A Specialization Calculus for Pruning Disjunctive Predicates to Support Verification

Wei-Ngan Chin[1] Cristian Gherghina[1] Răzvan Voicu[1]
Quang Loc Le[1] Florin Craciun[1] Shengchao Qin[2]

(1) Department of Computer Science, National University of Singapore
(2) School of Computing, Teesside University

Abstract. Separation logic-based abstraction mechanisms, enhanced with user-defined inductive predicates, represent a powerful, expressive means of specifying heap-based data structures with strong invariant properties. However, expressive power comes at a cost: the manipulation of such logics typically requires the unfolding of disjunctive predicates which may lead to expensive proof search. We address this problem by proposing a *predicate specialization* technique that allows efficient symbolic pruning of infeasible disjuncts inside each predicate instance. Our technique is presented as a calculus whose derivations preserve the satisfiability of formulas, while reducing the subsequent cost of their manipulation. Initial experimental results have confirmed significant speed gains from the deployment of predicate specialization. While specialization is a familiar technique for code optimization, its use in program verification is new.

1 Introduction

Abstraction mechanisms are important for modelling and analyzing programs. Recent developments allow richer classes of properties to be expressed via user-defined predicates for capturing commonly occurring patterns of program properties. Separation logic-based abstraction mechanisms represent one such development. As an example, the following predicate captures an abstraction of a sorted doubly-linked list.

$$\begin{array}{l}\textbf{data}\ \texttt{node}\ \{\ \texttt{int val};\ \texttt{node prev};\ \texttt{node next};\ \}\\ \texttt{dll}(\texttt{root},\texttt{p},\texttt{n},\texttt{S}) \equiv \texttt{root}{=}\texttt{null} \wedge \texttt{n}{=}0 \wedge \texttt{S}{=}\{\}\\ \quad \vee\ \exists \texttt{v},\texttt{q},\texttt{S}_1 \cdot \texttt{root}{\mapsto}\texttt{node}(\texttt{v},\texttt{p},\texttt{q}) * \texttt{dll}(\texttt{q},\texttt{root},\texttt{n}{-}1,\texttt{S}_1)\\ \quad\ \ \wedge \texttt{S} = \texttt{S}_1 \cup \{\texttt{v}\} \wedge \forall \texttt{a} {\in} \texttt{S}_1 \cdot \texttt{v} {\leq} \texttt{a} \qquad \textbf{inv}\ \texttt{n}{\geq}0;\end{array}$$

In this definition `root` denotes a pointer into the list, `n` the length of the list, `S` represents its set of values, whereas `p` denotes a backward pointer from the first node of the doubly-linked list. The invariant $\texttt{n}{\geq}0$ must hold for all instances of this predicate.

We clarify the following points. Firstly, spatial conjunction, denoted by the symbol $*$, provides a concise way of describing disjoint heap spaces. Secondly, this abstraction mechanism is inherently infinite, due to recursion in predicate definition. Thirdly, a predicate definition is capable of capturing multiple features of the data structure it models, such as its size and set of values. While this richer set of features can enhance the precision of a program analysis, it inevitably leads to larger disjunctive formulas.

This paper is concerned with a novel way of handling disjunctive formulas, in conjunction with abstraction via user-defined predicates. While disjunctive forms are natural and expressive, they are major sources of redundancy and inefficiency. The goal of this paper is to ensure that disjunctive predicates can be efficiently supported in a program analysis setting, in general, and program verification setting, in particular.

To achieve this, we propose a *specialization calculus* for disjunctive predicates that supports symbolic pruning of infeasible states within each predicate instance. This allows for the implementation of both *incremental pruning* and *memoization* techniques. As a methodology, predicate specialization is not a new concept, since general specialization techniques have been extensively used in the optimization of logic programs [18, 17, 11]. The novelty of our approach stems from applying specialization to a new domain, namely program verification, with its focus on pruning infeasible disjuncts, rather than a traditional focus on propagating static information into callee sites. This new use of specialization yields a fresh approach towards optimising program verification. This approach has not been previously explored, since pervasive use of user-defined predicates in analysis and verification has only become popular recently (e.g. [14]). Our key contributions are:

- We propose a *new specialization calculus* that leads to more effective program verification. Our calculus specializes proof obligations produced in the program verification process, and can be used as a preprocessing step before the obligations are fed into third party theorem provers or decision procedures.
- We adapt *memoization* and *incremental pruning* techniques to obtain an optimized version of the specialization calculus.
- We present a prototype implementation of our specialization calculus, integrated into an existing program verification system. The use of our specializer yields significant reductions in verification times, especially for larger problems.

Section 2 illustrates the technique of specializing disjunctive predicates. Section 3 introduces the necessary terminology. Section 4 presents our calculus for specializing disjunctive predicates and outlines its formal properties. Section 5 presents inference mechanisms for predicate definitions to support our specialization calculus. Section 6 presents experimental results which show multi-fold improvement to verification times for larger problems. Section 7 discusses related work, prior to a short conclusion.

2 Motivating Example

Program states that are built from predicate abstractions are more concise, but may require properties that are hidden inside predicates. As an example, consider :

$$\mathtt{dll}(\mathtt{x}, \mathtt{p}_1, \mathtt{n}, \mathtt{S}_1) * \mathtt{dll}(\mathtt{y}, \mathtt{p}_2, \mathtt{n}, \mathtt{S}_2) \wedge \mathtt{x} \neq \mathtt{null}$$

This formula expresses the property that the two doubly-linked lists pointed to by $\mathtt{x}$ and $\mathtt{y}$ have the same length. Ideally, we should augment our formula with the property: $\mathtt{y} \neq \mathtt{null}$, $\mathtt{n} > 0$, $\mathtt{S}_1 \neq \{\}$ and $\mathtt{S}_2 \neq \{\}$, currently hidden inside the two predicate instances but may be needed by the program verification tasks at hand.

A naive approach would be to unfold the two predicate instances, but this would blow up the number of disjuncts to four, as shown:

$$\begin{array}{l}
\mathtt{x}{=}\mathtt{null} \wedge \mathtt{y}{=}\mathtt{null} \wedge \mathtt{n}{=}0 \wedge \mathtt{S}_1{=}\{\} \wedge \mathtt{S}_2{=}\{\} \wedge \mathtt{x}{\neq}\mathtt{null} \\
\vee\ \mathtt{y}{\mapsto}\mathtt{node}(\mathtt{v}_2,\mathtt{p}_2,\mathtt{q}_2) * \mathtt{dll}(\mathtt{q}_2,\mathtt{y},\mathtt{n}{-}1,\mathtt{S}_4) \wedge \mathtt{x}{=}\mathtt{null} \\
\quad \wedge \mathtt{S}_1{=}\{\} \wedge \mathtt{n}{=}0 \wedge \mathtt{S}_2{=}\{\mathtt{v}_2\} \cup \mathtt{S}_4 \wedge \mathtt{n}{-}1{\geq}0 \wedge \mathtt{x}{\neq}\mathtt{null} \\
\vee\ \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v}_1,\mathtt{p}_1,\mathtt{q}_1) * \mathtt{dll}(\mathtt{q}_1,\mathtt{x},\mathtt{n}{-}1,\mathtt{S}_3) \wedge \mathtt{y}{=}\mathtt{null} \wedge \mathtt{n}{=}0 \\
\quad \wedge \mathtt{S}_1{=}\{\mathtt{v}_1\} \cup \mathtt{S}_3 \wedge \mathtt{S}_2{=}\{\} \wedge \mathtt{n}{-}1{\geq}0 \wedge \mathtt{x}{\neq}\mathtt{null} \\
\vee\ \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v}_1,\mathtt{p}_1,\mathtt{q}_1) * \mathtt{y}{\mapsto}\mathtt{node}(\mathtt{v}_2,\mathtt{p}_2,\mathtt{q}_2) * \mathtt{dll}(\mathtt{q}_1,\mathtt{x},\mathtt{n}{-}1,\mathtt{S}_3) \\
\quad * \mathtt{dll}(\mathtt{q}_2,\mathtt{y},\mathtt{n}{-}1,\mathtt{S}_4) \wedge \mathtt{S}_1{=}\{\mathtt{v}_1\} \cup \mathtt{S}_3 \wedge \mathtt{S}_2{=}\{\mathtt{v}_2\} \cup \mathtt{S}_4 \wedge \mathtt{n}{-}1{\geq}0 \wedge \mathtt{x}{\neq}\mathtt{null}
\end{array}$$

