

E-Matching with Free Variables

Philipp Rümmer

Department of Information Technology, Uppsala University, Sweden

Abstract. E-matching is the most commonly used technique to handle quantifiers in SMT solvers. It works by identifying characteristic sub-expressions of quantified formulae, named triggers, which are matched during proof search on ground terms to discover relevant instantiations of the quantified formula. E-matching has proven to be an efficient and practical approach to handle quantifiers, in particular because triggers can be provided by the user to guide proof search; however, as it is heuristic in nature, e-matching alone is typically insufficient to establish a complete proof procedure. In contrast, free variable methods in tableau-like calculi are more robust and give rise to complete procedures, e.g., for first-order logic, but are not comparable to e-matching in terms of scalability. This paper discusses how e-matching can be combined with free variable approaches, leading to calculi that enjoy similar completeness properties as pure free variable procedures, but in which it is still possible for a user to provide domain-specific triggers to improve performance.

1 Introduction

SAT and SMT solvers form the backbone of many of today’s verification systems, responsible for discharging verification conditions that encode correctness properties of hardware or software designs. Such verification conditions are often generated in the context of intricate theories, including various kinds of arithmetic, uninterpreted functions and equality, the theory of arrays, or the theory of quantifiers. Despite much research over the past years, efficient and scalable reasoning in the combination of such theories remains challenging: in particular for handling quantifiers, most state-of-the-art SMT solvers have to resort to heuristic techniques like e-matching and triggers [5, 6], which is a popular method due to its simplicity and performance, but which offers little completeness guarantees and is sensitive to syntactic manipulations of input formulae.

This paper takes the standpoint that heuristics like e-matching should be considered as *optimisations*, and triggers as *hints*, possibly affecting the performance, but not the completeness of an SMT solver. In other words, the set of formulae that a solver can prove should be independent from chosen triggers. Working towards this goal, the paper presents calculi integrating constraint-based free variable reasoning with e-matching, the individual contributions being (i) a free variable sequent calculus for first-order logic (Sect. 3), with support for e-matching and user-provided triggers to guide instantiation of quantified formulae, partly inspired by the positive unit hyper-resolution calculus [11, 12]; (ii) a

similar calculus for first-order logic modulo linear integer arithmetic (Sect. 5), extending the calculus in [20]; (iii) as a component of both calculi, an approach to encode functions and congruence closure procedures (commonly used in SMT) as uninterpreted predicate (Sect. 4); (iv) a complete implementation of the calculus (ii), called PRINCESS, and experimental evaluation against SMT solvers competing in the last SMT competition (AUFLIA category) (Sect. 6).

The calculi in (i) and (ii) are sound and complete for fragments such as first-order logic, Presburger arithmetic, the universal and the existential fragment of first-order logic modulo integers, and the languages accepted by related methods like $\mathcal{ME}(LIA)$ [2] and the complete instantiation method in [7]. Since our procedure has strong similarities with the DPLL(T) architecture used in SMT solvers, many optimisations developed in the SMT context are expected to be applicable also to our calculus.

1.1 Introductory Example

We start by illustrating e-matching and free variable methods using an example. The first-order theory of arrays [13] is often encoded using uninterpreted function symbols sel and sto by means of the following axioms:

$$\forall x, y, z. \underline{sel(sto(x, y, z), y)} \doteq z \quad (1)$$

$$\forall x, y_1, y_2, z. (y_1 \doteq y_2 \vee \underline{sel(sto(x, y_1, z), y_2)}) \doteq sel(x, y_2) \quad (2)$$

Intuitively, $sel(x, y)$ retrieves the element of array x stored at position y , while $sto(x, y, z)$ denotes the array that is identical to x , except that position y stores value z . In order to prove that some formula holds over the theory of arrays, the underlined expressions can be used as *triggers* that determine when and how the axioms should be instantiated. Generally, triggers consist of a single or multiple expressions (normally sub-expressions in the body of the quantified formula) that contain all quantified variables. For instance, to prove that the implication

$$b \doteq sto(a, 1, 2) \rightarrow sel(b, 2) \doteq sel(a, 2) \quad (3)$$

holds over the theory of arrays, we can observe that the term $sel(sto(a, 1, 2), 2)$ occurs in the implication, modulo some equational reasoning. This term matches the underlined pattern in (2), and suggests to instantiate (2) to obtain the instance $1 \doteq 2 \vee \underline{sel(sto(a, 1, 2), 2)} \doteq sel(a, 2)$. In fact, (3) follows for this instance of (2), when reasoning in the theories of uninterpreted functions and arithmetic, which allows us to conclude the validity of (3).

The axioms and triggers shown above are commonly used in SMT solvers, and give rise to an efficient decision procedure for ground problems over arrays. However, in the presence of quantifiers, e-matching might be unable to determine the right instantiations, possibly because required instantiations do not yet exist as ground terms in the formula. For instance, variants of (3) might include:

$$b \doteq sto(a, 1, 2) \rightarrow \exists x. sel(b, x) \doteq sel(a, 2) \quad (4)$$

$$b \doteq sto(a, 1, 2) \rightarrow \exists x. sel(b, x + 1) \doteq sel(a, 2) \quad (5)$$

$$b \doteq sto(a, 1, 2) \rightarrow \exists x. sel(b, x) \doteq sel(a, x) \quad (6)$$

Though the formulae are still valid over the domain of integers, the match $sel(sto(a, 1, 2), 2)$ used previously has been eliminated, which makes proof search more intricate. The state-of-the-art e-matching-based SMT solver CVC3 ([1], version 2.4.1) is able to solve (3), but none of (4), (5), (6). A more realistic example, though similar in nature to the formulae shown here, was reported in [10, Sect. 3.3], where a simple modification (Skolemisation) of a small formula prevented Z3 [16] from finding a proof. The goal of the calculus developed in this paper (and of our implementation PRINCESS) is to obtain a system that is more robust against such modifications, by combining e-matching with constraint-based free variable reasoning, while retaining the scalability of SMT solvers.

The general philosophy of free variable methods [9] is to delay the choice of instantiations for quantified formulae with the help of symbolic reasoning. For example, we could instantiate the formula $\exists x.sel(b, x + 1) \doteq sel(a, 2)$ using a free variable X , resulting in $sel(b, X + 1) \doteq sel(a, 2)$. Modulo equational reasoning, this creates the term $sel(sto(a, 1, 2), X + 1)$, which can be unified with the trigger in (2) under the constraint $X \doteq 1$. It is then possible to proceed with the proof as described above. After closing the proof, we can conclude that (5) indeed holds, since the derived constraint $X \doteq 1$ is satisfiable: it is possible (retrospectively) to instantiate $\exists x.sel(b, x + 1) \doteq sel(a, 2)$ with the concrete term $X = 1$.

This example demonstrates that a free variable calculus can be used to compute answers to queries, in a manner similar to constraint logic programming. The system developed in this paper is more general than “ordinary” logic programming, however, since no restrictions on the use of quantifiers are imposed.

2 Background

2.1 Syntax and Semantics of Considered Logics

We assume familiarity with classical first-order logic (e.g., [9]). Let x range over an infinite set X of variables, c over an infinite set C of constants, 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. (Note the distinction between constant *symbols*, such as c , and integer *literals*, such as 42). The syntax of logics considered in this paper is defined by the following grammar:

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

The symbol t denotes terms of linear arithmetic. A formula ϕ is called *closed* if all variables in ϕ are bound by quantifiers, and *ground* if it does not contain variables or quantifiers. A location within a formula ϕ is called *positive* if it is underneath an even number of negations \neg , otherwise *negative* (we also speak of positive/negative *polarity*). Simultaneous substitution of terms $\bar{t} = (t_1, \dots, t_n)$ for variables $\bar{x} = (x_1, \dots, x_n)$ in ϕ is denoted by $[\bar{x}/\bar{t}]\phi$; we assume that variable capture is avoided by renaming bound variables as necessary. For simplicity, we

sometimes write $s \doteq t$ as a shorthand of $s - t \doteq 0$. The abbreviation *true* (*false*) stands for $0 \doteq 0$ ($1 \doteq 0$), and implication $\phi \rightarrow \psi$ for $\neg\phi \vee \psi$.

We consider fragments of the syntax shown above, including function-free first-order logic (Sect. 2.3, 3), full first-order logic (Sect. 4), and first-order logic with linear integer arithmetic (Sect. 5). Semantics of any such logic \mathcal{L} is defined by identifying a class $\mathcal{S}_{\mathcal{L}}$ of structures (U, I) , where U is a non-empty *universe*, and I is an *interpretation* that maps predicates $p \in P$ to relations over U , functions $f \in F$ to set-theoretic functions over U , and constants $c \in C$ to values in U . Given a structure (U, I) (and a variable assignment), the evaluation of terms and formulae is defined recursively as is common. A closed formula is called *valid* if it evaluates to *true* for all structures $(U, I) \in \mathcal{S}_{\mathcal{L}}$, and *satisfiable* if it evaluates to *true* for at least one structure.

2.2 Sequent Calculi with Constraints

Throughout the paper we will work with the *constraint sequent calculus* that is introduced in [20]. The calculus differs from normal Gentzen-style sequent calculi [9] in that every sequent $\Gamma \vdash \Delta$ is annotated with a constraint C (written $\Gamma \vdash \Delta \Downarrow C$) that captures unification conditions derived in a sub-proof. Such unification conditions come into play when free variables (which technically are treated as constants) are used to instantiate quantified formulae. All calculi in this paper are designed such that constraints cannot contain uninterpreted predicates or functions, so that validity/satisfiable of constraints is decidable.

