Shape Analysis by Augmentation, Abstraction, and Transformation

by

Ittai Balaban

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 2007

Amir Pnueli and Lenore D. Zuck — Advisers

To Mor, to my parents, and to Rotem and her friends Winnie & the Pooh

Acknowledgements

I would like to thank Amir Pnueli and Lenore Zuck, who have tried their best to teach me how to think and express myself rationally (and have not despaired as of yet).

Thanks go to Dennis Dams and Kedar Namjoshi, for numerous illuminating discussions. In addition, Stephen Fink has my thanks for pointing out problems with the abstraction refinement algorithm that led directly toward an improvement to the method.

I wish to thank my thesis committee, which, in addition to Amir, Lenore, Dennis, and Kedar, includes Clark Barrett, Ken McMillan, and Benjamin Goldberg. I thank them for their constructive comments and penetrating questions, some of which resulted in followup research.

Finally I wish to thank the numerous anonymous reviewers of the [BPZ05, BPZ07b] and [BPZ07a] paper submissions, which form the bulk of this dissertation. There is nothing better for improving one's work than insightful criticism as to how bad it is $\ddot\smile$

Abstract

The goal of *shape analysis* is to analyze properties of programs that perform destructive updating on dynamically allocated storage (heaps). In the past decade various frameworks have been proposed, most notable being the line of work based on *shape graphs* and *canonical abstraction* [SRW99, LAS00]. Frameworks have been proposed since, among them based on counter automata, predicate abstraction, and separation logic. However, among these examples there has been little effort in dealing with liveness properties (e.g., termination) of systems whose verification depends on deep heap properties (a notable exception being [BCDO06]).

This dissertation presents a new shape analysis framework that is based on *predicate abstraction*, *program augmentation*, and *model checking*. The combination of predicate abstraction and program augmentation is collectively known as *Ranking Abstraction*, and provides a sufficiently powerful model for verification of liveness properties of sequential and concurrent systems. Furthermore, a new predicate abstraction method is presented, that allows for automatic computation of abstract systems that does not rely on theorem provers. This approach has several intrinsic limitations, most notably on the class of analyzable heap shapes. Thus several extensions are described that allow for complex heap shapes.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of *shape analysis* is to analyze properties of programs that perform destructive updating on dynamically allocated storage (heaps) [JM81]. Programs manipulating heap structures can be viewed as *parameterized* in the number of heap nodes, or, alternatively, the memory size.

This dissertation presents an approach for shape analysis based on *predicate abstraction* ([GS97]) and *model checking* ([CE81, EC80]) that allows for verification of functional specifications. The abstraction used does *not* require any abstract representation of the heap nodes (e.g. shape graphs), but rather, requires only reachability relations between the program variables.

Predicate abstraction has become one of the most successful methodologies for proving safety properties of programs. However, with no extension it cannot be used to verify general liveness properties. Therefore, in the framework presented here, predicate abstraction is complemented by *rank-*

ing abstraction, a method for verification of both safety and progress properties. Ranking abstraction is derived from *augmented finitary abstraction* which was introduced in [KP00]. It is based on an augmentation of the concrete program that is parameterized by a set of well-founded ranking functions. Based on these, new *compassion* (strong fairness) requirements as well as transitions are generated, all of which are synchronously composed with the program in a non-constraining manner. Unlike most methodologies, the ranking functions are not expected to decrease with each transition of the program.

The dissertation is organized as follows: After the computational model and formal background are defined in Chapter 2, Chapters 3 and 4 present the method of ranking abstraction and its embedding in an abstraction refinement framework, in a domain-neutral manner. Chapter 5 then presents an algorithm for extracting a deductive proof using a model-checker, once a successful verification effort (using ranking abstraction) has been carried out. Chapters 6 and 7 deal with shape analysis: Chapter 6 presents a method based on ranking abstraction that is fully automatic for *singly-linked* heaps – heaps where nodes have at most one outgoing edge. In Chapter 7 this limitation is overcome via reductions from complex heap shapes to singly-linked heaps.

1.1 Abstraction-Aided Verification

Deductive verification is a method of program verification that incrementally constructs proofs until the desired conclusion, a proof that the system meets its specification, is obtained. Its main advantage is that it allows verification of *infinite-state* systems. Deductive verification is often manual, and, like all deductive proofs, requires considerable human skill and time. The method generally involves two activities: **(1)** Devising auxiliary constructs (e.g., ranking functions and auxiliary assertions), and **(2)** Deciding validity of assertions known as verification conditions. The former involves creative effort, while the latter is typically characterized by mechanical effort and thus, while error-prone for humans, is a likely candidate for automation. The predicate abstraction framework makes such a distinction: The creative effort of devising ingredients (in the form of predicates) for an auxiliary construct is delegated to the user, while the task of checking if they combine to form an inductive assertion is relegated to the machine (e.g., a model checker).

In [DGG00] Dams, Gerth, and Grumberg point out the duality between activities in the verification of safety, and progress of programs. With *Ranking Abstraction* we present a dual to the predicate abstraction framework that similarly delegates to the user the task of devising ranking functions, while automating the effort of checking that they indeed combine to form an adequate global ranking function. The duality between predicate and ranking abstraction is expressed through the components given in Table 1.1.

Role	**Predicate Abstraction**	**Ranking Abstraction**
Initial abstraction	Heuristics to choose a predicate base	Heuristics to choose a predicate base and a *ranking core* (set of well-founded ranking functions)
Handling spurious counterexamples	Predicate refinement	Predicate refinement or ranking refinement
Automatic generation of deductive proof constructs	Inductive invariants as a boolean combination of predicates	Global concrete ranking function as a lexicographical combination of the core ranking functions

Table 1.1: Comparison Between Components of Predicate and Ranking Abstraction

1.2 Shape Analysis

Our approach to modeling dynamic memory is based on an idealized model that represents the heap as a graph, or *linked structure*, that is mutated throughout the computation of a program by deletion or creation of edges, or *links*. Thus a system state is given by an interpretation of a set of variables, together with a graph *shape*. In our approach, states are abstracted using a predicate base that contains reachability relations among program variables pointing into the heap. The computation of the abstract states and transition relation is precise and automatic and does not require the use of a theorem prover. Rather, we use a small model theorem to identify a truncated (small) finite-state version of the program whose abstraction is identical to the abstraction of the unbounded-heap version of the same program. The abstraction of the finite-state version is then computed by BDD techniques.

For proving properties of system behavior over time, the original system is augmented by a well-founded ranking function, which is then abstracted together with the system. Well-foundedness is abstracted into strong fairness (compassion). We show that, for a restricted class of programs (that still includes numerous interesting cases), the small model theorem can be applied to this joint abstraction.

The small model theorem that is at the basis of the abstraction method applies to a restricted first order logic with transitive closure, one that is expressive enough for describing mutation of singly-linked structures. Using a technique known as *structure simulation*, we show that this restriction can be lifted to represent heaps with multiple links in which *sharing* (where a node has multiple incoming edges) is disallowed. Here, graph shapes such as trees are represented by sets of singly-linked lists, with list edges representing reversed tree edges. This allows for the verification of a wide array of algorithms over trees.

Moving to yet more complex shapes, we consider structures that can be partitioned into a hierarchy of heaps, where each heap on its own is singly-linked or multi-linked and sharing-free, but in addition contains links pointing to nodes of heaps that are lower in the hierarchy. This model allows algorithms that manipulate some "main" data structure, yet contain auxiliary data structures with pointers into the "main" structure, an example of which is a graph traversal algorithm that maintains an auxiliary stack or a queue.

1.3 Deriving Proofs from Abstractions

Model checking ([CE81, EC80]) is an automatic process for verifying temporal properties of finite-state systems. In this process, when executions violating the property exist, at least one is reported and serves as a counterexample. When the search for counterexamples fails one may conclude that the system satisfies its specification. However, our confidence in such a positive conclusion is tarnished by two possible factors:

- Due to complexity and decidability, the system being checked is often an oversimplification of the actual system. Hence, failure to find counterexamples for it does not necessarily imply that the actual system is fault free.

- There always exists the possibility that the model checker itself contains a bug causing it to report success, while the system is faulty.

Both these risks may cause us to treat with diffidence a result which purely claims success without providing some supporting evidence, a “witness” or “certificate” that the property does indeed hold over the considered system. This ‘proof by lack of counterexample’ is the main drawback of the model checking approach; some would even say that model checking is a tool for falsification rather than a tool for verification. An alternative approach to model checking, deductive verification, while often manual, has the benefit that it often explains *why* the system satisfies its specification.

In [PZ01a, PPZ01a] Peled, Pnueli, and Zuck enhance the model checking process of Linear Time Temporal Logic (LTL) with the ability to automatically generate a deductive proof that the system meets its LTL specification, thus emphasizing the point of view that model checking can be used to *justify* why the system actually works. They show that, by exploiting the information in the graph that is generated during the search for counterexamples, they generate a fully deductive proof that the system meets its specification. The generated deductive proof can then be sent to a theorem prover to verify its validity.

The work in [PZ01a, PPZ01a] is restricted to finite-state systems. [Nam01] expanded these results to proof of the μ-calculus. One issue left unresolved is that of extending the method to deal with infinite-state systems. In this dissertation this issue is addressed for the class of specifications embodied by response properties. A response property is typically used to specify that for any computation of a system, if infinitely many states satisfying some logical property p, then infinitely many states must satisfy some logical property q. An example of such a requirement is "Any process that continually requests access to a shared resource, must eventually be granted access to said resource." The method presented here extracts a proof from the control-flow graph of the (finite-state) abstract system, once the process of ranking abstraction has been applied successfully. This is done in a fully automatic manner. In other words, once we verify a system using the ranking abstraction method, we are able to produce a certificate, in the form of a proof, of the correctness of the system.

1.4 Implementation and Experiments

The shape analysis framework of Chapter 6 is illustrated on two examples, both using (singly) linked lists: List reversal and in-place sort. It is shown how various predicate abstractions can be used to establish various safety properties, and how, for each program, one of the abstractions can be augmented with a progress monitor to establish termination. The extensions of the framework in Chapter 7 are demonstrated on AVL tree insertion and graph traversal algorithms. These illustrate how multi-linked structures of varying complexity are represented within the restrictions of the basic framework. All examples have been implemented using the TLV programmable model checker [PS96].

1.5 Related Work

Ranking Abstraction In [PR05] Podelski and Rybalchenko extend predicate abstraction ([GS97]) by employing predicates over program transitions, rather than states. In this way, the abstraction preserves the argument for proving termination (general liveness is handled by a reduction to fair termination).

The body of work most comparable to ours is [CPR05], where Cook, Podelski, and Rybalchenko present a framework for verifying termination, which formalizes dual refinements – of transition predicate abstraction and of transition invariants [PR04b]. A transition invariant can be described as a union of transition relations. An important element common to the two methods is the modularity that decomposes the task of proving well-foundedness (absence of infinite executions) into several ingredients that are analyzed separately. In the transition invariant method, this is done by decomposing the invariant into separate transition relations, each of which is independently well-founded. In our case, we construct a separate progress monitor for each ranking function component.

Previous work on termination analysis of logic programs, notably of Lindenstrauss and Sagiv ([LS97]) and Codish and Taboch ([CT99]), applies concepts similar to transition invariants. Moreover, The abstract domain introduced in [CT99] was specialized in ([LJBA01]) into what was termed *size change graphs*, the purpose of both being automatic termination analysis using abstract interpretation. In these lines of work, the formal justification of the termination analyses is similar to what is termed *disjunctive well-foundedness* in [PR04b]. Further work in [BCG$^+$07] uses a similar notion in its decomposition of the termination argument into *size norms*.

Comparable to our abstraction/ranking refinement algorithm, the algorithm in [CPR05], when presented with an abstract counterexample, analyzes the cause of its "spuriousness", and refines either the predicate abstraction or the transition invariant. The method has been implemented in the Terminator tool, and applied to practical examples (e.g., Windows device drivers) in [CPR06]. While our framework is inherently applicable to systems with (weak and strong) fairness constraints (e.g., concurrent systems), the framework as presented in [CPR05] lacks any notion of fairness. Therefore, [PPR05, CGP$^+$07] extend it to allow for fairness requirements.

Dams, Gerth, and Grumberg [DGG00] point out the duality between verification of safety and progress of programs. Like us, they aim to lift this duality to provide tools for proving progress properties, whose functionality is analogous to similar tools used for safety. Specifically, they propose a heuristic for discovering ranking functions from a program's text. In contrast, we concentrate on an analogy with predicate abstraction, a particular method for safety. Our approach is broader, however, in that we suggest a general framework for safety and progress properties where each of the activities in a verification process has an instantiation with respect to each of the dualities.

In [PR04a] Podelski and Rybalchenko present a method for synthesis of linear ranking functions. The method is complete for unnested loops, and is embedded successfully in a broader framework for proving liveness properties [PR03], as well as in [CPR05]. This method is one of several candidates that can be embedded in our framework.

The topic of refinement of state abstraction, specifically predicate abstraction, has been widely studied. A number of existing works in this area are [CGJ$^+$00, BR01, BPR02].

Proof Extraction There have been several works dealing with the extraction of proofs from successful application of a model checking run. The papers [PZ01b] and [PPZ01b] show how to construct a deductive temporal proof out of a successful run of model for checking the property of interest. Namjoshi in [Nam01] argues for the need of a certificate that confirms the correctness of a successful model checking run. This certificate can be viewed as a deductive proof of the property, where the proof is presented as an automaton similar to the verification diagrams of [MP94]. In a similar way, [KV05] show how to construct an automaton certificate that confirms the correctness of a model checked property, and can be checked automatically and efficiently. However, all of these methods were applied to propositional properties of finite-state systems and produced proofs (or certificates) that were propositional in nature. None of them could produce a proof that included a ranking function over an unbounded domain. Namjoshi in [Nam03] realizes the need to translate a proof of a finite-state abstraction of an infinite-state system. He refers to this proof concretization process as *lifting of the proof.* In this process, he takes into account the need to deal with ranking functions. The limitation of his method was that the lifting of ranking functions necessarily preserve the range of the functions. Starting from a finite-state system, the range of the abstract ranking functions and, therefore, the resulting range of the concrete ranking functions is necessarily bounded. In comparison, the methods presented here in Chapter 5 extract ranking functions over unbounded domains from their finitary representation as compassion requirements.

Shape Analysis The work in [SRW02] presents a parametric framework for shape analysis that deals with the specification language of the shape analysis framework and the construction of the shape analyzer from the specification. A 2-value logic is used to represent concrete stores, and a 3-valued logic is used to represent abstract stores. Properties are specified by first-order formulae with transitive closure; these also describe the transitions of the system. The shape analyzer computes a fixed point of the set of equations that are generated from the analysis specification. The systems considered in [SRW02] are more general than ours, e.g., we allow at most one "next pointer" for each node. Due to the restricted systems and properties we consider, we do not have to abstract the heap structure itself, and therefore our computation of the transition relation is precise. Moreover, their work does not handle liveness properties.

In [DN03], Dams and Namjoshi study shape analysis using predicate abstraction and model checking. Starting with shape predicates and a property, the method iteratively computes weakest preconditions to find more predicates and constructs abstract programs that are then model checked. As in the [SRW02] framework, the abstraction computed in not precise. Some manual intervention is required to apply widening-like techniques and guide the system into convergence. This work, too, does not handle liveness.

Some related but less relevant works are [DDP99, DD01] that study concurrent garbage collection using predicate abstraction, [FQ02] that study loop invariants using predicate abstraction, and [Nel83] that calculates weakest preconditions for reachability. All these works do not apply shape analysis or use shape predicates.

Our shape analysis approach, which is based on predicate abstraction and model-checking, relies on a decidable logic with transitive closure to automatically compute abstractions. Comparable works include [RBH07, MYRS05, DN03] that, as well as [Rey02, BCDO06, BBH$^+$06], are less expressive then the present framework as they do not allow bi-directional traversal of lists, nor graph reachability under universal quantification. As such they are not easily adaptable to modeling trees, nor are they rich enough to express properties of data in structures. Our assertional language is in some ways weaker but generally incomparable to the logic of reachable patterns of [YRS$^+$06]. The class of graphs expressible in our logic is subsumed by the canonical abstraction framework of [SRW02].

The correspondence between tree structures and singly-linked structures is the basis of the proof of decidability of first-order logic with one function symbol in [BGG97]. More generally, the observation that complex data structures with regular properties can be reduced to simpler structures has been utilized in [KS93, IRR$^+$04b, MS01, WKL$^+$06]. However, it is not always straightforward to apply, and has not been applied in the context of predicate abstraction. Several assumptions that hold true in analysis of "conventional" programs over singly-linked heaps (e.g., C or Pascal programs), cannot be relied upon when reducing trees to lists. For example, the number of roots of the heap is no longer bounded by the number of program variables.

Furthermore, the cited works typically use trees to simulate more complex structures, while our starting point is lists, which are used to simulate tree-like structures.

The use of path compression in heaps to prove small model properties of logics of linked structures, has been used before, e.g., in [BRS99] and more recently in [YRS+06]. Our work on parameterized systems relies on a small model theorem for checking inductiveness of assertions. The small model property there is similar to the one here with respect to *stratified data*. However, with respect to unstratified data (such as graphs), the work on parameterized systems suggests using logical instantiation as a heuristic (see, e.g., [APR+01]), whereas here completeness is achieved using graph-theoretic methods.

The idea of modeling composite structures using a static hierarchical separation (here referred to as *cascading heap structures*) has been used by [RS01] to perform interprocedural analysis of recursive heap-manipulating programs. The idea is to model the recursion stack as a linked structure with pointers into the main heap structure.

Chapter 2

Preliminaries

In this chapter we present our computational model, as well as the method of predicate abstraction. In the following, we refer to a first-order state formula as an *assertion*. We denote by $\vec{y}$ a set of variables, and write $\varphi(\vec{y})$ to denote that $\vec{y}$ is the set of free variables of an assertion φ. $\varphi(\vec{y})$ is said to be a *consistent* assertion if there exists an assignment $\overline{\eta}$ to $\vec{y}$ such that $\overline{\eta} \models \varphi$.

2.1 Fair Discrete Systems

As our computational model, we take a *fair discrete system* (FDS) [KPR98]. This generalizes the model of *fair transition systems* [MP95] by allowing a more general form of fairness requirements. An FDS is presented by a tuple $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

- V — A set of *system variables.* A *state* of $\mathcal{D}$ provides a type-consistent interpretation of the variables V. For a state s and a system variable $v \in V$, we denote by $s[v]$ the value assigned to v by the state s. Let Σ denote the set of all states over V.

- Θ — The *initial condition*: An assertion characterizing the initial states.
- $\rho(V, V')$ — The *transition relation*: An assertion, relating the values V of the variables in state $s \in \Sigma$ to the values V' in an $\mathcal{D}$-successor state $s' \in \Sigma$. We assume that every state has a ρ-successor.
- $\mathcal{J}$ — A set of *justice* (*weak fairness*) requirements (assertions); A computation must include infinitely many states satisfying each of the justice requirements.
- $\mathcal{C}$ — A set of *compassion (strong fairness)* requirements: Each compassion requirement is a pair $\langle p, q\rangle$ of state assertions; A computation should include either only finitely many p-states, or infinitely many q-states.

Definition 2.1 (Path)**.** *A* path *in an* FDS $\mathcal{D}$ *is a possibly infinite sequence of states* $\sigma : s_0, s_1, \ldots$ *that are* consecutive*, meaning that for each* $\ell = 0, 1, \ldots$*, the state* $s_{\ell+1}$ *is a* $\mathcal{D}$*-successor of* s_ℓ*. That is,* $\langle s_\ell, s_{\ell+1}\rangle \models \rho(V, V')$ *where, for each* $v \in V$*, we interpret* v *as* $s_\ell[v]$ *and* v' *as* $s_{\ell+1}[v]$.

Let ψ be an assertion and $s, s' \in \Sigma$ be states of an FDS. We say that s is a ψ-state if $s \models \psi$. s' is said to be *reachable* from s if there exists a finite path $\sigma : s = s_0, \ldots, s_k = s'$, for some $k \geq 0$. If s' is reachable from a Θ-state, it is simply called a *reachable state.* A path is said to be ψ*-free* if no state in the path is a ψ-state.

Definition 2.2 (Run). *A* run *of an* FDS $\mathcal{D}$ *is an* initialized *path* $\sigma : s_0, s_1, \ldots,$ *meaning that* $s_0 \models \Theta$*.*

Definition 2.3 (Computation). *A* computation *of an* FDS $\mathcal{D}$ *is an infinite run that satisfies*

- Justice — *for every* $J \in \mathcal{J}$*,* σ *contains infinitely many occurrences of* J*-states.*
- Compassion – *for every* $\langle p, q\rangle \in \mathcal{C}$*, either* σ *contains only finitely many occurrences of* p*-states, or* σ *contains infinitely many occurrences of* q*-states.*

We denote by $\mathcal{C}omp(\mathcal{D})$ the set of all computations of system $\mathcal{D}$.

Fair discrete systems are closed under two *composition operators* that are useful in describing concurrent systems. Fix two systems $\mathcal{D}_1 : \langle V_1, \Theta_1, \rho_1, \mathcal{J}_1, \mathcal{C}_1\rangle$ and $\mathcal{D}_2 : \langle V_2, \Theta_2, \rho_2, \mathcal{J}_2, \mathcal{C}_2\rangle$.

Definition 2.4 (Synchronous Parallel Composition). *The* synchronous parallel composition *of* $\mathcal{D}_1$ *and* $\mathcal{D}_2$*, denoted by* $\mathcal{D}_1 \mathbin{|||} \mathcal{D}_2$*, is defined as the system* $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$*, where*

$$\begin{array}{rclcrcl} V & = & V_1 \cup V_2 & \qquad & \mathcal{J} & = & \mathcal{J}_1 \cup \mathcal{J}_2 \\ \Theta & = & \Theta_1 \wedge \Theta_2 & & \mathcal{C} & = & \mathcal{C}_1 \cup \mathcal{C}_2 \\ \rho & = & \rho_1 \ \wedge \ \rho_2 & & & & \end{array}$$

As implied by the definition, each of the basic actions of a composed system $\mathcal{D}_1 \mathbin{|||} \mathcal{D}_2$ consists of the joint execution of an action of $\mathcal{D}_1$ and an action of $\mathcal{D}_2$. Thus, we can view the execution of $\mathcal{D}$ as the *joint execution* of $\mathcal{D}_1$ and $\mathcal{D}_2$.

Definition 2.5 (Asynchronous Parallel Composition)**.** *The* asynchronous parallel composition *of $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 \| \mathcal{D}_2$, is defined as the system $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where*

$$\begin{array}{rcl} V & = & V_1 \cup V_2 \\ \Theta & = & \Theta_1 \wedge \Theta_2 \\ \rho & = & \left(\begin{array}{ccc} & \rho_1 & \wedge & pres(V_2 - V_1) \\ \vee & \rho_2 & \wedge & pres(V_1 - V_2) \end{array} \right) \end{array} \qquad \begin{array}{rcl} \mathcal{J} & = & \mathcal{J}_1 \cup \mathcal{J}_2 \\ \mathcal{C} & = & \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}$$

The predicate $pres(U)$ stands for the assertion $U' = U$, implying that all the variables in U are preserved by the transition.

Asynchronous composition represents an *interleaving* model of concurrency, in which the system alternates (though not in a strict manner) between transitions of both components of the composition.

Programs

A *program* is a textual representation of an FDS using the notation of the *Simple Programming Language* (SPL), as proposed in [MP95]. A minimal SPL program consists of data definitions and a set of labeled statements. The FDS associated with a program has, in addition to variables declared in the data definitions, a *program counter* variable π, which ranges over the statement labels. The transition relation is derived from the program statements by translating variable assignments into a relation between primed and unprimed copies of the variables. Program control statements, such as

if or **while**, are handled as assignments to the program counter. For example, the program Nested-Loops in Fig. 3.1(a) is associated with the FDS $\langle V, \Theta, \rho, \emptyset, \emptyset \rangle$, where

$$
\begin{array}{lcl}
V & : & \{x, y : \mathbb{N}, \pi : [0 \ldots 6]\} \\
\Theta & : & x = y = \pi = 0 \\
\rho & : & \left\{ \begin{array}{ll}
 & \pi = 0 \ \wedge \ \pi' = 1 \ \wedge \ x' > 0 \ \wedge \ y' = y \\
\vee & \pi = 1 \ \wedge \ x > 0 \ \wedge \ \pi' = 2 \ \wedge \ x' = x \ \wedge \ y' = y \\
\vee & \pi = 1 \ \wedge \ x = 0 \ \wedge \ \pi' = 6 \ \wedge \ x' = x \ \wedge \ y' = y \\
\vee & \pi = 2 \ \wedge \ \pi' = 3 \ \wedge \ y' > 0 \ \wedge \ x' = x \\
\vee & \pi = 3 \ \wedge \ y > 0 \ \wedge \ \pi' = 4 \ \wedge \ x' = x \ \wedge \ y' = y \\
\vee & \pi = 3 \ \wedge \ y = 0 \ \wedge \ \pi' = 5 \ \wedge \ x' = x \ \wedge \ y' = y \\
\vee & \pi = 4 \ \wedge \ \pi' = 3 \ \wedge \ y' = y - 1 \ \wedge \ x' = x \\
\vee & \pi = 5 \ \wedge \ \pi' = 1 \ \wedge \ x' = x - 1 \ \wedge \ y' = y \\
\vee & \pi = 6 \ \wedge \ \pi' = 6 \ \wedge \ x' = x \ \wedge \ y' = y
\end{array} \right\}
\end{array}
$$

From here on, we will assume that the systems being verified are derived from programs. This implies that in every such system, control is specified via a program counter.

2.2 Temporal Specification of Systems

As the language for specifying properties of systems we use *linear-time temporal logic* (LTL) [MP91b].

Assume an underlying (first-order) assertion language. A *temporal formula* is constructed out of state formulas (assertions) to which we apply the boolean operators $\neg$ and $\vee$ and the basic temporal operators:

$\bigcirc$:	Next	Y:	Previous
$\mathcal{U}$:	Until	$\mathcal{S}$:	Since

Other temporal operators can be defined in terms of the basic ones as follows:

$\Diamond\, p$	$=$	$\textsc{True}\ \mathcal{U}\ p$:	Eventually
$\Box\, p$	$=$	$\neg \Diamond \neg p$:	Henceforth
$p\, \mathcal{W}\, q$	$=$	$\Box\, p \ \vee\ (p \mathcal{U} q)$:	Waiting-for, Unless, Weak Until
$\diamondminus\, p$	$=$	$\textsc{True}\ \mathcal{S}\ p$:	Sometimes in the past
$\boxminus\, p$	$=$	$\neg \diamondminus \neg p$:	Always in the past
$p\, \mathcal{B}\, q$	$=$	$\boxminus\, p \ \vee\ (p \mathcal{S} q)$:	Back-to, Weak Since

A *model* for a temporal formula p is an infinite sequence of states $\sigma : s_0, s_1, ...$ where each state s_j provides an interpretation for the variables of p.

Semantics of LTL

Given a model σ, we define the notion of a temporal formula p holding at a position $j \geq 0$ in σ, denoted by $(\sigma, j) \models p$:

- For an assertion p,

  $(\sigma, j) \models p \quad \iff \quad s_j \models p$

  That is, we evaluate p locally on state s_j.
- $(\sigma, j) \models \neg p \quad \iff \quad (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \quad \iff \quad (\sigma, j) \models p$ or $(\sigma, j) \models q$
- $(\sigma, j) \models \bigcirc p \quad \iff \quad (\sigma, j+1) \models p$
- $(\sigma, j) \models p \mathcal{U} q \quad \iff \quad$ for some $k \geq j, (\sigma, k) \models q$,

  and for every i such that $j \leq i < k$,

  $(\sigma, i) \models p$
- $(\sigma, j) \models \ominus p \quad \iff \quad j > 0$ and $(\sigma, j-1) \models p$
- $(\sigma, j) \models p \mathcal{S} q \quad \iff \quad$ for some $k \leq j, (\sigma, k) \models q$,

  and for every i such that $j \geq i > k, \quad (\sigma, i) \models p$

This implies the following semantics for the derived operators:

- $(\sigma, j) \models \Box p \quad \iff \quad (\sigma, k) \models p$ for *all* $k \geq j$
- $(\sigma, j) \models \Diamond p \quad \iff \quad (\sigma, k) \models p$ for *some* $k \geq j$

If $(\sigma, 0) \models p$ we say that p *holds over* σ and write $\sigma \models p$. Formula p is *satisfiable* if it holds over some model. Formula p is *(temporally) valid* if it holds over *all* models.

Formulas p and q are *equivalent*, denoted $p \sim q$, if $p \leftrightarrow q$ is valid. They are called *congruent*, denoted $p \approx q$, if $\Box(p \leftrightarrow q)$ is valid. If $p \approx q$ then p can be replaced by q in any context.

The *entailment* $p \Longrightarrow q$ is an abbreviation for $\Box(p \rightarrow q)$.

For an FDS $\mathcal{D}$ and an LTL formula φ, we say that φ is *$\mathcal{D}$-valid*, denoted $\mathcal{D} \models \varphi$, if all computations of $\mathcal{D}$ satisfy φ.

Classification of Formulas/Properties

A formula of the form $\Box\, p$ for some past formula p is called a *safety* formula. A formula of the form $\Box \Diamond\, p$ for some past formula p is called a *response* formula. An equivalent characterization is the form $p \Longrightarrow \Diamond\, q$. The equivalence is justified by

$$\Box(p \implies \Diamond\, q) \qquad \sim \qquad \Box \Diamond((\neg p)\, \mathcal{B}\, q)$$

Both formulas state that either there are infinitely many q's, or there are no p's, or there is a last q-position, beyond which there are no further p's.

A property is classified as a safety (resp. response) property if it can be specified by a safety (resp. response) formula.

Every temporal formula is equivalent to a conjunction of a *reactivity* formulas, i.e.

$$\bigwedge_{i=1}^{k}(\Box \Diamond\, p_i \ \vee\ \Diamond \Box\, q_i)$$

Hierarchy of the Temporal Properties

In Fig. 2.1 we present a hierarchy of the temporal properties. Every box in this diagram represents a class of properties together with the canonical formula corresponding to this class. The formulas p, p_i, q, q_i appearing in the canonical representations are arbitrary past formulas. Lines connecting the boxes in the diagram represent strict inclusion relations between the classes. Thus, the class of *safety properties* is strictly included in the class of *obligation properties*. This means that every safety property is also an obligation property, but there exists an obligation property which is not a safety property. Note that the obligation and reactivity classes each contain an internal strict hierarchy parameterized by k.

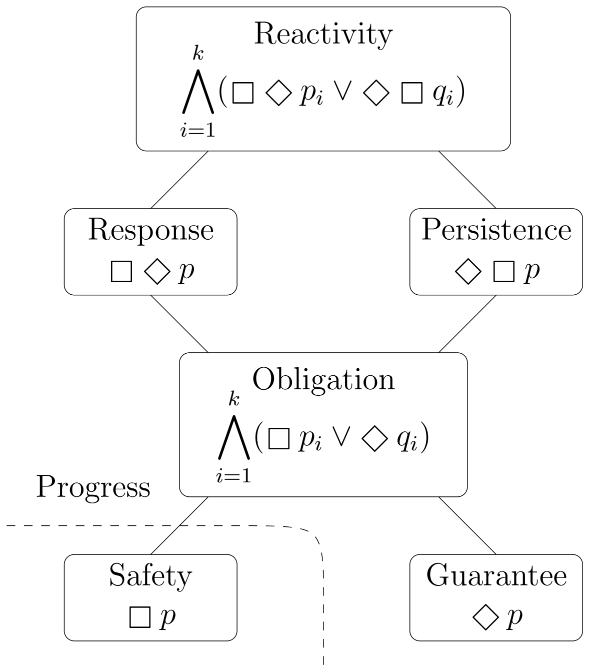


Figure 2.1: The Temporal Hierarchy of Properties

2.3 Ranking

A well-founded domain is a pair $(\mathcal{W}, \succ)$ such that $\mathcal{W}$ is a set and $\succ$ is a partial order over $\mathcal{W}$ that admits no infinite $\succ$-decreasing chains. A *ranking function* is a function mapping program states into some well-founded domain.

An assertion, like a transition relation, that refers to both unprimed and primed copies of the system variables is called a *bi-assertion*. A bi-assertion $\beta(V, V')$ is called *well founded over assertion p* if there does not exist an infinite sequence of states $s_0, s_1, \ldots$, such that $s_0 \models p$ and $\langle s_i, s_{i+1}\rangle \models \beta$, for every $i \geq 0$. If $p = 1$ (*true*), then we say simply that β is *well founded*.

In order to prove that β is well founded over p, it is sufficient [MP91a] to find an auxiliary assertion φ and a well-founded ranking δ, such that

$$p \rightarrow \varphi \qquad \text{and} \qquad \varphi(V) \ \wedge \ \beta(V, V') \rightarrow \varphi(V') \ \wedge \ \delta(V) \succ \delta(V')$$

In this case, we say that the well-founded ranking δ *proves the well foundedness of β over p.*

To illustrate our ability to verify liveness properties, we will demonstrate our proposed techniques on the class of *response properties* that have the form $p \Longrightarrow \Diamond\, q$ (abbreviating $\Box(p \rightarrow \Diamond\, q)$), where p and q are assertions. To verify such a property over a system, we use the notion of a *pending* state:

Definition 2.6 (Pending State)**.** *Let p and q be assertions. A state of an* FDS *D is said to be* pending *with respect to p and q if it is reachable by a q-free path from a reachable p-state.*

To verify a response property, it is sufficient to find a well-founded ranking δ such that δ decreases on every step that departs from a pending state.

2.4 Predicate Abstraction

The material here is a summary of [KP00]. We fix an FDS $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ whose set of states is Σ. We consider a set of *abstract variables* $V_A = \{u_1, \ldots, u_n\}$ that range over finite domains. An *abstract state* is an interpretation that assigns to each variable u_i a value in the domain of u_i. We denote by Σ_A the (finite) set of all abstract states. An *abstraction mapping* is presented by a set of equalities

$$\alpha_{\mathcal{E}} : \quad u_1 = \mathcal{E}_1(V),\ \ldots,\ u_n = \mathcal{E}_n(V),$$

where each $\mathcal{E}_i$ is an expression over V ranging over the domain of u_i. The abstraction $\alpha_{\mathcal{E}}$ induces a semantic mapping $\alpha_{\mathcal{E}} : \Sigma \mapsto \Sigma_A$, from the states of $\mathcal{D}$ to the set of abstract states.

Usually, most of the abstract variables are boolean, and then the corresponding expressions $\mathcal{E}_i$ are predicates over V. This is why this type of abstraction is often referred to as *predicate abstraction* with $\{\mathcal{E}_1, \ldots, \mathcal{E}_n\}$ being the *predicate base*. The abstraction mapping $\alpha_{\mathcal{E}}$ can be expressed succinctly by:

$$V_A = \mathcal{E}(V)$$

When there is no ambiguity, we refer to $\alpha_{\mathcal{E}}$ simply as α. For an assertion $p(V)$, we define its α-abstraction (with some overloading of notation) by:

$$\alpha(p): \quad \exists V \ . \ (V_A = \mathcal{E}(V) \ \wedge \ p(V))$$

The semantics of $\alpha(p)$ is $\|\alpha(p)\| : \{\alpha(s) \mid s \in \|p\|\}$. Note that $\|\alpha(p)\|$ is an *expanding*, or over-approximating, abstraction – an abstract state S is in $\|\alpha(p)\|$ iff *there exists* some concrete p-state that is abstracted into S. A bi-assertion $\beta(V, V')$ is abstracted by:

$$\alpha^2(\beta): \quad \exists V, V' \ . \ (V_A = \mathcal{E}(V) \ \wedge \ V'_A = \mathcal{E}(V') \ \wedge \ \beta(V, V'))$$

In a dual way to the abstraction of a concrete assertion, we can concretize an abstract assertion. Let Φ be an abstract assertion. The *concretization of* Φ, denoted by $\alpha^{-1}(\Phi)$, is defined as

$$\alpha^{-1}(\Phi): \quad \exists V_A \ . \ (V_A = \mathcal{E}(V) \ \wedge \ \Phi(V_A))$$

Similarly, an abstract bi-assertion $\Psi(V_A, V'_A)$ is concretized by

$$\alpha^{-1}(\Psi): \quad \exists V_A, V'_A \ . \ (V_A = \mathcal{E}(V) \ \wedge \ V'_A = \mathcal{E}(V') \ \wedge \ \Psi(V_A, V'_A))$$

For example, consider the abstract assertion $\Phi : \Pi = 3 \ \wedge \ X = 1 \ \wedge \ Y = 0$ over program Abstract-Augmented-Nested-Loops of Fig. 3.2. Its concretization is given by $\alpha^{-1}(\Phi) : \pi = 3 \ \wedge \ x > 0 \ \wedge \ y = 0$.

While the abstraction $\alpha(p)$ has been described as *expanding*, we define the *dual contracting abstraction* $\underline{\alpha}$ by

$$\underline{\alpha}(p) : \quad \alpha(1) \ \wedge \ \neg\alpha(\neg p)$$

The conjunct $\alpha(1)$, taking the α-abstraction of *true* $(= 1)$, restricts the range of $\underline{\alpha}$ to contain only abstract states that have at least some concrete source state mapped by α into S.

The abstraction α is said to be *precise with respect to the assertion* p if $\underline{\alpha}(p) = \alpha(p)$, implying that we cannot have a p-state and a $(\neg p)$-state both being abstracted into the same abstract state.

For a temporal formula ψ in positive normal form (where negation is applied only to state assertions), ψ^{α} is the formula obtained by abstracting every maximal state sub-formula p in ψ into $\underline{\alpha}(p)$.

With no loss of generality we assume that all temporal specifications of properties are given in positive normal form. Thus, when we write the property $p \Longrightarrow \Diamond\, q$, we will treat it as though it is presented in the (logically equivalent) form $\Box(\neg p \ \vee \ \Diamond\, q)$.

The abstraction of $\mathcal{D}$ by α is the system

$$\mathcal{D}^{\alpha} \ = \ \langle V_A, \alpha(\Theta), \alpha^2(\rho), \ \{\alpha(J) \mid J \in \mathcal{J}\}, \ \{\langle \underline{\alpha}(p), \alpha(q)\rangle \mid (p,q) \in \mathcal{C}\}\, \rangle$$

The soundness of abstraction is derived from [KP00]:

Theorem 2.7. *For a system $\mathcal{D}$, abstraction α, and a temporal formula ψ:*

$$\mathcal{D}^\alpha \models \psi^\alpha \qquad \Longrightarrow \qquad \mathcal{D} \models \psi$$

Thus, if an abstract system satisfies an abstract property, then the concrete system satisfies the concrete property.

In general we aim to abstract the data of the system, while preserving its *control structure*. This is formalized by the following definition:

Definition 2.8 (Control-Preserving Abstraction)**.** *Let $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ be an* FDS *with a program counter variable $\pi \in V$. Let $\alpha_\mathcal{E} : V_A = \mathcal{E}(V)$ be an abstraction mapping. $\alpha_\mathcal{E}$ is said to be* control-preserving *if it contains the equality* $(\Pi = \pi)$ *where $\Pi \in V_A$ is a variable ranging over the same finite domain as π.*

Definition 2.8 is easily extended to the case of a concurrent program with k processes: Instead of considering a single concrete program counter, we assume a set of concrete and abstract program counters $\{\pi_1, \ldots, \pi_k\}$ and $\{\Pi_1, \ldots, \Pi_k\}$, respectively. Then for each $i \in [1..k]$, a control-preserving abstraction should contain the equality $\Pi_i = \pi_i$.

In the following chapters, we assume that all abstractions are control-preserving and omit the explicit mapping of concrete and abstract program counters.

We close this section by demonstrating the process of predicate abstraction on a simple infinite-state program.

Example 2.1 (Predicate Abstraction)

```
                    x, y  :  natural   init x = y = 0
                                        ┌ m_0 :  while y = 0 do       ┐
┌ ℓ_0 :  while x = 0 do      ┐          │          m_1 :  skip        │
│         ℓ_1 :  y := y + 1  │    ||    │ m_2 :  x := y               │
└ ℓ_2 :                      ┘          └ m_3 :                       ┘
```

Figure 2.2: Program Any-X

```
                       X_p, Y_p  :  bool   init X_p = Y_p = False
                       X_0, Y_0  :  bool   init X_0 = Y_0 = True
                                                      ┌ m_0 :  while Y_0 do                ┐
┌ ℓ_0 :  while X_0 do                        ┐        │          m_1 :  skip               │
│         ℓ_1 :  (Y_0, Y_p) := (False, True) │   ||   │ m_2 :  (X_0, X_p) := (Y_0, Y_p)    │
└ ℓ_2 :                                      ┘        └ m_3 :                              ┘
```

Figure 2.3: Abstraction of Program Any-X

Consider program Any-X given in Fig. 2.2, which consists of two asynchronously composed processes that communicate using shared variables. Consider the specification given by the safety property

$$at\_\ell_2 \implies at\_m_3$$

To verify that Any-X satisfies its specification, we apply the predicate abstraction induced by the following mapping:

$$\mathcal{P} : \left\{ \begin{array}{ccc} X_p & \leftrightarrow & x > 0, \\ X_0 & \leftrightarrow & x = 0, \\ Y_p & \leftrightarrow & y > 0, \\ Y_0 & \leftrightarrow & y = 0 \end{array} \right\}$$

This results in the abstract program given in Fig. 2.3. We now model check that the abstract program satisfies the abstract property, which is given by

$$\Pi_1 = \ell_2 \implies \Pi_2 = m_3$$

┛

Chapter 3

Ranking Abstraction

This chapter deals with abstraction-aided verification of temporal specifications. State abstraction often does not suffice to verify progress properties ([PR05]). We consider *ranking abstraction*, a method of augmenting the concrete program by a non-constraining progress monitor, which measures the progress of program execution, relative to a given ranking function. Once a program is augmented, a conventional state abstraction can be used to preserve the ability to monitor progress in the abstract system. This method was introduced in [KP00] and further clarified and elaborated in [KPV01].

We demonstrate the use of ranking refinement for proving termination of a program with nested loops and unbounded random assignments, as well as a bubble sort algorithm on unbounded linked lists. Both examples entail the use of additional heuristics in order to synthesize core ranking functions.