As contradictions occur in the first three disjuncts, we can simplify our formula to:

$$\begin{array}{l}
\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v}_1,\mathtt{p}_1,\mathtt{q}_1) * \mathtt{y}{\mapsto}\mathtt{node}(\mathtt{v}_2,\mathtt{p}_2,\mathtt{q}_2)\ *\ \mathtt{dll}(\mathtt{q}_1,\mathtt{x},\mathtt{n}{-}1,\mathtt{S}_3) \\
\quad *\ \mathtt{dll}(\mathtt{q}_2,\mathtt{y},\mathtt{n}{-}1,\mathtt{S}_4) \wedge \mathtt{S}_1{=}\{\mathtt{v}_1\} \cup \mathtt{S}_3\ \wedge\ \mathtt{S}_2{=}\{\mathtt{v}_2\} \cup \mathtt{S}_4 \wedge \mathtt{n}{-}1{\geq}0 \wedge \mathtt{x}{\neq}\mathtt{null}
\end{array}$$

After removing infeasible disjuncts, the propagated properties are exposed in the above more *specialized* formula. However, this naive approach has the shortcoming that unfolding leads to an increase in the number of disjuncts handled, and its associated costs.

A better approach would be to avoid predicate unfolding, but instead apply predicate specialization to prune infeasible disjuncts and propagate hidden properties. Given a predicate $\mathtt{pred}(\cdots)$ that is defined by $\mathtt{k}$ disjuncts, we shall denote each of its specialized instances by $\mathtt{pred}(\cdots)@\mathtt{L}$, where $\mathtt{L}$ denotes a subset of the disjuncts, namely $\mathtt{L} \subseteq \{1 \ldots k\}$, that have not been pruned. Initially, we can convert each predicate instance $\mathtt{pred}(\cdots)$ to its most general form $\mathtt{pred}(\cdots)@\{1 \ldots k\}$, while adding the basic invariant of the predicate to its context. As an illustration, we may view the definition of $\mathtt{dll}$ as a predicate with two disjuncts, labelled informally by $\mathtt{1}$: and $\mathtt{2}$: prior to each of its disjuncts, as follows:

$$\begin{array}{l}
\mathtt{dll}(\mathtt{root},\mathtt{p},\mathtt{n},\mathtt{S}) \equiv \mathtt{1}{:}(\mathtt{root}{=}\mathtt{null} \wedge \mathtt{n}{=}0 \wedge \mathtt{S}{=}\{\}) \\
\quad \vee\ \mathtt{2}{:}(\mathtt{root}{\mapsto}\mathtt{node}(\mathtt{v},\mathtt{p},\mathtt{q}) * \mathtt{dll}(\mathtt{q},\mathtt{root},\mathtt{n}{-}1,\mathtt{S}_1)\ \wedge\ \mathtt{S} = \mathtt{S}_1{\cup}\{\mathtt{v}\} \wedge \forall \mathtt{a}{\in}\mathtt{S}_1 \cdot \mathtt{v}{\leq}\mathtt{a})
\end{array}$$

We may convert each $\mathtt{dll}$ predicate by adding its invariant $\mathtt{n}{\geq}0$, as follows:

$$\mathtt{dll}(\mathtt{x},\mathtt{p},\mathtt{n},\mathtt{S}) \implies \mathtt{dll}(\mathtt{x},\mathtt{p},\mathtt{n},\mathtt{S})@\{1,2\} \wedge \mathtt{n}{\geq}0$$

With our running example, this would lead to the following initial formula after the same invariant $\mathtt{n}{\geq}0$ (from the two predicate instances) is added.

$$\mathtt{dll}(\mathtt{x},\mathtt{p}_1,\mathtt{n},\mathtt{S}_1)@\{1,2\} * \mathtt{dll}(\mathtt{y},\mathtt{p}_2,\mathtt{n},\mathtt{S}_2)@\{1,2\} \wedge \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{n}{\geq}0$$

This predicate may be further specialized with the help of its context by pruning away disjuncts that are found to be infeasible. Each such pruning would allow more states to be propagated by the specialized predicate. By using the context, $\mathtt{x}{\neq}\mathtt{null}$, we can specialize the first predicate instance to $\mathtt{dll}(\mathtt{x},\mathtt{p}_1,\mathtt{n},\mathtt{S}_1)@\{2\}$ since this context contradicts the first disjunct of the $\mathtt{dll}$ predicate. With this specialization, we may strengthen the context with a propagated state, namely $\mathtt{n}{>}0 \wedge \mathtt{S}_1{\neq}\{\}$, that is implied by its specialized instance, as follows:

$$\mathtt{dll}(\mathtt{x},\mathtt{p}_1,\mathtt{n},\mathtt{S}_1)@\{2\} * \mathtt{dll}(\mathtt{y},\mathtt{p}_2,\mathtt{n},\mathtt{S}_2)@\{1,2\}\ \wedge\ \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{n}{>}0 \wedge \mathtt{S}_1{\neq}\{\}$$

Note that $\mathtt{n}{\geq}0$ is removed when a stronger constraint $\mathtt{n}{>}0$ is added. The new constraint

$$
\begin{array}{lll}
\mathit{pred} & ::= p(v^*) \equiv \Phi \ \ [\mathit{inv}\ \ \pi] & \\
\Phi & ::= \bigvee\ (\exists w^* \cdot \sigma)^* & \sigma ::= \kappa \wedge \pi \\
\kappa & ::= \texttt{emp} \mid v \mapsto c(v^*) \mid p(v^*) \mid \kappa_1 * \kappa_2 & \\
\pi & ::= \alpha \mid \pi_1 \wedge \pi_2 & \\
\alpha & ::= \beta \mid \neg\beta & \\
\beta & ::= v_1 = v_2 \mid v = \texttt{null} \mid a \leq 0 \mid a = 0 & \\
a & ::= k \mid k \times v \mid a_1 + a_2 \mid \mathit{max}(a_1,a_2) \mid \mathit{min}(a_1,a_2) & \\
\mathit{where} & \mathit{p\ is\ a\ predicate\ name};\ v, w\ \mathit{are\ variable\ names}; & \\
& \mathit{c\ is\ a\ data\ type\ name};\ k\ \mathit{is\ an\ integer\ constant}; & \\
& \kappa\ \mathit{represents\ heap\ formulas};\ \pi\ \mathit{represents\ pure\ formulas}; & \\
& \beta\ \mathit{represents\ atomic\ interpreted\ predicates} &
\end{array}
$$

Fig. 1. The Unannotated Specification Language.

$\texttt{n}{>}0$ now triggers a pruning of the second predicate instance, since its first disjunct can be shown to be infeasible. This leads to a specialization of the second predicate, with more propagation of atomic formulas, as follows:

$$
\begin{array}{l}
\texttt{dll}(\texttt{x}, \texttt{p}_1, \texttt{n}, \texttt{S}_1)@\{2\} * \texttt{dll}(\texttt{y}, \texttt{p}_2, \texttt{n}, \texttt{S}_2)@\{2\} \\
\quad \wedge\ \texttt{x} \neq \texttt{null} \wedge \texttt{n} > 0 \wedge \texttt{S}_1 \neq \{\} \wedge \texttt{y} \neq \texttt{null} \wedge \texttt{S}_2 \neq \{\}
\end{array}
$$

In a nutshell, the goal of our approach is to apply aggressive specialization to our predicate instances, without the need to resort to predicate unfolding, in the hope that infeasible disjuncts are pruned, where possible. In the process, our specialization technique is expected to propagate states that are consequences of each of the specialized predicate instances. We expect this proposal to support more efficient manipulation of program states, whilst keeping the original abstractions intact where possible.

3 Formal Preliminaries

Our underlying computation model is a state machine with a countable set of variables and a heap, which is a partial mapping from addresses to values.

