A Proof Slicing Framework for Program Verification

Ton Chanh Le, Cristian Gherghina, Razvan Voicu, and Wei-Ngan Chin
Department of Computer Science, National University of Singapore

Abstract. In the context of program verification, we propose a *formal framework* for *proof slicing* that can aggressively reduce the size of proof obligations as a means of performance improvement. In particular, each large proof obligation may be broken down into smaller proofs, for which the overall processing cost can be greatly reduced, and be even more effective under *proof caching*. Our proposal is built on top of existing automatic provers, including the state-of-the-art prover Z3, and can also be viewed as a re-engineering effort in proof decomposition that attempts to avoid large-sized proofs for which these provers may be particularly inefficient. In our approach, we first develop a calculus that formalizes a *complete proof slicing* procedure, which is followed by the development of an *aggressive proof slicing* method. Retaining completeness is important, and thus in our experiments the complete method serves as a backup for the cases when the aggressive procedure fails. The foundations of the aggressive slicing procedure are based on a novel lightweight annotation scheme that captures *weak links* between sub-formulas of a proof obligation; the annotations can be inferred automatically in practice, and thus both methods are fully automated. We support our theoretical developments with experimental results, which show significant improvements in the verification of complex programs, where richer specifications are often captured via loosely connected static properties.

1 Introduction

A significant challenge in the area of program verification is posed by the ever increasing number and complexity of proof obligations that need to be discharged by automated theorem provers. To overcome this challenge, a number of previous investigations have considered the approach of "shrinking" the generated proof obligations as a means of speeding up the solvers. [13] splits the proof obligations based on control flow to get smaller proofs. [15,21,22] detect and discard information that is not relevant to the problem at hand, thus streamlining the proof process. When this streamlining is performed aggressively, the size of the resulting proof obligations may be greatly reduced, leading to opportunities for significant performance improvement. In this context, an important technique is that of *proof caching* [10], which reuses proof results when multiple instances of the same sub-formulas are encountered. While the idea of *proof slicing* is not new in the context of automatic theorem provers, we believe that the procedure is more effectively carried out in the larger scope of program verification. In this regards, we make new contributions in three key directions, namely (i) the development of a *formal foundation* for proof slicing mechanisms, (ii) a general application of proof slicing that is *prover-independent* and tailored to *program verification*, and (iii) an *annotation scheme* that allows a more aggressive application of the mechanism, leading to improved performance.

A formal foundation in proof slicing is important for providing an avenue towards a more rigorous investigation into the field. To that end, we first develop a *complete* calculus for automatic slicing, which serves as a foundation for the implementation of our tool. Importantly, apart from completeness, this calculus also enjoys properties of convergence and completeness, which are crucial for its trustworthiness, and its potential for efficient implementation.

One important application area is that of program verification, whereby a typical approach is to employ a program verifier that processes the code of interest, annotated with pre/post-conditions, in order to produce a set of proof obligations that are subsequently passed on to off-the-shelf theorem prover. These proof obligations are fundamentally of the form $P \Longrightarrow Q$, whereby each P is an antecedent that captures some current program state, while Q is a goal (or assertion) that has to be proven. Since proof slicing remains complete only when the antecedent is satisfiable, and since satisfiability checks typically add a non-negligible overhead, existing state-of-the-art theorem provers, with formula reduction techniques such as relevancy propagation [4], or labelled splitting [8], do not employ this mechanism. However, with our slicing mechanism placed in-between the verifier and the theorem prover, we ensure that the satisfiability checks of antecedents are *incremental* and with low overhead, which is key to good performance.

As a further improvement, we designed an *annotation scheme* that captures *constraint linking properties*, that is, variable-sharing dependencies between interpreted atoms (*i.e.*, constraints) of a proof obligation; this scheme enables an *aggressive slicing* procedure. We believe that such an approach allows proof slicing to be viewed as a modular and extensible mechanism, rather than as a black box with limited functionality. This point is particularly poignant, as a good annotation scheme is also the basis for effective *annotation inference mechanisms*. These mechanisms can, in general, be completely automatic; several examples can be found in the experimental results section.

We summaries our research contributions, as follows:

- A formal and general framework for uniformly describing different proof slicing mechanisms (Sec. 3). We prove the proposed slicing mechanisms to be both sound and convergent, in the sense that, while non-deterministic, the framework always produces the same result for a given input. The immediate application of this framework is a *complete slicing* procedure (Sec. 4).
- An annotation scheme for slicing that is suitable for a variety of logics (Sec. 5). This is aimed at allowing parts of formulas to be identified as carrying information *linking* distinct properties. Then, an *aggressive proof slicing* mechanism can leverages on annotation schemes to obtain further reductions of the proof slices (Sec. 6). This also creates the opportunity for applying proof caching, which is particularly effective with smaller-sized proofs.
- An implementation of the both proof slicing mechanisms within an existing automated program verification system (Sec. 7). Our experiments show compelling performance gain of about 61% for complete proof slicing, and a further gain of 74% for aggressive proof slicing (see Fig. 7).

2 Proof Slicing for Program Verification

Depending on the context, we shall use the term "slicing" to denote either formula slicing or proof slicing. Formula slicing is the partitioning of a formula into "slices" – sub-formulas that group together related constraints. Two slices are said to be *disjoint* if they do not share any common variables, otherwise they are said to be *overlapping*. Proof slicing is the partitioning of a proof obligation into smaller sub-proofs to reduce the proof's complexity, thus improving performance of discharging proofs.

In the context of program verification, there are typically two major kinds of proof obligations, namely: (i) *Entailment checking*, of the form $P \vdash Q$ and (ii) *Unsatisfiability checking*, of the form $\mathit{UNSAT}(P)$ or $P \vdash \texttt{false}$. For unsatisfiability checking, the proof slicing mechanism partitions the initial formula P into a set of disjoint slices $\{P_1, \ldots, P_n\}$ whereby $P \leftrightarrow P_1 \wedge \cdots \wedge P_n$, and then incrementally applies unsatisfiability checks on some of these slices, *i.e.*, the slices that have been recently modified since the last unsatisfiability checks.

For entailment checking, proof slicing is the division of an initial, large entailment formula into smaller ones, obtained by slicing the original formula's antecedent with respect to each of its consequent. Given an antecedent P and a conjunctive consequent $Q_1 \wedge \cdots \wedge Q_n$, we partition P into possibly overlapping slices $\{P_1, \ldots, P_n\}$ such that each slice P_i is sufficient to prove the corresponding consequent Q_i. That is, the original entailment is replaced by a set of smaller entailments $\{P_i \vdash Q_i\}_{i=1}^{n}$. Importantly, this slicing step assumes that the sequent's antecedent is satisfiable, *i.e.*, it has been subjected to a prior unsatisfiability check. Loss of completeness occurs when weakening an unsatisfiable antecedent into a satisfiable one, and is the main reason for the limited adoption of this optimization in mainstream theorem provers.

Let consider the implication checks of the form $P_1 \wedge \cdots \wedge P_n \implies Q_1 \wedge \cdots \wedge Q_m$. Without proof slicing, a theorem prover needs to prove the unsatisfiability of $P_1 \wedge \cdots \wedge P_n \wedge (\neg Q_1 \vee \cdots \vee \neg Q_m)$. Due to the possibility of $P_1 \wedge \cdots \wedge P_n$ being unsatisfiable, the prover could not drop any constraint of the antecedents, unless it is willing to risk a loss of precision. By explicitly distinguishing between two kinds of proof obligations, our framework can avoid this problem by a prior unsatisfiable checking of the antecedents. Moreover, this distinction also allows us to exploit more aggressive pruning of irrelevant constraints from the antecedents with a novel annotation scheme (see Sec. 5).

Let us demonstrate how proof slicing can be applied to help with verifying the code snippet in Fig. 1(a). The pre- and post-conditions are provided by the `assume` and `assert` statements, respectively. To prove the total correctness of this program, we use the loop invariant $\texttt{x}{=}\texttt{2y} \wedge \texttt{n}{\geq}\texttt{0}$ for partial correctness proof, and the variant `n` as a well-founded measure for termination proof. The set of generated verification conditions are shown in Fig. 1(b). Observe that in these verification conditions, the constraints of `x` and `y` and the constraints of `n` are disjoint. As a result, they can be proven independently by the proof slicing mechanism, resulting in simpler proof obligations. For example, the verification condition $\texttt{VC}_4$ can be split into two separate entailments

$$\texttt{VC}_{\texttt{4a}} : \texttt{x}{=}\texttt{2y} \vdash \texttt{x}{+}\texttt{2}{=}\texttt{2(y}{+}\texttt{1)} \quad \texttt{VC}_{\texttt{4b}} : \texttt{n}{\geq}\texttt{0} \wedge \texttt{n}{>}\texttt{0} \wedge \texttt{n}{=}\texttt{N}_\texttt{0} \vdash \texttt{n}{-}\texttt{1}{\geq}\texttt{0} \wedge \texttt{n}{-}\texttt{1}{<}\texttt{N}_\texttt{0}$$

by partitioning the antecedent into two slices (i) $\texttt{x}{=}\texttt{2y}$ and (ii) $\texttt{n}{\geq}\texttt{0} \wedge \texttt{n}{>}\texttt{0} \wedge \texttt{n}{=}\texttt{N}_\texttt{0}$. Prior to the entailment checks, each new antecedent is subjected to a satisfiability check, if

```
1: assume(n ≥ 0);
2: x = 0; y = 0;
3: while (n > 0) {
4:    x = x + 2;
5:    y = y + 1;
6:    n = n − 1; }
7: assert(x = 2 ∗ y ∧ n = 0);
```

(a)

$$\mathtt{Inv(x,y,n)} \equiv \mathtt{x{=}2y} \wedge \mathtt{n{\geq}0}$$

$$\begin{array}{ll}
\mathtt{VC}_1: & \mathtt{x{=}0} \wedge \mathtt{y{=}0} \wedge \mathtt{n{\geq}0} \vdash \mathtt{Inv(0,0,n)} \\
\mathtt{VC}_2: & \mathtt{Inv(x,y,n)} \wedge \neg(\mathtt{n{>}0}) \vdash \mathtt{x{=}2y} \wedge \mathtt{n{=}0} \\
\mathtt{VC}_3: & \mathtt{Inv(x,y,n)} \wedge \mathtt{n{>}0} \vdash \mathtt{n{\geq}0} \\
\mathtt{VC}_4: & \mathtt{Inv(x,y,n)} \wedge \mathtt{n{>}0} \wedge \mathtt{n{=}N_0} \\
 & \quad \vdash \mathtt{Inv(x{+}2,y{+}1,n{-}1)} \wedge \mathtt{n{-}1{<}N_0}
\end{array}$$

(b)

Fig. 1. A code snippet and its verification conditions for total correctness proof

its slice has changed when compared to an earlier program point. We note that only formula slice (ii) has changed, with its invariant strengthened by the extra constraints $\mathtt{n{>}0} \wedge \mathtt{n{=}N_0}$. Thus, for $\mathtt{VC}_4$, we only need to check the satisfiability of the slice (ii), instead of the whole antecedent.

