Cyclic Proofs of Program Termination in Separation Logic *

James Brotherston
Imperial College, London, UK
J.Brotherston@imperial.ac.uk

Richard Bornat
Middlesex University, London, UK
R.Bornat@mdx.ac.uk

Cristiano Calcagno
Imperial College, London, UK
ccris@doc.ic.ac.uk

Abstract

We propose a novel approach to proving the termination of heap-manipulating programs, which combines separation logic with *cyclic proof* within a Hoare-style proof system. Judgements in this system express (guaranteed) termination of the program when started from a given line in the program and in a state satisfying a given precondition, which is expressed as a formula of separation logic. The proof rules of our system are of two types: logical rules that operate on preconditions; and symbolic execution rules that capture the effect of executing program commands.

Our logical preconditions employ inductively defined predicates to describe heap properties, and proofs in our system are cyclic proofs: cyclic derivations in which some inductive predicate is unfolded infinitely often along every infinite path, thus allowing us to discard all infinite paths in the proof by an infinite descent argument. Moreover, the use of this soundness condition enables us to avoid the explicit construction and use of ranking functions for termination. We also give a completeness result for our system, which is relative in that it relies upon completeness of a proof system for logical implications in separation logic. We give examples illustrating our approach, including one example for which the corresponding ranking function is non-obvious: termination of the classical algorithm for in-place reversal of a (possibly cyclic) linked list.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Logics of programs

General Terms Verification, theory, reliability

Keywords separation logic, termination, cyclic proof, program verification, inductive definitions, Hoare logic

1. Introduction

Termination is an essential requirement of many computer programs yet, as every undergraduate computing student learns, deciding whether a particular program terminates on a given input is, in general, impossible. Thus research into proving program termination must focus on the development of appropriate frameworks and proof search heuristics for termination proofs for various classes of program. In this paper, we give a proof system, based upon separation logic and employing the method of cyclic proof, that is tailored towards proving termination of simple imperative programs.

Many important computer programs in use today are written in low-level imperative languages, and the ability to establish termination of such programs is thus clearly desirable. However, imperative programs have often proven resistant to formal analysis, at least partially due to the difficulty of reasoning about their use of pointer arithmetic and similar operations that operate on data stored in shared structures, such as the heap. Recently, *separation logic* was developed to help overcome this impasse by providing logical tools to reason about shared, mutable resource (Reynolds 2002). As well as the usual additive connectives of first-order logic, separation logic employs multiplicative connectives to express properties involving heap resource. The multiplicative conjunction $*$ denotes a division of the heap into two parts in which each conjunct holds respectively, and the multiplicative implication $-\!*$ expresses a property of resource addition: if an arbitrary heap satisfies the antecedent formula and is disjoint with the current heap, then the combined heap satisfies the consequent formula. Separation logic has successfully underpinned several program verification applications to date, including the Smallfoot automated tool (Berdine et al. 2006a), local shape analysis (Distefano et al. 2006; Calcagno et al. 2006), inductive recursion synthesis (Guo et al. 2007), total correctness proofs of non-trivial algorithms (Torp-Smith et al. 2004; Bornat et al. 2004) and, pertinently for our purposes, automated termination checking (Berdine et al. 2006b).

Existing work on program verification in separation logic has relied on the identification and use of suitable inductive definitions to express shape properties of the heap (at some given point in the execution of the program). A recent paper by the first author (Brotherston 2007) developed a general framework for inductive definitions in O'Hearn and Pym's *logic of bunched implications* BI (O'Hearn and Pym 99), which underlies the pure-logical part of separation logic, and gave proof systems for the extension, including an appropriate notion of *cyclic proof*. For logics featuring inductive predicates or similar fixed-point constructions, cyclic proof provides an alternative to traditional inductive proof, modelled on Fermat's infinite descent (Brotherston 2006; Brotherston and Simpson 2007; Sprenger and Dam 2003). In the case of inductively defined relations, one way of stating this principle is: if in some case of a proof some inductive definition is unfolded infinitely often, then that case may be disregarded. Essentially, this principle is sound because each inductive definition has a least-fixed point interpretation which can be constructed as the union of a chain of approximations, indexed by ordinals; unfolding a definition infinitely often can thus be seen as inducing an infinite descending chain of these ordinals, which contradicts their well-foundedness. In cyclic proof systems, the capacity for unfolding a definition infinitely often is built in to the system by allowing proofs to be non-well-founded, i.e. to contain infinite paths. Such proof structures

* Supported by EPSRC. The authors also gratefully acknowledge Josh Berdine, Peter O'Hearn, Alex Simpson and Hongseok Yang for enlightening discussions, and the anonymous referees for their helpful comments and suggestions.

POPL'08, January 7–12, 2008, San Francisco, California, USA.
Copyright © 2008 ACM 978-1-59593-689-9/08/0001...$5.00

are, in general, unsound, so a global "well-formedness" condition is additionally imposed on proofs to ensure that every infinite path can be disregarded by the infinite descent principle outlined above. The portion of proof that is not disregarded by this principle is thus finite, and sound for standard reasons.

Our main contribution in this paper is the formulation of a cyclic proof system tailored to proving termination of programs written in a simple, yet relatively expressive imperative programming language. Given some fixed program, the judgements of this system express guaranteed (non-faulting) termination of the program when started from a given line in the program and in a state satisfying a given precondition, which is expressed as a formula of separation logic. The proof rules of the system are of two types: logical rules that operate on the precondition, and *symbolic execution* rules that simulate program execution steps. Thus, a program execution path corresponds to a path in a derivation, interleaved with logical inferences.

A program terminates just if no infinite computation is possible, i.e. if all potentially infinite computations can be dismissed as logically contradictory. An instance of this principle is the size change principle for program termination of Lee, Jones and Ben-Amran, which states that a program is terminating if every infinite computation induces an infinite descending sequence of values from a well-ordered set (Lee et al. 2001). Here, we employ the existing techniques of cyclic proof (Brotherston 2007, 2006, 2005; Sprenger and Dam 2003; Schöpp and Simpson 2002) to consider and then dismiss potentially infinite computations in the manner described above. A cyclic *pre-proof* in our system is formed from partial derivation trees by identifying every 'bud' (a node to which no proof rule has yet been applied) with a syntactically identical interior node, so that pre-proofs can be immediately understood as cyclic graphs in which infinite program computations correspond to infinite proof paths. To ensure that all such computations can be disregarded, we demand that, to count as a *bona fide* cyclic proof, a pre-proof must satisfy a *global trace condition* which formalises the fact that some inductively defined predicate is unfolded infinitely often along every infinite path. This ensures the soundness of our cyclic proofs. Moreover, the use of this condition means that we do not need to construct explicit ranking functions for termination. Indeed, it appears possible to construct cyclic termination proofs for which it is difficult to construct or deploy a ranking function.

The remainder of this paper is structured as follows. In Section 2 we give the syntax and small-step semantics of a simple imperative programming language, TOY-C, which is nonetheless sufficient to express many pointer-based algorithms. In Section 3, we give the syntax and semantics of our language for program preconditions, which is an extension of separation logic with a schema for inductive definitions as given in Brotherston (2007). In Section 4, we give the rules of our proof system for termination, and then proceed to formalise the notion of a *cyclic proof* in this system in Section 5, (along very similar lines to the notions of cyclic proof employed in Brotherston (2007, 2006)). We provide a soundness theorem and also a *relative completeness* theorem for our system, the latter result demonstrating that the question of provability of a valid judgement in our system can always be reduced to the question of provability of a valid logical judgement in pure separation logic. In Section 6, we give some sample termination proofs in our cyclic system, including one example — the in-place reversal of a cyclic linked list — for which the termination measure is far from obvious. Finally, in Section 7 we summarise and identify the directions for future work.

2. TOY-C: a small imperative programming language

In this section we give the syntax and semantics of a toy programming language, TOY-C, that nevertheless contains the following essential features: conditional branching; assignment; dereferencing; and memory allocation/deallocation. This allows us to deal with manipulation of the heap, the traditional domain of separation logic, but the cyclic proof method applies to a far wider range of programs than we can consider here.

We assume the existence of a denumerably infinite set Var of variables, and a first-order language Σ_{exp}, called the *expression language*, satisfying $\Sigma_{\text{exp}} \supseteq \mathsf{Var}$ and containing a distinguished constant symbol nil. The *expressions* E of TOY-C are just the terms (defined as usual) of the expression language. The syntax of branching conditions $Cond$ and atomic commands C is then given by the following grammar:

$$\begin{array}{rcl} Cond & ::= & E = E \mid E \neq E \\ C & ::= & x := E \mid x := [E] \mid [E] := E \mid x := \mathtt{new}() \\ & & \mid \mathtt{free}(E) \mid \mathtt{if}\, Cond\, \mathtt{goto}\, j \mid \mathtt{stop} \end{array}$$

where $x := [E]$ and $[E] := F$ access and assign to the heap cell with address E. A *program* in TOY-C is a finite sequence of indexed commands $1 : C_1; \cdots ; n : C_n$ (we say that C_i is the command at program point i). We write the command $\mathtt{goto}\, j$ as an abbreviation of $\mathtt{if}\, \mathsf{nil} = \mathsf{nil}\, \mathtt{goto}\, j$.

We give the semantics of TOY-C programs using a basic RAM model. We fix a set Val of *(program) values* and a set $\mathsf{Loc} \subset \mathsf{Val}$ of *(program) locations*, and assume a distinguished "nullary" value $nil \in \mathsf{Val} - \mathsf{Loc}$. A *stack* in this model is a function $s : \mathsf{Var} \to \mathsf{Val}$, and a *heap* is a partial, finitely-defined function $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$; we write Stacks and Heaps for the set of all stacks and the set of all heaps respectively. We write $s[x \mapsto v]$ for the stack defined exactly as s except that $(s[x \mapsto v])(x) = v$, and adopt a similar notation for heaps. We write $\circ$ to denote *composition* of heaps: if h_1 and h_2 are heaps with $dom(h_1) \cap dom(h_2) = \emptyset$, then $h_1 \circ h_2$ is the composite heap defined by:

$$(h_1 \circ h_2)(l) = \begin{cases} h_1(l) & \text{if } l \in dom(h_1) \\ h_2(l) & \text{if } l \in dom(h_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The interpretation $[\![E]\!]s$ of an expression E in a stack s is standard: $[\![\mathsf{nil}]\!]s = nil$ and $[\![x]\!]s = s(x)$ for any $x \in \mathsf{Var}$ (given some fixed interpretation for any function symbols in the expression language, s can then be extended to all expressions in the usual way). Similarly, for branching conditions, we have $s \in [\![E_1 = E_2]\!]$ iff $[\![E_1]\!]s = [\![E_2]\!]s$, and $s \in [\![E_1 \neq E_2]\!]$ iff $[\![E_1]\!]s \neq [\![E_2]\!]s$.