3.1 Modular Ranking Augmentation

Ranking abstraction allows us to get away with finding a set of possible ingredients for ranking functions, without having to design a comprehensive single ranking function, which is usually required in deductive verification of termination (of the style advocated in [MP91a]). This is accomplished by means of augmenting the system with several non-constraining monitors, and predicate-abstracting the resulting system.

Fix some system $\mathcal{D}\colon \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ and some well-founded domain $(\mathcal{W}, \succ)$, and let δ be some ranking function over the domain. Let *dec* be a fresh variable (not in V). The *augmentation of* $\mathcal{D}$ *by* δ, denoted by $\mathcal{D}{+}\delta$, is the system

$$\mathcal{D}{+}\delta\colon \langle V \cup \{dec\}, \Theta, \rho \ \wedge \ \rho_\delta, \mathcal{J}, \mathcal{C} \cup \{(dec > 0, dec < 0)\}\rangle$$

where the conjunct ρ_δ is defined by:

$$\rho_\delta\colon dec' = \begin{cases} 1, & \text{If } \delta \succ \delta' \\ 0, & \text{If } \delta = \delta' \\ -1, & \text{otherwise} \end{cases}$$

Thus, $\mathcal{D}{+}\delta$ behaves exactly like $\mathcal{D}$ and, in addition, keeps track of whether δ decreases, remains the same, or otherwise. The new compassion requirement captures the restriction that δ cannot decrease infinitely often without increasing infinitely often, which follows immediately from the well foundedness of $\mathcal{W}$. For the pervasive case that δ ranges over the naturals, we can express ρ_δ as $dec' = sign(\delta - \delta')$.

Since augmentation does not constrain the behavior of $\mathcal{D}$, any property over V is valid over $\mathcal{D}$ iff it is valid over $\mathcal{D}+\delta$. In order to verify a liveness property of $\mathcal{D}$, the augmentation $\mathcal{D}+\delta$ can be predicate abstracted and checked for satisfiability of the abstracted property. In that abstraction, it is not necessary to abstract the variable dec since it ranges over the finite domain $\{-1, 0, 1\}$. Therefore, the abstraction preserves the compassion requirement $(dec > 0, dec < 0)$. Note that we do not require that δ decreases on every step. As demonstrated below, it suffices to have δ capture some of the behavioral aspects of a "comprehensive" ranking function.

Combining such augmentation with subsequent predicate abstraction translates the effect of a well-founded ranking function in the concrete system into a compassion requirement in the abstract system. Both have the effect that they disallow a cycle in the execution of the program. A concrete cycle in which the ranking decreases at least once and never increases cannot exist within the set of pending states. Similarly, in the abstracted version of the same program we cannot have a cycle in which $dec = 1$ at least once and $dec \geq 0$ at all states.

Example 3.1 (Nested Loops) Consider program Nested-Loops in Fig. 3.1(a). In this program, the statements "$x :=\ ?$" and "$y :=\ ?$," in lines 0 and 1 respectively, denote random assignments of arbitrary positive integers to variables x and y. For this program, we are interested in proving that it always terminates, which can be specified by the response property $at\_0 \Longrightarrow \Diamond\, at\_6$, where the assertion at_k stands for $\pi = k$.

An initial attempt to prove termination of this program is to define the ranking function $\delta_y = y$. The augmentation $\mathcal{D}{+}\delta_y$ is shown in Fig. 3.1(b). Note that statements that in the original program assign values to y, are now replaced with a simultaneous assignment to *both* y and the augmentation variable dec_y. In the case of control statements such as **while**, the augmentation is not displayed explicitly. However, it is implicitly assumed that the assignment $dec_y := 0$ is executed in parallel with any of these statements. Note that the assignments to dec_y have been optimized in some of the statements, replacing the expression $sign(y - y')$ by its values that are known to be 1 and 0 at the execution of statements 4 and 5, respectively.

While this augmentation is not sufficient to prove termination of the entire program, it can be used to prove termination of the inner loop (lines 3, 4).

Consider the abstraction:

$$\alpha : \quad \Pi = \pi,\ X = (x > 0),\ Y = (y > 0),\ Dec_y = dec_y$$

where Π is the abstract program counter. The resulting abstract program is presented in Fig. 3.2. Note that α introduces nondeterministic assignments to both X and Y (lines 2, 4 and 5). It is now possible to verify, e.g. by model checking, the termination of the inner loop.

Deductive verification of termination (*à* la [MP91a]) of the inner loop consisting of statements 3 and 4, requires the use of the ranking function

```
        $x, y$ : **natural init** $x = 0, y = 0$
    [ 0 :  $x :=\ ?$
      1 :  **while** $x > 0$ **do**
             [ 2 :  $y :=\ ?$
                    3 : **while** $y > 0$ **do**
                        [ 4 :  $y := y - 1$ ]
               5 :  $x := x - 1$ ]
      6 : ]
```

(a) Program NESTED-LOOPS

```
        $x, y$ : **natural init** $x = 0, y = 0$
        $dec_y$ : $\{-1, 0, 1\}$
        **compassion** $(dec_y > 0, dec_y < 0)$
    [ 0 :  $(x, dec_y) :=\ (?, 0)$
      1 :  **while** $x > 0$ **do**
             [ 2 :  $(y, dec_y) :=\ (?, sign(y - y'))$
                    3 : **while** $y > 0$ **do**
                        [ 4 :  $(y, dec_y) := (y - 1, 1)$ ]
               5 :  $(x, dec_y) := (x - 1, 0)$ ]
      6 : ]
```

(b) Program AUGMENTED-NESTED-LOOPS

Figure 3.1: Program NESTED-LOOPS and Its Augmented Version

$2y + (\pi = 3)$ (where the boolean expression $(\pi = 3)$ evaluates to 1 on states in which π equals 3) or the function $\langle y, \pi = 3\rangle$ ranging over lexicographic pairs. However, supplying the model checker with the "ingredient rank" y suffices for the application of the ranking abstraction method. Obviously, to obtain the termination of the complete program, one need also consider the variable x. ⌟

As shown in Example 3.1, it is sometimes necessary to include several δ's in order to obtain a termination proof, by considering simultaneous augmentations by a set of ranking functions. A *ranking core* is a set of ranking functions. Let $\mathcal{R}$ be the ranking core $\{\delta_1, \ldots, \delta_k\}$. The *ranking augmentation*

```
        $X, Y$ : $\{0,1\}$ **init** $Y = 0, X = 0$
        $Dec_y$ : $\{-1,0,1\}$
        **compassion** $(Dec_y > 0, Dec_y < 0)$
0 : $(X, Dec_y) :=$ $(1, 0)$
1 : **while** $X$ **do**
        2 : $(Y, Dec_y) :=$ $(\{0,1\}, \{-1,0,1\})$
            3 : **while** $Y$ **do**
                4 : $(Y, Dec_y) := (\{0,1\}, 1)$
        5 : $(X, Dec_y) := (\{0,1\}, 0)$
6 :
```

Figure 3.2: Program Abstract-Augmented-Nested-Loops

$\mathcal{D}{+}\mathcal{R}$ is the system

$$\mathcal{D}{+}\mathcal{R}: \quad (\cdots((\mathcal{D}{+}\delta_1){+}\delta_2){+}\cdots){+}\delta_k$$

Just like the case of predicate abstraction, we lose nothing (except efficiency) by adding potentially redundant rankings. The main advantage here over direct use of ranking functions within deductive verification is that one may contribute as many elementary ranking functions as one wishes. It is then left to a model checker to sort out their interaction and relevance. To illustrate this, consider a full deductive proof of termination of program Nested-Loops. Due to the unbounded nondeterminism of the random assignments, a deductive termination proof is necessarily based on a ranking function over lexicographic tuples, an example of which is the following:

$$\langle(\pi = 0), 4x + 3(\pi = 1) + 2(\pi = 2) + (\pi \in \{3,4\}), 2y + (\pi = 3)\rangle$$

With multi-component ranking abstraction, however, one need only provide the well-founded ranking core $\mathcal{R} = \{x, y\}$. This also improves on the augmentation-based approaches presented in [KP00] and [KPV01] that, intent on exploring the theory of this method rather than its practical applications, imply the use of a single ranking function.

To abbreviate the notation, we will write $\mathcal{D}^{\mathcal{R},\alpha}$ as shorthand for $(\mathcal{D}+\mathcal{R})^{\alpha}$. Note that when we perform ranking abstraction w.r.t a core $\mathcal{R} : \delta_1, \ldots, \delta_k$, we use an abstraction mapping that extends α by the additional definitions:

$$Dec_1 = dec_1,\ \ldots,\ Dec_k = dec_k$$

Since augmentation induced by the ranking core $\mathcal{R}$ does not constrain the behavior of the original FDS $\mathcal{D}$, it follows that every $\sigma : s_0, s_1, \ldots$, a computation of $\mathcal{D}$, gives rise to $\widetilde{\sigma} : \widetilde{s}_0, \widetilde{s}_1, \ldots$, a computation of $\mathcal{D}+\mathcal{R}$ agreeing with σ on all variable except for the *dec* variables associated with $\mathcal{R}$. The computation $\widetilde{\sigma}$ can be abstracted into $\sigma^{\alpha} : S_0, S_1, \ldots$, a computation of $\mathcal{D}^{\mathcal{R},\alpha}$, such that $S_i = \alpha(\widetilde{s}_i)$, for all $i \geq 0$. Thus, the set of computations of $\mathcal{D}$ is, modulo augmentation and abstraction, a subset of the computations of $\mathcal{D}^{\mathcal{R},\alpha}$.

3.2 Soundness and Completeness of the Method

In order to establish soundness and completeness of the ranking abstraction method we have to consider a more general augmentation than just pure ranking abstraction. Such an augmentation may introduce additional auxiliary variables and their transitions, provided these additions do not constrain the behavior of the system $\mathcal{D}$.

Let $\mathcal{D}_1 : \langle V_1, \Theta_1, \rho_1, \mathcal{J}_1, \mathcal{C}_1 \rangle$ and $\mathcal{D}_2 : \langle V_2, \Theta_2, \rho_2, \mathcal{J}_2, \mathcal{C}_2 \rangle$ be two systems. We define the *synchronous parallel composition* of $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 \,|||\, \mathcal{D}_2$, to be the system $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where,

$$\begin{array}{rclcrclcrcl}
V & = & V_1 \cup V_2 & & \Theta & = & \Theta_1 \wedge \Theta_2 & & \rho & = & \rho_1 \;\wedge\; \rho_2 \\
\mathcal{J} & = & \mathcal{J}_1 \cup \mathcal{J}_2 & & \mathcal{C} & = & \mathcal{C}_1 \cup \mathcal{C}_2 & & & &
\end{array}$$

As implied by the definition, each of the basic actions of system $\mathcal{D}$ consists of the joint execution of an action of $\mathcal{D}_1$ and an action of $\mathcal{D}_2$. Thus, we can view the execution of $\mathcal{D}$ as the *joint execution* of $\mathcal{D}_1$ and $\mathcal{D}_2$.

We are interested in the synchronous parallel composition $\mathcal{A} : \; \mathcal{D} \,|||\, M$, where $\mathcal{D}$ is the system to be verified, while FDS M serves as a *monitor* that observes the behavior of system $\mathcal{D}$. Let $\sigma : s_0, s_1, \ldots$ be a computation of the parallel composition $\mathcal{A}$. We denote by $\sigma \Downarrow_{\mathcal{D}}$ the sequence of states obtained by projecting each state s_i on the variables of $\mathcal{D}$. Let $\mathcal{C}omp(\mathcal{A}) \Downarrow_{\mathcal{D}}$ denote the set of all computations of the composition $\mathcal{A}$ when projected on the variables of $\mathcal{D}$. In general, we have the following relation between the computations of $\mathcal{A}$ and the computations of $\mathcal{D}$:

$$\mathcal{C}omp(\mathcal{D} \,|||\, M) \Downarrow_{\mathcal{D}} \quad \subseteq \quad \mathcal{C}omp(\mathcal{D})$$

That is, any $\mathcal{D}$-projection of a computation of $\mathcal{A}$ is a computation of $\mathcal{D}$. However, there may be computations of $\mathcal{D}$ that are blocked due to the interaction with the monitor M. An augmentation $\mathcal{D} \;|||\; M$ is defined to be *non-constraining* if the set of $\mathcal{D}$-projections of the computation of $\mathcal{A}$ equals the set of $\mathcal{D}$-computations, i.e.

$$\mathcal{C}omp(\mathcal{D} \;|||\; M)\Downarrow_{\mathcal{D}} \quad = \quad \mathcal{C}omp(\mathcal{D})$$

That is, any computation of $\mathcal{D}$ can be extended by an appropriate assignment of values to the variables in $V_M - V_{\mathcal{D}}$ to a computation of $\mathcal{D} \;|||\; M$. The notion of a non-constraining monitor is referred to in [KP00] as an *accommodating monitor*.

It is not difficult to see that the ranking augmentation corresponding to the ranking function δ is equivalent to augmentation by the following non-constraining monitor

$$M_\delta = \left(\begin{array}{llll} V: & V_\delta \cup \{dec : \{-1, 0, 1\}\}, & \mathcal{J}: & \emptyset, \\ \Theta: & \textsc{True}, & \mathcal{C}: & \{(dec > 0, dec < 0)\} \\ \rho: & dec' = sign(\delta - \delta') & & \end{array}\right)$$

where V_δ is the set of variables occurring within δ.

The fact that M_δ is non-constraining follows from the transition relation, which can always assign an appropriate value to the fresh variable *dec*. The compassion requirement is non-constraining because it is a direct consequence of the fact that the ranking function δ ranges over a well-founded domain and, therefore, cannot decrease infinitely many times without also increasing infinitely many times.

Temporal Testers

One of the most useful non-constraining monitors that can be augmented to a system is a *temporal tester* for an LTL formula ψ. As shown in [KPR98], [KP00] and [KP05], it is possible to construct a *temporal tester* $T[\psi]$ for every LTL formula ψ. The tester $T[\psi]$ is an FDS with a distinguished (output) variable x such that, in any computation $\sigma : s_0, s_1, \ldots$ of $T[\psi]$, and at any position $j \geq 0$, $s_j[x] = 1$ iff $(\sigma, j) \models \psi$. Furthermore, $T[\psi]$ is non-constraining, which implies that any state sequence σ can be extended to a computation of $T[\psi]$ by assigning appropriate values to the output variable x. The main application of temporal testers is for model checking of LTL properties. It is based on the observation that

> For a system $\mathcal{D}$ and LTL formula ψ, $\mathcal{D} \models \psi$ iff the synchronous parallel composition $\mathcal{D} \mathbin{|||} T[\psi] \mathbin{|||} [\Theta : x = 0]$ has no computations.

Here $[\Theta : x = 0]$ is a trivial FDS that imposes the initial condition that, at position 0, the output variable x assumes the value 0. Thus, a computation of $\mathcal{D} \mathbin{|||} T[\psi] \mathbin{|||} [\Theta : x = 0]$ corresponds to a computation of $\mathcal{D}$ that satisfies $\neg\psi$. Claiming that this composition has no computations is equivalent to the claim that all computations of $\mathcal{D}$ satisfy the formula ψ.

In this paper we use temporal testers in order to enrich the abstraction by additional information that traces the satisfaction of an arbitrary LTL formula during execution of a system.

We are now ready to state the theorems of soundness and completeness for the ranking abstraction method in a more general context, in which the original system is augmented by an arbitrary non-constraining progress monitor M. The case that the progress monitor consists purely of a ranking augmentation according to a ranking core $\mathcal{R}$ is an important special case. To denote the dependence of the monitor M on the ranking core $\mathcal{R}$, we often write it as $M_{\mathcal{R}}$. The following theorem is taken from [KP00].

Theorem 3.1 (Soundness)**.**
For a system $\mathcal{D}$, a progress monitor $M_{\mathcal{R}}$ that is non-constraining w.r.t $\mathcal{D}$, abstraction α, and a temporal formula ψ:

$$(\mathcal{D} \parallel\!| M_{\mathcal{R}})^{\alpha} \models \psi^{\alpha} \qquad \Longrightarrow \qquad \mathcal{D} \models \psi$$

Thus, if an augmented and abstracted system satisfies an abstract property, then the concrete system satisfies the concrete property.
The notion that (augmented) ranking abstraction is more powerful than predicate abstraction for the verification of temporal properties is formalized by the following claim of completeness ([KP00]):

Theorem 3.2 (Completeness)**.**
The method of ranking abstraction is complete. Namely, for every system $\mathcal{D}$

and temporal formula ψ, such that $\mathcal{D} \models \psi$, there exist a progress monitor $M_{\mathcal{R}}$ and an abstraction α such that $(\mathcal{D} \,|||\, M_{\mathcal{R}})^{\alpha} \models \psi^{\alpha}$.

The theorem shows that every LTL property, and, in particular, all progress properties, can be verified using the ranking abstraction method. In comparison, state-based predicate abstraction can only verify safety properties.

A close study of the completeness proof, as presented in [KP00] yields the following observations:

- In many cases, the progress monitor is just the ranking augmentation given by $\mathcal{D}$+$\mathcal{R}$. This is typically the case when the temporal property ψ is not too complex. For example, for verifying response properties, there is no need for an augmentation beyond the ranking augmentation induced by $\mathcal{R}$.

- In all other cases, it is sufficient to augment the system by the temporal tester $T[\psi]$ and then apply the ranking abstraction induced by $\mathcal{R}$.

Chapter 4

Abstraction and Ranking Refinement

In this section we will show that, similarly to predicate abstraction, ranking abstraction also possesses a counterexample guided refinement process. Assume that, wishing to check that $\mathcal{D} \models \psi$, we have model checked $\mathcal{D}^{\mathcal{R},\alpha} \models \psi^{\alpha}$ and have obtained an abstract counterexample σ^{α}. There are two possibilities. Either there exists a concrete computation σ, such that σ^{α} is the abstraction of σ, or σ^{α} cannot be concretized. In the first case, σ is a true counterexample, implying that ψ is not valid over $\mathcal{D}$. In the second case, this means that our abstraction is too coarse and needs to be refined.

The process of counterexample guided refinement has to distinguish between these two cases, and in the case of a spurious counterexample, to utilize the failure to concretize in order to refine the two abstraction components: α

and $\mathcal{R}$. Note that the situation here is more complex than simple predicate abstraction, because the refinement may call for a refinement of α or of $\mathcal{R}$, or of both. The realization that, in addition to predicate refinement, there is also a need to refine the termination components has already been made in [CPR06]. However, they perform their version of refinement within the framework of transition invariants.

4.1 Abstract Runs and Their Concretizations

Let $\mathcal{D}\colon \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ be a system, $\mathcal{P}$ be a predicate base, α be the abstraction mapping (lifted to map assertions) $\alpha(p) : \exists V. V_A = \mathcal{P}(V) \wedge p(V)$, and α^{-1} be α's inverse, i.e., the mapping $\alpha^{-1}(\Phi) : \exists V_A. V_A = \mathcal{P}(V) \wedge \Phi(V_A)$, which we fix for the duration of this section. We refer to runs of $\mathcal{D}$ and $\mathcal{D}^{\mathcal{R},\alpha}$ as *concrete* and *abstract* runs, respectively. In this section, we assume that, unless explicitly stated otherwise, all runs are finite. For a run $\xi : s_0, \ldots, s_m$, we denote by $|\xi| = m$ the *length* of ξ.

Consider an abstract run $\Xi\colon S_0, \ldots, S_m$. A concrete run $\xi : s_0, \ldots, s_m$ is called a *concretization of* Ξ if $\alpha(s_i) = S_i$, for all $i \in [0..m]$. Rather then considering a single state concretization of the abstract run Ξ, we may wish to derive a characterization of all possible concretizations of Ξ. This may be captured by a sequence $\varphi_0, \ldots, \varphi_m$ of assertions over V. We thus define the *symbolic concretization of* Ξ (*with respect to* $\mathcal{D}$) to be the sequence

$\gamma(\Xi)\colon \varphi_0, \ldots, \varphi_m$ of concrete assertions inductively as follows:

$$\varphi_i\colon \begin{cases} \Theta \ \wedge \ \alpha^{-1}(S_0) & i = 0 \\ (\varphi_{i-1} \diamond \rho) \ \wedge \ \alpha^{-1}(S_i) & i \in [1..m] \end{cases}$$

where $\varphi \diamond \rho$ is the assertion characterizing the states that are ρ-successors of a φ-state. We sometimes refer to φ_i as $\gamma(\Xi)[i]$, or simply $\gamma[i]$ if Ξ is understood from the context.

Example 4.1 Recall program NESTED-LOOPS of Fig. 3.1(a), and consider the abstraction and ranking core of Example 3.1, that is:

$$\begin{aligned} &\alpha : X = (x > 0),\ Y = (y > 0),\ Dec_y = dec_y \\ &\mathcal{R} : \{\delta_1 = y\} \end{aligned}$$

The abstract system is shown in Fig. 3.2. Consider an abstract run $\Xi\colon S_0, \ldots, S_6$ of the system, where

$$\begin{array}{lclclcl} S_0 & : & \langle \Pi{:}0,\ X{:}0,\ Y{:}0,\ Dec_y{:}0 \rangle & & S_1 & : & \langle \Pi{:}1,\ X{:}1,\ Y{:}0,\ Dec_y{:}0 \rangle \\ S_2 & : & \langle \Pi{:}2,\ X{:}1,\ Y{:}0,\ Dec_y{:}0 \rangle & & S_3 & : & \langle \Pi{:}3,\ X{:}1,\ Y{:}1,\ Dec_y{:}-1 \rangle \\ S_4 & : & \langle \Pi{:}4,\ X{:}1,\ Y{:}1,\ Dec_y{:}0 \rangle & & S_5 & : & \langle \Pi{:}3,\ X{:}1,\ Y{:}0,\ Dec_y{:}1 \rangle \\ S_6 & : & \langle \Pi{:}5,\ X{:}1,\ Y{:}0,\ Dec_y{:}0 \rangle & & & & \end{array}$$

The symbolic concretization of Ξ is $\varphi_0, \ldots \varphi_6$ where:

$$\begin{array}{ll}
\varphi_0\colon & \pi = 0 \ \wedge \ x = 0 \ \wedge \ y = 0 \ \wedge \ dec_y = 0 \\
\varphi_1\colon & \pi = 1 \ \wedge \ x > 0 \ \wedge \ y = 0 \ \wedge \ dec_y = 0 \\
\varphi_2\colon & \pi = 2 \ \wedge \ x > 0 \ \wedge \ y = 0 \ \wedge \ dec_y = 0 \\
\varphi_3\colon & \pi = 3 \ \wedge \ x > 0 \ \wedge \ y > 0 \ \wedge \ dec_y = -1 \\
\varphi_4\colon & \pi = 4 \ \wedge \ x > 0 \ \wedge \ y > 0 \ \wedge \ dec_y = 0 \\
\varphi_5\colon & \pi = 3 \ \wedge \ x > 0 \ \wedge \ y = 0 \ \wedge \ dec_y = 1 \\
\varphi_6\colon & \pi = 5 \ \wedge \ x > 0 \ \wedge \ y = 0 \ \wedge \ dec_y = 0
\end{array}$$

┛

The following claim establishes the relation between symbolic concretizations of abstract and concrete runs:

Claim 4.1 (Feasibility). *For every abstract run Ξ, $k \in [0..|\Xi|]$, and concrete state s, s satisfies $\gamma[k]$ iff there exists a concrete run $s_0, \ldots, s_k = s$ that is a concretization of $\Xi[0..k]$.*

Proof. Assume that $s \models \gamma[k]$. Proceeding from k down to 0, we will construct a sequence of states $s = s_k, s_{k-1}, \ldots, s_0$, such that, for each $i = 0, \ldots, k$, $\Xi[i] = \alpha(s_i)$ and $s_i \models \gamma[i]$, s_0 is initial, and, for each $i \in [0..k-1]$, s_{i+1} is a ρ-successor of s_i.

For every $i \in [1..k]$ assume that we already constructed s_i, such that $s_i \models \gamma[i]$. The fact that s_i satisfies $\gamma[i] = (\gamma[i-1] \diamond \rho) \ \wedge \ \alpha^{-1}(\Xi[i])$ implies that $\Xi[i] = \alpha(s_i)$ and that there exists a state s_{i-1} that is a ρ-predecessor of s_i and satisfies $\gamma[i-1]$.

For $i = 0$, the fact that $s_0 \models \gamma[0] = \alpha^{-1}(S_0) \ \wedge \ \Theta$ implies that s_0 is an initial state such that $S_0 = \alpha(s_0)$.

Thus, the state sequence $\xi : s_0, \ldots, s_k = s$ is a concretization of $\Xi[0..k]$.

In the other direction the claim is straightforward. □

A corollary of Claim 4.1 is that an abstract run $\Xi[1..m]$ can be concretized iff $\gamma(\Xi)[m]$ is satisfiable.

The assertion $\gamma(\Xi)[i]$ characterizes all the states that can appear at position i of a concretization of the abstract run Ξ. We will generalize this notion by defining a bi-assertion $\beta_{i,j}(\Xi)$, for $0 \leq i \leq j \leq |\Xi|$, such that $\langle s_a, s_b \rangle \models \beta_{i,j}$ iff there exists $\xi : s_0, \ldots, s_{|\Xi|}$, such that $s_i = s_a$ and $s_j = s_b$. This bi-assertion will be used when attempting to concretize "abstract cycles" that are obtained in counterexamples. The generic presentation of the bi-assertion is $\beta_{i,j}(V_0, V)$ (rather than $\beta_{i,j}(V, V')$), where V_0 is a fresh copy of the system variables, and records the values of variables at state s_i.

Let $\Xi = S_0, \ldots, S_m$ be an abstract run. The bi-assertion $\beta_{i,j}(\Xi)$ is defined inductively, for all j, $i \leq j \leq m$ by:

$$\beta_{i,j} = \begin{cases} V = V_0 \ \wedge \ \alpha^{-1}(S_i) & j = i \\ (\beta_{i,j-1} \diamond \rho) \ \wedge \ \alpha^{-1}(S_j) & j > i \end{cases}$$

In this definition, $V = V_0$ is an abbreviation for $\bigwedge_{x \in V}(x = x_0)$, which states equality between all V-variables and their corresponding V_0-counterparts. The expression $\beta_{i,j-1} \diamond \rho$ stands for

$$\exists \widetilde{V} : (\beta_{i,j-1}(V_0, \widetilde{V}) \wedge \rho(\widetilde{V}, V))$$

Note in particular that this expression preserves the values of the V_0-variables from $\beta_{i,j-1}$ to $\beta_{i,j}$.

Example 4.2 Continuing Example 4.1, we compute $\beta_{1,1}, \ldots, \beta_{1,6}$ as follows:

$$\begin{array}{lcl}
\beta_{1,1} & : & \mathit{init} \ \wedge \ \pi = \pi_0 \ \wedge \ x = x_0 \ \wedge \ y = y_0 \ \wedge \ \mathit{dec}_y = \mathit{dec}_y^0 \\
\beta_{1,2} & : & \mathit{init} \ \wedge \ \pi = 2 \ \wedge \ x = x_0 \ \wedge \ y = y_0 \ \wedge \ \mathit{dec}_y = 0 \\
\beta_{1,3} & : & \mathit{init} \ \wedge \ \pi = 3 \ \wedge \ x = x_0 \ \wedge \ y > 0 \ \wedge \ \mathit{dec}_y = -1 \\
\beta_{1,4} & : & \mathit{init} \ \wedge \ \pi = 4 \ \wedge \ x = x_0 \ \wedge \ y > 0 \ \wedge \ \mathit{dec}_y = 0 \\
\beta_{1,5} & : & \mathit{init} \ \wedge \ \pi = 3 \ \wedge \ x = x_0 \ \wedge \ y = 0 \ \wedge \ \mathit{dec}_y = 1 \\
\beta_{1,6} & : & \mathit{init} \ \wedge \ \pi = 5 \ \wedge \ x = x_0 \ \wedge \ y = 0 \ \wedge \ \mathit{dec}_y = 0
\end{array}$$

where $\mathit{init} : \pi_0 = 1 \ \wedge \ x_0 > 0 \ \wedge \ y_0 = 0 \ \wedge \ \mathit{dec}_y^0 = 0$. ┛

4.2 A "Fluctuation-Proof" Ranking Augmentation

We now consider a somewhat more complex ranking augmentation that is designed to simplify the ranking components required by the counterexample-guided refinement described in the next section.

For an FDS $\mathcal{D} : \langle V_\mathcal{D}, \Theta_\mathcal{D}, \rho_\mathcal{D}, \mathcal{J}_\mathcal{D}, \mathcal{C}_\mathcal{D} \rangle$, a set of variables $V_\delta \subseteq V_\mathcal{D}$, a ranking function $\delta(V_\delta)$, and an assertion φ over $V_\mathcal{D}$, we define the monitor $M_{\delta,\varphi}$ as follows:

$$M_{\delta,\varphi} = \left[\begin{array}{ll} V: & V_{\mathcal{D}} \cup V_0 \cup \{dec : \{-1,0,1\}\}, \\ \Theta: & V_0 = V_\delta, \\ & \\ \rho: & \left(\begin{array}{c} \varphi' \ \wedge \ V_0' = V_\delta' \ \wedge \ dec' = sign(\delta(V_0) - \delta(V_\delta')) \\ \vee \\ \neg\varphi' \ \wedge \ V_0' = V_0 \ \wedge \ dec' = 0 \end{array}\right) \\ & \\ \mathcal{J}: & \emptyset, \\ \mathcal{C}: & \{(dec > 0, dec < 0)\} \end{array}\right]$$

where V_0 is a fresh copy of the variable set V_δ. In essence, the monitor $M_{\delta,\varphi}$ records changes to the ranking only upon visiting φ-states, at which time the variables of V_0 are updated. As long as φ does not hold, the value of V_0 remains unchanged. In the next section, this will be used to capture the effect of a control-flow loop iteration on the variables of $\mathcal{D}$.

Example 4.3

```
     x : natural
1 :  while x > 0 do
       [2 :  x := x + 1;]
       [3 :  x := x − 2; ]
4 :
```

(a) A Simple Loop with Fluctuating Ranking

```
x : natural,          dec : {−1, 0, 1}
1 :  while x > 0 do
       [2 :  (x, dec) := (x + 1, −1);]
       [3 :  (x, dec) := (x − 2, 1);  ]
4 :
  compassion(dec > 0, dec < 0)
```

(b) Augmentation with Ranking Core $\{x\}$

Figure 4.1: Motivating Example for Fluctuation-Proof Monitors

Consider the program of Fig. 4.1(a) and its augmentation, as defined in Section 3.1, with the ranking core $\{x\}$, which yields the program in Fig. 4.1(b). It is clear that x would serve as the main component in an

```
$x, x_0$ : natural,                    $dec : \{-1, 0, 1\}$
init $x = x_0$
1 :  while $x > 0$ do
     [2 :  $(x, x_0, dec) := (x + 1, x_0, 0)$;                  ]
     [3 :  $(x, x_0, dec) := (x - 2, x', sign(x_0 - x'))$;]
4 :
compassion$(dec > 0, dec < 0)$
```

Figure 4.2: Augmentation with Fluctuation-Proof Monitor

adequate ranking function for proving termination of the program. However, the ranking augmentation, as constructed in Section 3.1, fails in this case due to the fluctuating value of x, which causes *dec* to assume both a positive and a negative value during each loop iteration, which in turn causes the compassion requirement to be satisfied at each iteration. This can be remedied by employing the ranking core $\{5x(\pi = 1) + 4x(\pi = 2) + 3x\}$, which "distributes" the ranking function x over the loop body. Since this distribution of a ranking function is hard automate, we instead rely on the following observation: As long as the overall effect of a loop iteration is to decrease x, we may safely ignore occasional increases while inside the loop. Thus we augment the program of Fig. 4.1(a) with the progress monitor $M_{x,(\pi=1)}$, yielding the program in Fig. 4.2. An equivalent formulation of the same construct can be obtained by augmenting the program of Fig. 4.1(a) with the assignment $x_0 := x$ at location 1, and then applying the simpler ranking augmentation with the ranking core $\{x_0\}$. Note that if we proceed to predicate-abstract this program, the relation between x and its copy x_0 needs to be taken into account. For this purpose, the predicate base $\{x = x_0, x = x_0+1, x = x_0-1\}$

can be used. ┛

As in the construction of Section 3.1, the monitor $M_{\delta,\varphi}$ is non-constraining. Thus the soundness result of the previous section continues to apply. Since the present construction subsumes that of Section 3.1 (the constructions are equivalent when $\varphi = \text{TRUE}$), the completeness result applies as well.

Throughout the rest of this chapter we use a definition of ranking augmentation that is based on the new monitor construction. Namely, the augmentation of an FDS $\mathcal{D}$ by δ with respect to assertion φ, denoted by $\mathcal{D}{+}\langle\delta,\varphi\rangle$, is the synchronous parallel composition $\mathcal{D}|||M_{\delta,\varphi}$. Accordingly, as a ranking core we now use a set $\mathcal{R} : \{\langle\delta_1,\varphi_1\rangle,\ldots,\langle\delta_n,\varphi_n\rangle\}$, and define augmentation of $\mathcal{D}$ by $\mathcal{R}$ to be

$$\mathcal{D}{+}\mathcal{R} : \quad (\cdots((\mathcal{D}{+}\langle\delta_1,\varphi_1\rangle){+}\langle\delta_2,\varphi_2\rangle){+}\cdots){+}\langle\delta_k,\varphi_k\rangle$$

As before we use the notation $\mathcal{D}^{\mathcal{R},\alpha}$ as a shorthand for $(\mathcal{D}{+}\mathcal{R})^{\alpha}$.

4.3 Counterexample Guided Abstraction Refinement

The verification (or refutation) of a progress property ψ over an FDS begins with a (possibly empty) user-provided initial ranking $\mathcal{R}$ and a predicate abstraction $\mathcal{P}$. Following [GS97], initially $\mathcal{P}$ is chosen to be the set of atomic state formulas occurring in ρ, Θ, $\mathcal{J}$, $\mathcal{C}$ and the concrete formula ψ, excluding formulas that refer to primed variables.

Let ψ^α be the formula $\underline{\alpha}(\psi)$. We start by model checking the validity of ψ^α over $\mathcal{D}^{\mathcal{R},\alpha}$. If valid, we can safely conclude that $S \models \psi$. Otherwise, a counterexample is found in the form of a computation of $\mathcal{D}^{\mathcal{R},\alpha}$ that does not satisfy ψ^α. If such a computation exists then a standard model checker will return a counterexample that is finitely represented as a "lasso" – an abstract computation of the form $\Xi_1; \Xi_2^\omega$ where $\Xi_1 : S_0, \ldots, S_{k-1}$ is a finite abstract run, and $\Xi_2 : S_k, \ldots, S_{m-1}$ is a finite sequence of consecutive abstract states. As in the case of predicate abstraction refinement, we first attempt to concretize the counterexample $\Xi : S_0, \ldots, S_k, \ldots, S_{m-1}, S_m = S_k$. Namely, we compute $\gamma(\Xi) : \varphi_0, \ldots, \varphi_m$ and $\beta_{k,m}(\Xi)$. The following may occur:

Case 1. *The counterexample Ξ cannot be concretized.*

This case is identified by observing that $\varphi_m = \gamma[m]$ is unsatisfiable. This is a typical scenario in predicate abstraction refinement – the abstraction is too coarse, and should be refined so as to eliminate the spurious counterexample. One can apply any of the known predicate refinement techniques, e.g., [CGJ$^+$00, BPR02, BR01]. For all following cases, we may assume that φ_m is satisfiable.

Case 2. *The concretization of the counterexample contains a cycle compatible with Ξ_2 — the property is not valid.*

This case is identified by observing that $\varphi_k(V) \wedge \beta_{k,m}(V, V)$ is satisfiable. This implies that there exists a state s such that $s \models \varphi_k$ and $\langle s, s\rangle \models$

$\beta_{k,m}$, and therefore, there exists a state concretization of Ξ of the form $\xi : s_0, \ldots, s_k, \ldots, s_m = s_k = s$. It follows that the infinite concrete run $s_0, \ldots, s_{k-1}(s_k, \ldots, s_{m-1})^\omega$ is a computation of $\mathcal{D}$ that violates ψ. We conclude that ψ is not $\mathcal{D}$-valid.

Case 3. *The infinite abstract run $\Xi_1; \Xi_2^\omega$ cannot be concretized — the abstract counterexample is spurious; perform ranking refinement.*

This case is identified by observing that the bi-assertion $\beta_{k,m}$ is well founded over φ_k. Obviously, if $\Xi_1; \Xi_2^\omega$ could be concretized by the infinite concrete run $s_0, s_1, \ldots$, then we would have had an infinite state sequence, namely $s_k, s_{k+L}, s_{k+2L}, \ldots$, where $L = m - k$, such that $s_k \models \varphi_k$ and $\beta_{k,m}$ holds between any two consecutive states in this sequence. This would have contradicted the fact that $\beta_{k,m}$ is well founded over φ_k. We conclude that the counterexample is spurious.

This case is a typical scenario in *ranking abstraction refinement* – the ranking is too coarse, and should be refined to eliminate the spurious counterexample. The ranking core is refined by adding to it a well-founded ranking that proves the well foundedness of $\beta_{k,m}$ over φ_k. To avoid fluctuation of the ranking throughout a computation (for example, when the value of a variable decreases at each iteration of a control flow loop, but fluctuates at different points within the loop), the refined ranking augmentation is applied with respect $\alpha^{-1}(\Xi[k])$.

A number of methods have been proposed to synthesize such functions from well-founded relations, among them in [PR04a, DGG00]. In Section 4.4 we present an additional heuristic for the domain of unbounded linked lists.

Case 4. *The infinite abstract run $\Xi_1; \Xi_2^\omega$ can be concretized — the property is not valid.*

This case can be identified by observing that the bi-assertion $\beta_{k,m}$ is not well founded over φ_k. From the fact that $\beta_{k,m}$ is not well founded, we can infer the existence of an infinite sequence $s_k, s_{k+L}, s_{k+2L}, \ldots$, where $L = m - k$. This state sequence can be transformed into a computation by filling in the missing states (i.e., $s_1, \ldots, s_{k-1}$ as well as each interval $s_{k+iL}, \ldots, s_{k+(i+1)L}$, $i \geq 0$) to form a concretization of $\Xi_1; \Xi_2^\omega$ that is a computation of $\mathcal{D}$ violating the property ψ.

In this case we can declare the property ψ to be invalid over the concrete program, with the computation $s_1, \ldots, s_k, s_{k+1}, \ldots$ serving as a counterexample.

It can be shown that these four cases cover all the possibilities, even though some of the tests that have to be applied in order to distinguish between the cases are not, in general, decidable.

The process is described in Fig. 4.3. Lines 1 and 2 abstract the system and the property respectively with respect to the ranking core $\mathcal{R}$ and the predicate base $\mathcal{P}$. Line 3 (model-)checks whether the abstract property holds over the abstract program. If so, the algorithm returns "success" (line 4). Else, a

```
Cegar($\mathcal{D}, \psi, \mathcal{P}, \mathcal{R}$)
  1 : Let $\mathcal{D}_A = \mathcal{D}^{\mathcal{R},\alpha}$;
  2 : Let $\psi_A = \psi^{\alpha}$;
  3 : If $\mathcal{D}_A \models \psi_A$ then
        4 : Return "success";
      Else
        5 : Let $C = S_0 \cdots S_{k-1}(S_k \cdots S_{m-1})^{\omega}$ be a computation of $\mathcal{D}_A$
                such that $C \models \neg\psi_A$;
        6 : Let $\Xi = S_0, \ldots, S_k, \ldots, S_{m-1}, S_m = S_k$;
        7 : Compute $\gamma(\Xi) : \varphi_0, \ldots, \varphi_m$, and $\beta_{k,m}(\Xi)$;
        8 : If $(\exists i : 0 \leq i \leq m : \neg sat(\varphi_i))$ then                 —— Case 1
              9 : Let $\mathcal{P}'$ be a predicate refinement of $\mathcal{P}$ induced by the failure
                      to concretize $\Xi$;
             10 : Return Cegar($\mathcal{D}, \psi, \mathcal{P}', \mathcal{R}$);
       11 : Else if $sat(\varphi_k(V) \wedge \beta_{k,m}(V, V))$ then                     —— Case 2
             12 : Let $\xi : s_0, \ldots, s_k, \ldots, s_m = s_k$ be the concrete run concretizing $\Xi$;
             13 : Return "Property not valid. Counterexample: $\xi$";
       14 : Else if there exists well-founded relation $\Psi(V_0, V)$ over $\varphi_k$—— Case 3
                 such that $\beta_{k,m} \subseteq \Psi$ then
             15 : Let $\delta$ be a well-founded ranking proving the well-foundedness
                      of $\beta_{k,m}$ over $\varphi_k$;
             16 : Return Cegar($\mathcal{D}, \psi, P \cup \{\Psi\}, \mathcal{R} \cup \{\langle \delta, \alpha^{-1}(\exists \vec{Dec} \, . \, S_k) \rangle\}$);
       17 : Else return "Property not valid.                                          —— Case 4
                         Counterexample: $s_k, s_{k+L}, s_{k+2L}, \ldots$";
```

Figure 4.3: Counterexample Guided Abstraction Refinement Algorithm

finitely-representable counterexample is produced (by a model checker) in line 5, from which we construct the "lasso" $\Xi : S_0, \ldots, S_k, \ldots, S_{m-1}, S_m = S_k$ at line 6. Line 7 computes the symbolic concretization $\gamma(\Xi)$ and the bi-assertion $\beta_{k,m}(\Xi)$. Line 8 checks whether the symbolic concretization is satisfiable. If it is not satisfiable (Case 1), then predicate refinement is applied (line 9) and the algorithm is re-started with the augmented predicate base (line 10).

If the symbolic concretization is satisfiable, then we check in line 11 whether $\varphi_k(V) \wedge \beta_{k,m}(V, V)$ is satisfiable. If it is satisfiable (Case 2) then we can construct a concrete lasso $\xi : s_0, \ldots, s_k, \ldots, s_{m-1}, s_m = s_k$ concretizing Ξ that is therefore a concrete counterexample.

