

# Footprints in Local Reasoning

Mohammad Raza and Philippa Gardner

Department of Computing,  
Imperial College London, 180 Queen’s Gate, London SW7 2AZ, UK  
{mraza, pg}@doc.ic.ac.uk

**Abstract.** Local reasoning about programs exploits the natural local behaviour common in programs by focussing on the footprint - that part of the resource accessed by the program. We address the problem of formally characterising and analysing the footprint notion for abstract local functions introduced by Calcagno, O’Hearn and Yang. With our definition, we prove that the footprints are the only essential elements required for a complete specification of a local function. We formalise the notion of small specifications in local reasoning and show that for well-founded resource models, a smallest specification always exists that only includes the footprints, and also present results for the non-well-founded case. Finally, we investigate the conditions under which the footprints correspond to the smallest safe states, and present a heap model in which, unlike the standard model, this is the case for every program.

**Key words:** Footprints, Hoare Logic, Local Reasoning, Separation Logic

## 1 Introduction

Local reasoning about programs focusses on the collection of resources directly acted upon by the program. It has recently been introduced and used to substantial effect in *local* Hoare reasoning about memory update. Researchers previously used Hoare reasoning based on First-order Logic to specify how programs interacted with the *whole* memory. O’Hearn, Reynolds and Yang instead introduced local Hoare reasoning based on Separation Logic [14, 11]. The idea is to reason only about the local parts of the memory—the *footprints*—that are directly accessed by a program. Intuitively, the footprints form the pre-conditions of the *small* axioms, which provide the smallest complete specification of the program. All the true Hoare triples are derivable from the small axioms and the general Hoare rules. In particular, the *frame rule* extends the reasoning to properties about the rest of the heap which has not been changed by the command.

O’Hearn, Reynolds and Yang originally introduced Separation Logic to solve the problem of how to reason about the mutation of data structures in memory. They have applied their reasoning to several memory models, including heaps based on pointer arithmetic [14], heaps with permissions [4], and the combination of heaps with variable stacks which views variables as resource [5, 17]. In each case, the basic soundness and completeness results for local Hoare reasoning are essentially the same. For this reason, Calcagno, O’Hearn and Yang [9] recently introduced abstract local functions over abstract resource models (separation algebras), generalising their specific examples of

local imperative commands for manipulating memory models. They develop Abstract Separation Logic to provide local Hoare reasoning about such local functions, and give general soundness and completeness results.

We believe that the general concept of a local function is a fundamental step towards establishing the theoretical foundations of local reasoning, and Abstract Separation Logic is an important generalisation of the local Hoare reasoning systems now widely studied in the literature. However, Calcagno, O’Hearn and Yang do not characterise the footprints and small axioms in this general theory, which is a significant omission. O’Hearn, Reynolds and Yang, in one of their first papers on the subject [14], state the local reasoning viewpoint as:

‘to understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged.’

A complete understanding of the foundations of local Hoare reasoning therefore requires a formal characterisation of the footprint notion. O’Hearn tried to formalise footprints in his work on Separation Logic (personal communication with O’Hearn). His intuition was that the footprints should be the smallest states on which the program is safe - the *access footprint*, and that the *small axioms* arising from these footprints should give rise to a complete specification using the general rules for local Hoare reasoning. However, Yang discovered this notion of footprint does not work, since it does not always yield complete specifications. In this paper, we resolve this problem, providing a definition of footprint which does give rise to complete specifications.

Consider the local program<sup>1</sup>

$$AD ::= x := new(); dispose(x)$$

This *allocate-deallocate* program allocates a new cell, stores its address value in the stack variable  $x$ , and then deallocates the cell. It is local because all its atomic constituents are local. This tiny example captures the essence of a common type of program; there are many programs which, for example, create a list, work on the list, and then destroy the list.

The smallest heap on which the  $AD$  program is safe is the empty heap  $emp$ . The specification using this pre-condition is:

$$\{emp\} AD \{emp\} \tag{1}$$

We can extend our reasoning to larger heaps by applying the frame rule: for example, extending to a one-cell heap with arbitrary address  $l$  and value  $v$  gives

$$\{l \mapsto v\} AD \{l \mapsto v\} \tag{2}$$

---

<sup>1</sup> Yang’s example was the ‘allocate-deallocate-test’ program  $ADT ::= 'x := new(); dispose(x); if (x=1) then z:=0 else z:=1;x=0'$ . Our  $AD$  program provides a more standard example of program behaviour.

However, axiom (1) does not give the complete specification of the *AD* program. In fact, it captures very little of the spirit of allocation followed by de-allocation. For example, the following triple is also true:

$$\{l \mapsto v\} \quad AD \quad \{l \mapsto v \wedge x \neq l\} \quad (3)$$

This triple (3) is true because, if  $l$  is already allocated, then the new address cannot be  $l$  and hence  $x$  cannot be  $l$ . It cannot be derived from (1). However, the combination of axiom (1) and axiom (3) for arbitrary one-cell heaps does provide the smallest complete specification. This example illustrates that O’Hearn’s intuitive view of the footprints as the minimal safe states just does not work for common imperative programs.

In this paper, we introduce the definition of the footprint of a local function. For our *AD* example, our definition identifies *emp* and the arbitrary one-cell heaps  $l \mapsto v$  as footprints, as expected. We prove the general result that, for any local function, the footprints are the only elements which are *essential* to specify completely the behaviour of this function.

We then investigate the question of *sufficiency*. For well-founded resource, we show that the footprints are also always sufficient: that is, a complete specification always exists that only uses the footprints. We also explore results for non-well-founded resource. For models without negativity (no inverse elements except the identity), such as heaps with permissions, either the footprints are sufficient (such as for the *write* command in the permissions model) or there is no smallest complete specification (such as for the *read* command in the permissions model). For models with negativity, we show that there can exist smallest complete specifications based on elements which are not essential and hence not footprints.

In the final section, we reflect on the *AD* problem in light of our analysis of footprints. We address a question that arose from discussions with O’Hearn and Yang, which is whether there is an alternative model of heaps in which the access footprint does correspond to the actual footprint. We present such a model and also characterise a general condition for arbitrary models under which the footprint of *every* program is exactly the access footprint.

**Acknowledgements** We thank Calcagno, O’Hearn and Yang for detailed discussions on footprints. Raza acknowledges support of an ORS award. Gardner acknowledges support of a Microsoft/Royal Academy of Engineering Senior Research Fellowship.

## 2 Background

This is a background section that describes the separation algebras, local functions and Hoare reasoning introduced in [9]. Further details and motivation can be found in [9].

### 2.1 Separation Algebras and Local Functions

**Definition 1 (Separation Algebra).** A **separation algebra** is a cancellative, partial commutative monoid  $(\Sigma, \bullet, u)$ .  $\Sigma$  is a set, and  $\bullet$  is a partial binary operator with unit  $u$  which satisfies the familiar axioms of associativity, commutativity and unit, using

a partial equality on  $\Sigma$  where either both sides are defined and equal, or both are undefined. It also satisfies the cancellative property stating that, for each  $\sigma \in \Sigma$ , the partial function  $\sigma \bullet (\cdot) : \Sigma \mapsto \Sigma$  is injective. **Separateness** ( $\#$ ) and **substate** ( $\preceq$ ) relations are given by  $\sigma_0 \# \sigma_1$  iff  $\sigma_0 \bullet \sigma_1$  is defined and  $\sigma_0 \preceq \sigma_2$  iff  $\exists \sigma_1. \sigma_2 = \sigma_0 \bullet \sigma_1$ .

Examples of separation algebras include multisets under union (with unit  $\emptyset$ ), the natural numbers with addition (with unit 0), heaps as finite partial functions from locations to values ([9] and example 1), heaps with permissions [9,4], and the combination of heaps and variable stacks enabling us to model programs with variables as local functions ([9], [17] and example 1). These examples all have an intuition of resource, with  $\sigma_1 \bullet \sigma_2$  intuitively giving more resource than just  $\sigma_1$  and  $\sigma_2$  for  $\sigma_1, \sigma_2 \neq u$ . However, there are also examples that do not conform to this resource intuition, such as  $\{a, u\}$  with  $a \bullet a = u$ . We shall overload notation, using  $\Sigma$  to denote  $(\Sigma, \bullet, u)$ .

**Lemma 1 (Subtraction).** For  $\sigma_1, \sigma_2 \in \Sigma$ , if  $\sigma_1 \preceq \sigma_2$  then there exists a unique element  $\sigma_2 - \sigma_1 \in \Sigma$  such that  $(\sigma_2 - \sigma_1) \bullet \sigma_1 = \sigma_2$ .

Following [9], we model commands on separation algebras as functions of the form  $f : \Sigma \rightarrow P(\Sigma)^\top$ , where  $\top$  is an extra top element added to the powerset. The range  $P(\Sigma)^\top$  is used to model non-determinism and faulting: elements can map either to a set of elements (to allow for non-determinism) or  $\top$  (which represents faulting and returning an error). Mapping to the empty set represents divergence (non-termination).

**Definition 2.** Define the set  $P(\Sigma)^\top = P(\Sigma) \cup \{\top\}$  with the standard subset relation extended with a new greatest element  $\top$ : that is,  $p \sqsubseteq \top$  for all  $p \in P(\Sigma)^\top$ . It is a total commutative monoid with  $\{u\}$  as the unit and a binary operator  $*$  given by:

$$\begin{aligned} p * q &= \{\sigma_0 \bullet \sigma_1 \mid \sigma_0 \# \sigma_1 \wedge \sigma_0 \in p \wedge \sigma_1 \in q\} \quad \text{if } p, q \in P(\Sigma) \\ &= \top \quad \text{otherwise} \end{aligned}$$

For functions  $f : \Sigma \rightarrow P(\Sigma)^\top$ ,  $f \sqsubseteq g$  iff  $f(\sigma) \sqsubseteq g(\sigma)$  for all  $\sigma \in \Sigma$ .

Intuitively, we think of a command acting on resource to be *local* if whenever the command executes safely on any resource element, then any more resource that may be added will not be affected by the command. This intuition was first formalised in [21] (for the RAM model) with the following constraints:

- *Safety monotonicity*: if the command is safe on some resource element, then it does not fault when more resource is added.
- *Frame property*: if the command is safe on some resource element, then any additional resource will remain unchanged after execution if the execution terminates.

In the abstract setting of [9] which we use in this paper, these two restrictions were amalgamated into the following succinct definition of a local function.

**Definition 3 (Local Function).** A local function on  $\Sigma$  is a total function  $f : \Sigma \rightarrow P(\Sigma)^\top$  which satisfies the **locality condition**:

$$\sigma \# \sigma' \text{ implies } f(\sigma' \bullet \sigma) \sqsubseteq \{\sigma'\} * f(\sigma)$$

$f$  faults on  $\sigma$  when it is not sufficient resource for safe execution ( $f(\sigma) = \top$ ). Adding more resource may make the execution safe ( $f(\sigma' \bullet \sigma) \sqsubseteq f(\sigma) = \top$ ). Safety monotonicity follows since if  $f$  is safe on  $\sigma$  ( $f(\sigma) \sqsubset \top$ ) then  $f(\sigma' \bullet \sigma)$  is also safe ( $f(\sigma' \bullet \sigma) \sqsubseteq \{\sigma'\} * f(\sigma) \sqsubset \top$ ). The frame property follows by the fact that  $\sigma'$  is preserved by  $f$ .

