicit array axioms are given to a theorem prover, but our procedure is also complete in the combined theory of QFPA with arrays.

In [11], Jhala et al. define a "split prover" that can compute interpolants in the theories of difference bounds and a fragment of the theory of arrays, besides others. The main objective of [11] is to derive interpolants in restricted languages, which makes it possible to guarantee convergence and a certain form of completeness in model checking. While our procedure is more general in that the full combined theory of QFPA with arrays can be handled, we consider it as important future work to integrate techniques to restrict interpolant languages into our procedure.

Kapur et al. [9] present an interpolation method for arrays that works by reduction to the theory of uninterpreted functions. To some degree, our interpolation procedure can be considered as a lazy version of the procedure in [9]: while [9] uses the array axioms to compile away the *store*-function (and equality on arrays) upfront, producing a formula that can equivalently be proven in the theory of uninterpreted functions, our procedure only instantiates the array axioms on demand during the construction of a proof. It would be possible to combine our calculus for Presburger arithmetic with uninterpreted functions with the approach from [9]; an empirical comparison with our interpolation procedure for arrays would be interesting.

2 Model Checking by Lazy Abstraction with Interpolants

Our program verification method combines the work on *lazy interpolation-based model checking* in [6] with an interpolation procedure for QFPA and arrays. Generalising earlier work on hardware model checking, the approach from [6] proceeds by incrementally unwinding the control-flow graph of a program to a tree. Whenever a path from the program entry point to a program assertion (i.e., a safety specification) is found, a verification condition $\phi = T_1(s_0,s_1) \wedge T_2(s_1,s_2) \wedge \cdots \wedge T_n(s_{n-1},s_n) \wedge \neg C(s_n)$ is generated, where each s_i is a vector of variables representing the program state after execution of i statements on the path, $T_i(s_{i-1},s_i)$ is the transition relation corresponding to a statement, and $C(s_n)$ is the assertion to be verified.

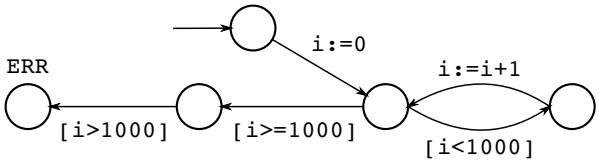
Under the assumption that the program to be verified is correct, formula ϕ is unsatisfiable. From a proof of unsatisfiability, it is then possible to generate a chain of $n+1$ interpolants $I_0(s_0), I_1(s_1), \ldots, I_n(s_n)$ with the properties:

$$I_0(s_0) = \mathit{true}, \qquad I_i(s_i) \wedge T_{i+1}(s_i,s_{i+1}) \Rightarrow I_{i+1}(s_{i+1}) \quad (\text{for } i \in \{0,\ldots,n-1\}), \qquad I_n(s_n) \Rightarrow C(s_n)$$

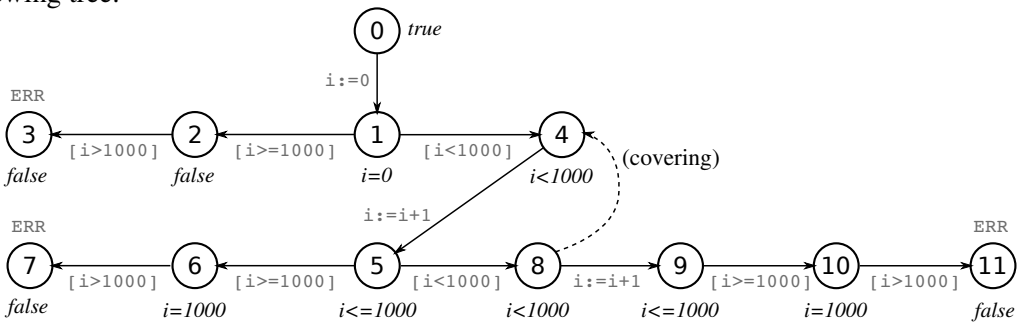
such that each $I_i(s_i)$ only talks about the program state s_i. In other words, each formula $I_i(s_i)$ represents an intermediate program assertion.

The interpolants I_i are used to label the nodes of the program unwinding tree, and are candidates for inductive invariants. To check whether the interpolants actually are inductive, the notion of a *covering relation* is introduced, which is a binary relation between nodes of the unwinding tree. We illustrate this concept using the example of a program counting from 0 to 1000; the diagram on the right shows the control-flow graph of the program, in which failing assertions are modelled via an explicit error state:

```
int i = 0;
while (i < 1000)
  i = i + 1;
assert(i <= 1000);
```



In order to prove that the error state is unreachable, the control-flow graph is partially unwound to the following tree:



The formulae written underneath or to the right of the vertexes are generated using interpolation when analysing the paths of the tree:

- $0 \to 1 \to 2 \to 3$: this path is infeasible, generating the interpolant chain *true*, $i \doteq 0$, *false*, *false*, which is used to label the nodes of the tree. The interpolants approximate the sets of states reachable in the various control-flow locations on the path.
- $0 \to 1 \to 4 \to 5 \to 6 \to 7$: similarly, this path is infeasible and generates the interpolants *true*, $i \doteq 0$, $i < 1000$, $i \leq 1000$, $i \doteq 1000$, *false*.

- $0 \to 1 \to 4 \to 5 \to 8 \to 9 \to 10 \to 11$: from this infeasible path we derive, in particular, the further annotation $i < 1000$ for node 8. It can now be observed that node 8 is *covered* by node 4: both refer to the same control-flow location, and the annotation $i < 1000$ of node 8 implies the annotation of node 4 (resembling an inductive loop invariant).

The covering of node 8 closes the proof and shows that the program is correct [6].

For our experiments, we use a development version of the model checker Wolverine [5], which implements the lazy abstraction approach to model checking. Our interpolation procedure is implemented in the tool iPrincess [2] and available for download.[2]

3 Background: Interpolation in Presburger Arithmetic

3.1 Preliminaries

Presburger arithmetic. We assume familiarity with classical first-order logic (e.g., [12]). Let x range over an infinite set X of variables, c over an infinite set C of constant symbols, p over a set P of uninterpreted predicates with fixed arity, f over a set F of uninterpreted functions with fixed arity, and α over the set $\mathbb{Z}$ of integers. The syntax of terms and formulae considered in this paper is defined by the following grammar:

$$\begin{aligned} \phi &::= t \doteq 0 \mid t \leq 0 \mid \alpha \mid t \mid p(t,\ldots,t) \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \forall x.\phi \mid \exists x.\phi \\ t &::= \alpha \mid c \mid x \mid \alpha t + \cdots + \alpha t \mid f(t,\ldots,t) \end{aligned}$$

The symbol t denotes terms of linear arithmetic. For simplicity, we only allow 0 as the right-hand side of equalities and inequalities.