If the above two tests were answered negatively, we decide in line 14 whether the bi-assertion $\beta_{k,m}(\Xi)$ is well founded over φ_k by searching for a well-founded relation Ψ that contains it. If it is (Case 3) then we know that the abstract counterexample is spurious. In line 15, we attempt to construct a well-founded ranking δ that proves the well foundedness over φ_k of $\beta_{k,m}(\Xi)$. If we succeed, then the ranking core is refined with δ, along with the assertion $\alpha^{-1}(\exists \vec{Dec} \ . \ S_k)$ characterizing $\mathcal{D}$-states in which control (modulo *Dec*-variables) is at the head of the repeating period of the lasso. In addition, the predicate base is extended with the predicate[1] Ψ, which identifies a relationship between the system variables occurring in δ and their copies, which are maintained by the monitor $M_{\delta,\alpha^{-1}(\exists \vec{Dec}.S_k)}$. The algorithm then reiterates with the extended ranking core. This step is the least constructive in the algorithm, and the best that can be offered is a set of heuristics for finding a well-founded ranking that can prove the well foundedness of a given bi-assertion.

Finally, if all preceding tests fail, we reach line 17 (Case 4). In this case, $\beta_{k,m}$ is known not to be well founded over φ_k. This implies that there exists a concrete counter example, but not necessarily one that can be presented in finite terms. The best that we can do is present to the user a prefix of a

[1]In practice, the assertion denoting Ψ is split into its component subformulae, each of which is added as a separate predicate.

potentially infinite counterexample, as explained in the preceding discussion of Case 4.

The algorithm may not terminate (assuming even an extremely powerful model checker). For one, predicate refinement is not guaranteed to terminate. Similarly, ranking refinement may not terminate. Furthermore the test at line 14, which decides whether we are in Case 3 or Case 4, is, in general, undecidable. Thus, to apply this algorithm, we must invoke various heuristics that were designed in order to check whether a given bi-assertion is well founded.

Example 4.4 (Termination of NESTED-LOOPS**)** Recall program NESTED-LOOPS, which was presented in Fig. 3.1(a), and the related termination property expressed as $at\_0 \Longrightarrow \Diamond\, at\_6$. Following Example 4.1, we begin with the initial abstraction and ranking used in Example 3.1. The first iteration of CEGAR results in an abstract counterexample consisting of the single-state prefix S_0 and the repeating period $(S_1, \ldots, S_6)$, where $S_0, \ldots, S_6$ are as in Example 4.1. The abstract lasso derived from this counterexample is $\Xi : S_0, S_1, \ldots, S_6, S_7 = S_1$. We follow the computation of Example 4.2 to obtain the bi-assertions $\beta_{1,1}, \ldots, \beta_{1,6}$, and also compute

$$\beta_{1,7}\colon \mathit{init} \;\wedge\; \pi = 1 \;\wedge\; x = x_0 - 1 \;\wedge\; x > 0 \;\wedge\; y = 0 \;\wedge\; \mathit{dec}_y = 0$$

It follows that $\beta_{1,7}(V_0, V)$ implies $x_0 > x > 0$, which is well founded. A well-founded ranking function proving well foundedness of $\beta_{1,7}$ is $\delta_2 = x$ over the domain $(\mathbb{N}, >)$. Our ranking augmentation construction requires a concrete assertion that identifies the head of the loop, which is where the ranking will be measured[2], for which we use the assertion $\alpha^{-1}(\exists Dec_y \ . \ S_1)$. We refine the predicate base with the predicates $\{x_0 > x, x > 0\}$ and reiterate with the refined ranking core $\mathcal{R}'$: $\{\langle \delta_1, \textsc{True} \rangle, \langle \delta_2, \alpha^{-1}(\exists Dec_y \ . \ S_1) \rangle\}$. At this point, model-checking of the abstract program fails, yielding a spurious counterexample that falls under case 1 (line 8). On refinement with the predicate $(x = x_0)$ and subsequent reiteration, the model-checker successfully verifies termination. ┛

The discussion in this section only considered the case that the sole augmentation applied is a ranking augmentation, while the completeness theorem implies that, in some cases, it is necessary to use a more general progress monitor. However, according to the comments following Theorem 3, there are cases in which we are guaranteed that ranking augmentation is adequate. For all other cases, it is sufficient to take the tester $T[\psi]$ as a standard additional augmentation.

[2]For the program at hand, the ranking augmentation of Section 3.1, which requires no such identification of the loop head, is just as effective.

4.4 Synthesizing Elementary Ranking Functions

A number of methods have been suggested for synthesis of ranking functions that establish (prove) well foundedness of a well-founded bi-assertion. In our examples we have used the simple heuristic of searching for simple linear constraints implied by the transition relation of a control-flow loop ([PR04a] provides a more general method for doing this. Indeed, their method is complete for the reals). For example, given a set of variables V and a bi-assertion β, we check validity of implications such as $\beta \rightarrow v > v'$, for each $v \in V$. As demonstrated, this has been sufficient for dealing program Nested-Loops. A more general approach based on linear algebra may look for ranking functions that are linear combinations of system variables.

Such an extraction is useful in two contexts in which bi-assertions may arise. The first has been demonstrated in the ranking refinement process. The second is related to the determination of the ranking components that should be placed in the initial ranking core. This can be based on a heuristic that analyzes the various loops in the program. Assume a control loop identified by a sequence of locations $L = \ell_1, \ldots, \ell_n = \ell_1$, such that, for each $i = 1, \ldots, n-1$, ℓ_{i+1} can be reached from ℓ_i in a single step. For a loop L, we can define a sequence of bi-assertions as follows:

$$\beta_{i,j} = \begin{cases} V_0 = V \ \wedge \ \pi = \ell_i & j = i \\ (\beta_{i,j-1} \diamond \rho) \ \wedge \ \pi = \ell_j & j > i \end{cases}$$

It only remains to check whether the bi-assertion $\beta_L = \beta_{1,n}$ is well founded, and identify well-founded ranking functions that prove the well foundedness of β_L. Such an identification is, in general, undecidable, but we can use any of the heuristics mentioned above, such as linear analysis.

In Subsection 6.4.3 we use a variant of this heuristic to deal with programs that manipulate unbounded pointer structures.

Chapter 5

Deriving Proofs from Abstractions

5.1 Extracting A Deductive Proof

There are situations in which verification alone is not sufficient, and an actual proof is required. This is the case, for example, when the verification effort is embedded in a larger proof-generating effort, because of either considering only a component of the system, or verifying a property that is only a part of the full specification. When dealing with safety properties, it is straightforward to generate a concrete logical formula that represents an inductive invariant, based on the set of reachable abstract states, to be used as the basis of a deductive proof. The analogous constructs in the case of a response-property proof consist of an assertion that over-approximates the set of pending states, and a well-founded, always-decreasing ranking function.

In this section we present algorithms that extract the necessary auxiliary constructs from a successful application of the ranking abstraction method. The algorithm is based on the LTL model checking algorithm of [LP85] in carrying out a similar analysis of strongly connected components. For simplicity, we consider first the case of an FDS that has no fairness (justice or compassion) requirements. This is typically the case of an FDS derived from a sequential program. We will consider the more general case in which the system has justice requirements in the next section.

5.1.1 Extracting Deductive Proofs of Invariance Properties

For the sake of completeness and emphasizing the analogy between predicate abstraction and ranking abstraction, we present here the process of extracting a deductive proof of an invariance property from a successful application of predicate abstraction. In Fig. 5.1(a), we present the deductive rule INV for establishing the validity of the invariance property $\square\, p$. The application of the rule calls for the identification of an auxiliary assertion φ that, together with p, satisfies premises I1 — I3. We refer to an assertion that satisfies premises I1 and I2 as *inductive*.

Let $\mathcal{D}$ be an FDS for which we wish to verify the invariance property $\psi : \square\, p$. Assume that we employed the predicate abstraction $\alpha : V_A = \mathcal{P}(V)$

Rule INV
For assertions p, φ,

$$\begin{array}{ll} \text{I1.} & \Theta \rightarrow \varphi \\ \text{I2.} & \varphi \wedge \rho \rightarrow \varphi' \\ \text{I3.} & \varphi \rightarrow p \\ \hline & \Box\, p \end{array}$$

(a) Deductive Rule INV

```
Algorithm Extract-invariance ($\mathcal{D}, \alpha$)
  1.  **Compute** $\mathcal{D}^\alpha$;
  2.  **let** $\Phi := \Theta^\alpha \diamond (\rho^\alpha)^*$;
  3.  **let** $\varphi := \alpha^{-1}(\Phi)$;
```

(b) Inductive Assertion Extraction Algorithm

Figure 5.1: Deductive Rule and Extraction Algorithm for Invariance Properties

and verified, by model checking, that $\mathcal{D}^\alpha \models \Box\, p^\alpha$. By soundness of the predicate abstraction method we can conclude that $\mathcal{D} \models \Box\, p$. It only remains to extract a deductive proof of this fact.

In Fig. 5.1(b), we present algorithm Extract-invariance, which extracts an auxiliary assertion φ from the abstracted system $\mathcal{D}^\alpha$. The algorithm first computes, in Φ, an abstract assertion that characterizes all abstract states that are reachable in $\mathcal{D}^\alpha$. It then concretizes Φ into φ by applying the concretization mapping α^{-1}.

The correctness of the algorithm is stated by the following claim:

Claim 5.1 (Extraction of inductive assertion)**.**
For any $\mathcal{D}$ and α, the assertion φ extracted by Algorithm Extract-invariance *is inductive over $\mathcal{D}$. If $\mathcal{D}^\alpha \models \Box\, p^\alpha$ then also $\varphi \rightarrow p$ is valid.*

It follows that if we apply the extraction algorithm to a system after a successful application of the predicate abstraction method, then the extracted assertion φ satisfies all the premises of rule INV.

Example 5.1 (Extraction of a deductive proof of invariance)

Consider program Nested-Loops presented in Fig. 3.1(a). For this program, we wish to prove the invariance of the assertion $p : \pi = 6 \rightarrow \neg(y > 0)$, claiming that when execution reaches location 6, then $y = 0$. Applying the predicate abstraction α introduced in Example 3.1, we obtain the abstract program presented in Fig. 3.2 when we omit all references to variable Dec_y. The property $\square\, p$ is abstracted by α into the abstract property $\square(\Pi = 6 \rightarrow Y \neq 1)$.

Computing the set of reachable states in this abstract program we obtain a set that is captured by the following abstract assertion:

$$\begin{array}{llcll} \Phi : & (X \rightarrow \Pi \in [1..5]) & \wedge & (\Pi \in [2..5] \rightarrow X) & \wedge \\ & (Y \rightarrow \Pi \in [3..4]) & \wedge & (\Pi = 4 \rightarrow Y) & \end{array}$$

Concretizing by α^{-1}, we obtain the following candidate assertion for φ:

$$\begin{array}{llcll} \varphi : & (x > 0 \rightarrow \pi \in [1..5]) & \wedge & (\pi \in [2..5] \rightarrow x > 0) & \wedge \\ & (y > 0 \rightarrow \pi \in [3..4]) & \wedge & (\pi = 4 \rightarrow y > 0) & \end{array}$$

It is not difficult to verify independently that φ is indeed inductive, and that φ implies $\pi = 6 \rightarrow \neg(y > 0)$. ┛

5.1.2 Deductive Rules for Response Properties

Moving to response properties, we consider a property of the form $p \Longrightarrow \diamondsuit\, q$. As previously explained, in this section we focus only on fairness-free FDS's. A basic proof rule BASIC-RESPONSE for the deductive verification of a response property over a fairness-free FDS is presented in Fig. 5.2.

Rule BASIC-RESPONSE
For a well-founded domain $\mathcal{A} : (W, \succ)$,
assertions p, q, φ,
and ranking function $\Delta : \Sigma \mapsto \mathcal{A}$

$$\begin{array}{lccl}
\text{B1.} & p & \Longrightarrow & q \vee \varphi \\
\text{B2.} & \varphi \wedge \rho & \Longrightarrow & q' \ \vee \ (\varphi' \wedge \Delta \succ \Delta') \\
\hline
 & p & \Longrightarrow & \diamondsuit\, q
\end{array}$$

Figure 5.2: Deductive Rule BASIC-RESPONSE

The rule calls for the identification of an auxiliary assertion φ and a ranking function Δ over the well-founded domain $\mathcal{A}$. Assertion φ is intended to be an over-approximation of the set of pending states w.r.t. assertions p and q. It is possible to view this rule as stating that the property $\psi : p \Longrightarrow \diamondsuit\, q$ is valid over $\mathcal{D}$ whenever the transition relation ρ, when restricted to the pending states (or their over-approximation φ), forms a well-founded bi-assertion. The well-founded ranking Δ is a ranking that proves the well foundedness of the bi-assertion derived from ρ.

In practice, it is often useful to partition φ into several disjoint assertions that cover different cases. This leads to rule RESPONSE, which is presented in Fig. 5.3. This rule, as well as BASIC-RESPONSE, has been adapted from [MP91a].

Rule RESPONSE

For a well-founded domain $\mathcal{A} : (W, \succ)$,
assertions $p, q = \varphi_0, \varphi_1, \ldots, \varphi_m$,
and ranking functions $\Delta_0, \Delta_1, \ldots, \Delta_m$ where each $\Delta_i : \Sigma \mapsto \mathcal{A}$

$$\begin{array}{lccl}
\text{R1.} & p & \Longrightarrow & \bigvee_{j=0}^{m} \varphi_j \\
\multicolumn{4}{l}{\text{For each } i = 1, \ldots, m,} \\
\text{R2.} & \varphi_i \wedge \rho & \Longrightarrow & \bigvee_{j=0}^{m} (\varphi'_j \wedge \Delta_i \succ \Delta'_j) \\
\hline
 & p & \Longrightarrow & \Diamond q
\end{array}$$

Figure 5.3: Deductive Rule RESPONSE

The rule uses assertions $\varphi_0, \ldots, \varphi_m$, where $\varphi_0 = q$. It is not difficult to see that if we can find a set of constructs (assertions and ranking functions) satisfying the premises of rule RESPONSE, we can immediately construct the appropriate constructs necessary for rule BASIC-RESPONSE. This can be done by taking

$$\begin{array}{lll}
\varphi & : & \varphi_1 \vee \cdots \vee \varphi_m \\
\Delta & : & \textbf{case} \\
 & & \qquad \begin{array}{rcl} \varphi_1 & : & \Delta_1 \\ \cdots & & \\ \varphi_m & : & \Delta_m \\ \textbf{otherwise} & : & 0 \end{array} \\
 & & \textbf{end-case}
\end{array}$$

It is customary to refer to assertions $\varphi_0, \ldots, \varphi_m$ as the *helpful assertions*.

In the rest of the section we will show how the constructs needed for rule RESPONSE, i.e. the ranks $\Delta_0, \ldots, \Delta_m$ and helpful assertions $\varphi_0, \ldots, \varphi_m$, can be extracted from a successful application of the ranking abstraction method.

5.1.3 Extracting the Ranking Functions

Let $\mathcal{D}$ and α be a concrete system and an abstraction mapping, respectively. Let $\mathcal{R} : \{\delta_1, \ldots, \delta_\ell\}$ be a ranking core. Assume that the specification of $\mathcal{D}$ is given by the response property $\psi : p \Longrightarrow \diamondsuit\, q$. Let $\mathcal{D}^{\mathcal{R},\alpha}$ be the abstract system and ψ^α be the abstract property, and assume that we have established by model checking that $\mathcal{D}^{\mathcal{R},\alpha} \models \psi^\alpha$. We now wish to extract the ranking functions required by rule RESPONSE.

The extraction process proceeds in two steps, where in the first step we extract the ranking functions $0 = \Delta_0, \Delta_1, \ldots, \Delta_m$ and, in the second step, we construct the assertions $q = \varphi_0, \varphi_1, \ldots, \varphi_m$. The well-founded domain $\mathcal{A}$ will be constructed incrementally together with the construction of the Δ_i's.

We start by constructing a transition graph $G\colon \langle N, E\rangle$, whose set of nodes is $N = \mathit{pend} \cup \{g\}$, where *pend* is the set of all pending states of $\mathcal{D}^{\mathcal{R},\alpha}$, and g is a special *goal* node representing all q^α-states that are reachable from a pending state in one step. Recall that the pending states are all the states that are reachable by a q^α-free path from a reachable p^α-state. The edges consist of all transitions connecting one pending state to another. We also include an edge connecting $n \in \mathit{pend}$ to g, if there exists a transition connecting state n to any non-pending state. We will refer to the nodes of the graph as $N = \{g, S_1, \ldots, S_m\}$, where g is the goal node and $S_1, \ldots, S_m$ are the abstract pending states. We refer to G as the *pending graph* of system

$\mathcal{D}^{\mathcal{R},\alpha}$.

The ranking function will be represented as a mapping $\mathit{Rank}\colon N \to$ Tuples, where Tuples is the type of lexicographic tuples whose elements are either natural numbers or ranking functions present in the ranking core $\mathcal{R}$. The ranking *Rank* is initialized as $\mathit{Rank}[n] = \bot$ for each $n \in N$, where $\bot$ is the empty tuple. Then the recursive procedure Rank-Graph(G), shown in Fig. 5.4, is invoked. The algorithm is based on partitioning the transition graph into *maximal strongly connected components* (MSCCs) and recursively dealing with each MSCC. For an MSCC C and a state n, we say that n is a C*-state* when $n \in C$. We denote by $|C|$ the number of nodes in C.

In each iteration, G is decomposed into a sorted set of MSCCs (line 1), where sortedness means that for two components C_i and C_j in the set, $i > j$ if there is an edge from a C_i-node to a C_j-node. The algorithm updates the mapping *Rank* by concatenating natural numbers or elements of the ranking core to the right end of tuples, denoted by the notations $\mathit{Rank}(n) * i$ and $\mathit{Rank}(n) * \delta_j$, respectively. In line 4, each node is assigned a rank based on the index of the MSCC it belongs in.

In line 6, for each non-singleton MSCC C a compassion requirement $\langle \mathit{Dec}_j > 0, \mathit{Dec}_j < 0\rangle$ is found that is violated by C. This means that there exists $n \in C$ such that $n \models (\mathit{Dec}_j > 0)$ and that for any $n \in C$, $n \not\models (\mathit{Dec}_j < 0)$. This search is guaranteed to succeed under the assumption that the model checker has already verified that $\mathcal{D}^{\mathcal{R},\alpha} \models \psi^\alpha$, the reasoning being that, by contradiction, if some component C violates no compassion

```
Algorithm RANK-GRAPH($G, \mathcal{C}, \mathcal{R}$)
Input:   Graph $G = (N, E)$ of pending states for $\mathcal{D}^{\mathcal{R},\alpha}$

         Compassion requirements of $\mathcal{D}^{\mathcal{R},\alpha}$, given by $\mathcal{C} = \left\{\begin{array}{c}\langle Dec_1 > 0, Dec_1 < 0\rangle,\\ \dots,\\ \langle Dec_\ell > 0, Dec_\ell < 0\rangle\end{array}\right\}$

         Ranking core $\mathcal{R} = \{\delta_1, \dots, \delta_\ell\}$
Output: $Rank$, an array $N \to$ TUPLES
Initially: For every $n \in N$, $Rank(n) = \bot$.

RANK($G$):
1 :  Decompose $G$ into a sorted set of MSCCs $G = C_0, ..., C_k$;
2 :  Forall $i \in [0..k]$ do
       3 : Forall $n \in C_i$ do
             4 : $Rank[n] := Rank[n] * i$;
5 :  Forall $i \in [0..k]$ such that $|C_i| > 1$ do
     6 :  Let $j \in [0..\ell]$ such that $\left(\begin{array}{l}\exists n \in C_i \,.\, n \models (Dec_j > 0) \wedge\\ \forall n \in C_i \,.\, n \not\models (Dec_j < 0)\end{array}\right)$
          in
          7 :  Forall $n \in C_i$ do
                 8 : $Rank[n] := Rank[n] * \delta_j$;
          9 :  Let $D$ be the subgraph obtained by removing every edge in
                     $C_i$ leading into a $(Dec_j > 0)$-node
               in
               10 : Call RANK($D$);
```

Figure 5.4: Procedure RANK-GRAPH

requirement, i.e., is *fair*, then a computation of $\mathcal{D}^{\mathcal{R},\alpha}$ exists that enters C and never leaves it. Since all states in C are pending, then the computation does not satisfy ψ^α. Hence any MSCC is necessarily *unfair*.

In line 8 the rank of every state of the current MSCC C_i is updated with the ranking core element δ_j associated with the compassion requirement found in line 6. Finally, C_i is broken (line 9), and the procedure is recursively applied to the resulting subgraph.

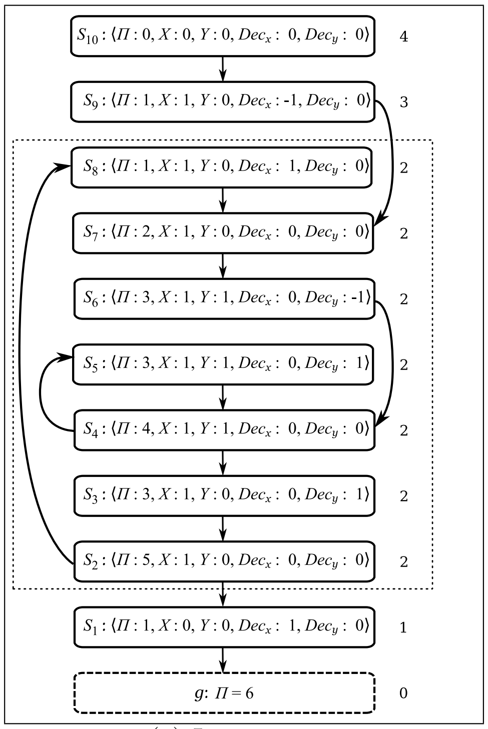
When Algorithm RANK-GRAPH terminates, it produces a list of ranking functions $\Delta_0, \Delta_1, \ldots, \Delta_m$, where Δ_0 is the rank associated with node g (usually 0), while $\Delta_1, \ldots, \Delta_m$ correspond to abstract states $S_1, \ldots, S_m$, respectively.

```
        X, Y          :  {0,1}        init Y = 0, X = 0
        Dec_y, Dec_x  :  {−1,0,1}
compassion {(Dec_x > 0, Dec_x < 0), (Dec_y > 0, Dec_y < 0)}
[ 0 :  (X, Dec_y, Dec_x) :=  (1, 0, −1)
  1 :  while X do
         [ 2 :  (Y, Dec_y, Dec_x) :=  (1, −1, 0)
           3 :  while Y do
                  [ 4 :  (Y, Dec_y, Dec_x) := ({0,1}, 1, 0) ]
           5 :  (X, Dec_y, Dec_x) := ({0,1}, 0, 1) ]
  6 :
]
```

Figure 5.5: Abstract Nested-Loops, Augmented with Ranking Core $\{y, x\}$

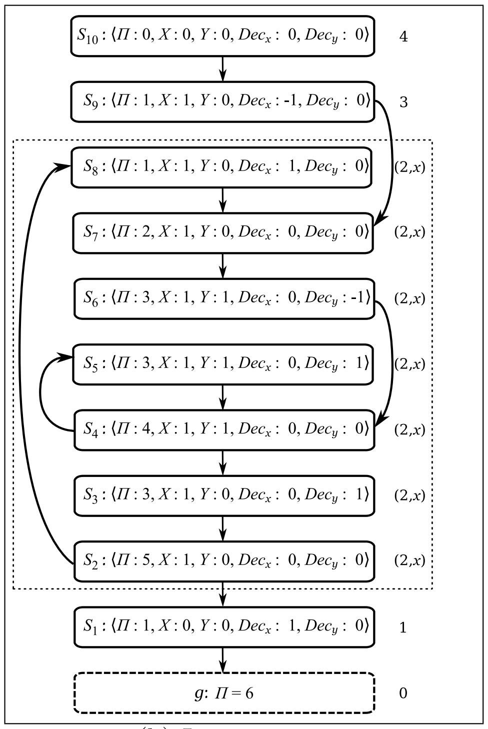
Example 5.2 (Extracting a Ranking Function for Nested-Loops**)** We illustrate the algorithm by extracting the ranking function for the deductive proof of termination of program Nested-Loops, given the abstract program shown in Fig. 5.5. This is a version of the augmented abstract version from Fig. 3.2, after refinement with the additional ranking function $\delta_2 : x$. The response property we wish to verify is that of termination, which can be specified by the formula $at\_0 \Longrightarrow \Diamond\, at\_6$, where, as usual at_i is an abbreviation of the assertion $\pi = i$.

Fig. 5.6 visualizes iterations in the progress of the algorithm, as a series of graphs of the pending states of the abstract FDS, with nodes representing states and directed edges representing transitions. The goal state for the property appears as a graph node that is labeled with $\Pi = 6$ at the bottom



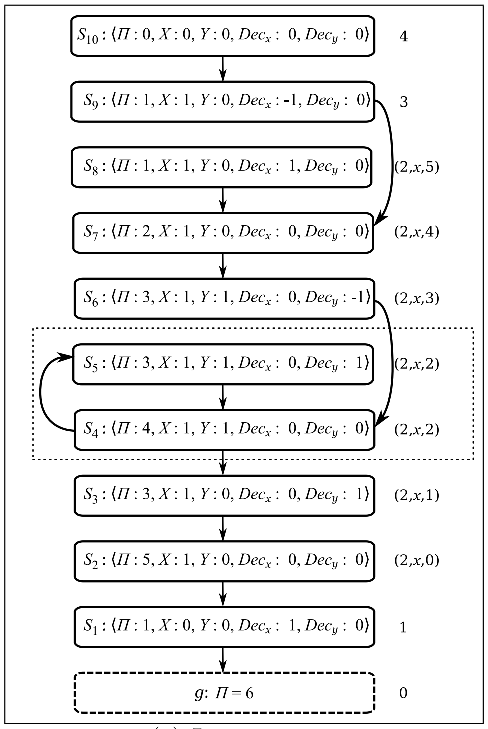
S10 : ⟨Π : 0, X : 0, Y : 0, Dec_x : 0, Dec_y : 0⟩ 4
S9 : ⟨Π : 1, X : 1, Y : 0, Dec_x : -1, Dec_y : 0⟩ 3
S8 : ⟨Π : 1, X : 1, Y : 0, Dec_x : 1, Dec_y : 0⟩ 2
S7 : ⟨Π : 2, X : 1, Y : 0, Dec_x : 0, Dec_y : 0⟩ 2
S6 : ⟨Π : 3, X : 1, Y : 1, Dec_x : 0, Dec_y : -1⟩ 2
S5 : ⟨Π : 3, X : 1, Y : 1, Dec_x : 0, Dec_y : 1⟩ 2
S4 : ⟨Π : 4, X : 1, Y : 1, Dec_x : 0, Dec_y : 0⟩ 2
S3 : ⟨Π : 3, X : 1, Y : 0, Dec_x : 0, Dec_y : 1⟩ 2
S2 : ⟨Π : 5, X : 1, Y : 0, Dec_x : 0, Dec_y : 0⟩ 2
S1 : ⟨Π : 1, X : 0, Y : 0, Dec_x : 1, Dec_y : 0⟩ 1
g: Π = 6 0

(a) Iteration 1



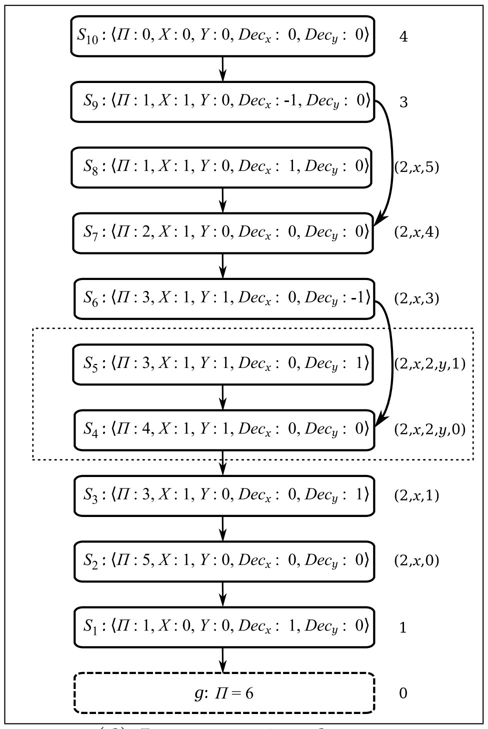
S10 : ⟨Π : 0, X : 0, Y : 0, Dec_x : 0, Dec_y : 0⟩ 4
S9 : ⟨Π : 1, X : 1, Y : 0, Dec_x : -1, Dec_y : 0⟩ 3
S8 : ⟨Π : 1, X : 1, Y : 0, Dec_x : 1, Dec_y : 0⟩ (2,x)
S7 : ⟨Π : 2, X : 1, Y : 0, Dec_x : 0, Dec_y : 0⟩ (2,x)
S6 : ⟨Π : 3, X : 1, Y : 1, Dec_x : 0, Dec_y : -1⟩ (2,x)
S5 : ⟨Π : 3, X : 1, Y : 1, Dec_x : 0, Dec_y : 1⟩ (2,x)
S4 : ⟨Π : 4, X : 1, Y : 1, Dec_x : 0, Dec_y : 0⟩ (2,x)
S3 : ⟨Π : 3, X : 1, Y : 0, Dec_x : 0, Dec_y : 1⟩ (2,x)
S2 : ⟨Π : 5, X : 1, Y : 0, Dec_x : 0, Dec_y : 0⟩ (2,x)
S1 : ⟨Π : 1, X : 0, Y : 0, Dec_x : 1, Dec_y : 0⟩ 1
g: Π = 6 0

(b) Iteration 2



S10 : ⟨Π : 0, X : 0, Y : 0, Dec_x : 0, Dec_y : 0⟩ 4
S9 : ⟨Π : 1, X : 1, Y : 0, Dec_x : -1, Dec_y : 0⟩ 3
S8 : ⟨Π : 1, X : 1, Y : 0, Dec_x : 1, Dec_y : 0⟩ (2,x,5)
S7 : ⟨Π : 2, X : 1, Y : 0, Dec_x : 0, Dec_y : 0⟩ (2,x,4)
S6 : ⟨Π : 3, X : 1, Y : 1, Dec_x : 0, Dec_y : -1⟩ (2,x,3)
S5 : ⟨Π : 3, X : 1, Y : 1, Dec_x : 0, Dec_y : 1⟩ (2,x,2)
S4 : ⟨Π : 4, X : 1, Y : 1, Dec_x : 0, Dec_y : 0⟩ (2,x,2)
S3 : ⟨Π : 3, X : 1, Y : 0, Dec_x : 0, Dec_y : 1⟩ (2,x,1)
S2 : ⟨Π : 5, X : 1, Y : 0, Dec_x : 0, Dec_y : 0⟩ (2,x,0)
S1 : ⟨Π : 1, X : 0, Y : 0, Dec_x : 1, Dec_y : 0⟩ 1
g: Π = 6 0

(c) Iteration 3



S10 : ⟨Π : 0, X : 0, Y : 0, Dec_x : 0, Dec_y : 0⟩ 4
S9 : ⟨Π : 1, X : 1, Y : 0, Dec_x : -1, Dec_y : 0⟩ 3
S8 : ⟨Π : 1, X : 1, Y : 0, Dec_x : 1, Dec_y : 0⟩ (2,x,5)
S7 : ⟨Π : 2, X : 1, Y : 0, Dec_x : 0, Dec_y : 0⟩ (2,x,4)
S6 : ⟨Π : 3, X : 1, Y : 1, Dec_x : 0, Dec_y : -1⟩ (2,x,3)
S5 : ⟨Π : 3, X : 1, Y : 1, Dec_x : 0, Dec_y : 1⟩ (2,x,2,y,1)
S4 : ⟨Π : 4, X : 1, Y : 1, Dec_x : 0, Dec_y : 0⟩ (2,x,2,y,0)
S3 : ⟨Π : 3, X : 1, Y : 0, Dec_x : 0, Dec_y : 1⟩ (2,x,1)
S2 : ⟨Π : 5, X : 1, Y : 0, Dec_x : 0, Dec_y : 0⟩ (2,x,0)
S1 : ⟨Π : 1, X : 0, Y : 0, Dec_x : 1, Dec_y : 0⟩ 1
g: Π = 6 0

(d) Iterations 4 and 5

Figure 5.6: Progress of Algorithm RANK-GRAPH for NESTED-LOOPS

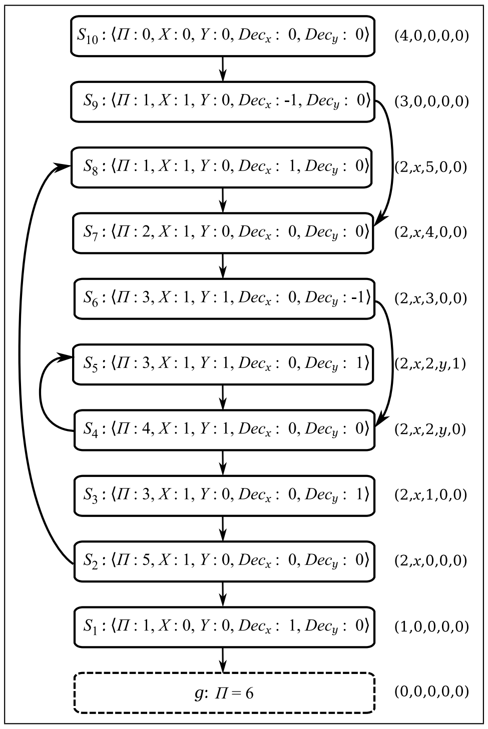

$S_{10} : \langle \Pi : 0, X : 0, Y : 0, Dec_x : 0, Dec_y : 0 \rangle$ (4,0,0,0,0)
$S_9 : \langle \Pi : 1, X : 1, Y : 0, Dec_x : -1, Dec_y : 0 \rangle$ (3,0,0,0,0)
$S_8 : \langle \Pi : 1, X : 1, Y : 0, Dec_x : 1, Dec_y : 0 \rangle$ (2,x,5,0,0)
$S_7 : \langle \Pi : 2, X : 1, Y : 0, Dec_x : 0, Dec_y : 0 \rangle$ (2,x,4,0,0)
$S_6 : \langle \Pi : 3, X : 1, Y : 1, Dec_x : 0, Dec_y : -1 \rangle$ (2,x,3,0,0)
$S_5 : \langle \Pi : 3, X : 1, Y : 1, Dec_x : 0, Dec_y : 1 \rangle$ (2,x,2,y,1)
$S_4 : \langle \Pi : 4, X : 1, Y : 1, Dec_x : 0, Dec_y : 0 \rangle$ (2,x,2,y,0)
$S_3 : \langle \Pi : 3, X : 1, Y : 0, Dec_x : 0, Dec_y : 1 \rangle$ (2,x,1,0,0)
$S_2 : \langle \Pi : 5, X : 1, Y : 0, Dec_x : 0, Dec_y : 0 \rangle$ (2,x,0,0,0)
$S_1 : \langle \Pi : 1, X : 0, Y : 0, Dec_x : 1, Dec_y : 0 \rangle$ (1,0,0,0,0)
g: $\Pi = 6$ (0,0,0,0,0)

Figure 5.7: End Result of Rank-Graph for Nested-Loops

of each diagram.

As the algorithm proceeds, each node is associated with a tuple that denotes the ranking generated thus far.

Table 5.1 summarizes the process as a table. The ranking procedure proceeds as follows: Initially the graph of Fig. 5.6 is decomposed into components $0, \ldots, 4$, and nodes are assigned ranks according to the index of their MSCC. This is shown in Fig. 5.6(a) and in the "Iteration 1" column of Table 5.1. Since nodes $\{S_2, \ldots, S_8\}$ form an MSCC that violates the compassion requirement $\langle Dec_x > 0, Dec_x < 0 \rangle$, the corresponding ranking function x is appended to their ranking tuples, as shown in Fig. 5.6(b) and in the

node	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Final Ranking
S_{10}	(4)					$(4,0,0,0,0)$
S_9	(3)					$(3,0,0,0,0)$
S_8	(2)	$(2,x)$	$(2,x,5)$			$(2,x,5,0,0)$
S_7	(2)	$(2,x)$	$(2,x,4)$			$(2,x,4,0,0)$
S_6	(2)	$(2,x)$	$(2,x,3)$			$(2,x,3,0,0)$
S_5	(2)	$(2,x)$	$(2,x,2)$	$(2,x,2,y)$	$(2,x,2,y,1)$	$(2,x,2,y,1)$
S_4	(2)	$(2,x)$	$(2,x,2)$	$(2,x,2,y)$	$(2,x,2,y,0)$	$(2,x,2,y,0)$
S_3	(2)	$(2,x)$	$(2,x,1)$			$(2,x,1,0,0)$
S_2	(2)	$(2,x)$	$(2,x,0)$			$(2,x,0,0,0)$
S_1	(1)					$(1,0,0,0,0)$
g	(0)					$(0,0,0,0,0)$

Table 5.1: Iterative Ranking for NESTED-LOOPS

"Iteration 2" column. The ranking procedure is now applied recursively to the subgraph that consists of nodes $\{S_2, \ldots .S_8\}$ and all edges except for the one entering the $(Dec_x > 0)$ node (S_8). The subgraph, which is no longer strongly connected due to edge removal, is re-decomposed into MSCCs, and each tuple is updated (by concatenation to the right) with the new MSCC indices (shown in Fig. 5.6(c) and in the "Iteration 3" column). The component consisting of nodes S_4 and S_5 is now a non-singleton MSCC, and it violates the compassion requirement $\langle Dec_y > 0, Dec_y < 0\rangle$. Therefore, the corresponding ranking function y is appended to the rankings of nodes S_4 and S_5, as shown in Fig. 5.6(d) and in the "Iteration 4" column. Again, the procedure is applied recursively to the subgraph with nodes S_4 and S_5 that has all edges but the one leading into the $(Dec_y > 0)$ node (S_5). The rankings of nodes S_4 and S_5 are appended index values corresponding to a new sorting (Fig. 5.6(d) and column "Iteration 5"). At this point there are no non-singleton MSCCs left in the graph and the procedure terminates. The final ranking, with zeroes padded to the right where appropriate, is shown in Fig. 5.7 and in the "final ranking" column. ┛

Correctness of the Algorithm

In order to state the correctness of the algorithm we require new terminology. Let $\Delta_i = (a_1, \ldots, a_r)$ and $\Delta_j = (b_1, \ldots, b_r)$ be two ranks. The formula

$$gt(\Delta_i, \Delta_j) : \quad \bigvee_{k=1}^{r} (a_1 = b'_1) \ \wedge \ \cdots \ \wedge \ (a_{k-1} = b'_{k-1}) \ \wedge \ (a_k \succ b'_k)$$

formalizes the condition for $\Delta_i \succ \Delta'_j$ in lexicographic order. In general, we cannot determine whether the formula $gt(\Delta_i, \Delta_j)$ is true or false, because some of the a_k, b_k can be functions such as x or y. Let E be a consistent conjunction whose conjuncts are expressions of the form $\delta_k = \delta'_k$ or $\delta_k \succ \delta'_k$ for some $\delta_k \in \mathcal{R}$.

Definition 5.2. *We say that* Δ_i dominates Δ_j under E*, written* $\Delta_i \succ_E \Delta_j$*, if the implication* $E \rightarrow gt(\Delta_i, \Delta_j)$ *is valid.*

Thus, E lists some assumptions about the relations between δ_k, E, and δ'_k under which $gt(\Delta_i, \Delta_j)$ can be evaluated. For example, for $\Delta_i : (2, x, 2, y, 0)$, $\Delta_j : (2, x, 2, y, 1)$ and $E : x = x' \wedge y > y'$, Δ_i dominates Δ_j under E, that is, $\Delta_i \succ_E \Delta_j$. A special environment is $E_0 = \bigwedge_{\delta_k \in \mathcal{R}} (\delta_k = \delta'_k)$, which assumes that all core-ranking components are equal. If Δ_i dominates Δ_j

under E_0, we denote this fact by $\Delta_i > \Delta_j$. Each abstract state S induces an environment, denoted $E(S)$, which, for each $\delta_k \in \mathcal{R}$, contains the equality $\delta_k = \delta'_k$ iff $S[Dec_k] = 0$, and contains the inequality $\delta_k \succ \delta'_k$ iff $S[Dec_k] = 1$. We denote the fact that S_i dominates S_j under the environment $E(S)$ by writing $S_i \succ_S S_j$.

For example, consider the abstract states $S_4 : \langle \Pi{:}4,\ X{:}1,\ Y{:}1,\ Dec_x{:}0,\ Dec_y{:}0\rangle$ and $S_5 : \langle \Pi{:}3,\ X{:}1,\ Y{:}1,\ Dec_x{:}0,\ Dec_y{:}1\rangle$, and the ranks $\Delta_4 : (2, x, 2, y, 0)$ and $\Delta_5 : (2, x, 2, y, 1)$ associated with them. According to the definition $E(S_5) = (x = x' \wedge y > y')$ and $E(S_4) = (x = x' \wedge y = y')$. It follows that both $\Delta_4 \succ_{S_5} \Delta_5$ and $\Delta_5 \succ_{S_4} \Delta_4$ are true.

Partial correctness of procedure RANK-GRAPH is stated by the following lemma, which is the basis for the main correctness theorem given in Subsection 5.1.5.

Lemma 5.3 (Correctness of RANK-GRAPH)**.** *The following properties of pending graphs and their final rankings are valid:*

P1. *There is a rank decrease $\Delta_i \succ_{S_j} \Delta_j$ across every edge (S_i, S_j), with associated ranks Δ_i and Δ_j.*

P2. *If Δ_i and Δ_j are the ranks associated with states S_i and S_j, respectively, then $S_i \neq S_j$ implies $\Delta_i \neq \Delta_j$.*

P3. *The relation $>$ between ranks is a total order over the set of final ranks computed by Algorithm* RANK-GRAPH.

P4. *If $\Delta_i \succ_{S_j} \Delta_j$ and $\Delta_j > \Delta_k$, then $\Delta_i \succ_{S_j} \Delta_k$.*

P5. *If states S_i and S_j agree on the values of their non-Dec variables (i.e., those not contributed by augmentations with progress monitors), then they have the same set of successors.*

Proof. Property **P4** follows from the definition of $\succ_{S_j}$ and $>$. Property **P5** follows from the construction of progress monitors and abstract FDS's.

As for properties **P1**, **P2**, and **P3**, we prove the claim by induction on the maximal number of nested recursive steps in a call to Rank, that $\psi : \mathbf{P1} \ \wedge \ \mathbf{P2} \ \wedge \ \mathbf{P3}$ is a postcondition of Rank. In the base case, when a call to Rank entails no recursive calls, we conclude that G consists only of singleton MSCC's. Thus, ψ holds on termination of the loop of line 2.