We let  $LocFunc$  be the set of local functions on  $\Sigma$ . It can be checked that locality is preserved under sequential composition, defined in [9] as

$$(f;g)(\sigma) = \begin{cases} \top & \text{if } f(\sigma) = \top \\ \bigsqcup \{g(\sigma') \mid \sigma' \in f(\sigma)\} & \text{otherwise} \end{cases}$$

*Example 1 (Separation algebras and local functions).*

1. **Plain heap model.** We can model heaps by the separation algebra  $(H, \bullet, u_H)$ , where  $H = L \rightarrow_{fin} Val$  are finite partial functions from a set of locations to a set of values, the partial operator  $\bullet$  is the union of partial functions with disjoint domains, and the unit  $u_H$  is the empty function. For  $h \in H$ , let  $dom(h)$  be the domain of  $h$ . We write  $l \mapsto v$  for the partial function with domain  $\{l\}$  that maps  $l$  to  $v$ . For  $h_1, h_2 \in H$ , if  $h_2 \preceq h_1$  then  $h_1 - h_2 = h_1 \upharpoonright_{dom(h_1) - dom(h_2)}$ , and is undefined otherwise. An example of a local function is the  $dispose[l]$  command that deletes the cell at location  $l$ :

$$dispose[l](h) = \begin{cases} \{h - (l \mapsto v)\} & h \succeq (l \mapsto v) \\ \top & \text{otherwise} \end{cases}$$

The function is local: if  $h \not\succeq (l \mapsto v)$  then  $dispose[l](h) = \top$ , and  $dispose[l](h' \bullet h) \sqsubseteq \top$ . Otherwise,  $dispose[l](h' \bullet h) = \{(h' \bullet h) - (l \mapsto v)\} \sqsubseteq \{h'\} * \{h - (l \mapsto v)\} = \{h'\} * dispose[l](h)$ .

2. **Heap and stack.** We can incorporate the variable stack into the model by using the set  $H = L \cup Var \rightarrow_{fin} Val$ , where  $L$  and  $Val$  are as before, and  $Var$  is the set of stack variables  $\{x, y, z, \dots\}$ . The  $\bullet$  operator as before combines states with disjoint domains, and is undefined otherwise. The unit  $u_H$  is the empty state, where both heap and stack are empty. Although this approach is limited to disjoint reference to stack variables, this constraint can be lifted by enriching the separation algebra with *permissions* [4]. However, this added complexity can be avoided for the discussion in this paper. For a state  $h \in H$ , we let  $loc(h)$  and  $var(h)$  denote the set of heap locations and stack variables in  $h$  respectively. In this model we can define the allocation and deallocation commands as

$$new[x](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet l \mapsto w \mid w \in Val, l \in L \setminus loc(h')\} & h = h' \bullet x \mapsto v \\ \top & \text{otherwise} \end{cases}$$

$$dispose[x](h) = \begin{cases} \{h' \bullet x \mapsto l\} & h = h' \bullet x \mapsto l \bullet l \mapsto v \\ \top & \text{otherwise} \end{cases}$$

Commands for heap mutation and lookup can be defined as

$$mutate[x, v](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet l \mapsto v\} & h = h' \bullet x \mapsto l \bullet l \mapsto w \\ \top & \text{otherwise} \end{cases}$$

$$\text{lookup}[x, y](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet l \mapsto v \bullet y \mapsto v\} & h = h' \bullet x \mapsto l \bullet l \mapsto v \bullet y \mapsto v \\ \top & \text{otherwise} \end{cases}$$

The *AD* command described in the introduction, which is the sequential composition  $\text{new}[x]; \text{dispose}[x]$ , corresponds to the following local function

$$\text{AD}(h) = \begin{cases} \{h' \bullet x \mapsto w \mid w \notin \text{loc}(h')\} & h = h' \bullet x \mapsto v \\ \top & \text{otherwise} \end{cases}$$

3. **Integers.** We consider the separation algebra of integers under addition with identity 0. It can be seen that any ‘adding’ function  $f(x) = \{x+c\}$  that adds a constant  $c$  is local, while a function that multiplies by a constant  $c$ ,  $f(x) = \{cx\}$ , is non-local.

## 2.2 Predicates, Specifications and Local Hoare Reasoning

We now present the reasoning framework for local functions on separation algebras. Predicates are treated simply as subsets of the separation algebra.

**Definition 4.** A predicate  $p$  over  $\Sigma$  is an element of the powerset  $P(\Sigma)$ .

Note that the top element  $\top$  is not a predicate and that the  $*$  operator, although defined on  $P(\Sigma)^\top \times P(\Sigma)^\top \rightarrow P(\Sigma)^\top$ , acts as a binary connective on predicates. We have the distributive law for union:  $(\bigsqcup X) * p = \bigsqcup \{x * p \mid x \in X\}$  where  $X \subseteq P(\Sigma)$ . The same is not true for intersection in general, but does hold for *precise* predicates.

**Definition 5 (Precise predicate).** A predicate  $p \in P(\Sigma)$  is **precise** iff, for every  $\sigma \in \Sigma$ , there exists at most one  $\sigma_p \in p$  such that  $\sigma_p \preceq \sigma$ .

$\{l \mapsto v \mid v \in \text{Val}\}$  for some  $l$  is precise, while  $\{l \mapsto v \mid l \in L\}$  for some  $v$  is not. Also, any singleton predicate  $\{\sigma\}$  is precise.

**Lemma 2 (Precision characterization).** A predicate  $p$  is precise iff, for all  $X \subseteq P(\Sigma)$ ,  $(\prod X) * p = \prod \{x * p \mid x \in X\}$

Our Hoare reasoning system is a slight adaptation of Abstract Separation Logic [9], the difference being that we emphasise the notion of a specification as a tuple of pre- and post- conditions, rather than the usual Hoare triples that include the function. A triple is then equivalent to saying that a function  $f$  *satisfies* a tuple  $(p, q)$ , written  $f \models (p, q)$ . This approach is very similar to the notion of the *specification statement* (a Hoare triple with a ‘hole’) introduced in [12], which is used in refinement calculi, and was also used to prove completeness of a local reasoning system in [21].

**Definition 6 (Specification).** Let  $\Sigma$  be a separation algebra. A **statement** on  $\Sigma$  is a tuple  $(p, q)$ , where  $p, q \in P(\Sigma)$  are predicates representing pre- and post- conditions. A **specification**  $\phi$  on  $\Sigma$  is a set of statements. We let  $\Phi_\Sigma = P(P(\Sigma) \times P(\Sigma))$  be the set of all specifications on  $\Sigma$ . We shall exclude the subscript when it is clear from the context. The **domain** of a specification is defined as  $D(\phi) = \bigsqcup \{p \mid (p, q) \in \phi\}$ . **Domain equivalence** is defined as  $\phi \cong_D \psi$  iff  $D(\phi) = D(\psi)$ .

Thus the domain is the union of the preconditions of all the statements in the specification. It is one possible measure of *size*: how much of  $\Sigma$  the specification is referring to. We also adapt the notions of precise and saturated predicates to specifications.

**Definition 7.** *A specification is saturated iff its domain is saturated. It is precise iff its domain is precise.*

**Definition 8 (Satisfaction).** *A local function  $f$  satisfies a statement  $(p, q)$ , written  $f \models (p, q)$ , iff, for all  $\sigma \in p$ ,  $f(\sigma) \sqsubseteq q$ .  $f$  satisfies a specification  $\phi \in \Phi$ , written  $f \models \phi$ , iff  $f \models (p, q)$  for all  $(p, q) \in \phi$ .*

**Definition 9 (Semantic consequence).** *Let  $p, q, r, s \in P(\Sigma)$  and  $\phi, \psi \in \Phi$ . Each judgement  $(p, q) \models (r, s)$ ,  $\phi \models (p, q)$ ,  $(p, q) \models \phi$ , and  $\phi \models \psi$  holds iff all local functions that satisfy the left hand side also satisfy the right hand side.*

**Proposition 1 (Order Characterization).**  *$f \sqsubseteq g$  iff, for all  $p, q \in P(\Sigma)$ ,  $g \models (p, q)$  implies  $f \models (p, q)$ .*

For every specification  $\phi$ , there is a ‘best’ local function satisfying  $\phi$  (lemma 3). This is of the same nature as the best local action of [9], except that we generalise to specifications rather than statements (single pre- and post-condition pairs).

**Definition 10 (Best local action).** *For a specification  $\phi \in \Phi$ , the best local action of  $\phi$ , written  $bla[\phi]$ , is the function of type  $\Sigma \rightarrow P(\Sigma)^\top$  defined by*

$$bla[\phi](\sigma) = \bigsqcap_{\sigma' \preceq \sigma, \sigma' \in p, (p, q) \in \phi} \{\sigma - \sigma'\} * q$$

**Lemma 3.** *Let  $\phi \in \Phi$ . The following hold:*

- $bla[\phi]$  is local
- $bla[\phi] \models \phi$
- if  $f$  is local and  $f \models \phi$  then  $f \sqsubseteq bla[\phi]$

**Proof:** *To show that  $bla[\phi]$  is local, consider  $\sigma_1, \sigma_2$  such that  $\sigma_1 \# \sigma_2$ . We then calculate*

$$\begin{aligned} & bla[\phi](\sigma_1 \bullet \sigma_2) \\ &= \bigsqcap_{(p, q) \in \phi} \bigsqcap \{ \{\sigma'\} * q \mid \sigma_1 \bullet \sigma_2 = \sigma' \bullet \sigma'', \sigma'' \in p \} \\ &\sqsubseteq \bigsqcap_{(p, q) \in \phi} \bigsqcap \{ \{\sigma_1 \bullet \sigma'\} * q \mid \sigma_2 = \sigma' \bullet \sigma'', \sigma'' \in p \} \\ &= \bigsqcap_{(p, q) \in \phi} \bigsqcap \{ \{\sigma_1\} * \{\sigma'\} * q \mid \sigma_2 = \sigma' \bullet \sigma'', \sigma'' \in p \} \\ &= \{\sigma_1\} * \bigsqcap_{(p, q) \in \phi} \bigsqcap \{ \{\sigma'\} * q \mid \sigma_2 = \sigma' \bullet \sigma'', \sigma'' \in p \} \\ &= \{\sigma_1\} * bla[\phi](\sigma_2) \end{aligned}$$

*In the second-last step we used that  $\{\sigma_1\}$  is precise (2).*

$\frac{(p, q)}{(p * r, q * r)}$	$\frac{p' \sqsubseteq p \quad (p, q) \quad q \sqsubseteq q'}{(p', q')}$	$\frac{(p_i, q_i), \text{ all } i \in I}{(\bigsqcup_{i \in I} p_i, \bigsqcup_{i \in I} q_i)}$	$\frac{(p_i, q_i), \text{ all } i \in I, I \neq \emptyset}{(\prod_{i \in I} p_i, \prod_{i \in I} q_i)}$
<i>Frame</i>	<i>Consequence</i>	<i>Union</i>	<i>Intersection</i>

**Fig. 1.** Inference Rules for local Hoare reasoning

To show that  $\text{bla}[\phi]$  satisfies  $\phi$ , consider any  $(p, q) \in \phi$  and  $\sigma \in p$ . Then  $\text{bla}[\phi](\sigma) \sqsubseteq \{u\} * q = q$ .