In summary, the division of proof obligations into two classes, of entailments and unsatisfiability checks, both of which benefit in performance from proof slicing, distinguishes our work from the techniques employed in current theorem provers. In entailment checks, the size of the antecedent can be greatly reduced when subjected to a prior unsatisfiability check. A similar mechanism is used for unsatisfiability checks, where only changed slices need be re-checked. Without this early analysis on the potential satisfiability of antecedents, current theorem provers would have to process much larger sets of constraints[1] when discharging proof obligations produced by a verification system.

3 A Framework for Proof Slicing

The starting point of our formalization is that of entailment or unsatisfiability obligations whose left hand side is an unquantified conjunction of constraints and uninterpreted predicates. For reasons of simplicity, we shall confine our presentation to unquantified formulas; the system is, nevertheless, capable of handling quantifiers. Informally, the slicing mechanism will preprocess the input by always floating outwards the constraints that appear under quantifiers but are independent of the corresponding quantified variables, and treat the remaining quantified constraints as atomic.

$$(\wedge N)\ \frac{X_{i_0}{=}X'_{j_0}}{\bigwedge_i X_i \vee \bigwedge_j X'_j \hookrightarrow X_{i_0} \wedge (\bigwedge_{i \neq i_0} X_i \vee \bigwedge_{j \neq j_0} X'_j)}$$

$$(\wedge R)\ \frac{P \vdash Q_1 \quad P \vdash Q_2}{P \vdash Q_1 \wedge Q_2} \qquad (\vee L)\ \frac{P_1 \vdash Q \quad P_2 \vdash Q}{P_1 \vee P_2 \vdash Q}$$

Consequently, we consider a first-order language with equality and interpreted function symbols. The atoms of the language are formed in the usual way, and denote *constraints*, *i.e.*, predicates that have a fixed interpretation with respect to an external automated reasoning tool. Sequents are denoted by $P \vdash Q$, where P and Q are formulas. Our slicing mechanism is specified by the rules in Fig. 2, and works by taking in a sequent, and outputting a set

[1] A theorem prover might group relevant constraints into classes, such as congruence classes in the theory of equality, or classes of different theories in the Nelson-Oppen theory combination, or more generally, classes of constraints which share some common symbols.

$$\lfloor\mathbf{SPLIT{-}E1}\rfloor\quad \frac{}{\textsc{Split}(\emptyset)=\emptyset}$$

$$\lfloor\mathbf{SPLIT{-}E2}\rfloor\quad \frac{\begin{array}{c}\textsc{Split}(P)=R \qquad P_1=\{Q\in R \mid\ \exists\beta\in Q.\textsc{SameSlice}(\alpha,\beta)\}\\ P_2=\{Q\in R\mid\neg\exists\beta\in Q.\textsc{SameSlice}(\alpha,\beta)\}\end{array}}{\textsc{Split}(\{\alpha\}\cup P)=P_2\cup\{\{\alpha\}\cup\bigcup_{X\in P_1}X\}}$$

$$\lfloor\mathbf{GETCTR{-}E1}\rfloor\quad \frac{}{\textsc{GetCtr}_0(Q,PS)=\emptyset}$$

$$\lfloor\mathbf{GETCTR{-}E2}\rfloor\quad \frac{\{S\in PS\mid\textsc{IsRelevant}(Q,S)\}=\emptyset}{\textsc{GetCtr}_n(Q,PS)=\emptyset}$$

$$\lfloor\mathbf{GETCTR{-}E3}\rfloor\quad \frac{\begin{array}{c}S_1=\{S\in PS\mid\ \textsc{IsRelevant}(Q,S)\}\\ R=\bigcup_{X\in S_1}X \qquad R'=\textsc{GetCtr}_{n-1}(R,PS\setminus S_1)\end{array}}{\textsc{GetCtr}_n(Q,PS)=R\cup R'}$$

$$\lfloor\mathbf{P{-}ENTAIL}\rfloor\quad \frac{\begin{array}{c}\textsc{Split}(\{P_i\}_{i=0}^{m})=PS\\ \textsc{GetCtr}_n(Q,PS)\Rightarrow Q\end{array}}{\bigwedge_{i=0}^{m}P_i\vdash Q}$$

$$\lfloor\mathbf{P{-}UNSAT}\rfloor\quad \frac{\begin{array}{c}\textsc{Split}(\{P_i\}_{i=0}^{m})=PS\\ \exists X{\in}PS\cdot\textsc{GetCtr}_n(X,PS)\Rightarrow\texttt{false}\end{array}}{\mathit{UNSAT}(\bigwedge_{i=0}^{m}P_i)}$$

Fig. 2. Framework for Proof Slicing Mechanisms

of sliced sequents that are meant to be discharged by off-the-shelf provers. However, the input sequent must first undergo a pre-processing stage with the beside rewrite rule $(\wedge N)$ and two structural rules $(\wedge R)$ and $(\vee L)$, which yields a set of sequents in a form where the effect of the slicing rules in Fig. 2 is maximized, while retaining completeness. The result of this decomposition is a set of sequents whose LHS is a conjunctive formula and RHS is either a disjunctive or atomic formula. However, to avoid increasing the number of sub-sequents when these rules are applied, that may lead to some performance loss, rule $(\wedge N)$ should take precedence over rules $(\wedge R)$ and $(\vee L)$, if applicable, and rule $(\wedge R)$ can be stopped early if the pair of conjunctive consequents in the RHS share the same set of variables.

We distinguish between two calculi: a *complete slicing* calculus, and an *aggressive slicing* calculus. Both calculi formalize mechanisms for partitioning the conjuncts of a sequent, yielding sets of smaller sequents whose discharge is sufficient for establishing the proof of the original sequent. The assumption here is that the total effort of proving the set of smaller sequents by means of external provers is, in general, lighter than the effort of proving the original sequent by the same means. In the optimal case, the application of slicing decomposes the entailment $P_1 \wedge \ldots \wedge P_n \models Q$ into several subformulas, of the form $\bigwedge_{P\in X_i} P \models Q$, such that the sets X_i satisfy three properties: (i) *inclusion*: $\forall i.X_i{\subseteq}\{P_1,\ldots,P_n\}$, (ii) *relevance*: all X_i constraints are relevant to Q, *i.e.*, $\forall R.R\in X_i \rightarrow \bigwedge_{P\in X_i\setminus\{R\}} P \not\models Q$ and (iii) *correlation*: for each pair of constraints $P, P'\in X_i$, there exists a chain $P=P_1,\ldots,P_k=P'$ such that every two consecutive constraints P_j, P_{j+1} are overlapping. Similarly, an unsatisfiability check for a formula $P_1\wedge\ldots\wedge P_n$ is sliced into several unsatisfiability checks for $\bigwedge_{P\in X_i} P$ such that X_i satisfies the inclusion and correlation properties.

Unfortunately, this formulation is not practical, as even establishing the relevance for a given slice is costly, let alone discovering the slices. Our proposal relies on a more syntactic formulation for the relevance and correlation properties, by using two meta-predicates, IsRelevant and SameSlice, as approximations of the relevance and correlation tests. The actual definitions dictate the slicing strategies each calculus uses. In the following sections, we expand more on their formulation and usage.

The complete and aggressive slicing calculi share the set of rules given in Fig. 2, which we shall call the *slicing framework* and differ in the definitions used for the two meta-predicates. Specifically, to obtain the *complete* (or *aggressive*) slicing calculus, we add the rules in Fig. 3 (or in Fig. 5, resp.) to the framework. We shall discuss the framework in the remainder of this section, and we shall devote Sec. 4 and 6 to each of the two calculi.

The conjunct partitioning procedure Split calculates PS, a set of slices, from a set of conjuncts. Each slice is either extended with a new conjunct or not, in accordance with the SameSlice meta-predicate. This meta-predicate's role is to establish if two conjuncts should be kept in the same slice or not. Intuitively, it works by checking how information is shared between its two arguments. The result of applying the Split relation to a formula P is a set of sets of constraints that represent the partitioning into *slices* of P. Each set of constraints can be interpreted as a formula that is formed by a conjunction of its constraints. Propertywise, we have:

$$\bigcup \textsc{Split}(P){=}P \wedge (\forall X,Y{\in}\textsc{Split}(P){\cdot}X{\neq}Y \rightarrow X{\cap}Y{=}\{\})$$

The formulation of [**split−e2**] allows for arbitrary slicing decisions from the picking of α. Nevertheless, the slicing mechanism needs to be *convergent*, that is, to yield the same set of sliced sequents upon termination. Slicing convergence can be ensured by requiring the rewrite system formed by [**split**] to be confluent. In the following sections, we shall investigate convergence properties for the complete and aggressive slicing calculi.

Another operation of interest is the computation of relevant slices for a given formula from a set of slices. [**getctr−e3**] and [**getctr−e2**] describe a family GetCtr$_n$ of such functions that differ only in the exhaustiveness of the relevance computation. All start by picking the slices that are in the IsRelevant relation with the input formula Q. This step can be repeated using each of the previously selected slices as input for the next iteration. Such a refinement is important because, depending on the actual definition used for SameSlice, a single step might not be sufficient to gather all relevant constraints[2]. The default GetCtr function to use is GetCtr$_1$, but we can gradually increase its coverage through GetCtr$_2$, GetCtr$_3$, . . ., if needed. This family of operators satisfies the following two properties

(i) GetCtr$_n(Q, PS) \subseteq PS$ (ii) GetCtr$_n(Q, PS) \subseteq$ GetCtr$_{n+1}(Q, PS)$

Continuing on with the description of the slicing rules in Fig. 2, the rule [**p−unsat**] defines slicing for unsatisfiability obligations. The formula P is first partitioned, and then a search is performed for an unsatisfiable slice. Each slice is considered together

[2] Such is the case for the *aggressive slicing calculus* with an *annotation scheme* that will be introduced later.

with its relevant counterparts as computed by GETCTR$_n$. The ⇒ notation signifies the invocation of an external prover.