In the general case, we again consider the loop of line 2. On termination of the loop, it is easy to see that ψ holds in G, when restricted to state pairs (S, S') belonging to *disjoint* MSCC's. From the inductive hypothesis, following a recursive call in line 10, ψ holds in D, i.e., in the subgraph C_i in which all edges leading into a $(Dec_j > 0)$-state have been removed. Thus ψ holds in C_i, excluding states that satisfy $(Dec_j > 0)$. We thus show that ψ holds for an edge (S, S') in C_i such that $S' \models (Dec_j > 0)$. This follows from the fact that for some ranking prefix σ, due to the assignment in line 8, for every $S \in C_i$, $Rank[S]$ has the prefix $\sigma * \delta_j$. Thus for any such S, $S \succ_{S'} S'$.

Since the loop of line 5 makes a recursive call for every non-singleton component of G, then on termination of this loop, ψ holds for all pairs of

G-states. □

To complete the correctness proof of Procedure Rank-Graph, we prove:

Theorem 5.4. *Procedure* Rank-Graph *terminates.*

Proof. The claim follows from the fact that there is a strict decrease in the size of the input graph on every recursive call to Rank. This is easy to see from the definition of the graph D (line 9), which breaks a component C_i into at least two MSCC's. □

5.1.4 Forming an Abstract Verification Diagram

The ranked pending graph still contains too many details. In particular, it assigns different ranks to two abstract states that agree on all non-*Dec* variables. The states $S_5 : \langle \Pi{:}3,\ X{:}1,\ Y{:}1,\ Dec_x{:}0,\ Dec_y{:}1\rangle$ and $S_6 : \langle \Pi{:}3,\ X{:}1,\ Y{:}1,\ Dec_x{:}0,\ Dec_y{:}-1\rangle$, which are assigned different ranks in Fig. 5.7, are an example of this. In the next step, we group together all abstract states that agree on the values of all non-*Dec* variables.

To eliminate this redundant distinction, we form an *abstract verification diagram* in the spirit of [MP94]. This is a directed graph whose nodes are labeled with assertions $\Phi_0, \Phi_1, \ldots, \Phi_m$ and are also ranked by a well-founded ranking. Such diagrams are often used to provide a succinct representation of the auxiliary constructs needed for a proof rule such as Response.

The abstract verification diagram is constructed as follows:

1. For each set of states in the ranked pending graph that agree on the values of their non-*Dec* variables we construct a node and label it by an assertion Φ. Assertion Φ is a conjunction that specifies the values of all non-*Dec* variables.

   Thus, the set consisting of state $S_5 : \langle \Pi{:}3,\ X{:}1,\ Y{:}1,\ Dec_x{:}0,\ Dec_y{:}1\rangle$ and state $S_6 : \langle \Pi{:}3,\ X{:}1,\ Y{:}1,\ Dec_x{:}0,\ Dec_y{:}-1\rangle$ will be represented by a single node labeled by the assertion $\Phi : \Pi = 3 \wedge X = 1 \wedge Y = 1$.

   For simplicity, we write $S \in \Phi$ as synonymous to $S \models \Phi$.

2. We draw an edge from the node (labeled by the assertion) Φ_i to node Φ_j, whenever there are states $S_i \in \Phi_i$ and $S_j \in \Phi_j$ such that S_i is connected to S_j in the ranked pending graph.

3. A node Φ is ranked by a rank Δ that is the $>$-minimum among the ranks associated with the states that are grouped in Φ.

   Thus, the rank assigned to node $\Phi : \Pi = 3 \wedge X = 1 \wedge Y = 1$, which has been obtained by grouping together states S_5 and S_6 with associated ranks $(2, x, 2, y, 1)$ and $(2, x, 3)$, is $(2, x, 2, y, 1)$, which is the smaller of the two ranks.

In Fig. 5.8(a) we present the abstract verification diagram obtained for program NESTED-LOOPS.

We state the correctness of the construction in the following lemma.

Lemma 5.5. *If Φ_i is connected to Φ_j in the verification diagram and $S_j \in \Phi_j$, then $\Delta_i \succ_{S_j} \Delta_j$, where Δ_i, Δ_j are the ranks associated with Φ_i, Φ_j, respectively.*

Proof. From property **P5** and from the construction of the abstract verification diagram, S_j is the successor of any $S_i \in \Phi_i$. Thus we choose a state S_i that is associated in the pending graph with the ranking Δ_i. From property **P1** we have $\Delta_i \succ_{S_j} \Delta_{S_j}$, where Δ_{S_j} is the ranking associated in the pending graph with S_j. From construction of the abstract verification diagram, we have $\Delta_{S_j} > \Delta_j$. Thus the claim follows from the "semi-transitivity" of property **P4**. $\square$

5.1.5 Obtaining the Concrete Helpful Assertions

As the last step in the extraction of the auxiliary constructs needed by rule RESPONSE, we compute the concrete helpful assertions $\varphi_0, \ldots, \varphi_m$. These are obtained simply by concretization of the abstract assertions $\Phi_0, \ldots, \Phi_m$. That is, for each $i = 0, \ldots, m$, we let $\varphi_i = \alpha^{-1}(\Phi_i)$. Thus, for program NESTED-LOOPS, we obtain the helpful assertions presented in the table of Fig. 5.8(b).

A property that leads to the overall correctness of the construction is given by:

Lemma 5.6. *If for some concrete pending states s_i and s_j, $s_i \models \varphi_i$ and $s_j \models \varphi_j$, and s_j is a $\mathcal{D}$-successor of s_i, then $s_i[\Delta_i] \succ s_j[\Delta_j]$.*

Proof. Since s_j is a $\mathcal{D}$-successor of s_i, there exist $\widetilde{s}_i, \widetilde{s}_j$, pending states of the augmented system $\mathcal{D}$+$\mathcal{R}$, such that $\widetilde{s}_i \models \varphi_i, \widetilde{s}_j \models \varphi_j$, and $\widetilde{s}_j$ is a $\mathcal{D}$+$\mathcal{R}$-successor of $\widetilde{s}_i$. Abstracting these states by α, we obtain abstract states S_i and S_j, such that $S_i \models \Phi_i, S_j \models \Phi_j$, and S_j is a $\mathcal{D}^{\mathcal{R},\alpha}$-successor of S_i (and hence Φ_i is connected to Φ_j in the abstract verification diagram). By Lemma 5.5, it follows that $\Delta_i \succ_{S_j} \Delta_j$. The definition of $\succ_{S_j}$ implies that $\Delta_i \succ \Delta_j$ under the assumption that every ranking component $\delta_k \in \mathcal{R}$ decreases or increases as determined by the value of variable Dec_k in state S_j. Since, by definition, the abstraction α preserves the value between Dec_k and its concrete counterpart dec_k in $\widetilde{s}_j$, then Dec_k faithfully represents whether δ_k decreases or increases on the transition from s_i to s_j. It therefore follows that $s_i[\Delta_i] \succ s_j[\Delta_j]$. $\square$

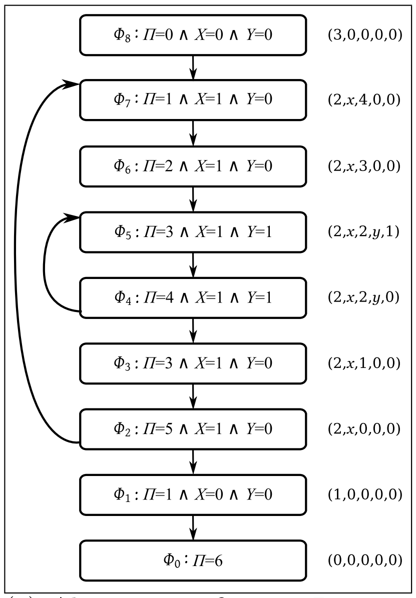
We can now summarize the complete algorithm for extraction of the auxiliary constructs necessary for a successful application of rule Response.

1. Construct the pending graph and apply Algorithm Rank-Graph in order to assign well-founded ranks to the nodes in the graph.

2. Construct an abstract verification diagram by abstracting away the *Dec*-variables.

3. Derive the helpful assertions by concretization of the assertions labeling the nodes in the abstract verification diagrams.

Theorem 5.7 (Correctness of the Extraction Algorithm)**.** *Let* $\Delta_0, \dots, \Delta_m$ *be the ranking functions generated by Procedure* Rank-Graph*, and* $\varphi_0, \dots, \varphi_m$



$\Phi_8 : \Pi{=}0 \wedge X{=}0 \wedge Y{=}0$ (3,0,0,0,0)
$\Phi_7 : \Pi{=}1 \wedge X{=}1 \wedge Y{=}0$ (2,x,4,0,0)
$\Phi_6 : \Pi{=}2 \wedge X{=}1 \wedge Y{=}0$ (2,x,3,0,0)
$\Phi_5 : \Pi{=}3 \wedge X{=}1 \wedge Y{=}1$ (2,x,2,y,1)
$\Phi_4 : \Pi{=}4 \wedge X{=}1 \wedge Y{=}1$ (2,x,2,y,0)
$\Phi_3 : \Pi{=}3 \wedge X{=}1 \wedge Y{=}0$ (2,x,1,0,0)
$\Phi_2 : \Pi{=}5 \wedge X{=}1 \wedge Y{=}0$ (2,x,0,0,0)
$\Phi_1 : \Pi{=}1 \wedge X{=}0 \wedge Y{=}0$ (1,0,0,0,0)
$\Phi_0 : \Pi{=}6$ (0,0,0,0,0)

(a) Abstract Verification Diagram for NESTED-LOOPS

Index	φ_i	Δ_i
8	$\pi = 0 \wedge x = 0 \wedge y = 0$	$(3, 0, 0, 0, 0)$
7	$\pi = 1 \wedge x > 0 \wedge y = 0$	$(2, x, 4, 0, 0)$
6	$\pi = 2 \wedge x > 0 \wedge y = 0$	$(2, x, 3, 0, 0)$
5	$\pi = 3 \wedge x > 0 \wedge y > 0$	$(2, x, 2, y, 1)$
4	$\pi = 4 \wedge x > 0 \wedge y > 0$	$(2, x, 2, y, 0)$
3	$\pi = 3 \wedge x > 0 \wedge y = 0$	$(2, x, 1, 0, 0)$
2	$\pi = 5 \wedge x > 0 \wedge y = 0$	$(2, x, 0, 0, 0)$
1	$\pi = 1 \wedge x = 0 \wedge y = 0$	$(1, 0, 0, 0, 0)$
0	$\pi = 6$	$(0, 0, 0, 0, 0)$

(b) Assertions and Rankings for NESTED-LOOPS

Figure 5.8: Simplified Result of RANK-GRAPH with Concrete Helpful Assertions

the assertions resulting from concretization of $\Phi_0, \ldots, \Phi_m$. *For assertions* p *and* q, *if* $\mathcal{D} \models p \implies \diamondsuit\, q$ *then premises R1 and R2 of rule* RESPONSE *are valid.*

Proof. Premise R1 requires to show that one of $\varphi_0, \ldots, \varphi_m$ is implied by p. This follows trivially from the soundness of abstraction and construction of the pending graph, which, by definition, contains a state S such that $S \models \alpha(p)$.

As for premise R2, by construction of the pending graph, for every successor of a concrete pending state there exists an edge to a corresponding node (representing either an abstract pending state, or an abstract goal state).

Therefore, from the construction of $\varphi_0, \ldots, \varphi_m$, we have $\varphi_i \ \wedge \ \rho \implies \varphi_j$, for some $j \in [0..m]$. From Lemma 5.6 we then have $\Delta_i \succ \Delta'_j$ if $\varphi_i \ \wedge \ \rho$. $\square$

5.2 Extracting Proofs for Systems with Justice Requirements

In the previous section we showed how to extract a deductive proof of response properties for fairness-free FDS's, that are adequate for representing sequential programs. In this Section we extend the method to FDS's containing justice requirements that can, therefore, represent the majority of concurrent programs.

Consequently, in this section we focus on the class of *Just Discrete Systems* (JDS), that allow an arbitrary number of justice requirements, but no native compassion requirements. The ranking abstraction method introduces its own compassion requirements into the augmented system prior to abstraction, but we allow no compassion requirements in the original system $\mathcal{D}$. A further extension of the method will consider the most general case of FDS's, which allows both justice and compassion requirements. This will be addressed in future research.

5.2.1 A Deductive Rule for Response under Justice

In the case of fairness-free systems, we could require a well-founded ranking that decreases on *every* execution step. This is no longer possible in the presence of justice requirements. Here, we partition the space of pending steps into regions characterized by assertions $\varphi_1, \ldots . \varphi_m$ where for each $i = 1, \ldots, m$, $\varphi_i \rightarrow \neg J_i$, so that any step from a φ_i-state that causes J_i to be fulfilled should cause the ranking to decrease. However, as long as we remain within φ_i, we may take an arbitrary number of steps and the rank need not decrease.

In Fig. 5.9 we present proof rule JUST-RESPONSE [MP91a], which establishes the response property $p \Longrightarrow \diamondsuit\, q$ for a JDS $\mathcal{D}$. Premise R1 of the rule requires that any p-state is also a φ_i-state for some $i = 0, \ldots, m$. Premise R2 of the rule requires that any step from a φ_i-state $(i > 0)$ either causes the ranking to decrease or preserves the value of Δ_i, provided we stay in the φ_i-region. By premise R3, justice requirement J_i is not satisfied by any φ_i-state. It follows that any infinite run that enters the pending domain without ever leaving it, either causes the ranking to decrease infinitely many times, which is impossible, or remains forever within some φ_i-region from a certain point on. However, in the latter case, justice requirement J_i will be satisfied only finitely many times. It follows that such a run cannot be a computation as it violates the justice requirement J_i. We conclude that no computation stays contained forever within the domain of pending states. Hence a computation that enters the pending domain must eventually exit, and satisfy q.

Rule JUST-RESPONSE
For a well-founded domain $\mathcal{A} : (W, \succ)$,
assertions $p, q = \varphi_0, \varphi_1, \ldots, \varphi_m$,
justice requirements $J_1, \ldots, J_m$,
and ranking functions $\Delta_0, \Delta_1, \ldots, \Delta_m$ where each $\Delta_i : \Sigma \mapsto \mathcal{A}$

$$
\begin{array}{lccl}
\text{R1.} & p & \Longrightarrow & \bigvee_{i=0}^{m} \varphi_i \\
\multicolumn{4}{l}{\text{For each } i = 1, \ldots, m,} \\
\text{R2.} & \varphi_i \wedge \rho & \Longrightarrow & (\varphi_i' \wedge \Delta_i = \Delta_i') \ \vee \ \bigvee_{j=0}^{m} (\varphi_j' \wedge \Delta_i \succ \Delta_j') \\
\text{R3.} & \varphi_i & \Longrightarrow & \neg J_i \\
\hline
 & p & \Longrightarrow & \diamondsuit\, q
\end{array}
$$

Figure 5.9: Deductive Rule JUST-RESPONSE

5.2.2 Ranking Abstraction and Concurrent Programs

The method of ranking abstraction can be applied, with no change, to JDS's. We illustrate this application on program UP-DOWN, presented in Fig. 5.10, for which we wish to prove the response property $(\pi_1 = 0 \wedge \pi_2 = 0) \Longrightarrow \diamondsuit(\pi_1 = 4)$, where π_1 and π_2 are the location counters for P_1 and P_2 respectively. To distinguish between locations of processes P_1 and P_2, we denote them by ℓ_i, and m_j, respectively. We also use the notation at_ℓ_i to denote $\pi_1 = i$, and, similarly, we use at_m_j to denote $\pi_2 = j$. The justice requirements of this program are given by $\mathcal{J} = \{\neg at\_\ell_0, \neg at\_\ell_1, \neg at\_\ell_2, \neg at\_\ell_3, \neg at\_m_0\}$. Thus, every statement at location ℓ (i.e., ℓ_i or m_j) is associated with a justice requirement of the form $\neg at\_\ell$, guaranteeing that the statement is eventually executed, and execution does not remain stuck at ℓ forever.

Employing the predicate base $\mathcal{P} : \{x > 0, y > 0\}$ and the ranking core

```
$x, y$ : natural init $x = 0, y = 1$

P_1 ::  $\ell_0$ :  while $x = 0$ do
              [$\ell_1$ : $y := y + 1$]
        $\ell_2$ :  while $y > 0$ do
              [$\ell_3$ : $y := y - 1$]
        $\ell_4$ :
                                        ||   P_2 ::  $m_0$ :  $x := 1$
                                                     $m_1$ :
```

Figure 5.10: Program Up-Down

$\mathcal{R} : \{\delta_y : y\}$, we obtain the abstraction

$$\alpha: \quad \Pi_1 = \pi_1,\ \Pi_2 = \pi_2,\ X = (x > 0),\ Y = (y > 0),\ Dec_y = dec_y,$$

Augmenting and abstracting program Up-Down, we obtain the abstract program Abstract-Up-Down, presented in Fig. 5.11.

```
$X, Y$ :  natural init $X = 0, Y = 1$
$Dec_y$ :  $\{-1, 0, 1\}$
compassion $(\boldsymbol{Dec}_y > 0, \boldsymbol{Dec}_y < 0)$

P_1 ::  $\ell_0$ :  while $X = 0$ do
              [$\ell_1$ : $(Y, Dec_y) := (1, -1)$]
        $\ell_2$ :  while $Y = 1$ do
              [$\ell_3$ : $(Y, Dec_y) := (Y \dot{-} 1, (Y > 0))$]
        $\ell_4$ :
                                        ||   P_2 ::  $m_0$ :  $X := 1$
                                                     $m_1$ :
```

Figure 5.11: Program Abstract-Up-Down

In the program, the operation $\dot{-}$ is defined by $Y \dot{-} 1 = \max(Y - 1, 0)$. The justice requirements of the abstract program are the same as for the concrete program – $\mathcal{J} = \{\neg at\_\ell_0, \neg at\_\ell_1, \neg at\_\ell_2, \neg at\_\ell_3, \neg at\_m_0\}$, except that at_ℓ_i and at_m_j are now interpreted as $\Pi_1 = i$ and $\Pi_2 = j$, respectively.

Model checking the abstracted property $\Psi^\alpha : (\Pi_1 = 0 \wedge \Pi_2 = 0) \Longrightarrow \diamondsuit(\Pi_1 = 4)$ over program ABSTRACT-UP-DOWN, we find out that it is valid. We conclude that the concrete program UP-DOWN terminates.

5.2.3 Extracting a Deductive Proof

We proceed to outline the algorithm by which we can extract the necessary ingredients for a deductive proof by rule JUST-RESPONSE from a successful application of the ranking abstraction method. As in the fairness-free case, the process proceeds in three steps. Initially we construct the pending graph, and assign ranks to the abstract states belonging to this graph. This step also assigns a helpful justice requirement to each abstract state (as required by rule JUST-RESPONSE). The second step constructs an abstract verification diagram, which contains abstracted versions of the helpful assertions. In the third and final step we construct the (concrete) helpful assertions.

Once again, we start by constructing a transition graph $G\colon \langle N, E\rangle$, which represents the set of pending states plus a goal state. In the just version of the construction, it is important to label the edges by a label that can be viewed either as the transition that leads from one state to the next, or the justice requirement that the transition causes to be satisfied. This correspondence results from the fact that every transition τ in the program can be associated with a justice requirement J_τ that holds iff τ is disabled. To illustrate this construction, we present in Fig. 5.12 the labeled transition graph corresponding to the pending states of program ABSTRACT-UP-DOWN. As seen in the diagram, we represent the justice requirements $\neg at\_\ell_i$ and $\neg at\_m_j$ simply by the locations ℓ_i and m_j, respectively.

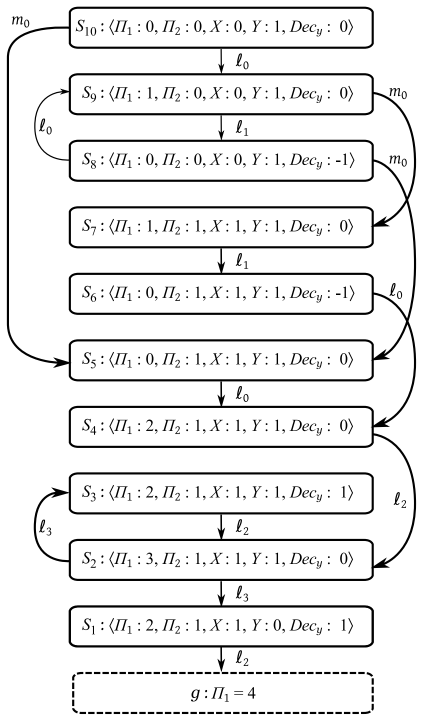

$S_{10} : \langle \Pi_1 : 0, \Pi_2 : 0, X : 0, Y : 1, Dec_y : 0\rangle$
$S_9 : \langle \Pi_1 : 1, \Pi_2 : 0, X : 0, Y : 1, Dec_y : 0\rangle$
$S_8 : \langle \Pi_1 : 0, \Pi_2 : 0, X : 0, Y : 1, Dec_y : -1\rangle$
$S_7 : \langle \Pi_1 : 1, \Pi_2 : 1, X : 1, Y : 1, Dec_y : 0\rangle$
$S_6 : \langle \Pi_1 : 0, \Pi_2 : 1, X : 1, Y : 1, Dec_y : -1\rangle$
$S_5 : \langle \Pi_1 : 0, \Pi_2 : 1, X : 1, Y : 1, Dec_y : 0\rangle$
$S_4 : \langle \Pi_1 : 2, \Pi_2 : 1, X : 1, Y : 1, Dec_y : 0\rangle$
$S_3 : \langle \Pi_1 : 2, \Pi_2 : 1, X : 1, Y : 1, Dec_y : 1\rangle$
$S_2 : \langle \Pi_1 : 3, \Pi_2 : 1, X : 1, Y : 1, Dec_y : 0\rangle$
$S_1 : \langle \Pi_1 : 2, \Pi_2 : 1, X : 1, Y : 0, Dec_y : 1\rangle$
$g : \Pi_1 = 4$
m_0
ℓ_0
m_0
ℓ_0
ℓ_1
m_0
ℓ_1
ℓ_0
ℓ_0
ℓ_2
ℓ_3
ℓ_2
ℓ_3
ℓ_2

Figure 5.12: Pending Graph for Program ABSTRACT-UP-DOWN

Having constructed the pending transition graph G we proceed to analyze it and determine the ranks associated with the abstract states. A basic process in the algorithm for rank determination is the decomposition of subgraphs into their MSCC's (maximally strongly connected components). An MSCC C is said to be *just with respect to justice requirement* J_i if C contains a state satisfying J_i. Component C is defined to be *just* if it is just with respect to all justice requirements. In Fig. 5.13, we present the algorithm for computing the ranks for a pending graph produced for a JDS.

```
Algorithm RANK-JUST-GRAPH($G, \mathcal{J}, \mathcal{C}, \mathcal{R}$)
Input:   Graph $G = (N, E)$ of pending states for $\mathcal{D}^{\mathcal{R},\alpha}$

         Justice requirements $\mathcal{J}$ of $\mathcal{D}^{\mathcal{R},\alpha}$
         Compassion requirements of $\mathcal{D}^{\mathcal{R},\alpha}$, given by $\mathcal{C} = \left\{\begin{array}{c}\langle Dec_1 > 0, Dec_1 < 0\rangle,\\ \ldots,\\ \langle Dec_\ell > 0, Dec_\ell < 0\rangle\end{array}\right\}$

         Ranking core $\mathcal{R} = \{\delta_1, \ldots, \delta_\ell\}$
Output:  $Rank$, an array $N \mapsto$ TUPLES
         $helpful$, an array $N \mapsto \mathcal{J}$
Initially: For every $n \in N$, $Rank(n) = \bot$.

JUST-RANK($G$):
1 :  Decompose $G$ into a sorted set of MSCCs $G = C_0, ..., C_k$;
2 :  Forall $i \in [0..k]$ do
       3 : Forall $n \in C_i$ do
             4 : $Rank(n) := Rank(n) * i$;
5 :  Forall $i \in [0..k]$ such that $\exists J \in \mathcal{J}, n \in C_i \,.\, n \not\models J$ do
       6 : $helpful(C_i) := J$;
7 :  Forall $i \in [1..k]$ such that $\forall J \in \mathcal{J}\ \exists n \in C_i \,.\, n \models J$ do
       [ 8 : Let $j \in [0..\ell]$ such that $\left(\begin{array}{l}\exists n \in C_i \,.\, n \models (Dec_j > 0) \wedge \\ \forall n \in C_i \,.\, n \not\models (Dec_j < 0)\end{array}\right)$
             in
             [  9 : Forall $n \in C_i$ do
                      10 : $Rank(n) := Rank(n) * \delta_j$;
               11 : Let $D$ be the subgraph obtained by removing every edge in
                          $C_i$ leading into a $(Dec_j > 0)$-node
                    in
                    12 : Call JUST-RANK($D$);                              ]]
```

Figure 5.13: Procedure RANK-JUST-GRAPH

In the table of Fig. 5.14, we present the progress of algorithm RANK-JUST-GRAPH when applied to the pending graph of program ABSTRACT-UP-DOWN, which is given in Fig. 5.12. The last column in the table lists, for each node, the justice requirement identified as helpful for that node. These entries are determined in line 6 of Algorithm RANK-JUST-GRAPH.

Node	Iteration 1	Iteration 2	Iteration 3	Final Ranking	Helpful Justice Requirement
S_{10}	8			$(8,0,0)$	m_0
S_9	7			$(7,0,0)$	m_0
S_8	7			$(7,0,0)$	m_0
S_7	6			$(6,0,0)$	ℓ_1
S_6	5			$(5,0,0)$	ℓ_0
S_5	4			$(4,0,0)$	ℓ_0
S_4	3			$(3,0,0)$	ℓ_2
S_3	2	$(2,y)$	$(2,y,1)$	$(2,y,1)$	ℓ_2
S_2	2	$(2,y)$	$(2,y,0)$	$(2,y,0)$	ℓ_3
S_1	1			$(1,0,0)$	ℓ_2
g	0			$(0,0,0)$	

Figure 5.14: Progress of Algorithm RANK-JUST-GRAPH

In the first iteration, the MSCC decomposition yields a sorted list as follows:

$$g,\ S_1,\ \{S_2,\ S_3\},\ S_4,\ S_5,\ S_6,\ S_7,\ \{S_8,\ S_9\},\ S_{10}$$

Consequently, we assign to nodes $g, S_1, \ldots, S_{10}$ the sequence of ranks:

$$0,\ 1,\ 2,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 7,\ 8$$

Next, we examine each of the components, excluding g. We find that the only just component is $\{S_2, S_3\}$. This is because each of the other components is unjust w.r.t some justice requirement, as shown by

Component	S_1	S_4	S_5	S_6	S_7	$\{S_8,\ S_9\}$	S_{10}
Violates	$\neg at\_\ell_2$	$\neg at\_\ell_2$	$\neg at\_\ell_0$	$\neg at\_\ell_0$	$\neg at\_\ell_1$	$\neg at\_m_0$	$\neg at\_m_0$

Since component $\{S_2, S_3\}$ is just, we search for a compassion requirement that it violates. Indeed, we observe that $(Dec_y > 0, Dec_y < 0)$ is violated because State S_3 has a positive value of Dec_y, but no state in this component assigns a negative value to Dec_y . We therefore augment the ranks of S_2 and S_3 by the ranking element $\delta_y\colon y$, remove the edges entering S_3, and invoke the procedure Just-Rank with a graph D whose nodes are $\{S_2, S_3\}$ and which has the single edge $(S_3 \to S_2)$. Decomposing the subgraph D, we obtain the decomposition S_2, S_3. Consequently, in the 3rd (and last) iteration, we append to nodes S_2, S_3 the ranks $0, 1$, respectively.

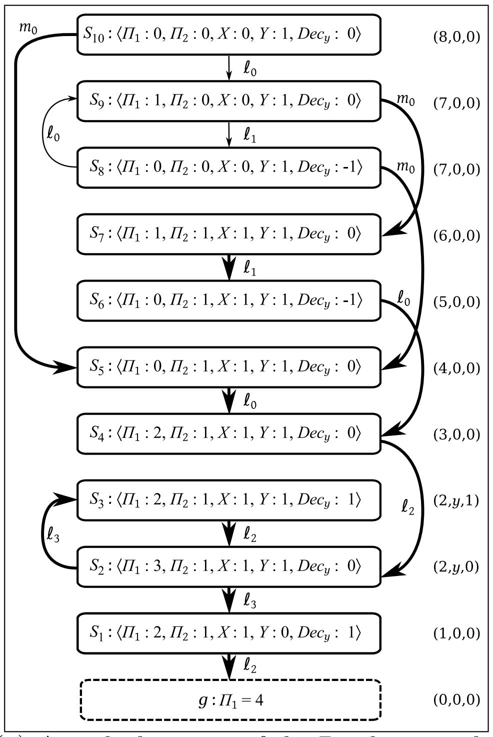
Note that, once we identify that some components are unjust, we do not process them any further. Note also that, while the sequential version of the ranking computation algorithm always terminates with a graph consisting of singleton components, the just version may leave several components intact, such as $\{S_8,\ S_9\}$.

In Fig. 5.15(a), we present a ranked version of the pending graph. Edges labeled with the helpful justice requirements are drawn in bold type.

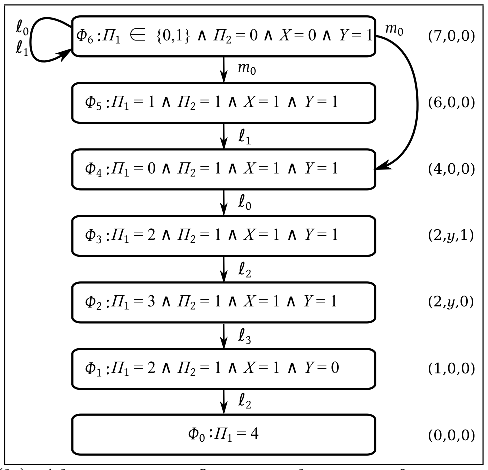
The partial correctness of RANK-JUST-GRAPH is stated by the following lemma, the proof of which is similar to that of properties **P1**, **P2**, and **P3** in Lemma 5.3.

Lemma 5.8 (Correctness of RANK-JUST-GRAPH)**.** *Let S_i and S_j be states in the pending graph, and Δ_i and Δ_j be their associated ranks produced by Algorithm* RANK-JUST-GRAPH. *Then on termination of the algorithm, the following hold:*

***C1**. If S_i is connected to S_j and both states do* not *belong in the same unjust*


$S_{10}: \langle \Pi_1: 0, \Pi_2: 0, X: 0, Y: 1, Dec_y: 0\rangle$ (8,0,0)
$S_9: \langle \Pi_1: 1, \Pi_2: 0, X: 0, Y: 1, Dec_y: 0\rangle$ (7,0,0)
$S_8: \langle \Pi_1: 0, \Pi_2: 0, X: 0, Y: 1, Dec_y: -1\rangle$ (7,0,0)
$S_7: \langle \Pi_1: 1, \Pi_2: 1, X: 1, Y: 1, Dec_y: 0\rangle$ (6,0,0)
$S_6: \langle \Pi_1: 0, \Pi_2: 1, X: 1, Y: 1, Dec_y: -1\rangle$ (5,0,0)
$S_5: \langle \Pi_1: 0, \Pi_2: 1, X: 1, Y: 1, Dec_y: 0\rangle$ (4,0,0)
$S_4: \langle \Pi_1: 2, \Pi_2: 1, X: 1, Y: 1, Dec_y: 0\rangle$ (3,0,0)
$S_3: \langle \Pi_1: 2, \Pi_2: 1, X: 1, Y: 1, Dec_y: 1\rangle$ (2,y,1)
$S_2: \langle \Pi_1: 3, \Pi_2: 1, X: 1, Y: 1, Dec_y: 0\rangle$ (2,y,0)
$S_1: \langle \Pi_1: 2, \Pi_2: 1, X: 1, Y: 0, Dec_y: 1\rangle$ (1,0,0)
$g: \Pi_1 = 4$ (0,0,0)
m_0
ℓ_0
ℓ_1
ℓ_2
ℓ_3

(a) A ranked version of the Pending graph for program ABSTRACT-UP-DOWN.


$\Phi_6: \Pi_1 \in \{0,1\} \wedge \Pi_2 = 0 \wedge X = 0 \wedge Y = 1$ (7,0,0)
$\Phi_5: \Pi_1 = 1 \wedge \Pi_2 = 1 \wedge X = 1 \wedge Y = 1$ (6,0,0)
$\Phi_4: \Pi_1 = 0 \wedge \Pi_2 = 1 \wedge X = 1 \wedge Y = 1$ (4,0,0)
$\Phi_3: \Pi_1 = 2 \wedge \Pi_2 = 1 \wedge X = 1 \wedge Y = 1$ (2,y,1)
$\Phi_2: \Pi_1 = 3 \wedge \Pi_2 = 1 \wedge X = 1 \wedge Y = 1$ (2,y,0)
$\Phi_1: \Pi_1 = 2 \wedge \Pi_2 = 1 \wedge X = 1 \wedge Y = 0$ (1,0,0)
$\Phi_0: \Pi_1 = 4$ (0,0,0)
m_0
ℓ_0
ℓ_1
ℓ_2
ℓ_3

(b) Abstract verification diagram for program ABSTRACT-UP-DOWN.

Figure 5.15: Ranking and Verification Diagram for ABSTRACT-UP-DOWN

MSCC, then there is a rank decrease $\Delta_i \succ_{S_j} \Delta_j$.

C2. $\Delta_i = \Delta_j$ *iff* S_i *and* S_j *belong in the same unjust MSCC.*

C3. $\Delta_i > \Delta_j$ *or* $\Delta_i < \Delta_j$ *iff* S_i *and* S_j *do* not *belong in the same unjust MSCC.*

In addition, properties **P4** and **P5** of Lemma 5.3 continue to hold here.

5.2.4 Forming an Abstract Verification Diagram

In the second step of the extraction process, we form the abstract verification diagram by merging each MSCC of the pending graph into a single assertion.

1. In the first step we merge abstract states that differ only in their *Dec* variables. This is done by identifying two such abstract states S_i and S_j, retaining the representative with the smaller rank, and redirecting edges previously connecting to the node with higher ranks into the node with the lower rank.

   Thus, in the graph of Fig. 5.12, we can merge S_{10} into S_8, S_6 into S_5, and S_4 into S_3.

2. Next, we construct for each *unjust* MSCC of the graph resulting from the previous step a single assertion Φ that is a disjunction of the valuations of the non-*Dec* variables of all the states contained in the MSCC.

   Thus, the assertion corresponding to the MSCC that contains the two states $S_8 : \langle \Pi_1{:}0,\ \Pi_2{:}0,\ X{:}0,\ Y{:}1,\ Dec_y{:}{-1}\rangle$ and $S_9 : \langle \Pi_1{:}1,\ \Pi_2{:}0,\ X{:}0,\ Y{:}1,\ Dec_y{:}0\rangle$ is $\Phi : \Pi_1 \in \{0,1\} \wedge \Pi_2 = 0 \wedge X = 0 \wedge Y = 1$.

3. We draw an edge labeled by J connecting node Φ_i to node Φ_j, whenever there are states $S_i \in \Phi_i$ and $S_j \in \Phi_j$ such that S_i is connected to S_j by a J-labeled edge in the ranked pending graph.

4. A node Φ is ranked by a rank Δ, which is the common rank associated with the states that belong to the MSCC.

In Fig. 5.15(b), we present the abstract verification diagram obtained from the graph of Fig. 5.15(a).

An important property of the abstract verification diagram is the following:

Lemma 5.9. *If Φ_i is connected to Φ_j in the verification diagram and $S_j \in \Phi_j$, then $\Delta_i \succ_{S_j} \Delta_j$, where Δ_i, Δ_j are the ranks associated with Φ_i, Φ_j, respectively.*

Proof. We first show that $\Delta_i \succ_{S_j} \Delta_{S_j}$. Let $S_i \in \Phi_i$ such that S_i is connected to S_j in the pending graph and let Δ_{S_i} be the rank associated with S_i. Then from property **C1** we have $\Delta_{S_i} \succ_{S_j} S_j$. If $\Delta_{S_i} = \Delta_i$ then we have $\Delta_i \succ_{S_j} \Delta_{S_j}$.

Otherwise, assume $\Delta_{S_i} \neq \Delta_i$. Then from step 1 of the construction of the abstract verification diagram we conclude that there exists a state $S \in \Phi_i$ that agrees with S_i on the values of all non-*Dec* variables, such that the rank associated with S is Δ_i. From property **P5** we conclude that S_j is a successor of S. Thus from property **C1** we have $\Delta_i \succ_{S_j} \Delta_{S_j}$.

By construction, either $\Delta_{S_j} = \Delta_j$, or $\Delta_{S_j} > \Delta_j$. Thus, by property **P4** we have $\Delta_i \succ_{S_j} \Delta_j$. □

5.2.5 Obtaining the Concrete Helpful Assertions

As the last step in the extraction of the auxiliary constructs needed by rule JUST-RESPONSE, we compute the concrete helpful assertions $\varphi_0, \ldots, \varphi_m$. As in the sequential case, these are obtained simply by concretization of the abstract assertions $\Phi_0, \ldots, \Phi_m$. In the table presented in Table 5.2, we present the auxiliary constructs extracted for program Up-Down.

i	φ_i	Δ_i	J_i
6	$at\_\ell_{0,1} \wedge at\_m_0 \wedge x = 0 \wedge y > 0$	$(7, 0, 0)$	$\neg at\_m_0$
5	$at\_\ell_1 \wedge at\_m_1 \wedge x > 0 \wedge y > 0$	$(6, 0, 0)$	$\neg at\_\ell_1$
4	$at\_\ell_0 \wedge at\_m_1 \wedge x > 0 \wedge y > 0$	$(4, 0, 0)$	$\neg at\_\ell_0$
3	$at\_\ell_2 \wedge at\_m_1 \wedge x > 0 \wedge y > 0$	$(2, y, 1)$	$\neg at\_\ell_2$
2	$at\_\ell_3 \wedge at\_m_1 \wedge x > 0 \wedge y > 0$	$(2, y, 0)$	$\neg at\_\ell_3$
1	$at\_\ell_2 \wedge at\_m_1 \wedge x > 0 \wedge y = 0$	$(1, 0, 0)$	$\neg at\_\ell_2$
0	at_ℓ_4		

Table 5.2: Extracted Auxiliary Constructs for Program Up-Down

As in the sequential case, the following property, whose proof is similar to that of Lemma 5.6, leads to the overall correctness of the construction:

Lemma 5.10. *If for concrete states s_i and s_j, $s_i \models \varphi_i$, $s_j \models \varphi_j$, and s_j is a $\mathcal{D}$-successor of s_j, then $s_i[\Delta_i] \succ s_j[\Delta_j]$.*

Chapter 6

Shape Analysis by Ranking Abstraction

In this chapter the generic ranking abstraction method is specialized to deal with heap-manipulating programs. The chapter is organized as follows: Section 6.1 describes the formal model of *finite heap systems*, which is a specialization of the fair discrete systems defined in Section 2.1. Section 6.2 deals with predicate abstraction of heap systems, and the symbolic computation of abstractions. It states and proves a small model property, and describes how to apply it to construct the finite-state FDS representing the abstraction of a finite heap system. Section 6.3 deals with proving liveness of heap systems. Sections 6.1–6.3 use a list reversal program as a running example. Section 6.4 presents a more involved example of a nested loop bubble sort, and shows its formal verification using the new method.

6.1 Finite Heap Systems

In this section the computational model of Section 2.1 is specialized to model systems that manipulate heaps. To allow the automatic computation of abstractions, we place further restrictions on the systems we study, leading to the model of *finite heap systems* (FHS), that is essentially the model of bounded discrete systems of [APR$^+$01] specialized to the case of heap programs. For brevity, we describe here a simplified two-type model; the extension for the general multi-type case is straightforward.

We allow the following data types parameterized by the positive integer H, intended to specify the heap size:

1. **bool**: boolean and finite-range scalars; With no loss of generality, we assume that all finite domain values are encoded as booleans.
2. **index**: $[0..h]$;
3. Arrays of the types **index** $\mapsto$ **bool** (**bool** array) and **index** $\mapsto$ **index** (**index** array).

We assume a signature of variables of all of these types. Constants are introduced as variables with reserved names. Thus, we admit the boolean constants FALSE and TRUE, and the **index** constant 0. In order to have all functions in the model total, we define both **bool** and **index** arrays as having the domain **index**. A well-formed program should never assign a value to $Z[0]$ for any (**bool** or **index**) array Z. On the other hand, unless stated otherwise, all quantifications are taken over the range $[1..h]$.

We often refer to an element of type **index** as a *node*. If the interpretation of an **index** variable x in a state s is ℓ, then we say that in s, x *points to the node* ℓ. An **index** *term* is an **index** variable or an expression $Z[y]$, where Z is an **index** array and y is an **index** variable.

Atomic formulae are defined as follows:

- If x is a boolean variable, B is a **index** $\mapsto$ **bool** array, and y is an **index** variable, then x and $B[y]$ are atomic formulas.

- If t_1 and t_2 are **index** terms, then $t_1 = t_2$ is an atomic formula.

- A *Transitive closure* formula (*tcf*) of the form $Z^*(x_1, x_2)$, denoting that x_2 is Z-reachable from x_1, where x_1 and x_2 are **index** variables and Z is an **index** array.

We find it convenient to include "preservation statements" for each transition, that describe the variables that are not changed by the transition. There are two types of such statements :

1. Assertions of the form $\mathit{pres}(\{v_1, \ldots, v_k\}) = \bigwedge_{i=1}^{k} v_i' = v_i$ where all v_i's are scalar (**bool** or **index**) variables;

2. Assertions of the form $\mathit{pres}_H(\{a_1, \ldots, a_k\}) = \bigwedge_{i=1}^{k} \forall h \notin H \ . \ a_i'[h] = a_i[h]$ where all a_i's are arrays and H is a (possible empty) set of **index** variables. Such an assertion denotes that all but finitely many (usually

a none or a single) entries of arrays indexed by certain nodes remain intact.

Note that preservation formulae are at most universal. We abuse notation and use the expression $\mathit{presEx}(v_1, \ldots, v_k)$ to denote the preservation of all variables, excluding the terms $v_1, \ldots, v_k$, which are either variables or array terms of the form $A[x]$.