For the last point, suppose  $f \models \phi$  and  $f$  is local. Then for any  $\sigma$  such that  $\sigma = \sigma_1 \bullet \sigma_2$  and  $\sigma_2 \in p$  and  $(p, q) \in \phi$ ,

$$\begin{aligned} f(\sigma) &= f(\sigma_1 \bullet \sigma_2) \\ &\sqsubseteq \{\sigma_1\} * f(\sigma_2) \\ &\sqsubseteq \{\sigma_1\} * q \end{aligned}$$

Thus  $f(\sigma) \sqsubseteq \text{bla}[\phi](\sigma)$ .

In the case that  $\nexists \sigma_1, \sigma_2$  such that  $\sigma = \sigma_1 \bullet \sigma_2$  and  $\sigma_2 \in D(\phi)$ , then

$$\begin{aligned} \text{bla}[\phi](\sigma) &= \prod \emptyset \\ &= \top \end{aligned}$$

So in this case also  $f(\sigma) \sqsubseteq \text{bla}[\phi](\sigma)$ . ■

**Lemma 4.** For  $\phi \in \Phi$  and  $p, q \in P(\Sigma)$ ,  $\text{bla}[\phi] \models (p, q) \Leftrightarrow \phi \models (p, q)$ .

**Proof:**

$$\begin{aligned} &\text{bla}[\phi] \models (p, q) \\ \Leftrightarrow &\forall f : \Sigma \rightarrow P(\Sigma)^\top. f \models \phi \Rightarrow f \models (p, q) \quad (\text{by lemma 3}) \\ \Leftrightarrow &\phi \models (p, q) \quad (\text{by definition 9}). \end{aligned}$$

■

The inference rules of the proof system are given in figure 1, and the system is sound and complete with respect to the satisfaction relation.

**Definition 11 (Proof-theoretic consequence).** For statements  $p, q, r, s$  and specifications  $\phi, \psi$ , each of the judgements  $(p, q) \vdash (r, s)$ ,  $\phi \vdash (p, q)$ ,  $(p, q) \vdash \phi$ , and  $\phi \vdash \psi$  holds iff the right-hand side is derivable from the left-hand side by the rules in figure 1.

**Theorem 1 (Soundness and Completeness).**  $\phi \models (p, q) \Leftrightarrow \phi \vdash (p, q)$

**Proof:** Soundness can be checked by checking each of the proof rules in figure 1. The frame rule is sound by the locality condition, and the others are easy to check.

For completeness, assume we are given  $\phi \models (p, q)$ . By lemma 4, we have  $bla[\phi] \models (p, q)$ . So for all  $\sigma \in p$ ,  $bla[\phi](\sigma) \sqsubseteq q$ , which implies

$$\bigsqcup_{\sigma \in p} bla[\phi](\sigma) \sqsubseteq q \quad (*)$$

Now we have the following derivation:

$$\frac{\frac{\frac{\phi}{(r, s) \quad (r, s) \in \phi}}{(\{\sigma'\}, s) \quad \sigma' \in r, (r, s) \in \phi}}{(\{\sigma - \sigma'\} * \{\sigma'\}, \{\sigma - \sigma'\} * s) \quad \sigma' \in r, (r, s) \in \phi, \sigma' \preceq \sigma, \sigma \in p}}{\left( \bigsqcap_{\substack{\sigma' \preceq \sigma \\ \sigma' \in r \\ (r, s) \in \phi}} \{\sigma - \sigma'\} * \{\sigma'\}, \bigsqcap_{\substack{\sigma' \preceq \sigma \\ \sigma' \in r \\ (r, s) \in \phi}} \{\sigma - \sigma'\} * s \right) \quad \sigma \in p} \frac{(\{\sigma\}, bla[\phi](\sigma)) \quad \sigma \in p}{\left( \bigsqcup_{\sigma \in p} \{\sigma\}, \bigsqcup_{\sigma \in p} bla[\phi](\sigma) \right)} \frac{}{(p, q)}$$

The last step in the proof is by (\*) and the rule of consequence. Note that the intersection rule can be safely applied because the argument of the intersection is necessarily non-empty (if it were empty then  $bla[\phi](\sigma) = \top$ , which contradicts  $bla[\phi](\sigma) \sqsubseteq q$ ). ■

### 3 Properties of Specifications

We discuss certain properties of specifications as a prerequisite for our main discussion on footprints in Section 4. We introduce the notion of a *complete* specification for a local function: a specification from which all properties that hold for the function can be derived in the proof system. However, a function may have many complete specifications, so we introduce a canonical form for specifications. We show that of all the complete specifications of a local function, there exists a unique canonical complete specification for every domain. As discussed in the introduction, an important notion of local reasoning is the *small specification* which completely describes the behaviour of a local function by mentioning only the footprint. Thus, as a prerequisite to investigating their existence, we formalise small specifications as complete specifications with the smallest possible domain. Similarly, we define *big* specifications as complete specifications with the biggest domain.

**Definition 12 (Complete Specification).** A specification  $\phi \in \Phi$  is a **complete specification** for  $f$ , written  $complete(\phi, f)$ , iff, for all  $p, q \in P(\Sigma)$ ,  $f \models (p, q) \Leftrightarrow \phi \models (p, q)$ . Let  $\Phi_{comp(f)}$  be the set of all complete specifications of  $f$ .

$\phi$  is complete for  $f$  whenever the tuples that hold for  $f$  are *exactly* the tuples that follow from  $\phi$ . This also means that any two complete specifications  $\phi$  and  $\psi$  for a local function are semantically equivalent, that is,  $\phi \models \psi$ . The following proposition illustrates how the notions of best local action and complete specification are closely related.

**Proposition 2.** For all  $\phi \in \Phi$  and local functions  $f$ ,  $\text{complete}(\phi, f) \Leftrightarrow f = \text{bla}[\phi]$ .

**Proof:** Assume  $f = \text{bla}[\phi]$ . Then, by lemma 4 we have that  $\phi$  is a complete specification for  $f$ .

For the converse, assume  $\text{complete}(\phi, f)$ . We shall show that for any  $\sigma \in \Sigma$ ,  $f(\sigma) = \text{bla}[\phi](\sigma)$ .

**case 1:**  $f(\sigma) = \top$ . If  $\text{bla}[\phi](\sigma) \neq \top$ , then  $\text{bla}[\phi] \models (\{\sigma\}, \text{bla}[\phi](\sigma))$ . This means that  $\phi \models (\{\sigma\}, \text{bla}[\phi](\sigma))$  (by lemma 4), and so  $f \models (\{\sigma\}, \text{bla}[\phi](\sigma))$ , but this is a contradiction. Therefore,  $\text{bla}[\phi](\sigma) = \top$

**case 2:**  $\text{bla}[\phi](\sigma) = \top$ . If  $f(\sigma) \neq \top$ , then  $f \models (\{\sigma\}, f(\sigma))$ . This means that  $\phi \models (\{\sigma\}, f(\sigma))$ , and so  $\text{bla}[\phi] \models (\{\sigma\}, f(\sigma))$ , but this is a contradiction. Therefore,  $f(\sigma) = \top$

**case 3:**  $\text{bla}[\phi](\sigma) \neq \top$  and  $f(\sigma) \neq \top$ . We have

$$\begin{aligned} f &\models (\{\sigma\}, f(\sigma)) \\ \Rightarrow \text{bla}[\phi] &\models (\{\sigma\}, f(\sigma)) \\ \Rightarrow \text{bla}[\phi](\sigma) &\sqsubseteq f(\sigma) \end{aligned}$$

$$\begin{aligned} \text{bla}[\phi] &\models (\{\sigma\}, \text{bla}[\phi](\sigma)) \\ \Rightarrow f &\models (\{\sigma\}, \text{bla}[\phi](\sigma)) \\ \Rightarrow f(\sigma) &\sqsubseteq \text{bla}[\phi](\sigma) \end{aligned}$$

Therefore  $f(\sigma) = \text{bla}[\phi](\sigma)$  ■

Any specification is therefore only complete for a unique local function, which is its best local action. However, a local function may have lots of complete specifications. We therefore introduce a canonical form for specifications.

**Definition 13 (Canonicalisation).** The canonicalisation of a specification  $\phi$  is defined as  $\phi_{can} = \{(\{\sigma\}, \text{bla}[\phi](\sigma)) \mid \sigma \in D(\phi)\}$ . A specification is in **canonical form** if it is equal to its canonicalisation. Let  $\Phi_{can}(f)$  denote the set of all canonical complete specifications of  $f$ .

**Proposition 3.** For any specification  $\phi$ , we have  $\phi \Vdash \phi_{can}$ .

**Proof:** We first show  $\phi \models \phi_{can}$ . For any  $(p, q) \in \phi_{can}$ ,  $(p, q)$  is of the form  $(\{\sigma\}, \text{bla}[\phi](\sigma))$  for some  $\sigma \in D(\phi)$ . So we have  $\text{bla}[\phi] \models (p, q)$ , and so  $\phi \models (p, q)$  by lemma 4.

We now show  $\phi_{can} \models \phi$ . For any  $(p, q) \in \phi$ , we have  $\text{bla}[\phi] \models (p, q)$ . So for all  $\sigma \in p$ ,  $\text{bla}[\phi](\sigma) \sqsubseteq q$ , which implies

$$\bigsqcup_{\sigma \in p} \text{bla}[\phi](\sigma) \sqsubseteq q \quad (*)$$

Now we have the following derivation:

$$\frac{\frac{\phi_{can}}{(\{\sigma\}, \text{bla}[\phi](\sigma)) \quad \sigma \in p}}{(\bigsqcup_{\sigma \in p} \{\sigma\}, \bigsqcup_{\sigma \in p} \text{bla}[\phi](\sigma))}}{(p, q)}$$

The last step is by  $(*)$  and consequence. So we have  $\phi_{can} \vdash \phi$ , and by soundness  $\phi_{can} \models \phi$ . ■

Thus, the canonicalisation of a specification is logically equivalent to the specification. The following corollary shows that all complete specifications that have the same domain have a unique canonical form, and specifications of different domains have different canonical forms.

