FOOTPRINTS IN LOCAL REASONING [*]

MOHAMMAD RAZA [a] AND PHILIPPA GARDNER [b]

[a,b] Department of Computing,, Imperial College London, 180 Queen's Gate, London SW7 2AZ, UK,
e-mail address: {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 notion of footprint 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. We also present results for the non-well-founded case. Finally, we use this theory of footprints to investigate the conditions under which the footprints correspond to the smallest safe states. We present a new model of RAM in which, unlike the standard model, the footprints of every program correspond to the smallest safe states. We also identify a general condition on the primitive commands of a programming language which guarantees this property for arbitrary models.

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 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 which they call separation algebras. They generalised their specific examples of local imperative commands and memory models in this abstract framework. They introduced Abstract Separation Logic to provide local Hoare reasoning about such 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 *safety 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 that this notion of footprint does not work, since it does not always yield a *complete* specification for the program. Consider the program[1]

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

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:

$$\{emp\}\quad AD\quad \{emp\} \tag{1.1}$$

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

$$\{l \mapsto v\}\quad AD\quad \{l \mapsto v\} \tag{1.2}$$

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:

$$\{l \mapsto v\}\quad AD\quad \{l \rightarrow v \wedge x \neq l\} \tag{1.3}$$

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.

---

[1]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.

In this paper, we introduce the formal definition of the footprint of a local function that does yield a complete specification for the 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 the non-well-founded case, which depend on the presence of *negativity*. A resource has negativity if it is possible to combine two non-unit elements to get the unit, which is like taking two non-empty pieces of resource and joining them to get nothing. For non-well-founded models without negativity, such as heaps with infinitely divisible fractional 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, such as the integers under addition, we show that there do exist smallest complete specifications based on elements that are not essential and hence not footprints.

In the final section, we apply our theory of footprints to the issue of regaining the safety footprints. We address a question that arose from discussions with O'Hearn and Yang, which is whether there is an alternative model of RAM in which the safety footprint does correspond to the actual footprint, yielding complete specifications. We present such a model based on an examination of the cause of the *AD* problem in the original model. We prove that in this new model the footprint of *every* program, including *AD*, does correspond to the safety footprint. Moreover, we identify a general condition on the primitive commands of a programming language which ensures that this property holds in arbitrary models.

A preliminary version of this paper was presented at the FOSSACS 2008 conference. The final section reports on work that is new to this journal version. This paper also contains the proofs which were excluded from the conference paper.

2. BACKGROUND

The discussion in this paper is based on the framework introduced in [9], where the approach of local reasoning about programs with separation logic was generalised to local reasoning about *local* functions that act on an abstract model of resource. Our objective in this work is to investigate the notion of footprint in this abstract setting, and this section gives a description of the underlying framework.

2.1. **Separation Algebras and Local Functions.** We begin by describing separation algebras, which provide a model of resource which generalises over the specific heap models used in separation logic works. Informally, a separation algebra models resource as a set of elements that can be 'glued' together to create larger elements. The 'glueing' operator satisfies properties in accordance with this resource intuition, such as commutativity and associativity, as well as the cancellation property which requires that, if we are given an element and a subelement, then 'ungluing' that subelement gives us a unique element.

Definition 2.1 (Separation Algebra)**.** A **separation algebra** is a cancellative, partial commutative monoid $(\Sigma, \bullet, u)$, where Σ is a set and $\bullet$ is a partial binary operator with unit u. The operator satisfies the familiar axioms of associativity, commutativity and unit, using a partial equality on Σ

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.

We shall sometimes overload notation, using Σ to denote the separation algebra $(\Sigma, \bullet, u)$. Examples of separation algebras include multisets with union and unit $\emptyset$, the natural numbers with addition and unit 0, heaps as finite partial functions from locations to values ([9] and example 2.8), 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 2.8). These examples all have an intuition of resource, with $\sigma_1 \bullet \sigma_2$ intuitively giving more resource than just σ_1 and σ_2 for $\sigma_1, \sigma_2 \neq u$. However, notice that the general notion of a separation algebra also permits examples which may not have this resource intuition, such as $\{a, u\}$ with $a \bullet a = u$. Since our aim is to investigate general properties of local reasoning, our inclination is to impose minimal restrictions on what counts as resource and to work with a simple definition of a separation algebra.