TCF-assertions are assertions of the form $\forall \vec{y} \;.\; P(\vec{u}, \vec{y})$ where $\vec{u}$ and $\vec{y}$ are disjoint sets of variables, and for an **index** $\mapsto$ **index** array Z, $P(\vec{u}, \vec{y})$ is a positive combination of formulae of the form $\neg Z^*(u, y)$ or $\mathbb{B}(y))$ where u is an **index** variable and $\mathbb{B}$ is a boolean combination of formulae of the form $B_k[y]$ where B_k is a **bool** array.

Definition 6.1 (Restricted EA-Assertion)**.** *A* restricted EA-assertion *has the form $\exists \vec{x} \;.\; \psi(\vec{u}, \vec{x})$, where $\vec{u}$ and $\vec{x}$ are disjoint sets of* **index** *variables; ψ is a boolean combination of atomic assertions, TCF-assertions under positive polarity, and the following two types of preservation formulae, both of which appear under positive polarity:*

- *$\mathit{pres}_H(Z)$, where Z is an* **index** *array;*
- *$\mathit{pres}_H(B)$, where B is a* **bool** *array*

where in each case H denotes a (possibly empty) set of **index** *variables.*

Note that in restricted EA-assertions, universally quantified variables may occur in a TCF only as the *second* parameter. As the initial condition Θ we

```
        $x, y$   :  $[0..h]$ **init** $y = 0$
        $Nxt$    :  **array** $[0..h] \rightarrow [0..h]$
    1 :  **while** $x \neq 0$ **do**
    2 :        $(x, y, Nxt[x]) := (Nxt[x], x, y)$
         **end**
    3 :
```

Figure 6.1: Program LIST-REVERSAL

allow restricted EA-assertions, and in the transition relation ρ and fairness requirements we only allow restricted EA-assertions without TCFs. Properties of systems are restricted EA-assertions. Abstraction predicates are existentially-quantified boolean combinations of atomic formulae and TCF-assertions under positive polarity. The restriction on predicates ensures that their language is closed under negation, an assumption needed for abstraction computation.

The definition of restricted EA-assertions allows for programs that manipulate heap elements strictly via a constant set of *reference* variables, which is in accordance with many programming languages (e.g., Pascal and Java). The set of operations that are allowed is however greatly restricted. For example, arithmetic operations are not allowed. While the present definition doesn't allow inequalities, it is not hard to extend it to support them.

Example 6.1 (List Reversal) Consider program LIST-REVERSAL in Fig. 6.1, which is a simple list reversal program. The array *Nxt* describes the pointer structure. We ignore the actual data values, but they can easily be added as **bool** type variables.

Fig. 6.2 describes the FHS corresponding to program LIST-REVERSAL. The expression $\mathit{pres}(V_1)$ is an abbreviation for $\bigwedge_{v \in V_1}(v' = v)$, i.e., $\mathit{pres}(V_1)$ means that all the variables in V_1 are not changed by the transition. The expression $\mathit{pres\text{-}array}(\mathit{Nxt}, U)$ is an abbreviation for $\forall u \in \textbf{index}.u \notin U \rightarrow (\mathit{Nxt}'[u] = \mathit{Nxt}[u])$. Note that all the clauses in Fig. 6.2 are restricted assertions. The justice requirement states that as long as the program has not terminated, its execution continues.

$$
\begin{array}{ll}
V: & \left\{ \begin{array}{ll} x, y: & [0..h] \\ \mathit{Nxt}: & \textbf{array } [0..h] \rightarrow [0..h] \\ \pi: & [1..3] \end{array} \right. \\
\Theta: & \pi = 1 \;\wedge\; y = 0 \\
\rho: & \left[\begin{array}{ll} & \pi = 1 \;\wedge\; x = 0 \;\wedge\; \pi' = 3 \;\wedge\; \mathit{presEx}(\pi) \\ \vee & \pi = 1 \;\wedge\; x \neq 0 \;\wedge\; \pi' = 2 \;\wedge\; \mathit{presEx}(\pi) \\ \vee & \pi = 2 \;\wedge\; x' = \mathit{Nxt}[x] \;\wedge\; y' = x \;\wedge\; \mathit{Nxt}'[x] = y \;\wedge\; \pi' = 1 \;\wedge \\ & \mathit{presEx}(\mathit{Nxt}[x]) \\ \vee & \pi = 3 \;\wedge\; \pi' = 3 \;\wedge\; \mathit{presEx}(\pi) \end{array} \right] \\
\mathcal{J}: & \{\pi \neq 1, \pi \neq 2\} \\
\mathcal{C}: & \emptyset
\end{array}
$$

Figure 6.2: FHS for Program LIST-REVERSAL

6.1.1 Predicate Abstraction of Finite Heap Systems

The assertional language for FHS's allows for abstraction predicates that are defined by boolean combinations of atomic formulae. This allows one to abstract the system according to a user-specified set of graph-theoretic properties over the heap and **index** variables. As an example of this approach we show how to apply abstraction to our running example.

Example 6.2 (List Reversal Abstraction) Consider program List-Reversal of Example 6.1. One of the safety properties one wishes to prove is that no elements are removed from the list, i.e., that every element initially reachable from x is reachable from y upon termination. This property can be expressed by:

$$\forall t.(\pi = 1 \wedge t \neq 0 \wedge Nxt^*(x,t)) \rightarrow \Box(\pi = 3 \rightarrow Nxt^*(y,t)) \tag{6.1}$$

We augment the program with a generic variable t, which is a variable whose initial value is unconstrained and remains fixed henceforth. Then validity of Formula (6.1) reduces to the validity of:

$$(\pi = 1 \wedge t \neq 0 \wedge Nxt^*(x,t)) \rightarrow \Box(\pi = 3 \rightarrow Nxt^*(y,t)) \tag{6.2}$$

Following the above discussion, to prove the safety property of Formula (6.2), the set $\mathcal{P}$ consists of $x = 0$, $t = 0$, $Nxt^*(x,t)$, and $Nxt^*(y,t)$, which we denote as the abstract variables x_0, t_0, r_xt, and r_yt respectively.

The abstract program is Abstract-List-Reversal, shown in Fig. 6.3, and the abstract property corresponding to Formula (6.2) is:

$$\psi^\alpha : (\Pi = 1 \wedge \neg t\_0 \wedge r\_xt) \rightarrow \Box(\Pi = 3 \rightarrow r\_yt)$$

where Π is the program counter of the abstract program.

```
        $x\_0, t\_0, r\_xt, r\_yt$ : bool
        init $x\_0 = t\_0 = \text{FALSE}, r\_xt = \text{TRUE}, r\_yt = t\_0$
  1 :   while $\neg x\_0$ do
           $(r\_xt, r\_yt) :=$  case
                                  $\neg r\_xt \wedge \neg r\_yt$  : $(\text{FALSE}, \text{FALSE})$
                                  $\neg r\_xt \wedge r\_yt$   : $\left\{ (\text{FALSE}, \text{TRUE}), (\text{TRUE}, \text{TRUE}) \right\}$
  2 :                             otherwise   : $\left\{ (\text{FALSE}, \text{TRUE}), (\text{TRUE}, \text{FALSE}), (\text{TRUE}, \text{TRUE}) \right\}$
                              esac
           $x\_0 :=$ if $r\_xt$ then FALSE else $\{\text{FALSE}, \text{TRUE}\}$
        end
  3 :
```

Figure 6.3: Program Abstract-List-Reversal

It is now left to check whether $S^\alpha \models \psi^\alpha$, which can be done, e.g., using a model checker. Here, the initial abstraction is precise enough, and program Abstract-List-Reversal satisfies ψ^α. In Section 6.4 we present a more challenging example requiring several iterations of refinement. ┛

6.2 Symbolic Computation of Abstractions

This section describes a methodology for symbolically computing an abstraction of an FHS. The methodology is based on a small model property that establishes that satisfiability of a restricted assertion can be checked on a small instantiation of a system.

Let $\mathcal{V}$ be a *vocabulary* of typed variables, whose types are taken from the restricted type system allowed in an FHS, as well as the *primed* version of said variables. Furthermore, assume that there is a single unprimed **index** array in $\mathcal{V}$ as well as a single primed one. A *model* M for $\mathcal{V}$ consists of the following elements:

- A positive integer $h > 0$.
- For each boolean variable $b \in \mathcal{V}$, a boolean value $M[b] \in \{\textsc{False}, \textsc{True}\}$. It is required that $M[\textsc{False}] = \textsc{False}$ and $M[\textsc{True}] = \textsc{True}$.
- For each **index** variable $x \in \mathcal{V}$, a natural value $M[x] \in [0..h]$. It is required that $M[0] = 0$.
- For each boolean array $B \in \mathcal{V}$, a boolean function $M[B] : [0..h] \mapsto \{\textsc{False}, \textsc{True}\}$.
- For each **index** array $Z \in \mathcal{V}$, a function $M[Z] : [0..h] \mapsto [0..h]$.

We define the *size* of model M to be $h + 1$. Let $\varphi(\vec{u})$ be a restricted EA-assertion which we fix for this section, and assume, for the time being, that φ has no existentially quantified variables. $\vec{u}$ is the set of free variables appearing in φ. We require that, for the unprimed and primed **index** arrays Z and Z', if a term of the form $Z'[u]$ occurs in φ where u is a free or existentially quantified variable in φ, then φ also contains the preservation formula associated with Z. Note that this requirement is satisfied by any reasonable φ — assertions that contain primed variables occur only in proofs for abstraction computation (rather than in properties of systems), and are generated automatically by the proof system. In such cases, the assertion generated includes also the transition relation, which includes all preservation formulae. We include this requirement explicitly since the proof of the small model theorem depends on it. The requirement is formalized by the following definition:

Definition 6.2 (Uniformity)**.** *Let φ be a restricted A-assertion. A model M is called a Z-*uniform model *(*uniform model *for short), if for every $k \in [0..h]$ and every* **index** *arrays Z and Z' such that $M[Z](k) = k_1$ and $M[Z'](k) = k_2$ for $k_1 \neq k_2$, then k and at least one of k_1 or k_2 are M-interpretations of a free term of φ. A restricted A-assertion is called a Z-uniform assertion (uniform assertion for short) if all its models are Z-uniform.*

Throughout the rest of this chapter, we assume that all assertions are uniform.

For a given $\vec{u}$-model M, we can evaluate the formula φ over the model M. Model M is called a *satisfying model* for φ if $M \models \varphi$. An **index** term $t \in \{x, Z[x]\}$ is called a *free term* in φ.

Definition 6.3 (History Closure)**.** *A set of* **index** *terms $\mathcal{T}$ is said to be* history closed *if it has the following properties:*

- *For every array $Z \in \mathcal{V}$ and* **index** *variable $u \in \mathcal{V}$, if $Z[u] \in \mathcal{T}$ then $u \in \mathcal{T}$;*
- *For a primed* **index** *array $Z' \in \mathcal{V}$ and* **index** *variable u, if $Z'[u] \in \mathcal{T}$ then $Z[u] \in \mathcal{T}$.*

The following is a direct consequence of history closure:

Lemma 6.4 (Strong Uniformity). *For any uniform assertion φ, model M of φ, nodes k, k_1, and k_2, and a history-closed term set $\mathcal{T}$ that contains all free terms in φ, if $M[Nxt](k) = k_1 \neq k_2 = M[Nxt'](k)$, then all of k, k_1, and k_2 are pointed to by terms in $\mathcal{T}$.*

For an assertion φ, let $\mathcal{T}_\varphi$ denote the minimal set that satisfies the following:

- $\mathcal{T}_\varphi$ contains the term 0 and all free terms in φ;
- $\mathcal{T}_\varphi$ is history-closed.

Let M be a model of φ and $\mathcal{N}$ be the set of $[0..h]$ values assigned by M to terms in $\mathcal{T}_\varphi$. Assume that $\mathcal{N} = \{n_0, \ldots, n_\alpha\}$ where $0 = n_0 < \cdots < n_\alpha$. Obviously, $\alpha \leq |\mathcal{T}_\varphi|$. Let $i \in [1..\alpha]$, and Z be a **index** $\mapsto$ **index** array.

Definition 6.5 (Representative Node). *If for some $\ell > 0$, $M[Z]^\ell(n_i) \in \mathcal{N}$, then we say that n_i* has a Z-representative (in M)*. Let ℓ be the minimal such that $M[Z]^\ell(n_i) \in \mathcal{N}$. We then say that $M[Z]^{\ell-1}(n_i)$* is the Z-representative of n_i (in M) *and denote it by $r^Z_M(n_i)$, or simply $r^Z(n_i)$ when M is apparent from the context.*

Note that if $r^Z_M(n_i)$ is defined, then it is either n_i itself or some node not in $\mathcal{N}$.

Definition 6.6 (Escape Node). *If for some ℓ, for every $j \geq \ell$, $M[Z]^\ell(n_i) \notin \mathcal{N}$, then we say that n_i* is Z-escaping (in M)*. Let ℓ be the minimal such that*

$M[Z]^j(n_i) \notin \mathcal{N}$ for every $j \geq \ell$. We then say that $M[Z]^\ell(n_i)$ is the *Z*-escape of *n_i* (in *M*) *and denote it by $e^Z_M(n_i)$, or simply $e^Z(n_i)$ when M is apparent from the context.*

Let $\mathcal{R}_\varphi$ be the set of pairs (Z, t) such that $Z^*(t, y)$ occurs in φ and y is universally quantified in φ. If the set

$$\{e^Z_M(t) : (Z, t) \in \mathcal{R}_\varphi \text{ and } t \text{ is } Z\text{-escaping}\}$$

is empty, then define $\mathcal{S} = \{n_{\alpha+1}\}$ where $n_{\alpha+1}$ is the M-minimal node not in $\mathcal{N}$. Else, denote the above set by $\mathcal{S}$ and assume that $\mathcal{S} = \{n_{\alpha+1}, \ldots, n_{\alpha+\beta}\}$. Note that $|\mathcal{S}| \leq |\mathcal{R}_\varphi|$. For every $i \in [0..\alpha + \beta]$, define $\Gamma(n_i) = i$. Define $f(\varphi) = |\mathcal{T}_\varphi| + \max(|\mathcal{R}_\varphi|, 1)$. We construct a model whose size is at most $|\mathcal{N}| + |\mathcal{S}| \leq f(\varphi)$.

Definition 6.7 (Model Reduction)**.** *The reduction of model M of φ is the model $\overline{M}$ defined as follows:*

- *$\overline{M}[h] = \alpha + \beta$;*
- *For each* **bool** *variable b, $\overline{M}[b] = M[b]$;*
- *For each (type) variable $u \in \mathcal{T}_\varphi$, $\overline{M}[u] = \Gamma(M[u])$;*
- *For each Z: type $\mapsto$ type and $i \in [0..\alpha + \beta]$, we define $\overline{M}[Z](i)$ as follows, where each case assumes that none of the previous ones are*

true.

$$\overline{M}[Z](i) = \begin{cases} 0 & i = 0, \\ \Gamma(r_M^Z(n_i)) & i \in [1..\alpha] \textit{ and } n_i \textit{ has a } Z\textit{-representative} \\ \Gamma(e_M^Z(n_i)) & i \in [1..\alpha] \textit{ and for some } t \in \mathcal{T}_\varphi,\ (Z,t) \in \mathcal{R}_\varphi \\ & \textit{and } M[Z]^*(t,n_i) \textit{ or } M[Z]^*(n_i,t)) \\ \alpha + 1 & i \in [1..\alpha],\ \textit{ and for no } t,\ (Z,t) \in \mathcal{R}_\varphi \\ \Gamma(n_i) & i \in [\alpha+1..\beta] \textit{ and for every } \ell \geq 1, \\ & M[Z]^\ell(n_i) \notin \mathcal{S} \\ \Gamma(M[Z]^\ell(n_i)) & i \in [\alpha+1..\beta] \textit{ and } \ell \geq 1 \textit{ is the minimal} \\ & \textit{such that } M[Z]^\ell(n_i) \in \mathcal{S} \end{cases}$$

- *For every* **bool** *array B and $j \in [1..(\alpha+\beta)]$, $\overline{M}[B](j) = M[B](n_j)$.*

The following theorem states that if φ has a satisfying model, then it has a small satisfying model.

Theorem 6.8. *Let $\varphi(\vec{u})$ be a restricted EA-assertion without existentially quantified variables. Then φ has a satisfying model iff it has a satisfying model of size not exceeding $f(\varphi) = |\mathcal{T}_\varphi| + \max(|\mathcal{R}_\varphi|, 1)$.*

Before approaching the proof, we first prove:

Observation 6.9. *The following are properties of the construction:*

P0. *Given two nodes $n_i, n_j \in \mathcal{N}$ that are Z-reachable from one another (in whichever direction), then if one is Z-escaping then so is the other, and they both have the same Z-escapes.*

P1. *For every i and j that are* both *in* $[0..\alpha]$ *or* both *in* $[(\alpha+1)..\beta]$, *for every* **index** $\mapsto$ **index** *array* Z,

$$M[Z](n_i) = n_j \implies \overline{M}[Z](i) = j \quad \textit{and} \quad \overline{M}[Z](i) = j \implies M[Z]^*(n_i, n_j)$$

P2. *For every i and j that are* both *in* $[0..\alpha]$ *or* both *in* $[(\alpha+1)..\beta]$, *for every* **index** $\mapsto$ **index** *array* Z,

$$M[Z]^*(n_i, n_j) \iff \overline{M}[Z]^*(i, j)$$

P3. *For every $i \in [1..\alpha]$ and $j \in [(\alpha+1)..\beta]$, for every pair $(Z, t) \in R_\varphi$, if $M[Z]^*(n_i, t)$ or $M[Z]^*(t, n_i)$, then*

$$\overline{M}[Z]^*(i, j) \iff M[Z]^*(n_i, n_j)$$

Proof. Properties **P0** and **P1** follow immediately from the construction.

As for **P2**, in one direction assume that $(M[Z])^*(n_i, n_j)$. Thus, there exists a Z-chain σ from n_i to n_j in M. Project the path onto its $\mathcal{N}$-nodes if $i, j \leq \alpha$, or onto its $\mathcal{S}$-nodes if $i, j > \alpha$. Let $n_i = v_1, \ldots, v_k = n_j$ be the remaining nodes. From the definition of $\overline{M}[Z]$ it follows that for every $\ell = 1, \ldots, k-1$, $\overline{M}[Z](\gamma(v_i)) = \gamma(v_{i+1})$, and, therefore, $(\overline{M}[Z])^*(\gamma(v_0), \gamma(v_k))$. Since $\gamma(v_0) = i$ and $\gamma(v_k) = j$, the claim follows. In the other the claim follows directly from the construction and property **P1**.

As for **P3**, in one direction, assume that $\overline{M}[Z]^*(i,j)$. From the construction, there exists an Z-chain $n_{i_1},\ldots,n_{i_k}$ in $\overline{M}$ such that $i_1 = i$, $i_k = j$. Moreover, for some $\ell > 1$, for every $k \leq \ell$, $n_{i_k} \leq \alpha$, and for every $k \geq \ell$, $n_{i_k} > \alpha$. From **P1** it follows that it suffices to show that $M[Z](n_{i_\ell}) = n_{i_{\ell+1}}$. From the construction is follows that $n_{i_{\ell+1}} = e^Z_M(n_{i_\ell}) = e^Z_M(t)$. The claim now follows trivially. In the other direction, assume that $M[Z]^*(n_i,n_j)$. Thus, there exists an Z-chain $n_i = u_1,\ldots,u_k = n_j$ in M such that for some $\ell > 1$, $u_\ell \in \mathcal{N}$ and $u_{\ell+1} \in \mathcal{S}$. Since $M[Z]^*(n_i,u_\ell)$, and u_ℓ is reachable from some t such that $(Z,t) \in \mathcal{R}_\varphi$, it follows from **P0** that $e^Z_M(n_i) = e^Z_M(u_\ell) = u_{\ell+1}$, and therefore $\overline{M}[Z](\Gamma(u_\ell)) = \Gamma(u_{\ell+1})$. The claim now follows from **P1** since $M[Z]^*(n_i,u_\ell)$ and $M[Z]^*(u_{\ell+1},n_j)$. □

We can now prove the small model property, i.e., that $M \models \varphi$ iff $\overline{M} \models \varphi$.

Proof. To prove the theorem, it suffices to show that if $M \models \varphi$ and M's size is greater than $|N| + |\mathcal{S}|$, then $\overline{M} \models \varphi$. The proof is by induction on the structure of φ. Recall that φ is a positive boolean combination of atomic assertions and their negation, preservation assertions, and TCF-assertions.

Atomic Assertions We first deal with atomic assertions. For an atomic assertions ψ, we have to show that $M \models \psi$ iff $\overline{M} \models \psi$. There are three base cases:

ψ **is a bool variable** b **or bool term** $B[u]$**.** The claim follows trivially from the construction of $\overline{M}$;

ψ **is of the form** $t_1 = t_2$ **for free** *type* **terms** t_1 **and** t_2**.** Since $t_1, t_2 \in \mathcal{T}_\varphi$, it follows from the construction that $M \models t_1 = t_2$ iff $\overline{M} \models t_1 = t_2$.

ψ **is a TCF formula** $Z^*(t_1, t_2)$ **where** $t_1, t_2 \in \mathcal{T}_\varphi$**.** The claim follows immediately from property **P2**.

The inductive step follows immediately.

Preservation Assertions Next we deal with preservation assertions. Since such an assertion, say ψ, can only appear in positive polarity, we have to show that if $M \models \psi$ then $\overline{M} \models \psi$. We distinguish between preservation of **index** $\mapsto$ **index** arrays and preservation of **index** $\mapsto$ **bool** arrays. In the first case, ψ is of the form

$$\psi\colon\ \forall y. Z'[y] = Z[y] \ \lor \ \bigvee_{i=j}^{n} (y = y_i)$$

where $y_1, \ldots, y_n$ are **index** variables in $\mathcal{T}_\varphi$ and Z is an **index** $\mapsto$ **index** array. Denote by Y the set $\{y_1, \ldots, y_n\}$. Denote by $M[Y]$ and by $\Gamma(M[Y])$ the sets $\{M[y_1], \ldots, M[y_n]\}$ and $\{\Gamma(M[y_1]), \ldots, \Gamma(M[y_k])\}$, respectively. Thus ψ can be rewritten as

$$\psi\colon\ \forall y. Z'[y] = Z[y] \ \lor \ y \in Y$$

Assume that $M \models \psi$. We have to show that for every $i \in [0..\alpha + \beta]$, $\overline{M}[Z]'(i) = \overline{M}[Z](i)$ or $i \in \Gamma(M[Y])$. Assume, by way of contradiction, that for some $i \in [0..\alpha + \beta]$, $\overline{M}[Z'](i) \neq \overline{M}[Z](i)$ and $i \notin \Gamma(M[Y])$. Consider the Z- and Z'-chains in M originating with n_i, given by $n_i = u_0, u_1, \ldots$ and $n_i = v_0, v_1, \ldots$ respectively. Since we assumed that $i \notin \Gamma(M[Y])$, and $M \models \psi$, it follows that $M[Z]'(u_0) = M[Z](v_0)$. Hence $v_1 = u_1$. Proceeding like this, we obtain that either

1. For all $j \geq 0$, $u_j = v_j$ (and then $\overline{M}[Z'](i) = \overline{M}[Z](i)$), or

2. For some $\ell \geq 1$, $u_\ell = v_\ell \in M[Y]$, and for all $j = 0, \ldots, \ell - 1$, $u_j = v_j \notin M[Y]$.

Since $\mathcal{N}$-nodes are unreachable from $\mathcal{S}$-nodes, and since $M[Y] \subseteq \mathcal{N}$, we conclude that $n_i \notin \mathcal{S}$, or equivalently $i \leq \alpha$. Since $u_\ell = v_\ell \in M[Y]$, then we obtain that n_i has identical Z- and Z'-representatives, and thus by construction we have $\overline{M}[Z](i) = \overline{M}[Z'](i)$, which contradicts our assumption that $\overline{M}[Z'](i) \neq \overline{M}[Z](i)$.

Assume now that ψ is a preservation formula of a **bool** array B. Following the notation of the previous part, assume p is of the form $\forall y \;.\; B'[y] = B[y] \vee y \in Y$ where Y is a set of **index** variables in $\mathcal{T}_\varphi$. Since, by construction $\overline{M}[B'](i) = M[B'](i)$ and $\overline{M}[B](i) = M[B](i)$, for any $i \in [0..(\alpha + \beta)]$, the claim follows trivially.

TCF-assertions For a TCF-assertion $\psi : \forall \vec{y} \;.\; P$, since P is a positive boolean combination of formulae of the form $\neg Z^*(u, y)$ and $\mathbb{B}(y)$, it suffices to show that if M satisfies a clause ϕ which has either of the two forms, then $\overline{M}$ satisfies ϕ.

Assume, therefore that $M \models \phi$. To interpret the formula P over $\overline{M}$, we consider an arbitrary assignment $\overline{\eta}$ to the quantified variables $\vec{y}$ that assigns to each variable y in $\vec{y}$ a value $\overline{\eta}[y] \in [0..\alpha + \beta]$. For convenience, as η we choose the assignment $\eta(y) = \Gamma^{-1}(\overline{\eta}(y))$. Denote by M_η the joint interpretation (M, η) which interprets all quantified variables according to η and all other terms according to M. Similarly, denote by $\overline{M}_\eta$ the joint interpretation $(\overline{M}, \overline{\eta})$. It remains to prove that $\overline{M}_\eta \models \phi$ under the assumption that $M_\eta \models \phi$. Assume that $M_\eta \models \phi$ and let i_y be $\overline{M}_\eta[y]$.

Assume first that ϕ is the formula $\mathbb{B}(y)$. Let $B[y]$ be a clause in $\mathbb{B}$. Then $M_\eta[y] = n_{i_y}$ by definition. From the construction of $\overline{M}$ we have $\overline{M}_\eta[B[y]] = \overline{M}[B](i_y) = M_\eta[B[y]] = M[B](n_{i_y})$.

Assume that ϕ is the formula $\neg Z^*(u, y)$. Since u is a free term in ψ, it follows that $M[u] \leq \alpha$. It then follows from Properties **P2** and **P3** that $\overline{M}_\eta \not\models Z^*(u, y)$. $\square$

Corollary 6.10 (Small Model Property)**.** *Let $\varphi : \exists \vec{x} \,.\, \psi(\vec{u}, \vec{x})$ be a restricted EA-assertion, where $\vec{x}$ and $\vec{u}$ are disjoint sets of* **index** *variables. If $M \models \varphi$ then φ is satisfiable by a model of size at most $|\mathcal{T}_\psi| + 1$.*

Proof. Let M be a model of φ, and (M, η) be the augmentation of M with an interpretation η of the existential variables $\vec{x}$. Then $M \models \varphi$ iff there exists some η such that $(M, \eta) \models \psi$. Since ψ is a restricted EA-assertion without existentially-quantified variables, the claim follows from Theorem 6.8. $\square$

For example, consider a formula φ and a set $\mathcal{T} = \{0, v_1, v_2, v_3\}$ that includes $\mathcal{T}_\varphi$. Let M be a uniform model with $H = 7$; $M[v_1] = 1; M[v_2] = 3, M[v_3] = 5, M[Nxt] = [6, 6, 7, 5, 5, 5, 7]$. Then, according to the construction, $\overline{H} = 4; \overline{M}[0] = 0; \overline{M}[v_1] = 1; \overline{M}[v_2] = 2; \overline{M}[v_3] = 3; \overline{M}[Nxt] = [3, 4, 3, 4]$.

Given a restricted EA-assertion φ and a positive integer h_0, we define the h_0-*bounded* version of φ, denoted $\lfloor\varphi\rfloor_{h_0}$, to be the conjunction $\varphi \wedge \forall y \,.\, y \leq h_0$. Corollary 6.10 can be interpreted as stating that φ is satisfiable iff $\lfloor\varphi\rfloor_{|\mathcal{T}|}$ is satisfiable.

Next, we would like to extend the small model theory to the computation of abstractions. Consider first the case of a restricted A-assertion φ which only refers to unprimed variables. As explained in Section 2.4, the abstraction of φ is given by $\alpha(\varphi) = \exists V \,.\, (V_A = \mathcal{E}_A(V)) \;\wedge\; \varphi(V)$. Assume that the set of (finitely many combinations of) values of the abstract system variables V_A is $\{U_1, \ldots, U_k\}$. Let $sat(\varphi)$ be the subset of indices $i \in [1..k]$, such that $U_i = \mathcal{E}_\alpha(V) \;\wedge\; \varphi(V)$ is satisfiable. Then, it is obvious that the abstraction $\alpha(\varphi)$ can be expanded into

$$\alpha(\varphi)(V_A) = \bigvee_{i \in sat(\varphi)} (V_A = U_i) \tag{6.3}$$

Consider the assertion $\psi_0 : U_i = \mathcal{E}_\alpha(V) \;\wedge\; \varphi(V)$. Let $h_0 = |\mathcal{T}_{\psi_0}|$. Our reinterpretation of Corollary 6.10 states that ψ_0 is satisfiable iff $\lfloor\psi_0\rfloor_{h_0}$ is satisfiable. Therefore, $sat(\lfloor\varphi\rfloor_{h_0}) = sat(\varphi)$. Thus, $\alpha(\varphi)(V_A) \leftrightarrow \alpha(\lfloor\varphi\rfloor_{h_0})(V_A)$. This can be extended to abstraction of assertions that refer to primed variables, such as a transition relation ρ. Recall that the abstraction of such an assertion involves a double application of the abstraction mapping, an unprimed version and a primed version. Consider such an assertion $\rho(V, V')$, as well as $\psi_1 : (U_i = \mathcal{E}_A(V)) \wedge (U_j = \mathcal{E}_A(V')) \wedge \rho(V, V')$. Let $h_1 = |\mathcal{T}_{\psi_1}|$. By the same reasoning, we have $\alpha(\rho)(V_A, V'_A) \leftrightarrow \alpha(\lfloor\rho\rfloor_{h_1}(V_A, V'_A))$. This reasoning is summarized by the following theorem:

Theorem 6.11. *Let φ be an assertion that refers to unprimed variables and possible primed variables and $\alpha : V_A = \mathcal{E}_A(V)$ be an abstraction mapping. Let ψ be the formula $(U_i = \mathcal{E}_A(V)) \wedge \varphi(V)$ in the case that φ refers only to unprimed variables, or $(U_i = \mathcal{E}_A(V)) \wedge (U_j = \mathcal{E}_A(V')) \wedge \varphi(V, V')$ if φ also refers to primed variables. Let $\mathcal{T}$ be a set of free terms that contains $\mathcal{T}_\psi$, and $h_0 = |\mathcal{T}|$. Then*

$$\alpha(\varphi) \quad \leftrightarrow \quad \alpha(\lfloor\varphi\rfloor_{h_0})$$

Next we generalize these results to entire systems. For an FHS $S = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ and positive integer h_0, we define the h_0-bounded version of S, denoted $\lfloor S\rfloor_{h_0}$, as $\langle V \cup \{H\}, \lfloor\rho\rfloor_{h_0}, \lfloor\mathcal{J}\rfloor_{h_0}, \lfloor\mathcal{C}\rfloor_{h_0}\rangle$, where $\lfloor\mathcal{J}\rfloor_{h_0} = \{\lfloor J\rfloor_{h_0} \mid J \in \mathcal{J}\}$ and $\lfloor\mathcal{C}\rfloor_{h_0} = \{(\lfloor p\rfloor_{h_0}, \lfloor q\rfloor_{h_0}) \mid (p, q) \in \mathcal{C}\}$. Let h_0 be the maximum size of the sets of free terms for all the abstraction formulas necessary for computing the abstraction of all the components of S. Then we have the following theorem:

Theorem 6.12. *Let S be an* FHS*, α be an abstraction mapping, and h_0 the maximal size of the relevant sets of free terms as described above. Then the abstract system S^α is equivalent to the abstract system $\lfloor S\rfloor^\alpha_{h_0}$.*

We use TLV [PS96] to compute the abstract system $\lfloor S \rfloor_{h_0}^{\alpha}$. Note that h_0 is linear in the number of system variables. The only manual step in the process is the choice of the state predicates. The exact manner by which predicates themselves are derived (e.g., by user input or as part of a refinement loop) is orthogonal to the method presented here.

Example 6.3 Consider again program LIST-REVERSAL of Example 6.1. In Example 6.2 (of Section 2.4) we described its abstraction, which was manually derived. In order to obtain an automatic abstraction for the system whose set of free terms is $\mathcal{T} = \{0, H, x, y, t, x', y', Nxt'[x]\}$, we bounded the system by $h_0 = 8$.

We compute the abstraction in TLV by initially preparing an input file describing the concrete truncated system. We then use TLV's capabilities for dynamically constructing and updating a model to construct the abstract system by separately computing the abstraction of the concrete initial condition, transition relation, and fairness requirements.

Having computed the abstract system, we check the safety property ψ^{α}, which, of course, holds. All code is in *http://www.cs.nyu.edu/acsys/shape-analysis.*

6.3 Liveness

A finitary state abstraction such as predicate abstraction often does not suffice to verify liveness properties and needs to be complemented with some form of *transition abstraction*. To this end we use the modular ranking augmentation of Section 3.1. Let $(\mathcal{D}, \succ)$ be a partially ordered well founded domain, and assume a *ranking function* $\delta \colon \Sigma \to \mathcal{D}$. Define a function *decrease* by:

$$decrease = \begin{cases} 1, & \text{if } \delta \succ \delta' \\ 0, & \text{if } \delta = \delta' \\ -1, & \text{otherwise} \end{cases}$$

Following the ranking abstraction method, the transitions of a system are abstracted into changes in the value of δ by synchronously composing the system with a *progress monitor* [KP00] as shown in Fig. 6.4. The compassion

```
dec : {-1, 0, 1}
compassion ($dec = 1, dec = -1$)
[ loop forever do
    1 :  $dec := decrease$ ]
```

Figure 6.4: Progress Monitor $M(\delta)$ for a Ranking δ

requirement corresponds to the well-foundedness of $(\mathcal{D}, \succ)$: the ranking cannot decrease infinitely many times without increasing infinitely many times. To incorporate this in a state abstraction α, we add the defining equation

$(Dec = dec)$ to α.

Example 6.4 (List Reversal Termination) Consider program List-Reversal and the termination property $\diamondsuit(\pi = 3)$. The loop condition $x \neq nil$ in line 1 implies that the set of nodes starting with x is a measure of progress. This suggests the ranking $\delta = \{i \mid Nxt^*(x, i)\}$ over the well founded domain $(2^{\mathbb{N}}, \supset)$. That is, the rank of a state is the set of all nodes which are currently reachable from x. As the computation progresses, this set loses more and more of its members until it becomes empty. Using a sufficiently precise state abstraction, one can model check that the abstract property $\diamondsuit(\Pi = 3)$ indeed holds over the program. ⌟

Computing the Augmented Abstraction

We aim to apply symbolic abstraction computation of Section 6.2 to systems augmented with progress monitors. However, since progress monitors are not limited to restricted A-assertions, such systems are not necessarily FHS's. Thus, for any ranking function δ, one must show that Theorem 6.12 is applicable to such an extended form of FHS's. Since all assertions in the definition of an augmented system, with the exception of the transition relation, are restricted A-assertions, we need only consider the augmented transition relation $\rho \ \wedge \ \rho_\delta$, where ρ is the unaugmented transition relation and ρ_δ is defined as $dec' = decrease$. Consider the formula $\rho \ \wedge \ \rho_\delta$. Let $\mathcal{T}$ be a history-closed term set that contains $\mathcal{T}_{\rho \ \wedge \ \rho_\delta}$, as well as all free terms in $\mathcal{E}_\alpha(V)$ and

$\mathcal{E}_\alpha(V')$. Then Theorem 6.12 holds if it is the case that

$$sat(\lfloor \rho \;\wedge\; \rho_\delta \rfloor_{|\mathcal{T}|}) = sat(\rho \;\wedge\; \rho_\delta) \tag{6.4}$$

Since proving Formula (6.4) for an arbitrary ranking is potentially a significant manual effort, we specifically consider the following commonly used ranking functions over the well founded domain $(2^{\mathbb{N}}, \supset)$:

$$\delta_1(x) = \{i \mid Nxt^*(x, i)\} \tag{6.5}$$

$$\delta_2(x, y) = \{i \mid Nxt^*(x, i) \;\wedge\; Nxt^*(i, y)\} \tag{6.6}$$

In the above, x and y are **index** variables and Nxt is an **index** array. Ranking δ_1 is used to measure the progress of a forward moving pointer x, while ranking δ_2 is used to measure the progress of pointers x and y toward each other. Throughout the rest of this section we assume that the variables x and y appearing in δ_1 or δ_2 are free terms in the unaugmented transition relation.

The following theorem establishes the soundness of our method for proving liveness for the two ranking functions we consider.

Theorem 6.13. *Let S be an unaugmented* FHS *with transition relation ρ, δ_i be a ranking with $i \in \{1, 2\}$, M be a uniform model satisfying $\rho \;\wedge\; \rho_\delta$, and $\mathcal{T}$ be a history-closed term set that contains $\mathcal{T}_{\rho \,\wedge\, \rho_\delta}$ as well as the free terms in $\mathcal{E}_\alpha(V)$ and $\mathcal{E}_\alpha(V')$. Let $\overline{M}$ be the appropriate reduced model of size $h_0 = |\mathcal{T}|$.*

Then $\overline{M} \models \rho_{\delta_i}$ only if $M \models \rho_{\delta_i}$.

We first make the following observation:

Observation 6.14. *For any $j \in [(\alpha+1)..(\alpha+\beta)]$ and unprimed and primed* **index** *arrays Nxt and Nxt':*

***P4**. $M[Nxt](n_j) = M[Nxt'](n_j)$, and*

***P5**. $\overline{M}[Nxt](j) = \overline{M}[Nxt'](j)$.*

Proof. To prove property **P4** assume, by contradiction, that $M[Nxt](n_j) \neq M[Nxt'](n_j)$. Then, from uniformity, we conclude that there exists a term $t \in \mathcal{T}$ such that $n_j = M[t]$, and hence $n_j \in \mathcal{N}$. This contradicts the fact that $\mathcal{N} \cap \mathcal{S} = \emptyset$.

As for property **P5**, if, by way of contradiction, $\overline{M}[Nxt](j) = i \neq \overline{M}[Nxt'](j) = i'$, then $\ell \in [(\alpha+1)..(\alpha+\beta)]$. From the construction of $\overline{M}$, we conclude that there exist $M[Nxt]$– and $M[Nxt']$-chains $n_j = u_0, u_1, \ldots, u_\ell = n_i$ and $n_j = v_0, v_1, \ldots, v_{\ell'} = n_{i'}$, respectively, that diverge at some index $k > 0$; I.e., k is the minimal index such that $M[Nxt](u_k) \neq M[Nxt'](v_k)$. From uniformity, we conclude, as above, that $u_k = v_k \in \mathcal{N}$, and apply the same reasoning as in the proof of property **P4** to lead to a contradiction. □

Proof. We prove the claim for a ranking δ_1 of the form $\delta_1(x) = \{i \mid Nxt^*(x,i)\}$ specified in equation (6.5). The case of δ_2 is justified by similar arguments.

The evaluation of δ_1 in M, written $M[\delta_1]$, is the set $\{i \mid M[Nxt^*](M[x], i)\}$, i.e, the set of all M-nodes which are reachable from $M[x]$ by $M[Nxt]$-links. The evaluation of δ_1 in $\overline{M}$ and of δ_1' in M and $\overline{M}$ are defined similarly.

First note the following property of terms in $\mathcal{T}$: It follows directly from Property **P3** of Theorem 6.8 that, for any term t in $\mathcal{T}$ and $\delta \in \{\delta_1, \delta_1'\}$, $M[t] \in M[\delta]$ iff $\overline{M}[t] \in \overline{M}[\delta]$.

To prove the claim it is enough to show that both properties $\delta_1 \supset \delta_1'$ and $\delta_1 = \delta_1'$ are satisfied by M iff they are satisfied by $\overline{M}$. First assume $M \models \delta_1 \supset \delta_1'$. It is easy to show that $\delta_1 \supseteq \delta_1'$ is satisfied in $\overline{M}$. This is true since by construction, any node $i \in [0 \ldots N]$ is pointed to in $\overline{M}$ by a term in $\mathcal{T}$, and membership in δ_1, δ_1' is preserved for such terms.

It is left to show that $\delta_1 \neq \delta_1'$ is satisfied in $\overline{M}$. We do this by identifying a term in $\mathcal{T}$ that M interprets as a node in $M[\delta_1] - M[\delta_1']$. Such a term must point to a node in $\overline{M}$ that is a member of $\overline{M}[\delta_1] - \overline{M}[\delta_1']$. To perform this identification, let ℓ be a node in $M[\delta_1] - M[\delta_1']$. Let $M[x] = r_1, \ldots, r_q = \ell$ denote the shortest *Nxt*-path in M from the node $M[x]$ to ℓ, i.e., for $i = 1, \ldots, q-1$, $M[\mathit{Nxt}](r_i) = r_{i+1}$. Let j be the maximal index in $[1..q]$ such that $r_j \in \{n_0, \ldots, n_m\}$, i.e., r_j is the M-image of some term $t \in \mathcal{T}$. If $r_j \notin M[\delta_1']$, our identification is complete.

Assume therefore that $r_j \in M[\delta_1']$. According to our construction, there exists an $M[\mathit{Nxt}]$-chain connecting r_j to ℓ, proceeding along $r_{j+1}, r_{j+2}, \ldots, \ell$. Consider the chain of $M[\mathit{Nxt}']$-links starting from r_j. At one of the intermediate nodes: $r_j, \ldots, \ell$, the $M[\mathit{Nxt}]$-chain and the $M[\mathit{Nxt}']$-chain must diverge, otherwise ℓ would also belong to $M[\delta_1']$. Assume that the two chains diverge at r_k, for some $j \leq k < q$. Then, according to strong uniformity (Lemma 6.4), $r_{k+1} \in \{n_0, \ldots, n_m\}$, contradicting the assumed maximality of j.