**Corollary 1.**  $\Phi_{can(f)}$  is isomorphic to the quotient set  $\Phi_{comp(f)} / \cong_D$ , under the equality-preserving isomorphism that maps  $[\phi]_{\cong_D}$  to  $\phi_{can}$ .

**Proof:** By proposition 2, all complete specifications for  $f$  have the same best local action, which is  $f$  itself. So by the definition of canonicalisation, it can be seen that complete specifications with different domains have different canonicalisations, and complete specifications with the same domain have the same canonicalisation. This shows that the mapping is well-defined and injective. Every canonical complete specification  $\phi$  is also complete, and  $[\phi]_{\cong_D}$  would map to  $\phi_{can} = \phi$ , so the mapping is surjective. ■

**Definition 14 (Small and Big specifications).**  $\phi$  is a **small specification** for  $f$  iff  $\phi \in \Phi_{comp(f)}$  and there is no  $\psi \in \Phi_{comp(f)}$  such that  $D(\psi) \sqsubset D(\phi)$ . A **big specification** is defined similarly.

Small and big specifications are thus the specifications with the smallest and biggest domains respectively. The question is if/when small and big specifications exist. The following result shows that a canonical big specification exists for every local function.

**Proposition 4 (Big Specification).** For any local function  $f$ , the canonical big specification for  $f$  is given by  $\phi_{big(f)} = \{(\{\sigma\}, f(\sigma)) \mid f(\sigma) \sqsubset \top\}$ .

**Proof:**  $f \models \phi_{big(f)}$  is trivial to check. To show  $complete(\phi_{big(f)}, f)$ , assume  $f \models (p, q)$  for some  $p, q \in P(\Sigma)$ . Note that for any  $\sigma \in p$ ,  $f(\sigma) \sqsubseteq q$ , and so  $\bigsqcup_{\sigma \in p} f(\sigma) \sqsubseteq q$ .

We then have the derivation

$$\frac{\frac{\phi_{big(f)}}{(\{\sigma\}, f(\sigma)) \quad f(\sigma) \sqsubset \top}}{(\bigsqcup_{\sigma \in p} \{\sigma\}, \bigsqcup_{\sigma \in p} f(\sigma))}}{(p, q)}$$

By soundness we get  $\phi_{big(f)} \models (p, q)$ .  $\phi_{big(f)}$  has the biggest domain because  $f$  would fault on any element not included in  $\phi_{big(f)}$ , and so it cannot be a domain element for a specification of  $f$ . ■

Small specifications are used in local reasoning to completely specify the behaviour of an update command by only mentioning the behaviour of the command on the part of the resource that is affected by the command [14, 4, 7]. The question of the existence of small specifications is therefore strongly related to the concept of footprints. Finding a small specification is about finding the complete specification with the smallest possible domain, and therefore enquiring about which elements of  $\Sigma$  are essential and sufficient for a complete specification. This requires a formal characterisation of the footprint notion, which we shall now present.

## 4 Footprints

In the introduction we discussed how the *AD* program demonstrates that the footprints of a local function do not correspond simply to the smallest safe states, as these states alone do not always yield complete specifications. In this section we introduce the definition of footprint that does yield complete specifications. In order to understand what the footprint of a local function should be, we begin by analysing the definition of locality. Recall that the locality definition 3 says that the action on a certain state  $\sigma_1$  imposes a *limit* on the action on a bigger state  $\sigma_2 \bullet \sigma_1$ . This limit is  $\{\sigma_2\} * f(\sigma_1)$ , that is, we have  $f(\sigma_2 \bullet \sigma_1) \sqsubseteq \{\sigma_2\} * f(\sigma_1)$ . A reformulation of this definition is that for any state  $\sigma$ , the action on that state has to be within the limit imposed by *every* state  $\sigma'$  that is smaller than it, and we therefore have

$$f(\sigma) \sqsubseteq \prod_{\sigma' \prec \sigma} \{\sigma - \sigma'\} * f(\sigma')$$

We define this overall constraint imposed on  $\sigma$  by all the smaller states as the *local limit* of  $f$  on  $\sigma$ , and show that the locality of a function is equivalent to it satisfying the local limit constraint.

**Definition 15 (Local limit).** *For a local function  $f$  on  $\Sigma$ , and  $\sigma \in \Sigma$ , the **local limit** of  $f$  on  $\sigma$  is defined as*

$$L_f(\sigma) = \prod_{\sigma' \prec \sigma} \{\sigma - \sigma'\} * f(\sigma')$$

**Proposition 5.**  $f$  is local  $\Leftrightarrow f(\sigma) \sqsubseteq L_f(\sigma)$  for all  $\sigma \in \Sigma$

**Proof:** Assume  $f$  is local. So for any  $\sigma$ , for every  $\sigma' \prec \sigma$ ,  $f(\sigma) \sqsubseteq \{\sigma - \sigma'\} * f(\sigma')$ .  $f(\sigma)$  is therefore smaller than the intersection of all these sets, which is  $L_f(\sigma)$ .

For the converse, assume the rhs and that  $\sigma_1 \bullet \sigma_2$  is defined. If  $\sigma_1 = u$  then  $f(\sigma_1 \bullet \sigma_2) \sqsubseteq \{\sigma_1\} * f(\sigma_2)$  and we are done. Otherwise,  $\sigma_2 \prec \sigma_1 \bullet \sigma_2$  and we have  $f(\sigma_1 \bullet \sigma_2) \sqsubseteq L_f(\sigma_1 \bullet \sigma_2) \sqsubseteq \{\sigma_1\} * f(\sigma_2)$ . ■

With this intuition that the locality of  $f$  is determined by the local limit restricting the action of  $f$ , we define the footprints as those elements which further reduce this limit:  $\sigma$  is a footprint of  $f$  if and only if  $f(\sigma) \sqsubset L_f(\sigma)$ . If the result of the function is equal to the local limit on a certain element, then this can be determined just by the fact that  $f$  is a local function. If the result is strictly smaller, then this behaviour is additional to  $f$  being local. The intuition is that this characteristic property of the function would have to be explicitly stated in a complete specification of the function, which would make footprints the essential elements required to describe this function's behaviour. We formally prove this central result after stating the definition and illustrating it with examples.

**Definition 16 (Footprint).** *For a local function  $f$  and  $\sigma \in \Sigma$ ,  $\sigma$  is a footprint of  $f$ , written  $F_f(\sigma)$ , iff  $f(\sigma) \sqsubset L_f(\sigma)$ . We denote the set of footprints of  $f$  by  $F(f)$ .*

Note that an element  $\sigma$  is therefore not a footprint iff the action of  $f$  on  $\sigma$  is at the local limit, that is  $f(\sigma) = L_f(\sigma)$ .

**Lemma 5.** *For any local function  $f$ , all the smallest safe states of  $f$  are footprints of  $f$ .*

**Proof:** *Let  $\sigma$  be a smallest safe state for  $f$ . Then for any  $\sigma' \prec \sigma$ ,  $f(\sigma') = \top$ . Therefore  $L_f(\sigma) = \top$  and so  $f(\sigma) \sqsubset L_f(\sigma)$ . ■*

However, the smallest safe states are not always the *only* footprints. An example is the *AD* command discussed in the introduction. The empty heap is a footprint as it is the smallest safe heap, but the heap cell  $l \mapsto v$  is also a footprint.

*Example 2 (Dispose).* The footprints of the  $dispose[l]$  command in the plain heap model (example 1.1) are the cells at location  $l$ . We check this by considering the following cases

1. The empty heap,  $u_H$ , is not a footprint since  $L_{dispose[l]}(u_H) = \top = dispose[l](u_H)$
2. Every cell  $l \mapsto v$  for some  $v$  is a footprint

$$\begin{aligned} L_{dispose[l]}(l \mapsto v) &= \{l \mapsto v\} * dispose[l](u_H) = \{l \mapsto v\} * \top = \top \\ dispose[l](l \mapsto v) &= \{u_H\} \sqsubset L_{dispose[l]}(l \mapsto v) \end{aligned}$$

3. Every state  $\sigma$  such that  $\sigma \succ (l \mapsto v)$  for some  $v$  is not a footprint

$$L_{dispose[l]}(\sigma) \sqsubseteq \{\sigma - (l \mapsto v)\} * dispose[l](l \mapsto v) = \{\sigma - (l \mapsto v)\} = dispose[l](\sigma)$$

By proposition 5, we have  $L_{dispose[l]}(\sigma) = dispose[l](\sigma)$ . The intuition is that  $\sigma$  does not characterise any ‘new’ behaviour of the function: its action on  $\sigma$  is just a consequence of its action on the cells at location  $l$  and the locality property of the function.

4. Every state  $\sigma$  such that  $\sigma \not\succeq (l \mapsto v)$  for some  $v$  is not a footprint

$$L_{dispose[l]}(\sigma) \sqsubseteq \{\sigma\} * dispose[l](u_H) = \{\sigma\} * \top = \top = dispose[l](\sigma)$$

Again by proposition 5,  $L_{dispose[l]}(\sigma) = dispose[l](\sigma)$ .

*Example 3 (AD command).* The *AD* (Allocate-Deallocate) command was defined on the heap and stack model in example 1.2. We have the following cases for  $\sigma$ .

1.  $\sigma \not\succeq x \mapsto v_1$  for some  $v_1$  is not a footprint, since  $L_{AD}(\sigma) = \top = AD(\sigma)$ .
2.  $\sigma = x \mapsto v_1$  for some  $v_1$  is a footprint since  $L_{AD}(\sigma) = \top$  (by case (1)) and  $AD(\sigma) = \{x \mapsto w \mid w \in L\} \sqsubset L_{AD}(\sigma)$ .
3.  $\sigma = l \mapsto v_1 \bullet x \mapsto v_2$  for some  $l, v_1, v_2$  is a footprint.

$$\begin{aligned} L_{AD}(\sigma) &= \{l \mapsto v_1\} * AD(x \mapsto v_2) \\ &\quad (\text{AD faults on all other elements strictly smaller than } \sigma) \\ &= \{l \mapsto v_1\} * \{x \mapsto w \mid w \in L\} \\ &= \{l \mapsto v_1 \bullet x \mapsto w \mid w \in L\} \end{aligned}$$

$$AD(\sigma) = \{l \mapsto v_1 \bullet x \mapsto w \mid w \in L, w \neq l\} \sqsubset L_{AD}(\sigma)$$

4.  $\sigma = h \bullet x \mapsto v_1$  for some  $v_1$ , and where  $|loc(h)| > 1$ , is not a footprint.

$$\begin{aligned} L_{AD}(\sigma) &\sqsubseteq \bigsqcap_{(l \mapsto v) \prec h} \{(h - l \mapsto v) * AD(l \mapsto v \bullet x \mapsto v_1)\} \\ &= \{h \bullet x \mapsto w \mid w \notin loc(h)\} = AD(\sigma) \end{aligned}$$

By proposition 5 we get  $L_{AD}(\sigma) = AD(\sigma)$ .