Divisibility atoms $\alpha \mid t$ are equivalent to formulae $\exists s.\ \alpha s - t \doteq 0$, but are required for quantifier elimination and (quantifier-free) interpolation. Further, we use the abbreviations *true* and *false* for the equalities $0 \doteq 0$ and $1 \doteq 0$, and $\phi \to \psi$ to abbreviate $\neg\phi \vee \psi$. Simultaneous substitution of terms $t_1,\ldots,t_n$ for variables $x_1,\ldots,x_n$ in ϕ is denoted by $[x_1/t_1,\ldots,x_n/t_n]\phi$; we assume that variable capture is avoided by renaming bound variables as necessary.

Presburger arithmetic (PA) consists of those terms and formulae that do not contain uninterpreted predicates or functions. *Quantifier-free Presburger arithmetic (QFPA)* consists of those terms and formulae of PA that do not contain quantifiers. Until Sect. 4, we focus on the fragment QFPA. The semantics of Presburger arithmetic is defined over the universe $\mathbb{Z}$ of integers in the standard way [12].

Gentzen-style sequent calculi. If Γ, Δ are finite sets of formulae and C is a formula, all without free variables, then $\Gamma \vdash \Delta$ is a *sequent*. The sequent is *valid* if the formula $\bigwedge\Gamma \to \bigvee\Delta$ is valid. A calculus *rule* is a binary relation between a finite set of sequents called the premises, and a sequent called the conclusion. A sequent calculus rule is *sound* if, for all instances

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

whose premises $\Gamma_1 \vdash \Delta_1$, …, $\Gamma_n \vdash \Delta_n$ are valid, the conclusion $\Gamma \vdash \Delta$ is valid, too. Proof trees are defined to grow upwards. Each node is labeled with a sequent, and each non-leaf node is related to the node(s) directly above it through an instance of a calculus rule. A proof is *closed* if it is finite and all leaves are justified by an instance of a rule without premises.

[2] `http://www.philipp.ruemmer.org/iprincess.shtml`

3.2 An Interpolating Sequent Calculus for QFPA

This section briefly presents the interpolating sequent calculus for quantifier-free Presburger arithmetic introduced in [2]. For sake of brevity, most of the rules related to arithmetic reasoning (which are orthogonal to the extensions defined in Sect. 4 and 5) are left out in this paper; we refer the reader to [2] for more details.

Interpolating sequents. To extract interpolants from proofs of unsatisfiable conjunctions $A \wedge B$, Gentzen-style sequents (Sect. 3.1) are extended to *interpolating sequents*. Formulae in interpolating sequents are labeled either with the letter L ("left") to indicate that they are derived purely from A or with R ("right") for formulae derived purely from B. The labels L/R are sufficient to handle analytic rules that operate only on subformulae of the input formulae (similar to [12]). More formally, if ϕ is a formula without free variables, then $\lfloor \phi \rfloor_L$ and $\lfloor \phi \rfloor_R$ are L/R-labeled formulae. Furthermore, if Γ, Δ are sets of labeled formulae and I is an unlabeled formula such that (i) none of the formulae contains free variables, (ii) Γ only contains formulae $\lfloor \phi \rfloor_L$ or $\lfloor \phi \rfloor_R$, and (iii) Δ only contains formulae $\lfloor \phi \rfloor_L$ or $\lfloor \phi \rfloor_R$, then $\Gamma \vdash \Delta \blacktriangleright I$ is an *interpolating sequent*.

The semantics of interpolating sequents is defined using projections $\Gamma_L =_{\text{def}} \{\phi \mid \lfloor \phi \rfloor_L \in \Gamma\}$ and $\Gamma_R =_{\text{def}} \{\phi \mid \lfloor \phi \rfloor_R \in \Gamma\}$ that extract the L/R-parts of a set Γ of labeled formulae. A sequent $\Gamma \vdash \Delta \blacktriangleright I$ is *valid* if (i) the (Gentzen-style) sequent $\Gamma_L \vdash I, \Delta_L$ is valid, (ii) the sequent $\Gamma_R, I \vdash \Delta_R$ is valid, and (iii) the constants and uninterpreted predicates in I occur in both $\Gamma_L \cup \Delta_L$ and $\Gamma_R \cup \Delta_R$. As special cases, $\lfloor A \rfloor_L \vdash \lfloor C \rfloor_R \blacktriangleright I$ reduces to I being an interpolant of the implication $A \Rightarrow C$, while $\lfloor A \rfloor_L, \lfloor B \rfloor_R \vdash \blacktriangleright I$ captures the concept of interpolants for conjunctions $A \wedge B$ common in formal verification.

Note that rewriting rules for arithmetic may mix parts of A and B. This requires the additional notion of *partial interpolants*. We refer to [2] for further details on partial interpolants and on the rules handling Presburger arithmetic.

Interpolating rules. The propositional rules of the calculus are presented in Fig. 1. As usual in sequent calculi, the rules are applied in upward direction, starting from a sequent $\Gamma \vdash \Delta \blacktriangleright ?$ with unknown interpolant that is supposed to be proven (the proof root) and applying rules to successively decompose and simplify the sequent until a closure rule becomes applicable. The unknown interpolants of sequents have to be left open while building a proof and can only be filled in once all proof branches are closed.

To construct a proof for an interpolation problem $A \wedge B$, we start with the sequent $\lfloor A \rfloor_L, \lfloor B \rfloor_R \vdash \blacktriangleright ?$ that only contains L/R-labeled formulae and apply propositional rules to decompose A and B. Similarly, to construct a proof for an interpolation problem $A \Rightarrow C$, we start with the sequent $\lfloor A \rfloor_L \vdash \lfloor C \rfloor_R \blacktriangleright ?$, as is done in the example below. Once the decomposition of formulae results in arithmetic literals, arithmetic rules from [2] can be applied, possibly leading to a closed branch and an interpolant. Alternatively, it might be possible to close proof branches using the propositional closure rule CLOSE.