More formally, if Γ, Δ are finite sets of closed formulae (the *antecedent* and *succedent*) and C is a closed formula, then $\Gamma \vdash \Delta \Downarrow C$ is called a *constrained sequent*. A sequent $\Gamma \vdash \Delta \Downarrow C$ is called *valid* if the formula $(\bigwedge \Gamma \wedge C) \rightarrow \bigvee \Delta$ is valid. A calculus rule is a binary relation between finite sets of sequents (the premises) and single sequents (the conclusion). Proof trees are defined as is common as trees growing upwards in which each node is labeled with a constrained sequent, and in which each node that is not a leaf is related with the nodes directly above through an instance of a calculus rule. A proof is closed if it is finite, and if all leaves are justified by a rule instance without premises.

2.3 The Basic Calculus for Function-Free First-Order Logic

At the core of all calculi introduced in this paper is a calculus for basic first-order logic (FOL) with equality, at this point including uninterpreted predicates, but no functions:

$$\phi_{\text{FOL}} ::= \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \forall x.\phi \mid \exists x.\phi \mid s \doteq s \mid p(\bar{s}) \quad s ::= c \mid x$$

Since functions and arithmetic were not included in the logic, terms can only be (symbolic) constants or bound variables. Semantics is defined over the class \mathcal{S}_{FOL} of structures (U, I) with arbitrary non-empty universe U . The constraint calculus PredEq^C for the logic is shown in Fig. 1, with constraints consisting of (possibly negated) equalities, Boolean connectives, and quantifiers. The validity

$$\begin{array}{c}
\frac{\Gamma, \phi \vdash \Delta \Downarrow C \quad \Gamma, \psi \vdash \Delta \Downarrow D}{\Gamma, \phi \vee \psi \vdash \Delta \Downarrow C \wedge D} \vee L \quad \frac{\Gamma, \phi, \psi \vdash \Delta \Downarrow C}{\Gamma, \phi \wedge \psi \vdash \Delta \Downarrow C} \wedge L \quad \frac{\Gamma \vdash \phi, \Delta \Downarrow C}{\Gamma, \neg \phi \vdash \Delta \Downarrow C} \neg L \\
\frac{\Gamma \vdash \phi, \Delta \Downarrow C \quad \Gamma \vdash \psi, \Delta \Downarrow D}{\Gamma \vdash \phi \wedge \psi, \Delta \Downarrow C \wedge D} \wedge R \quad \frac{\Gamma \vdash \phi, \psi, \Delta \Downarrow C}{\Gamma \vdash \phi \vee \psi, \Delta \Downarrow C} \vee R \quad \frac{\Gamma, \phi \vdash \Delta \Downarrow C}{\Gamma \vdash \neg \phi, \Delta \Downarrow C} \neg R \\
\frac{\Gamma, [x/c]\phi, \forall x. \phi \vdash \Delta \Downarrow [x/c]C}{\Gamma, \forall x. \phi \vdash \Delta \Downarrow \exists x. C} \forall L \quad \frac{\Gamma, [x/c]\phi \vdash \Delta \Downarrow [x/c]C}{\Gamma, \exists x. \phi \vdash \Delta \Downarrow \forall x. C} \exists L \quad \frac{\Gamma \vdash \Delta \Downarrow C}{\Gamma, s \doteq t \vdash \Delta \Downarrow s \not\doteq t \vee C} =L \\
\frac{\Gamma \vdash [x/c]\phi, \exists x. \phi, \Delta \Downarrow [x/c]C}{\Gamma \vdash \exists x. \phi, \Delta \Downarrow \exists x. C} \exists R \quad \frac{\Gamma \vdash [x/c]\phi, \Delta \Downarrow [x/c]C}{\Gamma \vdash \forall x. \phi, \Delta \Downarrow \forall x. C} \forall R \quad \frac{*}{\Gamma \vdash s \doteq t, \Delta \Downarrow s \doteq t} =R \\
\frac{*}{\Gamma, p(s_1, \dots, s_n) \vdash p(t_1, \dots, t_n), \Delta \Downarrow \bigwedge_i s_i \doteq t_i} \text{PC} \quad \frac{[s/t]\Gamma, s \doteq t \vdash [s/t]\Delta \Downarrow C}{\Gamma, s \doteq t \vdash \Delta \Downarrow C} =\text{RED}
\end{array}$$

Fig. 1. The rules of the calculus PredEq^C for first-order predicate logic. In all rules, c is a constant that does not occur in the conclusion: in contrast to the use of Skolem functions and free variables in tableaux, the same kinds of symbols (constants) are used to handle both existential and universal quantifiers. Arbitrary renaming of bound variables is allowed in the constraints when necessary to avoid variable capture.

of formulae of this kind is decidable by quantifier elimination [9]. The calculus is analytic and contains two rules for each formula constructor, as well as a closure rule PC to unify complementary literals. As an optimisation, the rule =RED can be used to destructively apply equations; the rule is not necessary to establish completeness, but relevant (together with further refinements) to turn PredEq^C into a practical calculus [20, 19].

Lemma 1 (Soundness). *If a sequent $\Gamma \vdash \Delta \Downarrow C$ is provable in PredEq^C , then it is valid (holds in all \mathcal{S}_{FOL} -structures).*

In particular, proving a sequent $\Gamma \vdash \Delta \Downarrow C$ with a valid constraint C implies that also the implication $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is valid. This gives rise to a constraint-based proof procedure that iteratively constructs proof trees for an input sequent $\Gamma \vdash \Delta \Downarrow ?$ with a yet unknown constraint. The constraints in a proof can be filled in once all proof branches have been closed. In each iteration, the procedure checks whether the constraint generated by the current proof is valid, in which case the procedure can terminate with the result that the input problem has been proven; otherwise, the current proof has to be unfolded further. Strategies for generating proofs (without the need for backtracking, i.e., undoing previous proof steps) are discussed in [20].

Example 2. We show how to prove $\neg \forall x. (\neg p(x) \vee x \doteq c) \vee \neg p(d) \vee p(c)$, in which $p \in P$ is a unary predicate and $c, d \in C$ are constants:

$$\begin{array}{c}
\frac{*}{p(d) \vdash p(a) \Downarrow d \doteq a} \text{PC} \quad \frac{*}{a \doteq c, p(d) \vdash p(c) \Downarrow d \doteq c} \text{PC} \\
\frac{\neg p(a), p(d) \vdash \dots \Downarrow d \doteq a \quad \bar{a} \doteq \bar{c}, p(d) \vdash p(c) \Downarrow a \not\doteq c \vee d \doteq c}{\dots, \neg p(a) \vee a \doteq c, p(d) \vdash p(c) \Downarrow d \doteq a \wedge (a \not\doteq c \vee d \doteq c)} \vee L \\
\frac{\forall x. (\neg p(x) \vee x \doteq c), p(d) \vdash p(c) \Downarrow R}{\vdash \neg \forall x. (\neg p(x) \vee x \doteq c) \vee \neg p(d) \vee p(c) \Downarrow R} \vee R^*, \neg R^*
\end{array}$$

In order to instantiate the universal quantifier, the fresh constant a is introduced; the constant is quantified existentially in the derived constraints, and therefore can be seen as a “free variable.” The constraints on the right-hand side of \Downarrow are practically filled in *after* closing the proof using PC, and do not contain the predicate p , so that their validity can be decided. The validity of the original formula $\neg\forall x.(\neg p(x) \vee x \doteq c) \vee \neg p(d) \vee p(c)$ follows from the validity of the final constraint $R = \exists x.(d \doteq x \wedge (x \not\equiv c \vee d \doteq c))$.

Lemma 3 (Completeness). *Suppose ϕ is closed and valid. Then there is a valid constraint C such that $\vdash \phi \Downarrow C$ is provable in PredEq^C .*

3 Positive Unit Hyper-Resolution

As argued in Sect. 1.1, axioms and quantified formulae (in particular in verification problems) are often manually formulated with a clear, directed application strategy in mind. This makes it possible to systematically instantiate axioms in a manner that more resembles the execution of a functional or logic program than the search for a proof. From a practical point of view, providing support for this style of reasoning (even if it is only applicable to a subset of input problems) is crucial to achieve the scalability needed for applications. We integrate such user-guided reasoning into our calculus with the help of concepts from the *positive unit hyper-resolution* (PUHR) calculus, an approach first used in the SATCHMO theorem prover [11, 12]. PUHR will be used in Sect. 4 to simulate the e-matching method common in SMT solvers.

PUHR is a tableau procedure in which clauses are instantiated by matching negative literals on (ground) literals already present on a proof branch. Starting from the calculus PredEq^C defined in the last section, we introduce a similar rule in our hyper-resolution sequent calculus PredEqHR^C , instantiating quantified formulae that are “guarded” by negative literals $\neg p_1(\bar{t}_1), \dots, \neg p_n(\bar{t}_n)$ using symbols from matching literals $p_n(\bar{s}_n), \dots, p_n(\bar{s}_n)$ in the antecedent of a sequent:

$$\frac{\Gamma, \{p_i(\bar{s}_i)\}_{i=1}^n, \forall \bar{x}. (\bigvee_{i=1}^n \neg p_i(\bar{t}_i) \vee \phi), \text{simp}(\forall \bar{x}. (\bigvee_{i=1}^n \bar{s}_i \not\equiv \bar{t}_i \vee \phi)) \vdash \Delta \Downarrow C}{\Gamma, \{p_i(\bar{s}_i)\}_{i=1}^n, \forall \bar{x}. (\bigvee_{i=1}^n \neg p_i(\bar{t}_i) \vee \phi) \vdash \Delta \Downarrow C} \forall\text{L-M}$$