Definition 2.2 (Separateness and substate)**.** Given a separation algebra $(\Sigma, \bullet, u)$, the **separateness** ($\#$) relation between two states $\sigma_0, \sigma_1 \in \Sigma$ is given by $\sigma_0 \# \sigma_1$ iff $\sigma_0 \bullet \sigma_1$ is defined. The **substate** ($\preceq$) relation is given by $\sigma_0 \preceq \sigma_1$ iff $\exists \sigma_2. \sigma_1 = \sigma_0 \bullet \sigma_2$. We write $\sigma_0 \prec \sigma_1$ when $\sigma_0 \preceq \sigma_1$ and $\sigma_0 \neq \sigma_1$.

Lemma 2.3 (Subtraction)**.** *For $\sigma_1, \sigma_2 \in \Sigma$, if $\sigma_1 \preceq \sigma_2$ then there exists a unique element denoted $\sigma_2 - \sigma_1 \in \Sigma$, such that $(\sigma_2 - \sigma_1) \bullet \sigma_1 = \sigma_2$.*

Proof. Existence follows by definition of $\preceq$. For uniqueness, assume there exist $\sigma', \sigma'' \in \Sigma$ such that $\sigma' \bullet \sigma_1 = \sigma_2$ and $\sigma'' \bullet \sigma_1 = \sigma_2$. Then we have $\sigma' \bullet \sigma_1 = \sigma'' \bullet \sigma_1$, and thus by the cancellation property we have $\sigma' = \sigma''$. $\square$

We consider functions on separation algebras that generalise imperative programs operating on heaps. Such programs can behave non-deterministically, and can also *fault*. To model non-determinism, we consider functions from a separation algebra Σ to its powerset $\mathcal{P}(\Sigma)$. To model faulting, we add a special top element $\top$ to the powerset. We therefore consider total functions of the form $f : \Sigma \to \mathcal{P}(\Sigma)^\top$. On any element of Σ, the function can either map to a set of elements, which models *safe* execution with non-deterministic outcomes, or to $\top$, which models a faulting execution. Mapping to the empty set represents divergence (non-termination).

Definition 2.4. The standard subset relation on the powerset is extended to $\mathcal{P}(\Sigma)^\top$ by defining $p \sqsubseteq \top$ for all $p \in \mathcal{P}(\Sigma)^\top$. The binary operator $*$ on $\mathcal{P}(\Sigma)^\top$ is given by

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

$\mathcal{P}(\Sigma)^\top$ is a total commutative monoid under $*$ with unit $\{u\}$.

Definition 2.5 (Function ordering)**.** For functions $f, g : \Sigma \to \mathcal{P}(\Sigma)^\top$, $f \sqsubseteq g$ iff $f(\sigma) \sqsubseteq g(\sigma)$ for all $\sigma \in \Sigma$.

We shall only consider functions that are *well-behaved* in the sense that they act *locally* with respect to resource. For imperative commands on the heap model, the locality conditions were first characterised in [21], where a soundness proof for local reasoning with separation logic was demonstrated for the specific heap model. The conditions identified were

- *Safety monotonicity*: if the command is safe on some heap, then it is safe on any larger heap.
- *Frame property*: if the command is safe on some heap, then in any outcome of applying the command on a larger heap, the additional heap portion will remain unchanged by the command.

In [9], these two properties were amalgamated and formulated for abstract functions on arbitrary separation algebras.

Definition 2.6 (Local Function). A **local function on** Σ is a total function $f : \Sigma \to \mathcal{P}(\Sigma)^{\top}$ which satisfies the **locality condition**:

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

We let $LocFunc$ be the set of local functions on Σ.

Intuitively, we think of a command to be local if, whenever the command executes safely on any resource element, then the command will not 'touch' any additional resource that may be added. Safety monotonicity follows from the above definition because, if f is safe on σ ($f(\sigma) \sqsubset \top$), then it is safe on any larger state, since $f(\sigma' \bullet \sigma) \sqsubseteq \{\sigma'\} * f(\sigma) \sqsubset \top$.

The frame property follows by the fact that the additional state σ' is preserved in the output of $f(\sigma' \bullet \sigma)$. Note, however, that the $\sqsubseteq$ ordering allows for reduced non-determinism on larger states. This, for example, is the case for the AD command from the introduction which allocates a cell, assigns its address to stack variable x, and then deallocates the cell. On the empty heap, its result would allow all possible values for variable x. However, on the larger heap where cell 1 is already allocated, its result would allow all values for x except 1, and we therefore have a more deterministic outcome on this larger state.