Similarly, [P−ENTAIL] defines the treatment of entailment obligations. The rule prescribes partitioning of the antecedent and the consequent, pairing consequent slices with relevant antecedent slices, and enforcing the implication relation on the resulting pairs. The [P−ENTAIL] rule corresponds to the conjunction introduction rules of Gentzen's sequent calculus [3]. Intuitively, a sequent with conjunctions on the right hand side can be split into separate sequents, each retaining one conjunct. Similarly, sequents with conjunctions on the left hand side can have any number (desirably, all but one) of conjuncts discarded. We state the lemma for soundness as follows

Lemma 1 (Soundness). *All sequents proven using the rules of the slicing framework are true.*

Proof Sketch: Rule [P−UNSAT] is a syntactic conversion of a unsatisfiability obligation into an implication obligation. Rule [P−ENTAIL] is an instance of conjunction introduction rule of the sequent calculus [3]. Thus, every proof of the slicing framework is a proof of the sequent calculus, and consequently, the slicing framework rules are sound. □

4 Complete Proof Slicing

In this section we introduce a completely automatic slicing mechanism. This mechanism uses the slicing framework rules given in Fig. 2, together with the meta-predicates SAMESLICE and ISRELEVANT given in Fig. 3. Essentially, this mechanism produces slices whose sets of free variables are disjoint. This is based on the idea that if a hypothesis and the conclusion of a proof obligation have disjoint sets of free variables, then the hypothesis cannot be directly contributing to the proof of the conclusion, and can thus be discarded.

[CS−CORRELATION]

$\text{SAMESLICE}(P_1, P_2) = \mathcal{V}(P_1) \cap \mathcal{V}(P_2) \neq \emptyset$

[CS−RELEVANCE]

$\text{ISRELEVANT}(Q, P) = \mathcal{V}(Q) \cap \mathcal{V}(P) \neq \emptyset$

Fig. 3. Complete Slicing Mechanism

Whenever two conjuncts of the hypothesis share free variables, we say that they are *correlated*, and under the current slicing scheme, they should belong to the same slice. This is reflected in the rule [CS−CORRELATION], where the meta-predicate SAMESLICE is defined to keep two conjuncts together if their sets of free variables are correlated. Here, the symbol $\mathcal{V}$ denotes a function that returns the set of free variables from its input.

Similarly, if a conjunct in the hypothesis shares variables with the consequent, we say that the conjunct is *relevant* to proving the conclusion. The definition of the meta-predicate ISRELEVANT given in the rule [CS−RELEVANCE] captures precisely this idea. We have taken the approach of utilizing these two rules to make our proof slicing framework more general. In the next section, we shall define a new variant of our proof slicing framework with annotation guidance, by simply redefining these two rules, without having to change any of the rules in Fig. 2.

In the previous section, we mentioned that [SPLIT] rules are expected to be convergent. This can be ensured by the convergence of our calculi. The following lemma substantiates this claim.

Lemma 2. [SPLIT] *with* [CS−CORRELATION] *is confluent.*

Proof: Firstly, due to the set intersection operator being symmetric, the [CS−CORRELATION] relation is symmetric as well. Secondly, note that the [SPLIT] rule considers every constraint in the initial constraint set. The only possibility for the outcomes to be different is if the order is important. However due to the symmetry of the [CS−CORRELATION] and the fact that $P_1 \cup P_2$ covers all the elements in the partially constructed slicing R, the partitioning ensures that all previously considered constraints that are in the [CS−CORRELATION] relation with the current constraint will be part of the same slice. □

An important property of the complete slicing mechanism is that it does not alter the level of completeness of the underlying solver. The slicing mechanism converts provable sequents into new sequents that are still provable in the same logic, provided that the antecedent of the sequent at hand is satisfiable. To formalize this claim, we assume that the underlying prover is formalized as a calculus LK^T, obtained from Gentzen's calculus LK [3], augmented with a theory T capable of handling the interpreted symbols of the language. Moreover, we assume that the axioms of T do not discharge sequents of the form $P \vdash Q$ when $\mathcal{V}(P) \cap \mathcal{V}(Q) = \emptyset$.

Lemma 3 (Relative completeness). *Let $P' \vdash Q$ be the sequent obtained by applying the complete slicing rules to the sequent $P \vdash Q$, where Q is atomic. Let LK^T be a sequent calculus obtained from LK by augmenting it with rules from a theory T that can handle the interpreted symbols of our formulas. If $P \vdash Q$ is provable, and P is satisfiable in LK^T, then $P' \vdash Q$, is also provable in LK^T.*

Proof: The slicing mechanism will first convert P into the conjunction $P' \wedge P''$, where $\mathcal{V}(P'') \cap \mathcal{V}(Q) = \emptyset$. It can then be decided that P'' can be discarded, and $P' \vdash Q$ is retained as a viable proof obligation. At this point, we have to make use of the statement that a sequent $R_1 \wedge R_2 \vdash R$ can be reduced to $R_1 \vdash R$ if $\mathcal{V}(R_2) \cap \mathcal{V}(R) = \emptyset$, and $R_1 \wedge R_2$ is satisfiable. This statement can be proved by structural induction on the proof tree of $R_1 \wedge R_2 \vdash R$. Based on this statement, repeated eliminations of irrelevant hypotheses would not change the LK^T provability of $P' \vdash Q$, which establishes the original claim. □

5 An Annotation Scheme for Proof Slicing

The complete proof slicing mechanism is particularly effective in the case of formulas that can be neatly partitioned into disjoint slices. It is, however, not as effective in the presence of constraints that seemingly link together sub-formulas that would otherwise be disjoint; for such cases, slicing needs to be applied more aggressively. To highlight this need, let us now consider a more expressive logic, capable of specifying and verifying heap-manipulating programs, with the possibility of generating more complex proof obligations. Consider the following definitions of a binary tree node and an inductive predicate that specifies an AVL tree rooted at its first argument and height-balanced.

$$\boxed{\begin{array}{l}
\mathbf{data}\ \mathtt{node}\ \{\ \mathtt{int\ val};\ \mathtt{node\ left};\ \mathtt{node\ right};\ \}\\
\mathtt{avl}(\mathtt{root},\mathtt{n},\mathtt{h},\mathtt{B}) \equiv \mathtt{root}{=}\mathtt{null}\wedge\mathtt{n}{=}0\wedge\mathtt{h}{=}0\wedge\mathtt{B}{=}\{\}\\
\ \vee\ \exists \mathtt{v},\mathtt{p},\mathtt{q},\mathtt{n}_1,\mathtt{n}_2,\mathtt{h}_1,\mathtt{h}_2\cdot\mathtt{root}{\mapsto}\mathtt{node}(\mathtt{v},\mathtt{p},\mathtt{q})\\
\quad *\ \mathtt{avl}(\mathtt{p},\mathtt{n}_1,\mathtt{h}_1,\mathtt{B}_1) * \mathtt{avl}(\mathtt{q},\mathtt{n}_2,\mathtt{h}_2,\mathtt{B}_2)\\
\quad \wedge\ \mathtt{n}{=}1{+}\mathtt{n}_1{+}\mathtt{n}_2\wedge\mathtt{h}{=}1{+}\mathtt{max}(\mathtt{h}_1,\mathtt{h}_2)\wedge{-1}{\leq}\mathtt{h}_1{-}\mathtt{h}_2{\leq}1\\
\quad \wedge\ \mathtt{B}{=}\{\mathtt{v}\}{\cup}\mathtt{B}_1{\cup}\mathtt{B}_2\wedge(\forall\mathtt{a}{\in}\mathtt{B}_1{\cdot}\mathtt{a}{<}\mathtt{v})\wedge(\forall\mathtt{b}{\in}\mathtt{B}_2{\cdot}\mathtt{v}{\leq}\mathtt{b})\\
\ \mathbf{inv}\ \mathtt{n}{\geq}0\wedge\mathtt{h}{\geq}0\wedge\mathtt{n}{\geq}\mathtt{h};
\end{array}}$$

This predicate captures four aspects of the AVL tree property. Parameter `root` is a pointer to the root of the tree, whereas $\mathtt{n}$, $\mathtt{h}$, and $\mathtt{B}$ (and their subscripted variants) capture, respectively, numbers of nodes in trees, their heights, and their sets of values. The constraint $-1{\leq}\mathtt{h}_1{-}\mathtt{h}_2{\leq}1$ states that the tree is nearly height-balanced, whereas the quantified set constraint $(\forall\mathtt{a}{\in}\mathtt{B}_1{\cdot}\mathtt{a}{<}\mathtt{v})\wedge(\forall\mathtt{b}{\in}\mathtt{B}_2{\cdot}\mathtt{v}{\leq}\mathtt{b})$ enforces the binary search tree property. The formula specified after the **inv** keyword denotes the invariant property that holds for all instances of the predicate. Moreover, the *separating conjunction* operator $*$ (cf. [18]) is used to concisely capture the memory disjointness property.