Given literals $\{p_i(\bar{s}_i)\}_{i=1}^n$ in a sequent, a quantified formula $\forall \bar{x}. (\bigvee_{i=1}^n \neg p_i(\bar{t}_i) \vee \phi)$ can be instantiated using the argument terms \bar{s}_i by simultaneously solving the systems $\bar{s}_i \doteq \bar{t}_i$ of equalities. In contrast to the original PUHR [11], we do not require formulae to be range restricted. Note that the formula ϕ might be *false* and disappear, and that the literals $\{p_i(\bar{s}_i)\}_{i=1}^n$ are not necessarily distinct. The solving of equalities is formulated using a recursive simplification function *simp*:

$$\begin{aligned} \text{simp}(\forall \bar{x}. (t \not\equiv t \vee \phi)) &= \text{simp}(\forall \bar{x}. \phi) \\ \text{simp}(\forall \bar{x}. (x_i \not\equiv t \vee \phi)) &= \text{simp}(\forall \bar{x}. [x_i/t]\phi) && (x_i \neq t) \\ \text{simp}(\forall \bar{x}. (t \not\equiv x_i \vee \phi)) &= \text{simp}(\forall \bar{x}. [x_i/t]\phi) && (x_i \neq t) \\ \text{simp}(\forall \bar{x}. (s \not\equiv t \vee \phi)) &= s \not\equiv t \vee \text{simp}(\forall \bar{x}. \phi) && (s, t \notin \bar{x}) \\ \text{simp}(\forall \bar{x}. \phi) &= \forall (\bar{x} \cap \text{fv}(\phi)). \phi && (\text{otherwise}) \end{aligned}$$

A rule $\exists\text{R-M}$ similar to $\forall\text{L-M}$ is introduced for existentially quantified formulae $\exists\bar{x}. (\bigwedge_{i=1}^n p_i(\bar{t}_i) \wedge \phi)$ in the succedent. The soundness of the new rules is immediate, since the rules only introduce instances of quantified formulae already present in a sequent. After adding $\forall\text{L-M}$ and $\exists\text{R-M}$, it is possible to impose the side-condition that the rule $\forall\text{L}$ is no longer allowed to be applied to formulae $\forall\bar{x}. (\bigvee_{i=1}^n \neg p_i(\bar{t}_i) \vee \phi)$; similarly for $\exists\text{R}$. In other words, the ordinary rules $\forall\text{L}$ and $\exists\text{R}$ may only be applied to formulae that do not start with negative literals. We denote the resulting calculus by PredEqHR^C .

Example 4. We show how the proof from Example 2 can be carried over to PredEqHR^C . To this end, observe that the formula $\forall x. (\neg p(x) \vee x \doteq c)$ in the antecedent is amenable to hyper-resolution, so that it is no longer necessary to introduce the constant a in the proof. Also proof splitting can now be avoided:

$$\frac{\frac{\frac{d \doteq c, p(d) \vdash p(c) \Downarrow d \doteq c}{\dots, d \doteq c, p(d) \vdash p(c) \Downarrow d \neq c \vee d \doteq c} =\text{L}}{\forall x. (\neg p(x) \vee x \doteq c), p(d) \vdash p(c) \Downarrow \text{true}} \forall\text{L-M}}{\vdash \neg \forall x. (\neg p(x) \vee x \doteq c) \vee \neg p(d) \vee p(c) \Downarrow \text{true}} \forall\text{R}^*, \neg\text{R}^*$$

$\forall\text{L-M}$ introduces the formula $\text{simp}(\forall x. (d \neq x \vee x \doteq c))$, which can directly be simplified to $d \doteq c$. A further optimization is the use of $=\text{RED}$ to minimize occurring constraints.

Lemma 5 (Completeness). *Suppose ϕ is closed, valid, and does not contain constants. Then there is a valid constraint C such that $\vdash \phi \Downarrow C$ is provable in PredEqHR^C .*

4 E-Matching through Relational Encoding

For practical applications, uninterpreted functions are more common and often more important than uninterpreted predicates. Uninterpreted functions and equalities are in SMT solvers normally represented using congruence closure methods [18], which build a *congruence graph* (also called *e-graph*) containing nodes for all function terms present in a problem, with edges representing asserted equalities. More formally, given a finite subterm-closed set T of terms and a finite set E of equalities, the congruence graph is the undirected graph (T, E') , where $E' \supseteq E$ is the smallest transitive and reflexive set of edges satisfying:

if $f(s_1, \dots, s_n), f(t_1, \dots, t_n) \in T$ are nodes with $\{(s_1, t_1), \dots, (s_n, t_n)\} \subseteq E'$,
then also $(f(s_1, \dots, s_n), f(t_1, \dots, t_n)) \in E'$.

The relation E' is normally constructed by fixed-point iteration, starting from the given equalities E . Congruence graphs can be used to efficiently decide whether an equality $s \doteq t$ follows from the set E of equalities. In an SMT solver, congruence graphs are dynamically updated each time new terms or equalities occur. The congruence graph is also used as the underlying datastructure for e-matching, since matching terms (modulo equations) can efficiently be found using the congruence graph. We discuss in this section how both congruence closure and e-matching can be encoded using uninterpreted predicates.

4.1 Relational Encoding of Functions

We consider first-order logic including function symbols, which means that the grammar for terms shown in the beginning of Sect. 2.3 is extended to:

$$s ::= c \mid x \mid f(s, \dots, s)$$

where $f \in F$ ranges over function symbols. For the purpose of the encoding of functions into relations, we assume that a fresh $(n + 1)$ -ary uninterpreted predicate $f_p \in P$ exists for every n -ary uninterpreted function $f \in F$, representing the graph of f . The relation f_p satisfies two axioms, *functionality* and *totality*:

$$Fun_f = \forall \bar{x}, y_1, y_2. (\neg f_p(\bar{x}, y_1) \vee \neg f_p(\bar{x}, y_2) \vee y_1 \doteq y_2), \quad Tot_f = \forall \bar{x}. \exists y. f_p(\bar{x}, y).$$

We can then translate from formulae ϕ over the functional vocabulary F (and relational vocabulary P) to formulae ϕ_{Rel} purely over the relational vocabulary P . This can be done by means of the following rewriting rules:

$$\begin{aligned} \exists\text{-enc:} \quad & \psi[f(\bar{t})] \rightsquigarrow \exists x. (f_p(\bar{t}, x) \wedge \psi[x]) \\ \forall\text{-enc:} \quad & \psi[f(\bar{t})] \rightsquigarrow \forall x. (\neg f_p(\bar{t}, x) \vee \psi[x]) \end{aligned}$$

Both rules have the side condition that rewritten occurrences of $f(\bar{t})$ must not be in the scope of quantifiers (in ϕ) binding variables in the terms \bar{t} ; furthermore, the variable x must be fresh in ϕ . It is possible, however, to apply the rewriting rules to arbitrary sub-formulae of a given formula ϕ ; in other words, the predicate and quantifier that encode a function application $f(\bar{t})$ can be placed arbitrarily in the rewritten formula, as long as the function application remains in the scope of the quantifier. Rewriting strategies are discussed later in this section.

Lemma 6. *Suppose ϕ is a closed formula over the vocabulary F , and ϕ_{Rel} is a function-free formula obtained from ϕ by application of the rewriting rules \exists -enc and \forall -enc. Then ϕ is valid iff $\bigwedge_{f \in F} (Fun_f \wedge Tot_f) \rightarrow \phi_{Rel}$ is valid.*

Since the calculi $PredEq^C$ and $PredEqHR^C$ are sound and complete for first-order logic without function symbols, we can therefore construct calculi for first-order logic including functions by first encoding functions as relations.

4.2 Ground Reasoning and Congruence Closure

We first concentrate on quantifier-free first-order formulae with functions. In this setting, it is easy to see that the hyper-resolution calculus $PredEqHR^C$, in combination with the functionality axioms Fun_f for functions f , is able to simulate congruence closure procedures. This is supported by the following strengthened version of Lem. 6, which observes that it is sufficient to consider partial functions when solving essentially ground formulae: it is not necessary to take totality axioms into account to derive an equivalence similar to the one in Lem. 6.

Lemma 7. *Suppose ϕ is a closed formula over the vocabulary F , and ϕ_{Rel} a function-free formula obtained from ϕ by application of the rewriting rules \exists -enc and \forall -enc that contains \forall -quantifiers only in positive positions, and \exists -quantifiers only in negative positions. Then ϕ is valid iff $\bigwedge_{f \in F} Fun_f \rightarrow \phi_{Rel}$ is valid.*

Example 9. Consider the quantified formula $\forall x.f(x) \doteq g(x)$. Four possible ways of encoding the formula using relations, corresponding to different strategies when applying the rules \forall -enc and \exists -enc, are:

$$\forall x.\exists y, z.(f_p(x, y) \wedge g_p(x, z) \wedge y \doteq z) \quad (7)$$

$$\forall x, y.(\neg f_p(x, y) \vee \exists z.(g_p(x, z) \wedge y \doteq z)) \quad (8)$$

$$\forall x, z.(\neg g_p(x, z) \vee \exists y.(f_p(x, y) \wedge y \doteq z)) \quad (9)$$

$$\forall x, y, z.(\neg f_p(x, y) \vee \neg g_p(x, z) \vee y \doteq z) \quad (10)$$

Each of the relational formulae corresponds to a particular selection of triggers in $\forall x.f(x) \doteq g(x)$:

- in (7), no triggers have been chosen, with the result that the hyper-resolution rule \forall L-M is not applicable. Instantiation of (7) is only possible using the rule \forall L, replacing the bound variable x with an existentially quantified constant that can later unified with some term.
- in (8), the term $f(x)$ (corresponding to the negative literal $f_p(x, y)$) has been selected as trigger. In the calculus PredEqHR^C , (8) can only be instantiated using the rule \forall L-M, and only in case a literal $f_p(s, t)$ occurs in the antecedent of a sequent, substituting the terms s, t for the variables x, y . This corresponds to e-matching the expression $f(x)$ on a node $f(t)$ of a congruence graph. No free variables are needed to instantiate (8).
- similarly, in (9) the term $g(x)$ is trigger.
- in (10), both terms $f(x), g(x)$ have been chosen as a *multi-trigger*, which means that (10) only can be instantiated if literals $f_p(s, t)$ and $g_p(s', t')$ occur in an antecedent. In this case, the instance $s \not\equiv s' \vee t \doteq t'$ will be generated, expressing that the equality $t \doteq t'$ can be assumed if s and s' are unifiable. In terms of e-graphs, the formula would only be instantiated if the e-graph contains nodes $f(s), g(s')$ such that s, s' are in the same equivalence class.