Lemma 2.7. *Locality is preserved under sequential composition, non-deterministic choice and Kleene-star, which are defined as*

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

$$(f + g)(\sigma) = f(\sigma) \sqcup g(\sigma)$$

$$f^*(\sigma) = \bigsqcup_n f^n(\sigma)$$

Example 2.8 (Separation algebras and local functions).

(1) **Plain heap model**. A simple example is the separation algebra of heaps $(H, \bullet, u_H)$, where $H = L \rightharpoonup_{fin} Val$ are finite partial functions from a set of locations L to a set of values Val with $L \subseteq Val$, the partial operator $\bullet$ is the union of partial functions with disjoint domains, and the unit u_H is the function with the empty domain. 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 $h_1 - h_2 = h_1|_{dom(h_1)-dom(h_2)}$. 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**. There are two approaches to modelling the stack in the literature. One is to treat the stack as a total function from variables to values, and only combine two heap and stack pairs if the stacks are the same. The other approach, which we use here, is to allow splitting of the variable stack and treat it as part of the resource. We can incorporate the variable stack into the heap model by using the set $H = L \cup Var \rightharpoonup_{fin} Val$, where L and Val are as before and Var is the set of stack variables $\{x, y, z, ...\}$. The $\bullet$ operator combines heap and stack

portions with disjoint domains, and is undefined otherwise. The unit u_H is the function with the empty domain which represents the empty heap and empty stack. 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 using permissions 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 the domain of h respectively. In this model we can define the allocation and deallocation commands as

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

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

Commands for heap mutation and lookup can be defined as

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

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

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

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

Note that in all cases, any stack variables that the command refers to should be in the stack in order for the command to execute safely, otherwise the command will be acting non-locally.

(3) **Integers**. The integers form a separation algebra under addition with identity 0. In this case we have 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 in general. However, the integers under multiplication also form a separation algebra with identity 1, and in this case every multiplying function is local but not every adding function. This illustrates the point that the notion of locality of commands depends on the notion of separation of resource that is being used.

2.2. **Predicates, Specifications and Local Hoare Reasoning.** We now present the local reasoning framework for local functions on separation algebras. This is an adaptation of Abstract Separation Logic [9], with some minor changes in formulation for the purposes of this paper. Predicates over separation algebras are treated simply as subsets of the separation algebra.

Definition 2.9. A **predicate** p over Σ is an element of the powerset $\mathcal{P}(\Sigma)$.

Note that the top element $\top$ is not a predicate and that the $*$ operator, although defined on $\mathcal{P}(\Sigma)^\top \times \mathcal{P}(\Sigma)^\top \to \mathcal{P}(\Sigma)^\top$, acts as a binary connective on predicates. We have the distributive law for union that, for any $X \subseteq \mathcal{P}(\Sigma)$,

$$\left(\bigsqcup X \right) * p = \bigsqcup \{x * p \mid x \in X\}$$

The same is not true for intersection in general, but does hold for *precise* predicates. A predicate is precise if, for any state, there is at most a single substate that satisfies the predicate.

Definition 2.10 (Precise predicate). A predicate $p \in \mathcal{P}(\Sigma)$ is **precise** iff, for every $\sigma \in \Sigma$, there exists at most one $\sigma_p \in p$ such that $\sigma_p \preceq \sigma$.

Thus, with precise predicates, there is at most a unique way to break a state to get a substate that satisfies the predicate. Any singleton predicate $\{\sigma\}$ is precise. Another example of a precise predicate is $\{l \mapsto v \mid v \in Val\}$ for some l, while $\{l \mapsto v \mid l \in L\}$ for some v is not precise.

Lemma 2.11 (Precision characterization). *A predicate p is precise iff, for all $X \subseteq \mathcal{P}(\Sigma)$, $(\bigsqcap X) * p = \bigsqcap \{x * p \mid x \in X\}$*

Proof. We first show the left to right direction. Assume p is precise. We have to show that for all $X \subseteq \mathcal{P}(\Sigma)$, $(\bigsqcap X) * p = \bigsqcap \{x * p \mid x \in X\}$. Assume $\sigma \in (\bigsqcap X) * p$. Then there exist σ_1, σ_2 such that $\sigma = \sigma_1 \bullet \sigma_2$ and $\sigma_1 \in \bigsqcap X$ and $\sigma_2 \in p$. Thus for all $x \in X$, $\sigma \in x * p$, and hence $\sigma \in \bigsqcap \{x * p \mid x \in X\}$. Now assume $\sigma \in \bigsqcap \{x * p \mid x \in X\}$. Then $\sigma \in x * p$ for all $x \in X$. Hence there exists $\sigma_1 \preceq \sigma$ such that $\sigma_1 \in p$. Since p is precise, σ_1 is unique. Let $\sigma_2 = \sigma - \sigma_1$. Thus we have $\sigma_2 \in x$ for all $x \in X$, and so $\sigma_2 \in \bigsqcap X$. Hence we have $\sigma \in (\bigsqcap X) * p$.