To prove an invariant of the AVL predicate (*e.g.*, $\mathtt{n}{\geq}0$), the entailment proof (*e.g.*, $\mathtt{avl}(\mathtt{x},\mathtt{n},\mathtt{h},\mathtt{B})\vdash\mathtt{n}{\geq}0$, resp.) can be discharged inductively by applying the definition of the predicate `avl`. For example, the below LHS is the resulting proof obligations (after each points-to $\mapsto$ is approximated by a non-null constraint, and each predicate is approximated by its invariant) while RHS is the same two entailments after applying *complete* proof slicing. For brevity, we use $\mathtt{n_i},\mathtt{h_i}{\geq}0$ to denote the conjunction $\mathtt{n_i}{\geq}0\wedge\mathtt{h_i}{\geq}0$.

$$\begin{array}{ll}
\quad\mathtt{x}{=}\mathtt{null}\wedge\mathtt{n}{=}0\wedge\mathtt{h}{=}0\wedge\mathtt{B}{=}\{\}\vdash\mathtt{n}{\geq}0 & \qquad\mathtt{n}{=}0\vdash\mathtt{n}{\geq}0\\
 & \\
\mathtt{x}{\neq}\mathtt{null}\wedge(\mathtt{n}_1,\mathtt{h}_1{\geq}0{\wedge}\mathtt{n}_1{\geq}\mathtt{h}_1){\wedge}(\mathtt{n}_2,\mathtt{h}_2{\geq}0{\wedge}\mathtt{n}_2{\geq}\mathtt{h}_2) & (\mathtt{n}_1,\mathtt{h}_1{\geq}0{\wedge}\mathtt{n}_1{\geq}\mathtt{h}_1){\wedge}(\mathtt{n}_2,\mathtt{h}_2{\geq}0{\wedge}\mathtt{n}_2{\geq}\mathtt{h}_2)\\
\wedge\,\mathtt{n}{=}1{+}\mathtt{n}_1{+}\mathtt{n}_2 & \wedge\,\mathtt{n}{=}1{+}\mathtt{n}_1{+}\mathtt{n}_2\\
\wedge\,\mathtt{h}{=}1{+}\mathtt{max}(\mathtt{h}_1,\mathtt{h}_2){\wedge}{-1}{\leq}\mathtt{h}_1{-}\mathtt{h}_2{\leq}1 & \wedge\,\mathtt{h}{=}1{+}\mathtt{max}(\mathtt{h}_1,\mathtt{h}_2){\wedge}{-1}{\leq}\mathtt{h}_1{-}\mathtt{h}_2{\leq}1\\
\wedge\,\mathtt{B}{=}\{\mathtt{v}\}{\cup}\mathtt{B}_1{\cup}\mathtt{B}_2{\wedge}(\forall\mathtt{a}{\in}\mathtt{B}_1{\cdot}\mathtt{a}{<}\mathtt{v}){\wedge}(\forall\mathtt{b}{\in}\mathtt{B}_2{\cdot}\mathtt{v}{\leq}\mathtt{b}) & \\
\vdash\mathtt{n}{\geq}0 & \vdash\mathtt{n}{\geq}0
\end{array}$$

Though sound, the second (sliced) entailment is unnecessarily verbose due to the presence of constraints $\mathtt{n}_1{\geq}\mathtt{h}_1$ and $\mathtt{n}_2{\geq}\mathtt{h}_2$ which act to link the constraints relating to size and height for the `avl` predicate. We refer to such constraints as *weakly linking* constraints, and propose to deploy a more aggressive proof slicing mechanism that can selectively disregard the relationship between variables occurring in such linkages. Though this decision may suffer from a risk of losing completeness, it would allow for a more aggressive application of the slicing mechanism. Applying this mechanism, we are able to obtain the following more compact entailment proof (*e.g.*, $\mathtt{n}_1{\geq}0\wedge\mathtt{n}_2{\geq}0\wedge\mathtt{n}{=}1{+}\mathtt{n}_1{+}\mathtt{n}_2\vdash\mathtt{n}{\geq}0$). To provide a systematic way to deal with weakly linking constraints, we propose the following annotation scheme.

Informal Definition 1 (Weakly Linking Constraint) *A constraint ϕ can be annotated as a* weakly linking *constraint $\phi\#$ if it is a weak constraint, such as inequality constraint (e.g., $\leq$ or $\neq$), that links together multiple variables from disjoint properties.*

In addition, for proving the invariant $\mathtt{n}{\geq}\mathtt{h}$ of the AVL predicate, our annotated proof slicing mechanism would keep the constraints related to both the size and the height properties and their weakly linking constraints, as follows:

$$\begin{array}{l}
\mathtt{n}_1,\mathtt{n}_2{\geq}0\wedge\mathtt{h}_1,\mathtt{h}_2{\geq}0\wedge(\mathtt{n}_1{\geq}\mathtt{h}_1)\#\wedge(\mathtt{n}_2{\geq}\mathtt{h}_2)\#\\
\wedge\,\mathtt{n}{=}1{+}\mathtt{n}_1{+}\mathtt{n}_2\wedge\mathtt{h}{=}1{+}\mathtt{max}(\mathtt{h}_1,\mathtt{h}_2)\wedge{-1}{\leq}\mathtt{h}_1{-}\mathtt{h}_2{\leq}1\vdash\mathtt{n}{\geq}\mathtt{h}
\end{array}$$

Aside from weakly linking constraints, we propose to support two additional kinds of weak linkages, namely:

Informal Definition 2 (Weakly Linking Variable) *A variable occurrence v can be annotated as a* weakly linking *variable $v\#$ if it does not belong to any particular property, but appears in the constraints of multiple distinct properties.*

Informal Definition 3 (Weakly Linking Expression) *An expression e can be annotated as a* weakly linking *expression $e\#$ if its definition has been captured by another variable, in a constraint such as $v{=}e$. This variable (or property) is only weakly linked with variables inside the linking expression.*

To showcase the need for *weakly linking variables*, we introduce an additional predicate that describes a binary tree with only positive elements. This new definition will capture two new properties: the set of elements $\mathtt{B}$ and the sum of the elements $\mathtt{s}$ in the tree. Such a predicate can be precisely defined as:

$$\begin{array}{l} \mathtt{btree}(\mathtt{root},\mathtt{B},\mathtt{s}) \equiv \ \ \mathtt{root}{=}\mathtt{null} \wedge \mathtt{s}{=}0 \wedge \mathtt{B}{=}\{\} \\ \ \vee\ \exists \mathtt{v},\mathtt{p},\mathtt{q},\mathtt{B}_1,\mathtt{B}_2,\mathtt{s}_1,\mathtt{s}_2 \cdot \mathtt{root}{\mapsto}\mathtt{node}(\mathtt{v},\mathtt{p},\mathtt{q}) \\ \qquad * \ \mathtt{btree}(\mathtt{p},\mathtt{B}_1,\mathtt{s}_1) * \mathtt{btree}(\mathtt{q},\mathtt{B}_2,\mathtt{s}_2) \wedge \mathtt{v}{>}0 \\ \qquad \wedge\ \mathtt{s}{=}\mathtt{s}_1{+}\mathtt{s}_2{+}\mathtt{v}\# \ \wedge \mathtt{B}{=}\mathtt{B}_1{\cup}\mathtt{B}_2{\cup}\{\mathtt{v}\#\} \\ \quad \mathbf{inv}\ \mathtt{s}{\geq}0 \wedge (\forall \mathtt{b}{\in}\mathtt{B}{\cdot}\mathtt{b}{\geq}0); \end{array}$$

With this definition, the two properties, although distinct, seem inseparable. None of the constraints can be truly considered weakly linking. A slicing algorithm that detects only weakly linking constraints will fail. This is due to a different kind of link between the two properties. In this example, the link is not established through a particular constraint but through the variable $\mathtt{v}$. Allowing linkage annotations to appear not only on constraints, but on individual variables as well, yields a refinement of the proof slicing algorithm, capable of better partitioning. For example, in proving:

$$\mathtt{btree}(\mathtt{x},\mathtt{B},\mathtt{s}) \wedge \mathtt{x}{\neq}\mathtt{null} \vdash \mathtt{s}{>}0,$$

the initial proof obligation will be:

$$\begin{array}{l} \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{v}{>}0 \wedge \mathtt{B}{=}\mathtt{B}_1{\cup}\mathtt{B}_2{\cup}\{\mathtt{v}\#\} \wedge (\forall \mathtt{a}{\in}\mathtt{B}_1{\cdot}\mathtt{a}{\geq}0) \\ \wedge\ (\forall \mathtt{b}{\in}\mathtt{B}_2{\cdot}\mathtt{b}{\geq}0) \wedge \mathtt{s}{=}\mathtt{s}_1{+}\mathtt{s}_2{+}\mathtt{v}\# \wedge \mathtt{s}_1,\mathtt{s}_2{\geq}0 \ \vdash\ \mathtt{s}{>}0 \end{array}$$

This obligation can be sliced by eliminating both the constraints on the bag of elements, as well as the constraints connected purely via weakly linking variables, thus obtaining a cleaner implication proof, devoid of set constraints, as follows:

$$\mathtt{v}{>}0 \wedge \mathtt{s}{=}\mathtt{s}_1{+}\mathtt{s}_2{+}\mathtt{v}\# \wedge \mathtt{s}_1,\mathtt{s}_2{\geq}0 \ \vdash\ \mathtt{s}{>}0$$

Furthermore, in proving:

$$\mathtt{btree}(\mathtt{x},\mathtt{B},\mathtt{s}) \vdash \forall \mathtt{c}{\in}\mathtt{B}{\cdot}\mathtt{c}{\geq}0,$$

we have the following initial proof obligation for the inductive case:

$$\begin{array}{l} \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{v}{>}0 \wedge \mathtt{B}{=}\mathtt{B}_1{\cup}\mathtt{B}_2{\cup}\{\mathtt{v}\#\} \wedge (\forall \mathtt{a}{\in}\mathtt{B}_1{\cdot}\mathtt{a}{\geq}0) \\ \wedge\ (\forall \mathtt{b}{\in}\mathtt{B}_2{\cdot}\mathtt{b}{\geq}0) \wedge \mathtt{s}{=}\mathtt{s}_1{+}\mathtt{s}_2{+}\mathtt{v}\# \wedge \mathtt{s}_1,\mathtt{s}_2{\geq}0 \ \vdash\ \forall \mathtt{c}{\in}\mathtt{B}{\cdot}\mathtt{c}{\geq}0 \end{array}$$

With the guidance obtained from weakly linking variables, our annotated proof slicing yields a more concise inductive proof:

$$\texttt{v}>0 \wedge \texttt{B}{=}\texttt{B}_1{\cup}\texttt{B}_2{\cup}\{\texttt{v}\#\} \wedge \forall \texttt{a}{\in}\texttt{B}_1{\cdot}\texttt{a}{\geq}0 \wedge \forall \texttt{b}{\in}\texttt{B}_2{\cdot}\texttt{b}{\geq}0 \vdash \forall \texttt{c}{\in}\texttt{B}{\cdot}\texttt{c}{\geq}0$$

Lastly, let us consider yet another scenario where *weakly linking expressions* are helpful. Consider a different AVL predicate that tracks the height and balance factor of its height-balanced sub-trees.