The following proof fragment illustrates how (9) can be instantiated referring to a literal $g_p(a, b)$ in the antecedent, effectively adding $f_p(a, b)$ to the sequent:

$$\frac{\frac{\frac{g_p(a, b), (9), f_p(a, b), b \doteq u \vdash \Downarrow [y/u]C}{g_p(a, b), (9), f_p(a, u), u \doteq b \vdash \Downarrow [y/u]C} =_{\text{RED}}}{g_p(a, b), (9), \exists y.(f_p(a, y) \wedge y \doteq b) \vdash \Downarrow \forall y.C} \exists\text{L}, \wedge\text{L}}{g_p(a, b), (9) \vdash \Downarrow \forall y.C} \forall\text{L-M} \quad (11)$$

The way in which a formula ϕ is translated to ϕ_{Rel} determines how quantified sub-formulae are instantiated, in the same way as SMT solvers can be guided by specifying triggers (Alg. 1 shows how the translation can be done systematically, for a given set of triggers). However, it can be observed that the four encodings (7)–(10) are all equivalent w.r.t. provability of theorems: in combination with the axioms $\text{Fun}_f, \text{Fun}_g, \text{Tot}_f, \text{Tot}_g$ each of the formulae can simulate each other formula. *The choice of triggers in formulae therefore only influences efficiency, not completeness.* For instance, formula (9) in (11) can be replaced

Algorithm 1: ENCODETRIGGER: relational encoding of a quantified formula for a specific set of triggers

Input: Formula $\forall \bar{x}.\phi$, set T of trigger terms with variables from \bar{x}

Output: Relational formula ϕ_{Rel}

$qvars \leftarrow \{x \mid x \in \bar{x}\};$

$premises \leftarrow \emptyset;$

while T contains function terms **do**

 pick (sub)term $f(\bar{t})$ in T s.t. \bar{t} does not contain functions;

 pick fresh variable y ;

$qvars \leftarrow qvars \cup \{y\};$

$premises \leftarrow premises \cup \{f_p(\bar{t}, y)\};$

 substitute y for $f(\bar{t})$ everywhere in T and ϕ ;

end

apply \exists -enc exhaustively to ϕ ;

return $\forall_{x \in qvars} . (\bigvee_{p \in premises} \neg p \vee \phi)$;

with (8) in the following way (the constraints of the sequents have been left out for sake of brevity):

$$\begin{array}{c}
\frac{*}{\dots \vdash x \doteq a} =R \quad \frac{\frac{f_p(x, b), g_p(x, b), g_p(a, b), v \doteq b \vdash}{f_p(x, v), g_p(x, v), g_p(a, b), v \doteq b \vdash} =RED}{\dots, f_p(x, v), g_p(x, v), g_p(a, b), x \not\equiv a \vee v \doteq b \vdash} \forall L, \neg L \\
\frac{\dots, f_p(x, v), g_p(x, v), g_p(a, b), x \not\equiv a \vee v \doteq b \vdash}{\frac{Fun_g, f_p(x, v), g_p(x, v), v \doteq u, g_p(a, b) \vdash}{Fun_g, f_p(x, u), g_p(x, v), u \doteq v, g_p(a, b) \vdash} =RED} \forall L-M \\
\frac{Fun_g, f_p(x, u), g_p(x, v), u \doteq v, g_p(a, b) \vdash}{Fun_g, \dots, f_p(x, u), \exists z. (g_p(x, z) \wedge u \doteq z), g_p(a, b) \vdash} \exists L, \wedge L \\
\frac{Fun_g, \dots, f_p(x, u), \exists z. (g_p(x, z) \wedge u \doteq z), g_p(a, b) \vdash}{\frac{Fun_g, Tot_f, f_p(x, u), g_p(a, b), (8) \vdash}{Fun_g, Tot_f, g_p(a, b), (8) \vdash} \forall L, \exists L} \forall L-M
\end{array}$$

This illustrates that PUHR/e-matching-based reasoning (through $\forall L-M$ and $\exists R-M$) can be mixed freely with free variable reasoning (through $\forall L$ and $\exists R$). Proofs constructed without applying the rules $\forall L$ and $\exists R$ closely correspond to the ground reasoning in an SMT solver, while each application of $\forall L$ or $\exists R$ conceptually introduces a free variable that, at a later point during proof construction, can be unified with other terms, extracting unification conditions in the form of constraints.

5 Extension to Linear Integer Arithmetic

All techniques discussed so far carry over to first-order logic modulo the theory of linear integer arithmetic (FOL(LIA)), via integration into the calculus defined in [20]. The syntax of FOL(LIA) is defined by the grammar in the beginning of Sect. 2.1 and combines first-order logic (with uninterpreted predicates and functions) with arithmetic terms and predicates. Semantics is defined over structures (\mathbb{Z}, I) with the set of integers as universe.

$$\begin{array}{c}
\frac{\Gamma, t \doteq 0 \vdash \phi[s + \alpha \cdot t], \Delta \Downarrow C}{\Gamma, t \doteq 0 \vdash \phi[s], \Delta \Downarrow C} =_{\text{RED-Z}} \quad \frac{\Gamma, s \leq 0, t \leq 0, \alpha s + \beta t \leq 0 \vdash \Delta \Downarrow C}{\Gamma, s \leq 0, t \leq 0 \vdash \Delta \Downarrow C} \leq_{\text{L-Z}} \\
\frac{\Gamma, p(s_1, \dots, s_n) \vdash p(t_1, \dots, t_n), \bigwedge_i s_i - t_i \doteq 0, \Delta \Downarrow C}{\Gamma, p(s_1, \dots, s_n) \vdash p(t_1, \dots, t_n), \Delta \Downarrow C} \text{PU-Z} \\
\frac{*}{\Gamma, \phi_1, \dots, \phi_n \vdash \psi_1, \dots, \psi_m, \Delta \Downarrow \neg\phi_1 \vee \dots \vee \psi_1 \vee \dots} \text{CLOSE}
\end{array}$$

Fig. 2. A selection of rules of the calculus PresPred^C ; for a complete list see [20]. In $=_{\text{RED-Z}}$, α is a literal; we write $\phi[s]$ in the succedent to denote that s occurs in an arbitrary formula in the sequent, which can in particular also be in the antecedent. In $\leq_{\text{L-Z}}$, $\alpha, \beta > 0$ are positive literals. In CLOSE-Z , the formulae $\phi_1, \dots, \phi_n, \psi_1, \dots, \psi_m$ do not contain uninterpreted predicates.

As for FOL, we first introduce a calculus for the function-free fragment of FOL(LIA). The integration of functions is then done in the same way as in Sect. 3, 4 with the help of a relational encoding. The calculus PresPred^C for the function-free fragment consists of the rules in Fig. 1, together with a number of rules specific for linear integer arithmetic, a selection of which are shown in Fig. 2 (as a result, the rules $=_{\text{L}}$, $=_{\text{R}}$, $=_{\text{RED}}$, and PC of the first-order calculus can be removed); in the full calculus, also simplification and splitting rules are needed [20]. A more general closure rule CLOSE has to be used than in PredEq^C : since integer arithmetic is not convex, also disjunctive constraints have to be considered. Constraints in PresPred^C are always formulae in Presburger arithmetic (PA), i.e., do not contain uninterpreted predicates.

Lemma 10 (Soundness [20]). *If a sequent $\Gamma \vdash \Delta \Downarrow C$ can be proven in PresPred^C , then it is valid.*

The logic FOL(LIA) subsumes Presburger arithmetic. Since the logic of quantified Presburger arithmetic with predicates is Π_1^1 -complete [8], no complete calculi can exist for FOL(LIA); however, it can be shown that the calculi introduced in this section are complete for relevant and non-trivial fragments:

Lemma 11 (Completeness [20]). *Suppose ϕ is a closed function-free formula in one of the following fragments:*

- (i) ϕ does not contain uninterpreted predicates (i.e., in Presburger arithmetic);
- (ii) ϕ contains universal (exist.) quantifiers only in positive (negative) positions;
- (iii) ϕ contains universal (exist.) quantifiers only in negative (positive) positions;
- (iv) ϕ is of the form $\forall \bar{x}.(\sigma \rightarrow \psi)$, where σ is a formula in Presburger arithmetic (without uninterpreted predicates) that has only finitely many solutions in \bar{x} , and ψ contains universal (existential) quantifiers only in negative (positive) positions (i.e., a formula accepted by the $\mathcal{ME}(\text{LIA})$ calculus [2]).

Then there is a valid constraint C such that $\vdash \phi \Downarrow C$ is provable in PresPred^C .

Practically, it can be observed that PresPred^C can often also be applied successfully to formulae outside of those fragments.

5.1 Hyper-Resolution and E-Matching for FOL(LIA)

The unit hyper-resolution rule $\forall\text{L-M}$ (and similarly the rule $\exists\text{R-M}$) defined in Sect. 3 can be integrated in the calculus PresPred^C in the same way as in the earlier first-order calculi, in order to instantiate formulae $\forall\bar{x}.(\bigvee_{i=1}^n \neg p_i(\bar{t}_i) \vee \phi)$ by matching. In this context, the simplification function *simp* can be (but does not have to be) replaced with a function tailored to integer arithmetic, i.e., a function that is able to solve the system $\bigvee_{i=1}^n \bar{s}_i \neq \bar{t}_i$ modulo integer arithmetic.