For the other direction, we assume that p is not precise and show that there exists an X such that $(\bigsqcap X) * p \neq \bigsqcap \{x * p \mid x \in X\}$. Since p is not precise, there exists $\sigma \in \Sigma$ such that, for two distinct $\sigma_1, \sigma_2 \in p$, we have $\sigma_1 \preceq \sigma$ and $\sigma_2 \preceq \sigma$. Let $\sigma_1' = \sigma - \sigma_1$ and $\sigma_2' = \sigma - \sigma_2$. Now let $X = \{\{\sigma_1'\}, \{\sigma_2'\}\}$. Since $\sigma \in \{\sigma_1'\} * p$ and $\sigma \in \{\sigma_2'\} * p$, we have $\sigma \in \bigsqcap \{x * p \mid x \in X\}$. However, because of the cancellation property, we also have that $\sigma_1' \neq \sigma_2'$, and so $(\bigsqcap X) * p = \emptyset * p = \emptyset$. Hence, $\sigma \notin (\bigsqcap X) * p$, and we therefore have $(\bigsqcap X) * p \neq \bigsqcap \{x * p \mid x \in X\}$. $\square$

Our Hoare reasoning framework is formulated with tuples of pre- and post- conditions, rather than the usual Hoare triples that include the function as in [9]. In our case the standard triple shall be expressed as a function f *satisfying* a tuple (p, q), written $f \models (p, q)$. The reason for this is that we shall be examining the properties that a pre- and post- condition tuple may have with respect to a given function, such as whether a given tuple is complete for a given function. 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 2.12 (Specification). Let Σ be a separation algebra. A **statement** on Σ is a tuple (p, q), where $p, q \in \mathcal{P}(\Sigma)$ are predicates. A **specification** ϕ on Σ is a set of statements. We let $\Phi_\Sigma = \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$ denote the set of all specifications on Σ. 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 Σ the specification is referring to. We also adapt the notion of precise predicates to specifications.

Definition 2.13. A specification is precise iff its domain is precise.

Definition 2.14 (Satisfaction). A local function f satisfies a statement (p, q), written $f \models (p, q)$, iff, for all $\sigma \in p$, $f(\sigma) \sqsubseteq q$. It satisfies a specification $\phi \in \Phi$, written $f \models \phi$, iff $f \models (p, q)$ for all $(p, q) \in \phi$.

Definition 2.15 (Semantic consequence). Let $p, q, r, s \in \mathcal{P}(\Sigma)$ 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 2.16 (Order Characterization). $f \sqsubseteq g$ *iff, for all $p, q \in \mathcal{P}(\Sigma)$, $g \models (p, q)$ implies $f \models (p, q)$.* $\square$

For every specification ϕ, there is a 'best' local function satisfying ϕ (lemma 2.18), in the sense that all statements that the best local function satisfies are satisfied by any local function that satisfies ϕ. For example, in the heap and stack separation algebra of example 2.8.2, consider the specification

$$\phi_{new} = \{(\{x \mapsto v\}, \{x \mapsto l \bullet l \mapsto w \mid l \in L, w \in Val\}) \mid v \in Val\}$$

There are many local functions that satisfy this specification. Trivially, the local function that always diverges satisfies it. Another example is the local function that assigns the value w of the newly allocated cell to be 0, rather than any non-deterministically chosen value. However, the best local function for this specification is the $new[x]$ function described in example 2.8.2, as it can be checked that for any local function f satisfying ϕ_{new}, we have $f \sqsubseteq new[x]$. The notion of the best local function shall be used when addressing questions about completeness of specifications. It is adapted from [9], except that we generalise to the best local function of a specification rather than a single pre- and post-condition pair.

Definition 2.17 (Best local function). For a specification $\phi \in \Phi$, the best local function of ϕ, written $bla[\phi]$, is the function of type $\Sigma \to \mathcal{P}(\Sigma)^\top$ defined by

$$bla[\phi](\sigma) = \bigsqcap\{\{\sigma'\} * q \mid \sigma = \sigma' \bullet \sigma'', \sigma'' \in p, (p, q) \in \phi\}$$

As an example, it can be checked that the best local function $bla[\phi_{new}]$ of the specification ϕ_{new} given above is indeed the function $new[x]$ described in example 2.8.2. The following lemma presents the important properties which characterise the best local function.