Fig. 1 defines the syntax of our (unannotated) specification language. We denote sequences of variables $v_1, \ldots, v_n$ by the notation v^*, and by β atomic interpreted predicates such as equality and disequality of program variables and arithmetic expressions. Conjunctions of (possibly negated) atomic predicates form *pure* formulas, which we denote by the symbol π. Heap formulas, denoted by κ, model the configuration of the heap. They rely on two important components: data constructors $c(v^*)$, which model simple data records (e.g. the node of a tree), and inductively defined predicates, which are generated by the non-terminal *pred* in Fig. 1.

Definition 1 (Heap Formula and Predicate Definition) *A* heap formula κ *is either the symbol* emp*, denoting the empty heap, or a formula of the form $v \mapsto c(v^*)$, denoting a singleton heap, or a predicate $p(v^*)$, or finally, a formula of the form $\kappa_1 * \kappa_2$, where κ_1 and κ_2 are heap formulas, and $*$ is the separating conjunction connective. Predicates*

$$
\begin{array}{lll}
\textit{spred} & ::= p(v^*) \equiv \hat{\Phi}; \mathcal{I}; \mathcal{R} & \\
\hat{\Phi} & ::= \bigvee\ (\exists w^* \cdot \hat{\sigma} \mid C)^* & \hat{\sigma} ::= \hat{\kappa} \wedge \pi \\
\hat{\kappa} & ::= \mathtt{emp} \mid v \mapsto c(v^*) \mid p(v^*)@L\#R \mid \hat{\kappa}_1 * \hat{\kappa}_2 &
\end{array}
$$

where p, v, w, c, π *denote the same as in Fig. 1;*
$\mathcal{I}$ *is a family of invariants*;
$\mathcal{R}$ *is a set of pruning conditions*.
C *is a pure formula denoting a context computed in the specialization process.*

Fig. 2. The Annotated Specification Language.

are defined inductively as the equivalence between a predicate symbol $p(v^*)$*, and disjunctions of formulas of the form* $\exists w^* \cdot (\kappa \wedge \pi)$*, where variables* v^* *may appear free. Predicate definitions may be augmented with invariants specified by the* inv *keyword.*

Unannotated formulas become annotated in the specialization process. Fig. 2 defines the syntax of the *annotated* specification language. Annotated predicate definitions are generated by the nonterminal *spred*.

Definition 2 (Annotated Predicates and Formulas) *Given a predicate definition* $p(v^*) \equiv \bigvee\ (\exists w^* \cdot \kappa \wedge \pi)^*$*, the corresponding* annotated predicate definition *has the form* $p(v^*) \equiv \bigvee\ (\exists w^* \cdot \hat{\kappa} \wedge \pi \mid C)^*; \mathcal{I}; \mathcal{R}$*, where* $\mathcal{I}$ *is a family of invariants, and* $\mathcal{R}$ *is a set of pruning conditions. Each disjunct* $\exists w^* \cdot \hat{\kappa} \wedge \pi \mid C$ *now contains the annotated counterpart* $\hat{\kappa}$ *of* κ*, and is augmented with a context* C*, which is a pure formula for which* $C \rightarrow \pi$ *always holds. Intuitively,* C *captures also the consequences of the specialized states of* $\hat{\kappa}$*. An* annotated formula *is a formula where all the predicate instances are annotated. An* annotated predicate *instance is of the form* $p(v^*)@L\#R$*, where* $L \subseteq \{1, .., n\}$ *is a set of labels denoting the unpruned disjuncts, and where* $R \subseteq \mathcal{R}$ *is a set of remaining pruning conditions. The* set of invariants $\mathcal{I}$ *is of the form* $\{(L \rightarrow \pi_L) \mid \emptyset \subset L \subseteq \{1, .., n\}\}$*. For each set of labels* L*,* π_L *represents the invariant for the specialized predicate instance* $p(v^*)@L$*. For a given annotated predicate instance* $p(v^*)@L\#R$*, it is possible for* $L = \emptyset$*. When this occurs, it denotes that none of the predicate's disjuncts are satisfiable. Moreover, we have* $\pi_\emptyset = false$ *which will contribute towards a false state (or contradiction) for its given context.*

Definition 3 (Pruning Condition) *The set of pruning conditions* $\mathcal{R}$ *is of the form* $\{(\alpha \leftarrow L) \mid \cdots\}$*. A* pruning condition *is a pair between an atomic predicate instance* α *and a set of labels* L*, written* $\alpha \leftarrow L$*. Its intuitive meaning is that the disjuncts in* L *should be kept if* α *is satisfiable in the current context.*

Given a predicate definition $p(v^*) \equiv \bigvee_{i=1}^{n} (\exists w^* \cdot \hat{\sigma_i} \mid C_i);\ \mathcal{I};\ \mathcal{R}$, we call $\mathtt{D}_i =_{df} (\exists w^* \cdot \hat{\sigma_i} \mid C_i)$ the i^{th} disjunct of p ; i will be called the *label* of its disjunct. We shall use $\mathtt{D}_i$ freely as the i^{th} disjunct of the predicate at hand whenever there is no risk of confusion. We employ a notion of *closure* for a given conjunctive formula. Consider a formula $\pi(w^*) = \exists v^* \cdot \alpha_1 \wedge \cdots \wedge \alpha_m$, where α_i are atomic predicates, and variables w^* appear free. We denote by $\mathtt{S} = closure(\pi(w^*))$ a set of atomic predicates (over the free variables w^*) such that each element $\alpha \in \mathtt{S}$ is entailed by $\pi(w^*)$. Some of the variables w^* may appear free in α but *not* v^*. To ensure this closure set be finite, we also impose a

requirement that weaker atomic constraints are never present in the same set, as follows: $\forall \alpha_i \in \mathtt{S} \cdot \neg(\exists \alpha_j \in \mathtt{S} \cdot i{\neq}j \wedge \alpha_i \implies \alpha_j)$. Ideally, *closure*$(\pi(w^*))$ contains *all* stronger atomic formulas entailed by $\pi(w^*)$, though depending on the abstract domain used, this set may not be computable. A larger closure set leads to more aggressive pruning.

Our specialization calculus (Sec 4) is based on the annotated specification language. We have an initialization and inference process (Sec 5) to automatically generate all annotations (including $\mathcal{I}, \mathcal{R}$) that are required by specialization. For simplicity of presentation, we only include normalized linear arithmetic constraints in our language. Our system currently supports both arbitrary linear arithmetic constraints, as well as set constraints. This is made possible by integrating the Omega [19] and MONA solvers [9] into the system. In principle, the system may support arbitrary constraint domains, provided that a suitable solver is available for the domain of interest. Such a solver should be capable of handling conjunctions efficiently, as well as computing approximations of constraints that convert disjunctions into conjunctions (e.g. hulling).

4 A Specialization Calculus

Our specialization framework detects infeasible disjuncts in predicate definitions without explicitly unfolding them, and computes a corresponding strengthening of the pure part while preserving satisfiability. We present this as a calculus consisting of *specialization rules* that can be applied exhaustively to convert a non-specialized annotated formula[1] into a fully specialized one, with stronger pure parts, that can be subsequently extracted and passed on to a theorem prover for satisfiability/entailment checking. Apart from being syntactically correct, annotated formulas must satisfy the following well-formedness conditions.