The calculus PresPredHR^C is derived from PresPred^C by adding the rules $\forall\text{L-M}$ and $\exists\text{R-M}$, and by imposing the side condition that the rule $\forall\text{L}$ is no longer applied to formulae of the shape $\forall\bar{x}.(\bigvee_{i=1}^n \neg p_i(\bar{t}_i) \vee \phi)$; similarly for the rule $\exists\text{R}$. As before, the soundness of the rules $\forall\text{L-M}$ and $\exists\text{R-M}$ is immediate. We can also observe that PresPredHR^C is relatively complete, in the sense that formulae that are provable in PresPred^C can also be proven using PresPredHR^C :

Lemma 12. *Suppose $\Gamma \vdash \Delta \Downarrow C$ is provable in PresPred^C , where C is a valid constraint. Then there is a valid constraint C' such that $\Gamma \vdash \Delta \Downarrow C'$ is provable in PresPredHR^C .*

Encoding of functions. The relational encoding of functions from Sect. 4 can be used to obtain a calculus for the full logic FOL(LIA) with functions. Although there are no complete calculi for the full logic, we can observe that PresPred^C (and therefore, by Lem. 12, PresPredHR^C) can handle at least all formulae that can be proven by considering a finite set of ground instances:

Lemma 13. *Suppose $\exists\bar{x}.\phi$ is a closed formula in FOL(LIA), with functions ranging over the finite set F , such that ϕ is quantifier-free. If there is a valid disjunction $\bigvee_{i=1}^n [\bar{x}/\bar{t}_i]\phi$ of ground instances of $\exists\bar{x}.\phi$, then there is a valid constraint C such that $\{\text{Fun}_f, \text{Tot}_f\}_{f \in F} \vdash (\exists\bar{x}.\phi)_{\text{Rel}} \Downarrow C$ is provable in PresPred^C .*

The lemma directly generalises to disjunctions of existentially quantified formulae, which in particular entails that PresPred^C is complete for the class of *essentially uninterpreted formulae* F (modulo linear integer arithmetic) with finite ground instantiation F^* defined in [7], and thus also for the array property fragment [3] (it is not a decision procedure for those fragments, however).

6 Experiments and Related Work

We have implemented the described calculus PresPredHR^C for FOL(LIA) in the theorem prover PRINCESS¹, and are in the process of adding further optimisations. PRINCESS uses the relational encoding from Sect. 4 to eagerly encode functions, and heuristics similar to the ones in Simplify [5] to automatically identify triggers in quantified formulae. PRINCESS is able to handle all of the examples discussed in Sect. 1.1.

In order to evaluate the overhead of handling functions using the relational encoding, we compared the performance of PRINCESS with the SMT solvers

¹ <http://www.philipp.ruemmer.org/princess.shtml>

	AUFLIA+p (193)	AUFLIA-p (193)
Z3	191	191
Princess	145	137
CVC3	132	128

Fig. 3. Number of solved benchmarks, out of 2×193 unsatisfiable (scrambled) AUFLIA benchmarks selected in the SMT competition 2011. Experiments with PRINCESS were done on an Intel Core i5 2-core machine with 3.2GHz, with a timeout of 1200s, heap-space limited to 4Gb. The benchmarks in AUFLIA+p contain hand-written triggers for most of the quantified formulae, while all triggers have been removed in AUFLIA-p. The corresponding figures for Z3 and CVC3 are the results obtained during the SMT competition 2011 (<http://www.smtexec.org/exec/?jobs=856>).

CVC3 [1] and Z3 [16], using benchmarks selected in the SMT competition 2011. Since our work concentrates on the construction of proofs, we only considered unsatisfiable benchmarks, removing 13 satisfiable AUFLIA problems in each category. The results show that PRINCESS, while currently not being able to compete with the fastest SMT solver Z3, performs better than the (state-of-the-art) e-matching-based CVC3. This is a promising result, since PRINCESS does, at the moment, not use SMT techniques like lemma learning and back-jumping, which are important for large or propositionally complex problems. PRINCESS can solve most benchmarks using e-matching alone, but uses free variables in 17 of the (solved) benchmarks, typically in smaller (but harder) instances.

Related Work E-matching is today used in most SMT solvers, based on techniques that go back to the Simplify prover [5] and The Stanford Pascal Verifier [17]; since then, various refinements of the e-matching approach have been published, for instance [6, 15]. To the best of our knowledge, e-matching has not previously been combined with free variable methods. An instantiation method similar to e-matching, but with much stronger completeness results, has been published in [7] and is used in Z3; a comparison with our method, in terms of provability, is given in Sect. 5.1.

The model evolution calculus has recently been extended to theories, including integer arithmetic [2]; our approach resembles model evolution in that it also uses free variables in a tableaux setting, albeit in a more “rigid” (less branch-local) manner. Further differences are that $\mathcal{ME}(\text{LIA})$ works on clauses, only supports a restricted form of existential quantification, and has a more explicit representation of models. We are not aware of any $\mathcal{ME}(\text{LIA})$ implementations.

There has been much work on integrating theories into resolution: [21] works with constraints that are solved in a theory, but requires to enumerate the solutions of constraints (whereas it is enough to check the validity of constraints in our work). In [4], while it is enough to check constraint satisfiability, no uninterpreted functions or predicates are supported.

Vice versa, resolution has been integrated into SMT solvers [14] as a method to find instantiations of quantified formulae. This approach does not explicitly take theories into account.

References

1. Barrett, C., Tinelli, C.: CVC3. In: Proceedings, CAV. LNCS, vol. 4590, pp. 298–302. Springer (Jul 2007), berlin, Germany
2. Baumgartner, P., Fuchs, A., Tinelli, C.: ME(LIA) – Model Evolution With Linear Integer Arithmetic Constraints. In: LPAR. LNCS, vol. 5330. Springer (2008)
3. Bradley, A.R., Manna, Z., Sipma, H.B.: What’s decidable about arrays? In: VMCAI. LNCS, vol. 3855, pp. 427–442. Springer (2006)
4. Bürckert, H.J.: A resolution principle for clauses with constraints. In: Stickel, M.E. (ed.) CADE. LNCS, vol. 449. Springer (1990)
5. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. *Journal of the ACM* 52(3) (2005)
6. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: CADE. LNCS, vol. 4603. Springer (2007)
7. Ge, Y., de Moura, L.M.: Complete instantiation for quantified formulas in satisfiability modulo theories. In: CAV. pp. 306–320 (2009)
8. Halpern, J.Y.: Presburger arithmetic with unary predicates is Π_1^1 complete. *Journal of Symbolic Logic* 56 (1991)
9. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press (2009)
10. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholtz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st Verified Software Competition: Extended experience report (2011)
11. Manthey, R., Bry, F.: A hyperresolution-based proof procedure and its implementation in Prolog. In: GWAI. pp. 221–230. Springer (1987)
12. Manthey, R., Bry, F.: SATCHMO: A theorem prover implemented in Prolog. In: Proceedings, CADE. pp. 415–434. LNCS, Springer (1988)
13. McCarthy, J.: Towards a mathematical science of computation. In: Popplewell, C.M. (ed.) Information Processing 1962: Proceedings IFIP Congress 62. pp. 21–28. North Holland, Amsterdam (1963)
14. de Moura, L., Bjørner, N.: Engineering DPLL(T) + saturation. In: Proceedings, IJCAR. LNCS, vol. 5195. Springer (2008)
15. de Moura, L.M., Bjørner, N.: Efficient e-matching for SMT solvers. In: Proceedings, CADE. pp. 183–198. LNCS, Springer (2007)
16. de Moura, L.M., Bjørner, N.: Z3: An efficient smt solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
17. Nelson, G.: Techniques for program verification. Tech. Rep. CSL-81-10, Xerox Palo Alto Research Center (1981)
18. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. *J. ACM* 27, 356–364 (April 1980)
19. Rümmer, P.: Calculi for Program Incorrectness and Arithmetic. Ph.D. thesis, University of Gothenburg (2008)
20. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: LPAR. LNCS, Springer (2008)
21. Stickel, M.E.: Automated deduction by theory resolution. *Journal of Automated Reasoning* 1(4), 333–355 (1985)

A Proof of Lemma 3

We use a lifting argument and first show that a ground version of the calculus is complete. In the ground version, the rules $\forall L$ and $\exists R$ are replaced with the following rules:

$$\frac{\Gamma, [x/c]\phi, \forall x.\phi \vdash \Delta \Downarrow C}{\Gamma, \forall x.\phi \vdash \Delta \Downarrow C} \forall L-G \qquad \frac{\Gamma \vdash [x/c]\phi, \exists x.\phi, \Delta \Downarrow C}{\Gamma \vdash \exists x.\phi, \Delta \Downarrow C} \exists R-G$$

where c is now allowed to be an arbitrary constant.

The completeness of this ground calculus can be shown using Hintikka-style model construction: assume that a proof is constructed in a fair manner, which means whenever a rule (other than PC and =R) is applicable to a formula on a proof branch, it is eventually applied. Moreover, the rules $\forall L-G$ and $\exists R-G$ are systematically applied to all quantified formulae, enumerating all constants of the vocabulary. This construction will either eventually make it possible to close all branches using PC/=R (with a valid constraint), or will produce a (potentially infinite) saturated branch that cannot be closed. In the latter case it is possible to construct a model of all formulae occurring on the branch (including the input formulae) with the help of well-founded induction.