A *(program) state* in our model is a triple (i, s, h), where $i \in \mathbb{N}$ represents the next line of the program to be executed (i.e. the value of the program counter), s is a stack and h is a heap. The small-step semantics of programs, presented in Figure 1, is given by a binary relation $\rightsquigarrow$ on program states, with the intended meaning that $(i, s, h) \rightsquigarrow (i', s', h')$ holds if the execution of the next command in the state (i, s, h) can result in the new program state (i', s', h'). We write $(i, s, h)\downarrow$ iff there is no infinite $\rightsquigarrow$-sequence beginning with $(i, s, h) \rightsquigarrow \ldots$, i.e. iff the program terminates when started in the state (i, s, h). When a command tries to access unallocated memory, the execution continues from the special location *fault*, and loops. This effectively equates memory errors and divergence. (Note also that the absence of a case for stop in the small-step semantics implies that our programs terminate immediately on encountering a stop command.)

$$\frac{C_i \equiv x := E}{(i,s,h) \rightsquigarrow (i+1, s[x \mapsto [\![E]\!]s], h)} \qquad \frac{C_i \equiv x := [E] \quad [\![E]\!]s \in \mathit{dom}(h)}{(i,s,h) \rightsquigarrow (i+1, s[x \mapsto h([\![E]\!]s)], h)} \qquad \frac{C_i \equiv [E] := E' \quad [\![E]\!]s \in \mathit{dom}(h)}{(i,s,h) \rightsquigarrow (i+1, s, h[[\![E]\!]s \mapsto [\![E']\!]s])}$$

$$\frac{C_i \equiv x := \mathtt{new}() \quad \ell \in \mathsf{Loc} \setminus \mathit{dom}(h) \quad v \in \mathsf{Val}}{(i,s,h) \rightsquigarrow (i+1, s[x \mapsto \ell], h[\ell \mapsto v])} \qquad \frac{C_i \equiv \mathtt{free}(E) \quad [\![E]\!]s \in \mathit{dom}(h)}{(i,s,h) \rightsquigarrow (i+1, s, (h \restriction (\mathit{dom}(h) \setminus \{[\![E]\!]s\}))}$$

$$\frac{C_i \equiv \mathtt{if}\,\mathit{Cond}\,\mathtt{goto}\,j \quad s \in [\![\mathit{Cond}]\!]}{(i,s,h) \rightsquigarrow (j,s,h)} \qquad \frac{C_i \equiv \mathtt{if}\,\mathit{Cond}\,\mathtt{goto}\,j \quad s \notin [\![\mathit{Cond}]\!]}{(i,s,h) \rightsquigarrow (i+1,s,h)}$$

$$\frac{}{(\mathit{fault},s,h) \rightsquigarrow (\mathit{fault},s,h)} \qquad \frac{C_i \equiv x := [E] \mid [E] := E' \mid \mathtt{free}(E) \quad [\![E]\!]s \notin \mathit{dom}(h)}{(i,s,h) \rightsquigarrow (\mathit{fault},s,h)}$$

Figure 1. Small-step operational semantics of TOY-C, given by the binary relation $\rightsquigarrow$ over $(\mathbb{N} \times \mathsf{Stacks} \times \mathsf{Heaps})$.

Example 2.1 (List traversal program)**.** Consider the C-like program for traversing a linked list:[1]

```
while (x!=nil) x:=[x];
```

The equivalent TOY-C program can be written as follows:

$$1:\ \mathtt{if}\,x = \mathsf{nil}\,\mathtt{goto}\,4;\ 2:\ x := [x];\ 3:\ \mathtt{goto}\,1;\ 4:\ \mathtt{stop}$$

Clearly this program terminates if the heap consists of an acyclic, singly-linked list whose first node is pointed to by x and whose last node contains the null pointer nil (because the length of the list remaining to be traversed clearly decreases at each iteration of the loop, and is initially finite). We give a formal proof in Section 6.

Our language has the termination monotonicity property (see Yang and O'Hearn (2002)). If a program terminates in a heap h then it terminates in every extension of h.

Proposition 2.2 (Termination monotonicity)**.** *If $(i,s,h)\downarrow$ and $h \circ h'$ is defined then $(i,s,h\circ h')\downarrow$.*

Proof. (Sketch) By contraposition, it suffices to show that the existence of an infinite $\rightsquigarrow$-sequence starting from $(i,s,h\circ h')$ implies the existence of an infinite $\rightsquigarrow$-sequence starting from (i,s,h). This follows from the following claim:
Claim. If $(i_1,s_1,h_1\circ h') \rightsquigarrow (i_2,s_2,h_2)$ then either:

1. $h_2 = h'' \circ h'$ for some h'', and $(i_1,s_1,h_1) \rightsquigarrow (i_2,s_2,h'')$, or;
2. $(i_1,s_1,h_1) \rightsquigarrow (fault,s_1,h_1)$.

Then given any infinite $\rightsquigarrow$-sequence starting from $(i,s,h\circ h')$, it is either the case that no step in this sequence accesses h', in which case every step in this sequence can be equally well carried out starting from (i,s,h), or that some step in the sequence accesses h', in which case the equivalent step in the sequence starting from (i,s,h) causes a fault and thus this sequence immediately diverges. In either case we have the required infinite $\rightsquigarrow$-sequence starting from (i,s,h).

It only remains to substantiate the claim, which follows from a straightforward case analysis on $(i_1,s_1,h_1\circ h') \rightsquigarrow (i_2,s_2,h_2)$. □

3. Separation logic with inductive definitions

In this section, we reprise the syntax and semantics of separation logic extended with a framework for inductive definitions, as given in Brotherston (2007). This gives us a framework for expressing heap properties using customised inductive definitions, which will be useful later in the expression of suitable preconditions for our TOY-C programs.

We assume a fixed first-order language Σ, and for reasons of expressivity stipulate that $\Sigma \supseteq \Sigma_{\text{exp}}$, i.e., that our logical language extends the TOY-C expression language. The *terms* of Σ are defined as usual, with variables drawn from the set Var. We write $t(x_1,\ldots,x_k)$ for a term t all of whose variables occur in $\{x_1,\ldots,x_k\}$, and we often use vector notation to abbreviate sequences, e.g. $\mathbf{x}$ for $(x_1,\ldots,x_k)$. The interpretation $[\![t]\!]s$ of a term t of Σ in a stack s is then defined as for expressions (provided we have given an interpretation for any constant or function symbol that is not in Σ_{exp}[2]).

In contrast to the usual situation in first-order logic, but in keeping with prior developments for first-order logic with inductive definitions (Brotherston 2006), we designate finitely many of the predicate symbols of Σ as special *inductive* symbols; a predicate symbol that is not inductive is called *ordinary*. For each predicate symbol Q of arity k (say) we assign an intepretation $[\![Q]\!] \in \mathsf{Pow}(\mathsf{Heaps}\times\mathsf{Val}^k)$. We insist that Σ contains a 0-ary ordinary predicate symbol emp and a binary ordinary predicate symbol $\mapsto$, whose interpretations are given respectively by:

$$\begin{array}{rcl} [\![\mathsf{emp}]\!] & = & \{h \mid \mathit{dom}(h) = \emptyset\} \\ [\![\mapsto]\!] & = & \{(h,v_1,v_2) \mid \mathit{dom}(h) = \{v_1\} \text{ and } h(v_1) = v_2\} \end{array}$$

Thus the predicate emp denotes the empty heap, while $v_1 \mapsto v_2$ denotes those heaps having exactly one location, which is denoted by v_1 and whose contents are denoted by v_2.

Our *formulas* are the usual formulas of predicate BI[3], given by the following grammar:

$$\begin{array}{l} F ::= \top \mid \bot \mid \mathsf{emp} \mid Q(t_1,\ldots,t_k)\ (k = \text{arity of } Q) \mid t_1 = t_2 \mid \\ \quad F \wedge F \mid F \vee F \mid F \rightarrow F \mid F * F \mid F \mathrel{-\!\!*} F \mid \exists x F \mid \forall x F \end{array}$$

where $t_1,\ldots,t_k$ range over the terms of Σ and Q ranges over the predicate symbols (both ordinary and inductive) of Σ. We use the standard precedences on the logical connectives, with $*$ and $\mathrel{-\!\!*}$ having the same logical precedence as $\wedge$ and $\rightarrow$ respectively, and use parentheses to disambiguate where necessary. Also, we write $\neg F$ as an abbreviation of the formula $F \rightarrow \bot$.

We may then define the standard satisfaction relation $s,h \models F$, by induction on the structure of F, as shown in Figure 2. However, we shall further insist that the interpretation $[\![P]\!]$ of each inductive predicate symbol P coincides with the standard interpretation, which is fixed by a given inductive definition for P. Our inductive definition schema (essentially the one formulated in Brotherston (2007)) is given by the following definition:

[1] For simplicity, in this and other examples we treat lists which contain nothing but pointers to successor cells.

[2] Our extension of stacks to arbitrary logical terms is a technical simplification. We could instead use stack-extending environments to interpret terms.

[3] However, note that here we write emp for the multiplicative unit I of BI.

$$
\begin{array}{rcl}
s, h \models \top & \Leftrightarrow & \text{true} \\
s, h \models \bot & \Leftrightarrow & \text{false} \\
s, h \models t_1 = t_2 & \Leftrightarrow & \llbracket t_1 \rrbracket s = \llbracket t_2 \rrbracket s \\
s, h \models Q(t_1, \dots, t_k) & \Leftrightarrow & (h, \llbracket t_1 \rrbracket s, \dots, \llbracket t_k \rrbracket s) \in \llbracket Q \rrbracket \\
 & & (Q \text{ ordinary or inductive}) \\
s, h \models F_1 \vee F_2 & \Leftrightarrow & s, h \models F_1 \text{ or } s, h \models F_2 \\
s, h \models F_1 \wedge F_2 & \Leftrightarrow & s, h \models F_1 \text{ and } s, h \models F_2 \\
s, h \models F_1 \rightarrow F_2 & \Leftrightarrow & s, h \models F_1 \text{ implies } s, h \models F_2 \\
s, h \models F_1 * F_2 & \Leftrightarrow & \exists h_1, h_2.\, h = h_1 \circ h_2 \text{ and} \\
 & & s, h_1 \models F_1 \text{ and } s, h_2 \models F_2 \\
s, h \models F_1 \mathrel{-\!\!*} F_2 & \Leftrightarrow & \forall h'.\, h \circ h' \text{ defined and } s, h' \models F_1 \\
 & & \text{implies } s, h \circ h' \models F_2 \\
s, h \models \forall x F & \Leftrightarrow & \forall v \in \mathsf{Val}.\, s[x \mapsto v], h \models F \\
s, h \models \exists x F & \Leftrightarrow & \exists v \in \mathsf{Val}.\, s[x \mapsto v], h \models F
\end{array}
$$

Figure 2. The relation $s, h \models F$ for satisfaction of a separation logic formula F by a stack s and heap h.

Definition 3.1 (Inductive definition)**.** An *inductive definition* of an inductive predicate symbol P is a set of statements:

$$C_1(\mathbf{x_1}) \Rightarrow P\,\mathbf{t_1}(\mathbf{x_1}), \dots, C_k(\mathbf{x_k}) \Rightarrow P\,\mathbf{t_k}(\mathbf{x_k})$$

where $k \in \mathbb{N}$ and $C_1(\mathbf{x_1}), \dots, C_k(\mathbf{x_k})$ are *inductive clauses*, given by the following grammar:

$$\begin{array}{rl} C(\mathbf{x}) ::= & P\mathbf{t}(\mathbf{x}) \mid \hat{F}(\mathbf{x}) \mid C(\mathbf{x}) \wedge C(\mathbf{x}) \mid C(\mathbf{x}) * C(\mathbf{x}) \\ & \mid \hat{F}(\mathbf{x}) \rightarrow C(\mathbf{x}) \mid \hat{F}(\mathbf{x}) \mathrel{-\!\!*} C(\mathbf{x}) \mid \forall x C(\mathbf{x}) \end{array}$$

where P ranges over the inductive predicate symbols of Σ and $\hat{F}(\mathbf{x})$ ranges over all formulas in which no inductive predicates occur and whose free variables are contained in $\{\mathbf{x}\}$.

Each statement $C_i(\mathbf{x_i}) \Rightarrow P_i\mathbf{t_i}(\mathbf{x_i})$ is read as a disjunctive clause of the definition of the inductive predicate symbol P_i. As in Brotherston (2007), our use of $\hat{F}(\mathbf{x})$ on the left of implications in inductive clauses is designed to ensure monotonicity of our inductive definitions. A more liberal definition scheme might allow inductively defined predicates to occur negatively subject to an appropriate stratification of inductive definitions, as in iterated inductive definitions (Martin-Löf 1971).