$$\begin{array}{l}
\texttt{avl}(\texttt{root},\texttt{h},\texttt{b}) \equiv \quad \texttt{root}{=}\texttt{null} \wedge \texttt{h}{=}0 \wedge \texttt{b}{=}0 \\
\quad \vee\ \exists \texttt{v},\texttt{p},\texttt{q},\texttt{b}_1,\texttt{b}_2,\texttt{h}_1,\texttt{h}_2 \cdot \texttt{root}{\mapsto}\texttt{node}(\texttt{v},\texttt{p},\texttt{q}) \\
\qquad * \texttt{avl}(\texttt{p},\texttt{h}_1,\texttt{b}_1) * \texttt{avl}(\texttt{q},\texttt{h}_2,\texttt{b}_2) \wedge \texttt{b}{=}(\texttt{h}_1{-}\texttt{h}_2)\# \\
\qquad \wedge\ \texttt{h}{=}1{+}\texttt{max}(\texttt{h}_1,\texttt{h}_2) \wedge -1{\leq}\texttt{b}{\leq}1 \\
\quad\ \textbf{inv}\ \texttt{h}{\geq}0 \wedge -1{\leq}\texttt{b}{\leq}1;
\end{array}$$

Here, the constraint $\texttt{b} = (\texttt{h}_1{-}\texttt{h}_2)\#$ bears a weak linkage between the balance factor and the height property. However, this type of weak linkage is different, in the sense that $\texttt{b}$ is related to constraints containing the expression $\texttt{h}_1{-}\texttt{h}_2$, but not to constraints containing the individual variables $\texttt{h}_1$ and $\texttt{h}_2$. In proving:

$$\texttt{avl}(\texttt{x},\texttt{h},\texttt{b}) \wedge \texttt{x}{\neq}\texttt{null} \wedge \texttt{b}{=}0 \vdash \texttt{h}_1{-}\texttt{h}_2{\neq}1$$

we would initially obtain the following proof obligation:

$$\begin{array}{l}
\texttt{x}{\neq}\texttt{null} \wedge \texttt{b}{=}0 \wedge \texttt{b}{=}(\texttt{h}_1{-}\texttt{h}_2)\# \wedge \texttt{h}_1,\texttt{h}_2{\geq}0 \wedge \\
\texttt{h}{=}1{+}\texttt{max}(\texttt{h}_1,\texttt{h}_2) \wedge -1{\leq}\texttt{b}{\leq}1 \quad \vdash \texttt{h}_1{-}\texttt{h}_2{\neq}1
\end{array}$$

We observe that the expression $\texttt{h}_1{-}\texttt{h}_2$ has already been captured by the balance factor property as the value of variable $\texttt{b}$. By tracking this information and exploiting it, we can deduce that the property relevant to our goal is the balance factor, rather than the constraints related to the height property. Applying annotated proof slicing, we can obtain a much simpler proof obligation:

$$\texttt{b}{=}0 \wedge \texttt{b}{=}(\texttt{h}_1{-}\texttt{h}_2)\# \wedge -1{\leq}\texttt{b}{\leq}1 \vdash \texttt{h}_1{-}\texttt{h}_2{\neq}1$$

We note here that each weakly linking annotation is added only once (mostly in predicate definitions and specifications), with the intent of being used across the entire program verification process.

In summary, the key points on the use of weakly linking annotations in support of more aggressive proof slicing are: (i) Proof obligations containing multiple weakly linked properties are commonly generated from richer specifications. (ii) The use of weakly linking annotations leads to loosely connected partitions that can be split when necessary, thus easily regaining the performance benefits of proof slicing. (iii) Multiple instances of the same (small) slice are frequently encountered in practice, which are shown in our experiments; thus, the use of proof caching would yield further performance gains.

Moreover, in a goal driven approach, it is possible to select only a small set of (loosely connected) partitions that have a higher chance of being relevant for the current proof obligation. Should this attempt fail, the algorithm can retry with a broader set of partitions, preserving the precision of the approach. Since failure rate is small in practice, this aggressive approach yields a significant improvement in efficiency. In our experiments, we have obtained multi-fold reductions in prover execution times.

6 Aggressive Proof Slicing

In this section, we propose a novel *annotation* mechanism, capable of pinpointing locations where proof slicing can be applied more aggressively.

6.1 Annotation Scheme

As mentioned in Sec. 3, the target of our framework is a first-order language with equality and interpreted function symbols. This language, more precisely described in Fig. 4, imposes no restrictions on the versatility of our framework. Without loss of generality we can safely assume that the annotations described in Sec. 5 will be transparently translated into annotations in our target language.

6.2 Annotation Reduction

$$
\begin{array}{lll}
\pi & ::= \alpha_{\mathcal{L}} \mid \neg\alpha_{\mathcal{L}} \mid \pi_1 {\wedge} \pi_2 & \\
\alpha_{\mathcal{L}} & ::= \alpha \mid (\alpha)\# \qquad v_{\mathcal{L}} \ ::= \ v \mid v\# & \\
\alpha & ::= \mathtt{true} \mid f_{\mathcal{L}}(v^*_{\mathcal{L}}) \mid v_{\mathcal{L}}{=}f_{\mathcal{L}}(v^*_{\mathcal{L}}) \mid v_{\mathcal{L}1}{=}v_{\mathcal{L}2} & \\
f_{\mathcal{L}}(v^*_{\mathcal{L}}) & ::= f(v^*_{\mathcal{L}}) \mid (f(v^*_{\mathcal{L}}))\# & \\
\textit{where} & \# \textit{ is the annotated slicing label}; & \\
 & \alpha \textit{ denotes atomic predicates}; & \\
 & \pi \textit{ denotes pure formulas}; \ v \textit{ is a variable}; & \\
 & v_{\mathcal{L}} \textit{ is a variable with or without \# label}; & \\
 & f_{\mathcal{L}} \textit{ is an interpreted symbol, possibly labeled}; &
\end{array}
$$

Fig. 4. Support Logic with Annotation Scheme

To simplify the formulation of our core calculus, we shall restrict our annotations for proof slicing to only weakly linking variables. Through a preprocessing step, we can transform each weakly linking constraint and each weakly linking expression into weakly linking variables, by transferring the weakly linking annotation to the free variables of a linking constraint or linking expression. Such a translation, named red, can be formalized as follows:

$$
\begin{array}{llll}
red_\beta(\pi_1 \wedge \pi_2) & \hookrightarrow red_\beta(\pi_1){\wedge}red_\beta(\pi_2) & red_\beta(f_{\mathcal{L}}(v^*_{\mathcal{L}})) & \hookrightarrow f_{\mathcal{L}}(red_\beta(v_{\mathcal{L}})^*) \\
red_\beta(\neg\alpha_{\mathcal{L}}) & \hookrightarrow \neg red_\beta(\alpha_{\mathcal{L}}) & red_\beta(v_{\mathcal{L}}{=}f_{\mathcal{L}}(v^*_{\mathcal{L}})) & \hookrightarrow red_\beta(v_{\mathcal{L}}){=}f_{\mathcal{L}}(red_\beta(v_{\mathcal{L}})^*) \\
red_\beta((\alpha)\#) & \hookrightarrow red_{\mathtt{true}}(\alpha) & red_\beta(v_{\mathcal{L}1}{=}v_{\mathcal{L}2}) & \hookrightarrow red_\beta(v_{\mathcal{L}1}){=}red_\beta(v_{\mathcal{L}2}) \\
red_\beta(\mathtt{true}) & \hookrightarrow \mathtt{true} & red_\beta(v\#) & \hookrightarrow v\# \\
red_\beta(f(v^*_{\mathcal{L}})) & \hookrightarrow f(red_\beta(v^*_{\mathcal{L}})) & red_{\mathtt{true}}(v) & \hookrightarrow v\# \\
red_\beta((f(v^*_{\mathcal{L}}))\#) & \hookrightarrow f(red_{\mathtt{true}}(v^*_{\mathcal{L}})) & red_{\mathtt{false}}(v) & \hookrightarrow v
\end{array}
$$

With this translation scheme, the free variable set of each constraint is divided into two disjoint sets, namely *weakly* and *strongly linking* variables. The set of *weakly linking* variables of a constraint can be computed by a simple function $\mathcal{V}_{\mathcal{W}}$ over the structure of the constraint α that picks up all (weakly) annotated variables, $\mathcal{V}_{\mathcal{W}}(v\#) = \{v\}$ while the set of *strongly linking* variables of a constraint α is its complement, namely $\mathcal{V}_{\mathcal{S}}(\alpha) = \mathcal{V}(\alpha) \setminus \mathcal{V}_{\mathcal{W}}(\alpha)$, where $\mathcal{V}(\alpha)$ returns the free variable set (without annotation) of the constraint α.

The translation scheme described above converts away all non-variable annotations. Nevertheless, a weakly linking constraint can still be distinguished from a constraint with weakly linking expressions or a constraint with a mix of weakly and strongly linking variables. At this point, we can make the following general observations: (i) a strongly linking constraint expresses knowledge specific to one property, and does not have any weakly linking variables; (ii) a weakly linking constraint encodes only weakly linking information, and thus has an empty set of strongly linking variables; (iii) constraints with weakly linking expressions or some weakly linking variables will express some relation between weakly linking entities and some other variables; thus neither set of weakly or strongly linking variables is empty. These observations allow us to support a uniform way of handling different kinds of linkages using a simpler variable-only annotation scheme.

6.3 Slicing Criterion

$$\lfloor\underline{\text{AS-CORRELATION}}\rfloor$$

$$\textsc{SameSlice}(P_1, P_2) = \begin{array}{l} \mathcal{V}_{\mathcal{W}}(P_1) = \mathcal{V}_{\mathcal{W}}(P_2) \wedge \\ \mathcal{V}_{\mathcal{S}}(P_1) \cap \mathcal{V}_{\mathcal{S}}(P_2) \neq \emptyset \end{array}$$

$$\lfloor\underline{\text{AS-RELEVANCE}}\rfloor$$

$$\textsc{IsRelevant}(Q, P) = \begin{array}{l} (\mathcal{V}(Q) \cap \mathcal{V}_{\mathcal{S}}(P) \neq \emptyset) \vee \\ (\mathcal{V}_{\mathcal{S}}(P) = \emptyset \wedge \mathcal{V}_{\mathcal{W}}(P) \subseteq \mathcal{V}(Q)) \end{array}$$

Fig. 5. Annotated Slicing Mechanism

To take advantage of weakly connected components, our aggressive slicing mechanism will create partitions (or slices) by ignoring links that are due to solely weakly linking variables. This is achieved by allowing two constraints to be in the same slice if they satisfy the following two conditions: (i) they share one or more strongly linking variables, and (ii) they have the same set of weakly linking variables. These two conditions are captured in a new definition for the SAMESLICE meta-predicate in Fig. 5. According to this definition, each weakly linking constraint will be kept as a separate slice. Furthermore, two constraints that share the same set of weakly linking variables will only be kept in the same slice if they share one or more strongly linking variables.