In the other direction, assume that $\overline{M}$ satisfies $\delta_1 \supset \delta_1'$. We first show that M satisfies $\delta_1 \supseteq \delta_1'$. Let n be a node in $M[\delta_1']$, and consider a Nxt'-path from $M[x']$ to n in M. Let m be the ancestor nearest to n that is pointed to by a term in $\mathcal{T}$. By assumption, $m \in \overline{M}[\delta_1]$. From property **P2** of Observation 6.9 it follows that $m \in M[\delta_1]$. The fact $n \in M[\delta_1]$ follows by induction on path length from m to n and by uniformity of M and $\overline{M}$. Therefore $M[\delta_1] \supseteq M[\delta_1']$. We now show that M satisfies $\delta_1 \supset \delta_1'$. Let j be a node such that $j \in \overline{M}[\delta_1] - \overline{M}[\delta_1']$. By construction, either for some term $t \in \mathcal{T}$, $j = \overline{M}[t]$, or $j \in [\alpha..(\alpha + \beta)]$. In the first case, t points to a node n_j in M, such that $n_j \in M[\delta_1] - M[\delta_1']$, and we are done. In the latter case, from Observation 6.14 we have $\overline{M}[Nxt](j) = \overline{M}[Nxt'](j)$. Therefore, if j is not Nxt'-reachable from $\overline{M}[x']$, there must exist an $\overline{M}$-node $i \leq \alpha$, such that $i \in \overline{M}[\delta_1] - \overline{M}[\delta_1']$ and $\overline{M}[Nxt](i) \neq \overline{M}[Nxt'](i)$. From the construction of $\overline{M}$ and Observation 6.9, we have $n_i \in M[\delta_1] - M[\delta_1']$ and $M[Nxt](n_i) \neq M[Nxt'](n_i)$.

It is left to show that $M \models (\delta_1 = \delta_1')$ iff $\overline{M} \models (\delta_1 = \delta_1')$. This is done by similar arguments.

The case of δ_2, while not presented here, is shown by generalization: While δ_1 involves nodes reachable from a single distinguished pointer x, δ_2 involves nodes on a path between x and a pointer y. Thus, given node ℓ satisfying some combination of properties of membership in δ_2 and δ_2', we identify a node satisfying the same properties, that is also pointed to by a term in $\mathcal{T}$. Here, however, we consider not only distant ancestors of ℓ on the path from

x, but also distant successors on the path to y. □

Example 6.5 (List Reversal Termination, concluded) In Example 6.4 we propose the ranking δ_1 to verify termination of program LIST-REVERSAL. From the Theorem 6.13 it follows that there is a small model property for the augmented program. The bound of the truncated system, according to Theorem 6.12, is

$$h_0 = |\mathcal{T}| = |\{0, x, y, x', y', Nxt'[x], Nxt[x]\}| = 7$$

We have computed the abstraction, and proved termination of LIST-REVERSAL using TLV.

6.4 Example: Bubble Sort

We present our experience in verifying a bubble sort algorithm on acyclic, singly-linked lists. The program is given in Fig. 6.5. The requirement of acyclicity is expressed in the initial condition $Nxt^*(x, 0)$ on the array Nxt. In Subsection 6.4.1 we summarize the proof of some safety properties. In Subsection 6.4.2 we discuss issues of computational efficiency, and in Subsection 6.4.3 we present a ranking abstraction for proving termination.

```
        $x, y, yn, prev, last$ : $[0..h]$
        $Nxt$                  : **array** $[0..h] \to [0..h]$ **init** $Nxt^*(x, 0)$
        $D$                    : **array** $[0..h] \to$ **bool**
0 :  $(prev, y, yn, last) := (0, x, Nxt[x], 0)$;
1 :  **while** $last \neq Nxt[x]$ **do**
       2 :  **while** $yn \neq last$ **do**
              3 :  **if** $(D[y] > D[yn])$ **then**
                     4 :  $(Nxt[y], Nxt[yn]) := (Nxt[yn], y)$;
                     5 :  **if** $(\boldsymbol{prev} = 0)$ **then**
                             6 :  $x := yn$
                          **else**
                             7 :  $Nxt[prev] := yn$;
                     8 :  $(prev, yn) := (yn, Nxt[y])$
                   **else**
                     9 : $(prev, y, yn) := (y, yn, Nxt[y])$
10 : $(prev, y, yn, last) := (0, x, Nxt[x], y)$;
11 :
```

Figure 6.5: Program Bubble Sort

6.4.1 Safety

Two safety properties of interest are preservation and sortedness, expressed as follows:

$$\forall t.(\pi = 0 \ \land \ t \neq nil \ \land \ Nxt^*(x,t)) \to \Box(Nxt^*(x,t)) \qquad (6.7)$$

$$\forall t, s.(\pi = 11 \ \land \ Nxt^*(x,t) \ \land \ Nxt^*(t,s)) \Rightarrow D[t] \leq D[s] \qquad (6.8)$$

As in Example 6.2 we augment the program with a generic variable for each universal variable. The initial abstraction consists of predicates collected from atomic formulas in properties (6.7) and (6.8) and from conditions in the program. These predicates are

$$last = Nxt[x],\ yn = last,\ D[y] > D[yn],\ prev = 0,\ t = 0,$$

$$Nxt^*(x, 0),\ Nxt^*(x, t),\ Nxt^*(t, s),\ D[t] \leq D[s]$$

This abstraction is too coarse for either property, requiring several iterations of refinement. Since we presently have no heuristic for refinement, new predicates must be derived manually from concretized counterexamples. In shape analysis typical candidates for refinement are reachability properties among program variables that are not expressible in the current abstraction. For example, the initial abstraction cannot express any nontrivial relation among the variables $x, last, y, yn$, and $prev$. Indeed, our final abstraction includes, among others, the predicates $Nxt^*(x, prev)$ and $Nxt^*(yn, last)$. In the case of *prev, y*, and *yn*, it is sufficient to use 1-step reachability, which is more efficiently computed. Hence we have the predicates $Nxt[prev] = y$ and $Nxt[y] = yn$.

6.4.2 Optimizing the Computation

When abstracting Bubble Sort, one difficulty, in terms of time and memory, is in computing the BDD representation of the abstraction mapping. This becomes apparent as the abstraction is refined with new graph reachability predicates. Naturally, computing the abstract program is also a major bottleneck.

One optimization technique used is to compute a series of increasingly more refined (and complex) abstractions $\alpha_1, \ldots, \alpha_n$, with α_n being the desired abstraction. For each $i = 1, \ldots, n-1$, we abstract the program using α_i and compute the set of abstract reachable states. Let φ_i be the concretization of this set, which represents the strongest invariant expressible by the predicates in α_i. We then proceed to compute the abstraction according to α_{i+1}, while using the invariant φ_i to limit the state space. This technique has been invaluable in limiting state explosion, almost doubling the size of models we have been able to handle.

6.4.3 Liveness

Proving termination of Bubble Sort is more challenging than that of List-Reversal due to the nested loop. While a deductive framework would require constructing a global ranking function, the current framework requires only to identify individual rankings of each loop. Our aim is to automate the proof using the counter-example guided abstraction refinement algorithm of Fig. 4.3. The inputs to the algorithm are, in addition to the fds to be verified and property specification, an initial predicate base and ranking core. The algorithm is parameterized by a heuristic for synthesizing new predicates (used in *Case 1*), and another for synthesizing new ranking functions (used in *Case 3*). As an initial predicate base we use the predicates used in Subsection 6.4.1 to verify safety, which turn out to be sufficient (hence we avoid the need to deal with predicate refinement).

In order to provide a heuristic for synthesis of new ranking functions, we adapt the simple heuristic discussed in Section 4.4 of searching for simple linear constraints to the domain of heaps with single successors, with the relation $Nxt^+(v_1, v_2) : Nxt^*(v_1, v_2) \ \wedge \ v_1 \neq v_2$ imposing a partial order over heap nodes. Here, however, we must overcome the fact that Nxt^+ does not remain constant throughout a computation, due to updates performed by the FHS to the Nxt array.

Given the loop part of an abstract counterexample, the heuristic attempts to show that there exists a path between two **index** variables that becomes shorter after an iteration of the loop. Let $L : S_k, \ldots, S_m$ be such an abstract loop, and let $\beta_{k,m}$, or β_L for short, be a bi-assertion as defined in Section 4.1. Then two variables v_1 and v_2 are sought such that the following constraint holds:

$$iter_{v_1,v_2} : \beta_L \quad \rightarrow \quad Nxt^+(v_1, v_1') \ \vee \ Nxt^+(v_2', v_2)$$

Intuitively, this constraint captures the notion of iteration of either pointer toward the other, or possibly the removal of nodes inside the path from v_1 to v_2 (via edge deletion).

Since the FHS is allowed updates to the Nxt array, the Constraint $iter_{v_1,v_2}$ on its own is not sufficient to ensure the decrease of the nodes on the path from v_1 to v_2. Thus we impose the following additional conditions:

- $Nxt^*(v_1, v_2) \;\wedge\; Nxt^*(v_2, 0)$ is a loop invariant, i.e.,

$$\begin{aligned}\varphi_{v_1,v_2} : \beta_L \;\; \rightarrow \;\; & Nxt^*(v_1, v_2) \;\wedge\; Nxt^*(v_2, 0) \;\wedge \\ & Nxt'^*(v_1', v_2') \;\wedge\; Nxt'^*(v_2', 0)\end{aligned}$$

- No nodes are added to the path from v_1 to v_2. Assuming that φ_{v_1,v_2} holds[1], then this is expressed by

$$\begin{aligned}\psi_{v_1,v_2} : \beta_L \;\; \rightarrow \;\; & \forall i \;.\; (Nxt'^*(v_1', i) \;\wedge\; \neg Nxt'^*(v_2', i) \rightarrow \\ & \quad Nxt^*(v_1, i) \;\wedge\; \neg Nxt^*(v_2, i))\end{aligned}$$

Then it can be shown that if the conjunction $iter_{v_1,v_2} \wedge \varphi_{v_1,v_2} \wedge \psi_{v_1,v_2}$ holds, then the loop L is well-founded, as proven by the ranking function $\delta_2(v_1, v_2)$. Formally,

$$iter_{v_1,v_2} \;\wedge\; \varphi_{v_1,v_2} \;\wedge\; \psi_{v_1,v_2} \;\wedge\; \beta_L \rightarrow (\delta_2(v_1, v_2) \supset \delta_2(v_1', v_2'))$$

In the case that v_2 is the constant 0 then the function $\delta_1(v_1)$ is adequate as well.

Returning to termination of Bubble Sort, we begin the abstraction refinement loop with the aforementioned predicate base and an empty ranking core. On the first iteration a counterexample is generated in which the inner loop does not terminate. The ranking heuristic suggests the ranking $\delta_1(yn)$

[1]Our language is not expressive enough to denote the general property that a node v_3 lies *between* nodes v_1 and v_2. Hence we restrict to the case of acyclic chains.

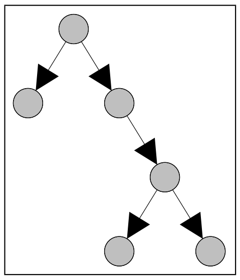
(which is a simplified version of $\delta_2(yn, 0)$). In this example, the more complex function $\delta_2(yn, \mathit{last})$ is another candidate. The next iteration results in a counterexample with a nonterminating outer loop, for which the heuristic suggests the ranking $\delta_2(x, \mathit{last})$.

Chapter 7

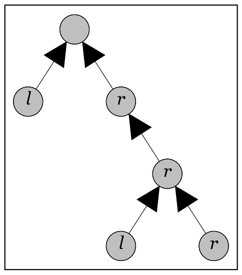
Complex Heap Shapes

In Chapter 6 a framework is proposed for shape analysis of singly-linked graphs based on a small model property of a restricted class of first order assertions with transitive closure. Extending this framework to allow for heaps with multiple links per node entails extending the assertional language and proving a stronger small model property. At this point, it is not clear whether such a language extension is decidable (see [GOR97, IRR$^+$04a] for relevant results).

This chapter deals with verification of programs that perform destructive updates of heaps consisting only of trees of bounded or unbounded arity, to which we refer as *multi-linked* heaps. We bypass the need to handle trees directly by transforming heaps consisting of multiple trees into structures consisting of singly-linked lists (possibly with shared suffixes). This is accomplished by "reversing" the parent-to-child edges of the trees populating



(a) A Multi-Linked Heap



(b) The Corresponding Single-Parent Heap

Figure 7.1: Multi-Linked to Single-Parent Heap Transformation

the heap, as well as associating scalar data with nodes. We refer to the transformed heap as a *single-parent* heap. Fig. 7.1(a) and Fig. 7.1(b) together demonstrate the transformation of a multi-linked heap that consists of a binary tree to its single-parent counterpart. In the latter graph, edges are directed from children to parents, and each child is annotated with boolean information denoting whether it is a left or right child.

Verification of temporal properties of multi-linked heap systems can be performed as follows: Given a multi-linked system and a temporal property, the system and property are (automatically) transformed into their single-parent counterparts. Then, the shape analysis framework of Chapter 6 is applied. If a counter-example (on the transformed system) is produced, it is automatically mapped into a counter-example of the original (multi-linked) system.

The chapter is organized as follows: Section 7.1 defines systems over single-parent heaps and Section 7.2 describes their model reduction. Section 7.3 defines systems over multi-linked heaps, and Section 7.4 shows how to transform them to single-parent heap systems.

7.1 Single-Parent Heaps

A *single-parent heap system* is an extension of the model of *finite heap systems* (FHS) of Chapter 6 specialized for representing trees. Such a system is parameterized by a positive integer h, which is the heap size. Some auxiliary arrays may be used to specify more complex structures (e.g., ordered trees). However, each node u has a single link to which we refer as its "parent," and denote it by $parent(u)$.

Example 7.1 (Tree Insertion) For example, we present in Fig. 7.2 a program that inserts a node into a binary sort tree rooted at a node r. If the data contained in node n is not already contained in the tree, then n is inserted as a new leaf. Otherwise the tree is not modified. Obviously, n is to be an isolated node, i.e., to be both a root and a leaf. To allow for the presentation of a sorted binary tree, we use an array ct (child-type) such that $ct[u]$ equals *left* or *right* if node u is, respectively, the left or right child of its parent. We also require that any two children of the same parent must have different child-types. In Subsection 7.1.2 we expand on the notion of sibling order. One may wish to show, for example, that program TREE-INSERT satisfies the following for every x:

$$\textit{no-loss}: \quad \textit{parent}^*(x,r) \rightarrow \Box\, \textit{parent}^*(x,r)$$

$$\textit{no-gain}: \quad x \neq n \ \wedge\ \neg\textit{parent}^*(x,r) \rightarrow \Box\, \neg\textit{parent}^*(x,r)$$

```
r, t   : [1..h]                          init  t = r
n      : [0..h]                          init  n > 0
parent : array [0..h] of [0..h]          init  parent[n] = parent[r] = 0 ∧ parent[0] = 0 ∧
                                                ∀u . parent[u] ≠ n
ct     : array [0..h] of {left, right}   init  ∀i ≠ j . parent[i] = parent[j] ≠ 0 → ct[i] ≠ ct[j]
data   : array [0..h] of [1..k]
done   : bool                            init  done = FALSE
         [ 1 :  while ¬done do
                  [ 2 :  if data[n] = data[t] then
                            3 : done := TRUE
                    4 :  elseif data[n] < data[t] then
                            [ 5 :  if ∀j.parent[j] ≠ t ∨ ct[j] ≠ left then
                                      6 : (parent[n], ct[n]) := (t, left)
                                      7 : done := TRUE
                                   else
                                      8 : t := ϵ j . parent[j] = t ∧ ct[j] = left ]
                    9 :  elseif ∀j.parent[j] ≠ t ∨ ct[j] ≠ right then
                            10 : (parent[n], ct[n]) := (t, right)
                            11 : done := TRUE
                         else
                            12 : t := ϵ j . parent[j] = t ∧ ct[j] = right ]
           13 : ]
```

Figure 7.2: Program Tree-Insert

The ϵ-expressions, $\epsilon\ j\ .\ \textit{cond}$ in lines 8 and 12 denote "choose any node j that satisfies *cond*." For both statements in this program, it is easy to see that there is exactly one node j that meets *cond*. However, this is not always the case, and then such an assignment is interpreted non-deterministically. In case *cond* is unsatisfiable we arbitrarily fix the chosen node to be 0. We also allow for universal tests, as those in lines 5 and 9, that test for existence of a particular node's left or right child. ┘

$$
\begin{array}{l}
\mathit{error} \ \wedge \ \mathit{error}' \ \wedge \ \mathit{presEx}(\mathit{error}) \\
\quad \vee \\
\neg \mathit{error} \ \wedge \\
\left[
\begin{array}{ll}
 & \pi = 1 \ \wedge \ \neg \mathit{done} \ \wedge \ \pi' = 2 \ \wedge \ \mathit{presEx}(\pi) \\
\vee & \pi = 1 \ \wedge \ \mathit{done} \ \wedge \ \pi' = 13 \ \wedge \ \mathit{presEx}(\pi) \\
\vee & \pi = 2 \ \wedge \ \mathit{data}[t] = \mathit{data}[n] \ \wedge \ \pi' = 3 \ \wedge \ \mathit{presEx}(\pi) \\
\vee & \pi = 2 \ \wedge \ \mathit{data}[t] \neq \mathit{data}[n] \ \wedge \ \pi' = 4 \ \wedge \ \mathit{presEx}(\pi) \\
\vee & \pi = 3 \ \wedge \ \pi' = 1 \ \wedge \ \mathit{done}' \ \wedge \ \mathit{presEx}(\pi, \mathit{done}) \\
\vee & \pi = 4 \ \wedge \ \mathit{data}[n] < \mathit{data}[t] \ \wedge \ \pi' = 5 \ \wedge \ \mathit{presEx}(\pi) \\
\vee & \pi = 4 \ \wedge \ \mathit{data}[t] \leq \mathit{data}[n] \ \wedge \ \pi' = 9 \ \wedge \ \mathit{presEx}(\pi) \\
\vee & \pi = 5 \ \wedge \ \pi' = 6 \ \wedge \ (\forall j.\mathit{parent}[j] \neq t \ \vee \ \mathit{ct}[j] \neq \mathit{left}) \ \wedge \ \mathit{presEx}(\pi) \\
\vee & \pi = 5 \ \wedge \ \pi' = 8 \ \wedge \ (\exists j.\mathit{parent}[j] = t \ \wedge \ \mathit{ct}[j] = \mathit{left}) \ \wedge \ \mathit{presEx}(\pi) \\
\vee & \pi = 6 \ \wedge \ n = 0 \ \wedge \ \mathit{error}' \ \wedge \ \mathit{presEx}(\mathit{error}) \\
\vee & \pi = 6 \ \wedge \ \pi' = 7 \ \wedge \ n \neq 0 \ \wedge \ \mathit{parent}'[n] = t \ \wedge \ \mathit{ct}'[n] = \mathit{left} \ \wedge \\
 & \mathit{presEx}(\pi, \mathit{parent}[n], \mathit{ct}[n]) \\
\vee & \pi = 7 \ \wedge \ \pi' = 1 \ \wedge \ \mathit{done}' \ \wedge \ \mathit{presEx}(\pi, \mathit{done}) \\
\vee & \pi = 8 \ \wedge \ \pi' = 1 \ \wedge \ (\exists j \ . \ \mathit{parent}[j] = t \ \wedge \ \mathit{ct}[j] = \mathit{left} \ \wedge \ t' = j) \ \wedge \ \mathit{presEx}(\pi, t) \\
\vee & \pi = 9 \ \wedge \ \pi' = 10 \ \wedge \ (\forall j.\mathit{parent}[j] \neq t \ \vee \ \mathit{ct}[j] \neq \mathit{right}) \ \wedge \ \mathit{presEx}(\pi) \\
\vee & \pi = 9 \ \wedge \ \pi' = 12 \ \wedge \ (\exists j.\mathit{parent}[j] = t \ \vee \ \mathit{ct}[j] = \mathit{right}) \ \wedge \ \mathit{presEx}(\pi) \\
\vee & \pi = 10 \ \wedge \ n = 0 \ \wedge \ \mathit{error}' \ \wedge \ \mathit{presEx}(\mathit{error}) \\
\vee & \pi = 10 \ \wedge \ \pi' = 11 \ \wedge \ n \neq 0 \ \wedge \ \mathit{parent}'[n] = t \ \wedge \ \mathit{ct}'[n] = \mathit{right} \ \wedge \\
 & \mathit{presEx}(\pi, \mathit{parent}[n], \mathit{ct}[n]) \\
\vee & \pi = 11 \ \wedge \ \pi' = 1 \ \wedge \ \mathit{done}' \ \wedge \ \mathit{presEx}(\pi, \mathit{done}) \\
\vee & \pi = 12 \ \wedge \ \pi' = 1 \ \wedge \ (\exists j \ . \ \mathit{parent}[j] = t \ \wedge \ \mathit{ct}[j] = \mathit{right} \ \wedge \ t' = j) \ \wedge \ \mathit{presEx}(\pi, t) \\
\vee & \pi = 13 \ \wedge \ \pi' = 13 \ \wedge \ \mathit{presEx}(\pi)
\end{array}
\right]
\end{array}
$$

Figure 7.3: Transition Relation of TREE-INSERT

7.1.1 Unordered Single-Parent Heaps

We define the class of single-parent heap systems as an extension of the FHS model of Chapter 6, with extensions to the assertional language. To allow for backward and forward traversal of tree-like structures, we extend the definition of Section 6.1 and define a restricted EA-assertion to be of the form $\exists \vec{x} \ . \ \psi(\vec{u}, \vec{x})$ where ψ is a positive boolean combination of atomic assertions and their negations, preservation assertions, TCF-assertions, and the following forms, which we label Z-assertions:

- $\forall y \ . \ Z[y] \neq u$,
- $\forall y \ . \ Z[y] \neq u \ \vee \ B[y]$, and

- $\forall y \ . \ Z[y] \neq u \ \vee \ \neg B[y]$,

Fig. 7.3 presents the transition relation associated with the program of Fig. 7.2. The implied encoding introduces an additional **bool** variable *error* which is set to True whenever there is an attempt to assign a value to $A[0]$, for some array A. Consequently, the transitions corresponding to statements 6 and 10 set *error* to True if $n = 0$, which is tested before assigning values to $\mathit{parent}[n]$ and to $\mathit{ct}[n]$.

7.1.2 Ordered Single-Parent Heaps

We now formalize the notion of order among siblings, as seen in Example 7.1. An *ordered* single-parent heap system is one that includes a distinguished $\mathit{ct} : \mathbf{index} \rightarrow [1..k]$ array, for some constant k, that denotes for each heap node its place among its siblings. This allows the subtrees of a given root node to be distinguished by their *ct* order. We now extend the assertional language with a new type of atomic formula: For each $i \in [1..k]$, the formula $i\text{-}\mathit{subtree}_Z(x_1, x_2)$ denotes that x_1 is in the i^{th} subtree of x_2, where x_1 and x_2 are **index** variables and Z is an **index** array. This is formally expressed by the formula

$$i\text{-}\mathit{subtree}_Z(x_1, x_2) : \ \exists u \ . \ Z[u] = x_2 \ \wedge \ \mathit{ct}[u] = i \ \wedge \ Z^*(x_1, u)$$

We support these predicates explicitly rather than as derived forms because, due to the transitive closure over a quantified variable, they would otherwise be outside of the assertional language allowed for abstraction predicates. Throughout the paper, when the **index** array Z is apparent from the context, we use the short form $i\text{-}subtree(x_1, x_2)$. For example, in the context of program TREE-INSERT of Example 7.1, the predicates *left-subtree* and *right-subtree* denote the left and right subtree relations among nodes of the *parent* array, whereas *left-subtree′* and *right-subtree′* denote subtree relations among nodes of the *parent′* array.

7.2 Computing Symbolic Abstractions of Single-Parent Heaps

In order to apply the methodology of Chapter 6 to compute the abstraction of a single-parent heap system symbolically, we must show a small model property establishing that satisfiability of a restricted EA-assertion is checkable on a small instantiation of a system. The main effort here is dealing with the extensions to the assertional language introduced for single-parent heap systems. For simplicity, it is assumed that all scalar values are represented by multiple boolean values.

Let φ be a restricted EA-assertion without any existentially quantified variables. To justify this limitation, the reader is referred to Corollary 6.10, which generalizes the small property to existentially quantified assertions. We define a set of terms $\mathcal{T}_\varphi$ to be the minimal set that satisfies the following:

- $\mathcal{T}_\varphi$ contains the term 0 and all free terms in φ;
- For every **bool** array $B \in \mathcal{V}$, if $B[u] \in \varphi$, then if B is unprimed, $\mathit{parent}[u] \in \mathcal{T}_\varphi$, and if B is primed, $\mathit{parent}'[u] \in \mathcal{T}_\varphi$;
- $\mathcal{T}_\varphi$ is history-closed.

In comparison to the term sets that are dealt with in Chapter 6, here for every free boolean term $B[u]$, the corresponding **index** term ($\mathit{parent}[u]$ or $\mathit{parent}'[u]$) is also taken into consideration. The increase in the size of $\mathcal{T}_\varphi$ will entail a slight increase in the small model bound presented in this section.

Let M be a model that satisfies φ with size greater then $f(\varphi) = |\mathcal{T}_\varphi| + \max(|\mathcal{R}_\varphi|, 1)$. We show how to construct a reduced model $\overline{M}$ of a size that is linear in $f(\varphi)$, and then proceed to show that if φ is satisfiable, then it is satisfiable by a model of such bounded size.

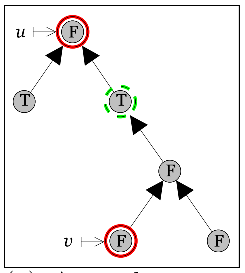
Let $\mathcal{N}$ and $\mathcal{S}$ be the sets of nodes $\{n_0, \ldots, n_\alpha\}$ and $\{n_{\alpha+1}, \ldots, n_{\alpha+\beta}\}$ as defined in Section 6.2. As defined there, let Γ be the $\mathcal{N} \cup \mathcal{S} \mapsto [0..\{\alpha+\beta\}]$ mapping $\Gamma(n_i) = i$. The construction of the reduced model is given by the definition below.

Definition 7.1 (Single-Parent Model Reduction)**.** *The reduction of a model M of φ is the model $\overline{M}$, which is constructed as in Definition 6.7, with the following exception:*

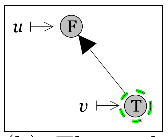
- *For every unprimed* **bool** *array B and $j \in [1..\alpha]$, if n_j has a parent-representative, then $\overline{M}[B](j) = M[B](r_M^{\mathit{parent}}(n_j))$. Otherwise $\overline{M}[B](j) =$*

$M[B](n_j)$. For every primed **bool** *array B and $j \in [1..\alpha]$, if n_j has a parent′-representative, then $\overline{M}[B](j) = M[B](r_M^{parent'}(n_j))$. Otherwise $\overline{M}[B](j) = M[B](n_j)$. Finally, for every $j \in [\alpha + 1..\alpha + \beta]$, $\overline{M}[B](j) = M[B](n_j)$.*

Example 7.2 (Model Reduction)



(a) A single-parent heap model M



(b) The reduction $\overline{M}$ of M

Figure 7.4: Model Reduction

Let *parent* and *data* be **index** and **bool** arrays respectively, and let φ be the assertion:

$$\varphi: \quad u \neq v \ \wedge \ \forall y \ . \ (parent[y] \neq u \ \vee \ data[y])$$

Since no array term refers to the u^{th} or v^{th} element, it follows that $\mathcal{T}_\varphi$ consists only of the **index** terms 0, u, and v. Let M be a model of φ of size 7, as shown in Fig. 7.4(a). The interpretations by M of terms in $\mathcal{T}_\varphi$ are the highlighted nodes. Each node y is annotated with the value $M[data](y)$ (e.g., the node pointed to by u has a data value of False). $\overline{M}$, which is the reduction of M with respect to $\mathcal{T}_\varphi$, is given in Fig. 7.4(b). The M representative of *parent* for $M[v]$ is given by the node highlighted by a dashed line in Fig. 7.4(a). As shown here, the node pointed to by v in $\overline{M}$ takes on the properties of this representative node. ┘

Theorem 7.2. *Let φ be a restricted EA-assertion without existentially quantified variables. If $M \models \varphi$ then φ is satisfiable by a model of size at most $f(\varphi)$.*

Proof. We begin by observing that the properties **P1**, **P2**, and **P3** of the model construction given in Observation 6.9 continue to hold over the small model $\overline{M}$ defined above. We make an additional observation, which follows immediately from the construction:

P6. If $B'[u]$ occurs in φ for some $u \in \mathcal{T}_\varphi$ and a **bool** array $B \in \mathcal{V}$, then u, *parent*$[u]$, and *parent*$'[u]$ are all in $\mathcal{T}_\varphi$.

Assume that $M \models \varphi$. As in the proof of Theorem 6.8, to show that $\overline{M} \models \varphi$ we show that (1) every atomic formula ψ is true in $\overline{M}$ iff it is true in M, and (2) every universal sub-formula ψ, which only appears under positive polarity, that is satisfied in M is also satisfied in $\overline{M}$. We only present the cases for which the proof differs from that of Theorem 6.8.

ψ is an atomic assertion of the form $i\text{-}subtree_Z(x_1, x_2)$. Z, x_1, and x_2 are assumed to be an **index** array and **index** variables, respectively. In this case, we are dealing with an ordered heap as defined in Subsection 7.1.2, and assume the presence of an array ct : **index** $\rightarrow [1..k]$, with $i \in [1..k]$. In one direction, assume that $M \models \psi$. Expanding the definition of ψ to $\exists u\, .Z[u] = x_2 \;\wedge\; ct[u] = i \;\wedge\; Z^*(x_1, u)$, we conclude that $M \models Z^*(x_1, x_2)$.

We first identify the Z-chain from x_1 to x_2 in M, i.e. the node sequence $M[x_1] = u_1, \ldots, u_\ell, u_{\ell+1} = M[x_2]$ such that $M[Z](u_j) = u_{j+1}$, for every $j = 1, \ldots, \ell$. Let n_j be the node u_a, for the maximal $a \in [1..\ell]$, such that $n_j \in N$. Then u_ℓ is the Z-representative of n_j. Since $M[Z](u_\ell) = u_{\ell+1} = M[x_2]$, it must be the case that $M[ct](u_\ell) = i$. By construction, $\overline{M}[ct](j) = M[ct](u_\ell) = i$, and $\overline{M}[Z](j) = \Gamma(M[Z](u_\ell)) = \Gamma(M[x_2])$. Furthermore, from property **P3** we conclude that node j is Z-reachable from node $\overline{M}[x_1]$ in $\overline{M}$. Thus, x_1 is in the i^{th} subtree of x_2 in $\overline{M}$, i.e., $\overline{M} \models \exists u\, .Z[u] = x_2 \;\wedge\; ct[u] = i \;\wedge\; Z^*(x_1, u)$, and the claim holds.

In the other direction, assume that $\overline{M} \models \psi$. Let $\overline{M}[x_1] = j \leq \alpha$ and $\overline{M}[x_2] = \ell \leq \alpha$. The claim is proven by considering the Z-chain in $\overline{M}$ from j to ℓ and, based on the construction of $\overline{M}$, constructing a corresponding Z-chain in M from $M[x_1] = n_j$ to $M[x_2] = n_\ell$ in which n_j is in the i^{th} subtree of n_ℓ.

ψ is an atomic assertion of the form $B[u]$. u is an **index** variable and B is a **bool** array. It then follows that $\mathit{parent}[u]$ or $\mathit{parent}'[u]$ is in $\mathcal{T}_\varphi$, according to whether B is unprimed or primed, and then it follows from the construction that $\overline{M}[B](u) = M[B](u)$.

ψ is a Z-assertion. Thus ψ has one of the forms $\forall y.Z[y] \neq u$, $\forall y.Z[y] \neq$

$u \lor B[y]$, or $\forall y.Z[y] \neq u \lor \neg B[y]$. We show here the second case; The other two are similar. Recall that u must be in $\mathcal{T}_\varphi$, and assume that $M(u) = n_j$. Assume, by way of contradiction, that $M \models \forall y.Z[y] \neq u \lor B[y]$ and for some $i \in [0..m{+}1]$, $\overline{M} \models Z[i] = j \land \neg B[i]$. If $i = m{+}1$, then obviously $M(Z)[i] = m{+}1$, and thus $\overline{M} \not\models Z[i] = u$. Hence, $i \neq m{+}1$. From property **P1** it follows that $M[Z](n_i) \neq n_j$. Thus, there exists a Z-representative $v \neq n_i$ for i in M. From the construction it follows that $\overline{M}(Z)[i] = \gamma(M(Z)[v])$ and that $\overline{M}(B)[i] = M(B)[v]$. From the assumption that $\overline{M} \models \neg B[i]$, it follows that $\neg M(B)[v]$, and from the assumption that $\overline{M} \models p$ it then follows that $M(Z)[v] \neq n_j$, contradicting the assumption that $\overline{M}(Z)[i] = j$.

ψ **is a preservation formula of a bool array.** Assume p is of the form $\forall y.B'[y] = B[y] \;\lor\; y \in Y$ where Y is a set of **index** variables in $\mathcal{T}_\varphi$. Assume that $M \models \psi$, and that $\overline{M} \not\models \psi$, i.e., for some $i \in [0..\alpha + \beta]$, $\overline{M} \models B'[i] \neq B[i] \;\land\; i \notin \gamma(Y)$. Since $\overline{M}[B](\alpha + j) = M[B](n_{\alpha+j})$, $\overline{M}[B'](\alpha + j) = M[B'](n_{\alpha+j})$, and $n_{\alpha+j} \notin Y$, for every $j \in [1..\beta]$, it follows that $i \leq \alpha$.

Similar to the case of preservation of **index** arrays, we consider the $M[Z]$-chain $n_i = u_0, \ldots$ and $M[Z']$-chain $n_i = v_0, \ldots$ in M, and conclude that $\overline{M}[B'](i) = \overline{M}[B](i)$. The only difference is in the inductive step: Let $k \geq 0$, and assume that for all $j \leq k$, $u_j = v_j$ and $u_j \notin Y$. If $M[Z'](v_k) = M[Z](v_k)$, then obviously $v_{k+1} = u_{k+1}$. Otherwise,

$M[Z'](v_k) \neq M[Z](v_k)$. From property **P6** it follows that v_k, u_{k+1}, and v_{k+1} are the interpretations of some terms in $\mathcal{T}_\varphi$, and hence are $\mathcal{N}$-nodes. It thus follows that n_i has the same $Z-$ and Z'-representative in M (which is either v_0, v_j for some $j < k$, or v_k) and therefore $\overline{M}(B)[i] = \overline{M}(B')[i]$.

ψ is a TCF-assertion of the form $\psi : \forall \vec{y}\ .\ P$. Recall that P is a positive boolean combination of formulae of the form $\neg Z^*(u, y)$ and $\mathbb{B}(y)$. Therefore it suffices to show that if M satisfies a clause ϕ which has either of the two forms, then $\overline{M}$ satisfies ϕ. As when proving Theorem 6.8, we interpret P by considering an arbitrary assignment $\overline{\eta}$ to the quantified variables $\vec{y}$ that assigns to each variable y in $\vec{y}$ a value $\overline{\eta}[y] \in [0..\alpha + \beta]$. We now choose an assignment η defined as follows:

$$\eta(y) = \begin{cases} r^Z(\Gamma^{-1}(\overline{\eta}[y])), & \text{if } \overline{\eta}[y] < \alpha \text{ and } \Gamma^{-1}(\overline{\eta}[y]) \text{ has a} \\ & Z\text{-representative} \\ \Gamma^{-1}(\overline{\eta}[y]), & \text{otherwise} \end{cases}$$

Denote by M_η the joint interpretation (M, η) that interprets all quantified variables according to η and all other terms according to M. Similarly, denote by $\overline{M}_\eta$ the joint interpretation $(\overline{M}, \overline{\eta})$. It remains to prove that $\overline{M}_\eta \models \phi$ under the assumption that $M_\eta \models \phi$. Let i_y be $\overline{M}_\eta[y]$. Assume first that ϕ is the formula $\mathbb{B}(y)$. Let $B[y]$ be a clause in $\mathbb{B}$. If $i_y > \alpha$ or if n_{i_y} has no Z-representative, then $M_\eta[y] = n_{i_y}$ by definition. Then from the construction of $\overline{M}$ we have $\overline{M}_\eta[B[y]] = \overline{M}[B](i_y) = M_\eta[B[y]] = M[B](n_{i_y})$. If $i_y \leq \alpha$, then first assume that n_{i_y} has a Z-representative in M. Thus, $\overline{M}_\eta[B](i_y) = M[B](r^Z_M(n_{i_y}))$, and, from the definition of η, it follows that $\overline{M}_\eta[B](i_y) = M[B](y)$. Thus, $\overline{M}_\eta \models \mathbb{B}(y)$ iff $M_\eta \models \mathbb{B}(i_y)$.

Assume that ϕ is the formula $\neg Z^*(u, y)$. Since u is a free term in ψ, it follows that $M[u] \leq \alpha$. It then follows from Properties **P2** and **P3** that $\overline{M}_\eta \not\models Z^*(u, y)$.

□

7.3 Multi-Linked Heap Systems

In this section we define *multi-linked heap systems* with a bounded out-degree on nodes. A multi-linked heap is represented similar to a single-parent heap, only, instead of having a single **index** array, we allow for some $k > 1$ **index** arrays, each describing one of the links a node may have. We denote these arrays by $link_1, \ldots, link_k$. Thus, each $link_i$ is an array $[0..h] \rightarrow [0..h]$. We are mainly interested in *non-sharing heaps*, defined as follows:

Definition 7.3. *A* non-sharing heap *is one that satisfies the following requirements:*

1. *For every $i = 1, \ldots, k$, $link_i[0] = 0$.*
2. *For every* **bool** *array B, $\neg B[0]$.*
3. *No two distinct positive nodes may share a common positive child. This*

requirement can be formalized as

$$\forall j, \ell \in [1..h], i, r \in [1..k] \,.\, (j \neq \ell) \wedge (link_i[j] = link_r[\ell]) \;\rightarrow\; link_i[j] = 0$$

4. *No two distinct links of a positive node may point to the same positive child. This can be formalized as*

$$\forall j \in [1..h], s, t \in [1..k] \,.\, (s \neq t) \wedge (link_s[j] = link_t[j]) \;\rightarrow\; link_s[j] = 0$$

┛

We refer to the conjunction of the requirements in Definition 7.3 by the formula *no_sharing*. A state violating one of these three requirements is called a *sharing state.*

A multi-linked system is called *sharing-free* if none of its computations ever reaches a sharing state, nor does a computation ever attempt to assign a value to $A[0]$ for some array A.

Let $\mathcal{D}$: $\langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ be a k-bounded multi-linked heap system. Table 7.1 describes a BNF-like syntax of the assertions used in describing $\mathcal{D}$, which we refer to as M-assertions. There, `Ivar` denotes an unprimed **index** variable, `Iarr` denotes an unprimed **index** array, `Bvar` denotes an unprimed **bool** variable, and `Barr` denotes an unprimed **bool** array. The expression *reach*(x, y) abbreviates $(x, y) \in (\bigcup_{i=1}^{k} link_i)^*$, and the expression *cycle*(x) abbreviates $(x, x) \in (\bigcup_{i=1}^{k} link_i)^+$. The **Preservation** assertion is just like in the single-parent case and we require that if `Assign` appears in τ, then the **Preservation** assertion that is conjoined with it includes preservation of all variables that don't appear in the left-hand-side of any clause of `Assign`.

$\texttt{MCond1}$	$::=$	$\textsc{True} \mid \texttt{Bvar} \mid \texttt{Barr}[\texttt{Ivar}] \mid \texttt{Ivar} = \texttt{Ivar} \mid \texttt{Ivar} = 0 \mid \texttt{Iarr}[\texttt{Ivar}] = \texttt{Ivar} \mid \texttt{Iarr}[\texttt{Ivar}] = 0 \mid \texttt{MCond1} \vee \texttt{MCond1} \mid \neg\texttt{MCond1}$
$\texttt{MCond2}$	$::=$	$\texttt{MCond1} \mid \mathit{reach}(\texttt{Ivar},\texttt{Ivar}) \mid \mathit{cycle}(\texttt{Ivar}) \mid \neg\texttt{MCond2} \mid \texttt{MCond2} \vee \texttt{MCond2}$
$\texttt{Assign}$	$::=$	$\epsilon \mid \texttt{Bvar}' \mid \neg\texttt{Bvar}' \mid \texttt{Barr}'[\texttt{Ivar}] \mid \neg\texttt{Barr}'[\texttt{Ivar}] \mid \texttt{Bvar}' = \texttt{Bvar} \mid \texttt{Ivar}' = 0 \mid \texttt{Ivar}' = \texttt{Ivar} \mid \texttt{Iarr}'[\texttt{Ivar}] = \texttt{Ivar} \mid \texttt{Iarr}'[\texttt{Ivar}] = 0 \mid \texttt{Assign} \wedge \texttt{Assign}$
Θ	$::=$	$\texttt{MCond2} \wedge \mathit{no\_sharing}$
ρ	$::=$	$\textsc{True} \mid \texttt{MCond1} \wedge \texttt{Assign} \wedge \textbf{Preservation} \mid \rho \vee \rho$
$\mathcal{J}$	$::=$	$\emptyset \mid \mathcal{J} \cup \{\texttt{MCond1}\}$
$\mathcal{C}$	$::=$	$\emptyset \mid \mathcal{C} \cup \{(\texttt{MCond1},\texttt{MCond1})\}$

Table 7.1: Grammar for Assertions for Multi-Linked Systems

For example, consider a binary tree, which is a multi-linked heap with bound 2 and no sharing. Each of *left* and *right* is a *link*. Program Tree-Insert in Fig. 7.5 is the standard algorithm for inserting a new node, n, into a sorted binary tree rooted at r.

```
left, right  :  array [0..h] of [0..h]   init  no_sharing
data         :  array [0..h] of bool
r, n         :  [1..h]                   init  ¬reach(r, n) ∧ ¬cycle(r) ∧
                                               left[n] = 0 ∧ right[n] = 0
t            :  [0..h]                   init  t = r
done         :  bool                     init  done = FALSE
     ⎡ 1 :  while ¬done do                                  ⎤
     ⎢         ⎡ 2 :  if data[n] = data[t] then          ⎤  ⎢
     ⎢         ⎢         3 : done := TRUE                ⎢  ⎢
     ⎢         ⎢ 4 :  elseif data[n] < data[t] then      ⎢  ⎢
     ⎢         ⎢         ⎡ 5 :  if left[t] = 0 then   ⎤  ⎢  ⎢
     ⎢         ⎢         ⎢         6 : left[t] := n   ⎢  ⎢  ⎢
     ⎢         ⎢         ⎢         7 : done := TRUE   ⎢  ⎢  ⎢
     ⎢         ⎢         ⎢      else                  ⎢  ⎢  ⎢
     ⎢         ⎢         ⎣         8 : t := left[t]   ⎦  ⎢  ⎢
     ⎢         ⎢ 9 :  elseif right[t] = 0 then           ⎢  ⎢
     ⎢         ⎢         10 : right[t] := n              ⎢  ⎢
     ⎢         ⎢         11 : done := TRUE               ⎢  ⎢
     ⎢         ⎢      else                               ⎢  ⎢
     ⎢         ⎣         12 : t := right[t]              ⎦  ⎢
     ⎣ 13 :                                                 ⎦
```

Figure 7.5: Multi-Linked Tree Insertion Algorithm

7.4 Reducing Multi-Linked into Single-Parent Heaps

We now show how to transform multi-linked heap systems into single-parent heap systems.