The standard interpretation of an inductive predicate symbol P is then, as usual, the least prefixed point of a monotone operator constructed from the inductive definitions. This operator, and the process for constructing its least fixed point, are essential to understanding the soundness of our cyclic proof system, so we include the details here:

Definition 3.2 (Definition set operator)**.** Let the inductive predicate symbols of Σ be $P_1, \dots, P_n$ with arities $a_1, \dots, a_n$ respectively, and suppose we have a unique inductive definition for each predicate symbol P_i. Then, for each $i \in \{1, \dots, n\}$, from the inductive definition for P_i, say:

$$C_1(\mathbf{x_1}) \Rightarrow P_i\,\mathbf{t_1}(\mathbf{x_1}), \dots, C_k(\mathbf{x_k}) \Rightarrow P_i\,\mathbf{t_k}(\mathbf{x_k})$$

we obtain a corresponding n-ary function φ_i : $(\mathsf{Pow}(\mathsf{Heaps} \times \mathsf{Val}^{a_1}) \times \ldots \times \mathsf{Pow}(\mathsf{Heaps} \times \mathsf{Val}^{a_n})) \rightarrow \mathsf{Pow}(\mathsf{Heaps} \times \mathsf{Val}^{a_i})$ as follows:

$$\begin{array}{rl} \varphi_i(\mathbf{X}) = \bigcup_{1 \le j \le k} \{ & (h, \llbracket \mathbf{t_j} \rrbracket (s[\mathbf{x_j} \mapsto \mathbf{d}])) \mid \\ & s[\mathbf{x_j} \mapsto \mathbf{d}], h \models_{\llbracket \mathbf{P} \rrbracket \mapsto \mathbf{X}} C_j(\mathbf{x_j}) \} \end{array}$$

where s is an arbitrary stack and $\models_{\llbracket \mathbf{P} \rrbracket \mapsto \mathbf{X}}$ is the satisfaction relation defined exactly as in Figure 2 except that $\llbracket P_i \rrbracket = \pi_i^n(\mathbf{X})$ for each $i \in \{1, \dots, n\}$. Note that any variables occurring in the right hand side but not the left hand side of the set comprehension in the definition of φ_i above are, implicitly, existentially quantified over the entire right hand side of the comprehension. Then the *definition set operator* for $P_1, \dots, P_n$ is the operator $\varphi_{\mathbf{P}}$, with domain and codomain $\mathsf{Pow}(\mathsf{Heaps} \times \mathsf{Val}^{a_1}) \times \ldots \times \mathsf{Pow}(\mathsf{Heaps} \times \mathsf{Val}^{a_n})$, defined by:

$$\varphi_{\mathbf{P}}(\mathbf{X}) = (\varphi_1(\mathbf{X}), \dots, \varphi_n(\mathbf{X}))$$

Example 3.3. Consider the following inductive definition for a binary inductive predicate symbol ls:

$$\begin{array}{rcl} \mathsf{emp} & \Rightarrow & ls\,x\,x \\ x \mapsto x' * ls\,x'\,y & \Rightarrow & ls\,x\,y \end{array}$$

Then $\llbracket ls \rrbracket$ is the least prefixed point of the following operator, with domain and codomain $\mathsf{Pow}(\mathsf{Heaps} \times \mathsf{Val}^2)$:

$$\begin{array}{rcl} \varphi_{ls}(X) & = & \{(\mathsf{emp}, (v, v)) \mid v \in \mathsf{Val}\} \\ & \cup & \{(h_1 \circ h_2, (v, v')) \mid \exists w \in \mathsf{Val}.\, (h_1, (v, w)) \in \llbracket \mapsto \rrbracket \\ & & \qquad \text{and } (h_2, (w, v')) \in X\} \end{array}$$

The predicate ls denotes singly-linked list segments; $ls\,x\,y$ is true of those heaps that represent a (possibly cyclic) linked list segment whose first element is pointed to by x and whose last element contains the pointer y. (If the list is acyclic, then y is a dangling pointer.)

The operator generated from a set of inductive definitions by Definition 3.2 is monotone (Brotherston 2007), and consequently has a least (pre)fixed point, which gives the standard interpretation for the inductively defined predicates of the language. As is well-known (see e.g. Aczel (1977)), this least prefixed point can be iteratively approached in ordinal-indexed stages or *approximants*:

Definition 3.4 (Approximants)**.** Let $\varphi_{\mathbf{P}}$ be the definition set operator for the inductive predicates $P_1, \dots, P_n$ of Σ as in Definition 3.2. Define a chain of ordinal-indexed sets $(\varphi_{\mathbf{P}}^{\alpha})_{\alpha \ge 0}$ by transfinite induction: $\varphi_{\mathbf{P}}^{\alpha} = \bigcup_{\beta < \alpha} \varphi_{\mathbf{P}}(\varphi_{\mathbf{P}}^{\beta})$ (note that this implies $\varphi_{\mathbf{P}}^{0} = (\emptyset, \dots, \emptyset)$). Then for each $i \in \{1, \dots, n\}$, the set $P_i^{\alpha} = \pi_i^n(\varphi_{\mathbf{P}}^{\alpha})$ is called the α^{th} *approximant of* P_i.

Definition 3.5 (Standard interpretation)**.** The function $\llbracket - \rrbracket$ is said to respect the standard interpretation of the inductive predicates $P_1, \dots, P_n$ of Σ if for all $i \in \{1, \dots, n\}$, we have $\llbracket P_i \rrbracket = \bigcup_{\alpha} P_i^{\alpha}$.

From now on, we assume that $\llbracket - \rrbracket$ always respects the standard interpretation of any inductive predicate symbols.

4. Proof rules for termination judgements

In this section we give the rules of a Hoare-style proof system for proving termination of programs written in TOY-C. We write *termination judgements* of the form $\Gamma \vdash_i \downarrow$, where i is a program point and Γ is a *bunch* (O'Hearn and Pym 99):

Definition 4.1 (Bunch)**.** A *bunch* is a tree whose leaves are formulas (as defined in Section 3) and whose internal nodes are ';' or ',' (which are logically equivalent to $\wedge$ and $*$, respectively).

We write $\Gamma(\Delta)$ to mean that Γ is a bunch of which Δ is a subtree (also called a "sub-bunch"), and write $\Gamma(\Delta')$ for the bunch obtained by replacing the considered instance of Δ by Δ' in $\Gamma(\Delta)$. We observe that a bunch Γ can be considered as a formula by replacing every occurrence of ';' by '$\wedge$' and every occurrence of ',' by '$*$', and thus we extend our notion of satisfaction (cf. Fig. 2) to bunches in the obvious manner.

Definition 4.2 (Coherent equivalence)**.** *Coherent equivalence*, $\equiv$, is the smallest binary relation on bunches satisfying the following (commutative monoid) equations:

$$\begin{array}{cc} \Gamma_1; (\Gamma_2; \Gamma_3) \equiv (\Gamma_1; \Gamma_2); \Gamma_3 & \Gamma_1, (\Gamma_2, \Gamma_3) \equiv (\Gamma_1, \Gamma_2), \Gamma_3 \\ \Gamma_1; \Gamma_2 \equiv \Gamma_2; \Gamma_1 & \Gamma_1, \Gamma_2 \equiv \Gamma_2, \Gamma_1 \\ \top; \Gamma \equiv \Gamma & \mathsf{emp}, \Gamma \equiv \Gamma \end{array}$$

plus the rule of congruence: $\Delta \equiv \Delta'$ implies $\Gamma(\Delta) \equiv \Gamma(\Delta')$.

Definition 4.3 (Validity)**.** A termination judgement $\Gamma \vdash_i\downarrow$ is *valid* iff $s, h \models \Gamma$ implies $(i, s, h)\downarrow$ for all $s \in \mathsf{Stacks}$ and $h \in \mathsf{Heaps}$.

Our Hoare logic rules for termination judgements are given in Figure 3. The symbolic execution rules for commands are adaptations of standard rules for separation logic (Berdine et al. 2005). The convention is that primed variables x' and x'' in the preconditions must be chosen fresh. The general rules are similar to rules in a proof system for implication. In places, our rules differ slightly from the typical presentation of Hoare logic in order to simplify the tracking of occurrences of inductive predicates, which is crucial for cyclic proofs. For example, our use of the cut rule in place of the usual rule of consequence in Hoare logic allows us to track inductive predicates in the "context part" Γ of the rule. Our cut rule is presented with just one premise, with the usual second premise $\Delta \vdash F$ instead given as a side condition (which can be seen as an analogue of the usual side condition, involving semantic entailment, on the rule of consequence). This separates our Hoare reasoning, and tracking of inductive predicates, from an external reasoning process for establishing pure implication (which itself might use cyclic proofs (Brotherston 2007) or an existing theorem prover such as Berdine et al. (2006a)). The multiplicative weakening rule (WkM) does not normally hold in separation logic but is valid in the case of termination, justified by Proposition 2.2.

The case-split rule for inductive predicates will play a core role in our cyclic proof system, and deserves more explanation. Suppose that the inductive predicate P has definition:

$$C_1(\mathbf{x_1}) \Rightarrow P\,\mathbf{t_1}(\mathbf{x_1}), \ldots, C_k(\mathbf{x_k}) \Rightarrow P\,\mathbf{t_k}(\mathbf{x_k})$$

Our case-split rule:

$$\frac{(\Gamma(\mathbf{t} = \mathbf{t_j}(\mathbf{x}); C_j(\mathbf{x})) \vdash_i\downarrow)_{1\le j\le k}}{\Gamma(P\mathbf{t}) \vdash_i\downarrow}\ \begin{array}{l}\text{(Case } P\text{)}\\ \forall x \in \{\mathbf{x}\}.\, x \notin FV(\Gamma(P\mathbf{t}))\end{array}$$

unfolds an occurrence $P\mathbf{t}$ of an inductive predicate according to its definition (with the surrounding context Γ remaining unaffected).

Example 4.4 (List segment)**.** Let ls be the inductive predicate for (possibly cyclic) list segments defined in Example 3.3. The inductive definition of ls determines the following case-split rule:

$$\frac{\Gamma(t = u; \mathsf{emp}) \vdash_i\downarrow \quad \Gamma(t{\mapsto}x' * ls\, x'\, u) \vdash_i\downarrow}{\Gamma(ls\, t\, u) \vdash_i\downarrow}\ \text{(Case } ls\text{)}$$

Proposition 4.5. *The Hoare logic rules given in Figure 3 are locally sound. That is to say, if all of the premises of any rule instance are valid, and any relevant side conditions are satisfied, then the conclusion of the rule instance is also valid.*

5. Cyclic proofs of program termination

We now define a notion of *cyclic proof* for our termination judgements. First we define cyclic *pre-proofs*: finite derivation trees together with a function that, for each leaf node in the tree to which no proof rule has been applied (called a *bud*), assigns an interior node in the tree that is labelled with an identical judgement; this interior node is called the *companion* of the bud. A cyclic pre-proof can thus be seen as a representation of a regular, infinite derivation tree by a cyclic graph.

Definition 5.1 (Companion)**.** Let B be a bud of a derivation tree $\mathcal{D}$. An internal node C in $\mathcal{D}$ is said to be a *companion* for B if they have the same judgement labelling.

By assigning a companion to each bud node in a finite derivation tree, one obtains a finite representation of an associated (regular) infinite tree:

Definition 5.2 (Cyclic pre-proof)**.** A *cyclic pre-proof* of a termination judgement $F \vdash_i\downarrow$ is a pair $\mathcal{P} = (\mathcal{D}, \mathcal{R})$, where $\mathcal{D}$ is a derivation tree constructed according to the Hoare logic rules for termination judgements given in Section 4 and whose root is labelled by $F \vdash_i\downarrow$, and $\mathcal{R}$ is a function assigning a companion to every bud of $\mathcal{D}$.

If $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ is a cyclic pre-proof, we write $\mathcal{G}_\mathcal{P}$ for the graph obtained from $\mathcal{D}$ by identifying each bud node B in $\mathcal{D}$ with its companion $\mathcal{R}(B)$.