Example 1. *We illustrate the interpolating calculus at the propositional level by deriving an interpolant for* $A \Rightarrow C$, *with* $A = (c \neq 0 \vee d \doteq 0) \wedge c \doteq 0$ *and* $C = d \doteq 0$. *An interpolating proof of this implication* ***after*** *the interpolants have been filled in looks as follows:*

$$\dfrac{\dfrac{\dfrac{*}{\lfloor c \doteq 0 \rfloor_L \vdash \lfloor d \doteq 0 \rfloor_R, \lfloor c \doteq 0 \rfloor_L \blacktriangleright \mathit{false}}\,\text{CLOSE}}{\lfloor c \neq 0 \rfloor_L, \lfloor c \doteq 0 \rfloor_L \vdash \lfloor d \doteq 0 \rfloor_R \blacktriangleright \mathit{false}}\,\text{NOT-LEFT} \qquad \dfrac{*}{\lfloor d \doteq 0 \rfloor_L, \lfloor c \doteq 0 \rfloor_L \vdash \lfloor d \doteq 0 \rfloor_R \blacktriangleright d \doteq 0}\,\text{CLOSE}}{\lfloor c \neq 0 \vee d \doteq 0 \rfloor_L, \lfloor c \doteq 0 \rfloor_L \vdash \lfloor d \doteq 0 \rfloor_R \blacktriangleright \mathit{false} \vee d \doteq 0}\,\text{OR-LEFT-L}}{\lfloor (c \neq 0 \vee d \doteq 0) \wedge c \doteq 0 \rfloor_L \vdash \lfloor d \doteq 0 \rfloor_R \blacktriangleright \mathit{false} \vee d \doteq 0}\,\text{AND-LEFT}$$

The shaded regions of the proof indicate the parts of the formula that are matched against the rules in

$$\frac{\Gamma, \lfloor\phi\rfloor_L \vdash \Delta \blacktriangleright I \quad \Gamma, \lfloor\psi\rfloor_L \vdash \Delta \blacktriangleright J}{\Gamma, \lfloor\phi \vee \psi\rfloor_L \vdash \Delta \blacktriangleright I \vee J}\ \text{OR-LEFT-L} \qquad \frac{\Gamma, \lfloor\phi\rfloor_R \vdash \Delta \blacktriangleright I \quad \Gamma, \lfloor\psi\rfloor_R \vdash \Delta \blacktriangleright J}{\Gamma, \lfloor\phi \vee \psi\rfloor_R \vdash \Delta \blacktriangleright I \wedge J}\ \text{OR-LEFT-R}$$

$$\frac{\Gamma \vdash \lfloor\phi\rfloor_L, \Delta \blacktriangleright I \quad \Gamma \vdash \lfloor\psi\rfloor_L, \Delta \blacktriangleright J}{\Gamma \vdash \lfloor\phi \wedge \psi\rfloor_L, \Delta \blacktriangleright I \vee J}\ \text{AND-RIGHT-L} \qquad \frac{\Gamma \vdash \lfloor\phi\rfloor_R, \Delta \blacktriangleright I \quad \Gamma \vdash \lfloor\psi\rfloor_R, \Delta \blacktriangleright J}{\Gamma \vdash \lfloor\phi \wedge \psi\rfloor_R, \Delta \blacktriangleright I \wedge J}\ \text{AND-RIGHT-R}$$

$$\frac{\Gamma, \lfloor\phi\rfloor_D, \lfloor\psi\rfloor_D \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\phi \wedge \psi\rfloor_D \vdash \Delta \blacktriangleright I}\ \text{AND-LEFT} \qquad \frac{\Gamma \vdash \lfloor\phi\rfloor_D, \lfloor\psi\rfloor_D, \Delta \blacktriangleright I}{\Gamma \vdash \lfloor\phi \vee \psi\rfloor_D, \Delta \blacktriangleright I}\ \text{OR-RIGHT}$$

$$\frac{\Gamma \vdash \lfloor\phi\rfloor_D, \Delta \blacktriangleright I}{\Gamma, \lfloor\neg\phi\rfloor_D \vdash \Delta \blacktriangleright I}\ \text{NOT-LEFT} \qquad \frac{\Gamma, \lfloor\phi\rfloor_D \vdash \Delta \blacktriangleright I}{\Gamma \vdash \lfloor\neg\phi\rfloor_D, \Delta \blacktriangleright I}\ \text{NOT-RIGHT}$$

$$\frac{*}{\Gamma, \lfloor\phi\rfloor_L \vdash \lfloor\phi\rfloor_L, \Delta \blacktriangleright \mathit{false}}\ \text{CLOSE} \qquad \frac{*}{\Gamma, \lfloor\phi\rfloor_R \vdash \lfloor\phi\rfloor_R, \Delta \blacktriangleright \mathit{true}}\ \text{CLOSE}$$

$$\frac{*}{\Gamma, \lfloor\phi\rfloor_L \vdash \lfloor\phi\rfloor_R, \Delta \blacktriangleright \phi}\ \text{CLOSE} \qquad \frac{*}{\Gamma, \lfloor\phi\rfloor_R \vdash \lfloor\phi\rfloor_L, \Delta \blacktriangleright \neg\phi}\ \text{CLOSE}$$

Figure 1: The interpolating rules for propositional logic. Parameter D in the rules AND-LEFT, OR-RIGHT, and NOT-* stands for either L or R.

$$\frac{\Gamma, \lfloor[x/t]\phi\rfloor_L, \lfloor\forall x.\phi\rfloor_L \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\forall x.\phi\rfloor_L \vdash \Delta \blacktriangleright \forall_{Rt}\, I}\ \text{ALL-LEFT-L} \qquad \frac{\Gamma, \lfloor[x/t]\phi\rfloor_R, \lfloor\forall x.\phi\rfloor_R \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\forall x.\phi\rfloor_R \vdash \Delta \blacktriangleright \exists_{Lt}\, I}\ \text{ALL-LEFT-R}$$

$$\frac{\Gamma, \lfloor[x/c]\phi\rfloor_D \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\exists x.\phi\rfloor_D \vdash \Delta \blacktriangleright I}\ \text{EX-LEFT} \qquad \frac{\Gamma \vdash \lfloor[x/c]\phi\rfloor_D, \Delta \blacktriangleright I}{\Gamma \vdash \lfloor\forall x.\phi\rfloor_D, \Delta \blacktriangleright I}\ \text{ALL-RIGHT}$$

Figure 2: Interpolating rules to handle quantifiers. In ALL-LEFT-L, the quantifier $\forall_{Rt}$ denotes universal quantification over all constants occurring in t but not in $\Gamma_L \cup \Delta_L$; likewise, $\exists_{Lt}$ in ALL-LEFT-R denotes existential quantification over all constants occurring in t but not in $\Gamma_R \cup \Delta_R$. Parameter D stands for either L or R.

Fig. 1. The proof starts with the sequent $\lfloor(c \not\doteq 0 \vee d \doteq 0) \wedge c \doteq 0\rfloor_L \vdash \lfloor d \doteq 0\rfloor_R \blacktriangleright ?$; the ? acts as a place holder for the interpolant, to be filled in during the second phase of the proof.