Definition 4 (Well-formedness) *For each annotated predicate* $\mathtt{p}(\mathtt{v}^*)@L\#R$ *in the formula at hand, assuming the definition* $p(v^*){\equiv}\bigvee_{i=1}^{n}\mathtt{D_i};\mathcal{I};\mathcal{R}$*, we have that (a)* $L \subseteq \{1,\ldots,n\}$ *; (b)* $R \subseteq \mathcal{R}$ *; and (c) forall* $\alpha{\leftarrow}L_0 \in R$ *we have* $L \cap L_0 \neq \emptyset$*.*

Definition 5 (Specialization Step) *A* specialization step *has the form* $\hat{\Phi}_1 \mid C_1 \xrightarrow{sf} \hat{\Phi}_2 \mid C_2$*, and denotes the relation allows the annotated formula* $\hat{\Phi}_1$ *with context* C_1 *to be transformed into a more specialized formula* $\hat{\Phi}_2$ *with context* C_2*.*

Our calculus produces specialization steps, which are applied in sequence, exhaustively, to produce *fully specialized* formulas (a formal definition of such formulas will be given below). Relation $\xrightarrow{\text{sf}}$ depends on relation $\xrightarrow{\text{sp}}$, which produces predicate specialization steps defined by the following:

Definition 6 (Predicate Specialization Step) *A* predicate specialization *step has form*

$$(1) \qquad p(v^*)@L_1\#R_1 \mid C_1 \xrightarrow{sp} p(v^*)@L_2\#R_2 \mid C_2.$$

and signifies that annotated predicate $p(v^*)@L_1\#R_1 \mid C_1$ *can be specialized into* $p(v^*)@L_2\#R_2 \mid C_2$*, where* $L_2 \subseteq L_1$*,* $R_2 \subset R_1$*, and* C_2 *is stronger than* C_1*.*

[1] The conversion of non-annotated formulas into annotated ones shall be presented in Sec. 5.

$$\frac{R_f = \{(\alpha \leftarrow L_0) \mid (\alpha \leftarrow L_0) \in R,\ (L \cap L_0 = \emptyset) \vee (C \Longrightarrow \alpha)\}}{C, L \vdash R \text{ –filter}\rightarrow (R - R_f)} \quad [\textbf{SP}\text{–}[\textbf{FILTER}]]$$

$$\frac{\begin{array}{c} C \wedge \alpha \Longrightarrow \texttt{false} \quad (\alpha \leftarrow L_0) \in R \quad L \cap L_0 \neq \emptyset \quad L_2 = L - L_0 \\ C_1 = \mathcal{I}nv(p(v^*), L_2) \qquad C \wedge C_1, L_2 \vdash R \text{ –filter}\rightarrow R_1 \end{array}}{p(v^*)@L\#R \mid C \text{ –sp}\rightarrow p(v^*)@L_2\#R_1 \mid C \wedge C_1} \quad [\textbf{SP}\text{–}[\textbf{PRUNE}]]$$

$$\frac{C, L \vdash R \text{ –filter}\rightarrow \emptyset \qquad R \neq \emptyset}{p(v^*)@L\#R \mid C \text{ –sp}\rightarrow p(v^*)@L\#\emptyset \mid C} \quad [\textbf{SP}\text{–}[\textbf{FINISH}]]$$

Fig. 3. Single-step Predicate Specialization

Here, the sets L_1 and L_2 denote sets of disjuncts of $p(v^*)$ that have not been detected to be infeasible. Each specialization step aims at detecting new infeasible disjuncts and removing them during the transformation. Thus L_2 is expected to be a subset of L_1.

The sets of pruning conditions R_1 and R_2 may be redundant, but are instrumental in making specialization efficient. They record incremental changes to the state of the specializer, and represent information that would be expensive to re-compute at every step. Essentially, a pruning condition $\alpha \leftarrow L_0$ states that whenever $\neg\alpha$ is entailed by the current context, the disjuncts whose labels are in L_0 can be pruned. The initial set of pruning conditions is derived when converting formulas into annotated formulas, and is formally discussed in Section 5.

In a nutshell, each specialization step of the form (1) detects (if possible) a pruning condition $\alpha \leftarrow L_0 \in R$ such that if $\neg\alpha$ is entailed by the current context, then the disjuncts whose labels occur in L_0 are infeasible and can be pruned. Given the notations in (1), this is achieved by setting $L_2 = L_1 - L_0$. Subsequently, the current set of pruning conditions is reduced to contain only elements of the form $\alpha' \leftarrow L_0'$ such that $L_0' \cap L_2 \neq \emptyset$. Thus, the well-formedness of the annotated formula is preserved

A key aspect of specialization is that context strengthening helps reveal and prune *mutually infeasible* disjuncts in groups of predicates, which leads to a more aggressive optimization as compared to the case where predicates are specialized in isolation.

Definition 7 (Fully Specialized Formula; Complete Specialization) *An annotated formula is* fully specialized *w.r.t a context when all its annotated predicates have empty pruning condition sets. If the initial pruning condition sets are computed using a notion of* strongest closure*, then for each predicate in the fully specialized formula, all the remaining labels in the predicate's label set denote feasible disjuncts with respect to the current context, and in that sense, the specialization is* complete*.*

This procedure is formalized in the calculus rules given in Figures 3 and 4. Figure 3 defines the predicate specialization relation –sp→. This relation has two main components: the one represented by the rule [SP–[FILTER]], which restores the well-formedness of an annotated predicate, and the one represented by the rule [SP–[PRUNE]], which detects infeasible disjuncts and removes the corresponding labels from the annotation. A third rule, [SP–[FINISH]] produces the fully specialized predicate.

$$\frac{p(v^*)@L\#R \mid C \multimap p(v^*)@L_2\#R_2 \mid C_2}{p(v^*)@L\#R * \hat{\kappa} \mid C \multimap p(v^*)@L_2\#R_2 * \hat{\kappa} \mid C_2} \; [\textsc{sf}\text{-}[\textsc{prune}]]$$

$$\frac{\vdash C \Longrightarrow \alpha_1 \vee \alpha_2 \quad \vdash \alpha_1 \wedge \alpha_2 \Longrightarrow \texttt{false} \quad \forall i \in \{1,2\} \cdot \hat{\kappa} \mid C \wedge \alpha_i \multimap \hat{\kappa}_i \mid C_i}{\hat{\kappa} \mid C \multimap (\hat{\kappa}_1 \mid C_1) \vee (\hat{\kappa}_2 \mid C_2)} \; [\textsc{sf}\text{-}[\textsc{case-split}]]$$

$$\frac{\hat{\kappa}_1 \mid C_1 \multimap \hat{\kappa}_3 \mid C_3}{(\hat{\kappa}_1 \mid C_1) \vee (\hat{\kappa}_2 \mid C_2) \multimap (\hat{\kappa}_3 \mid C_3) \vee (\hat{\kappa}_2 \mid C_2)} \; [\textsc{sf}\text{-}[\textsc{or}]]$$

Fig. 4. Single-step Formula Specialization

The predicate specialization relation can be weaved into the formula specialization relation given in Fig. 4. The first rule, $[\textsc{sf}\text{-}[\textsc{prune}]]$, defines the part of the $\multimap$ relation which picks a predicate in a formula and transforms it using the $\multimap$ relation, leaving the rest of the predicates unchanged. However, the transformation of the predicate's context is incorporated into the transformation of the formula's context. This rule realizes the potential for cross-specialization of predicates, eliminating disjuncts of different predicates that are mutually unsatisfiable.

The rule $[\textsc{sf}\text{-}[\textsc{case-split}]]$ allows further specialization via case analysis. It defines the part of the $\multimap$ relation that produces two instances of the same formula, joined in a disjunction, each of the new formulas having a stronger context. Each stronger context is produced by conjunction with an atom α_i, $i \in \{1, 2\}$, with the requirement that the two atoms be disjoint and their disjunction cover the original context C. This rule is instrumental in guaranteeing that all predicates reach a fully specialized status. Indeed, whenever an annotated predicate has a pruning condition $\alpha \leftarrow L_0$, such that α is not entailed by the context C, yet $\alpha \wedge C$ is satisfiable, the only way to further specialize the predicate is by case analysis with the atoms α and $\neg\alpha$. Finally, the rule $[\textsc{sf}\text{-}[\textsc{or}]]$ handles formulas with multiple disjunctions.

In the remainder of this section, we formalize the notion that our calculus produces terminating derivations, and is sound and complete.

Property 1 *Relations $\multimap$ and $\multimap$ preserve well-formedness. Thus, given two annotated predicate instances* $\mathtt{p}(\mathtt{v}^*)@L_1\#R_1$ *and* $\mathtt{p}(\mathtt{v}^*)@L_2\#R_2$*, if*

$$\mathtt{p}(\mathtt{v}^*)@L_1\#R_1 \mid C_1 \multimap \mathtt{p}(\mathtt{v}^*)@L_2\#R_2 \mid C_2$$

can be derived from the calculus, and $\mathtt{p}(\mathtt{v}^*)@L_1\#R_1$ *is well-formed, then* $\mathtt{p}(\mathtt{v}^*)@L_2\#R_2$ *is well-formed as well. Moreover, for all annotated formulas $\hat{\Phi}_1$ and $\hat{\Phi}_2$, if*

$$\hat{\Phi}_1 \mid C_1 \multimap \hat{\Phi}_2 \mid C_2$$

can be derived from the calculus, and $\hat{\Phi}_1$ is well-formed, then $\hat{\Phi}_2$ is well formed as well.