Definition 5.3 (Trace)**.** Let $\mathcal{P}$ be a pre-proof and let $(\Gamma_k \vdash_{i_k}\downarrow)_{k\ge 0}$ be a path in $\mathcal{G}_\mathcal{P}$. A *trace following* $(\Gamma_k \vdash_{i_k}\downarrow)_{k\ge 0}$ is a sequence $(\tau_k)_{k\ge 0}$ such that, for all k, τ_k is the position of a leaf F_{τ_k} of Γ_k. Furthermore, for each $k \ge 0$, one of the following conditions must hold:

1. (F_{τ_k} is in the active part of the rule) $\Gamma_k \vdash_{i_k}\downarrow$ is the conclusion of one of the following inferences, τ_k is the position of the underlined formula in the conclusion and τ_{k+1} is the position of one of the underlined formulae in the premise:

$$\frac{\Gamma(\underline{F_1}; \underline{F_2}) \vdash_i\downarrow}{\Gamma(\underline{F_1 \wedge F_2}) \vdash_i\downarrow}\ (\wedge) \qquad \frac{\Gamma(\underline{F_2}) \vdash_i\downarrow}{\Gamma(\Delta, \underline{F_1 \mathrel{-\!\!*} F_2}) \vdash_i\downarrow}\ \Delta \vdash F_1\ (\mathrel{-\!\!*})$$

$$\frac{\Gamma(\underline{F_1}, \underline{F_2}) \vdash_i\downarrow}{\Gamma(\underline{F_1 * F_2}) \vdash_i\downarrow}\ (*) \qquad \frac{\Gamma(\Delta; \underline{F_2}) \vdash_i\downarrow}{\Gamma(\Delta; \underline{F_1 \rightarrow F_2}) \vdash_i\downarrow}\ \Delta \vdash F_1\ (\rightarrow)$$

$$\frac{\Gamma(\underline{F[t/x]}) \vdash_i\downarrow}{\Gamma(\underline{\forall x F}) \vdash_i\downarrow}\ (\forall) \qquad \frac{\Gamma(\mathbf{t} = \mathbf{t_j}(\mathbf{x}); \underline{C_j(\mathbf{x})}) \vdash_i\downarrow\ \ldots}{\Gamma(\underline{P\mathbf{t}}) \vdash_i\downarrow}\ \text{(Case } P\text{)}$$

(We remark that, due to the form of our inductive definitions (cf. Defn 3.1), we never need to trace formulas through the active part of the rules $(\vee)$ or $(\exists)$.) In the case where (Case P) is applied with conclusion $\Gamma_k \vdash_{i_k}\downarrow$ as above, and τ_k and τ_{k+1} are the positions of the leaves of Γ_k and Γ_{k+1} indicated by the underlining, the trace is said to *progress at* k. An *infinitely progressing trace* is a trace that progresses at infinitely many points.

2. (F_{τ_k} is not in the active part of the rule) τ_{k+1} is the position of the leaf in Γ_{k+1} corresponding to τ_k in Γ_k, modulo any splitting of Γ_k performed by the rule. (Thus $F_{\tau_{k+1}} = F_{\tau_k}$, modulo any substitution performed by the rule.) E.g. if $\Gamma_k \vdash_{i_k}\downarrow$ is the conclusion of the inference:

$$\frac{\Gamma(F_2) \vdash_i\downarrow}{\Gamma(\Delta, F_1 \mathrel{-\!\!*} F_2) \vdash_i\downarrow}\ \Delta \vdash F_1\ (\mathrel{-\!\!*})$$

then τ_{k+1} and τ_k are the same position in $\Gamma(-)$ (and thus $F_{\tau_{k+1}} = F_{\tau_k}$). Similarly, if $\Gamma_k \vdash_{i_k}\downarrow$ is the conclusion of the inference:

$$\frac{x = t[x'/x]; (E \mapsto t, \Gamma)[x'/x] \vdash_{i+1}\downarrow}{E \mapsto t, \Gamma \vdash_i\downarrow}\ C_i \equiv x := [E]$$

then τ_{k+1} and τ_k are the same position in $\Gamma(-)$ (and thus $F_{\tau_{k+1}} = F_{\tau_k}[x'/x]$). As a final example, if $\Gamma_k \vdash_{i_k}\downarrow$ is the conclusion of the inference:

$$\frac{\Gamma' \vdash_i\downarrow}{\Gamma \vdash_i\downarrow}\ \Gamma \equiv \Gamma'\ \text{(Equiv)}$$

then the position of τ_{k+1} in the rearranged bunch Γ' must respect the original position of τ_k in Γ in the obvious way (and thus $F_{\tau_{k+1}} = F_{\tau_k}$).

Symbolic execution rules:

$$\dfrac{x = E[x'/x]; \Gamma[x'/x] \vdash_{i+1}\downarrow}{\Gamma \vdash_i\downarrow}\, C_i \equiv x := E \qquad \dfrac{Cond; \Gamma \vdash_j\downarrow \quad \neg Cond; \Gamma \vdash_{i+1}\downarrow}{\Gamma \vdash_i\downarrow}\, C_i \equiv \mathtt{if}\, Cond\, \mathtt{goto}\, j$$

$$\dfrac{x = t[x'/x]; (E \mapsto t, \Gamma)[x'/x] \vdash_{i+1}\downarrow}{E \mapsto t, \Gamma \vdash_i\downarrow}\, C_i \equiv x := [E] \qquad \dfrac{x \mapsto x'', \Gamma[x'/x] \vdash_{i+1}\downarrow}{\Gamma \vdash_i\downarrow}\, C_i \equiv x := \mathtt{new}()$$

$$\dfrac{E_0 \mapsto E_1, \Gamma \vdash_{i+1}\downarrow}{E_0 \mapsto t, \Gamma \vdash_i\downarrow}\, C_i \equiv [E_0] := E_1 \qquad \dfrac{\Gamma \vdash_{i+1}\downarrow}{E \mapsto t, \Gamma \vdash_i\downarrow}\, C_i \equiv \mathtt{free}(E) \qquad \dfrac{}{\Gamma \vdash_i\downarrow}\, C_i \equiv \mathtt{stop}$$

General rules:

$$\dfrac{\Gamma(\Delta) \vdash_i\downarrow}{\Gamma(\Delta; \Delta') \vdash_i\downarrow}\,\text{(WkA)} \qquad \dfrac{\Gamma(\Delta) \vdash_i\downarrow}{\Gamma(\Delta, \Delta') \vdash_i\downarrow}\,\text{(WkM)} \qquad \dfrac{\Gamma(\Delta; \Delta) \vdash_i\downarrow}{\Gamma(\Delta) \vdash_i\downarrow}\,\text{(Contr)} \qquad \dfrac{\Gamma' \vdash_i\downarrow}{\Gamma \vdash_i\downarrow}\, \Gamma \equiv \Gamma' \;\text{(Equiv)}$$

$$\dfrac{\Gamma(F) \vdash_i\downarrow}{\Gamma(\Delta) \vdash_i\downarrow}\; \Delta \vdash F \;\text{(Cut)} \qquad \dfrac{\Gamma \vdash_i\downarrow}{\Gamma[t/x] \vdash_i\downarrow}\; x \text{ not a program variable} \;\text{(Subst)} \qquad \dfrac{\Gamma(\top)[t_2/x, t_1/y] \vdash_i\downarrow}{\Gamma(t_1 = t_2)[t_1/x, t_2/y] \vdash_i\downarrow}\,(=)$$

$$\dfrac{}{\bot \vdash_i\downarrow}\,(\bot) \qquad \dfrac{\Gamma(F_1) \vdash_i\downarrow \quad \Gamma(F_2) \vdash_i\downarrow}{\Gamma(F_1 \vee F_2) \vdash_i\downarrow}\,(\vee) \qquad \dfrac{\Gamma(F_1; F_2) \vdash_i\downarrow}{\Gamma(F_1 \wedge F_2) \vdash_i\downarrow}\,(\wedge) \qquad \dfrac{\Gamma(\Delta; F_2) \vdash_i\downarrow}{\Gamma(\Delta; F_1 \rightarrow F_2) \vdash_i\downarrow}\; \Delta \vdash F_1 \;(\rightarrow)$$

$$\dfrac{\Gamma(F[t/x]) \vdash_i\downarrow}{\Gamma(\forall x F) \vdash_i\downarrow}\,(\forall) \qquad \dfrac{\Gamma(F[z/x]) \vdash_i\downarrow}{\Gamma(\exists x F) \vdash_i\downarrow}\; z \notin FV(\Gamma(\exists x F)) \;(\exists) \qquad \dfrac{\Gamma(F_1, F_2) \vdash_i\downarrow}{\Gamma(F_1 * F_2) \vdash_i\downarrow}\,(*) \qquad \dfrac{\Gamma(F_2) \vdash_i\downarrow}{\Gamma(\Delta, F_1 \mathrel{-\!\!*} F_2) \vdash_i\downarrow}\; \Delta \vdash F_1 \;(\mathrel{-\!\!*})$$

Case-split rule:

$$\dfrac{(\Gamma(\mathbf{t} = \mathbf{t_j}(\mathbf{x}); C_j(\mathbf{x})) \vdash_i\downarrow)_{1 \le j \le k}}{\Gamma(P\mathbf{t}) \vdash_i\downarrow}\; \begin{array}{c} C_1(\mathbf{x_1}) \Rightarrow P\mathbf{t_1}(\mathbf{x_1}), \ldots, C_k(\mathbf{x_k}) \Rightarrow P\mathbf{t_k}(\mathbf{x_k}) \\ \forall x \in \{\mathbf{x}\}.\, x \notin FV(\Gamma(P\mathbf{t})) \end{array} \;\text{(Case } P\text{)}$$

Figure 3. Hoare logic rules for termination judgements.

Definition 5.4 (Proof)**.** A pre-proof $\mathcal{P}$ is a *proof* if for every infinite path π in $\mathcal{G}_{\mathcal{P}}$, there is an infinitely progressing trace following some tail of π.

Theorem 5.5. *If there is a proof of* $\Gamma \vdash_i\downarrow$ *then* $\Gamma \vdash_i\downarrow$ *is valid.*

An outline proof of Theorem 5.5 is given in Appendix A.

Proposition 5.6. *It is decidable whether a pre-proof is a proof.*

As well as soundness, our proof system enjoys the following *relative completeness* property:

Theorem 5.7 (Relative Completeness)**.** *Under the assumption that the underlying proof system for ordinary implications* $\Gamma \vdash F$ *is complete, if* $\Gamma \vdash_i\downarrow$ *is valid then there is a proof of* $\Gamma \vdash_i\downarrow$.

Intuitively, it is possible to define inductive predicates $term_i$ which ensure termination starting from program point i, and the proof that Γ ensures termination at i can be reduced to an ordinary implication $\Gamma \vdash term_i\, \mathbf{x}$. This situation is analogous to what happens in ordinary Hoare logic, where relative completeness proofs are typically expressed with respect to an oracle for implications.

We give an outline proof of Theorem 5.7 in Appendix B.

6. Examples of cyclic termination proofs

We show three examples, two straightforward and one more intricate. In each example we have treated goto _ as if it had a single-premise rule of its own.

Example 6.1 (Termination of list traversal)**.** Consider the program from Example 2.1 with precondition $ls\, x\, \mathsf{nil}$, with predicate ls as in Example 3.3. Figure 4 gives a proof of termination. We show the association of a suitable companion to the only bud in the pre-proof with a (red) arrow. Note that there is only one infinite path in the pre-proof – which goes around the cycle – and the associated trace (underlined) makes progress infinitely often at the case-split rule. Therefore we have termination.

This proof uses multiplicative weakening (WkM), throwing away part of the heap, in this case the cells of the list that have already been traversed. This is a peculiar thing to do in separation logic, in which proofs are careful to account for all resource. In the next two examples we show that proofs which more carefully account are possible, but for termination of list traversal we really only need to know that the list yet to be traversed diminishes, and the proof of figure 4 is all that is required.