The AND-LEFT *rule allows us to split the L-labeled conjunction $(c \not\doteq 0 \vee d \doteq 0) \wedge c \doteq 0$, retaining the L label in both constituents; the interpolant ? is unchanged. We can now apply* OR-LEFT-L *to the disjunction $c \not\doteq 0 \vee d \doteq 0$. This rule requires, however, that the interpolant in the conclusion also be a disjunction. We therefore replace ? by $I \vee J$, with fresh symbols I and J, before applying* OR-LEFT-L.

Since OR-LEFT-L *has two premises, the proof splits into two branches. The right branch, with interpolant J, can immediately be closed using the* CLOSE *rule in the lower-left corner of Fig. 1. At this point, the place holder J is instantiated by $d \doteq 0$, as dictated by* CLOSE. *The left branch, with interpolant I, requires an application of* NOT-LEFT *(which does not change the interpolant). Finally, closing this branch using the first of the four* CLOSE *rules instantiates I to false.*

Propagating the values found for I and J down to the root of the proof yields $I \vee J = \mathit{false} \vee d \doteq 0$, which is equivalent to $d \doteq 0$, as the final interpolant.

4 Interpolating QFPA with Uninterpreted Predicates

In this section, we consider interpolation for Presburger arithmetic with uninterpreted predicates. We use uninterpreted predicates later in Sect. 5 to enrich our arithmetic fragment by uninterpreted functions (via relational encodings of such functions). Uninterpreted functions in turn allow us to represent arrays and their operations in our logic, which is the main objective of this paper.

The logic of quantified Presburger arithmetic with predicates is Π^1_1-complete, which means that no complete calculi exist [13]. We therefore instead give a complete interpolating calculus for the *quantifier-free* fragment of Presburger arithmetic with predicates, QFPAUP.

We first observe that one cannot always avoid quantifiers in interpolants for QFPAUP:

Theorem 2. *QFPAUP is not closed under interpolation.*

In other words, there are unsatisfiable conjunctions $A \wedge B$ in QFPAUP for which no interpolants expressible in QFPAUP exists. In order to prove this theorem, we need an intermediate result.

Lemma 3. *Let y be a constant and $S = \{\alpha_i y + \beta_i \mid \alpha_i, \beta_i \in \mathbb{Z}, i \in \{1,\ldots,n\}\}$ be a finite set of terms in QFPA. Then there exists an even number $a \in 2\mathbb{Z}$ such that $\frac{a}{2} \notin \{val_{y \mapsto a}(t) \mid t \in S\}$.*

Proof. Choose $a \in 2\mathbb{Z}$ such that $a > 2 \cdot \max_i |\beta_i|$. Let us suppose that, for some $t = \alpha y + \beta \in S$, we have $val_{y\mapsto a}(\alpha y + \beta) = \alpha a + \beta = \frac{a}{2}$. This implies $2\alpha a + 2\beta = a$ and thus $(2\alpha - 1)a = -2\beta$. Since $2\alpha - 1 \neq 0$, we distinguish two cases:

- $2\alpha - 1 > 0$: this yields a contradiction because $(2\alpha-1)a \geq a > 2 \cdot |\beta| = |-2\beta| \geq -2\beta$.
- $2\alpha - 1 < 0$: this yields a contradiction because $(2\alpha-1)a \leq -a < -2 \cdot |\beta| = -|2\beta| \leq -2\beta$. □

We can now prove Theorem 2.

Proof of Theorem 2. We construct an example of inconsistent formulae A and B in QFPAUP whose interpolant requires quantification. Consider:

$$A = \ 2c - y \doteq 0 \wedge p(c) \qquad\qquad B = \ 2d - y \doteq 0 \wedge \neg p(d)$$

The symbols p and y are common, while c and d are local. The conjunction $A \wedge B$ is unsatisfiable. The strongest and the weakest interpolants for A and B are, respectively:

$$I_s = \ \exists x.\, (2x - y \doteq 0 \wedge p(x)) \qquad\qquad I_w = \ \forall x.\, (2x - y \doteq 0 \rightarrow p(x))$$

Now suppose I is a quantifier-free interpolant for $A \wedge B$; in particular, I contains only the common symbols p and y. Let $S = \{t \mid p(t) \text{ occurs in } I\}$ be the set of all terms occurring in I as arguments of p. All elements of S are QFPA terms over the symbol y. By Lem. 3, there is an even number $a \in 2\mathbb{Z}$ such that $\frac{a}{2} \notin \{val_{y\mapsto a}(t) \mid t \in S\}$.

Since I is an interpolant, the implications $I_s \Rightarrow I$ and $I \Rightarrow I_w$ hold. In particular, observe that

$$(2 \mid y) \models (I_s \leftrightarrow I) \wedge (I \leftrightarrow I_w)\,. \tag{1}$$

Choose an interpretation K with $K(y) = a$ that satisfies I (this is possible, because such satisfying interpretations exist for I_s). Because of (1) and because $K(y)$ is even, it holds that $\frac{K(y)}{2} \in K(p)$. However, we know that I does not contain any atom $p(t)$ such that $val_K(t) = \frac{K(y)}{2}$. This means that I is also satisfied by the interpretation K' that coincides with K, with the only exception that $\frac{K'(y)}{2} \notin K'(p)$. But K' violates I_w, contradicting the assumption that I is an interpolant. □

Proof rules for uninterpreted predicates. In the non-interpolating calculus [14], predicates are integrated using the following *unification* rule:

$$\frac{\Gamma, p(s_1,\ldots,s_n) \vdash p(t_1,\ldots,t_n), \bigwedge_i s_i - t_i \doteq 0, \Delta}{\Gamma, p(s_1,\ldots,s_n) \vdash p(t_1,\ldots,t_n), \Delta}\ \text{PRED-UNIFY}'$$

We simulate this rule by means of an explicit *predicate consistency* axiom:

$$PC_p = \quad \forall \bar{x}, \bar{y}.\ (p(\bar{x}) \wedge \bar{x} - \bar{y} \doteq 0) \rightarrow p(\bar{y}) \tag{2}$$

Axiom (2) can be viewed as an *L*- or *R*-labeled formula (depending on whether p occurs in $\Gamma_L \cup \Delta_L$, $\Gamma_R \cup \Delta_R$, or both) that is implicitly present in any sequent.

To make use of (2) in a proof, we need additional proof rules to instantiate quantifiers, which are given in Fig. 2. The rules ALL-LEFT-L/R are the potential source of quantifiers that can occur in QF-PAUP interpolants, because they can be used to instantiate *L*/*R*-labelled quantified formulae with terms containing alien symbols (which have to be eliminated from resulting interpolants through quantifiers).