Our footprint definition therefore works properly for these specific examples. Now we give the formal general result which captures the underlying intuition of local reasoning that the footprints of a local function are the only essential elements for a complete specification of the function.

**Theorem 2 (Essentiality).** *The footprints of a local function are the essential domain elements for any complete specification of that function, that is,*

$$F_f(\sigma) \Leftrightarrow \forall \phi \in \Phi_{comp(f)}. \sigma \in D(\phi)$$

**Proof:** We first show the right hand direction, that is, every footprint is essential: for all  $\phi \in \Phi_{comp(f)}, \sigma \notin D(\phi) \Rightarrow \neg F_f(\sigma)$ . So assume  $\sigma \notin D(\phi)$ . Since *complete*( $\phi, f$ ), by proposition 2, we have  $f = bla[\phi]$ . So

$$f(\sigma) = \bigsqcap_{\sigma_1 \preceq \sigma, \sigma_1 \in p, (p,q) \in \phi} \{\sigma - \sigma_1\} * q$$

Now for any set  $\{\sigma - \sigma_1\} * q$  in the above intersection, we have that  $\sigma_1 \in p$ , and  $(p, q) \in \phi$  for some  $p$ .  $\sigma_1 \in p$  implies  $f(\sigma_1) \sqsubseteq q$ , and therefore  $\{\sigma - \sigma_1\} * f(\sigma_1) \sqsubseteq \{\sigma - \sigma_1\} * q$ . Also,  $\sigma_1 \neq \sigma$ , because otherwise we would have  $\sigma \in p$ , which would contradict the assumption that  $\sigma \notin D(\phi)$ . So  $\sigma_1 \prec \sigma$  and we have

$$L_f(\sigma) \sqsubseteq \{\sigma - \sigma_1\} * f(\sigma_1) \sqsubseteq \{\sigma - \sigma_1\} * q$$

So the local limit is smaller than each set  $\{\sigma - \sigma_1\} * q$  in the intersection, and therefore it is smaller than the intersection itself:  $L_f(\sigma) \sqsubseteq f(\sigma)$ . We know from proposition 5 that  $f(\sigma) \sqsubseteq L_f(\sigma)$ , so we get  $f(\sigma) = L_f(\sigma)$  and therefore  $\neg F_f(\sigma)$ .

For the converse, we need to show that every non-footprint is not essential, that is, there exists a complete specification that does not include it:  $\neg F_f(\sigma) \Rightarrow \exists \phi \in \Phi_{comp(f)}. \sigma \notin D(\phi)$ . Assume that  $\sigma$  is not a footprint of  $f$ . We shall use the big specification,  $\phi_{big(f)}$ , to construct a complete specification of  $f$  in which  $\sigma$  is not a domain element. If  $f(\sigma) = \top$  then the big specification itself is such a specification, and we are done. Otherwise assume  $f(\sigma) \sqsubset \top$ . Let  $\phi = \phi_{big(f)} / \{(\{\sigma\}, f(\sigma))\}$ . It can be seen that  $\sigma \notin D(\phi)$ . Now we need to show that  $\phi$  is complete for  $f$ . For this it is sufficient to show  $\phi \dashv\vdash \phi_{big(f)}$  because we know that  $\phi_{big(f)}$  is complete for  $f$ .  $\dashv$  is trivial.

For  $\vdash$ , we just need to show  $\phi \vdash (\{\sigma\}, f(\sigma))$ . We have the following derivation:

$$\frac{\frac{\frac{\phi}{(\{\sigma'\}, f(\sigma'))} \quad \sigma' \prec \sigma, f(\sigma') \sqsubset \top}{(\{\sigma - \sigma'\} * \{\sigma'\}, \{\sigma - \sigma'\} * f(\sigma'))} \quad \sigma' \prec \sigma, f(\sigma') \sqsubset \top}{(\{\sigma\}, \bigsqcap_{\sigma' \prec \sigma, f(\sigma') \sqsubset \top} \{\sigma - \sigma'\} * f(\sigma'))} \quad \sigma' \prec \sigma, f(\sigma') \sqsubset \top}{(\{\sigma\}, L_f(\sigma))}$$

The intersection rule can be safely applied as there is at least one  $\sigma' \prec \sigma$  such that  $f(\sigma') \sqsubset \top$ . This is because  $f(\sigma) \sqsubset \top$ , so if there were no such  $\sigma'$  then  $\sigma$  would be a footprint, which is a contradiction. Note that the last step uses the fact that

$$\bigsqcap_{\sigma' \prec \sigma, f(\sigma') \sqsubset \top} \{\sigma - \sigma'\} * f(\sigma') = \bigsqcap_{\sigma' \prec \sigma} \{\sigma - \sigma'\} * f(\sigma') = L_f(\sigma)$$

because adding the top element to an intersection does not change its value. Since  $\sigma$  is not a footprint,  $f(\sigma) = L_f(\sigma)$ , and so  $\phi \vdash (\{\sigma\}, f(\sigma))$ . ■

## 5 Sufficiency and Small Specifications

We know that the footprints are the only elements that are *essential* for a complete specification of a local function in the sense that every complete specification must include them. Now we ask when a set of elements is *sufficient* for a complete specification of a local function, in the sense that there exists a complete specification of the function that only includes these elements. In particular, we wish to know if the footprints alone are sufficient. To study this, we begin by identifying the notion of the *basis* of a local function.

### 5.1 Bases

The local limit of a function on a state is the constraint imposed by *all* the substates. We now consider the constraint imposed by only some of the substates.

**Definition 17 (Local limit imposed by a set).** For a subset  $A$  of  $\Sigma$ , the **local limit imposed by  $A$  on the action of  $f$  on  $\sigma$**  is defined by

$$L_{A,f}(\sigma) = \bigsqcap_{\sigma' \preceq \sigma, \sigma' \in A} \{\sigma - \sigma'\} * f(\sigma')$$

Sometimes, the local limit imposed by  $A$  is enough to completely determine  $f$ . In this case, we call  $A$  a *basis* for  $f$ .

**Definition 18 (Basis).**  $A \sqsubseteq \Sigma$  is a **basis** for  $f$ , written  $\text{basis}(A, f)$ , iff  $L_{A,f} = f$ .

This means that, when given the action of  $f$  on elements in  $A$  alone, we can determine the action of  $f$  on any element in  $\Sigma$  by just using the locality property of  $f$ . Every local function has at least one basis, namely the trivial basis  $\Sigma$  itself. We next show the correspondence between the bases and complete specifications of a local function.

**Lemma 6.** Let  $\phi_{A,f} = \{(\{\sigma\}, f(\sigma)) \mid \sigma \in A, f(\sigma) \sqsubset \top\}$ . Then we have  $\text{basis}(A, f) \Leftrightarrow \text{complete}(\phi_{A,f}, f)$ .

**Proof:** We have  $L_{A,f} = \text{bla}[\phi_{A,f}]$  by definition. The result follows by proposition 2 and the definition of basis. ■

For every canonical complete specification  $\phi \in \Phi_{can(f)}$ , we have  $\phi = \phi_{D(\phi),f}$ . By the previous lemma it follows that  $D(\phi)$  forms a basis for  $f$ . The lemma therefore shows that every basis determines a complete canonical specification, and vice versa. This correspondence also carries over to all complete specifications for  $f$  by the fact that every domain-equivalent class of complete specifications for  $f$  is represented by the canonical complete specification with that domain (corollary 1). It is a simple consequence of the Essentiality theorem (2) that the footprints are present in every basis for a local function.

**Lemma 7.** *The footprints of  $f$  are included in every basis of  $f$ .*

**Proof:** Every basis  $A$  of  $f$  determines a complete specification for  $f$  the domain of which is a subset of  $A$ . By the essentiality theorem (2), the domain includes the footprints. ■

The question of sufficiency is about how small the basis can get. Given a local function, we wish to know if it has a smallest basis.

## 5.2 Well-founded Resource

We know that every basis must contain the footprints. Thus if the footprints alone form a basis, then the function will have a *smallest* complete specification whose domain are just the footprints. We find that for well-founded resource models, this is indeed the case.

**Theorem 3 (Sufficiency I).** *If a separation algebra  $\Sigma$  is well-founded under the  $\preceq$  relation, then the footprints of any local function form a basis for it, that is,  $f = L_{F(f),f}$ .*

**Proof:** Assume that  $\Sigma$  is well-founded under  $\preceq$ . We shall show by induction that  $L_{F(f),f}(\sigma) = f(\sigma)$  for all  $\sigma \in \Sigma$ . The induction hypothesis is that for all  $\sigma' \prec \sigma$ ,  $L_{F(f),f}(\sigma') = f(\sigma')$

**case 1:**  $F_f(\sigma)$ . We have  $f(\sigma) = \{u\} * f(\sigma)$  is in the intersection in the definition of  $L_{F(f),f}(\sigma)$ , and so  $L_{F(f),f}(\sigma) \sqsubseteq f(\sigma)$ . We have by locality that  $f(\sigma) \sqsubseteq L_{F(f),f}(\sigma)$ , and so  $f(\sigma) = L_{F(f),f}(\sigma)$ .

**case 2:**  $\neg F_f(\sigma)$ . We have

$$\begin{aligned}
f(\sigma) &= L_f(\sigma) \quad (\text{because } \sigma \text{ is not a footprint of } f) \\
&= \prod_{\sigma' \prec \sigma} \{\sigma - \sigma'\} * f(\sigma') \\
&= \prod_{\sigma' \prec \sigma} (\{\sigma - \sigma'\} * \prod_{\sigma'' \preceq \sigma', F_f(\sigma'')} \{\sigma' - \sigma''\} * f(\sigma'')) \quad (\text{by the induction hypothesis}) \\
&= \prod_{\sigma' \prec \sigma, \sigma'' \preceq \sigma', F_f(\sigma'')} \{\sigma - \sigma'\} * \{\sigma' - \sigma''\} * f(\sigma'') \quad (\text{by the precision of } \{\sigma - \sigma'\}) \\
&= \prod_{\sigma'' \prec \sigma, F_f(\sigma'')} \{\sigma - \sigma''\} * f(\sigma'') \\
&= \prod_{\sigma'' \preceq \sigma, F_f(\sigma'')} \{\sigma - \sigma''\} * f(\sigma'') \quad (\text{because } \sigma \text{ is not a footprint of } f) \\
&= L_{F(f),f}(\sigma)
\end{aligned}$$

■

In section 3, the notions of big and small specifications were introduced, and the existence of a big specification was shown. We are now in a position to show the existence of the small specification for well-founded resource. If  $\Sigma$  is well-founded, then every local function has a small specification whose domain is the footprints of the function.