Example 6.2 (Termination of in-place list reversal)**.** The classical in-place reverse algorithm reverses a list in place, using two variables and no additional heap space:

$$y := \mathsf{nil};\ \mathsf{while}\ x \neq \mathsf{nil}\ \mathsf{do}\ z := x; x := [x]; [z] := y; y := z\ \mathsf{od}$$

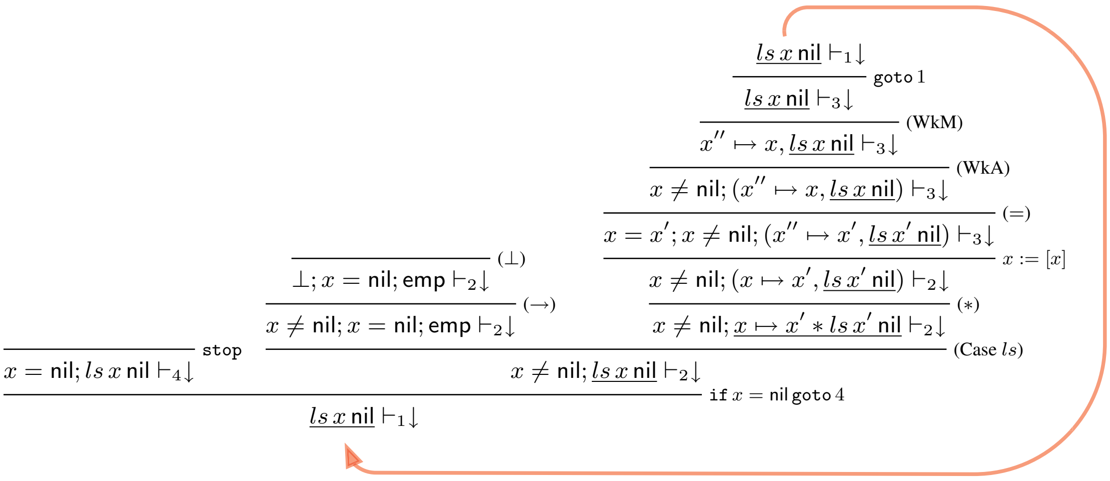

$\underline{ls\,x\,\mathsf{nil}} \vdash_1 \downarrow$
goto 1
$\underline{ls\,x\,\mathsf{nil}} \vdash_3 \downarrow$
(WkM)
$x'' \mapsto x, \underline{ls\,x\,\mathsf{nil}} \vdash_3 \downarrow$
(WkA)
$x \neq \mathsf{nil}; (x'' \mapsto x, \underline{ls\,x\,\mathsf{nil}}) \vdash_3 \downarrow$
(=)
$x = x'; x \neq \mathsf{nil}; (x'' \mapsto x', \underline{ls\,x'\,\mathsf{nil}}) \vdash_3 \downarrow$
$x := [x]$
(⊥)
$\bot; x = \mathsf{nil}; \mathsf{emp} \vdash_2 \downarrow$
(→)
$x \neq \mathsf{nil}; (x \mapsto x', \underline{ls\,x'\,\mathsf{nil}}) \vdash_2 \downarrow$
(∗)
$x \neq \mathsf{nil}; x = \mathsf{nil}; \mathsf{emp} \vdash_2 \downarrow$
$x \neq \mathsf{nil}; \underline{x \mapsto x'} * \underline{ls\,x'\,\mathsf{nil}} \vdash_2 \downarrow$
stop
(Case ls)
$x = \mathsf{nil}; ls\,x\,\mathsf{nil} \vdash_4 \downarrow$
$x \neq \mathsf{nil}; \underline{ls\,x\,\mathsf{nil}} \vdash_2 \downarrow$
if $x = \mathsf{nil}$ goto 4
$\underline{ls\,x\,\mathsf{nil}} \vdash_1 \downarrow$

Figure 4. A termination proof of list traversal

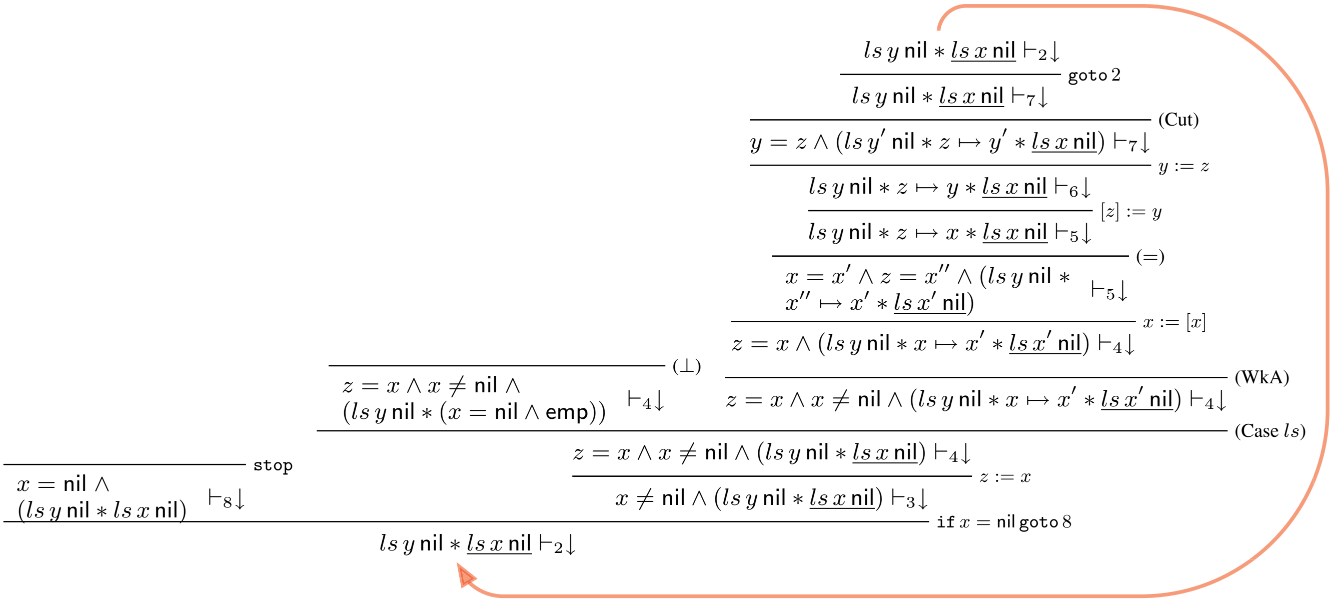

$ls\,y\,\mathsf{nil} * \underline{ls\,x\,\mathsf{nil}} \vdash_2 \downarrow$
goto 2
$ls\,y\,\mathsf{nil} * \underline{ls\,x\,\mathsf{nil}} \vdash_7 \downarrow$
(Cut)
$y = z \wedge (ls\,y'\,\mathsf{nil} * z \mapsto y' * \underline{ls\,x\,\mathsf{nil}}) \vdash_7 \downarrow$
$y := z$
$ls\,y\,\mathsf{nil} * z \mapsto y * \underline{ls\,x\,\mathsf{nil}} \vdash_6 \downarrow$
$[z] := y$
$ls\,y\,\mathsf{nil} * z \mapsto x * \underline{ls\,x\,\mathsf{nil}} \vdash_5 \downarrow$
(=)
$x = x' \wedge z = x'' \wedge (ls\,y\,\mathsf{nil} * x'' \mapsto x' * \underline{ls\,x'\,\mathsf{nil}}) \vdash_5 \downarrow$
$x := [x]$
$z = x \wedge (ls\,y\,\mathsf{nil} * x \mapsto x' * \underline{ls\,x'\,\mathsf{nil}}) \vdash_4 \downarrow$
(⊥)
$z = x \wedge x \neq \mathsf{nil} \wedge (ls\,y\,\mathsf{nil} * (x = \mathsf{nil} \wedge \mathsf{emp})) \vdash_4 \downarrow$
(WkA)
$z = x \wedge x \neq \mathsf{nil} \wedge (ls\,y\,\mathsf{nil} * x \mapsto x' * \underline{ls\,x'\,\mathsf{nil}}) \vdash_4 \downarrow$
(Case ls)
stop
$z = x \wedge x \neq \mathsf{nil} \wedge (ls\,y\,\mathsf{nil} * \underline{ls\,x\,\mathsf{nil}}) \vdash_4 \downarrow$
$z := x$
$x = \mathsf{nil} \wedge (ls\,y\,\mathsf{nil} * ls\,x\,\mathsf{nil}) \vdash_8 \downarrow$
$x \neq \mathsf{nil} \wedge (ls\,y\,\mathsf{nil} * \underline{ls\,x\,\mathsf{nil}}) \vdash_3 \downarrow$
if $x = \mathsf{nil}$ goto 8
$ls\,y\,\mathsf{nil} * \underline{ls\,x\,\mathsf{nil}} \vdash_2 \downarrow$

Figure 5. Termination of in-place list reversal

In TOY-C the program becomes

```
1:  y := nil;               5:  [z] := y;
2:  if x = nil goto 8;      6:  y := z;            (1)
3:  z := x;                 7:  goto 2;
4:  x := [x];               8:  stop
```

With precondition $ls\,x\,\mathsf{nil}$ the invariant of the loop (lines 2–8) is $ls\,y\,\mathsf{nil} * ls\,x\,\mathsf{nil}$, and the interesting problem is then $ls\,y\,\mathsf{nil} * ls\,x\,\mathsf{nil} \vdash_2 \downarrow$. The proof is shown in figure 5. (This and the next example are not completely formal: we have used '$*$' and '$\wedge$' rather than ',' and ';' in bunches, and we have omitted steps which reorganise bunches.)

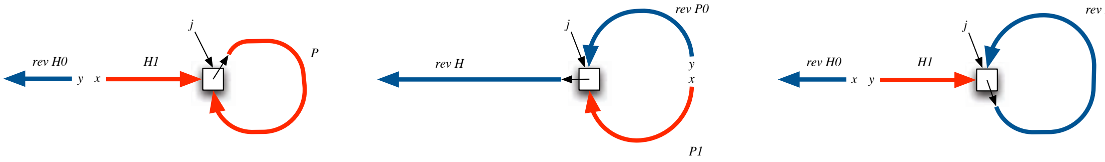
This proof could have used multiplicative weakening in place of cut, but we have chosen not to do so in order to emphasise the connection with the next example.

Example 6.3 (Termination of in-place "frying-pan list" reversal)**.** The previous example algorithm will reverse a cyclic list segment, but the loop measure, and hence the proof of termination, is tricky.

A cyclic list segment $ls\,x\,j$ in which the terminating pointer j points to a node already in the segment can be seen as a separated three-part structure of two acyclic list segments and a "join node", represented in separation logic as:

$$\exists k.(ls\,x\,j * j \mapsto k * ls\,k\,j) \tag{2}$$

Diagrammatically, such segments resemble a frying pan in which $ls\,x\,j$ is the handle and $ls\,k\,j$ is the pan. The reversal algorithm goes down the handle, reversing it until it reaches the join node, which it redirects towards the reversed handle; then it goes round the pan, reversing that; then it re-redirects the join node to the reversed pan; finally it comes back up the handle, re-reversing it.



(a) going in (b) going round (c) coming out

Figure 6. Stages of reversing a frying-pan list with handle H and pan P

The precondition is (2) and the invariant is:

$$\exists k1, k2, k3 \cdot \left(\begin{array}{l} (ls\, x\, j * ls\, y\, \mathsf{nil} * j \mapsto k1 * ls\, k1\, j) \vee \\ (ls\, k2\, \mathsf{nil} * j \mapsto k2 * ls\, x\, j * ls\, y\, j) \vee \\ (ls\, x\, \mathsf{nil} * ls\, y\, j * j \mapsto k3 * ls\, k3\, j) \end{array} \right) \quad (3)$$

in which each of the disjuncts corresponds directly to one of the pictures in figure 6.