Practically, formula (2) can be instantiated with techniques similar to the e-matching used in SMT solvers [15]: it is sufficient to generate a ground instance of (2) by applying ALL-LEFT-L/R whenever literals $p(\bar{s})$ and $p(\bar{t})$ occur in the antecedent and succedent, respectively, of a sequent [14]:

$$\frac{\Gamma, \lfloor p(\bar{s})\rfloor_D, \lfloor (p(\bar{s}) \wedge \bar{s} - \bar{t} \doteq 0) \rightarrow p(\bar{t})\rfloor_L \vdash \lfloor p(\bar{t})\rfloor_E, \Delta \blacktriangleright I}{\Gamma, \lfloor p(\bar{s})\rfloor_D \vdash \lfloor p(\bar{t})\rfloor_E, \Delta \blacktriangleright \forall_{R\bar{s}\bar{t}}\, I}\ \text{ALL-LEFT-L}^+$$

where $D, E \in \{L, R\}$ are arbitrary labels, and $\forall_{R\bar{s}\bar{t}}$ denotes universal quantification over all constants occurring in the terms $\bar{s}, \bar{t}$ but not in the set of left formulae $\left(\Gamma, \lfloor p(\bar{s})\rfloor_D\right)_L \cup \left(\Delta, \lfloor p(\bar{t})\rfloor_E\right)_L$ (like in Fig. 2). Similarly, instances of (2) labelled with R can be generated using ALL-LEFT-R. To improve efficiency, refinements can be formulated that drastically reduce the number of generated instances [16].

Example 4. *Fig. 3 shows how to derive an interpolant for* $\big(2c - y \doteq 0 \wedge p(c)\big) \wedge \big(2d - y \doteq 0 \wedge \neg p(d)\big)$*, a formula known from the proof of Theorem 2. The first steps in the proof are concerned with the decomposition of the input formulae to literals. Next, we instantiate* PC_p *with the predicate arguments* c *and* d*, due to the occurrences of the literals* $p(c)$ *and* $p(d)$ *in the sequent. After this, the proof can be closed by means of propositional rules, complementary literals, and arithmetic reasoning (the grey part, which uses rules and notation from [2]). The final interpolant is the formula* $I = \forall x.\ (y - 2x \neq 0 \vee p(x))$*, in which a quantifier has been introduced* ALL-LEFT-L *for the constant* d*.*

Soundness and completeness. The calculus consisting of the rules in Fig. 1 and 2 and the arithmetic rules of [2] generates correct interpolants. That is, whenever a sequent $\lfloor A\rfloor_L \vdash \lfloor C\rfloor_R \blacktriangleright I$ is derived, the implications $A \Rightarrow I$ and $I \Rightarrow C$ are valid, and all constants in I occur in both A and C. More precisely:

Lemma 5 (Soundness)**.** *If an interpolating sequent* $\Gamma \vdash \Delta \blacktriangleright I$ *is provable in the calculus, then it is valid. This implies, in particular, that the sequent* $\Gamma_L, \Gamma_R \vdash \Delta_L, \Delta_R$ *is valid.*

Vice versa, whenever an implication $A \Rightarrow C$ holds, the calculus permits the derivation of an interpolant:

Lemma 6 (Completeness)**.** *Suppose* Γ, Δ *are sets of labeled formulae* $\lfloor\phi\rfloor_L$ *and* $\lfloor\phi\rfloor_R$ *such that all occurrences of existential quantifiers in* Γ/Δ *are under an even/odd number of negations, and all occurrences of universal quantifiers in* Γ/Δ *are under an odd/even number of negations. If* $\Gamma_L, \Gamma_R \vdash \Delta_L, \Delta_R$ *is valid, then there is a formula* I *such that* $\Gamma \vdash \Delta \blacktriangleright I$ *is provable.*

As shown in [2], these two lemmas hold for the calculus consisting solely of the arithmetic and propositional rules. It is easy to see that the additional rules presented in this paper are sound and ensure completeness also in the presence of uninterpreted predicates.

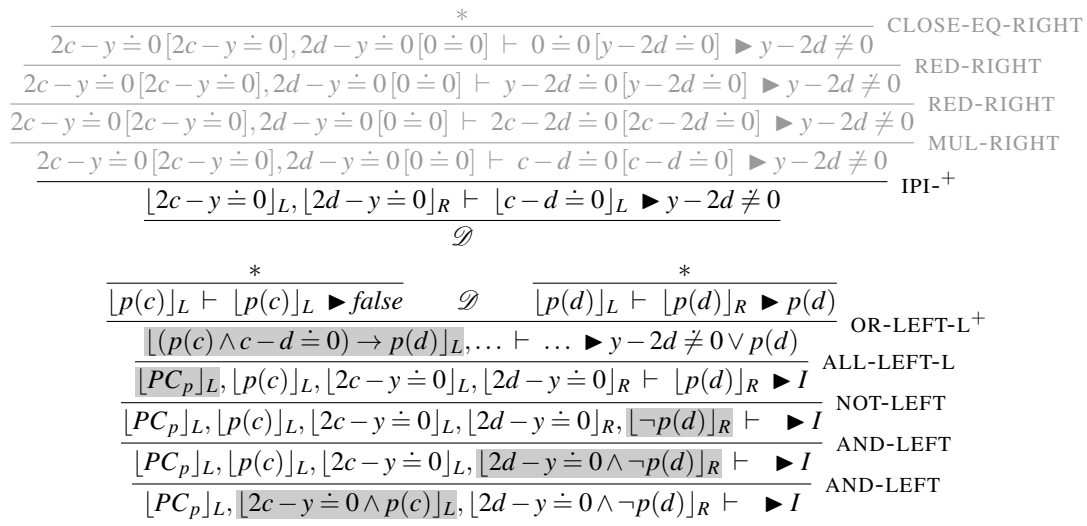


Figure 3: Example proof involving uninterpreted predicates.

Chain interpolation. As mentioned in Sect. 2, our calculus is used to generate interpolant *chains*. More precisely, given a path $\Gamma = T_1 \wedge \cdots \wedge T_n$, an interpolant chain is a sequence of interpolants such that (i) $I_0 = true$, $I_n = false$, (ii) $I_i \wedge T_{i+1} \Rightarrow I_{i+1}$ for all $0 \leq i \leq n-1$ and (iii) the constants and uninterpreted predicates in I_i occur in both Γ_L and Γ_R . The goal is to derive such chains by reusing a single proof of the sequent $T_1 \wedge \cdots \wedge T_n \vdash false$. To this end, we need to make sure that changing the label in a root $\Gamma \vdash \Delta$ of a proof does not affect the structure of the proof but only its labels and intermediate interpolants. This way, computing an interpolant for two *L*/*R* partitions of $T_1 \wedge \cdots \wedge T_n \vdash false$, for example, only requires the labels and the intermediate interpolants to be adjusted. We say that two proofs are *structurally equivalent* if they are identical up to their labels and interpolants.