Ground proofs can be lifted to proofs in the original calculus PredEq^C by replacing applications of $\forall L-G/\exists R-G$ with the original rules $\forall L/\exists R$. Given a ground proof with valid constraint C , this will turn C into a constraint C' that possibly also contains existential quantifiers. Because $\phi \Rightarrow \exists c.\phi$, the validity of C implies the validity of C' .

B Proof of Lemma 5

The proof can be conducted in different ways: either a direct Hintikka-style completeness proof can be given, which follows the same lines as the proof given for Lemma 3; alternatively, it is possible to show that PredEq^C -proofs can be transformed into proofs in the calculus PredEqHR^C by systematically replacing applications of $\forall L$ with $\forall L-M$. We will at this point follow the second way, which is more robust w.r.t. the later extension of the calculus to integer arithmetic.

Suppose that a PredEq^C -proof \mathcal{P} of a sequent $\vdash \phi \Downarrow C$ with valid constraint C has been found, and that \mathcal{Q} is a minimal sub-proof of \mathcal{P} starting with an application of $\forall L$ to instantiate a quantified formula with negative literals (i.e., a formula to which also $\forall L-M$ can be applied):

$$\frac{\begin{array}{c} \vdots \\ \Gamma, [\bar{x}/\bar{c}](\bigvee_{i=1}^n \neg p_i(\bar{t}_i) \vee \phi), \forall \bar{x}.(\bigvee_{i=1}^n \neg p_i(\bar{t}_i) \vee \phi) \vdash \Delta \Downarrow [\bar{x}/\bar{c}]C \end{array}}{\Gamma, \forall \bar{x}.(\bigvee_{i=1}^n \neg p_i(\bar{t}_i) \vee \phi) \vdash \Delta \Downarrow \exists \bar{x}.C} \forall L^*$$

\mathcal{Q}

We transform \mathcal{Q} by shifting applications of the rules $\forall L$, $\forall L$, and $\neg L$ that are concerned with decomposing the instantiated formula towards the leaves of the

proof, in such a way that every instantiation of the quantified formula eventually has the following shape:

$$\begin{array}{c}
\left\{ \frac{\overline{\Gamma, p_i(\bar{s}_i) \vdash p_i([\bar{x}/\bar{c}]\bar{t}_i), \Delta \Downarrow D_i} \text{ PC}}{\Gamma, p_i(\bar{s}_i), \overline{\neg p_i([\bar{x}/\bar{c}]\bar{t}_i) \vdash \Delta \Downarrow D_i} \neg\text{L}} \right\}_{i=1}^n \quad \frac{\mathcal{R}}{\Gamma, [\bar{x}/\bar{c}]\phi, \dots \vdash \Delta \Downarrow [\bar{x}/\bar{c}]C} \\
\frac{\Gamma, \{p_i(\bar{s}_i)\}_{i=1}^n, \bigvee_{i=1}^n \overline{\neg p_i([\bar{x}/\bar{c}]\bar{t}_i) \vee [\bar{x}/\bar{c}]\phi}, \dots \vdash \Delta \Downarrow \bigwedge_{i=1}^n D_i \wedge [\bar{x}/\bar{c}]C}{\Gamma, \{p_i(\bar{s}_i)\}_{i=1}^n, \overline{\forall \bar{x}. (\bigvee_{i=1}^n \neg p_i(\bar{t}_i) \vee \phi)} \vdash \Delta \Downarrow \exists \bar{x}. \bigwedge_{i=1}^n D_i \wedge C} \forall\text{L}^* \\
\vdots \\
\end{array} \tag{12}$$

where D_i represents the constraint $\bar{s}_i \doteq [\bar{x}/\bar{c}]\bar{t}_i$.

This transformation is justified through rewriting rules on PredEq^C -proofs that permute applications of the rules $\forall\text{L}$, $\forall\text{L}$, and $\neg\text{L}$ with trailing applications of other rules; e.g.:

$$\begin{array}{c}
\frac{\Gamma, [x/c]\phi, \forall x.\phi, [y/d]\psi \vdash \Delta \Downarrow [x/c][y/d]C}{\Gamma, [x/c]\phi, \forall x.\phi, \exists y.\psi \vdash \Delta \Downarrow [x/c]\forall y.C} \exists\text{L} \\
\frac{\Gamma, \forall x.\phi, \exists y.\psi \vdash \Delta \Downarrow \exists x.\forall y.C}{\Gamma, \forall x.\phi, \forall y.\psi \vdash \Delta \Downarrow \exists x.\forall y.C} \forall\text{L} \\
\rightsquigarrow \frac{\Gamma, [x/c]\phi, \forall x.\phi, [y/d]\psi \vdash \Delta \Downarrow [y/d][x/c]C}{\Gamma, \forall x.\phi, [y/d]\psi \vdash \Delta \Downarrow [y/d]\exists x.C} \forall\text{L} \\
\frac{\Gamma, \forall x.\phi, [y/d]\psi \vdash \Delta \Downarrow [y/d]\exists x.C}{\Gamma, \forall x.\phi, \exists y.\psi \vdash \Delta \Downarrow \forall y.\exists x.C} \exists\text{L}
\end{array}$$

For $\forall\text{L}$, rewriting is applied such that the size of the left sub-proof (in which the unification of the literals $\bigvee_{i=1}^n \neg p_i(\bar{t}_i)$ takes place) is reduced. We make use of the fact that propositional rules are allowed to preserve the formula to which they are applied, e.g.:

$$\begin{array}{c}
\frac{\frac{\overline{A}}{\Gamma, \phi \vdash \phi', \Delta \Downarrow A} \quad \frac{\overline{B}}{\Gamma, \phi \vdash \psi', \Delta \Downarrow B}}{\Gamma, \phi \vdash \overline{\phi' \wedge \psi'}, \Delta \Downarrow A \wedge B} \wedge\text{R} \quad \frac{\overline{C}}{\Gamma, \psi \vdash \phi' \wedge \psi', \Delta \Downarrow C} \forall\text{L} \\
\frac{\Gamma, \phi \vee \psi \vdash \overline{\phi' \wedge \psi'}, \Delta \Downarrow A \wedge B \wedge C}{\Gamma, \phi \vee \psi \vdash \overline{\phi' \wedge \psi'}, \Delta \Downarrow A \wedge B \wedge C} \forall\text{L} \\
\rightsquigarrow \frac{\frac{\overline{A}}{\Gamma, \phi \vdash \phi', \dots, \Delta \Downarrow A} \quad \frac{\overline{C}}{\Gamma, \psi \vdash \phi' \wedge \psi', \Delta \Downarrow C}}{\Gamma, \phi \vee \psi \vdash \overline{\phi' \wedge \psi'}, \Delta \Downarrow A \wedge C} \forall\text{L} \quad \vdots \quad \wedge\text{R} \\
\frac{\Gamma, \phi \vee \psi \vdash \overline{\phi' \wedge \psi'}, \Delta \Downarrow A \wedge C \wedge B \wedge C}{\Gamma, \phi \vee \psi \vdash \overline{\phi' \wedge \psi'}, \Delta \Downarrow A \wedge C \wedge B \wedge C} \wedge\text{R}
\end{array}$$

It can be observed that none of the transformations strengthen constraints, so that the overall constraint generated by the proof stays valid.

Once the instantiation of a quantified formula is in shape (12), it can be replaced with an application of the rule $\forall\text{L-M}$. The formula introduced by $\forall\text{L-M}$ has the shape

$$\text{simp}(\forall \bar{x}. (\bigvee_{i=1}^n \bar{s}_i \neq \bar{t}_i \vee \phi)) = \bar{u} \neq \bar{v} \vee \forall \bar{x}'. \sigma \phi$$

where σ is the accumulated substitution applied to the tail formula ϕ , and \bar{x}' is a (possibly empty) subset of variables from \bar{x} that could not be eliminated. The

resulting proof is:

$$\begin{array}{c}
\left\{ \frac{\Gamma \vdash u_j \doteq v_j, \Delta \Downarrow u_j \doteq v_j}{\Gamma, u_j \not\equiv v_j \vdash \Delta \Downarrow u_j \doteq v_j} \begin{array}{l} \text{=R} \\ \text{\neg L} \end{array} \right\}_{j=1}^m \frac{\mathcal{R}'}{\Gamma, [\bar{x}'/\bar{d}]\sigma\phi, \dots \vdash \Delta \Downarrow [\bar{x}'/\bar{d}]C'} \forall L^* \\
\frac{\Gamma, \bar{u} \not\equiv \bar{v} \vee \forall \bar{x}'. \sigma\phi, \dots \vdash \Delta \Downarrow \bar{u} \doteq \bar{v} \wedge \exists \bar{x}'. C'}{\Gamma, \bar{u} \not\equiv \bar{v} \vee \forall \bar{x}'. \sigma\phi, \dots \vdash \Delta \Downarrow \bar{u} \doteq \bar{v} \wedge \exists \bar{x}'. C'} \forall L^* \\
\vdots \\
\frac{\Gamma, \text{simp}(\forall \bar{x}. (\bigvee_{i=1}^n \bar{s}_i \not\equiv \bar{t}_i \vee \phi)), \dots \vdash \Delta \Downarrow \bar{u} \doteq \bar{v} \wedge \exists \bar{x}'. C'}{\Gamma, \{p_i(\bar{s}_i)\}_{i=1}^n, \forall \bar{x}. (\bigvee_{i=1}^n \neg p_i(\bar{t}_i) \vee \phi) \vdash \Delta \Downarrow \bar{u} \doteq \bar{v} \wedge \exists \bar{x}'. C'} \forall L\text{-M} \\
\vdots
\end{array}$$

Here, the sub-proof \mathcal{R}' can be derived from \mathcal{R} in (12) by substituting the constants \bar{c} with $[\bar{x}'/\bar{d}]\sigma\bar{x}$.

Repeating this transformation will eventually turn the PredEq^C -proof into a PredEqHR^C -proof.