This invariant is sufficiently obvious that it can be discovered automatically (Distefano et al. 2006), but it's hard to see what formula to use as a measure of the number of loop executions, because x plays different rôles at different stages of the proof. A termination proof might perhaps be hacked up using auxiliary variables to record the join point and the stage of the proof, but the proof stages and the join point are proof artefacts, and it should be unnecessary to reveal them.

Figure 7 shows a pre-proof of the judgement $I \vdash_2 \downarrow$, where I is the invariant (3) (with existential quantifications omitted for simplicity, and double-line steps indicating implicit additive weakening (WkA)). The cycles in the proof are:

B reversing the handle, progressing by unrolling $ls\, x\, j$;
D reversing the pan, progressing by unrolling $ls\, x\, j$;
E re-reversing the handle, progressing (like figure 5) by unrolling $ls\, x\, \mathsf{nil}$;
A redirect the join-node to the reversed handle, move to reverse the pan;
C redirect the join-node to the reversed pan, move to re-reverse the handle.

It looks more complicated than it is.

- The right-hand stack, which deals with the third disjunct of the invariant and the re-reversal of the handle, is a straight-line list reversal, extremely similar to figure 5, with 'stop' as the left antecedent of the if-goto step, and contradiction (because $x = \mathsf{nil} \wedge x \neq \mathsf{nil}$) the left alternative of the unrolling of $ls\, x\, \mathsf{nil}$;
- The left-hand stack, which deals with the first disjunct of the invariant and the original reversal of the handle, is also similar to figure 5, but in place of 'stop' it has contradiction (x can't be nil whilst $ls\, x\, j * j \mapsto k1$) and in place of contradiction it has a sequence of executions (when you exhaust $ls\, x\, j$ you reach the join-node $j \mapsto k1$, and one execution of the loop deals with that);
- The middle stack is very like the left-hand one.

There are no infinite paths in this proof which don't involve unrolling one or more predicates infinitely often. Hence we have termination. Incidentally we observe that the number of loop executions is now clear: it's exactly twice the length of the handle (cycles B and E) plus the length of the pan (cycle D) plus 2 (paths A and C).

The proof is long and tedious, but it involves no arithmetic and is very much the sort of thing an automatic tool (see e.g. Berdine et al. (2006a)) might be expected to do. One can imagine list processing in such a tool being so stylised that it could recognise the need to unroll the ls definitions to form the cycles B, D and E in the proof, and invent the empty definitions to form the paths A and C.

7. Conclusions and future work

In this paper we outline a novel approach to the problem of proving termination of imperative programs. Our approach builds upon previous theoretical work in cyclic proof (Brotherston 2007; Brotherston and Simpson 2007; Brotherston 2005; Sprenger and Dam 2003) and relies heavily upon separation logic techniques (Berdine et al. 2005; Reynolds 2002; Bornat et al. 2004). The infinite descent flavour of our cyclic proofs is highly reminiscent of Lee, Jones and Ben-Amran's size-change termination principle (Lee et al. 2001). However, since the soundness requirement for our proofs is based on unfolding an inductive definition infinitely often along every infinite path, we avoid the need to explicitly construct and reason with ranking functions. In the case of our termination proof for reversal of a frying-pan list (Example 6.3) we did not have to introduce auxiliary variables to represent phases of the algorithm or deal with the conditional measure function which would be needed to exploit knowledge of the phase (and then the phase annotation would require a lexicographical ranking function).

The Terminator and Mutant automated termination proving tools rely on a theoretical result concerning well-founded relations due to Podelski and Rybalchenko (Rybalchenko et al. 2006; Berdine et al. 2006b, 2007). The relationship between this principle and our cyclic proof principle is not yet clear to us. Nor does there seem to be a straightforward comparison with termination tools based on term rewriting (see e.g. Hofbauer and Serebrenik (2007)). However, we observe that our approach is amenable to interactive as well as automatic theorem proving, and we believe that the Smallfoot assertion checking tool for separation logic (Berdine et al. 2006a) is a promising candidate platform for implementing our approach. In general, any proof search mechanism for our formalism would need to extend the usual heuristics with the notion of searching for cycles (by identifying suitable companions elsewhere in the proof tree for the current buds, which correspond to unproven subgoals). The recent work on shape analysis for separation logic (see e.g. Guo et al. (2007); Lee et al. (2005); Distefano et al. (2006)) provides one obvious direction, based on abstract interpretation, with a finite domain built from separation logic formulas. Proof search combined with abstraction immediately gives a finite number of derivation trees, and cyclic pre-proofs for free. This immediately suggests an algorithm, and we can already see that it applies to the in-place list reversal program.

Thus far we have dealt only with small iterative algorithms. We already understand how to deal with iterative problems that normally require lexicographic measures; we have not yet considered more complex ranking functions (but we note that Lee and Ben-Amram (Ben-Amram and Lee 2007) have shown that measures

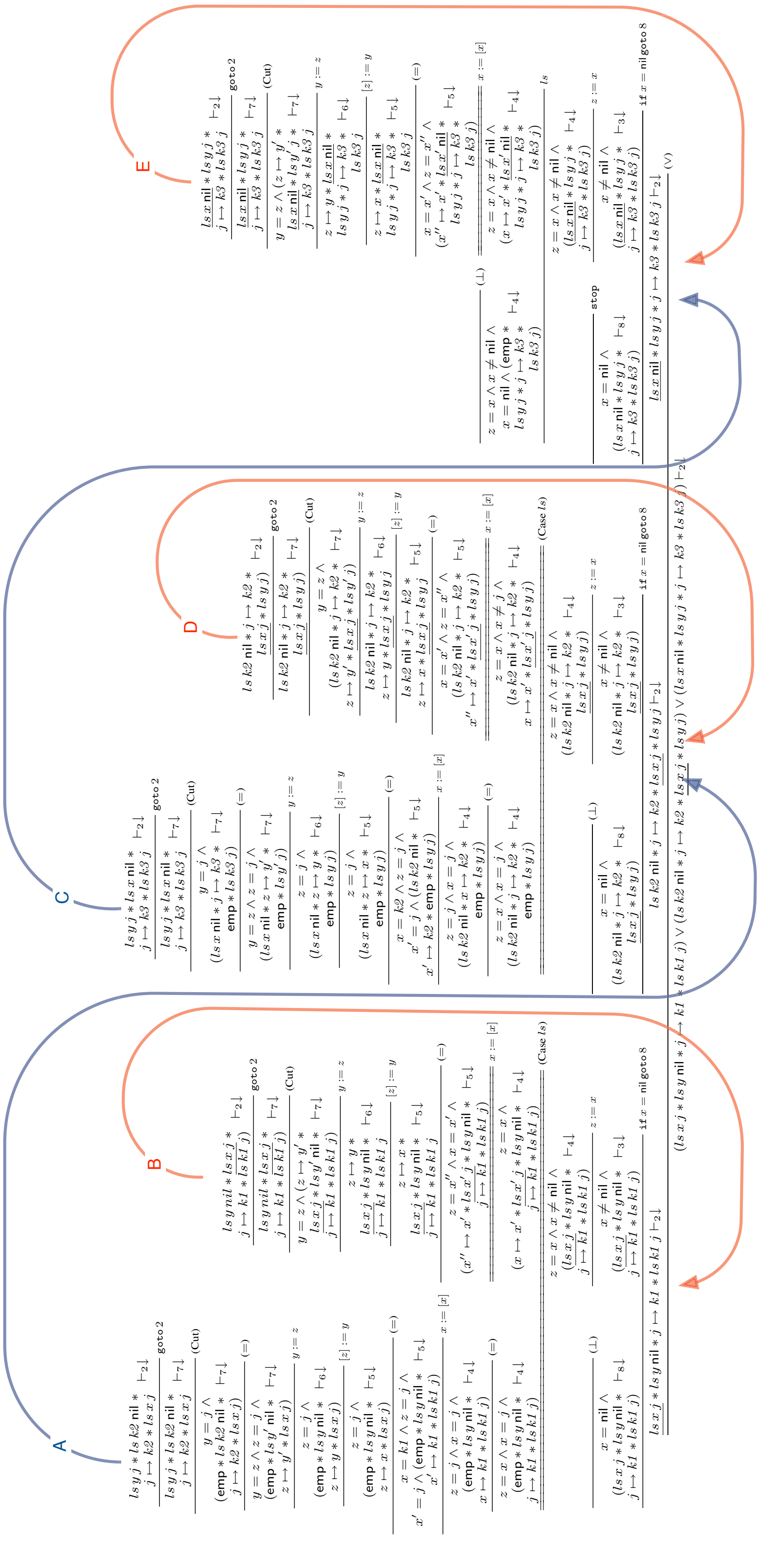


Figure 7. A termination proof of in-place reversal of a frying-pan list

need not be arbitrarily complicated in size-change transition problems). We intend to consider more difficult problems such as tree algorithms: to deal with recursive algorithms we will need at least to include postconditions in our termination judgements; to deal with iterative tree algorithms we will have to consider subtle termination measures.

We do not claim to have invented a panacea. We have uncovered a novel approach to termination which gives natural-seeming proofs which has already, in the frying-pan list example, simplified a previously difficult problem and which appears to have the potential to be extended in other directions.

References

Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. N-H, 1977.

Amir M. Ben-Amram and Chin Soon Lee. Ranking functions for size change termination II. Presented at (Hofbauer and Serebrenik 2007), 2007.

J. Berdine, C. Calcagno, and P.W. O'Hearn. Symbolic execution with separation logic. In *APLAS 2005*, volume 3780 of *LNCS*, pages 52–68, 2005.

J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Automatic modular assertion checking with separation logic. In *FMCO*, volume 4111 of *LNCS*, pages 115–137, 2006a.

J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV*, volume 4144 of *LNCS*, pages 386–400, 2006b.

J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *34th POPL*, 2007.

Richard Bornat, Cristiano Calcagno, and Peter O'Hearn. Local reasoning, separation and aliasing. In *SPACE* Workshop, 2004.

James Brotherston. Cyclic proofs for first-order logic with inductive definitions. In B. Beckert, editor, *TABLEAUX 2005*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005.

James Brotherston. Formalised inductive reasoning in the logic of bunched implications. In *SAS-14*, volume 4634 of *LNCS*, pages 87–103. Springer-Verlag, August 2007.

James Brotherston. *Sequent Calculus Proof Systems for Inductive Definitions*. PhD thesis, University of Edinburgh, November 2006.

James Brotherston and Alex Simpson. Complete sequent calculi for induction and infinite descent. In *LICS-22*, pages 51–60. IEEE Computer Society, July 2007.

C. Calcagno, D. Distefano, P.W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, volume 4134 of *LNCS*, pages 182–203, 2006.

D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920 of *LNCS*, pages 287–302, 2006.

Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *PLDI*, June 2007.

Dieter Hofbauer and Alexander Serebrenik. The 9th international workshop on termination. Paris, France, 2007.

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *28th POPL*, 2001.

O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, volume 3444 of *LNCS*, pages 124–140, 2005.

Per Martin-Löf. Haupstatz for the intuitionistic theory of iterated inductive definitions. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216. N-H, 1971.

P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

A. Rybalchenko, B. Cook, and A. Podelski. Termination proofs for systems code. In *PLDI*, pages 415–426, 2006.

Ulrich Schöpp and Alex Simpson. Verifying temporal properties using explicit approximants: Completeness for context-free processes. In *FoSSaCS 2002*, volume 2303 of *LNCS*, pages 372–386. Springer-Verlag, 2002.

Christoph Sprenger and Mads Dam. On the structure of inductive reasoning: circular and tree-shaped proofs in the μ-calculus. In *FOSSACS 2003*, volume 2620 of *LNCS*, pages 425–440, 2003.