A *predicate specialization sequence* is a sequence of annotated predicates such that each pair of consecutive predicates is in the relation $\multimap$. A *formula specialization se-*

quence is a sequence of annotated formulas such that each pair of consecutive formulas is in the $\multimap\!\!\!\!\!\!\text{sf}\!\!\!\!\!\!\rightarrow$ relation.

Definition 8 (Canonical Specialization Sequence) *A canonical specialization sequence is a formula specialization sequence where (a) the first element is well-formed; (b) specialization rules are applied exhaustively ; (c) the* $[\underline{\textbf{SF}-[\textbf{CASE}-\textbf{SPLIT}]}]$ *relation is only applied as a last resort (i.e. when no other relation is applicable); and (d) the case analysis atoms for the* $[\underline{\textbf{SF}-[\textbf{CASE}-\textbf{SPLIT}]}]$ *relation must be of the form* α, $\neg\alpha$, *where* $\alpha{\leftarrow}L_0$ *is a pruning condition occurring in an annotated predicate* $\mathtt{p}(\mathtt{v}^*)@L\#R$ *of the formula, such that* $L \cap L_0 \neq \emptyset$.

Property 2 (Termination) *All canonical specialization sequences are finite and produce either fully specialized formulas, or formulas whose context is unsatisfiable.*

Property 3 (Soundness) *The* $\multimap\!\!\!\!\!\!sp\!\!\!\!\!\!\rightarrow$ *and* $\multimap\!\!\!\!\!\!sf\!\!\!\!\!\!\rightarrow$ *relations preserve satisfiability. Thus, if* $\mathtt{p}(\mathtt{v}_{0..n})@L_1\#R_1 \mid C_1 \multimap\!\!\!\!\!\!sp\!\!\!\!\!\!\rightarrow \mathtt{p}(\mathtt{v}_{0..n})@L_2\#R_2 \mid C_2$ *can be derived from the calculus, then for all heaps* h *and stacks* s, $s, h \models \mathtt{p}(\mathtt{v}^*)@L_1\#R_1 \mid C_1$ *iff* $s, h \models \mathtt{p}(\mathtt{v}^*)@L_2\#R_2 \mid C_2$. *Moreover, if* $\hat{\Phi}_1 \mid C_1 \multimap\!\!\!\!\!\!sf\!\!\!\!\!\!\rightarrow \hat{\Phi}_2 \mid C_2$ *can be derived from the calculus, then* $s, h \models \hat{\Phi}_1|C_1$ *iff* $s, h \models \hat{\Phi}_2|C_2$.

We note here that the set R does not play a role in the way an annotated predicate is interpreted. Mishandling R (as long as no elements are added) may result in lack of termination or incompleteness, but does not affect soundness.

Finally, we address the issue of completeness. This property, however, is dependent on how "complete" the conversion of a predicate into its annotated form is. Thus, we shall first give an ideal characterization of such a conversion, after which we shall endeavour to prove the completeness property. Realistic implementations of this conversion shall be discussed in Section 5.

Definition 9 (Strongest Closure) *The strongest closure of unannotated formula* Φ, *denoted sclosure*(Φ), *is the largest set of atoms* α *with the following properties: (a) for all stacks* s, $s \models \alpha$ *whenever there exists* h *such that* $s, h \models \Phi$, *and (b) if* α *is not of the form* $v{\neq}\mathtt{null}$, *then there exists no atom* α' *strictly stronger than* α – *that is, it is not the case that for all* s, $s \models \alpha$ *whenever* $s \models \alpha'$. *For practical and termination reasons, we shall assume only closures which return finite sets in our formulation.*

In our conversion of an unannotated predicate definition for $p(v^*)$ into the annotated definition $p(v^*){\equiv}\bigvee_{i=1}^{n} \mathtt{D}_i; \mathcal{I}; \mathcal{R}$, we compute the following sets: $\mathtt{G}_i = \mathit{sclosure}(\mathtt{D}_i{\wedge}\pi)$, for $i = 1, .., n$, $\mathtt{H}_L = \{\alpha \mid \text{forall } i \in L, \text{exists } \alpha' \in \mathtt{G}_i \text{ s.t. forall } s, s \models \alpha' \text{ whenever } s \models \alpha\}$ and $\mathcal{I} = \{L{\rightarrow}\pi \mid L{\subseteq}\{1...n\},\ \pi = \bigwedge_{\alpha \in \mathtt{H}_L} \alpha\}$, and $\mathcal{R} = \{\alpha{\leftarrow}L \mid L \text{ is the largest set s.t. } \alpha{\in}\bigcap_{i \in L} \mathtt{G}_i\}$. Moreover, we introduce the notation $\mathcal{I}nv(p(v^*), L) = \pi_L$, where $(L{\rightarrow}\pi_L){\in}\mathcal{I}$. This notation is necessary in applying the rule $[\underline{\textbf{SP}-[\textbf{PRUNE}]}]$.

In practice, either the assumption holds, or the closure procedure computes a close enough approximation to the strongest closure so that very few, if any, infeasible disjuncts are left in the specialized formula.

Property 4 (Completeness) *Let* $\mathtt{p}(\mathtt{v}^*)@L\#\emptyset * \hat{\sigma}|C$ *be a fully specialized formula that resulted from a specialization process that started with an annotated formula. Denote by* π_i, $1 \leq i \leq n$ *the pure parts of the disjuncts in the definition of* $\mathtt{p}(\mathtt{v}^*)$, *and assume that C is satisfiable. Then, for all* $i \in L$, $\pi_i \wedge C$ *is satisfiable.*

Proofs of the above properties, as well as a more detailed discussion of our calculus rules, can be found in [3].

5 Inferring Specializable Predicates

We present inference techniques that must be applied to each predicate definition so that they can support the specialization process. We refer to this process as *inference for specializable predicates*. A predicate is said to be *specializable* if it has multiple disjuncts and it has a non-empty set of pruning conditions. These two conditions would allow a predicate instance to be specializable from one form to another specialized form. Our predicates are processed in a bottoms-up order with the following key steps:

- Transform each predicate definition to its specialized form.
- Compute an *invariant* (in conjunctive form) for each predicate.
- Compute a *family of invariants* to support all specialized instances of the predicate.
- Compute a *set of pruning conditions* for the predicate.
- Specialize recursive invocations of the predicate, if possible.

As a running example for this inference process, let us consider the following predicate which could be used to denote a list segment of singly-linked nodes:

$\mathbf{data}\ \mathtt{snode}\ \{\ \mathtt{int\ val};\ \mathtt{snode\ next};\ \}$
$\mathtt{lseg}(\mathtt{x},\mathtt{p},\mathtt{n})\ \equiv\ \mathtt{x}{=}\mathtt{p}\wedge\mathtt{n}{=}0\ \vee\ \exists\mathtt{q},\mathtt{m}\cdot\mathtt{x}{\mapsto}\mathtt{snode}(\_,\mathtt{q})*\mathtt{lseg}(\mathtt{q},\mathtt{p},\mathtt{m})\wedge\mathtt{m}{=}\mathtt{n}{-}1$