C Proof of Lemma 12

As in the proof of Lem. 5, show that applications of $\forall L$ ($\exists R$) can systematically be replaced with applications of $\forall L\text{-M}$ ($\exists R\text{-M}$), gradually turning the PresPred^C -proof into a PresPredHR^C -proof.

D Proof of Lemma 13

We show that the instantiation with ground terms $(\bar{t}_i)_{i=1}^n$ can be simulated in PresPred^C . For this, we first consider a ground version of PresPred^C , similar to the one in Appendix A, where the rules $\forall L$ and $\exists R$ are replaced with the following rules:

$$\frac{\Gamma, [x/t]\phi, \forall x.\phi \vdash \Delta \Downarrow C}{\Gamma, \forall x.\phi \vdash \Delta \Downarrow C} \forall L\text{-G} \quad \frac{\Gamma \vdash [x/t]\phi, \exists x.\phi, \Delta \Downarrow C}{\Gamma \vdash \exists x.\phi, \Delta \Downarrow C} \exists R\text{-G}$$

where t is allowed to be an arbitrary function-free term.

We construct a proof of $\{Fun_f, Tot_f\}_{f \in F} \vdash (\exists \bar{x}.\phi)_{Rel}$ in the resulting ground calculus. Consider the set of terms $T = \{\bar{t}_i \mid i \in \{1, \dots, n\}\}$. We use the totality axioms $\{Tot_f\}_{f \in F}$ to recursively introduce constants c_s representing function terms $s = f(s_1, \dots, s_m)$ that occur in T , using a similar loop as in Alg. 1:

Algorithm 2: Flattening of ground terms

```

while  $T$  contains function terms do
  pick (sub)term  $s = f(s_1, \dots, s_m)$  in  $T$  s.t.  $s_1, \dots, s_m$  do not contain
  functions;
  create an instance  $f_p(s_1, \dots, s_m, c_s)$  of the axiom  $Tot_f$  by first
  applying  $\forall L\text{-G}$   $m$  times, using the terms  $s_1, \dots, s_m$ , followed by one
  application of  $\exists L$ ;
  substitute  $c_s$  for  $f(s_1, \dots, s_m)$  everywhere in  $T$ ;
end

```

This leads to a sequent $\{Fun_f, Tot_f\}_{f \in F}, Lits \vdash (\exists \bar{x}.\phi)_{Rel}$, where $Lits$ is a set of literals of the form $f_p(s_1, \dots, s_m, c_s)$. We can continue this proof through the following steps:

- Use the rule $\exists R-G$ to generate ground instances of $(\exists \bar{x}.\phi)_{Rel}$ corresponding to $\bigvee_{i=1}^n [\bar{x}/\bar{t}_i]\phi$ by instantiation with the constants c_s .
- Use the axioms $\{Fun_f\}_{f \in F}$, together with the rules $\forall L-G$ and $PU-Z$ to create Ackermann instances for all (pairs of) functions terms in the sequent.
- Due to the completeness of $PresPred^C$ for Presburger arithmetic [20], the proof can be extended and closed using propositional and arithmetic rules.

The resulting ground proof of $\{Fun_f, Tot_f\}_{f \in F} \vdash (\exists \bar{x}.\phi)_{Rel}$ can be lifted to a proof in the original calculus $PresPred^C$, like in Appendix A.

E Detailed Experimental Results for AUFLIA+p

Benchmarks are available on:

<http://www.smtexec.org/exec/divisionResults.php?jobs=856&division=AUFLIA%2Bp>

Benchmark	Time/s	#free var.	Benchmark	Time/s	#free var.
AdditiveMethods_Owne	23.3		javafe.filespace.Pkg	466.6	
AdvancedTypes_W..cto	3.8		javafe.filespace.Que	0.1	
Alloc_Alloc.GimmieOn	1.7		javafe.filespace.Res	0.6	
Alloc_Alloc.M4.T_not	3.7		javafe.filespace.Res	0.5	
AndNumbers_T..ctor-1	0.9		javafe.filespace.Res	1.1	
AssignToNonInvariant	3.8		javafe.filespace.Tre	0.9	
AssignToNonInvariant	1.3		javafe.filespace.Zip	0.6	
Bag2_Bag..ctor	0.3		javafe.parser.Lex.00	T/O	
BasicMethodology_Bas	T/O		javafe.parser.Lex.01	T/O	
BasicMethodology_C..	2.7		javafe.parser.Lex.02	1.9	
BasicMethodology_Sub	0.4		javafe.parser.ParseE	T/O	
bignum_quant	0.1		javafe.parser.ParseE	T/O	
fuzzmark2	T/O		javafe.parser.ParseS	T/O	
Branching_T.P_System	0.2		javafe.parser.ParseS	T/O	
burns12	0.1		javafe.parser.test.T	T/O	
burns13	5.4		javafe.parser.test.T	8.7	
burns4	T/O		javafe.parser.test.T	0.5	
burns9	0.1		javafe.PrintSpec.12	167.1	88
Call.Call..ctor	1.3		javafe.reader.Cached	0.3	
Call_DifferentFormal	0.0		javafe.reader.Standa	59.0	
Cast_Cast.NO_System.	204.2		javafe.SrcTool.010	0.7	
checkTypeDeclElem	T/O		javafe.tc.CheckCompi	T/O	
Chunker10_Chunker.Sp	1.3		javafe.tc.Env.645	1.7	
Chunker7_Chunker..cc	0.3		javafe.tc.EnvForLoca	1.9	
Chunker_Chunker.Next	T/O		javafe.tc.FieldDeclV	T/O	
DefaultLoopInv0.Test	1.8		javafe.tc.FlowInsens	T/O	
equiv7_M-infer_e-vc.	0.1		javafe.tc.FlowInsens	T/O	
ExactTypes_F..ctor	0.4		javafe.tc.MethodDecl	2.7	9
ExposeVersion_A.Fiel	556.7		javafe.tc.MethodDecl	33.1	39
Finally_ReturnFinall	1.0		javafe.tc.MethodDecl	T/O	
getNextPragma	T/O		javafe.tc.MethodDecl	12.1	
heapaware0_M-infer_e	0.2		javafe.tc.PreptypeDe	T/O	
Immutable_test3.B..c	30.1		javafe.tc.Types.011	2.3	
Immutable_test3.C..c	T/O		javafe.tc.Types.023	0.6	
javafe.ast.Ambiguous	1.5		javafe.tc.Types.043	T/O	
javafe.ast.ArrayRefE	14.4		javafe.tc.TypeSig.72	32.5	
javafe.ast.ArrayType	24.6		javafe.tc.TypeSig.72	2.2	
javafe.ast.ArrayType	28.9		javafe.tc.TypeSigVec	80.5	84
javafe.ast.BinaryExp	29.0		javafe.tc.TypeSigVec	T/O	
javafe.ast.BreakStmt	1.5		javafe.test.Print.76	0.5	
javafe.ast.BreakStmt	2.2		javafe.util.Buffered	0.5	
javafe.ast.CastExpr.	1.5		javafe.util.FatalErr	1.5	
javafe.ast.CastExpr.	1.5		javafe.util.FilterCo	0.6	
javafe.ast.CastExpr.	2.3		javafe.util.FilterCo	0.8	
javafe.ast.CatchClau	T/O		javafe.util.Location	T/O	
javafe.ast.CompoundN	3.6		javafe.util.Location	0.8	
javafe.ast.Construct	1.8		javafe.util.Location	T/O	
javafe.ast.ContinueS	1.5		javafe.util.Set.806	0.9	
javafe.ast.DefaultVi	0.6		javafe.util.Set.816	0.4	
javafe.ast.Delegatin	0.4		list1	0.0	
javafe.ast.DoStmt.00	4.5		list3	0.0	
javafe.ast.ExprVec.0	T/O		LocalExposeD..ctor	T/O	
javafe.ast.ExprVec.1	0.4		MustOverride.Ex4.C..	1.4	
javafe.ast.FieldAcce	3.3		Passification_Array2	0.1	
javafe.ast.FieldDecl	2.6		Passification_good1-	0.0	
javafe.ast.FormalPar	T/O		PeerFields_PeerField	T/O	
javafe.ast.ForStmt.1	2.4		piVC_098a89	5.1	
javafe.ast.GenericVa	0.5		piVC_577945	0.4	11
javafe.ast.Identifie	0.2		piVC_647bf6	5.1	
javafe.ast.Identifie	0.2		piVC_675f8a	5.0	65
javafe.ast.Identifie	T/O		piVC_67ff5c	0.3	11