N. Torp-Smith, L. Birkedal, and J. Reynolds. Local reasoning about a copying garbage collector. In *POPL*, pages 220–231, 2004.

H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *5th FOSSACS*, LNCS 2303, 2002.

A. Outline proof of soundness (Theorem 5.5)

In this section we give a sketch of the proof of our main soundness result: Theorem 5.5. The proof employs the following auxiliary definition:

Definition A.1. Let Γ be a bunch and let τ be the position of a distinguished leaf F_τ of Γ. We observe that Γ can be inductively defined (up to coherent equivalence $\equiv$) by the following grammar, where F ranges over formulas:

$$\begin{array}{rcl} \Delta & ::= & F \ \ (F \neq F_\tau) \mid \Delta;\Delta \mid \Delta,\Delta \\ \Gamma & ::= & F_\tau \mid \Gamma;\Delta \mid \Gamma,\Delta \end{array}$$

Now let s be a stack and h be a heap. We define the relation $\sim_s$ by induction on the structure of Γ (as given above) as follows:

$$\frac{s, h_0 \models F_\tau}{\langle h_0, h_0\rangle \sim_s \langle F_\tau, F_\tau\rangle} \qquad \frac{\langle h, h_0\rangle \sim_s \langle \Gamma, F_\tau\rangle \quad s, h \models \Delta}{\langle h, h_0\rangle \sim_s \langle (\Gamma;\Delta), F_\tau\rangle}$$

$$\frac{\langle h_1, h_0\rangle \sim_s \langle \Gamma, F_\tau\rangle \quad s, h_2 \models \Delta}{\langle h_1 \circ h_2, h_0\rangle \sim_s \langle (\Gamma,\Delta), F_\tau\rangle}$$

Intuitively, $\langle h, h'\rangle \sim_s \langle \Gamma, F_\tau\rangle$ holds iff s and h satisfy Γ, F_τ is a leaf of Γ and h' is the sub-heap of h that satisfies F_τ. In other words, the splitting of Γ into F_τ and a surrounding "bunch context" is mirrored by the splitting of h into h' and a surrounding "heap context".

Lemma A.2. *Each of the Hoare logic rules for termination judgements given in Figure 3 enjoy the following two properties:*

1. *if the conclusion of the rule, say $\Gamma \vdash_i\downarrow$, is invalid, i.e. there is some stack s and heap h such that $s, h \models \Gamma$ but $(i, s, h)\downarrow$ does not hold, then there is some premise $\Gamma' \vdash_{i'}\downarrow$ of the rule, a stack s' and a heap h' such that $s', h' \models F'$ but $(i', s', h')\downarrow$ does not hold;*
2. *if there is a trace (τ, τ') following the edge $(\Gamma \vdash_i\downarrow, \Gamma' \vdash_{i'}\downarrow)$ then, given a heap h_0 satisfying $\langle h, h_0\rangle \sim_s \langle \Gamma, F_\tau\rangle$, there exists a heap h'_0 such that $\langle h', h'_0\rangle \sim_{s'} \langle \Gamma', F_{\tau'}\rangle$. Furthermore, the following relation holds (and is well-defined):*

   least α s.t. $s, h_0 \models_{[\mathbf{P}\mapsto\mathbf{P}^\alpha]} F_\tau \geq$ least α s.t. $s', h'_0 \models_{[\mathbf{P}\mapsto\mathbf{P}^\alpha]} F_{\tau'}$

   where $\models_{[\mathbf{P}\mapsto\mathbf{P}^\alpha]}$ is the satisfaction relation defined as in Figure 2, except that for all $i \in \{1, \ldots, n\}$ we have $[\![P_i]\!] = P_i^\alpha$, i.e. each inductive predicate is interpreted using its αth approximant (cf. Definition 3.4). Furthermore, if (τ, τ') is a progressing trace, then this relation holds with $>$ in place of $\geq$.

Proof. (Sketch) We just need to check that both properties of the lemma hold for each proof rule. The first property is just one way of stating that the rules are locally sound, i.e. that falsifiability of the conclusion of a rule implies the falsifiability of one of its premises (cf. Proposition 4.5). For the second property, we need to show that if there is a trace following the edge from the conclusion to this falsifiable premise and $\langle h, h_0\rangle \sim_s \langle \Gamma, F_\tau\rangle$ holds, i.e. h_0 is the sub-heap of h used to satisfy F_τ in the falsifying interpretation of $\Gamma \vdash_i\downarrow$, then we can construct a suitable substate h'_0 of h' that can be used to satisfy $F_{\tau'}$ in the constructed falsifying interpretation of $\Gamma' \vdash_{i'}\downarrow$. The main interesting case is when the rule applied is a case-split rule (Case P) and (τ, τ') is a progressing trace, with the strict inequality relying on the fact that if the formula $P\mathbf{t}$ unfolded by the rule is satisfied by s and h_0, i.e. $(h_0, [\![\mathbf{t}]\!]s)$ is in some approximant P^α of P, then for every case-descendant $Q\mathbf{u}$ of $P\mathbf{t}$ we must have $(h'_0, [\![\mathbf{u}]\!]s)$ in some strictly smaller approximant $Q^{\beta<\alpha}$ of Q. □

Having proved the above lemma, concerning edges in a proof tree, we can straightforwardly extend the two properties of the lemma to cover paths in a pre-proof graph:

Lemma A.3. *Let $\mathcal{P}$ be a pre-proof of $\Gamma_0 \vdash_{i_0}\downarrow$ and suppose that $\Gamma_0 \vdash_{i_0}\downarrow$ is invalid. Then there exists an infinite path $(\Gamma_j \vdash_{i_j}\downarrow)_{j\geq 0}$ in $\mathcal{G}_\mathcal{P}$, a sequence $(s_j)_{j\geq 0}$ of stacks and a sequence $(h_j)_{j\geq 0}$ of heaps such that the following two properties hold:*

1. *for all $j \geq 0$, the judgement $\Gamma_j \vdash_{i_j}\downarrow$ is false with respect to the stack s_j and heap h_j;*
2. *if there is a trace $(\tau_j)_{j\geq m}$ following a tail $(\Gamma_j \vdash_{i_j}\downarrow)_{j\geq m}$ of the path $(\Gamma_j \vdash_{i_j}\downarrow)_{j\geq 0}$, then there exists a second sequence of heaps $(h'_j)_{j\geq m}$ such that, for all $j \geq m$:*

   $$\textit{least } \alpha \textit{ s.t. } s_j, h'_j \models_\alpha F_\tau \geq \textit{least } \alpha \textit{ s.t. } s_{j+1}, h'_{j+1} \models_\alpha F_{\tau'}$$

   Furthermore, if j is a progress point of the trace, then this relation holds with $>$ in place of $\geq$.

Proof. $\Gamma_0 \vdash_{i_0}\downarrow$, s_0 and h_0 are given by assumption. If we inductively assume that we have constructed $\Gamma_k \vdash_{i_k}\downarrow$, s_k and h_k, then property 1 of Lemma A.2 tells us that we can construct $\Gamma_{k+1} \vdash_{i_{k+1}}\downarrow$, s_{k+1} and h_{k+1}.

Now if we suppose that there is a trace following some tail $(\Gamma_j \vdash_{i_j}\downarrow)_{m\leq j\leq k+1}$ of the path constructed so far, property 2 of Lemma A.2 tells us that we can construct the required sequence $(h'_j)_{m\leq j\leq k+1}$. (It is easy to see how to construct the first element h'_m of this sequence because we have $\langle h_m, h'_m\rangle \sim_{s_m} \langle \Gamma_m, F_{\tau_m}\rangle$.) □

Proof of Theorem 5.5. If we suppose that $\Gamma \vdash_i\downarrow$ has a proof $\mathcal{P}$ but is invalid, i.e. false in some stack s and heap h, then we can use property 1 of Lemma A.3 to construct an infinite path π in $\mathcal{G}_\mathcal{P}$ together with a sequence of stacks and heaps that falsify each sequent along the path. Since $\mathcal{P}$ is a proof, there is an infinitely progressing trace following some tail of π. Thus we can invoke property 2 of Lemma A.3 to create a monotonically decreasing chain of ordinals which, since the trace progresses infinitely often, must decrease infinitely often. This contradicts the well-foundedness of the ordinals, so $\Gamma \vdash_i\downarrow$ must indeed be valid. □

B. Outline proof of Theorem 5.7 (via termination weakest preconditions)

In this section we give a sketch of the proof of Theorem 5.7. First, we present a construction that transforms a program into a family of mutually defined inductive predicates, which capture the weakest precondition for termination of the program. Then, we show that every valid termination judgement has a cyclic proof in our system, which uses the inductive predicates obtained from the program.

Consider a program $1 : C_1; \cdots ; n : C_n$, and let $\mathbf{x}$ be the variables occurring in the program. For each program point i, we define a corresponding inductive predicate $\textit{term}_i\,\mathbf{x}$ in Figure 8 (we use the notation $E{\mapsto}-$ as an abbreviation for $\exists x.\, E{\mapsto}x$). The result is a collection of predicates such that cycles in the definitions correspond directly to cycles in the control flow of the program.

Example B.1 (List deletion program)**.** The list deletion program:

$$\begin{array}{llll}
1: & \texttt{if}\, x = \mathsf{nil}\, \texttt{goto}\, 6; & 4: & \texttt{free}(t); \\
2: & t := x; & 5: & \texttt{goto}\, 1; \\
3: & x := [x]; & 6: & \texttt{stop};
\end{array}$$

gives the following definitions.

$$\begin{array}{rcl}
(x = \mathsf{nil} \wedge \textit{term}_6\, xt) & \Rightarrow & \textit{term}_1\, x\, t \\
(x \neq \mathsf{nil} \wedge \textit{term}_2\, x\, t) & \Rightarrow & \textit{term}_1\, x\, t \\
\textit{term}_3\, x\, x & \Rightarrow & \textit{term}_2\, x\, t \\
x \mapsto x' * ((x{\mapsto}x') \mathrel{-\!\!*} \textit{term}_4\, x'\, t) & \Rightarrow & \textit{term}_3\, x\, t \\
(t \mapsto -) * \textit{term}_5\, x\, t & \Rightarrow & \textit{term}_4\, x\, t \\
\textit{term}_1\, x\, t & \Rightarrow & \textit{term}_5\, x\, t \\
\top & \Rightarrow & \textit{term}_6\, x\, t
\end{array}$$

By applying simplifications we obtain a single inductive predicate for location 1. First notice that inlining $\textit{term}_3, \textit{term}_4, \textit{term}_5$ we obtain

$$x \mapsto x' * (x \mapsto x' \mathrel{-\!\!*} (x \mapsto - * \textit{term}_1\, x'\, x)) \Rightarrow \textit{term}_2\, x\, t$$

and the left-hand side can be simplified to $x \mapsto x' * \textit{term}_1\, x'\, x$. By further inlining and simplification, and noticing that the parameter t is not used actively in the definition, we obtain

$$\begin{array}{rcl}
x = \mathsf{nil} & \Rightarrow & \textit{term}_1\, x \\
x \mapsto x' * \textit{term}_1\, x' & \Rightarrow & \textit{term}_1\, x
\end{array}$$

which is exactly analogous to $ls\, x\, \mathsf{nil}$ except that we can have garbage in the heap, since garbage does not affect termination.