Lemma 2.18. *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 σ_1, σ_2 such that $\sigma_1 \# \sigma_2$. We then calculate

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

In the second-last step we used the property that $\{\sigma_1\}$ is precise (lemma 2.11).

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

For the last point, suppose f is local and $f \models \phi$. Then, for any σ 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 $f(\sigma) \sqsubseteq bla[\phi](\sigma)$.

In the case that there do not exist σ_1, σ_2 such that $\sigma = \sigma_1 \bullet \sigma_2$ and $\sigma_2 \in D(\phi)$, then

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

So in this case also $f(\sigma) \sqsubseteq bla[\phi](\sigma)$. $\square$

$$\frac{(p,q)}{(p*r,q*r)} \qquad \frac{p' \sqsubseteq p \quad (p,q) \quad q \sqsubseteq q'}{(p',q')} \qquad \frac{(p_i,q_i),\text{ all } i \in I}{\left(\bigsqcup_{i \in I} p_i, \bigsqcup_{i \in I} q_i\right)} \qquad \frac{(p_i,q_i),\text{ all } i \in I, I \neq \emptyset}{\left(\bigsqcap_{i \in I} p_i, \bigsqcap_{i \in I} q_i\right)}$$

$$Frame \qquad\qquad Consequence \qquad\qquad Union \qquad\qquad Intersection$$

Figure 1: Inference rules for local Hoare reasoning

Lemma 2.19. *For $\phi \in \Phi$ and $p,q \in \mathcal{P}(\Sigma)$, $bla[\phi] \models (p,q) \Leftrightarrow \phi \models (p,q)$.*

Proof.

$$
\begin{aligned}
&\quad bla[\phi] \models (p,q) \\
\Leftrightarrow \quad &\text{for all local functions } f,\ f \models \phi \Rightarrow f \models (p,q) \quad \text{(by lemma 2.18)} \\
\Leftrightarrow \quad &\phi \models (p,q) \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{(by definition 2.15).}
\end{aligned}
$$
$\square$

The inference rules of the proof system are given in figure 1. Consequence, union and intersection are adaptations of standard rules of Hoare logic. The frame rule is what permits local reasoning, as it codifies the fact that, since all functions are local, any assertion about a separate part of resource will continue to hold for that part after the application of the function. We omit the standard rules for basic constructs such as sequential composition, non-deterministic choice, and Kleene-star which can be found in [9].

Definition 2.20 (Proof-theoretic consequence)**.** For predicates p,q,r,s and specifications ϕ,ψ, 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.

The proof system of figure 1 is sound and complete with respect to the satisfaction relation.

Theorem 2.21 (Soundness and Completeness)**.** $\phi \vdash (p,q) \Leftrightarrow \phi \models (p,q)$

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 2.19, 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:

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

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 $bla[\phi](\sigma) \sqsubseteq q$). $\qquad\square$

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, which is a specification from which follows every property that holds for the function. 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 3.1 (Complete Specification). A specification $\phi \in \Phi$ is a **complete specification** for f, written $complete(\phi, f)$, iff, for all $p, q \in \mathcal{P}(\Sigma), f \models (p, q) \Leftrightarrow \phi \models (p, q)$. Let $\Phi_{comp(f)}$ be the set of all complete specifications of f.

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

Proposition 3.2. *For all $\phi \in \Phi$ and local functions f, $complete(\phi, f) \Leftrightarrow f = bla[\phi]$.*

Proof. Assume $f = bla[\phi]$. Then, by lemma 2.19, we have that ϕ is a complete specification for f.

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

case 1: $f(\sigma) = \top$. If $bla[\phi](\sigma) \neq \top$, then $bla[\phi] \models (\{\sigma\}, bla[\phi](\sigma))$. This means that $\phi \models (\{\sigma\}, bla[\phi](\sigma))$ (by lemma 2.19), and so $f \models (\{\sigma\}, bla[\phi](\sigma))$, but this is a contradiction. Therefore, $bla[\phi](\sigma) = \top$

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

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

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

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

Therefore $f(\sigma) = bla[\phi](\sigma)$ $\qquad\square$

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. For example, if ϕ is a complete specification for f and $(p, q) \in \phi$, then $\phi \cup \{(p, q')\}$ is also complete for f if $q \subseteq q'$. For this reason it will be useful to have a canonical form for specifications.