Our inference technique derives the following specializable predicate definition:

$$\begin{array}{l}\mathtt{lseg}(\mathtt{x},\mathtt{p},\mathtt{n})\equiv\mathtt{x}{=}\mathtt{p}\wedge\mathtt{n}{=}0\mid\mathtt{x}{=}\mathtt{p}{\wedge}\mathtt{n}{=}0\vee\\ \exists\mathtt{q},\mathtt{m}\cdot\mathtt{x}{\mapsto}\mathtt{snode}(\_,\mathtt{q}){*}\mathtt{lseg}(\mathtt{q},\mathtt{p},\mathtt{m}){\wedge}\mathtt{m}{=}\mathtt{n}{-}1\mid\mathtt{x}{\neq}\mathtt{null}{\wedge}\mathtt{n}{>}0;\\ \mathcal{I}=\{\{1\}{\rightarrow}\mathtt{x}{=}\mathtt{p}{\wedge}\mathtt{n}{=}0,\ \{2\}{\rightarrow}\mathtt{x}{\neq}\mathtt{null}{\wedge}\mathtt{n}{>}0,\ \{1,2\}{\rightarrow}\mathtt{n}{\geq}0\};\\ \mathcal{R}=\{\mathtt{x}{=}\mathtt{p}{\leftarrow}\{1\},\ \mathtt{n}{=}0{\leftarrow}\{1\},\mathtt{x}{\neq}\mathtt{null}{\leftarrow}\{2\},\ \mathtt{n}{>}0{\leftarrow}\{2\}\}\end{array}$$

Note that we have a family of invariants, named $\mathcal{I}$, to cater to each of the specialized states. The most general invariant for the predicate is $\mathcal{I}\mathtt{nv}(\mathtt{lseg}(\mathtt{x},\mathtt{p},\mathtt{n}),\{1,2\})=\mathtt{n}{\geq}0$. This is computed by a fix-point analysis [4] on the body of the predicate. If we determine that a particular predicate instance can be specialized to $\mathtt{lseg}(\mathtt{x},\mathtt{p},\mathtt{n})@\{2\}$, we may use a stronger invariant $\mathcal{I}\mathtt{nv}(\mathtt{lseg}(\mathtt{x},\mathtt{p},\mathtt{n}),\{2\})=\mathtt{x}{\neq}\mathtt{null}\wedge\mathtt{n}{>}0$ to propagate this constraint from the specialized instance. Such a family of invariants allows us to enrich the context of the predicate instances that are being progressively specialized.

Furthermore, we must process the predicate definitions in a bottom-up order, so that predicates lower in the definition hierarchy are inferred before predicates higher in the hierarchy. This is needed since we intend to specialize the body of each predicate definition with the help of specialized definitions that were inferred earlier. In the case of a set of mutually-recursive predicate definitions, we shall process this set of predicates simultaneously. Initially, we shall assume that the set of pruning conditions for each recursive predicate is empty, which makes its recursive instances unspecializable. However, once its set of pruning conditions has been determined, we can apply further specialization so that the recursive invocations of the predicate are specialized as well.

The formal rules for inferring each specializable predicate are given in Fig. 5. The rule $[\underline{\textsc{init}-[\textsc{multi}-\textsc{spec}]}]$ converts an unannotated formula into its corresponding specialized form. It achieves this by an initialization step via the $\rightarrow$ relation given in Fig. 6, followed by a multi-step specialization using $\rightarrow^*$, without resorting to case specialization (that would otherwise result in an outer disjunctive formula). This essentially applies a transitive closure of $\rightarrow$ until no further reduction is possible.

$$\frac{}{\kappa \wedge \pi \ \text{–if}\rightarrow\ \hat{\kappa} \wedge \pi \mid C_1 \qquad \hat{\kappa} \mid C_1 \ \text{–sf}\rightarrow^{*}\ \hat{\kappa}_1 \mid C_2} \ [\text{ISP}-[\text{MUTLI}-\text{SPEC}]]$$
$$\kappa \wedge \pi \ \text{–msf}\rightarrow\ \hat{\kappa}_1 \wedge \pi \mid C_2$$

$$\frac{spred_{old} = (\mathtt{p}(\mathtt{v}^*) \equiv \bigvee_{i=1}^{n}(\exists u_i^* \cdot \sigma_i)) \qquad \forall i \in \{1,..,n\} \cdot \sigma_i \ \text{–msf}\rightarrow\ \hat{\sigma}_i \mid C_i \quad spred_{new} = (\mathtt{p}(\mathtt{v}^*) \equiv \bigvee_{i=1}^{n}(\exists u_i^* \cdot \hat{\sigma}_i \mid C_i))}{spred_{old} \ \text{–isp}\rightarrow\ spred_{new}} \ [\text{ISP}-[\text{SPEC}-\text{BODY}]]$$

$$\frac{spred_{old} = (\mathtt{p}(\mathtt{v}^*) \equiv \bigvee_{i=1}^{n}(\exists u_i^* \cdot \hat{\sigma}_i \mid C_i)) \qquad \rho = [inv_p(\mathtt{v}^*) \mapsto \mathtt{fix}(\bigvee_{i=1}^{n} \exists u_i^* \cdot C_i)] \quad \mathcal{I} = \{(L \rightarrow hull(\bigvee_{i \in L} \exists u_i^* \cdot \rho C_i) \mid \emptyset \subset L \subset \{1..n\}\} \cup \{\{1..n\} \rightarrow \rho(inv_p(\mathtt{v}^*))\} \quad spred_{new} = (\mathtt{p}(\mathtt{v}^*) \equiv \bigvee_{i=1}^{n}(\exists u_i^* \cdot \hat{\sigma}_i \mid \rho C_i);\ \mathcal{I})}{spred_{old} \ \text{–isp}\rightarrow\ spred_{new}} \ [\text{ISP}-[\text{BUILD}-\text{INV}-\text{FAMILY}]]$$

$$\frac{spred_{old} = (\mathtt{p}(\mathtt{v}^*) \equiv \bigvee_{i=1}^{n}(\exists u_i^* \cdot \hat{\sigma}_i \mid C_i);\ \mathcal{I}) \qquad G = \bigcup_{i=1}^{n} closure(\mathcal{I}(\{i\})) \quad \mathcal{R} = \bigcup_{\alpha \in G}\{\alpha \leftarrow \{i \mid 1 \leq i \leq n \wedge \mathcal{I}(\{i\}) \Longrightarrow \alpha\}\} \quad \forall i \in \{1,..,n\} \cdot \hat{\sigma}_i \mid C_i \ \text{–sf}\rightarrow^{*}\ \hat{\sigma}_{i,2} \mid C_{i,2} \quad spred_{new} = (\mathtt{p}(\mathtt{v}^*) \equiv \bigvee_{i=1}^{n}(\exists u_i^* \cdot \hat{\sigma}_{i,2} \mid C_{i,2});\ \mathcal{I};\ \mathcal{R})}{spred_{old} \ \text{–isp}\rightarrow\ spred_{new}} \ [\text{ISP}-[\text{BUILD}-\text{PRUNE}-\text{COND}]]$$

where –sf→* *is the transitive closure of* –sf→; *and* $\mathcal{I}(\{i\}) = \pi_i$, *given* $(\{i\} \rightarrow \pi_i) \in \mathcal{I}$.

Fig. 5. Inference Rules for Specializable Predicates

The rule [ISP–[SPEC–BODY]] converts the body of each predicate definition into its specialized form. For each recursive invocation, it will initially assume a symbolic invariant, named $inv_{\mathtt{p}}(\mathtt{v}^*)$, without providing any pruning conditions. This immediately puts each recursive predicate instance in the fully-specialized form.

After the body of the predicate definition has been specialized, we can proceed to build a constraint abstraction for its predicate's invariant, denoted by $inv_{\mathtt{p}}(\mathtt{v}^*)$, in the [ISP–[BUILD–INV–FAMILY]] rule. For example, we may denote the invariant of predicate $\mathtt{lseg}(\mathtt{x},\mathtt{p},\mathtt{n})$ symbolically using $inv_{\mathtt{lseg}}(\mathtt{x},\mathtt{p},\mathtt{n})$, before building the following recursive constraint abstraction:

$$inv_{\mathtt{lseg}}(\mathtt{x},\mathtt{p},\mathtt{n}) \equiv \mathtt{x}{=}\mathtt{p} \wedge \mathtt{n}{=}\mathtt{0} \ \vee \ \exists \mathtt{q},\mathtt{m} \cdot \mathtt{x} \neq \mathtt{null} \wedge \mathtt{n}{=}\mathtt{m}{+}\mathtt{1} \wedge inv_{\mathtt{lseg}}(\mathtt{q},\mathtt{p},\mathtt{m})$$

If we apply a classical fix-point analysis to the above abstraction, we would obtain a closed-form formula as the invariant of the `lseg` predicate, that is $inv_{\mathtt{lseg}}(\mathtt{x},\mathtt{p},\mathtt{n}) = \mathtt{n} \geq \mathtt{0}$. With this predicate invariant, we can now build a family of invariants for each proper subset L of disjuncts, namely $\mathtt{0} \subset L \subset \{\mathtt{1..n}\}$. This is done with the help of the convex hull approximation. The size of this family of invariants is exponential to the number of disjuncts. While this is not a problem for predicates with a small number of disjuncts, it could pose a problem for unusual predicates with a large number of disjuncts. To circumvent this problem, we could employ either a lazy construction technique or a more aggressive approximation to cut down on the number of invariants generated. For simplicity, this aspect is not considered in the present paper.