Interpolation chains can be generated by considering a sequence of structurally equivalent proofs (we state this as a conjecture, because we have not yet proven it for all arithmetic rules). To formalise this, we denote by $\lfloor \Pi \rfloor_D$ the set of formulae $\{\lfloor f \rfloor_D : f \in \Pi\}$, for a set of formulae Π and $D \in \{L, R\}$.

Conjecture 7. *Consider two sets Γ' and Δ' of (unlabelled) formulae. Given a proof of the sequent $\Gamma, \lfloor \Gamma' \rfloor_R \vdash \lfloor \Delta' \rfloor_R, \Delta \blacktriangleright I$, there is a structurally equivalent proof of $\Gamma, \lfloor \Gamma' \rfloor_L \vdash \lfloor \Delta' \rfloor_L, \Delta \blacktriangleright J$ such that $\Gamma', I \vdash \Delta', J$ is valid.*

In the special case of a proof of $\lfloor T_1 \rfloor_L, \ldots \lfloor T_i \rfloor_L, \lfloor T_{i+1} \rfloor_R, \lfloor T_{i+2} \rfloor_R, \ldots, \lfloor T_n \rfloor_R \vdash false \blacktriangleright I_i$, there is an equivalent proof of $\lfloor T_1 \rfloor_L, \ldots \lfloor T_i \rfloor_L, \lfloor T_{i+1} \rfloor_L, \lfloor T_{i+2} \rfloor_R, \ldots, \lfloor T_n \rfloor_R \vdash false \blacktriangleright I_{i+1}$ such that $I_i \wedge T_{i+1} \Rightarrow I_{i+1}$.

5 Interpolation in the Theory of Arrays

The (non-extensional) first-order theory of arrays [17] is typically formulated over the function symbols *select* and *store* by means of the following axioms:

$$\forall x, y, z.\ select(store(x, y, z), y) \doteq z \tag{3}$$

$$\forall x, y_1, y_2, z.\ \big(y_1 \doteq y_2 \vee select(store(x, y_1, z), y_2) \doteq select(x, y_2)\big) \tag{4}$$

In these and the following formulae, we use general equalities $s \doteq t$ as a shorthand-notation for $s - t \doteq 0$. Intuitively, $select(x, y)$ retrieves the element stored at position y in array x, while $store(x, y, z)$ denotes the array that is identical to x, with the exception that value z is stored at position y.

We have observed in Theorem 2 that QFPAUP is not closed under interpolation. This result directly carries over to QFPA combined with arrays; in fact, it has been noted in [18, 9] that not even the quantifier-free theory of arrays *without* arithmetic is closed under interpolation. An example are the following formulae:

$$\begin{aligned} A = \;& M' \doteq \mathit{store}(M,a,x) \\ B = \;& b \not\doteq c \;\wedge\; \mathit{select}(M',b) \not\doteq \mathit{select}(M,b) \;\wedge\; \mathit{select}(M',c) \not\doteq \mathit{select}(M,c) \end{aligned}$$

Interpolants of the unsatisfiable conjunction $A \wedge B$ must only contain the non-logical symbols M, M' and the theory symbols $\doteq, \mathit{select}, \mathit{store}, \wedge$, etc. The only interpolants over this vocabulary are quantified formulae such as

$$\forall x, y.\; \big(x \doteq y \;\vee\; \mathit{select}(M,x) \doteq \mathit{select}(M',x) \;\vee\; \mathit{select}(M,y) \doteq \mathit{select}(M',y)\big)$$

On the next pages, we will describe an interpolation procedure for arrays that is able to derive quantified interpolants. Our procedure is similar in flavour to the procedures in [11, 10] and works by explicit instantiation of the array axioms. As in Sect. 4, axioms are handled using the rules ALL-LEFT-L/R, which introduce quantifiers in interpolants as needed.

5.1 Presburger Arithmetic with Uninterpreted Functions

As a first step, we consider the combination of PA with uninterpreted functions, which are handled in proofs via an encoding into relations. Uninterpreted functions will later (Sect. 5.2) be used to represent the array operations *select* and *store*.

Recall that P denotes the vocabulary of uninterpreted predicates, and F the vocabulary of uninterpreted functions. We assume that a fresh $(n+1)$-ary uninterpreted predicate $f_p \in P$ exists for every n-ary uninterpreted function $f \in F$. Occurrences of f in a (sub)-formula ϕ can then be rewritten to f_p by means of the following rule:

$$\phi[f(t_1,\ldots,t_n)] \quad \rightsquigarrow \quad \exists x.\; (f_p(t_1,\ldots,t_n,x) \wedge \phi[x]) \tag{5}$$

provided that the terms $t_1,\ldots,t_n$ do not contain variables bound in ϕ. As a further side condition, we require that (5) is never applied underneath negations, which can be ensured by transformation to negation normal form.

Given an arbitrary formula ϕ, we write ϕ_{RE} for the function-free formula derived from ϕ by exhaustive application of (5). By adding explicit *functional consistency axioms*, we can establish the satisfiability-equivalence of ϕ and ϕ_{RE} in the quantifier-free case:[3]

$$FC_f = \quad \forall \bar{x}, y_1, y_2.\; \big(f_p(\bar{x}, y_1) \wedge f_p(\bar{x}, y_2) \;\rightarrow\; y_1 \doteq y_2\big) \tag{6}$$

Lemma 8. *The quantifier-free formula ϕ is satisfiable if and only if the following conjunction is satisfiable:*

$$\phi_{RE} \wedge \bigwedge_{f \in F} FC_f \tag{7}$$

By the lemma, it is sufficient to construct a proof of the negation of (7) (assuming the predicate consistency axioms (2)) in order to show that ϕ is unsatisfiable.

In a proof, the axioms FC_f can be handled by ground instantiation just like the predicate consistency axiom (2): whenever atoms $f_p(\bar{s}, s_0)$ and $f_p(\bar{t}, t_0)$ occur in the antecedent of a sequent, an instance of FC_f can be generated using the rules ALL-LEFT-L/R and the substitution $[\bar{x}/\bar{s}, y_1/s_0, y_2/t_0]$. This form of

[3] In case quantifiers are present, one also needs axioms for totality, which are not considered in this paper.

instantiation is sufficient, because predicates f_p only occur in positive positions in ϕ_{RE}, and therefore only turn up in antecedents. As before, the number of required instances can be kept feasible by formulating suitable refinements [16].