Definition 3.3 (Canonicalisation)**.** The **canonicalisation** of a specification ϕ is defined as $\phi_{can} = \{(\{\sigma\}, bla[\phi](\sigma)) \mid \sigma \in D(\phi)\}$. 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.

Notice that a given local function does not necessarily have a *unique* canonical complete specification. For example, both $\{(\{u\}, \{u\})\}$ and $\{(\{u\}, \{u\}), (\{\sigma\}, \{\sigma\})\}$, for some $\sigma \in \Sigma$, are canonical complete specifications for the identity function.

Proposition 3.4. *For any specification ϕ, we have $\phi \dashv\vDash \phi_{can}$.*

Proof. We first show $\phi \vDash \phi_{can}$. For any $(p, q) \in \phi_{can}$, (p, q) is of the form $(\{\sigma\}, bla[\phi](\sigma))$ for some $\sigma \in D(\phi)$. So we have $bla[\phi] \vDash (p, q)$, and so $\phi \vDash (p, q)$ by lemma 2.19.

We now show $\phi_{can} \vDash \phi$. For any $(p, q) \in \phi$, we have $bla[\phi] \vDash (p, q)$. 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:

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

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

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 3.5. $\Phi_{can(f)}$ *is isomorphic to the quotient set $\Phi_{comp(f)} / \cong_D$, under the isomorphism that maps $[\phi]_{\cong_D}$ to ϕ_{can}, for every $\phi \in \Phi_{comp(f)}$.*

Proof. By proposition 3.2, 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 ϕ is also complete, and $[\phi]_{\cong_D}$ maps to $\phi_{can} = \phi$, so the mapping is surjective. $\square$

Definition 3.6 (Small and Big specifications)**.** ϕ 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 3.7 (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 $p, q \in \mathcal{P}(\Sigma)$. 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

$$\cfrac{\cfrac{\cfrac{\phi_{big(f)}}{(\{\sigma\}, f(\sigma))} \quad \text{for all } f(\sigma) \sqsubset \top}{(\bigsqcup_{\sigma \in p}\{\sigma\}, \bigsqcup_{\sigma \in p} f(\sigma))}}{(p, q)}$$

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 $\phi_{big(f)}$. $\qquad\square$

The notion of a small specification has until now been used in an informal sense in local reasoning papers [14, 4, 7] as specifications that completely specify the behaviour of an update command by only describing the command's behaviour on the part of the resource that it affects. Although these papers present examples of such specifications for specific commands, the notion has so far not received a formal treatment in the general case. The question of the existence of small specifications is strongly related to the concept of footprints, since finding a small specification is about finding a complete specification with the smallest possible domain, and therefore enquiring about which elements of Σ 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 definition of locality (definition 2.6) says that the action on a certain state σ_1 imposes a *limit* on the action on a bigger state $\sigma_2 \bullet \sigma_1$. This limit is $\{\sigma_2\} * f(\sigma_1)$, as we have $f(\sigma_2 \bullet \sigma_1) \sqsubseteq \{\sigma_2\} * f(\sigma_1)$.

Another way of viewing this definition is that for any state σ, the action of the function on that state has to be within the limit imposed by *every* substate σ' of σ, that is, $f(\sigma) \sqsubseteq \{\sigma - \sigma'\} * f(\sigma')$. In the case where $\sigma' = \sigma$, this condition is trivially satisfied for any function (local or non-local). The distinguishing characteristic of local functions is that this condition is also satisfied by every strict substate of σ, and thus we have

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

We define this overall constraint imposed on σ by all of its strict substates as the *local limit* of f on σ, and show that the locality definition is equivalent to satisfying the local limit constraint.

Definition 4.1 (Local limit). For a local function f on Σ and $\sigma \in \Sigma$, the **local limit** of f on σ is defined as

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

Proposition 4.2. f is local $\quad\Leftrightarrow\quad f(\sigma) \sqsubseteq L_f(\sigma) \quad$ for all $\sigma \in \Sigma$

Proof. Assume f is local. So for any σ, 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 $f(\sigma_1 \bullet \sigma_2) \sqsubseteq L_f(\sigma_1 \bullet \sigma_2) \sqsubseteq \{\sigma_1\} * f(\sigma_2)$. □

Thus for any local function f acting on a certain state σ, the local limit determines a *smallest upper bound* on the possible outcomes on σ, based on the outcomes on all smaller states. If this smallest upper bound does correspond exactly to the set of all possible outcomes on σ, then σ is 'large enough' that just the action of f on smaller states and the locality of f determines the complete behaviour of f on σ. In this case we will not think of σ as a footprint of f, as smaller states are sufficient to determine the action of f on σ. With this observation, we define footprints as those states on which the outcomes cannot be determined only by the smaller states, that is, the set of outcomes is a *strict* subset of the local limit.