7.4.1 The Transformation

Let $\mathcal{D}_m\colon \langle \mathcal{V}_m, \Theta_m, \rho_m, \mathcal{J}_m, \mathcal{C}_m \rangle$ be a k-bounded multi-linked heap system. Thus, $\mathcal{V}_m$ includes the **index** arrays $link_1, \ldots, link_k$. We transform $\mathcal{D}_m$ into a single-parent heap system $\mathcal{D}_s\colon \langle \mathcal{V}_s, \Theta_s, \rho_s, \mathcal{J}_s, \mathcal{C}_s \rangle$ as follows:

The set of variables $\mathcal{V}_s$ consists of the following:

1. $\mathcal{V}_m \setminus \{link_1, \ldots, link_k\}$, i.e., we remove from $\mathcal{V}_m$ all the *link* arrays;
2. An **index** array $parent\colon [0..h] \mapsto [0..h]$ that does not appear in $\mathcal{V}_m$;
3. A **bool** array $ct\colon [0..h] \mapsto [0..k]$ that does not appear in $\mathcal{V}_m$ (recall our convention that "**bool**" can be any finite-domain type);
4. A new **bool** variable *error*; *error* is set when $\mathcal{D}_m$ contains an erroneous transition such as one that introduces sharing in the heap, or attempts to assign values to $A[0]$ for some array A.

Intuitively, we replace the **index** *link* arrays with a single **index** *parent* array that reverses the direction of the links, and assign to $ct[i]$ (*child type*) the "birth order" of i in the heap. The variable *error* is boolean and is set when $\mathcal{D}_m$ cannot be transformed into a singe-parent system. This is caused by either an assignment to $A[0]$ or by a violation of the non-sharing requirements. When such an error occurs, *error* is raised, and remains so, i.e., ρ_s implies $error \rightarrow error'$.

Definition 7.4. *A single-parent state is said to be* well formed *if the parent of 0 is itself, all the* **bool** *arrays $\mathcal{B} \subset \mathcal{V}_s$ associate 0 with the value* False*, and no parent has two distinct children with the same birth order, i.e.,*

$$\begin{aligned} \texttt{wf}\colon\quad & parent[0] = 0 \;\wedge\; \textstyle\bigwedge_{B\in\mathcal{B}}(\neg B[0]) \;\wedge \\ & \forall i \neq j \,.\, (parent[i] = parent[j] \neq 0 \;\rightarrow\; ct[i] \neq ct[j]) \end{aligned}$$

┘

To transform ρ_m, $\mathcal{J}_m$, and $\mathcal{C}_m$ into their $\mathcal{D}_s$ counterparts, it suffices to transform M-assertions over $\mathcal{V}_m \cup \mathcal{V}'_m$ into restricted EA-assertions over $\mathcal{V}_s \cup \mathcal{V}'_s$. To transform Θ_m, which is of the form $\mathit{no\_sharing} \wedge \varphi$, where φ is an `MCond2`, into Θ_s, we take the conjunction of `wf` and the transformation of φ. It thus remains to transform M-assertions. Recall that ρ_m is a disjunction of clauses (see Section 7.3), each one of the form

$$\varphi \ \wedge \ \tau \ \wedge \ \mathit{presEx}(\mathcal{V}_m - \{V\})$$

where $V \subseteq \mathcal{V}_m$, φ is an `MCond` over $\mathcal{V}_m$, and τ is an `Assign` statement of the form $\bigwedge_{v \in V} v' = E_v(\mathcal{V}_m)$ (where E_v is some expression). When we transform such a ρ_m-disjunct, we sometimes obtain several disjuncts. We assume that each has its obvious *presEx* assertions over $\mathcal{V}_s$. At times, for simplicity of representation, we do not express the transformation directly in DNF. Yet, in those cases, the DNF form is straightforward.

It thus remains to show how to transform M-assertions into restricted EA-assertions. This is done by induction on the M-assertions, where we ignore the preservation part (which, as discussed above, is defined by the transition relation for both $\mathcal{D}_m$ and $\mathcal{D}_s$.)

Let ψ be an M-assertion. In the following cases, ψ remains unchanged in the transformation:

1. ψ contains no reference to **index** variables and arrays;

2. ψ is of the form $x_1 = x_2$ where x_1 and x_2 are both primed, or both

unprimed, **index** variables;

3. ψ is of the form $x_1 = x_2$ where x_1 is a primed, and x_2 is an unprimed, **index** variable;

4. ψ is of the form $x = 0$ where x is a (either primed or unprimed) **index** variable;

5. ψ is of the form $B[x]$, where B is an unprimed **bool** array.

The other cases are treated below. We now denote primed variables explicitly, e.g., x_1 refers to an unprimed variable, and x_1' refers to a primed variable:

1. An assertion of the form $\mathit{link}_j[x_2] = x_1$ is transformed into

$$\begin{array}{ll} & (x_2 = 0 \;\wedge\; x_1 = 0) \\ \vee & (x_2 \neq 0 \;\wedge\; x_1 = 0 \;\wedge\; \forall z \,.\, (\mathit{parent}[z] \neq x_2 \vee \mathit{ct}[z] \neq j)) \\ \vee & (x_2 \neq 0 \;\wedge\; x_1 \neq 0 \;\wedge\; \mathit{parent}[x_1] = x_2 \;\wedge\; \mathit{ct}[x_1] = j) \end{array}$$

   In the case that $x_2 \neq 0$ and $x_1 = 0$, x_2 should have no j^{th} child. If $x_2 \neq 0$ and $x_1 \neq 0$, then x_1 should have x_2 as a parent and the child type of x_1 should be j.

2. A transitive closure formula $\mathit{reach}(x_1, x_2)$ is transformed into

$$(x_1 \neq 0 \;\wedge\; x_2 \neq 0 \;\wedge\; \mathit{parent}^*(x_2, x_1)) \;\vee\; (x_2 = 0)$$

   The first disjunct deals with the case where x_1 and x_2 are both non-0 nodes, and then the reachability direction is reversed, reflecting reversal of heap edges in the transformation to a single-parent heap. The second disjunct deals with the case that $x_2 = 0$, and then, since $k > 0$, there is a path from any node into 0.

3. A transitive closure formula $\mathit{cycle}(x)$, where x is an **index** variable, is transformed into $\mathit{parent}^*(\mathit{parent}[x], x)$.

4. An assertion of the form $x_1' = \mathit{link}_j[x_2]$ is transformed into:

$$\begin{array}{ll} & (x_2 = 0 \wedge x_1' = 0) \\ \vee & (x_2 \neq 0 \ \wedge \ x_1' = 0 \wedge \forall y \ . \ (\mathit{parent}[y] \neq x_2 \ \vee \ \mathit{ct}[y] \neq j)) \\ \vee & (x_2 \neq 0 \ \wedge \ \exists y \ . \ (\mathit{parent}[y] = x_2 \wedge \mathit{ct}[y] = j \wedge x_1' = y)) \end{array}$$

In case $x_2 = 0$, this transition sets x_1 to 0 since we assume that in non-sharing states $\mathit{link}_j[0] = 0$ for every $j = 1, \ldots, k$. Otherwise, if x_2 has no j^{th} child, then x_1 is set to 0. Otherwise, there exists a y which is the j^{th} child of x_2, and then x_1 is set to y.

5. An assertion of the form $B'[x]$, where B is an unprimed **bool** array, is transformed differently based on its polarity. If it appears under *positive* polarity, it is transformed into:

$$(x = 0 \ \wedge \ \mathit{error}') \ \vee \ (x \neq 0 \ \wedge \ B'[x])$$

The error condition reflects an attempt to assign True to $B[0]$. If the assertion $B'[x]$ appears under *negative* polarity, then no erroneous assignment is possible, and the assertion remains unchanged by the transformation.

6. An assertion of the form $\mathit{link}_j'[x_1] = x_2$ is transformed into:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{Err} \wedge \mathit{error}' \quad \vee} \\
 & \neg \mathit{Err} \\
\wedge & (x_2 = 0 \;\vee\; (x_2 \neq 0 \wedge \mathit{parent}'[x_2] = x_1 \;\wedge\; \mathit{ct}'[x_2] = j)) \\
\wedge & \left(\begin{array}{l} \quad \forall z\;.\;(\mathit{parent}[z] \neq x_1 \;\vee\; \mathit{ct}[z] \neq j) \\ \vee\; \exists z\;.\;(\mathit{parent}[z] = x_1 \;\wedge\; \mathit{ct}[z] = j \;\wedge\; (z = x_2 \vee \mathit{parent}'[z] = 0)) \end{array} \right)
\end{array}$$

Where *Err* is defined by:

$$(x_1 = 0 \wedge x_2 \neq 0) \vee (x_2 \neq 0 \wedge \mathit{parent}[x_2] \neq 0 \wedge (\mathit{parent}[x_2] \neq x_1 \vee \mathit{ct}[x_2] \neq j))$$

I.e., the assignment may cause an error by either attempting to assign a nonzero value to $\mathit{link}_j[0]$, or by introducing sharing (when x_2 either has a parent that is not x_1, or is x_1's i^{th} child for some $i \neq j$).

When there is no error, x_2 should become the j^{th} child of x_1 unless it is 0, which is expressed by the first conjunct of the non-error case; in addition, any node that was the j^{th} child of x_1 before the transition should become "orphaned," which is expressed by the second conjunct of the non-error case.

The following observation follows trivially from the construction above:

Observation 7.5. *The transformation of an* M*-assertion is a restricted EA-assertion.*

Having defined the system transformation, we can now demonstrate the complete verification process of the tree insertion program.

Example 7.3 (Verification of Tree-Insert)

$$
\begin{array}{rl}
\Theta: & \mathit{parent}[0] = 0 \ \land\ \forall i \neq j \ .\ (\mathit{parent}[i] = \mathit{parent}[j] \neq 0 \rightarrow \mathit{ct}[i] \neq \mathit{ct}[j]) \ \land \\
& \neg \mathit{parent}^*(n, r) \ \land\ \forall i \ .\ (\mathit{parent}[i] \neq n) \ \land\ t = r \ \land\ \neg \mathit{parent}^*(\mathit{parent}[r], r) \ \land\ \neg \mathit{done} \\
\rho: & \mathit{error} \ \land\ \mathit{error}' \ \land\ \mathit{presEx}(\mathit{error}) \\
& \quad \lor \\
& \neg \mathit{error} \ \land \\
& \left[\begin{array}{ll}
 & \pi = 1 \ \land\ \neg \mathit{done} \ \land\ \pi' = 2 \ \land\ \mathit{presEx}(\pi) \\
\lor & \pi = 1 \ \land\ \mathit{done} \ \land\ \pi' = 13 \ \land\ \mathit{presEx}(\pi) \\
\lor & \pi = 2 \ \land\ \mathit{data}[t] = \mathit{data}[n] \ \land\ \pi' = 3 \ \land\ \mathit{presEx}(\pi) \\
\lor & \pi = 2 \ \land\ \mathit{data}[t] \neq \mathit{data}[n] \ \land\ \pi' = 4 \ \land\ \mathit{presEx}(\pi) \\
\lor & \pi = 3 \ \land\ \pi' = 1 \ \land\ \mathit{done}' \ \land\ \mathit{presEx}(\pi, \mathit{done}) \\
\lor & \pi = 4 \ \land\ t \neq 0 \ \land\ \mathit{data}[n] < \mathit{data}[t] \ \land\ \pi' = 5 \ \land\ \mathit{presEx}(\pi) \\
\lor & \pi = 4 \ \land\ (t = 0 \ \lor\ \mathit{data}[t] \leq \mathit{data}[n]) \ \land\ \pi' = 9 \ \land\ \mathit{presEx}(\pi) \\
\lor & \mathit{try}(5, \mathit{left}) \ \lor\ \mathit{try}(9, \mathit{right}) \\
\lor & \pi = 13 \ \land\ \pi' = 13 \ \land\ \mathit{presEx}(\pi)
\end{array} \right] \\
\\
\mathit{try}(\mathit{link}, \pi_0): & \left[\begin{array}{ll}
 & \pi = \pi_0 \ \land\ \pi' = \pi_0 + 1 \ \land\ \mathit{presEx}(\pi) \ \land \\
 & (t = 0 \ \lor\ (t \neq 0 \ \land\ \forall j \ .\ \mathit{parent}[j] \neq t \ \lor\ \mathit{ct}[j] \neq \mathit{left})) \\
\lor & \pi = \pi_0 \ \land\ \pi' = \pi_0 + 3 \ \land\ t \neq 0 \ \land\ \mathit{presEx}(\pi) \\
 & (\exists j \ .\ \mathit{parent}[j] = t \ \land\ \mathit{ct}[j] = \mathit{left}) \\
\lor & \pi = \pi_0 + 1 \ \land\ \mathit{error}' \ \land\ \mathit{presEx}(\mathit{error}) \ \land \\
 & (t = 0 \ \lor\ (t \neq 0 \ \land\ \mathit{parent}[n] \neq 0 \ \land\ (\mathit{parent}[n] \neq t \ \lor\ \mathit{ct}[n] \neq \mathit{left}))) \ \land \\
\lor & \pi = \pi_0 + 1 \ \land\ \pi' = \pi_0 + 2 \ \land\ t \neq 0 \ \land \\
 & (\mathit{parent}[n] = 0 \ \lor\ (\mathit{parent}[n] = t \ \land\ \mathit{ct}[n] = \mathit{left})) \ \land \\
 & \mathit{parent}'[n] = t \ \land\ \mathit{ct}'[n] = \mathit{left} \ \land\ \mathit{presEx}(\pi, \mathit{parent}[n], \mathit{ct}[n]) \ \land \\
 & \left(\begin{array}{ll}
 & \forall j \ .\ (\mathit{parent}[j] \neq t \ \lor\ \mathit{ct}[j] \neq \mathit{left}) \\
 \lor & \exists j \ .\ (\mathit{parent}[j] = t \ \land\ \mathit{ct}[j] = \mathit{left} \ \land\ (j = n \ \lor\ \mathit{parent}'[j] = 0))
 \end{array} \right) \ \land \\
\lor & \pi = \pi_0 + 2 \ \land\ \pi' = 1 \ \land\ \mathit{done}' \ \land\ \mathit{presEx}(\pi, \mathit{done}) \\
\lor & \pi = \pi_0 + 3 \ \land\ \pi' = 1 \ \land\ \mathit{presEx}(\pi, t) \ \land \\
 & \left(\begin{array}{ll}
 & t = 0 \ \land\ t' = 0 \\
 \lor & t \neq 0 \ \land\ t' = 0 \ \land\ \forall j \ .\ (\mathit{parent}[j] \neq t \ \lor\ \mathit{ct}[j] \neq \mathit{left}) \\
 \lor & t \neq 0 \ \land\ \exists j \ .\ (\mathit{parent}[j] = t \ \land\ \mathit{ct}[j] = \mathit{left} \ \land\ t' = j)
 \end{array} \right)
\end{array} \right]
\end{array}
$$

Figure 7.6: Single-Parent Counterpart of Multi-Linked Tree Insertion

We wish to verify that the multi-linked tree insertion program given in Fig. 7.5 satisfies the following specification:

$$
\begin{aligned}
\mathit{no\text{-}loss}: &\quad \forall x \ .\ \mathit{reach}(r, x) \rightarrow \square\, \mathit{reach}(r, x) \\
\mathit{no\text{-}gain}: &\quad \forall x \ .\ x \neq n \ \land\ \neg \mathit{reach}(r, x) \rightarrow \square\, \neg \mathit{reach}(r, x) \\
\mathit{insertion}: &\quad (\forall u \ .\ \mathit{reach}(r, u) \rightarrow \mathit{data}[u] \neq \mathit{data}[n]) \rightarrow \square\, \mathit{at\_}13 \rightarrow \mathit{reach}(r, n)
\end{aligned}
$$

We begin by eliminating the universal quantifiers in the *no-loss* and *no-gain* properties by introducing a *skolem constant* x. This is done by augmenting the program with a *generic input constant* x – a variable with an undetermined initial value that stays constant throughout a computation. This is a purely syntactic transformation.

As for the *insertion* property, unfortunately the abstraction computation method of Section 7.2 disallows any occurrence of *reach* predicates under universal quantification. Therefore, we heuristically *instantiate* the universal variable u to derive the following (stronger) property:

$$\mathit{insertion}: \ \left(\bigwedge_{u\in\{r,n,t\}} \mathit{reach}(r,u) \rightarrow \mathit{data}[u] \neq \mathit{data}[n]\right) \rightarrow \Box\ \mathit{at\_}13 \rightarrow \mathit{reach}(r,n)$$

We proceed by applying the system transformation, resulting in the single-parent heap system[1] shown in Fig. 7.6. We now apply predicate abstraction. We use the predicate base given by the following set of assertions:

$$\mathcal{P}: \left\{\begin{array}{ll} p_1: & \forall j\ .\ \mathit{parent}[j] \neq n, \\ p_2: & \mathit{left\text{-}subtree}(n,r), \\ p_3: & \mathit{right\text{-}subtree}(n,r), \\ p_4: & \mathit{parent}^*(t,r), \\ p_5: & \exists j\ .\ \mathit{parent}[j] = t, \\ p_6: & \mathit{data}[t] = \mathit{data}[n], \\ p_7: & \mathit{parent}^*(x,r) \end{array}\right\}$$

Note that the predicate p_1 is in fact an inductive invariant, a fact that can be decided (without the use of abstraction) by directly applying Theorem 7.2

[1] Note that this automatically-derived version is less optimal than the manually-constructed single-parent system given in Fig. 7.2.

to check validity of the verification conditions

I1. $\Theta \rightarrow p_1$

I2. $p_1 \wedge \rho \rightarrow p'_1$

Having decided the invariance of p_1, it is possible to optimize the abstraction computation by removing p_1 from the predicate base, and by constraining the concrete state space to p_1-states only. ┛

In the following section we establish the soundness of the transformation.

7.4.2 Correctness of Transformation

In order for the above transformation to fit into the verification process proposed at the head of this chapter, we have to show that the result of the verification, as carried out on the transformed system and property, holds with respect to the untransformed counterparts. Such a result is provided by Theorem 7.13 below. To show that the abstraction computation method of Section 7.2 is sound with respect to a transformed program and property, we use Observation 7.5 and Theorem 7.14 below. For simplicity of presentation, in this section we do not take into account fairness requirements. However, it is straightforward to extend the results, i.e., show that the heap transformation preserves satisfiability of justice requirements, and that the computation transformation preserves compassion.

Let $\mathcal{D}_m : \langle \mathcal{V}_m, \Theta_m, \rho_m, \mathcal{J}_m, \mathcal{C}_s \rangle$ be a k-bounded multi-linked heap system over the set of variables $\mathcal{V}_m$, with $k > 1$, and let $\mathcal{D}_s : \langle \mathcal{V}_s, \Theta_s, \rho_s, \mathcal{J}_s, \mathcal{C}_s \rangle$ be its transformation into a single-parent heap system. The transformation into a single-parent heap system induces a mapping $\mathcal{S}\colon \Sigma_m \to \Sigma_s$. The mapping $\mathcal{S}$ is formally defined below.

Definition 7.6. *Let $\mathcal{S}$ be a mapping from Σ_m into Σ_s, such that for every $s_m \in \Sigma_m$, if $s_s = \mathcal{S}(s_m)$, then the following all hold:*

1. *For every* **bool** *variable $v \in \mathcal{V}_m$, $s_s[v] = s_m[v]$;*
2. *For every* **bool** *array $B \in \mathcal{V}_m$ and $x \in [0..h]$, $s_s[B](x) = s_m[B](x)$;*
3. *For every* **index** *variable $x \in \mathcal{V}_m$, $s_s[x] = s_m[x]$*
4. *$s_s[parent](0) = 0$ and $s_s[ct](0) = 1$.*
5. *Let $y \in [1..h]$. If for all $z \in [1..h]$ and $i \in [1..k]$, $s_m[link_i](z) \neq y$, then $s_s[parent](y) = 0$ and $s_s[ct](y) = 1$. Otherwise, $s_s[parent](y) = x$ and $s_s[ct](y) = j$ where (x, j) is the lexicographically minimal pair in $\{(z, i) : z \in [1..h],\ i \in [1..k],\ \text{and } s_m[link_i](z) = y\}$.*
6. $s_m[error] = \begin{cases} \textsc{False}, & \text{if } s_m \models no\_sharing \\ \textsc{True}, & \text{otherwise} \end{cases}$

┘

We first make the following observation regarding $\mathcal{S}$:

Observation 7.7. *The inverse $\mathcal{S}^{-1}$ is well defined for any well formed non-error state $s_s \in \Sigma_s$. That is, if $s_s \models$ `wf` $\wedge$ $\neg$error then there exists a state $s_m \in \Sigma_k$ such that $\mathcal{S}(s_m) = s_s$.*

Lemma 7.8. *Let $s_m \in \Sigma_m$, and let $s_s = \mathcal{S}(s_m)$. Then $s_m \models no\_sharing \iff s_s \models \texttt{wf} \ \wedge \ \neg error$.*

Proof. The reverse direction holds trivially. We now assume that $s_m \models no\_sharing$, and show that s_s satisfies `wf`, i.e.,

$$\begin{aligned}&\neg error \ \wedge \ parent[0] = 0 \ \wedge \ \textstyle\bigwedge_{B\in\mathcal{B}}(\neg B[0]) \ \wedge \\ &\forall i \neq j \,.\, (parent[i] = parent[j] \neq 0 \ \rightarrow \ ct[i] \neq ct[j])\end{aligned}$$

where $\mathcal{B} \subset \mathcal{V}_s$ is the set of **bool** arrays of $\mathcal{D}_s$. $s_s[error] = \text{FALSE}$, $s_s[parent](0) = 0$, and $s_s[B](0) = \text{FALSE}$, for all $B \in \mathcal{B}$, all follow from the definition of $\mathcal{S}$. The universal condition follows from two properties:

- The links in a multi-linked heap are functional, i.e., for every $i \in [1..k]$, every node has at most one $link_i$-child.
- From Item 5 of the definition of $\mathcal{S}$, we have that for any nodes u and v, and $i \in [1..k]$, we have $s_s[parent](u) = v$ and $s_s[ct](u) = i$ iff $s_m[link_i](v) = u$. □

□

Lemma 7.9. *Let $s_m \in \Sigma_m$ be a state that satisfies the no_sharing constraint, and let $s_s = \mathcal{S}(s_m)$. Let φ_m be a boolean combination of* M*-atomic formulae over $\mathcal{D}_m$, and let φ_s be its transformation into an assertion over $\mathcal{D}_s$. Then: $s_m \models \varphi_m \iff s_s \models \varphi_s$*

Proof. The claim follows immediately from Lemma 7.8 for the case that φ_m is an m-atomic non-*reach* and non-*cycle* formula. For the other cases, we distinguish between:

φ_m **is of the form** $reach(x_1, x_2)$. Then, φ_s is of the from

$$(x_1 \neq 0 \ \wedge \ x_2 \neq 0 \ \wedge \ parent^*(x_2, x_1)) \ \vee \ (x_2 = 0)$$

From the definition of $\mathcal{S}$ it follows that $s_s[x_1] = s_m[x_1]$ and $s_s[x_2] = s_m[x_2]$. In one direction, assume that $s_m \models \varphi_m$. If $s_m[x_2] = 0$, then obviously $s_s \models \varphi_s$. Otherwise, assume that $s_m[x_2] \neq 0$. Hence, for some $n \geq 1$ there exist nodes $s_m[x_1] = u_1, \dots, u_n = s_m[x_2]$ such that for every $i = 1, \dots, n$, there exists some $j_i \in [1..k]$ such that $s_m \models link_{j_i}[u_i] = u_{i+1}$, and $s_m[u_i] \neq 0$. Since $\mathcal{D}_m \models no\_sharing$, it follows that for every $i = 1, \dots, n-1$, $s_s[parent](u_{i+1}) = u_i$. Thus, $s_s \models parent^*(u_n, u_1)$. Thus $s_s \models \varphi_s$.

In the other direction, assume that $s_s \models \varphi_s$. If $s_s[x_1] = 0$, then $s_s[x_2] = 0$, and then $s_m \models \varphi_m$ trivially follows. Assume therefore that $s_s[x_1] \neq 0$. If $s_s[x_2] \neq 0$, an argument, similar to the one used for this case in the other direction, shows that $s_m \models \varphi_m$. If $s_s[x_2] = 0$, then let $u \neq 0$ be such that there is a $s_s[parent]$-path from u to $s_s[x_1]$, and for some $i \in [1..k]$, and for every y either $s_s[parent](y) \neq u$ or $M_k[ct](y) \neq i$. Thus, $s_m[link_i](u) \neq y$ for every y. It thus follows that $s_m[link_i](u) = 0$. Similar arguments to the previous direction show that there is a

$(\bigcup_{i=1}^{k} link_i)$-path from $s_m[x_1]$ to u. We can therefore conclude that $s_m \models reach^*(x_1, x_2)$.

φ_m **is of the form** $cycle(x)$. This case is similar to the previous case.

□

Since the initial condition of $\mathcal{D}_m$ is not a restricted A-assertion, it needs to be dealt with separately:

Lemma 7.10. *Let $s_m \in \Sigma_m$ such that $s_m \models no\_sharing$. Let $s_s = \mathcal{S}(s_m)$. Then: $s_m \models \Theta_m \iff s_s \models \Theta_s$*

Proof. As a consequence of the grammar in Table 7.1, Θ_m is of the form $\psi \wedge no\_sharing$ where ψ is a boolean combination of M-atomic formulae. Section 7.4 defines Θ_s as $\psi_s \wedge \texttt{wf}$, where ψ_s is the transformation of ψ by the rules of Section 7.4 and `wf` is given in Definition 7.4. From Lemma 7.9 we have that if $s_m \models no\_sharing$, then $s_m \models \psi$ iff $s_s \models \psi_s$. From Definition 7.6 we have $s_s \models \neg error$, and from Lemma 7.8 we have $s_m \models no\_sharing$ iff $s_s \models \neg error \wedge \texttt{wf}$. Thus $s_m \models \Theta_m$ iff $s_s \models \Theta_s$. □ □

We now extend Lemma 7.9 to show that transformation of the transition relation preserves the mapping $\mathcal{S}$:

Lemma 7.11. *Let $s_m \in \Sigma_m$ and $s_s = \mathcal{S}(s_m)$, such that $s_m \models no\_sharing$. Then for any state $s'_m \in \Sigma_m$, $\mathcal{S}(s'_m)$ is a ρ_s-successor of s_s if s'_m is a ρ_m-successor of s_m. Furthermore, if $s'_m \models no\_sharing$, then the reverse direction holds as well.*

Proof. Let $s'_m \in \Sigma_m$ be a state such that $s'_m \models \mathit{no\_sharing}$. Since ρ_m is a disjunction of clauses, Let $\varphi(\mathcal{V}_m) \wedge \tau(\mathcal{V}_m, \mathcal{V}'_m) \wedge \mathit{preserve}(\mathcal{V}_m, \mathcal{V}'_m)$ be one such arbitrary clause. Then the transformed clause is given by $\varphi_s(\mathcal{V}_s) \wedge \tau_s(\mathcal{V}_s, \mathcal{V}'_s)$, where $\varphi_s(\mathcal{V}_s)$ is the transformation of $\varphi(\mathcal{V}_m)$ and $\tau_s(\mathcal{V}_s, \mathcal{V}'_s)$ is the transformation of $\tau(\mathcal{V}_m, \mathcal{V}'_m)$ (recall that the preservation conjunct, present in the original clause, is discarded by the transformation, and that τ_s encapsulates variable preservation clauses).

From Lemma 7.9 and Lemma 7.10 we have $s_m \models \varphi(\mathcal{V}_m)$ iff $s_s \models \varphi_s(\mathcal{V}_s)$. Let $s'_s = \mathcal{S}(s'_m)$. It is left to show that $(s_m, s'_m) \models \tau(\mathcal{V}_m, \mathcal{V}'_m) \wedge \mathit{preserve}(\mathcal{V}_m, \mathcal{V}'_m)$ iff $(s_s, s'_s) \models \tau_s(\mathcal{V}_s, \mathcal{V}'_s)$. Since τ is a conjunction of `Assign` formulas, we show that for each type of atomic `Assign` formula $\psi(\mathcal{V}_m, \mathcal{V}'_m)$ and its transformation $\psi_s(\mathcal{V}_s, \mathcal{V}'_s)$, $(s_m, s'_m) \models \psi(\mathcal{V}_m, \mathcal{V}'_m) \implies (s_s, s'_s) \models \psi_s(\mathcal{V}_s, \mathcal{V}'_s)$, and if $s'_m \models \mathit{no\_sharing}$ then the reverse direction holds as well.

ψ has the form $x'_1 = t_2$ where t_2 is either an **index** variable or 0. In this case the claim holds trivially for both directions.

ψ has the form $B'[x]$ or $\neg B'[x]$, where B is a **bool** array and x is an **index** variable. In the case of $\neg B'[x]$, the claim follows trivially. In the case of $B'[x]$, ψ_s is the formula $(x = 0 \ \wedge \ \mathit{error}') \ \vee \ (x \neq 0 \ \wedge \ B'[x])$.

1. $s'_m \models \mathit{no\_sharing}$. Then $s'_m \models \neg B[0]$, and $s'_s \models \neg \mathit{error}$. If $(s_m, s'_m) \models B'[x]$, then x cannot be 0 in s_m, nor in s_s. From $\mathcal{S}$ we have $(s_s, s'_s) \models x \neq 0 \ \wedge \ B'[x]$. Otherwise, if $(s_s, s'_s) \models \psi_s$, then the claim follows from

the definition of $\mathcal{S}$ and the fact that *error* is FALSE in s'_s.

2. $s'_m \not\models$ *no_sharing*. Then $s'_s \models$ *error*. If $s_m[x] = 0$, then from the definition of $\mathcal{S}$ we have $(s_s, s'_s) \models x = 0 \ \wedge \ error'$. Thus $(s_m, s'_m) \models \psi \implies (s_s, s'_s) \models \psi_s$.

   Otherwise, $s_m[x] = s_s[x] \neq 0$. Since, by definition of $\mathcal{S}$, $s'_m[B](s_m[x]) = s'_s[B](s_s[x])$, then $(s_m, s'_m) \models x \neq 0 \wedge B'[x]$ iff $(s_s, s'_s) \models x \neq 0 \wedge B'[x]$.

ψ has the form $x'_1 = link_j[x_2]$. We focus on the nontrivial case that $s_m[x_2] \neq 0$ and $s'_m[x'_1] \neq 0$. First assume that x_2 is a leaf, i.e., $s_m[link_j](s_m[x_2]) = 0$. In this case $s'_m[x_1] = 0$, and by definition of $\mathcal{S}$, $s'_s[x_1] = 0$. From the assumption, we have $s_m \models link_j[x_1] = 0$. Then by Lemma 7.9, $s_s \models \forall y \ . \ (parent[y] \neq x_2 \ \vee \ ct[y] \neq j)$. Otherwise, assume that x_2 is not a leaf, i.e., $s_m[link_j](s_m[x_2]) \neq 0$. Then by definition of $\mathcal{S}$, there exists a node $u \neq 0$ such that $s'_m[x_1] = u$ and $s'_m[link_j](s_m[x_2]) = u$. Then by definition of $\mathcal{S}$, $s_s[parent](u) = s_s[x_2]$, $s_s[ct](u) = j$, and $s'_s[x_1] = u$. Thus $(s_s, s'_s) \models \exists y \ . \ (parent[y] = x_2 \ \wedge \ ct[y] = j \ \wedge \ x'_1 = y)$. In the reverse direction, if s_m and s'_m both satisfy the *no_sharing* constraint, then the claim follows trivially from the definition of $\mathcal{S}$.

ψ has the form $link'_j[x_1] = x_2$. Then ψ_s is the formula

$$\begin{array}{ll}
\mathit{Err} \wedge \mathit{error}' & (1) \\
\quad \vee & \\
\left[\begin{array}{ll}
 & \neg \mathit{Err} \\
\wedge & (x_2 = 0 \ \vee \ (x_2 \neq 0 \ \wedge \ \mathit{parent}'[x_2] = x_1 \ \wedge \ \mathit{ct}'[x_2] = j)) \\
\wedge & \left(\begin{array}{l}
\quad \forall z \ . \ (\mathit{parent}[z] \neq x_1 \ \vee \ \mathit{ct}[z] \neq j) \\
\vee \ \exists z \ . \ (\mathit{parent}[z] = x_1 \ \wedge \ \mathit{ct}[z] = j \ \wedge \\
\qquad (z = x_2 \vee \mathit{parent}'[z] = 0))
\end{array}\right)
\end{array}\right] & (2)
\end{array}$$

First assume $(s_m, s'_m) \models \psi$. Let $u_1 = s_m[x_1]$ and $u_2 = s_m[x_2]$. We consider two cases:

1. Node u_2 has multiple parents in s'_m, one of which must be u_1. In this case, we have $s'_m \not\models \mathit{no\_sharing}$. Furthermore, by definition of $\mathcal{S}$, we have $s'_s[\mathit{error}] = \text{True}$ and $s_s \models \mathit{Err}$. Thus $(s_s, s'_s) \models \psi_s$.

2. Node u_2 has a single parent in s'_m, which must be u_1. In this case it must be the case that $s_s \models \neg \mathit{Err}$. We now show that (s_s, s'_s) satisfies the other two conjuncts in disjunct (2) of ψ_s. The conjunct $(x_2 = 0 \vee (x_2 \neq 0 \ \wedge \ \mathit{parent}'[x_2] = x_1 \ \wedge \ \mathit{ct}'[x_2] = j))$ follows from the definition of $\mathcal{S}$. As for the third conjunct, consider first the case that u_1 has no j-child in s_m. Then by definition of $\mathcal{S}$, $s_s \models \forall z \ . \ \mathit{parent}[z] \neq x_1 \ \vee \ \mathit{ct}[z] \neq j$. Otherwise, there exists a node z that is the j-child of u_1 in s_m. If z is not u_2, then it is no longer the j-child of u_1 in s'_m. It follows from the definition of $\mathcal{S}$ that $(s_s, s'_s) \models \psi_s$.

It is left to show the reverse direction, under the assumption that $s'_m \models \mathit{no\_sharing}$. It follows that $s'_s[\mathit{error}] = \text{False}$. Thus, it must be the case that

(s_s, s'_s) satisfies disjunct (2) of ψ_s. Let $u_1 = s_s[x_1]$ and $u_2 = s_s[x_2]$. From the definition of $\mathcal{S}$ and the conjunct $(x_2 = 0 \ \vee \ (x_2 \neq 0 \ \wedge \ \mathit{parent}'[x_2] = x_1 \ \wedge \ \mathit{ct}'[x_2] = j))$ we conclude that if $u_2 \neq 0$, then u_2 is a j-child of u_1 in s'_m. If $u_2 = 0$, then from the third conjunct we conclude that u_1 has no child in s'_m. Therefore, $(s_m, s'_m) \models \psi$. □ □

Corollary 7.12. *Let $\mu : s_m^0, s_m^1, \ldots$ be a (finite or infinite) sequence of states that consists only of non-sharing states. Then μ is a run of $\mathcal{D}_m$ iff $\mathcal{S}(\mu)$: $\mathcal{S}(s_m^0), \mathcal{S}(s_m^1) \ldots$ is a run of $\mathcal{D}_s$ without error states.*

Proof. The proof is by induction on the run length. At the base case, from Lemma 7.10 we have that $\mathcal{S}(s_m^0) \models \Theta_s$ iff $s_m^0 \models \Theta_m$. Since Θ_m is defined to include the conjunct *no_sharing*, then s_m^0 satisfies the non-sharing constraint, and by definition of $\mathcal{S}$ we have $\mathcal{S}(s_m^0) \models \neg \mathit{error}$.

For the inductive step, let $s_m^0, \ldots, s_m^n$ be a run of $\mathcal{D}_m$ that is without sharing, and let $\mathcal{S}(s_m^0), \ldots, \mathcal{S}(s_m^n)$ be a run of $\mathcal{D}_s$ that is without error states. By Lemma 7.11 and the definition of $\mathcal{S}$, a $\mathcal{D}_m$-state s_m^{n+1} without sharing is a ρ_m-successor of s_m^n iff $\mathcal{S}(s_m^{n+1})$ is a ρ_s-successor of s_s such that $\mathcal{S}(s_m^{n+1})[\mathit{error}] = \textsc{False}$. □ □

From Lemma 7.9, Corollary 7.12, and Observation 7.7 we can now prove:

Theorem 7.13 (Soundness)**.** *Assume that for every reachable $\mathcal{D}_m$-state $s_m \in \Sigma_m$, $s \models$ no_sharing. Let φ_m be a temporal property over* M*-restricted A-assertions over $\mathcal{V}_m$, and let φ_s be φ_m, where every assertion over $\mathcal{V}_m$ is* *replaced with its transformation into a restricted EA-assertion over $\mathcal{V}_s$. Then:* $\mathcal{D}_s \models \varphi_s \iff \mathcal{D}_m \models \varphi_m$

While Theorem 7.13 shows that validity of temporal formulae carries from multi-linked systems into single-parent ones only when the former satisfy non-sharing, we prove that if the latter never reaches an error state, then the former never violates non-sharing:

Theorem 7.14 (Non-sharing)**.** *If $\mathcal{D}_s \models \Box \neg error$ then $\mathcal{D}_m \models \Box\, no\_sharing$.*

Proof. Assume that $\mathcal{D}_m$ has a computation with a prefix $s_m^0, \ldots, s_m^n$, where for any $0 \leq i < n$, $s_m^i \models no\_sharing$ and $s_m^n \not\models no\_sharing$. Following Corollary 7.12, the sequence $\mathcal{S}(s_m^0), \ldots, \mathcal{S}(s_m^{n-1})$ is an error-free run of $\mathcal{D}_s$. From Lemma 7.11, $\mathcal{S}(s_m^n)$ is a successor in $\mathcal{D}_s$ of $\mathcal{S}(s_m^{n-1})$. From the definition of $\mathcal{S}$ we have $\mathcal{S}(s_m^n) \models error$. □ □

Thus, to verify $\mathcal{D}_m \models \varphi_m$, one would initially perform a "sanity check" by verifying $\mathcal{D}_s \models \Box \neg error$. If this is successful, then the process outlined at the head of this chapter can be carried out. Theorem 7.13 guarantees not only that correctness of $\mathcal{D}_s$ implies correctness of $\mathcal{D}_m$, but also that a counterexample over $\mathcal{D}_s$ is mappable back into $\mathcal{D}_m$.

7.5 Examples of Verified Systems

We now describe two further examples on which the method has been tested, both of which are graph traversals. The first is a traversal algorithm that assumes no order between out-going edges of nodes. We demonstrate that, when the graph is a tree, edges can be reversed and the algorithm transformed so that it maintains its correctness. The second example is of a similar algorithm, with the restriction that out-going edges are ordered and the traversal should be done according to the "birth order" of nodes. These two examples demonstrate how we transform (1) algorithms that operate on non-sharing (unbounded) multi-linked heaps and (2) properties of the algorithms, into (1) translated algorithms operating on single-parent heaps and (2) translated properties so that a translated algorithm satisfies a translated property iff the original algorithm satisfies the original property.

7.5.1 Unordered Trees

Consider the algorithm of Fig. 7.7 that traverses a directed graph. Assume that the graph has no self-loops. Intuitively, the algorithm is similar to a strandard depth first search implementation. However, to "compensate" for its lack of stack, each edge is reversed when traversed, and since each edge is traversed twice, on termination the edges are all in their original orientation. To maintain this bookkeeping, each edge is associated with a counter that counts how many times it has been traversed (and reversed). This counter, called *visited*, is kept at the node into which the edge enters. This is possible since the assumption of no-sharing guarantees that every node has at most one incoming edge — a property that is preserved throughout the algorithm. When at a node x, the algorithm first searches for an untraversed edge, which would be identified by a node $k \in \mathit{Children}[x]$ such that $\mathit{visited}[k] = 0$. If no such edge is found, the algorithm searches for an edge that has been traversed once. If such a traversable edge is found in one of these two searches, its counter is incremented and the edge's direction is reversed. If no traversable edge is found, the algorithm terminates. In the algorithm of Fig. 7.7, the

```
$\ell_0$ :  $x := root$
$\ell_1$ :  **Repeat**
              $\ell_2$ :  $child := \epsilon\ k \in Children[x]\ .\ visited[k] = 0$
              $\ell_3$ :  **If** $child = 0$ **then**
                            $\ell_4$ :  $child := \epsilon\ k \in Children[x]\ .\ visited[k] = 1$
              $\ell_5$ :  **If** $child > 0$ **then**
                            $\ell_6$ :  $Children[x] := Children[x] - \{child\}$
                            $\ell_7$ :  $Children[child] := Children[child] \cup \{x\}$
                            $\ell_8$ :  $visited[x] := visited[child] + 1$
                            $\ell_9$ :  $x := child$
$\ell_{10}$ : **Until** $child = 0$
$\ell_{11}$ :
```

Figure 7.7: Algorithm TRAVERSAL-BY-REVERSAL

variable x is initialized to some designated node *root* (line ℓ_0). The search for a new *child* is at lines ℓ_2–ℓ_4. If found, *child* is removed from the list of x's children (line ℓ_6) and x is added to the list of *child*'s children (line ℓ_7). These two actions transform the edge $x \rightarrow \mathit{child}$ into an edge $\mathit{child} \rightarrow x$. At ℓ_8 the counter associated with this edge is incremented. This statement also relocates the site of this counter from its old site *child* to the new edge's tip at x. Finally x is set to *child* directing the next search to start at *child* (line ℓ_9).