After extracting an interpolant from a proof that contains relations encoding uninterpreted functions (like in (7)), the functions can be re-substituted in the interpolant:

$$f_p(t_1,\ldots,t_n,t_0) \quad \leadsto \quad f(t_1,\ldots,t_n) \doteq t_0 \tag{8}$$

In many practical cases (but not in general, as follows from Theorem 2), it is afterwards possible to eliminate quantifiers from interpolants using simplification rules such as:

$$\forall x.\ \left(x - t \doteq 0 \rightarrow \phi\right) \quad \leadsto \quad [x/t]\phi$$

provided that x does not occur in t. A more systematic study of cases and fragments in which quantifier-free interpolation is possible is planned as future work.

5.2 Presburger Arithmetic with Arrays

We can use the same relational encoding as in Sect. 5.1 for the non-extensional theory of arrays; it is only necessary to add the two axioms (3), (4). We lift these axioms to the relational encoding as follows:

$$AR_1 = \quad \forall x_1,x_2,y,z_1,z_2.\ \left(store_p(x_1,y,z_1,x_2) \wedge select_p(x_2,y,z_2) \ \rightarrow\ z_1 \doteq z_2\right)$$

$$AR_2 = \quad \forall x_1,x_2,y_1,y_2,z,z_1,z_2.\ \left(\begin{array}{ll} & store_p(x_1,y_1,z,x_2) \\ \wedge & select_p(x_1,y_2,z_1) \quad \rightarrow\ y_1 \doteq y_2 \vee z_1 \doteq z_2 \\ \wedge & select_p(x_2,y_2,z_2) \end{array}\right)$$

As in the previous sections, the axioms can be handled by ground instantiation based on literals that occur in antecedents of sequents. This yields an interpolating decision procedure for the combined theory of quantifier-free Presburger arithmetic with arrays:

Theorem 9. *Suppose $\{select, store\} \subseteq F$, and A,B are closed quantifier-free formulae over the vocabulary F (and arbitrary vocabularies C,P of constants and predicates, resp.). The conjunction $A \wedge B$ is unsatisfiable in integer structures that satisfy the axioms* (3), (4) *if and only if there is a formula I such that the following sequent is provable (in the calculus from Sect. 3 and 4):*

$$\begin{array}{c} \lfloor A_{RE} \rfloor_L,\ \lfloor B_{RE} \rfloor_R,\ \lfloor AR_1 \rfloor_L, \lfloor AR_2 \rfloor_L, \lfloor AR_1 \rfloor_R, \lfloor AR_2 \rfloor_R, \\ \{\lfloor PC_p \rfloor_L\}_{p\in P_A}, \{\lfloor PC_p \rfloor_R\}_{p\in P_B}, \{\lfloor FC_f \rfloor_L\}_{f\in F_A}, \{\lfloor FC_f \rfloor_R\}_{f\in F_B} \end{array} \quad \vdash \quad \blacktriangleright I \tag{9}$$

where P_A/P_B are the uninterpreted predicates and F_A/F_B the uninterpreted functions occurring in A/B.

Proofs can be constructed in a deterministic and terminating manner through ground instantiation of the axioms (as illustrated on the previous pages), and as discussed in [19] for arithmetic reasoning.

When instantiating any of the axioms PC_p, FC_p or AR_i, it is often possible to freely choose between the L- and the R-labelled version of the axiom. This choice affects the resulting interpolant and determines which literals will occur in the interpolant. As the instantiation of axioms is triggered by L/R-labelled literals in a sequent, it is usually meaningful to introduce an L-labelled instance of an axiom if all of the triggering literals are labelled with L, and similarly an R-instance if the triggering literals are labelled with R.

If a sequent (9) is provable, then an interpolant of $A \wedge B$ in the theory of quantifier-free Presburger arithmetic with arrays can be obtained from I using the re-substitution rule (8).

Specifically for arrays, we found that the following simplification rule is frequently useful to avoid quantifiers in interpolants:

$$\exists x.\ store(x,s,t) - u \doteq 0 \quad \leadsto \quad select(u,s) - t \doteq 0$$

6 Encoding of C Operations

The next two sections discuss details of the verification of C programs using our interpolation procedures, continuing Sect. 2. In our experiments, the model checker Wolverine [5] (which uses the same infrastructure as the SATABS tool and supports a large range of ANSI-C features) was used to process C programs. Wolverine repeatedly produces interpolation problems by encoding paths of the input program as conjunctions of transition relations of individual statements, formulated over the theory of bitvector arithmetic combined with arrays. In order to pass such conjunctions to an interpolation procedure for PA with arrays, a suitable encoding of bitvector operations into unbounded linear arithmetic has to be chosen. We specify this encoding in a compact way using uninterpreted predicates and functions and axioms, similarly to the axiomatisation of the array theory in Sect. 5.2.

As a natural encoding, we consider the set $\{-2^{n-1}, \ldots, 2^{n-1}-1\} \subset \mathbb{Z}$ as the domain of signed bitvector arithmetic of width n, and the set $\{0, \ldots, 2^n - 1\} \subset \mathbb{Z}$ as the domain of unsigned arithmetic. To specify that some integer is a legal bitvector value, domain predicates `inSigned`/`inUnsigned` are declared that receive the bit-width n as first argument, and the integer value in question as second argument. For each C operation, a corresponding uninterpreted function is introduced that receives, besides the operands, information such as the bit-width as explicit arguments.

As an example, we show the (somewhat simplified) definition of signed bitvector addition in the Princess input format:

Princess

```
/***  Declaration of uninterpreted predicates  ***/
\predicates {    inSigned(int, int);    }
/***  Declaration of uninterpreted functions  ***/
\functions { \partial int shiftLeft(int, int); \partial int addSigned(int, int, int); }

/***  Axioms  ***/
  \forall int x, y; {shiftLeft(x, y)} (
    y > 0 -> shiftLeft(x, y) = shiftLeft(2*x, y-1))
&
  \forall int x; {shiftLeft(x, 0)} shiftLeft(x, 0) = x
&
  \forall int x, width; (inSigned(width, x) ->
    x >= -shiftLeft(1, width - 1) & x < shiftLeft(1, width - 1))
&
  \forall int x, y, width; {addSigned(width, x, y)} (
    (addSigned(width, x, y) = x + y |
     addSigned(width, x, y) = x + y - shiftLeft(1, width) |
     addSigned(width, x, y) = x + y + shiftLeft(1, width)) &
    inSigned(width, addSigned(width, x, y)))
```

Princess

The axioms and declarations are mostly self-explanatory. As is common for SMT solvers, we specify *triggers* after universal quantifiers that state when and how an axiom is to be instantiated. For example, `{shiftLeft(x, y)}` states that the corresponding formula is to be instantiated whenever a term `shiftLeft(`s`,` t`)` occurs in a sequent. Internally, all uninterpreted functions are translated to uninterpreted predicates, and triggers are matched on the literals that occur in the antecedent of sequents.