Example B.2 (In-place list reversal program)**.** Consider the in-place list reversal program presented in Example 6.2. The corresponding inductive definition for termination at program point 2 can be simplified to obtain the following definition:

$$\begin{array}{rcl}
x = \mathsf{nil} & \Rightarrow & \textit{term}_2\, x\, y \\
x \mapsto x' * (x \mapsto y \mathrel{-\!\!*} \textit{term}_2\, x'\, x) & \Rightarrow & \textit{term}_2\, x\, y
\end{array}$$

In this inductive definition we cannot eliminate $\mathrel{-\!\!*}$. Instead we can give a characterisation using another predicate. If we define the cyclic list predicate cl by:

$$ls\, x\, x' * x' \mapsto x'' * ls\, x''\, x' \quad \Rightarrow \quad cl\, x$$

then the following equivalence holds:

$$\textit{term}_2\, x\, y \leftrightarrow (ls\, x\, \mathsf{nil}) \vee (cl\, x * ls\, y\, \mathsf{nil})$$

which means that the program terminates either by traversing the acyclic list starting from x and ending in nil, or by traversing the cyclic list and then the acyclic list starting from y and ending in nil.

The following lemma shows that the $\textit{term}_i$ predicate indeed guarantees termination from program point i.

Lemma B.3. $(i, s, h)\downarrow$ *implies* $s, h \models \textit{term}_i\,\mathbf{x}$.

Proof. The proof is by induction on the length n of the longest computation (i.e., $\rightsquigarrow$-sequence) starting at (i, s, h). We show some cases; other cases are analogous.

Case $C_i \equiv x := E$**.** The computation proceeds with $(i + 1, s', h)$, where $s' = s[x \mapsto [\![E]\!]s]$. Then we have $(i + 1, s', h)\downarrow$ with longest computation of length $n - 1$. By induction hypothesis we have $s', h \models \textit{term}_{i+1}\,\mathbf{x}$, therefore $s, h \models \textit{term}_{i+1}\,(\mathbf{x}[E/x])$. By definition of $\textit{term}_i$ we have $s, h \models \textit{term}_i\,\mathbf{x}$, which concludes the case.

Command	**Inductive definition**
$C_i \equiv x := E$	$\mathit{term}_{i+1}(\mathbf{x}[E/x]) \Rightarrow \mathit{term}_i\,\mathbf{x}$
$C_i \equiv x := [E]$	$E \mapsto x' * (E \mapsto x' \mathrel{-\!\!*} \mathit{term}_{i+1}(\mathbf{x}[x'/x])) \Rightarrow \mathit{term}_i\,\mathbf{x}$
$C_i \equiv [E] := F$	$(E \mapsto -) * ((E \mapsto F) \mathrel{-\!\!*} \mathit{term}_{i+1}\,\mathbf{x}) \Rightarrow \mathit{term}_i\,\mathbf{x}$
$C_i \equiv x := \texttt{new}()$	$\forall x', y'.\,(x' \mapsto y') \mathrel{-\!\!*} \mathit{term}_{i+1}(\mathbf{x}[x'/x]) \Rightarrow \mathit{term}_i\,\mathbf{x}$
$C_i \equiv \texttt{free}(E)$	$(E \mapsto -) * \mathit{term}_{i+1}\,\mathbf{x} \Rightarrow \mathit{term}_i\,\mathbf{x}$
$C_i \equiv \texttt{if}\,\mathit{Cond}\,\texttt{goto}\,j$	$\left\{\begin{array}{l} \mathit{Cond} \wedge \mathit{term}_j\,\mathbf{x} \Rightarrow \mathit{term}_i\,\mathbf{x} \\ \neg\mathit{Cond} \wedge \mathit{term}_{i+1}\,\mathbf{x} \Rightarrow \mathit{term}_i\,\mathbf{x} \end{array}\right.$
$C_i \equiv \texttt{stop}$	$\top \Rightarrow \mathit{term}_i\,\mathbf{x}$

Figure 8. Transformation of commands to inductive predicates

Case $C_i \equiv x := \texttt{new}()$. We need to show $s, h \models \mathit{term}_i\,\mathbf{x}$, that is $s, h \models \forall x', y'.\,(x' \mapsto y') \mathrel{-\!\!*} \mathit{term}_{i+1}(\mathbf{x}[x'/x])$ for x', y' fresh in $\mathbf{x}$. Take $v_1, v_2 \in \mathsf{Val}$, and h' such that $s[x' \mapsto v_1, y' \mapsto v_2], h' \models (x' \mapsto y')$ and $h \circ h'$ is defined. Then $h' = [v_1 \mapsto v_2]$ and $v_1 \in \mathsf{Loc} \setminus \mathit{dom}(h)$. It remains to show $s[x' \mapsto v_1, y' \mapsto v_2], h[v_1 \mapsto v_2] \models \mathit{term}_{i+1}\,(\mathbf{x}[x'/x])$.

Now the computation can proceed with $(i+1, s[x \mapsto \ell], h[\ell \mapsto v])$ for any $\ell \in \mathsf{Loc}\setminus\mathit{dom}(h)$ and $v \in \mathsf{Val}$, in particular with $\ell = v_1$ and $v = v_2$. Then we have $(i+1, s[x \mapsto v_1], h[v_1 \mapsto v_2])\!\downarrow$ with longest computation of length $n-1$. By induction hypothesis we have $s[x \mapsto v_1], h[v_1 \mapsto v_2] \models \mathit{term}_{i+1}\,\mathbf{x}$, therefore $s[x' \mapsto v_1, y' \mapsto v_2], h[v_1 \mapsto v_2] \models \mathit{term}_{i+1}\,(\mathbf{x}[x'/x])$, as required. □

The following lemma shows how to construct a cyclic proof that term_i is a termination precondition for program point i, which will be fundamental for the completeness result. Together with soundness of cyclic proofs (Theorem 5.5) and Lemma B.3, this implies that term_i denotes the weakest precondition for termination at program point i.

Lemma B.4. *There is a cyclic proof of $\mathit{term}_i\,\mathbf{x} \vdash_i\!\downarrow$.*

Proof. We give a direct construction of the proof. For each command C_i in the program, let $j_1 \cdots j_k$ be the possible program points where the execution might continue after executing C_i. We show how to construct a derivation tree of the form

$$\dfrac{\dfrac{(\mathit{term}_m\,\mathbf{x} \vdash_m\!\downarrow)_{m=j_1\cdots j_k}}{\cdots}}{\mathit{term}_i\,\mathbf{x} \vdash_i\!\downarrow}$$

which admits a progressing trace from $\mathit{term}_i\,\mathbf{x} \vdash_i\!\downarrow$ to each $\mathit{term}_m\,\mathbf{x} \vdash_m\!\downarrow$. The whole proof is then obtained by stacking the appropriate derivation tree on top of each bud, unless that bud is already matched with a companion in the derivation tree already constructed. At the end of the process, each bud will be assigned a companion, and every infinite path in the resulting graph will obviously progress infinitely often. We show in detail the construction for some interesting cases:

Case $C_i \equiv x := E$. We have the following inductive definition for $\mathit{term}_i\,\mathbf{x}$:

$$\mathit{term}_{i+1}(\mathbf{x}[E/x]) \Rightarrow \mathit{term}_i\,\mathbf{x}$$

We can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\underline{\mathit{term}_{i+1}\,\mathbf{x}} \vdash_{i+1}\!\downarrow}{x = E[x'/x]\,;\,\underline{\mathit{term}_{i+1}(\mathbf{x})} \vdash_{i+1}\!\downarrow}\,\text{(WkA)}}{x = E[x'/x]\,;\,\underline{\mathit{term}_{i+1}(\mathbf{x}[E[x'/x]/x])} \vdash_{i+1}\!\downarrow}\,(=)}{\underline{\mathit{term}_{i+1}(\mathbf{x}[E/x])} \vdash_i\!\downarrow}\,x := E}{\underline{\mathit{term}_i\,\mathbf{x}} \vdash_i\!\downarrow}\,\text{(Case } \mathit{term}_i\text{)}$$

Case $C_i \equiv x := \texttt{new}()$. We have the following inductive definition for $\mathit{term}_i\,\mathbf{x}$:

$$\forall x', y'.\,(x' \mapsto y') \mathrel{-\!\!*} \mathit{term}_{i+1}\,(\mathbf{x}[x'/x]) \Rightarrow \mathit{term}_i\,\mathbf{x}$$

We can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\underline{\mathit{term}_{i+1}\,\mathbf{x}} \vdash_{i+1}\!\downarrow}{x \mapsto x'', ((\underline{x \mapsto x'') \mathrel{-\!\!*} \mathit{term}_{i+1}\,(\mathbf{x}[x'/x][x/x'])}) \vdash_i\!\downarrow}\,(\mathrel{-\!\!*})}{x \mapsto x'', (\underline{\forall x', y'.\,(x' \mapsto y') \mathrel{-\!\!*} \mathit{term}_{i+1}\,(\mathbf{x}[x'/x])}) \vdash_{i+1}\!\downarrow}\,(\forall)}{\underline{\forall x', y'.\,(x' \mapsto y') \mathrel{-\!\!*} \mathit{term}_{i+1}\,(\mathbf{x}[x'/x])} \vdash_i\!\downarrow}\,x := \texttt{new}()}{\underline{\mathit{term}_i\,\mathbf{x}} \vdash_i\!\downarrow}\,\text{(Case } \mathit{term}_i\text{)}$$

Case $C_i \equiv \texttt{if}\,\mathit{Cond}\,\texttt{goto}\,j$. We have the following inductive definition for $\mathit{term}_i\,\mathbf{x}$:

$$\begin{array}{c} \mathit{Cond} \wedge \mathit{term}_j\,\mathbf{x} \Rightarrow \mathit{term}_i\,\mathbf{x} \\ \neg\mathit{Cond} \wedge \mathit{term}_{i+1}\,\mathbf{x} \Rightarrow \mathit{term}_i\,\mathbf{x} \end{array}$$

We can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\underline{\mathit{term}_j\,\mathbf{x}} \vdash_j\!\downarrow}{\mathit{Cond}; \underline{\mathit{term}_j\,\mathbf{x}} \vdash_j\!\downarrow}\,\text{(WkA)}}{\mathit{Cond}; \underline{\mathit{term}_j\,\mathbf{x}} \vdash_i\!\downarrow}\,\texttt{if}}{\underline{\mathit{Cond} \wedge \mathit{term}_j\,\mathbf{x}} \vdash_i\!\downarrow}\,(\wedge) \qquad \dfrac{\dfrac{\dfrac{\underline{\mathit{term}_{i+1}\,\mathbf{x}} \vdash_{i+1}\!\downarrow}{\neg\mathit{Cond}; \underline{\mathit{term}_{i+1}\,\mathbf{x}} \vdash_{i+1}\!\downarrow}\,\text{(WkA)}}{\neg\mathit{Cond}; \underline{\mathit{term}_{i+1}\,\mathbf{x}} \vdash_i\!\downarrow}\,\texttt{if}}{\underline{\neg\mathit{Cond} \wedge \mathit{term}_{i+1}\,\mathbf{x}} \vdash_i\!\downarrow}\,(\wedge)}{\underline{\mathit{term}_i\,\mathbf{x}} \vdash_i\!\downarrow}\,\text{(Case } \mathit{term}_i\text{)}$$

□

Proof of Theorem 5.7. Assume that $\Gamma \vdash_i\!\downarrow$ is valid. For any $s \in \mathsf{Stacks}$ and $h \in \mathsf{Heaps}$ such that $s, h \models \Gamma$, we have $(i, s, h)\!\downarrow$, hence $s, h \models \mathit{term}_i\,\mathbf{x}$ by Lemma B.3. Therefore $\Gamma \vdash \mathit{term}_i\,\mathbf{x}$ is valid, and it is derivable by the completeness assumption of the underlying proof system.

The required proof of $\Gamma \vdash_i\!\downarrow$ can be constructed using an instance of Cut as follows

$$\dfrac{\begin{array}{c} \cdots \\ \vdots \\ \mathit{term}_i\,\mathbf{x} \vdash_i\!\downarrow \end{array} \qquad \Gamma \vdash \mathit{term}_i\,\mathbf{x}}{\Gamma \vdash_i\!\downarrow}\,\text{(Cut)}$$

where the dots are a placeholder for the proof of $\mathit{term}_i\,\mathbf{x} \vdash_i\!\downarrow$ from Lemma B.4. □