**Corollary 2 (Small specification).** *For well-founded separation algebras, every local function has a small specification given by  $\phi_{F(f),f}$ .*

**Proof:** *It is complete by theorem 3 and lemma 6. It has the smallest domain by the essentiality theorem.* ■

Thus for well-founded resource, the footprints are always essential and sufficient, and specifications need not consider any other elements. In practice, small specifications may not always be in canonical form even though they always have the same domain as the canonical form. For example, the heap dispose command can have the specification  $\{(\{l \mapsto v \mid v \in Val\}, \{u_H\})\}$  rather than  $\{(\{l \mapsto v\}, \{u_H\}) \mid v \in Val\}$ .

Although well-founded resource is usually the case in practice, a notable exception is the fractional permissions model [4] in which the resource includes permissions that can be indefinitely divided.

### 5.3 Non-well-founded Resource

In analysing the non-well-founded case, we found it important to distinguish between models that do or do not have *negativity*.

**Definition 19 (Negativity).** *A separation algebra  $\Sigma$  has **negativity** iff there exists a non-unit element  $\sigma \in \Sigma$  that has an inverse, that is,  $\sigma \neq u$  and  $\sigma \bullet \sigma' = u$  for some  $\sigma' \in \Sigma$ . We say that  $\Sigma$  is **non-negative** if no such element exists.*

If a model has negativity then it is non-well-founded, because for elements  $\sigma$  and  $\sigma'$  such that  $\sigma \bullet \sigma' = u$ , the set  $\{\sigma, u\}$  forms an infinite descending chain (there is no least element). All well-founded models are therefore non-negative. In the general non-negative case, we find that either the footprints form a basis, or there is no smallest basis.

**Theorem 4 (Sufficiency II).** *If  $\Sigma$  is non-negative then, for any local  $f$ , either the footprints form a smallest basis or there is no smallest basis for  $f$ .*

**Proof:** Let  $A$  be a basis for  $f$  (we know there is at least one, which is the trivial basis  $\Sigma$  itself). If  $A$  is the footprints then we are done. So assume  $A$  contains some non-footprint  $\mu$ . We shall show that there exists a smaller basis for  $f$ , which is  $A/\{\mu\}$ . So it suffices to show  $f(\sigma) = L_{A/\{\mu\},f}(\sigma)$  for all  $\sigma \in \Sigma$ . We have

$$f(\sigma) = L_{A,f}(\sigma) = \prod_{\sigma' \preceq \sigma, \sigma' \in A} \{\sigma - \sigma'\} * f(\sigma')$$

**case 1:**  $\mu \not\leq \sigma$ . We have  $f(\sigma) = \prod_{\sigma' \preceq \sigma, \sigma' \in A/\{\mu\}} \{\sigma - \sigma'\} * f(\sigma') = L_{A/\{\mu\}, f}(\sigma)$

**case 2:**  $\mu \leq \sigma$ . In this case

$$f(\sigma) = \left( \prod_{\sigma' \preceq \sigma, \sigma' \in A/\{\mu\}} \{\sigma - \sigma'\} * f(\sigma') \right) \sqcap (\{\sigma - \mu\} * f(\mu))$$

We shall show that the left hand side of the intersection in the above expression is contained in the right hand side, and so the expression is equal to the LHS.

$$\begin{aligned} RHS &= \{\sigma - \mu\} * f(\mu) = \{\sigma - \mu\} * L_f(\mu) \quad (\text{because } \mu \text{ is not a footprint of } f) \\ &= \{\sigma - \mu\} * \prod_{\sigma' \prec \mu} \{\mu - \sigma'\} * f(\sigma') \\ &= \{\sigma - \mu\} * \prod_{\sigma' \prec \mu} \{\mu - \sigma'\} * \prod_{\sigma'' \preceq \sigma', \sigma'' \in A/\{\mu\}} \{\sigma' - \sigma''\} * f(\sigma'') \\ &\quad (\text{case 1 applies because } \Sigma \text{ is non-negative, so } \sigma' \prec \mu \Rightarrow \mu \not\leq \sigma') \\ &= \prod_{\sigma' \prec \mu} \prod_{\sigma'' \preceq \sigma', \sigma'' \in A/\{\mu\}} \{\sigma - \mu\} * \{\mu - \sigma'\} * \{\sigma' - \sigma''\} * f(\sigma'') \quad (\text{by precision}) \\ &= \prod_{\sigma' \prec \mu} \prod_{\sigma'' \preceq \sigma', \sigma'' \in A/\{\mu\}} \{\sigma - \sigma''\} * f(\sigma'') \\ &= \prod_{\sigma'' \prec \mu, \sigma'' \in A/\{\mu\}} \{\sigma - \sigma''\} * f(\sigma'') \\ &\sqsupseteq \prod_{\sigma'' \preceq \sigma, \sigma'' \in A/\{\mu\}} \{\sigma - \sigma''\} * f(\sigma'') = LHS \end{aligned}$$

■

**Corollary 3 (Small Specification).** *If  $\Sigma$  is non-negative, then every local function either has a small specification given by  $\phi_{F(f), f}$  or there is no smallest complete specification for that function.*

*Example 4 (Permissions).* The fractional permissions model [4] is non-well-founded and non-negative. It can be represented by the separation algebra  $HPerm = L \rightarrow_{fin} Val \times P$  where  $L$  and  $Val$  are as in example 1, and  $P$  is the interval  $(0, 1]$  of rational numbers. Elements of  $P$  represent ‘permissions’ to access a heap cell. A permission of 1 for a cell means both read and write access, while any permission less than 1 is read-only access.  $\bullet$  joins disjoint heaps and adds the permissions together for any cells that are present in both heaps only if the resulting permission for each heap cell does not exceed 1, and the operation is undefined otherwise. In this case, the write function that updates the value at a location requires a permission of at least 1 and faults on any smaller permission. It therefore has a small specification with precondition being the cell with permission 1. The read function, however, can execute safely on any positive permission, no matter how small. Thus this function can be completely specified with a specification that has a precondition given by the cell with permission  $z$ , for all  $0 < z \leq 1$ . However, this is not a *smallest* specification, as a smaller one can be given by further restricting  $0 < z \leq 0.5$ . We can therefore always find a smaller specification by reducing the value of  $z$  but keeping it positive.

For resource with negativity, we find that it is possible to have small specifications that include non-essential elements (which by theorem 2 are not footprints). These elements are non-essential in the sense that complete specifications exist that do not include them, but there is no complete specification that includes only essential elements.

*Example 5 (Integers).* An example of a model with negativity is the separation algebra of integers  $(\mathbb{Z}, +, 0)$ . In this case there can be local functions which can have small specifications that contain non-footprints. Let  $f : \mathbb{Z} \rightarrow P(\mathbb{Z})^\top$  be defined as  $f(n) = \{n + c\}$  for some constant  $c$ , as in example 1.  $f$  is local, but it has no footprints. This is because for any  $n$ ,  $f(n) = 1 + f(n - 1)$ , and so  $n$  is not a footprint of  $f$ . However,  $f$  does have small specifications, for example,  $\{\{0\}, \{c\}\}$ ,  $\{\{5\}, \{5 + c\}\}$ , or indeed  $\{\{n\}, \{n + c\}\}$  for any  $n \in \mathbb{Z}$ . So although every element is non-essential, some element is required to give a complete specification.

## 6 Regaining Access Footprints

In the introductions we discussed how the notion of footprints as the smallest safe states - the *access footprint*- is inadequate for giving complete specifications, even for the standard RAM model, as illustrated by the *AD* example. We therefore investigated the general notion of footprint that overcomes this problem for arbitrary local functions on arbitrary separation algebras. In light of this analysis, we consider the access footprint problem from a different point of view.

### 6.1 An alternative model

In this section we explore the possibility of an alternative heap model in which the access footprints do correspond to the actual footprints. We begin by taking a closer look at why the *AD* anomaly occurs in the standard heap and stack model described in example 1.2. Consider an application of the allocation command in this model:

$$\text{new}[x](42 \mapsto v \bullet x \mapsto w) = \{42 \mapsto v \bullet x \mapsto l \bullet l \mapsto r \mid l \in L \setminus \{42\}, r \in \text{Val}\}$$

The intuition of locality is that the initial state  $42 \mapsto v \bullet x \mapsto w$  is only describing a local region of the heap and the stack, rather than the whole global state. In this case it says that the address 42 is initially allocated, and the definition of the allocation command is that the resulting state will have a new cell, the address of which can be anything other than 42. However, we notice that the initial state is in fact not just describing only its local region of the heap. It does state that 42 is allocated, but it also implicitly states a very global property: that *all other addresses are not allocated*. This is why the allocation command can choose to allocate any location that is not 42. Thus in this model, every local state implicitly contains some global allocation information which is used by the allocation command. In contrast, a command such as mutate does not require this global ‘knowledge’ of the allocation status of any other cell that it is not affecting. Now the global information of which cells are free *changes* as more resource is added to the initial state, so this can lead to program behaviour being sensitive to the

addition of more resource to the initial state, and this sensitivity is apparant in the case of the *AD* program.

Based on this observation, we consider an alternative model. As before, a state will represent a local allocated region of the heap. However, unlike before, a given state will say nothing about the allocation status of any other locations. This information will be represented explicitly in a *free* set, which will contain all the locations that are not allocated in the global heap.

Formally, we work with a separation algebra  $(H, \bullet, u_H)$ . Let  $L$ ,  $Var$  and  $Val$  be locations, variables and values, as before. The states  $h \in H$  are given by the grammar:

$$h ::= u_H \mid l \mapsto v \mid x \mapsto v \mid F \mid h \bullet h$$

where  $l \in L$ ,  $v \in Val$ ,  $x \in Var$  and  $F \in P(L)$ . As before,  $\bullet$  is undefined for states with overlapping locations or variables. Let  $loc(h)$  and  $var(h)$  be the set of locations and variables in state  $h$  respectively. The set  $F$  carries the information of which locations are free in the global heap. Thus we allow at most one free set in a state, and the free set must be disjoint from all locations in the state. So  $h \bullet F$  is only defined when  $loc(h) \cap F = \emptyset$  and  $h \neq h' \bullet F'$  for some  $h'$  and  $F'$ . Associativity and commutativity of  $\bullet$  is imposed, and  $u_H$  is defined as the identity  $u_H \bullet h = h \bullet u_H = h$ .