The following lemma establishes the convergence of our splitting procedure in the presence of the new meta-predicate.

Lemma 4. $\lfloor\underline{\text{SPLIT}}\rfloor$ *with* $\lfloor\underline{\text{AS-CORRELATION}}\rfloor$ *is convergent.*

Proof Sketch: Since $\lfloor\underline{\text{AS-CORRELATION}}\rfloor$ is a symmetric relation, we can make a similar argument to the one used for the convergence of complete slicing. □

6.4 Relevance Criterion

In the case of complete proof slicing, the constraints referring to a given property are spread across multiple slices. To have a good balance between precision and efficiency, we should ideally find the smallest set of hypotheses that ensure the success of the entailment check, whenever possible. To properly exploit the weakly linking annotations, we propose a two-step approach to finding relevant hypotheses. First, we employ aggressive slicing, which uses GETCTR$_2$, in order to obtain constraints that are most closely linked to the given goal. In case this first step fails, we may apply a subsequent exhaustive search step in order to identify additional constraints using a higher-level operator $\textsc{GetCtr}_n$, where n is the cardinality of our set of slices. Using n as a limit, our aggressive proof slicing mechanism has a similar behavior to that of complete proof slicing. We can formalize these two steps as instances of the slicing framework defined in Sec. 3.

Given a goal Q, the aggressive slicing mechanism would consider a slice *relevant* if either of the following holds:

1. It contains strongly linking variables that overlap with the free variables of Q.
2. It contains weakly linking constraints whose set of variables are entirely subsumed by the set of free variables of Q.

In order to collect these two categories of constraints, the calculus need only use $\textsc{GetCtr}_2$ in the aggressive search mechanism. The formalization of the aggressive search relevance check is given by [**AS−RELEVANCE**] in Fig. 5. The condition $\mathcal{V}_{\mathcal{S}}(P) = \emptyset$ in the meta-predicate IsRelevant indicates that P is a slice of a weakly linking constraint.

6.5 Annotation Inference

The current paper has investigated a foundation for proof slicing through a lightweight annotation scheme. The focus, thus far, for this paper has been on allowing the annotation scheme to support efficient and effective proof slicing process, so that our program verification system would have better scalability when it is made to work with existing state-of-the-art provers.

In this section, we shall briefly consider the possibility of automatically inferring annotations that could support aggressive proof slicing. We consider annotation inference mechanisms to be orthogonal to our current contributions, that could be more systematically investigated in the near future by us or others. To persuade on the feasibility of pursuing in this direction, we outline two lightweight annotation inference mechanisms that could support a significant degree of aggressive proof slicing automatically.

The first mechanism is based on the observation that constraints pertaining to different theories (e.g., linear arithmetic, arrays, and sets) are independent and should be kept separate. The rationale for this approach is that constraints of different theories will most likely require separate provers. Therefore, it is possible to separately identify variables that appear in constraints of multiple theories and annotate these variables as *weakly linking variables*. Such annotations allow the effective delimitation of theory specific proof slices. To illustrate this approach, let us take the formula:

$$\begin{array}{l} \mathtt{B}{=}\mathtt{B}_1{\cup}\mathtt{B}_2{\cup}\{\mathbf{v}\} \wedge (\forall \mathtt{a}{\in}\mathtt{B}_1{\cdot}\mathtt{a}{\geq}0) \wedge (\forall \mathtt{b}{\in}\mathtt{B}_2{\cdot}\mathtt{b}{\geq}0) \\ \wedge\ \mathtt{s}{=}\mathtt{s}_1{+}\mathtt{s}_2{+}\mathbf{v} \wedge \mathtt{s}_1, \mathtt{s}_2{\geq}0 \wedge \mathtt{v}{>}0 \end{array}$$

The inference would identify variable v as appearing in both arithmetic and set constraints. Consequently, its appearances in these constraints (in bold) are marked as weakly linking variables, thus allowing a clean split into set-based proofs and arithmetic-based proofs.

The second mechanism centers on the observation that, most often, inequality constraints exhibit weaker links as compared to equality constraints. If the sequent of interest has, for instance, multiple equality and inequality constraints, it is often preferable to create groups of correlated equality constraints, and mark inequality constraints with variables from two different groups as *weakly linking*.

$$\begin{array}{l}\mathtt{h}{=}1{+}\mathtt{max}(\mathtt{h}_1,\mathtt{h}_2) \wedge -1{\leq}\mathtt{h}_1{-}\mathtt{h}_2{\leq}1 \wedge \mathtt{h}_1,\mathtt{h}_2{\geq}0 \wedge \\ \mathbf{n_1{\geq}h_1} \wedge \mathbf{n_2{\geq}h_2} \wedge \mathtt{n}{=}1{+}\mathtt{n}_1{+}\mathtt{n}_2 \wedge\ \mathtt{n}_1,\mathtt{n}_2{\geq}0\end{array}$$

For example, given the previous formula, two sets of related constraints will be identified: one pertaining to $\mathtt{n}$, $\mathtt{n}_1$, $\mathtt{n}_2$ and one to $\mathtt{h}$, $\mathtt{h}_1$, $\mathtt{h}_2$. This partitioning in turn allows all the inequalities between $\mathtt{n}$'s and $\mathtt{h}$'s to be annotated as weakly linking.

However, the annotations on inequality constraints based on this heuristic might be redundant, since it is possible that the combination of two inequalities would form a stronger link of their variables. For instance, with the following formula:

$$\mathtt{x}{=}\mathtt{y} \wedge \mathbf{y{\leq}z} \wedge \mathbf{y{\geq}z} \wedge \mathtt{z}{=}\mathtt{t}$$

the inference mechanism marks the two constraints $\mathtt{y}{\leq}\mathtt{z}$ and $\mathtt{y}{\geq}\mathtt{z}$ as linking, ignoring the potential simplification into the equality $\mathtt{y}{=}\mathtt{z}$. Nevertheless, these extra annotations do not affect the completeness of the search process. For example, given the above annotated formula as premise for proving $\mathtt{x}{=}\mathtt{t}$, the weakly linking constraints that would be missed in the first phase (by GetCtr$_1$) can be collected during the second phase (by GetCtr$_2$) of the aggressive search mechanism. As a result, GetCtr$_2$ is enough to ensure the completeness of aggressive slicing for this example. In our experiments, we also mark each disequality of form $\mathtt{a}{\neq}\mathtt{b}$ as weakly linking.

7 Experiments

We have integrated the proposed proof slicing mechanisms into a separation logic-based program verification system [17], where proof obligations are soundly approximated by formulas in heap-free pure logic that can be discharged by off-the-shelf back-end theorem provers. The theorem provers used in our current evaluation are the Omega Calculator [20], MONA [11], Reduce/Redlog [7] and Z3 [5]. The proof slicing mechanisms are implemented as intermediate layers between the verifier and the theorem provers, effectively acting as prover-independent pre-processors for the back-end. In our measurements, we were careful to quantify the sole effect of applying the slicing procedures on the running time of the theorem provers (including overheads of the proof slicing mechanisms, if any) and show the relative comparison (on percentage) of timings by charts. The detailed timings (in seconds) and additional information are given in Appendix A. For brevity, we use NS, CS and AS to indicate no, complete or aggressive proof slicing mechanism, respectively.

We used several benchmarks for evaluating the resulting system. The first benchmark includes a set of heap-manipulating programs, implementing typical operations for singly and doubly linked lists, as well as more complex tree data structures such as AVL and Red-Black trees. The benchmark also includes the BigInt program, which

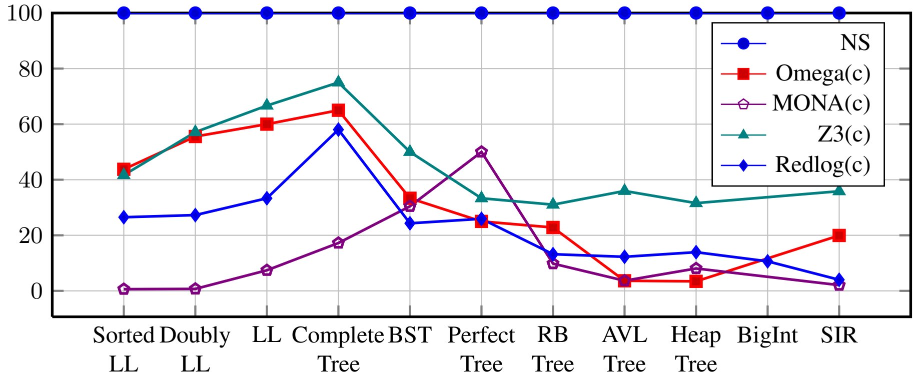

Fig. 6. Relative Comparison (%) of CS over NS with various theorem provers.

uses linked list to implement infinite precision integers and their arithmetic operations as well as the Karatsuba's fast multiplication method. The program is verified with non-linear constraints, which currently can only be handled by the Redlog prover. The second benchmark consists of programs taken from the SIR/Siemens test suite [6] with some data structures mentioned above and arrays.

Fig. 6 shows the comparison on percentage between the time spent on each underlying prover plus slicing overhead when CS is on (indicating by the prover name with the postfix (c)) and the time spent on the same prover without proof slicing mechanism (NS) for the first two benchmarks. [3] As can be seen, CS benefits all provers in general, especially on complex programs (*e.g.*, BigInt and SIR) with over 60% reduction. Moreover, on less scalable provers like Omega, MONA or Redlog, CS helps to reduce about 90% of the total prover time (or 10x faster). Those significant improvements come from the reduction on proof size for both unsatisfiability and entailment proofs by the effect of proof slicing. For Z3, the total reduction on the prover time is about 60% despite its own optimization mechanisms (*e.g.*, the relevancy propagation technique). Because our proof slicing mechanisms focus on the *higher level* tasks of checking entailments and detecting unsatisfiability, they are able to filter out irrelevant constraints more effectively whenever the relationships between constraints are preserved. Moreover, with proof slicing, the unsatisfiability checks on the antecedents of entailment proofs are performed incrementally and non-redundantly, thus bringing more performance gains.

The next set of experiments concerns annotated formulas, and the application of AS. The inductive predicates of data structures used in this benchmark are augmented with additional *linking constraints* that enhance their precision to move towards verification of full functional correctness but also greatly increase the complexity of the derived proof obligations. Annotations for those linking constraints are inferred automatically, via a number of heuristics. For example, each parameter of a heap predicate is regarded as an independent property, unless it is mutually-dependent on another parameter, leading to an approach where every constraint between two distinct properties is always marked as *weakly linking*. Fig. 7 illustrates the performance benefits of AS over CS in

[3] We did not pay attention to the verification overhead because it is almost constant across different provers with and without proof slicing.