Correctness of the algorithm is specified by the following:

$$\begin{array}{llll} \mathit{at\_}\ell_0 \ \wedge \ \mathit{reach}(\mathit{root}, x) \ \wedge \ x \neq \mathit{root} & \rightarrow & \Box(\mathit{at\_}\ell_{11} \ \rightarrow \ \mathit{visited}[x] > 0) & (S1) \\ \mathit{at\_}\ell_0 \ \wedge \ y \in \mathit{Children}[x] & \rightarrow & \Box(\mathit{at\_}\ell_{11} \ \rightarrow \ y \in \mathit{Children}[x]) & (S2) \\ \mathit{at\_}\ell_0 & \rightarrow & \Diamond(\mathit{at\_}\ell_{11}) & (L1) \end{array}$$

where x and y are generic nodes.

Property (S1) states that when the algorithm terminates, every node that is initially reachable from *root* is visited; (S2) states that when the algorithm terminates, each edge is in its initial orientation; (L1) states that the algorithm eventually terminates.

Suppose we wish to apply the algorithm to trees with unbounded arity. By reversing the edges directions, such trees will be transformed into an single-parent structures, letting each child point to its parent. Since edges are chosen non-deterministically, without assuming any order, the algorithm fits well into our framework.

In the transformed representation, instead of having the set *Children*$[x]$ pointing from node x to its descendants, each descendant i points, by *link*$[i]$, to its parent x.

The transformed algorithm is described in Fig. 7.8.

The correctness of the transformed algorithm can described by the fol-

```
$\ell_0$ :  $x := root$
$\ell_1$ :  **Repeat**
            $\ell_2$ :  $child := \epsilon\ k\ .\ link[k] = x\ \wedge\ visited[k] = 0$
            $\ell_3$ :  **If** $child = 0$ **then**
                  $\ell_4$ :  $child := \epsilon\ k\ .\ link[k] = x\ \wedge\ visited[k] = 1$
            $\ell_5$ :  **If** $child > 0$ then
                  $\ell_6$ :  $link[child] := 0$
                  $\ell_7$ :  $link[x] := child$
                  $\ell_8$ :  $visited[x] := visited[child] + 1$
                  $\ell_9$ :  $x := child$
$\ell_{10}$ : **Until** $child = 0$
$\ell_{11}$ :
```

Figure 7.8: Algorithm Traversal-by-Reversal for the single-parent heap representation

lowing:

$$at\_\ell_0 \wedge reach(root, x) \wedge x \neq root \quad \rightarrow \quad \Box(at\_\ell_{11} \ \rightarrow\ visited[x] > 0) \qquad (S1)$$

$$at\_\ell_0 \wedge link[y] = x \quad \rightarrow \quad \Box(at\_\ell_{11} \ \rightarrow\ link[y] = x) \qquad (S2)$$

$$at\_\ell_0 \quad \rightarrow \quad \Diamond(at\_\ell_{11}) \qquad (L1)$$

7.5.2 Ordered Trees

The transformation in the previous section was enabled due to the fact that the trees (or there, general singly-linked structures) have unordered children, and each node is accessed by the *choose* operator, regardless of its "birth order."

This is, however, not always the case. Consider a variant of the previous algorithm, presented in Fig. 7.9, where nodes are accessed according to their order.

```
$\ell_0$ :  $(x, done, Children[root]) := (root, 0, Children[root] * (root))$
$\ell_1$ :  Repeat
              $\ell_2$ :  $(child, Children[x]) := (hd(Children[x]), tl(Children[x]))$
              $\ell_3$ :  If $child = x$ then
                            $\ell_4$ :  $done := 1$
                          else
                            $\ell_5$ :  $Children[child] := Children[child] * (x)$
                            $\ell_6$ :  $x := child$
$\ell_7$ :  Until $done$=1
$\ell_8$ :
```

Figure 7.9: Algorithm Ordered-Traversal

In this presentation, each node x is associated with $Children[x]$ which is a *list* (rather than a *set*) of descendants. In statement ℓ_0, x is initialized to the start node *root*, and node *root* is appended to the end of its own children list. We use the fact that in a legitimate graph no node is a child of itself, and this special self-loop is created in order to detect termination of the search. Statement ℓ_2 removes the first child from the children list of x and places it in *child*. If *child* equals x then the search should terminate, and we set *done* to 1. Otherwise x is appended to the end of the children list of node *child* at statement ℓ_5. This completes the reversal of the edge $x \rightarrow child$. Finally, x is set to *child* to indicate that this is where the search will continue. The algorithm terminates when a node is encountered that is its own first child, which happens only on coming back to node *root* after having visited all nodes that are reachable from *root*.

We encode the unbounded arity tree as a binary tree, as suggested by

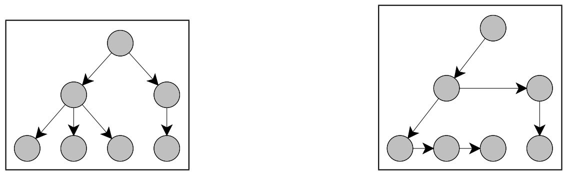


Figure 7.10: An Ordered Unbounded Tree and its Binary Encoding

Knuth [Knu69], and reverse the orientation of links so as to have a single-parent heap. Formally, consider a (directed) tree, where each node has unboundedly many ordered children. For simplicity of exposition, assume that if a node has a $(k+1)^{st}$ child, then it also has a k^{th} child. We add a new **bool** array *type*, whose range is $\{p, s\}$, where the value p denotes that the link outgoing the node is a parent link, and s that the link outgoing the node is a sibling link. If, in the original tree, a node x has children $y_1, \ldots, y_k$, then the transformed model includes the following: $\mathit{type}[y_1] = p$ and $\mathit{link}[y_1] = x$, denoting that y_1 has a parent link leading into x. For $i = 2, \ldots, k$, $\mathit{type}[y_i] = s$ and $\mathit{link}[y_i] = y_{i-1}$, denoting that y_i has a sibling link leading into y_{i-1}. Fig. 7.10 shows an example of the transformation, with which we can now treat ordered unbounded trees as binary trees and apply the methods we apply to single-parent heaps to them. For example, the algorithm of Fig. 7.9 now becomes the algorithm in Fig. 7.11, with properties similar to (S1), (S2), and (L1). The algorithm of Fig. 7.11 uses the procedure *append_child*(x, *child*), presented in Fig. 7.12, which appends node *child* to the end of the children list of node x.

The algorithm of Fig. 7.11 and procedure *append_child* both use the func-

```
$\ell_0$ :  $(x, done) := (root, 0)$;  $append\_child(root, root)$
$\ell_1$ :  Repeat
              $\ell_2$ :  $child := down\_link(x, p)$
              $\ell_3$ :  $sister := down\_link(child, s)$
              $\ell_4$ :  If $sister > 0$ then
                            $\ell_5$ :  $(link[sister], type[sister]) := (x, p)$
              $\ell_6$ :  If $child = x$ then
                            $\ell_7$ :  $done := 1$
                          else
                            $\ell_8$ :  $append\_child(child, x)$
                            $\ell_9$ :  $x := child$
$\ell_{10}$ : Until $done = 1$
$\ell_{11}$ :
```

Figure 7.11: Algorithm ORDERED-TRAVERSAL over a single-parent heap representation

tion $down\_link(n, t)$, for node n and value $t \in \{p, s\}$, as an abbreviation for

$$down\_link(n, t) = \textbf{choose } i.\ link[i] = n \ \wedge\ type[i] = t$$

$down\_link(n, t)$ returns 0 if there exists no node i such that $link[i] = n \ \wedge$ $type[i] = t$.

7.6 Composite Data Structures

Up to this point, it has been assumed that the data associated with heap nodes is of some constant (non-parameterized) domain. This precludes the modeling of structures that are embedded as data referenced by "enclosing" structures, for example a list of lists. This section advances the methodology

```
procedure $append\_child(x, child)$
  $m_0$: $daughter := down\_link(x, p)$
  $m_1$: If $daughter = 0$ then
           $m_2$: $(link[child], type[child]) := (x, p)$
         else
           $m_3$: $next := daughter$
           $m_4$: While $next > 0$ do
                    $m_5$: $last := next$
                    $m_6$: $next := down\_link(last, s)$
           $m_7$: $(link[child], type[child]) := (last, s)$
```

Figure 7.12: Procedure *append_child*

by accommodating multi-heap structures that refer to one another. The multi-heap structures have a single restriction – the underlying graph of the heap structure contains no non-trivial cycles. Such a structure is referred to as *cascading*. Each of the heaps my be single-parent or non-sharing multi-linked. A typical such structure is a multi-linked "main" heap and a stack or a queue (or both) pointing to main heap elements.

The extension of the structure allows to model structures such as $B+$ trees, lists of lists, etc. Consequently, we can automatically verify algorithms such as DFS, BFS, insertions and deletion in $B+$ trees, and more. This is achieved by expressing these structures as cascading heaps. For example, the verification of DFS uses a predicate that roughly says "every node on the stack points to a heap node that is not reachable from node r."

7.6.1 Cascading Heap Systems: An Example

To illustrate the model, we encode a depth first traversal on single-parent (main) heaps using a stack. We show how a stack can be encoded as a single-parent heap where stack nodes point into the main heap nodes, thus resulting in a cascading heap structure.

Consider a stack of at most n_2 "stack nodes," and a main heap with at most n_1 "main nodes." A stack node j may point, by $\mathit{StackNext}[j]$, to a stack node, and by $\mathit{StackData}[j]$ to a main node. Both pointers are initially 0. In addition, stack node j has a boolean flag $\mathit{allocated}[j]$ that is initially FALSE, denoting that it is not allocated. We also assume a stack node variable *stack* pointing to the stack head, which is initially 0. The implementation of stack *push* and *pop* operations is given in Fig. 7.13. For a stack *push* one must first ensure that there is a free stack node and, if so, update *stack* to point to it. With this data definition, it is possible to encode a depth first traversal over a non-sharing multi-linked main heap, as given in Fig. 7.13(a). The cascading single-parent encoding of the algorithm is given in Fig. 7.14.

The specification of DEPTH-FIRST-TRAVERSAL is given by

$$(\mathit{reach}(x,t) \ \wedge\ \Box\,\neg \mathit{error}) \rightarrow \Box(\mathit{at}\_\ell_9 \rightarrow \mathit{visited}[t])$$

where t refers to a *generic node* (modeled by an auxiliary variable). To verify that DEPTH-FIRST-TRAVERSAL meets its specification, we have used the following predicate base:

$$\left\{\begin{array}{l}(x=0),\ \mathit{parent}^*(x,0),\ (t=x),\ \mathit{parent}^*(t,x),\\ \forall i\ .\ (\mathit{StackNext}^*(\mathit{stack},i)\rightarrow \mathit{parent}^*(\mathit{StackData}[i],0)),\\ \exists i\ .\ (\mathit{StackNext}^*(\mathit{stack},i)\wedge \mathit{parent}^*(t,\mathit{StackData}[i]))\\ \mathit{StackNext}^*(\mathit{stack},0),\ (\mathit{StackData}[\mathit{stack}]=x)\end{array}\right\}$$

where t refers to a *generic node* (modeled by an auxiliary variable).

7.6.2 Data Types

The cascading heap model results from an extension to the type system that allows for multiple parameterized types, coupled with an extension to the assertional language. We first present the type system.

Let $n > 0$, and assume $n+1$ types, $\mathit{type}_0, \ldots, \mathit{type}_n$, where type_0 is the boolean type (or any finite type), and for each $i > 0$, type_i ranges over $[0_i..h_i^{m_i}]$. The n-tuple $(m_1, \ldots, m_n)$ is called the *parameter* of the system. A *cascading heap system* consists of a family n of single-parent heaps, say $H_1, \ldots, H_n$, where for every $i > 0$, heap H_i's nodes can point to a node of any H_j, for $j \leq i$, as well as have some type_0 data. Formally, a cascading heap system may have the following types of variables:

- For every i, $0 \leq i \leq n$, type_i scalar variables;
- For every i and j, $0 < j \leq i \leq n$, arrays of type $\mathit{type}_i \mapsto \mathit{type}_j$;
- For every i, $0 < i \leq n$, arrays of type $\mathit{type}_i \mapsto \mathit{type}_0$.

For simplicity of exposition we assume that $n = 2$. The definitions and results can be easily extended to general case. We refer to $type_0$ as **bool**, to $type_1$ as m- (for "main-heap"), and to $type_2$ as a- (for "auxiliary-heap"). An example of a cascading heap system is the system of Subsection 7.6.1, where the "stack nodes" there are a-nodes here. The examples of previous sections could all be categorized as 1-heap systems, consisting of $type_0$ (**bool**) and $type_1$ (**index**) scalars, as well as $type_1 \mapsto type_0$ and $type_1 \mapsto type_1$ arrays.

7.6.3 Assertional Language

For every $i \in \{1, 2\}$, a *$type_i$ term* is the constant 0_i, a $type_i$ variable, or $Z[t]$, where Z is a $type_i \mapsto type_i$ array and t is a $type_i$ term.

The language of restricted EA-assertions of the previous section is extended as follows, for $i \in \{1, 2\}$ and a $type_2 \mapsto type_1$ array C:

- Unquantified atomic assertions and preservation assertions may freely (though in a type-consistent manner) contain variables of the extended type system;

- Z-assertions are allowed that have the form $\forall y \,.\, (Z[C[y]]) \neq u \;\vee\; \mathbb{B}(C[y])$, where u is a $type_1$ term, Z is a $type_2 \mapsto type_1$ array.

- TCF-assertions have the extended form $\forall \vec{y} \,.\, P(\vec{u}, \vec{y})$ where $\vec{u}$ and $\vec{y}$ are disjoint sets of $type_1$ and $type_2$ variables, and for a pair of $type_1 \mapsto type_1$ and $type_2 \mapsto type_2$ arrays Z_1 and Z_2, $P(\vec{u}, \vec{y})$ is a positive combination of formulae of the form $\neg Z_i^*(u, f(y))$ or $\mathbb{B}(y))$ where $i \in \{1, 2\}$, u is a $type_i$ variable, $f(y)$ is either the term y or $C[y]$, and $\mathbb{B}$ is a boolean combination of formulae of the form $B_k[y]$ where B_k is a **bool** array.

7.6.4 Cascading Heap Systems: A Small Model Property

As in Section 7.2, in order to use cascading heap systems within our abstraction framework, we must show a small model property for the new assertional language. To construct a small model for a cascading heap systems, we reduce each individual heap as suggested in Section 7.2, and restrict to dealing with the "connecting" arrays, i.e., the arrays that connect the heaps to one another. As before, we restrict discussion to the 2-heap system case, and assume a single array, C, that connects them. That is, $C\colon type_2 \mapsto type_1$ and $\mathcal{V}$ consists of $\mathcal{V}_1$ (the variables that belong exclusively to heap 1), $\mathcal{V}_2$, as well as C and C'.

Let φ be a restricted A-assertion over $\mathcal{V}$. Define the set of terms in $\mathcal{T}_\varphi^1$ and $\mathcal{T}_\varphi^2$ as in Section 7.2. To form $\mathcal{T}_\varphi$, we add to the above sets the minimal set of terms such that for every free term t, if $C[t]$ (resp. $C'[t]$) is in $\mathcal{T}_\varphi$, then $parent[t]$ (resp. $parent'[t]$) is in $\mathcal{T}_\varphi$.

For every $i = 1, 2$, define $f_i(\varphi) = |\mathcal{T}_\varphi^i| + \max(1, |\mathcal{R}_\varphi^i|)$. Let M be a model of φ of size greater than $(f_1(\varphi), f_2(\varphi))$. We reduce M to a model $\overline{M}$ of size no greater than $(f_1(\varphi), f_2(\varphi))$ by reducing each heap according to the reduction of Definition 7.1. As for the arrays C and C', we use the auxiliary functions

$\zeta, \zeta' \colon \mathcal{N}_2 \cup \mathcal{S}_2 \to [0..M[h_1]]$ as follows:

$$\zeta(n) = \begin{cases} M[C](r^Z(n)), & \text{If } n \text{ has a } Z\text{-representative} \\ M[C](n), & \text{If } n \in \mathcal{S}_2 \text{ or } n \text{ has no } Z\text{-representative} \end{cases}$$

and ζ' is defined like ζ, with every symbol C substituted by C', and Z substituted by Z'. Let $\mathcal{M}$ be the set of new *type*$_1$-terms resulting from applying ζ and ζ' to $\mathcal{N}_2$ and $\mathcal{S}_2$. I.e.,

$$\mathcal{M} = (\zeta(\mathcal{N}_2 \cup \mathcal{S}_2) \cup \zeta'(\mathcal{N}_2 \cup \mathcal{S}_2)) - (\mathcal{N}_1 \cup \mathcal{S}_1)$$

and assume $\mathcal{M} = \{m_{\beta_1+1}, \ldots, m_\delta\}$. We define mappings $\Gamma_1 : \mathcal{N}_1 \cup \mathcal{S}_1 \cup \mathcal{M} \mapsto [0..\delta]$ and $\Gamma_2 : \mathcal{N}_2 \cup \mathcal{S}_2 \mapsto [0..\beta_2]$ in the obvious way.

The reduced model $\overline{M}$ is constructed as follows:

- *type*$_1$, *type*$_2$, and **bool** variables, and *type*$_2 \mapsto$ *type*$_2$ and *type*$_2 \mapsto$ **bool** arrays are interpreted as in Definition 7.1;
- For any $j \in [0_2..\beta_2]$, $\overline{M}[C](j) = \Gamma_1(\zeta(a_j))$, and $\overline{M}[C'](j) = \Gamma_1(\zeta'(a_j))$;
- For any $j \in [0_1..\alpha_1] \cup [\beta_1+1..\delta]$ and unprimed *type*$_1 \mapsto$ *type*$_1$ array Z and *type*$_1 \mapsto$ **bool** array B, $\overline{M}[Z](j) = \Gamma_1(M[Z](r^Z(m_j)))$ and $\overline{M}[B](j) = M[B](r^Z(m_j))$, if a Z-representative exists, or $\overline{M}[Z](j) = \Gamma_1(e^Z(m_j))$ and $\overline{M}[B](j) = M[B](e^Z(m_j))$ otherwise; For *primed* array Z' and B' the reduction is similarly defined.
- For any $j \in [\alpha_1 + 1..\beta_1]$, and *type*$_1 \mapsto$ *type*$_1$ array Z and *type*$_1 \mapsto$ **bool**

array B, $\overline{M}[B](j)$ is defined, as in Definition 7.1, to be $M[B](m_j)$. Similarly, $\overline{M}[Z](j)$ is defined to be $\Gamma(M[Z]^{\ell}(m_j))$ for the minimal $\ell \geq 0$ such that $M[Z]^{\ell}(m_j) \in \mathcal{S}_1$.

Theorem 7.15 (Cascading Small Model Theorem)**.** *Let φ be a restricted EA-assertion. Then φ is satisfiable iff it is satisfiable by a model of size no greater than $(f_1(\varphi), f_2(\varphi))$.*

In addition to Properties **P0**–**P3** for each heap, we also have:

Observation 7.16. *The following properties are valid:*

Q0. *For a type$_2$ node a_i such that $a_i \in \mathcal{S}_2$ or a_i has no Z- (resp. Z'-) representative, $M[C](a_i)$ (resp. $M[C'](a_i)$) is in $\mathcal{N}_1 \cup \mathcal{S}_1 \cup \mathcal{M}$, and $\Gamma_1(M[C](a_i)) = \overline{M}[C](i)$ (resp. $\Gamma_1(M[C'](a_i)) = \overline{M}[C'](i)$);*

Q1. *For every i and j that are* both *in $[0..\alpha_1] \cup [(\beta_1+1)..\delta]$, for every type$_1 \mapsto$ type$_1$ array Z, $M[Z](n_i) = n_j \implies \overline{M}[Z](i) = j$ and $\overline{M}[Z](i) = j \implies M[Z]^*(n_i, n_j)$.*

Q2. *For every i and j that are* both *in $[0_1..\alpha_1] \cup [(\beta_1+1)..\delta]$ or* both *in $[(\alpha_1+1)..\beta_1]$, for every type$_1 \mapsto$ type$_1$ array Z, $M[Z]^*(n_i, n_j) \iff \overline{M}[Z]^*(i, j)$.*

We can now prove Theorem 7.15.

Proof. As in the proof of Theorem 6.8 we proceed by induction on the structure of φ. Here we deal with Z– and TCF-assertions, since the proof of Theorem 7.2 holds for other types of subformulae.

Assume that ψ is a Z-assertion of the form $\forall y \,.\, Z[C[y]] \neq u \;\vee\; \mathbb{B}(C[y])$, for which we show that, under the assumption that $M \models \psi$, $\overline{M} \models \psi$ Since u is a free term, it follows that $M[u] \in \mathcal{N}$, hence, $M[u] = n_i$ for some $i \leq \alpha_1$. Assume, by way of contradiction, that $\overline{M} \not\models \psi$. That is, for some $j \in [0_2..\beta_2]$, $\overline{M}[Z](\overline{M}[C](j)) = i$ and $\overline{M}[\mathbb{B}](\overline{M}[C](j)) = \textsc{False}$. Since $i \leq \alpha_1$, then we have $\ell = \overline{M}[C](j) \in [0..\alpha_1] \cup [(\beta_1 + 1)..\delta]$. Thus, n_ℓ has a Z-representative. However, since $M \models \psi$, and since $M[Z](r^Z_M(n_\ell)) = M[u]$, it follows that $M[\mathbb{B}](r^Z_M(n_\ell)) = \textsc{True}$. From the construction it now follows that $\overline{M}[\mathbb{B}](\overline{M}[C](j)) = \textsc{True}$, contradicting the assumption that $\overline{M} \not\models \psi$.

If ψ is a Z-assertion of the form $\forall y \,.\, Z[Y] \neq u \;\vee\; \mathbb{B}(y)$ where u is a *type*$_1$-variable, then the claim follows, in addition to the model construction, from property **Q1**. For other forms of Z-assertion, the proof of Theorem 7.2 holds in the new model reduction.

Assume now that ψ is a tcf-assertion of the form $\forall \vec{y} \,.\, P(\vec{u}, \vec{y})$ over the *type*$_1 \mapsto$ *type*$_1$ and *type*$_2 \mapsto$ *type*$_2$ arrays Z_1 and Z_2, respectively. Generalizing the technique in the proof of Theorem 6.8, we consider a pair of arbitrary consistent assignments $\overline{\eta}_1$ and $\overline{\eta}_2$ to *type*$_1$ and *type*$_2$ $\vec{y}$-variables, that respectively assign *type*$_1$ values in the range $[0_1..\delta]$ and *type*$_2$ values in the range $[0_2..\beta_2]$. For convenience, we map $\overline{\eta}_1$ to the assignment η_1, which ranges over $\mathcal{N}_1 \cup \mathcal{S}_1 \cup \mathcal{M}$, and η_2, which ranges over $\mathcal{N}_2 \cup \mathcal{S}_2$, as follows:

$$\eta_2(y) = \begin{cases} r^{Z_2}(\Gamma_2^{-1}(\overline{\eta}(y))), & \text{If } \overline{\eta}(y) \leq \alpha_2 \text{ and } \Gamma_2^{-1}(\overline{\eta}(y)) \text{ has a } Z_2\text{-representative} \\ \Gamma_2^{-1}(\overline{\eta}(y)), & \text{otherwise} \end{cases}$$

$$\eta_1(y) = \begin{cases} r^{Z_1}(\Gamma_1^{-1}(\overline{\eta}(y))), & \text{If } \overline{\eta}(y) \leq \alpha_1 \ \vee \ \beta_1 < \overline{\eta}(y) \text{ and} \\ & \Gamma_1^{-1}(\overline{\eta}(y)) \text{ has a } Z_1\text{-representative} \\ \Gamma_1^{-1}(\overline{\eta}(y)), & \text{otherwise} \end{cases}$$

We define the augmented models $\overline{M}_\eta$ and M_η to be $(\overline{M}, \overline{\eta})$ and (M, η_1, η_2), respectively. Consider a clause ϕ of P. Assuming that $M_\eta \models \phi$, we need to show that $\overline{M}_\eta \models \phi$.

If ϕ is the formula $\mathbb{B}(y)$, then by the same reasoning as in the single-heap case, it follows that $\overline{M}_\eta \models \phi$, as is true if ϕ is either of the formulae $\neg Z_2^*(u, y)$ and $\neg Z_1^*(u, y)$. Thus we focus on the case that ϕ is $\neg Z_1^*(u, X[y])$, implying that y is a *type*$_2$-variable. Let $i_y = \overline{\eta}[y]$ and $j_y = \overline{M}[X](i_y)$. If either $i_y > \alpha_2$ or a_{i_y} has no Z_2-representative, then from property **Q0** it follows that $\Gamma_1(M_\eta[X[y]]) = \Gamma_1(M[X](a_{i_y})) = \overline{M}_\eta[X[y]] = j_y$. Since $M_\eta[u]$ is an $\mathcal{N}_1$-node, then it follows from property **Q2** that $M_\eta \models Z_1^*(u, X[y])$ iff $\overline{M}_\eta \models Z_1^*(u, X[y])$. $\square$

type T_2 :	$[0..H_2]$	**type** T_1 :	$[0..H_1]$
stack :	T_2	x :	T_1
$\mathit{StackNext}$:	$T_2 \mapsto T_2$	$\mathit{visited}$:	$T_1 \mapsto \mathbf{bool}$
$\mathit{allocated}$:	$T_2 \mapsto \mathbf{bool}$	$\mathit{children}, \mathit{next}$:	$T_1 \mapsto T_1$
$\mathit{StackData}$:	$T_2 \mapsto T_1$	error :	$\mathbf{bool}$

```
init  $\neg error \wedge (\forall i : T_1 . \neg visited[i]) \wedge$
      $(\forall j : T_2 . \neg allocated[j])$
$\ell_0$ : $stack := 0$
$\ell_1$ : PUSH$(x, stack)$
$\ell_2$ : while $stack \neq 0$
    $\ell_3$ : POP$(x, stack)$
    $\ell_4$ : $visited[x] :=$ TRUE
    $\ell_5$ : $x := children(x)$
    $\ell_6$ : while $x \neq 0$
        $\ell_7$ : PUSH$(x, stack)$
        $\ell_8$ : $x := next(x)$
$\ell_9$ :
```

(a) DEPTH-FIRST-TRAVERSAL

```
POP$(x, stack)$ ::
    $m_1$ : $x := StackData[stack]$
    $m_2$ : $stack := StackNext[stack]$
```

(b) Stack POP operation

```
PUSH$(x, stack)$ ::
    Let $j =$ choose $u . \neg allocated[u]$
    in
        $m_1$ : If $j \neq 0$ then
            $m_2$ : $allocated[j] :=$ TRUE
            $m_3$ : $StackNext[j] := stack$
            $m_4$ : $StackData[j] := x$
            $m_5$ : $stack := j$
        $m_6$ : Else $error :=$ TRUE
```

(c) Stack PUSH operation

Figure 7.13: A Cascading Heap System

type T_2 :	$[0..H_2]$	**type** T_1 :	$[0..H_1]$
$stack$:	T_2	x :	T_1
$StackNext$:	$T_2 \mapsto T_2$	$visited$:	$T_1 \mapsto$ **bool**
$allocated$:	$T_2 \mapsto$ **bool**	$parent$:	$T_1 \mapsto T_1$
$StackData$:	$T_2 \mapsto T_1$	$type$:	$T_1 \mapsto \{children, next\}$
$error$:	**bool**		

```
init  $\neg error \;\wedge\; (\forall i : T_1 \,.\, \neg visited[i]) \;\wedge\; (\forall j : T_2 \,.\, \neg allocated[j]) \;\wedge$
      $(\forall i, j : T_1 \,.\, \neg(i \neq j \;\wedge\; parent[i] = parent[j] \;\wedge\; type[i] = type[j]))$

$\ell_0$ : $stack := 0$
$\ell_1$ : PUSH$(x, stack)$
$\ell_2$ : while $stack \neq 0$
      $\ell_3$ : POP$(x, stack)$
      $\ell_4$ : $visited[x] :=$ TRUE
      $\ell_5$ : $x :=$ choose $j \,.\, parent[j] = x \;\wedge\; type[j] = c$
      $\ell_6$ : while $x \neq 0$
            $\ell_7$ : PUSH$(x, stack)$
            $\ell_8$ : $x :=$ choose $j \,.\, parent[j] = x \;\wedge\; type[j] = s$
$\ell_9$ :
```

Figure 7.14: Algorithm Depth-First-Traversal for the Cascading single-parent Representation

Chapter 8

Conclusion

This dissertation opens with a presentation of *ranking abstraction*, a method that combines predicate abstraction, program augmentation, and model-checking, with the purpose of verifying safety and liveness properties of sequential and concurrent systems. It extends the method of *abstraction refinement* to the domain of ranking functions, allowing for an automatic counterexample-driven process that incrementally eliminates not only spurious *traces*, but spurious *cycles* as well. Having defined and demonstrated the method, it is then shown that in most cases, following a successful verification effort, the model checker can be utilized to extract a proof that the system meets its specification.

While the ranking abstraction method is domain-neutral, this dissertation focuses on its application to *shape analysis* problems, i.e., verification of programs that requires modeling of deep heap properties. To this end an automatic abstraction computation method is shown that depends only on a symbolic model checker, or alternately a SAT solver. While numerous shape analyses have been proposed in the past, this method is novel in its ability to deal with arbitrary liveness properties.

Finally, the limitations of the abstraction method are transcended by a combination of *structure simulation* as well as composition of separate heaps into *cascading* structures. The former allows for representation of sharing-free structures (e.g., trees), and the latter allows to model composite data structures as in the case where an "auxiliary" structure contains pointers into a "main" structure.

Our abstraction computation method relies on decidability results, in the form of small model properties, for a family of restricted yet sufficiently powerful combinations of first order logic with transitive closure. Two such decidable logics are presented: One for expressing mutation and abstraction of singly-linked structures, and another for multi-linked but sharing-free structures.

Future Work

The proof extraction framework discussed above is limited in that it does not deal with systems with native compassion requirements (i.e., compassion requirements beyond what is contributed by the ranking augmentation). Thus a natural direction for future extension of this method is to allow for systems with native compassion.

In the context of abstraction of heap systems, we would like to explore optimizations to our current (BDD-based) method, based on SAT or SMT (e.g., [BB04]) solvers. Long term, we wish to expand as much as possible the class of structures that can be modeled, which requires investigation of richer transitive closure logics with small model properties. Finally, predicate abstraction as defined here is limited in the shape of the invariants it can express. We would like to explore richer abstractions in order to generate quantified invariants. To this end we would combine the notion of user-provided predicates, with invariant generation methods for parameterized systems [APR$^+$01], e.g., as in [LB04].

Bibliography

[APR$^+$01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *Proc. 13rd Intl. Conference on Computer Aided Verification*, volume 2102 of *Lect. Notes in Comp. Sci.*, pages 221–234. Springer-Verlag, 2001.

[BB04] Clark W. Barrett and Sergey Berezin. Cvc lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron Peled, editors, *Proc. 16th Intl. Conference on Computer Aided Verification*, Lect. Notes in Comp. Sci., pages 515–518. Springer-Verlag, 2004.

[BBH$^+$06] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomás Vojnar. Programs with lists are counter automata. In Thomas Ball and Robert B. Jones, editors, *Proc. 18th Intl. Conference on Computer Aided Verification*, volume

4144 of *Lect. Notes in Comp. Sci.*, pages 517–531. Springer-Verlag, 2006.

[BCDO06] Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In Thomas Ball and Robert B. Jones, editors, *Proc.* 18^{th} *Intl. Conference on Computer Aided Verification*, volume 4144 of *Lect. Notes in Comp. Sci.*, pages 386–400. Springer-Verlag, 2006.

[BCG$^+$07] Maurice Bruynooghe, Michael Codish, John P. Gallagher, Samir Genaim, and Wim Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Trans. Program. Lang. Syst.*, 29(2):10, 2007.

[BGG97] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem.* Perspectives of Mathematical Logic. Springer-Verlag, 1997. Second printing (Universitext) 2001.

[BPR02] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 158–172, 2002.

[BPZ05] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis by predicate abstraction. In *Proc. of the* 6^{th} *Int. Conference on*

Verification, Model Checking, and Abstract Interpretation, pages 164–180, 2005.

[BPZ07a] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Modular ranking abstraction. *Int. J. Found. Comput. Sci.*, 18(1):5–44, 2007.

[BPZ07b] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis of single-parent heaps. In *Proc. of the* 8th *Int. Conference on Verification, Model Checking, and Abstract Interpretation*, Lect. Notes in Comp. Sci. Springer-Verlag, 2007.

[BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057:103+, 2001.

[BRS99] Michael Benedikt, Thomas W. Reps, and Shmuel Sagiv. A decidable logic for describing linked data structures. In *European Symposium on Programming*, pages 2–19, 1999.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *Lect. Notes in Comp. Sci.*, pages 52–71. Springer-Verlag, 1981.

[CGJ$^+$00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[CGP$^+$07] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Vardi. Proving that programs eventually do something good. In Martin Hofmann and Matthias Felleisen, editors, *Proc.* 34th *ACM Symp. Princ. of Prog. Lang.* ACM Press, 2007.

[CPR05] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *Static Analysis Symposium*, pages 87–101, 2005.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In Michael I. Schwartzbach and Thomas Ball, editors, *PLDI*, pages 415–426. ACM, 2006.

[CT99] Michael Codish and Cohavit Taboch. A semantic basis for the termination analysis of logic programs. *J. Log. Program.*, 41(1):103–123, 1999.

[DD01] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the* 16th *Annual IEEE Symposium on Logic in Computer Science*, page 51. IEEE Computer Society, 2001.

[DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *Proceedings of the 11th Interna-*

tional Conference on Computer Aided Verification, Lect. Notes in Comp. Sci., pages 160–171. Springer-Verlag, 1999.

[DGG00] Dennis Dams, Rob Gerth, and Orna Grumberg. A heuristic for the automatic generation of ranking functions. In Ganesh Gopalakrishnan, editor, *Workshop on Advances in Verification*, pages 1–8, 2000.

[DN03] Dennis Dams and Kedar S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *Proc. of the* 4th *Int. Conference on Verification, Model Checking, and Abstract Interpretation*, Lect. Notes in Comp. Sci., pages 310–324. Springer-Verlag, 2003.

[EC80] E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proc.* 7th *Int. Colloq. Aut. Lang. Prog.*, volume 85 of *Lect. Notes in Comp. Sci.*, pages 169–181. Springer-Verlag, 1980.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proc.* 29th *ACM Symp. Princ. of Prog. Lang.*, pages 191–202. ACM Press, 2002.

[GOR97] Erich Grädel, Martin Otto, and Eric Rosen. Undecidability results on two-variable logics. In Rüdiger Reischuk and Michel Morvan, editors, *Proc.* 14th *Annual Symp. on Theoretical Aspects* *of Computer Science*, volume 1200 of *Lect. Notes in Comp. Sci.*, pages 249–260. Springer-Verlag, 1997.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[IRR$^+$04a] Neil Immerman, Alexander Moshe Rabinovich, Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Proc.* 18th *Intl. Workshop on Computer Science Logic*, volume 3210 of *Lect. Notes in Comp. Sci.*, pages 160–174. Springer-Verlag, 2004.

[IRR$^+$04b] Neil Immerman, Alexander Moshe Rabinovich, Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. Verification via structure simulation. In *Proc.* 16th *Intl. Conference on Computer Aided Verification*, Lect. Notes in Comp. Sci., pages 281–294. Springer-Verlag, 2004.

[JM81] N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Knu69] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison Wesley, 1969.

[KP00] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.

[KP05] Y. Kesten and A. Pnueli. A Compositional Approach to CTL* Verification. *Theor. Comp. Sci.*, 331(2–3):397–428, 2005.

[KPR98] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proc.* 25th *Int. Colloq. Aut. Lang. Prog.*, volume 1443 of *Lect. Notes in Comp. Sci.*, pages 1–16. Springer-Verlag, 1998.

[KPV01] Y. Kesten, A. Pnueli, and M. Vardi. Verification by augmented abstraction: The automata theoretic view. *J. Comp. Systems Sci.*, 62:668–690, 2001.

[KS93] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proc.* 20th *ACM Symp. Princ. of Prog. Lang.*, pages 196–205, New York, NY, USA, 1993. ACM Press.

[KV05] O. Kupferman and M.Y. Vardi. From complementation to certification. *Theor. Comp. Sci.*, 345:83–100, 2005.

[LAS00] Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In Jens Palsberg, editor, *Proc.* 7th *Intl. Symp.*

on Static Analysis, volume 1824, pages 280–301. Springer-Verlag, 2000.

[LB04] S. Lahiri and R. Bryant. Constructing quantified invariants via predicate abstraction. In *Proc. of the* 5th *Int. Conference on Verification, Model Checking, and Abstract Interpretation*, pages 267–281, 2004.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proc.* 28th *ACM Symp. Princ. of Prog. Lang.*, pages 81–92. ACM Press, 2001.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proc.* 12th *ACM Symp. Princ. of Prog. Lang.*, pages 97–107, 1985.

[LS97] Naomi Lindenstrauss and Yehoshua Sagiv. Automatic termination analysis of logic programs. In *ICLP*, pages 63–77, 1997.

[MP91a] Zohar Manna and Amir Pnueli. Completing the temporal picture. *TCS*, 83(1):97–130, 1991.

[MP91b] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, New York, NY, USA, 1991.

[MP94] Z. Manna and A. Pnueli. Temporal verification diagrams. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer*

Software, volume 789 of *Lect. Notes in Comp. Sci.*, pages 726–765. Springer-Verlag, 1994.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[MS01] Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.

[MYRS05] Roman Manevich, Eran Yahav, Ganesan Ramalingam, and Shmuel Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In Radhia Cousot, editor, *Proc. of the* 6^{th} *Int. Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lect. Notes in Comp. Sci.*, pages 181–198. Springer, 2005.

[Nam01] Kedar S. Namjoshi. Certifying model checkers. In *Proc.* 13^{rd} *Intl. Conference on Computer Aided Verification*, volume 2102 of *Lect. Notes in Comp. Sci.*, pages 2–13. Springer-Verlag, 2001.

[Nam03] Kedar S. Namjoshi. Lifting temporal proofs through abstractions. In L.D. Zuck, P.C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Proc. of the* 4^{th} *Int. Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *Lect. Notes in Comp. Sci.*, pages 174–188. Springer-Verlag, 2003.

[Nel83] G. Nelson. Verifying reachability invariants of linked structures. In *Proc. 10th ACM Symp. Princ. of Prog. Lang.*, pages 38–47, 1983.

[PPR05] Amir Pnueli, Andreas Podelski, and Andrey Rybalchenko. Separating fairness and well-foundedness for the analysis of fair discrete systems. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Proc. 11th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lect. Notes in Comp. Sci.*, pages 124–139. Springer-Verlag, 2005.

[PPZ01a] D. Peled, A. Pnueli, and L. Zuck. From falsification to verification. In *Proc. 21st Conf. on Foundations of Software Technology and Theroretical Computer Science*, volume 2245 of *Lect. Notes in Comp. Sci.*, pages 292–304. Springer-Verlag, 2001.

[PPZ01b] D. Peled, A. Pnueli, and L. Zuck. From falsification to verification. In *Proc. 21st Conference on Foundations of Software Technology and Theroretical Computer Science (FSTTCS'01)*, volume 2245 of *Lect. Notes in Comp. Sci.*, pages 292–304. Springer-Verlag, 2001.

[PR03] Andreas Podelski and Andrey Rybalchenko. Software model checking of liveness properties via transition invariants. Research Report MPI-I-2003-2-004, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, December 2003.

[PR04a] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Verification, Model Checking, and Abstract Interpretation*, pages 239–251, 2004.

[PR04b] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In 19^{th} *IEEE Symp. on Logic in Comp. Sci.*, pages 32–41. IEEE Computer Society, 2004.

[PR05] Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In Jens Palsberg and Martín Abadi, editors, *Proc.* 32^{th} *ACM Symp. Princ. of Prog. Lang.*, pages 132–144. ACM Press, 2005.

[PS96] A. Pnueli and E. Shahar. A platform combining deductive with algorithmic verification. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, page 184, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[PZ01a] D. Peled and L. Zuck. From model checking to a temporal proof. In *Proc. of the* 8^{th} *Int. SPIN Workshop on Model Checking of*

Software, volume 2057 of *Lect. Notes in Comp. Sci.*, pages 1–14. Springer-Verlag, 2001.

[PZ01b] D. Peled and L. Zuck. From model checking to a temporal proof. In *Proc. of the 8th International SPIN Workshop on Model Checking of Software (SPIN'2001)*, volume 2057 of *Lect. Notes in Comp. Sci.*, pages 1–14. Springer-Verlag, 2001.

[RBH07] Zvonimir Rakamaric, Jesse D. Bingham, and Alan Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In Byron Cook and Andreas Podelski, editors, *Proc. of the* 8^{th} *Int. Conference on Verification, Model Checking, and Abstract Interpretation*, Lect. Notes in Comp. Sci. Springer-Verlag, 2007.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In 17^{th} *IEEE Symp. on Logic in Comp. Sci.*, pages 55–74. IEEE Computer Society, 2002.

[RS01] Noam Rinetzky and Mooly Sagiv. Interprocedural shape analysis for recursive programs. *Lecture Notes in Computer Science*, 2027:133–149, 2001.

[SRW99] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc.* 26^{th} *ACM*

Symp. Princ. of Prog. Lang., pages 105–118. Springer-Verlag, 1999.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[WKL+06] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. On field constraint analysis. In *Proc. of the 7th Int. Conference on Verification, Model Checking, and Abstract Interpretation*, 2006.

[YRS+06] Greta Yorsh, Alexander Moshe Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A logic of reachable patterns in linked data-structures. In Luca Aceto and Anna Ingólfsdóttir, editors, *Proc. 9th Intl. Conference on Foundations of Software Science and Computation Structures*, volume 3921 of *Lect. Notes in Comp. Sci.*, pages 94–110. Springer-Verlag, 2006.

Index