In this setting, the allocation command requires ownership of the free set for safe execution, as it chooses the location to allocate from this set. It removes the chosen address from the free set as it allocates the cell. It is defined as

$$new[x](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet l \mapsto w \bullet F \setminus \{l\} \mid w \in Val, l \in F\} & h = h' \bullet x \mapsto v \bullet F \\ \top & \text{otherwise} \end{cases}$$

Note that the output states  $h' \bullet x \mapsto l \bullet l \mapsto w \bullet F \setminus \{l\}$  are defined, since we have  $l \notin F \setminus \{l\}$  and the input state  $h' \bullet x \mapsto v \bullet F$  implies that  $loc(h')$  is disjoint from  $F \setminus \{l\}$ . The deallocation command also requires the free set, as it updates the set with address of the cell that it deletes:

$$dispose[x](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet F \cup \{l\}\} & h = h' \bullet x \mapsto l \bullet l \mapsto v \bullet F \\ \top & \text{otherwise} \end{cases}$$

Again, the output states are defined, since the input state implies that  $loc(h') \cup \{l\}$  is disjoint from  $F$ , and so  $loc(h')$  is disjoint from  $F \cup \{l\}$ . Notice that in this setting only the allocation and deallocation commands refer to the free set, since commands such as mutation and lookup are completely independent of the allocation status of other cells and are defined exactly as in example 1.2:

$$mutate[x, v](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet l \mapsto v\} & h = h' \bullet x \mapsto l \bullet l \mapsto w \\ \top & \text{otherwise} \end{cases}$$

$$lookup[x, y](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet l \mapsto v \bullet y \mapsto v\} & h = h' \bullet x \mapsto l \bullet l \mapsto v \bullet y \mapsto w \\ \top & \text{otherwise} \end{cases}$$

**Lemma 8.** *The functions  $new[x]$ ,  $dispose[x]$ ,  $mutate[x, v]$  and  $lookup[x, y]$  are all local in the separation algebra  $(H, \bullet, u_H)$ .*

**Proof:** Let  $f = \text{new}[x]$  and assume  $h' \# h$ . We want to show  $f(h' \bullet h) \sqsubseteq \{h'\} * f(h)$ . Assume  $h = h'' \bullet x \mapsto l \bullet l \mapsto v \bullet F$  for some  $h'', x, l, v$  and  $F$ , because otherwise  $f(h) = \top$  and we are done. So we have

$$\begin{aligned} f(h' \bullet h) &= \{h' \bullet h'' \bullet x \mapsto l \bullet l \mapsto w \bullet F \setminus \{l\} \mid w \in \text{Val}, l \in F\} \\ &= \{h'\} * \{h'' \bullet x \mapsto l \bullet l \mapsto w \bullet F \setminus \{l\} \mid w \in \text{Val}, l \in F\} \\ &= \{h'\} * f(h) \end{aligned}$$

The other functions can be checked in a similar way. ■

## 6.2 Access footprints for AD

We consider the footprint of the *AD* command in the new model. In this model the sequential composition  $\text{new}[x]; \text{dispose}[x]$  gives the function

$$AD(h) = \begin{cases} \{h' \bullet x \mapsto l \bullet F \mid l \in F\} & h = h' \bullet x \mapsto v \bullet F \\ \top & \text{otherwise} \end{cases}$$

The smallest safe states are given by the set  $\{x \mapsto v \bullet F \mid v \in \text{Val}, F \in P(L)\}$ . By lemma 5, these smallest safe states are footprints. However, unlike before, in this model these are the *only* footprints of the *AD* command. To see this, consider a larger state  $h \bullet x \mapsto v \bullet F$ . We have

$$\begin{aligned} AD(h \bullet x \mapsto v \bullet F) &= \{h \bullet x \mapsto l \bullet F \mid l \in F\} \\ &= \{h\} * \{x \mapsto l \bullet F \mid l \in F\} \\ &= \{h\} * AD(x \mapsto v \bullet F) \end{aligned}$$

Since the local limit  $L_{AD}(h \bullet x \mapsto v \bullet F) \sqsubseteq \{h\} * AD(x \mapsto v \bullet F)$ , we have by proposition 5 that  $L_{AD}(h \bullet x \mapsto v \bullet F) = AD(h \bullet x \mapsto v \bullet F)$ , and so  $h \bullet x \mapsto v \bullet F$  is not a footprint of *AD*.

Hence, by corollary 2, in this model the *AD* command has a smallest complete specification in which the pre-condition only includes the smallest safe states. This specification is  $\{(\{x \mapsto v \bullet F\}, \{x \mapsto l \bullet F\}) \mid v \in \text{Val}, F \in P(L), l \in F\}$ . Intuitively, it says that if initially the variable  $x$  is present and we know which cells are free in the global heap, then after the execution, exactly the same cells will still be free, and  $x$  will point to one of those free cells. This completely describes the behaviour of the command for all larger states. For example, applying the frame rule gives us the complete specification on the larger state in which 42 is allocated:

$$\{(\{42 \mapsto v\} * \{x \mapsto w \bullet F\}, \{42 \mapsto v\} * \{x \mapsto l \bullet F\}) \mid v, w \in \text{Val}, F \in P(L), l \in F\}$$

In the pre-condition, the presence of location 42 in the heap means that 42 is not in the free set  $F$  (by definition of  $*$ ). Therefore, in the post-condition,  $x$  cannot point to 42.

$$\begin{aligned} \llbracket c \rrbracket &\in \text{LocFunc} & \llbracket \text{skip} \rrbracket(\sigma) &= \{\sigma\} \\ \llbracket C_1; C_2 \rrbracket &= \llbracket C_1 \rrbracket; \llbracket C_2 \rrbracket & \llbracket C_1 + C_2 \rrbracket &= \llbracket C_1 \rrbracket \sqcup \llbracket C_2 \rrbracket & \llbracket C^* \rrbracket &= \bigsqcup_n \llbracket C^n \rrbracket \end{aligned}$$

**Fig. 2.** Denotational Semantics

### 6.3 Access footprints for arbitrary programs

Now that we have regained the access footprints for  $AD$  in the new model, we want to know if this is generally the case for *any program*. We consider the general programming language given in [9]:

$$C ::= c \mid \text{skip} \mid C; C \mid C + C \mid C^*$$

where  $c$  ranges over an arbitrary collection of primitive commands,  $+$  is nondeterministic choice,  $;$  is sequential composition, and  $(\cdot)^*$  is Kleene-star (iterated  $;$ ). As discussed in [9], conditionals and while loops can be encoded using  $+$  and  $(\cdot)^*$  and assume statements. The denotational semantics of commands is given in Figure 2.

Taking the primitive commands to be  $\text{new}[x]$ ,  $\text{dispose}[x]$ ,  $\text{mutate}[x, v]$ , and  $\text{lookup}[x, y]$ , our original aim was to show that for every command  $C$ , the footprints of  $\llbracket C \rrbracket$  in the new model are the smallest safe states. However, in attempting to do this, we identified a general condition on primitive commands under which the result holds for arbitrary separation algebras.

Let  $f$  be a local function on a separation algebra  $\Sigma$ . If, for  $A \in P(\Sigma)$ , we define  $f(A) = \bigsqcup_{\sigma \in A} f(\sigma)$ , then the locality condition can be stated as

$$\forall \sigma', \sigma \in \Sigma. f(\{\sigma'\} * \{\sigma\}) \sqsubseteq \{\sigma'\} * f(\sigma)$$

As discussed before, the  $\sqsubseteq$  ordering allows local functions to be more deterministic on larger states. Also, this sensitivity of determinism to the ‘frame states’ is apparent in the  $AD$  command in the original model from example 1.2, and is what is avoided in the new model. With this observation, we analyse the general class of local functions in which this sensitivity is not present.

**Definition 20 (Determinism Constancy).** Let  $f$  be a local function and  $\text{safe}(f)$  the set of states on which  $f$  does not fault.  $f$  has the determinism constancy property iff for every  $\sigma \in \text{safe}(f)$ ,

$$\forall \sigma' \in \Sigma. f(\{\sigma'\} * \{\sigma\}) = \{\sigma'\} * f(\sigma)$$

**Lemma 9.** If a local function  $f$  has determinism constancy then its footprints are the smallest safe states.

**Proof:** Let  $\text{min}(f)$  be the smallest safe states of  $f$ . Then for any larger state  $\sigma' \bullet \sigma$  where  $\sigma \in \text{min}(f)$  and  $\sigma' \in \Sigma$ , we have

$$f(\sigma' \bullet \sigma) = f(\{\sigma'\} * \{\sigma\}) = \{\sigma'\} * f(\sigma)$$

Since  $L_f(\sigma' \bullet \sigma) \sqsubseteq \{\sigma'\} * f(\sigma)$ , by proposition 5 we have that  $L_f(\sigma' \bullet \sigma) = f(\sigma' \bullet \sigma)$ , and so  $\sigma' \bullet \sigma$  is not a footprint of  $f$ . ■

**Theorem 5.** *If all the primitive commands of the programming language have determinism constancy, then the footprint of every program is given by the smallest safe states.*

**Proof:** Assuming all primitive commands have determinism constancy, we shall show by induction that every command has determinism constancy and the result follows by lemma 9. So for commands  $C_1$  and  $C_2$ , let  $f = \llbracket C_1 \rrbracket$  and  $g = \llbracket C_2 \rrbracket$  and assume  $f$  and  $g$  have determinism constancy. For sequential composition, we have for  $\sigma \in \text{safe}(f; g)$  and  $\sigma' \in \Sigma$ ,

$$\begin{aligned}
& (f; g)(\{\sigma'\} * \{\sigma\}) \\
&= g(f(\{\sigma'\} * \{\sigma\})) \\
&= g(\{\sigma'\} * f(\{\sigma\})) \\
&\quad (f \text{ has determinism constancy and } \sigma \in \text{safe}(f) \text{ since } \sigma \in \text{safe}(f; g)) \\
&= g\left(\bigsqcup_{\sigma_1 \in f(\sigma)} \{\sigma'\} * \{\sigma_1\}\right) \\
&= \bigsqcup_{\sigma_1 \in f(\sigma)} g(\{\sigma'\} * \{\sigma_1\}) \\
&= \bigsqcup_{\sigma_1 \in f(\sigma)} \{\sigma'\} * g(\sigma_1) \\
&\quad (g \text{ has determinism constancy and } \sigma_1 \in \text{safe}(g) \text{ since } \sigma \in \text{safe}(f; g) \text{ and } \sigma_1 \in f(\sigma)) \\
&= \{\sigma'\} * \bigsqcup_{\sigma_1 \in f(\sigma)} g(\sigma_1) \quad (\text{distributivity}) \\
&= \{\sigma'\} * (f; g)(\sigma)
\end{aligned}$$

For non-deterministic choice, we have for  $\sigma \in \text{safe}(f + g)$  and  $\sigma' \in \Sigma$ ,

$$\begin{aligned}
& (f + g)(\{\sigma'\} * \{\sigma\}) \\
&= f(\{\sigma'\} * \{\sigma\}) \sqcup g(\{\sigma'\} * \{\sigma\}) \\
&= \{\sigma'\} * f(\{\sigma\}) \sqcup \{\sigma'\} * g(\{\sigma\}) \\
&\quad (f \text{ and } g \text{ have determinism constancy and } \sigma \in \text{safe}(f) \text{ and } \sigma \in \text{safe}(g) \text{ since } \sigma \in \text{safe}(f + g)) \\
&= \{\sigma'\} * (f(\{\sigma\}) \sqcup g(\{\sigma\})) \quad (\text{distributivity}) \\
&= \{\sigma'\} * (f + g)(\{\sigma\})
\end{aligned}$$