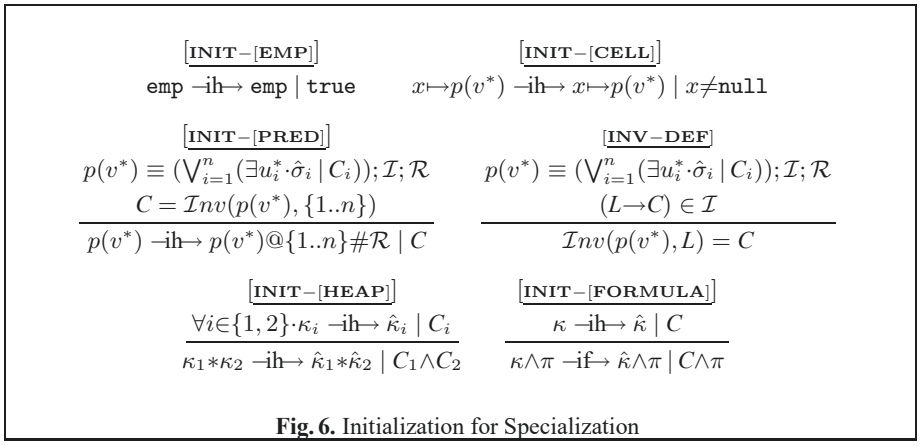

Fig. 6. Initialization for Specialization

Our last step is to build a set of pruning conditions for the disjunctive predicates using the [ISP−[BUILD−PRUNE−COND]] rule. This is currently achieved by applying a closure operation over the invariant $\mathcal{I}(\{\mathtt{i}\})$ for each of the disjuncts. To obtain a more complete set of pruning conditions, we are expected to generate a set of strong atomic constraints for each of the closure operations. For example, if we currently have a formula $\mathtt{a}{>}\mathtt{b} \wedge \mathtt{b}{>}\mathtt{c}$, a strong closure operation over this formula may yield the following set of atomic constraints $\{\mathtt{a}{>}\mathtt{b}, \mathtt{b}{>}\mathtt{c}, \mathtt{a}{>}\mathtt{c}{+}1\}$ as pruning conditions and omit weaker atomic constraints, such as $\mathtt{a}{>}\mathtt{c}$.

Definition 10 (Sound invariant and sound pruning condition) *Given a predicate definition* $p(v^*) \equiv \bigvee_{i=1}^{n} \mathtt{D_i};\ \cdots$*:*

(1) an invariant $L{\rightarrow}\pi$ *is said to be sound w.r.t. the predicate* p *if (1.a)* $\emptyset \subset L \subseteq \{1,..,n\}$*, and (1.b)* $p(v^*)@L\#\_ \models \pi$*.*

(2) a family of invariants $\mathcal{I}$ *is sound if every invariant from* $\mathcal{I}$ *is sound and the domain of* $\mathcal{I}$ *is the set of all non-empty subsets of* $\{1,..,n\}$*;*

(3) a pruning condition $(\alpha{\leftarrow}L)$ *is sound w.r.t. the predicate* p *if (3.a)* $\emptyset \subset L \subseteq \{1,..,n\}$*, (3.b)* $vars(\alpha) \subseteq \{v^*\}$*, and (3.c)* $\forall i{\in}L{\cdot}\mathtt{D_i} \models \alpha$*.*

(4) a set of pruning conditions R *is sound if every pruning condition in* R *is sound w.r.t. the predicate* p*.*

Property 5 *For each predicate* $p(v^*)$*, the family of invariants and the set of pruning conditions derived for* p *by our inference process are sound, assuming the fixpoint analysis and the hulling operation used by the inference are sound.*

A proof of this property can be found in [3].

6 Experiments

We have built a prototype system for our specialization calculus inside an existing program verification system for separation logic, called HIP [14]. Our implementation benefits greatly from two optimizations: *memoization* and *incremental pruning*. The key here is to support the early elimination of infeasible states, by attempting a proof

Programs (specified props)	LOC	HIP	HIP+Spec	HIP			HIP+Spec		
		Time(s)	Time(s)	Count	Disj	Size	Count	Disj	Size
17 small progs (size)	87	0.86	0.80	229	1.63	12.39	612	1.13	2.97
Bubble sort (size,sets)	80	2.20	2.23	296	2.13	18.18	876	1.09	2.79
Quick sort (size,sets)	115	2.43	2.13	255	3.29	17.97	771	1.27	3.08
Merge sort (size,sets)	128	3.10	2.15	286	2.04	16.74	1079	1.07	2.99
Complete (size,minheight)	137	5.01	2.94	463	3.52	43.75	2134	1.11	10.10
AVL (height, size,bal)	160	64.1	16.4	764	2.90	85.02	6451	1.07	9.66
Heap Trees (size, maxelem)	208	14.9	4.62	649	2.10	56.46	2392	1.02	8.68
AVL (height, size)	340	27.5	13.1	704	2.98	70.65	7078	1.09	10.74
AVL (height, size, sets)	500	657	60.7	1758	8.00	86.79	14662	1.91	10.11
Red Black (size, height)	630	25.2	15.6	2225	3.84	80.91	7697	1.01	3.79

Fig. 7. Verification Times and Proof Statistics (Proof Counts, Avg Disjuncts, Avg Size)

of each atomic constraint α being in contradiction with a given context C. In this way, the context C is allowed to evolve to a monotonically stronger context C_1, such that $C_1 \Longrightarrow C$. Hence, if indeed $C \Longrightarrow \neg\alpha$ is established, we can be assured that $C_1 \Longrightarrow \neg\alpha$ will also hold. This monotonic context change is the basis of the memoization optimization that leads to reuse of previous outcomes of implications and contradictions.

More specifically, we maintain a memoization set, I, for each context C. This denotes a set of atomic constraints that are implied by the context C; that is $\forall\alpha\in I \cdot C \Longrightarrow \alpha$. Contradictions of the form $(C\wedge\alpha) = \texttt{false}$ are also memoized in the same way, since we can model it as an implication check $C \Longrightarrow \neg\alpha$. These memoization recalls are only sound approximations of the corresponding implication checks. In case both membership tests fail for a given pruning condition α, we could turn to automated provers (as a last resort) to help determine $C \Longrightarrow \neg\alpha$. Memoization would, in general, help minimize on the number of invocations to the more costly provers.

The early elimination of infeasible states has an additional advantage. We can easily *slice out* relevant constraints from a (satisfiable) context C that is needed to prove an atomic constraint α. This is possible because we detect infeasible branches by proving only one atomic pruning constraint at a time. For example, consider a context $\texttt{x}\neq\texttt{null} \wedge \texttt{n}>0 \wedge \texttt{S}\neq\{\}$. If we need to prove its contradiction with $\texttt{n}=0$, a naive solution is to use $(\texttt{x}\neq\texttt{null} \wedge \texttt{n}>0 \wedge \texttt{S}\neq\{\}) \Longrightarrow \neg(\texttt{n}=0)$. A better solution is to slice out just the constraint $\texttt{n}>0$, and then proceed to prove the contradiction using $\texttt{n}>0 \Longrightarrow \neg(\texttt{n}=0)$, leading to an incremental pruning approach that uses smaller proof obligations. To implement this optimization, we partition each context into sets of connected constraints. Two atomic constraints in a context C are said to be *connected* if they satisfy the following relation.

$$\texttt{connected}(\alpha_1,\alpha_2) \;\texttt{:-}\; (\mathit{vars}(\alpha_1)\cap\mathit{vars}(\alpha_2)) \neq \{\}$$
$$\texttt{connected}(\alpha_1,\alpha_2) \;\texttt{:-}\; \exists\alpha\in\texttt{C} \cdot \texttt{connected}(\alpha_1,\alpha)\wedge\texttt{connected}(\alpha_2,\alpha)$$

Using this relation, we can easily slice out a set of constraints (from the context) that are connected to each pruning condition.