Similar encodings can be provided for all C operations. A precise translation of non-linear operations like multiplication or bit-wise operations can be done by case analysis over the values of their operands, which in general leads to formulae of exponential size, but is well-behaved in many cases that are practically relevant (e.g., if one of the operands is a literal).

7 Preliminary Experiments

The following listing shows a part of an open-source C program (initialisation and shutdown of an md5 implementation) that was successfully verified not to contain any array bound violations, dereferentiation of possibly dangling pointers, or assertion violations. Comments and layout of the program were changed to accommodate the lack of space, but no modifications were made otherwise.

C

```
/* nettle, low-level cryptographics library. Copyright (C) 2001 Niels Moeller */

void md5init(struct md5_ctx *ctx) {
  ctx->digest[0] = 0x67452301;  ctx->digest[1] = 0xefcdab89;
  ctx->digest[2] = 0x98badcfe;  ctx->digest[3] = 0x10325476;
  ctx->count_l = ctx->count_h = 0;  ctx->index = 0;                              /*1*/
}

static void md5transform(uint32_t *digest, const uint32_t *data) {
  uint32_t a, b, c, d; a = digest[0]; b = digest[1]; c = digest[2]; d = digest[3];
  ROUND(F1, a, b, c, d, data[ 0] + 0xd76aa478, 7);
  ROUND(F1, d, a, b, c, data[ 1] + 0xe8c7b756, 12);
  /* [...] */
}

static void md5final(struct md5_ctx *ctx) {
  uint32_t data[MD5_DATA_LENGTH]; unsigned i, words;
  i = ctx->index;

  assert(i < MD5_DATA_SIZE);
  ctx->block[i++] = 0x80;                                                        /*2*/
  for( ; i & 3; i++) ctx->block[i] = 0;

  words = i >> 2;
  for (i = 0; i < words; i++)
    data[i] = LE_READ_UINT32(ctx->block + 4*i);

  if (words > (MD5_DATA_LENGTH-2)) {
      for (i = words ; i < MD5_DATA_LENGTH; i++) data[i] = 0;
      md5transform(ctx->digest, data);
      for (i = 0; i < (MD5_DATA_LENGTH-2); i++) data[i] = 0;
  } else
    for (i = words ; i < MD5_DATA_LENGTH - 2; i++) data[i] = 0;

  data[MD5_DATA_LENGTH-1] = (ctx->count_h << 9) | (ctx->count_l >> 23);
  data[MD5_DATA_LENGTH-2] = (ctx->count_l << 9) | (ctx->index << 3);
  md5transform(ctx->digest, data);  }

void main(int argc, char **argv) {
  struct md5_ctx ctx; md5init(&ctx); md5final(&ctx);  }
```

C

In order to verify `main` and all functions called from it, 51 program paths are extracted and handed over to the interpolation procedure, from which altogether 519 interpolants are generated. The interpolants range from formulae like $select(\texttt{ctx}, 4) = 0$ at point `/*1*/` (structs are encoded as arrays) to inequalities like $\texttt{i} \leq 1$ at `/*2*/`.

8 Conclusion

In this preliminary report, we have shown how to extend our earlier interpolating calculus for quantifier-free Presburger arithmetic by uninterpreted predicates. We have demonstrated that this extension requires the admission of quantifiers into the logic, which we have accommodated using appropriate additional rules. More importantly, we have shown how to use uninterpreted predicates to encode array operations. These extensions make our interpolation engine a valuable aide in model checking C code with array expressions, as we have demonstrated with initial experimental examples.

In addition to a more detailed analysis of the power of our interpolator in model checking, we plan to investigate more thoroughly which kinds of uninterpreted predicates permit quantifier-free interpolation. We also want to investigate a combination of our calculus with the split-prover approach in [11].

References

[1] Craig, W.: Linear reasoning. a new form of the Herbrand-Gentzen theorem. The Journal of Symbolic Logic **22** (1957) 250–268

[2] Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An interpolating sequent calculus for quantifier-free Presburger arithmetic. In: Proceedings of IJCAR. LNCS, Springer (2010) To appear.

[3] Bultan, T., Gerber, R., Pugh, W.: Symbolic model checking of infinite state systems using Presburger arithmetic. In: CAV. (1997)

[4] Brinkmann, R., Drechsler, R.: RTL-datapath verification using integer linear programming. In: VLSI Design. (2002)

[5] Weissenbacher, G., Kroening, D.: An interpolating decision procedure for transitive relations with uninterpreted functions. In: Proceedings of HVC 2009. LNCS, Springer (2009) To appear.

[6] McMillan, K.L.: Lazy abstraction with interpolants. In: CAV. (2006)

[7] Cimatti, A., Griggio, A., Sebastiani, R.: Interpolant generation for UTVPI. In Schmidt, R.A., ed.: CADE, Berlin, Heidelberg (2009)

[8] Jain, H., Clarke, E., Grumberg, O.: Efficient interpolation for linear diophantine (dis)equations and linear modular equations. In: CAV. (2008)

[9] Kapur, D., Majumdar, R., Zarba, C.G.: Interpolation for data structures. In: SIGSOFT '06/FSE-14, New York, NY, USA, ACM (2006) 105–116

[10] McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: TACAS. (2008)

[11] Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In Hermanns, H., Palsberg, J., eds.: TACAS. Volume 3920 of LNCS, Springer (2006) 459–473

[12] Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer (1996)

[13] Halpern, J.Y.: Presburger arithmetic with unary predicates is Π^1_1 complete. Journal of Symbolic Logic **56** (1991)

[14] Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: LPAR. (2008)

[15] Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Journal of the ACM **52** (2005) 365–473

[16] Rümmer, P.: Calculi for Program Incorrectness and Arithmetic. PhD thesis, University of Gothenburg (2008)

[17] McCarthy, J.: Towards a mathematical science of computation. In Popplewell, C.M., ed.: Information Processing 1962: Proceedings IFIP Congress 62, Amsterdam, North Holland (1963) 21–28

[18] McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. **345** (2005)

[19] Rümmer, P.: A sequent calculus for integer arithmetic with counterexample generation. In: VERIFY. (2007)