For Kleene-star, we have for  $\sigma \in \text{safe}(f^*)$  and  $\sigma' \in \Sigma$ ,

$$\begin{aligned}
& (f^*)(\{\sigma'\} * \{\sigma\}) \\
&= \bigsqcup f^n(\{\sigma'\} * \{\sigma\}) \\
&= \bigsqcup_n \{\sigma'\} * f^n(\{\sigma\}) \\
&\quad \text{(determinism constancy preserved under sequential composition and } \sigma \in \text{safe}(f^n)) \\
&= \{\sigma'\} * \bigsqcup f^n(\{\sigma\}) \quad \text{(distributivity)} \\
&= \{\sigma'\} * (f^*)(\{\sigma\})
\end{aligned}$$

■

**Proposition 6.** *Let  $H_1$  be the stack and heap model of example 1.2 and  $H_2$  be the alternative model of section 6.1. For their respective semantics in the two models, the commands  $\text{new}[x]$ ,  $\text{mutate}[x, v]$  and  $\text{lookup}[x, y]$  all have determinism constancy in both models.  $\text{dispose}[x]$  has determinism constancy in  $H_2$  but not in  $H_1$ .*

**Proof:** We give the proofs for the new and dispose commands in both models, and the cases for mutate and lookup can be checked in a similar way. For  $\text{dispose}[x]$  in  $H_1$ , the following counterexample shows that it does not have determinism constancy.

$$\begin{aligned}
& \text{dispose}[x](\{l \mapsto v\} * \{x \mapsto l \bullet l \mapsto w\}) \\
&= \text{dispose}[x](\emptyset) \\
&= \emptyset \\
&\sqsubset \{l \mapsto v \bullet x \mapsto l\} \\
&= \{l \mapsto v\} * \text{dispose}[x](x \mapsto l \bullet l \mapsto w)
\end{aligned}$$

For  $\text{new}[x]$  in  $H_1$ , any safe state is of the form  $h \bullet x \mapsto v$ . For any  $h' \in H_1$ , we have

$$\{h'\} * \text{new}[x](h \bullet x \mapsto v) = \{h'\} * \{h \bullet x \mapsto l \bullet l \mapsto w \mid w \in \text{Val}, l \in L \setminus \text{loc}(h)\} \quad (\dagger)$$

If  $h' \bullet h \bullet x \mapsto v$  is undefined then  $h'$  shares locations with  $\text{loc}(h)$  or variables with  $\text{var}(h) \cup \{x\}$ . This means that the RHS in  $\dagger$  is the empty set, and we have  $\text{new}[x](\{h'\} * \{h \bullet x \mapsto v\}) = \text{new}[x](\emptyset) = \emptyset = \{h'\} * \text{new}[x](h \bullet x \mapsto v)$ . Otherwise,

$$\begin{aligned}
& \text{new}[x](\{h'\} * \{h \bullet x \mapsto v\}) \\
&= \text{new}[x](h' \bullet h \bullet x \mapsto v) \\
&= \{h' \bullet h \bullet x \mapsto l \bullet l \mapsto w \mid w \in \text{Val}, l \in L \setminus \text{loc}(h' \bullet h)\} \\
&= \{h'\} * \{h \bullet x \mapsto l \bullet l \mapsto w \mid w \in \text{Val}, l \in L \setminus \text{loc}(h' \bullet h)\} \\
&= \{h'\} * \{h \bullet x \mapsto l \bullet l \mapsto w \mid w \in \text{Val}, l \in L \setminus \text{loc}(h)\} \\
&= \{h'\} * \text{new}[x](h \bullet x \mapsto v)
\end{aligned}$$

For  $\text{dispose}[x]$  in  $H_2$ , any safe state is of the form  $h \bullet x \mapsto l \bullet l \mapsto v \bullet F$ . Let  $h' \in H_2$ . We have

$$\{h'\} * \text{dispose}[x](h \bullet x \mapsto l \bullet l \mapsto v \bullet F) = \{h'\} * \{h \bullet x \mapsto l \bullet F \cup \{l\}\} \quad (\ddagger)$$

If  $h' \bullet h \bullet x \mapsto l \bullet l \mapsto v \bullet F$  is undefined then either  $h'$  contains a free set or it contains locations in  $loc(h) \cup \{l\}$  or variables in  $var(h) \cup \{x\}$ . If  $h'$  contains a free set or it contains locations in  $loc(h)$  or variables in  $var(h) \cup \{x\}$ , then the RHS in  $\dagger\dagger$  is the empty set. If  $h'$  contains the location  $l$  then also the RHS in  $\dagger\dagger$  is the empty set since the free set  $F \cup \{l\}$  also contains  $l$ . Thus in both cases the RHS in  $\dagger\dagger$  is the empty set, and we have  $dispose[x](\{h'\} * \{h \bullet x \mapsto l \bullet l \mapsto v \bullet F\}) = \emptyset = \{h'\} * dispose[x](h \bullet x \mapsto l \bullet l \mapsto v \bullet F)$ .

If  $h' \bullet h \bullet x \mapsto l \bullet l \mapsto v \bullet F$  is defined then we have

$$\begin{aligned} & dispose[x](\{h'\} * \{h \bullet x \mapsto l \bullet l \mapsto v \bullet F\}) \\ &= dispose[x](h' \bullet h \bullet x \mapsto l \bullet l \mapsto v \bullet F) \\ &= \{h' \bullet h \bullet x \mapsto l \bullet F \cup \{l\}\} \\ &= \{h'\} * \{h \bullet x \mapsto l \bullet F \cup \{l\}\} \\ &= \{h'\} * dispose[x](h \bullet x \mapsto l \bullet l \mapsto v \bullet F) \end{aligned}$$

For  $new[x]$  in  $H_2$ , any safe state is of the form  $h \bullet x \mapsto v \bullet F$ . Let  $h' \in H_2$ . We have

$$\{h'\} * new[x](h \bullet x \mapsto v \bullet F) = \{h'\} * \{h \bullet x \mapsto l \bullet l \mapsto w \bullet F \setminus \{l\} \mid w \in Val, l \in F\} \quad (\dagger\dagger\dagger)$$

If  $h' \bullet h \bullet x \mapsto v \bullet F$  is undefined then either  $h'$  contains a free set or it contains locations in  $loc(h)$  or variables in  $var(h) \cup \{x\}$ . In all these cases the RHS in  $\dagger\dagger\dagger$  is the empty set, and so we have  $new[x](\{h'\} * \{h \bullet x \mapsto v \bullet F\}) = \emptyset = \{h'\} * new[x](h \bullet x \mapsto v \bullet F)$ .

If  $h' \bullet h \bullet x \mapsto v \bullet F$  is defined then we have

$$\begin{aligned} & new[x](\{h'\} * \{h \bullet x \mapsto v \bullet F\}) \\ &= new[x](h' \bullet h \bullet x \mapsto v \bullet F) \\ &= \{h' \bullet h \bullet x \mapsto l \bullet l \mapsto w \bullet F \setminus \{l\} \mid w \in Val, l \in F\} \\ &= \{h'\} * \{h \bullet x \mapsto l \bullet l \mapsto w \bullet F \setminus \{l\} \mid w \in Val, l \in F\} \\ &= \{h'\} * new[x](h \bullet x \mapsto v \bullet F) \end{aligned}$$

■

Thus theorem 5 and proposition 6 tell us that in the alternative model of section 6.1, the footprint of every program is given by the smallest safe states, and hence we have regained access footprints for all programs. In fact, the same is true for the original model of example 1.2 if we do not include the dispose command as a primitive command, since all the other primitive commands have determinism constancy. This, for example, would be the case when modelling a garbage collected language [16].

## 7 Conclusion and Future Work

We have defined the footprints of a local function and demonstrated that they are the only essential elements necessary to obtain complete specifications for local Hoare reasoning. For well-founded resource, the footprints are also sufficient, meaning that they do indeed yield the smallest complete specification. We have therefore solved the footprint problem highlighted in the introduction. We have also given results for the non-well-founded models and have introduced the natural class of one-step local functions

for which the footprints are the smallest safe states. Finally, we have introduced an alternative heap model in which access footprints are regained for all programs, and presented a general condition on primitive commands which yields access footprints for all programs in arbitrary models.

Although we now know what footprints are, there is more to investigate about their properties. Two important questions for future work are how footprints behave under the sequential composition of functions (we know from examples that it is not simply the union of the sets of footprints) and how the footprints impact on concurrent reasoning where identifying the minimal resource is important.

The discussion in this paper has been based on the static notion of footprints as *states* of the resource on which a program acts. A different notion of footprint has been described in [10], where footprints are viewed as *traces* of execution of a computation. O’Hearn has described how the *AD* problem is avoided in this more elaborate trace semantics, as the allocation of cells in an execution prevents the framing of those cells. Interestingly, however, our example model from section 6.1 illustrates that it is not essential to move to this more elaborate setting and incorporate dynamic, execution-specific information into the footprint in order to resolve the *AD* problem. In our model, with the explicit representation of free cells in states, one can remain in an extensional semantics and have a purely static, resource-based (rather than execution-based) view of footprints.

## References

1. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O’Hearn, T. Wies and H. Yang. Shape Analysis for Composite Data Structures. In *CAV*, 2007.
2. J. Berdine, C. Calcagno, and P.W. O’Hearn. Smallfoot: Automatic modular assertion checking with separation logic. In *4th FMCO*, 2006.
3. L. Birkedal and H. Yang. Relational parametricity and separation logic. In *10th FOSSACS*, 2007.
4. R. Bornat, C. Calcagno, P. O’Hearn, and M. Parkinson. Permission accounting in separation logic. In *32nd POPL*, 2005.
5. R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. In *21st MFPS*, 2005.
6. S. D. Brookes. A semantics for concurrent separation logic. In *Proceedings of the 15th CONCUR*, 2004.
7. C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *32nd POPL*, 2005.
8. C. Calcagno, P. Gardner, and U. Zarfaty. Local Reasoning about Data Update. In *Gordon Plotkin’s festschrift, ENTCS*, 2007.
9. C. Calcagno, P. O’Hearn, and H. Yang. Local Action and Abstract Separation Logic (Longer version). In *LICS*, 2007.
10. T. Hoare and P. O’Hearn. Separation Logic Semantics of Communicating Processes. In *FICS*, 2008.
11. S. Isthiaq and P. W. O’Hearn. BI as an assertion language for mutable data structures. In *28th POPL*, 2001.
12. C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 1988.
13. P. O’Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 2007. Preliminary version appeared in CONCUR’04.
14. P. O’Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *15th CSL*, 2001.
15. P. W. O’Hearn and D. J. Pym. The logic of bunched implications. In *Bulletin of Symbolic Logic*, 1999.

16. M. Parkinson. When separation logic met Java. In *FTfJP*, 2006.
17. M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *21st LICS*, 2006.
18. D. Pym, P. O’Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. In *Theoretical Computer Science*, 2004.
19. D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logic Series. Kluwer Academic Publishers, 2002.
20. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, 2002.
21. H. Yang and P. O’Hearn. A semantic basis for local reasoning. In *5th FOSSACS*, 2002.