javafe.ast.IfStmt.00	1.5		piVC.b124d6	0.0	
javafe.ast.IfStmt.18	2.3		piVC.bf055f	5.2	
javafe.ast.ImportDec	T/O		piVC.cdaaac	0.8	22
javafe.ast.InitBlock	9.5		PureCall_Cell.get_Va	0.8	
javafe.ast.LabelStmt	2.1		PureCall_Test.TestGo	18.6	
javafe.ast.LexicalPr	28.1	42	PureReceiverMightBeC	9.8	
javafe.ast.LexicalPr	T/O		quant0_Test.Main_Sys	0.7	
javafe.ast.LexicalPr	0.4		Quantifiers_S-noinfe	0.1	5
javafe.ast.LexicalPr	11.4		Quantifiers_T-noinfe	0.0	
javafe.ast.LiteralEx	3.3		Quantifiers_V2-noinf	0.0	
javafe.ast.LiteralEx	1.6		QuantifierVisibility	65.4	
javafe.ast.MethodDec	1.8		Recursion0_Recursion	1.1	
javafe.ast.ModifierP	3.0	6	ricart-agrawala13	0.2	
javafe.ast.ModifierP	T/O		ricart-agrawala5	11.8	
javafe.ast.ModifierP	T/O		SES.Atom..ctor_SES.A	T/O	
javafe.ast.PrettyPri	6.4		SES.Nary.JunctionFA_	T/O	
javafe.ast.StandardP	248.6	396	SES.Sx.Bool.System.B	536.8	440
javafe.ast.StmtVec.0	T/O		set10	0.0	
javafe.ast.TagConsta	0.3		set11	0.0	
javafe.ast.TypeDeclE	0.2		set13	0.0	
javafe.ast.TypeDeclE	T/O		set18	0.0	
javafe.ast.TypeDeclE	11.9		set1	T/O	
javafe.ast.TypeDeclE	T/O		set20	0.1	
javafe.ast.TypeDeclE	6.2		set3	T/O	
javafe.ast.TypeDeclV	8.7		set6	0.0	
javafe.ast.TypeDeclV	T/O		set7	0.0	
javafe.ast.TypeModif	T/O		set8	0.0	
javafe.ast.TypeNameV	T/O		Spouse_Person.NN	0.8	
javafe.ast.UnaryExpr	37.9		StrictReadOnly_C1.X1	1.2	
javafe.ast.VarInitVe	T/O		Strings_test3.MyStri	T/O	
javafe.ast.VarInitVe	T/O		textbook-DutchFlag.b	0.3	
javafe.ast.Visitor.0	0.5		Types.T.M-orderStren	3.9	
javafe.ast.VisitorAr	3.3		Types.Types.M0.B.not	2.1	
javafe.ast.VisitorAr	0.5		VisibilityBasedInvar	0.3	
javafe.filespace.Has	0.8		WhereClause_D..ctor	1.8	
javafe.filespace.Pat	T/O				

F Detailed Experimental Results for AUFLIA-p

Benchmarks are available on:

<http://www.smtexec.org/exec/divisionResults.php?jobs=856&division=AUFLIA-p>

Benchmark	Time/s	#free var.	Benchmark	Time/s	#free var.
AdditiveMethods_Owne	17.7		javafe.filespace.Slo	5.6	
AdditiveMethods_Owne	1.2		javafe.filespace.Str	0.6	
AddMethod_Bag.SpecSh	2.2		javafe.filespace.Uni	0.4	
AdvancedTypes_Q.cto	2.7		javafe.filespace.Uni	0.3	
Alloc_Alloc.M0.T	T/O		javafe.FrontEndTool.	0.1	
arr2	0.1		javafe.genericfile.U	0.1	
AssignToNonInvariant	3.1		javafe.genericfile.Z	1.3	
Assumptions_Assumpti	T/O		javafe.genericfile.Z	0.6	
BasicMethodology_Com	0.4		javafe.genericfile.Z	0.4	
BasicMethodology_Sub	T/O		javafe.parser.Lex.00	T/O	
bignum_quant	0.0		javafe.parser.Lex.01	T/O	
fuzzmark2	T/O		javafe.parser.ParseE	T/O	
burns13	5.4		javafe.parser.ParseE	T/O	
burns3	0.0		javafe.parser.ParseS	T/O	
Call_DifferentFormal	0.0		javafe.parser.ParseS	T/O	
Cast.Cast.NO.System.	276.9		javafe.parser.ParseT	23.6	
Change769_S.ctor.Sy	0.3		javafe.parser.test.T	T/O	
Change769_Test3.Foo	0.3		javafe.parser.test.T	7.8	
Change773_Test1.cto	1.1		javafe.reader.ASTClA	25.2	
checkTypeDeclElem	T/O		javafe.reader.Cached	0.6	
Chunker11-AdditiveEx	1.6		javafe.reader.Descri	T/O	
Chunker4_Chunker.ct	T/O		javafe.reader.SrcRea	4.4	
Chunker5_Chunker.ct	T/O		javafe.SrcTool.17	0.6	
Chunker_Chunker.cto	T/O		javafe.tc.CheckCompi	2.2	
Chunker_Chunker.Next	T/O		javafe.tc.ConstantEx	21.0	
equivpoly0_M-infer_e	0.1		javafe.tc.Env.010	2.4	
ExactTypes_W.ctor	19.2		javafe.tc.EnvForType	9.0	
ExposeVersion_S.cto	T/O		javafe.tc.FieldDeclV	T/O	
False_Test1-noinfer	0.0		javafe.tc.FieldDeclV	11.9	
FormulaTerm2_Q-noinf	0.0		javafe.tc.FlowInsens	T/O	
FormulaTerm_less-noi	0.0		javafe.tc.FlowInsens	T/O	
getNextPragma	T/O		javafe.tc.FlowInsens	99.1	
Immutable_test3.B.M	3.6		javafe.tc.MethodDecl	T/O	
javafe.ast.ArrayRefE	0.5		javafe.tc.PrepareTypeDe	T/O	
javafe.ast.ArrayType	89.9		javafe.tc.TypeCheck.	175.5	176
javafe.ast.ArrayType	28.9		javafe.tc.Types.749	1.5	
javafe.ast.ASTDecora	0.3		javafe.tc.TypeSig.00	88.2	
javafe.ast.BlockStnt	2.5		javafe.tc.TypeSigVec	T/O	
javafe.ast.BreakStnt	28.6		javafe.tc.TypeSigVec	T/O	
javafe.ast.CatchClau	1.6		javafe.tc.TypeSigVec	12.9	
javafe.ast.CatchClau	T/O		javafe.test.LocTool.	0.4	
javafe.ast.ClassLite	25.6		javafe.util.ErrorSet	37.0	
javafe.ast.ClassLite	3.6		javafe.util.FilterCo	1.3	
javafe.ast.Compilati	T/O		javafe.util.FilterCo	0.8	
javafe.ast.CompoundN	15.5		javafe.util.Location	0.1	
javafe.ast.CondExpr.	2.3		javafe.util.Set.808	0.3	
javafe.ast.Construct	2.4		list1	0.0	
javafe.ast.DefaultVi	2.1		list3	0.0	
javafe.ast.Delegatin	0.3		LocalExpose_T.SpecSh	9.9	
javafe.ast.DoStnt.13	2.2		Old_OldInCode0-noinf	0.1	
javafe.ast.ExprVec.0	T/O		Pack_Bad2.Cell.ctor	1.8	
javafe.ast.FieldDecl	14.5		Pack_Bad2.Cell.SpecS	1.3	
javafe.ast.FormalPar	T/O		Pack_Good2.Cell.ctct	0.3	
javafe.ast.Identifie	0.8		Passification_Unreac	0.0	
javafe.ast.Identifie	T/O		piVC_098a89	5.1	
javafe.ast.ImportDec	6.6		piVC_0f7c6a	4.7	65
javafe.ast.ImportDec	T/O		piVC_3937a0	0.0	
javafe.ast.InitBlock	3.0		piVC_600bf6	0.0	
javafe.ast.Instance0	T/O		piVC_675f8a	5.1	
javafe.ast.Instance0	T/O		piVC_67ff5c	0.3	11
javafe.ast.LexicalPr	T/O		piVC_8894c1	0.0	

javafe.ast.LexicalPr	T/O
javafe.ast.ModifierP	T/O
javafe.ast.OnDemandI	1.7
javafe.ast.SimpleNam	3.0
javafe.ast.StandardP	129.4
javafe.ast.StmtVec.0	T/O
javafe.ast.StmtVec.3	T/O
javafe.ast.SuperObjc	14.0
javafe.ast.SwitchLab	31.3
javafe.ast.SwitchStm	2.6
javafe.ast.SwitchStm	2.5
javafe.ast.SwitchStm	2.4
javafe.ast.TypeDeclE	24.5
javafe.ast.TypeDeclE	T/O
javafe.ast.TypeDeclE	T/O
javafe.ast.TypeDeclV	T/O
javafe.ast.TypeDeclV	0.7
javafe.ast.TypeDeclV	11.9
javafe.ast.TypeModif	T/O
javafe.ast.TypeModif	T/O
javafe.ast.TypeModif	11.3
javafe.ast.TypeNameV	T/O
javafe.ast.TypeNameV	T/O
javafe.ast.TypeNameV	11.9
javafe.ast.TypeObjec	2.4
javafe.ast.VarDeclSt	3.4
javafe.ast.VarInitVe	T/O
javafe.ast.Visitor.0	0.5
javafe.ast.Visitor.0	0.5
javafe.ast.VisitorAr	0.5
javafe.ast.VisitorAr	0.5
javafe.ast.VisitorAr	0.5
javafe.ast.WhileStmt	1.6
javafe.CopyLoaded.1	0.9
javafe.CopyLoaded.5	T/O
javafe.filespace.Pat	0.3
javafe.filespace.Res	1.1

piVC_8a5f8d	0.3
piVC_a04477	5.3
piVC_b124d6	0.0
piVC_b5f37c	5.1
piVC_bd0142	T/O
PureAxio.GetterClie	T/O
Quantifiers_S-noinf	0.1
Quantifiers_V0-noinf	0.0
QuantifierVisibility	T/O
ricart-agrawala4	6.9
ricart-agrawala6	0.2
ricart-agrawala9	0.2
SES.Sx.Bool.System.B	T/O
SES.Sx.get.IsTrue	T/O
set10	0.0
set12	0.0
set15	0.0
set18	0.0
set1	T/O
set20	0.0
set3	T/O
set4	0.0
set6	0.0
set7	0.0
StrictReadOnly_AlsoI	7.8
StrictReadOnly_C1.P	0.4
StrictReadOnly_C2.M	0.4
Strings_test3.MyStri	0.4
textbook-Find.bpl.2.	0.0
tohtml.DeclLinks.016	0.5
Types.MoreTypes..cto	2.2
Types.T.M-orderStren	0.4
Types.Types.NO.C.D-o	T/O
Unsigned.test.Q.Syst	0.3
WhereMotivation.Type	7.1

11