Definition 4.3 (Footprint). For a local function f and $\sigma \in \Sigma$, σ 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 σ is therefore not a footprint if and only if the action of f on σ is at the local limit, that is $f(\sigma) = L_f(\sigma)$.

Lemma 4.4. *For any local function f, the smallest safe states of f are footprints of f.*

Proof. Let σ 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 4.5 (Dispose). The footprints of the $dispose[l]$ command in the plain heap model (example 2.8.1) 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

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

(3) Every state σ 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 4.2, we have $L_{dispose[l]}(\sigma) = dispose[l](\sigma)$. The intuition is that σ does not characterise any 'new' behaviour of the function: its action on σ is just a consequence of its action on the cells at location l and the locality property of the function.

(4) Every state σ such that $\sigma \not\succ (l \mapsto v)$ 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 4.2, $L_{dispose[l]}(\sigma) = dispose[l](\sigma)$.

Example 4.6 (*AD* command). The *AD* (Allocate-Deallocate) command was defined on the heap and stack model in example 2.8.2. We have the following cases for σ.

(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.

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

By proposition 4.2, 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 4.7 (Essentiality). *The footprints of a local function are the essential domain elements for any complete specification of that function, that is,*

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

Proof. Assume some fixed f and σ. We establish the following equivalent statement :

$$\neg F_f(\sigma) \quad \Leftrightarrow \quad \exists \phi \in \Phi_{comp(f)}. \, \sigma \notin D(\phi)$$

We first show the right to left implication. So assume ϕ is a complete specification of f such that $\sigma \notin D(\phi)$. Since $complete(\phi, f)$, by proposition 3.2, 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. Since $\sigma_1 \in p$, we have $f(\sigma_1) \sqsubseteq q$, 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 4.2 that $f(\sigma) \sqsubseteq L_f(\sigma)$, so we get $f(\sigma) = L_f(\sigma)$ and therefore $\neg F_f(\sigma)$.

We now show the left to right implication. Assume that σ is not a footprint of f. We shall use the big specification, $\phi_{big(f)}$, to construct a complete specification of f which does not contain σ in its domain. 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 ϕ 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. The right to left direction, $\phi \dashv \phi_{big(f)}$, is trivial.

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

$$
\dfrac{\dfrac{\dfrac{\dfrac{\phi}{(\{\sigma'\}, f(\sigma')) \quad \text{for all } \sigma' \prec \sigma,\, f(\sigma') \sqsubset \top}}{(\{\sigma - \sigma'\} * \{\sigma'\}, \{\sigma - \sigma'\} * f(\sigma')) \quad \text{for all } \sigma' \prec \sigma,\, f(\sigma') \sqsubset \top}}{(\{\sigma\},\ \displaystyle\prod_{\sigma' \prec \sigma,\, f(\sigma') \sqsubset \top} \{\sigma - \sigma'\} * f(\sigma'))}}{(\{\sigma\}, L_f(\sigma))}
$$

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 σ' then σ would be a footprint, which is a contradiction. Note that the last step uses the fact that

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

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

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.** In the last section we defined the local limit of a function f on a state σ as the constraint imposed on f by all the strict substates of σ. This was used to identify the footprints as those states on which the action of f cannot be determined by just its action on the smaller states. We are now addressing the question of when a set of states is *sufficient* to determine the behaviour of f on any state. We shall do this by identifying a fixed set of states, which we call a *basis* for f, such that the action of f on any state σ can be determined by just the substates of σ taken from this set (rather than all the strict substates of σ). Thus we first generalise the local limit definition to consider the constraint imposed by only the substates taken from a given set.

Definition 5.1 (Local limit imposed by a set)**.** For a subset A of a separation algebra Σ, the **local limit** imposed by A on the action of f on σ is defined by

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

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

Definition 5.2 (Basis). $A \sqsubseteq \Sigma$ is a **basis** for f, written $basis(A, f)$, 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 Σ by just using the locality property of f. Every local function has at least one basis, namely the trivial basis Σ itself. We next show the correspondence between the bases and complete specifications of a local function.