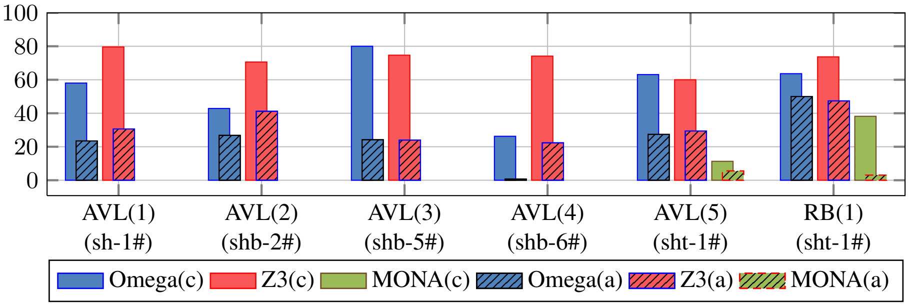

Fig. 7. Comparison of CS (c) and AS (a) over NS on examples with Weakly Linking Components (s: size, h: height, b: balance factor, t: sets, n#: number of (annotated) weakly linking components)

the relative comparison with NS. It shows that in the presence of more complex specifications, AS performs better than its complete counterpart. In these examples, proof obligations with set constraints are discharged by MONA.

The fourth benchmark, called *Spaguetti*, came from the SLP tool [16]. It includes a set of heap-based test cases; each of them comprises 1000 randomly-generated, parameterized by the number of heap variables, UNSAT checks of the form `F` ⊢ `false` with the success rate about 50%. The SLP tool is an optimized paramodulation prover, hardwired to support only the list segment predicate, together with equality and disequality constraints on heap addresses and thus yielding a very good performance (under 3 seconds for each Spaguetti test case). With the help of AS together with a simple heuristic that automatically marks each disequality as a weakly linking constraint, our general-purpose separation logic-based prover is expected to achieve comparable performance while allowing a much more expressive specification language.

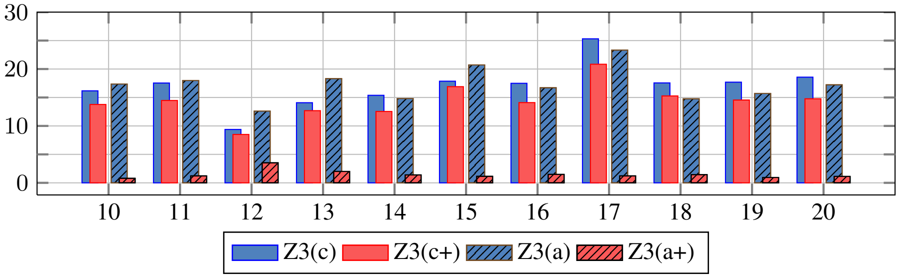

Fig. 8. Comparison (%) of CS and AS over NS on the Spaguetti Benchmark with the number of heap variables from 10 to 20 (+ indicates caching used)

Unfortunately, as shown in Fig. 8, while the use of CS helps reduce the prover times with Z3 (by about 76.2% in total), AS has only little extra effect due to high numbers of (smaller) proofs generated. To obtain further improvements, we have augmented our proof slicing framework with a simple *proof caching mechanism* that memoizes on string representations of normalized proof obligations. This brought about over 90% reduction (after including overheads of both caching and slicing) when AS is used; thus the performance is now comparable to the SPL tool. This outcome is supported by a much higher hit rate (over 99%) from caching of smaller proofs generated by AS, as compared to the hit rate from the combination of proof caching and CS. This effective result highlights the synergistic interplay between the proof caching and AS although the idea of proof caching is not new. Moreover, with the help of AS, an obsolete prover like Omega can catch up the performance of the advanced prover Z3 because the number of disequalities, which are expensively handled by Omega, is considerably reduced.

To investigate the portability of our proof slicing mechanisms, we have equipped AS for the Frama-C verification system [23]. For evaluation, we designed a family of contrived procedures, parameterized by the number of their parameters, that do computation on these independent variables, so as to illustrate the potential of AS. A version comprising two parameters is shown in Fig 9. Our AS (without proof caching) is interposed between the Frama-C verifier and the default Alt-Ergo prover. AS is supported by an annotation heuristic marking simple constraints of the form v=2 as weakly linking constraints. As can be seen from Fig. 10, the use of AS achieved good performance gains in conjunction with the default prover. We have also evaluated our proof slicing mechanism on a set of 20 small examples obtained from the Frama-C distribution, on which the use of proof slicing did not yield any noticeable gain. It remains our thesis that larger, more complex examples would, in general, benefit more from our proof slicing methods.

```
void spring2 (int *x0, int *x1)
/*@ requires *x0>2 ∧ *x1>2;
    ensures *x0=old(*x0)+2
           ∧ *x1=old(*x1)+2 */
{ int v = 2;
  *x0=*x0+v; *x1=*x1+v;
  if (*x0>4) {
    *x0++; *x1++;
    if (*x1>4) {
      *x0--; *x1--; }}}
```

Fig. 9. A simple contrived procedure

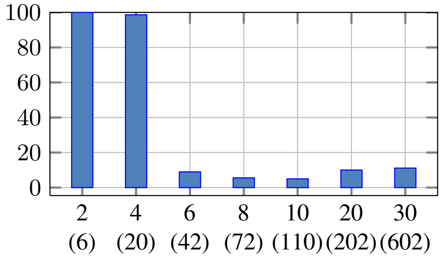

Fig. 10. Comparison (%) of AS over NS on the Spring Benchmark with Frama-C. The number of parameters ranges from 2 to 30 and the number of generated proof obligations are given in the parentheses.

8 Related Work and Conclusion

The problem of filtering irrelevant information has been studied under different guises in several research areas. In [12], the authors focus on filtering out non-relevant information in knowledge bases. They discuss the concept of free variable independence for a conservative partitioning scheme and the concept of forgetting constraints, by which they eliminate irrelevant variables and produce the strongest consequent of the initial formula containing only relevant variables. However, the lack of an aggressive slicing mechanism (which in our case was supported by annotating weak links between distinct properties) leads to higher overheads in both the elimination and the solving phases.

Huang et al. [10] focus on slicing proofs for the infeasibility of counterexamples generated from a model checking process. The insight of this work is that global proofs can be sliced into independent proofs of atomic predicates, and memoization can be used to store the smaller proofs. While the general slicing technique has also been refined via a myriad of proposals (such as combined with abstract interpretation [21]), no mechanism has been proposed to allow a more flexible tradeoff of effectiveness versus conservatism in the slicing process.

Yet another direction of related research focuses on conservatively slicing formulas in connected components in order to simplify the satisfiability and entailment checks. In [1], Amir et al. introduce a methodology for representing large knowledge bases, namely sets of axioms, as trees of loosely connected partitions. They also define a message passing mechanism for reasoning over individual partitions. This has the effect of maintaining the linking information, but leading to higher overheads.

Simpler schemes, *e.g.*, conservative partitioning, have been proposed for SAT solvers. The benefits of an union-find approach over the depth first search in identifying partitions are emphasized in [2]. In [24], a hypergraph cut method partitions the problem, then checks individual partitions and corroborates the results based on the assignments of the linking variables. In [19], SAT solvers are employed for each subproblem while delaying the assignments of linking variables to reduce the search space. In contrast to these methods, our approach refrains from converting implication checks into SAT checks, thus doing a better job at identifying weak linking constraints, and consequently yielding smaller proof slices. We also introduce customizable formula slicing capabilities that facilitate the exploration of new strategies. Our experiments shows that the approach is capable of speed gains without loss of completeness.

Finally, we mention Craig interpolation-based approaches, such as [9], that use interpolation to infer relevant predicates as a way of implementing abstraction refinement more efficiently. In these approaches, the notion of relevance is encoded in entailments and detected by an interpolating prover [14]. In contrast, relevance detection in our approach is largely syntactic, allowing the development of a generic proof slicing framework for automated program verification that would be effective for a broad range of off-the-shelf theorem provers used as back-end.

Conclusion. We have proposed a formal framework that allows the development of modular and extensible proof slicing mechanisms. Our proposal has been validated by an implementation and several experiments. Our technique shows considerable performance gains especially when weakly linking constraints are properly identified. Our aggressive proof slicing mechanism, based on the premise that a simple annotation scheme is sufficient to highlight weakly linking information, allowed us to develop a guided proof slicing process with surprisingly good performance. Experiments showed multi-fold reductions in verification times for each of the state-of-the-art provers used as back-end. We believe that our proposal is of importance for automated verification systems that are geared towards full functional correctness, where proof obligations are not only large and complex but may also be highly intertwined.

Acknowledgement This work is supported by the research grant MOE2009-T2-1-063.

References

1. Eyal Amir and Sheila McIlraith. Partition-based logical reasoning for first-order and propositional theories. In *Artificial Intelligence*, volume 162, pages 49–88, February 2005.
2. A. Biere and C. Sinz. Decomposing SAT problems into connected components. In *JSAT*, 2006.
3. Samuel R. Buss. An introduction to proof theory. In *Handbook of Proof Theory*, 1998.

4. L. de Moura and N. Bjørner. Relevancy propagation. Technical report, MSR, 2007.
5. L. de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
6. H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. In *ESE*, volume 10, 2005.
7. Andreas Dolzmann and Thomas Sturm. Redlog: computer algebra meets computer logic. In *SIGSAM Bulletin*, volume 31, pages 2–9, June 1997.
8. A. Fietzke and C. Weidenbach. Labelled splitting. In *Annals of MAI*, volume 55, 2009.
9. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.
10. Hai Huang, Wei-Tek Tsai, and Raymond A. Paul. Proof slicing with application to model checking web services. In *ISORC*, pages 292–299, 2005.
11. N. Klarlund and A. Moller. MONA Version 1.4 - User Manual. BRICS Notes Series, 2001.
12. J. Lang, P. Liberatore, and P. Marquis. Propositional independence: formula-variable independence and forgetting. In *Journal of Artificial Intelligence Research*, volume 18, 2003.
13. K.R.M Leino, M. Moskal, and W. Schulte. Verification condition splitting. 2008.
14. K. L. McMillan. An interpolating theorem prover. In *TACAS*, 2004.
15. J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. In *Journal of Applied Logic*, pages 41–57, 2009.
16. J. A. Navarro Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566, 2011.
17. H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape And Size Properties via Separation Logic. In *VMCAI*, pages 251–266, 2007.
18. P. W. O'Hearn, J. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In *CSL*, 2001.
19. T. J. Park and A. V. Gelder. Partitioning methods for satisfiability testing on large formulas. In *CADE*, pages 748–762, 1996.
20. W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. In *Communications of the ACM*, volume 8, pages 102–114, 1992.
21. Hyoung Seok Hong, Insup Lee, and Oleg Sokolsky. Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In *SCAM*, 2005.
22. Uffe Sørensen. Slicing for Uppaal. Technical report, AALBORG University, 2008.
23. Frama-C Software Analyser System. http://frama-c.com. 2012.
24. J. Torres-Jimenez, L. Vega-Garcia, C.A. Coutino-Gomez, and F.J. Cartujano-Escobar. SSTP: An approach to Solve SAT instances Through Partition. In *WSEAS*, 2004.