Fig 7 summarizes a suite of programs tested which included the 17 small programs (comprised of various methods on singly, doubly, sorted and circular linked lists, selection-sort, insertion-sort and methods for handling heaps, and perfect trees). Due to similar outcomes, we present the average of the performances for these 17 programs.

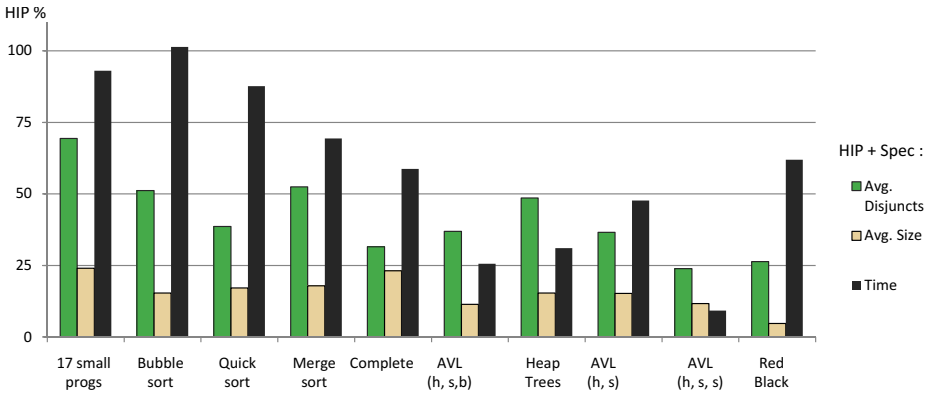

Fig. 8. Characteristic (disjunct, size, timing) of HIP+Spec compared to the Original HIP

We also experimented with a set of medium-sized programs that included complex shapes and invariants, to support full functional correctness. We measured the verification times taken for the original `HIP` system, and also the enhanced system, called `HIP+Spec`, with predicate specialization. For the suite of simple programs, the verifier with specializer runs about 7% faster. For programs with more complex properties (with the exception of bubble sort), predicate specialization manages to reduce verification times by between 12% and 90%. These improvements were largely due to the presence of smaller formulae with fewer disjuncts, as captured in Fig 8. This graph compares the characteristics (e.g. average disjuncts, sizes and timings) of formula encountered by `HIP+Spec`, as a percentage relative to the same properties of the original `HIP` system. For example, the average number of disjuncts per proof encountered went down from 3.2 to 1.1; while the size of each proof (based on as the number of atomic formulae) also decreased from an average of 48.0 to 6.5. This speed-up was achieved despite a six fold increase in the proof counts per program from 763 to 4375 used by the specialization and verification processes. We managed to achieve this improvement despite the overheads of a memoization mechanism and the time taken to infer annotations for specializable predicates. We believe this is due to smaller and simpler proof obligations generated with the help our specialization process.

We also investigated the effects of various optimizations on the specialization mechanism. Memoizing implications and contradictions saves 3.47%, while memoizing each context for state change saves 22.3%. For incremental pruning, we have utilized the slicing mechanism which saves 48% on average. We have not yet exploited the incremental proving capability based on strengthening of contexts since our current solvers, Omega and MONA, do not support such a feature. These optimizations were measured separately, with no attempt made to study their correlation. For an extended version of the present paper, including further experimental details, cf. [3].

7 Related Work and Conclusion

Traditionally, specialization techniques [8, 17, 11] have been used for code optimization by exploiting information about the context in which the program is executed.

Classical examples are the partial evaluators based on binding-time analysers that divide a program into static parts to be specialized and dynamic parts to be residualized. However, our work focusses on a different usage domain, proposing a predicate specialization for program verification to prune infeasible disjuncts from abstract program states. In contrast, partial evaluators [8] use unfolding and specialised methods to propagate static information. More advanced partial evaluation techniques which integrate abstract interpretation have been proposed in the context of logic and constraint logic programming [18, 17, 11]. They can control the unfolding of predicates by enhancing the abstract domains with information obtained from other unfolding operations. Our work differs in its focus on minimizing the number of infeasible states, rather than on code optimization. This difference allows us to use techniques, such as memoization and incremental pruning, that were not previously exploited for specialization.

SAT solvers usually use a conflict analysis [22] that records the causes of conflicts so as to recognize and preempt the occurrences of similar conflicts later on in the search. Modern SMT solvers (e.g. [15, 6]) use analogous analyses to reduce the number of calls to underlying theory solvers. Compared to our pruning approach, conflict analysis [22] is a backtracking search technique that discovers contexts leading to conflicts and uses them to prune the search space. These techniques are mostly complementary since they did not consider predicate specialization, which is important for expressive logics.

The primary goal of our work is to provide a more effective way to handle disjunctive predicates for separation logic [14, 13]. The proper treatment of disjunction (to achieve a trade-off between precision and efficiency) is a key concern of existing shape analyses based on separation logic [5, 10]. One research direction is to design parameterized heap materialization mechanisms (also known as focus operation) adapted to specific program statements and to specific verification tasks [21, 12, 20, 1, 16]. Another direction is to design partially disjunctive abstract domains with join operators that enable the analysis to abstract away information considered to be irrelevant for proving a certain property [7, 23, 2]. Techniques proposed in these directions are currently orthogonal to the contribution of our paper and it would be interesting to investigate if they could benefit from predicate specialization, and vice-versa.

Conclusion. We have proposed in this paper a specialization calculus for disjunctive predicates in a separation logic-based abstract domain. Our specialization calculus is proven sound and is terminating. It supports symbolic pruning of infeasible states within each predicate instance, under monotonic changes to the program context. We have designed inference techniques that can automatically derive all annotations required for each specializable predicate. Initial experiments have confirmed speed gains from the deployment of our specialization mechanism to handle separation logic specifications in program verification. Nevertheless, our calculus is more general, and is useful for program reasoning over any abstract domain that supports disjunctive predicates. This modular approach to verification is being enabled by predicate specialization.

Acknowledgement We thank the reviewers for insightful feedback. This work is being supported principally by MoE research grant R-252-000-411-112.

References

1. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, pages 178–192, 2007.

2. B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.
3. W.N. Chin, C. Gherghina, R. Voicu, Q.L. Le, F. Craciun, and Qin S.C. A specialization calculus for pruning disjunctive predicates to support verification. Technical report, School of Computing, National University of Singapore, 2011, `http://loris-7.ddns.comp.nus.edu.sg/~project/hippruning/`.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM POPL*, pages 238–252, 1977.
5. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.
6. B. Dutertre and L. M. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, pages 81–94, 2006.
7. B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *ACM PLDI*, pages 256–265, 2007.
8. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
9. N. Klarlund and A. Moller. MONA Version 1.4 - User Manual. BRICS Notes Series, January 2001.
10. V. Laviron, B.-Y. Evan Chang, and X. Rival. Separating shape graphs. In *ESOP*, pages 387–406, 2010.
11. M. Leuschel. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Trans. Program. Lang. Syst.*, 26(3):413–463, 2004.
12. R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In *TACAS*, pages 3–18, 2007.
13. H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *CAV*, pages 355–369, 2008.
14. 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, January 2007.
15. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.
16. A. Podelski and T. Wies. Counterexample-guided focus. In *ACM POPL*, pages 249–260, 2010.
17. G. Puebla and M. Hermenegildo. Abstract specialization and its applications. In *ACM SIGPLAN PEPM*, pages 29–43, 2003.
18. G. Puebla, M. Hermenegildo, and J. P. Gallagher. An integration of partial evaluation in a generic abstract interpretation framework. In *ACM SIGPLAN PEPM*, pages 75–84, January 1999.
19. W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.
20. N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. In *ESOP*, pages 220–236, 2007.
21. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
22. J. P. Marques Silva and K. A. Sakallah. GRASP—a new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, 1996.
23. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.