A Detailed Experimental Results

In this section, we provide the detailed numerical data presented by the plots in Fig. 6, 7, 8 and 10 via Table 2, 3, 4 and 1, respectively. Table 2 presents timing measurements (in seconds), with the complete proof slicing mechanism (CS) turned both on and off, for the first two benchmarks of heap-manipulating programs and SIR/Siemens. The CS timings also includes the slicing overhead, beside the time spent on each prover. Moreover, the last two rows of the table show the effect of proof slicing with significant reduction on the size of both unsatisfiability and entailment proofs, that brings the performance improvement.

Table 3 illustrates the performance benefits of aggressive proof slicing (AS) on programs whose specifications contain weakly linking components. Beside the effective reduction on prover time by AS, the table also shows that the number of annotated constraints in the specifications, which are inferred automatically via some heuristics, is small because these annotations can be reused across the whole verification process. Thus, the annotation inference overhead is not noticeable.

Table 4 presents the prover times (in seconds) or the percentage of completed proofs (with a timeout of 2 seconds for each UNSAT checks) for the Spaguetti benchmark. To facilitate a fair comparison in the case of timeouts, the estimated time of completing all 1000 tests in each Spaguetti testcase is calculated using:

$$t_{estimated} = (t_{measured} - t_{timeout} \times x) + (\frac{t_{timeout} \times n \times x}{n - x})$$

where x and n are the number of *timeouts* and the total number of *runs*, respectively. In this sum, the augend is the actual verification time taken by successful runs, and the addend is the estimated timing for the timeout cases. In this table, the number of heap constraints and arithmetic constraints (*e.g.*, disequalities on heap addresses) illustrate the complexity of the Spaguetti testcases.

Lastly, Table 1 shows the (actual or *estimated*) verification time (in seconds) with the 1s-timeout for each proof obligation for the Spring benchmark with Frama-C. The estimated time for incompleted proofs are calculated by the same formula used in the Spaguetti benchmark.

Spring	# P.O	**NS**	*Estimated* **NS**	**AS**
2	6	0.3	0.3	0.3
4	20	0.7	0.7	0.7
6	42	14.3 (71.4%)	*19.1*	1.7
8	72	35.8 (55.6%)	*61.4*	3.4
10	110	66.0 (45.5%)	*138.0*	6.34
20	202	124.4 (49.0%)	*231.6*	23.2
30	602	505.0 (20.6%)	*2347.6*	261.2

Table 1. Total Verification Times in seconds (and % Proof Obligation (P.O) completed before a 1s-timeout) for the Spring Benchmark with Frama-C.

Programs	LOC	**No Slicing (NS)**				**Complete Slicing (CS)**				Proof Size Reduction (%)	
		Omega	MONA	Z3	Redlog	Omega	MONA	Z3	Redlog	UNSAT	ENTAIL
Heap-Manipulating Programs											
AVL Tree	1098	187.4	6698.0	11.4	279.0	6.8	244.6	4.1	34.2	70.8	59.6
Linked List	307	0.5	36.43	0.3	2.7	0.3	2.7	0.2	0.9	65.8	46.6
Sorted Linked List	844	1.6	119.6	1.2	11.7	0.7	6.8	0.5	3.1	65.3	51.3
Doubly Linked List	278	0.9	603.9	0.7	7.7	0.5	4.3	0.4	2.1	70.7	51.5
Complete Tree	225	2.0	85.3	1.2	11.2	1.3	14.7	0.9	6.5	50.2	18.9
Heap Tree	214	32.1	227.0	1.9	46.7	1.1	18.3	0.6	6.5	70.2	58.4
Binary Search Tree	344	0.6	11.5	0.4	3.7	0.2	3.5	0.2	0.9	64.0	45.8
Perfect Tree	166	0.4	3.2	0.3	2.7	0.1	1.6	0.1	0.7	58.5	47.1
Red-Black Tree	1122	5.7	401.5	2.9	43.2	1.3	39.4	0.9	5.7	80.4	64.4
Big Int (w/ Karatsuba mult.)	235	-	-	-	329.4	-	-	-	35.0	61.0	37.4
Total / *Average*:	4833	231.1	8186.4	20.2	737.8	12.4	335.8	7.9	95.6	*67.1*	*54.3*
Prover time reduction (%):						**94.6**	**95.9**	**60.7**	**87.0**		
SIR/Siemens Benchmark											
printtokens, printtokens2, replace, tcas (w/ array)	2033	-	-	22.9	-	-	-	12.2	-	58.8	29.5
schedule, schedule2	786	33.6	386.1	16.7	406.1	6.7	8.0	2.0	16.2	83.8	65.0
Total / *Average*:	2819	33.6	386.1	39.7	406.1	6.7	8.0	14.2	16.2	*64.1*	*37.2*
Prover time reduction (%):						**80.1**	**97.9**	**64.1**	**96.0**		

Table 2. Prover Times (in seconds) for Program Verification **without** and **with** Proof Slicing mechanisms.

Programs (props: size (s), height (h), sets (t) balance factor (b), black height(bh))	anno. ctrs /total ctrs	**No Slicing (NS)**			**Complete Slicing (CS)**			**Aggressive Slicing (AS)**		
		Z3	MONA	Omega	Z3	MONA	Omega	Z3	MONA	Omega
AVL(1) (s, h)	1/103	4.9	-	8.1	3.9	-	4.7	1.5	-	1.9
AVL(2) (s, h, b)	2/125	3.4	-	5.6	2.4	-	2.4	1.4	-	1.5
AVL(3) (s, h, b)	5/135	7.5	-	9.5	5.6	-	7.6	1.8	-	2.3
AVL(4) (s, h, b)	6/136	8.5	-	305.2	6.3	-	80.1	1.9	-	2.3
AVL(5) (s, h, t)	1/148	8.5	1983.1	8.4	5.1	224.0	5.3	2.5	110.6	2.3
RB Tree(1) (s, bh, t)	1/371	1.9	469.0	2.2	1.4	179.1	1.4	0.9	14.5	1.1
Total:	16/1018	34.7	2452.1	339.0	24.7	403.1	101.5	10.0	125.1	11.4
NS→CS/AS (%):					**28.9**	**83.6**	**70.1**	**71.1**	**94.9**	**96.6**
CS→AS (%):								**65.3**	**69.0**	**88.8**

Table 3. Prover Times (in seconds) on Examples with Weakly Linking Components

Spaguetti (# Vars)	Heap Ctr.(K)	Arith. Ctr. (DisEq.)(K)	**No Slicing (NS)**					**Complete Slicing (CS)**					**Aggressive Slicing (AS)**				
			Omega	Omega(+)	Z3	Z3(+)	HR	Omega	Omega(+)	Z3	Z3(+)	HR	Omega	Omega(+)	Z3	Z3(+)	HR
10	8.0	9.0	(0.01)	(0.01)	16.7	12.6	24.1	(55.1)	(55.6)	2.7	2.3	16.9	3.7	0.2	2.9	0.1	99.6
11	8.8	8.3	(0.01)	(0.01)	22.8	16.4	25.9	(49.6)	(50.3)	4.0	3.3	19.5	5.4	0.3	4.1	0.2	99.7
12	10.9	7.1	(0.01)	(0.01)	34.1	14.2	25.7	(52.4)	(53.3)	3.2	2.9	22.0	8.2	0.4	4.3	0.5	99.7
13	11.4	8.6	(0.0)	(0.0)	35.5	25.0	26.3	(42.4)	(42.6)	5.0	4.5	20.9	9.0	0.3	6.5	0.5	99.7
14	11.6	10.1	(0.0)	(0.0)	35.1	28.8	26.6	(32.3)	(32.9)	5.4	4.4	20.2	10.1	0.4	5.2	0.4	99.7
15	11.6	12.5	(0.0)	(0.0)	42.0	35.0	27.2	(25.8)	(26.1)	7.5	7.1	18.9	11.8	0.4	8.7	0.4	99.8
16	10.9	20.3	(0.0)	(0.0)	38.3	27.0	27.0	(27.7)	(27.7)	6.7	5.4	16.7	8.2	0.4	6.4	0.4	99.8
17	12.8	17.5	(0.0)	(0.0)	40.3	41.0	27.2	(25.3)	(25.3)	10.2	8.4	16.8	12.4	0.6	9.4	0.5	99.8
18	11.4	30.7	(0.0)	(0.0)	39.3	27.7	26.7	(30.9)	(30.9)	6.9	6.0	16.2	7.2	0.3	5.8	0.4	99.8
19	12.7	25.6	(0.0)	(0.0)	60.5	42.6	27.5	(27.4)	(27.4)	10.7	8.8	16.0	12.9	0.4	9.5	0.4	99.8
20	14.1	21.0	(0.0)	(0.0)	89.4	62.2	28.3	(19.9)	(20.0)	16.6	13.2	18.9	20.5	0.9	15.4	0.7	99.9
Actual/*Estimated*:	124.2	170.7	-	-	454.0	332.5	26.6	*24025*	*23713*	79.1	66.4	18.5	109.5	4.6	78.2	4.4	99.7
NS→CS/AS (%):								-	-	**76.2**	**80.0**		-	-	**82.8**	**98.7**	
CS→AS (%):													**99.5**	**99.9**	**1.2**	**93.4**	

Table 4. Prover Times in seconds (or % completed) for the Spaguetti Benchmark. (+) indicates caching used; HR denotes Cache Hit Rate