Lemma 5.3. *Let* $\phi_{A,f} = \{(\{\sigma\}, f(\sigma)) \mid \sigma \in A, f(\sigma) \sqsubset \top\}$. *Then we have* $basis(A, f) \Leftrightarrow complete(\phi_{A,f}, f)$.

Proof. We have $L_{A,f} = bla[\phi_{A,f}]$ by definition. The result follows by proposition 3.2 and the definition of basis. $\qquad\square$

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 3.5). By the essentiality of footprints (theorem 4.7), it follows that the footprints are present in every basis of a local function.

Lemma 5.4. *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 (4.7), the domain includes the footprints. $\qquad\square$

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 5.5 (Sufficiency I). *If a separation algebra Σ 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 Σ is well-founded under $\preceq$. We shall show by induction that $f(\sigma) = L_{F(f),f}(\sigma)$ for all $\sigma \in \Sigma$. The induction hypothesis is that, for all $\sigma' \prec \sigma$, $f(\sigma') = L_{F(f),f}(\sigma')$

case 1: Assume σ is a footprint of f. 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: Assume σ is not a footprint of f. We have

$$f(\sigma) = L_f(\sigma) \quad \textit{(because σ is not a footprint of f)}$$

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

$$= \prod_{\sigma' \prec \sigma} \left(\{\sigma - \sigma'\} * \prod_{\sigma'' \preceq \sigma', F_f(\sigma'')} \{\sigma' - \sigma''\} * f(\sigma'') \right) \quad \textit{(by the induction hypothesis)}$$

$$= \prod_{\sigma' \prec \sigma, \sigma'' \preceq \sigma', F_f(\sigma'')} \{\sigma - \sigma'\} * \{\sigma' - \sigma''\} * f(\sigma'') \quad \textit{(by the precision of $\{\sigma - \sigma'\}$)}$$

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

$$= \prod_{\sigma'' \preceq \sigma, F_f(\sigma'')} \{\sigma - \sigma''\} * f(\sigma'') \quad \textit{(because σ is not a footprint of f)}$$

$$= L_{F(f),f}(\sigma) \qquad\qquad \square$$

In section 3, the notions of big and small specifications were introduced (definition 3.6), and the existence of a big specification was shown (proposition 3.7). We are now in a position to show the existence of the small specification for well-founded resource. If Σ is well-founded, then every local function has a small specification whose domain is the footprints of the function.

Corollary 5.6 (Small specification). *For well-founded separation algebras, every local function has a small specification given by $\phi_{F(f),f}$.*

Proof. $\phi_{F(f),f}$ is complete by theorem 5.5 and lemma 5.3. It has the smallest domain by the essentiality theorem. $\square$

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 the canonical one given by $\{(\{l \mapsto v\}, \{u_H\}) \mid v \in Val\}$.

In practical examples it is usually the case that resource is well-founded. A notable exception is the fractional permissions model [4] in which the resource includes 'permissions to access', which can be indefinitely divided. We next investigate the non-well-founded case.

5.3. **Non-well-founded Resource.** If a separation algebra is non-well-founded under the $\preceq$ relation, then there is some infinite descending chain of elements $\sigma_1 \succ \sigma_2 \succ \sigma_3....$ From a resource-oriented point of view, there are two distinct ways in which this could happen. One way is when it is possible to remove non-empty pieces of resource from a state indefinitely, as in the separation algebra of non-negative real numbers under addition. In this case any infinite descending chain does not have more than one occurrence of any element. Another way is when an infinite chain may exist because of repeated occurrences of some elements. This happens when there is *negativity* present in the resource: some elements have inverses in the sense that adding two non-unit elements together may give the unit. An example is the separation algebra of integers under addition, where $1 + (-1) = 0$, so adding -1 to 1 is like adding negative resource. Also, since $1 = 0 + 1$, we have that $1 \succ 0 \succ 1...$ forms an infinite chain.

Definition 5.7 (Negativity)**.** A separation algebra Σ 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 Σ is **non-negative** if no such element exists.

All separation algebras with negativity are non-well-founded because, for elements σ and σ' 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. For the general non-negative case, we find that either the footprints form a basis, or there is no smallest basis.

Theorem 5.8 (Sufficiceny II)**.** *If Σ 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 Σ itself). If A is the set of footprints then we are done. So assume A contains some non-footprint μ. 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 $\sigma \in \Sigma$.

case 1: $\mu \npreceq \sigma$. We have

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

as desired

case 2: $\mu \preceq \sigma$. This implies

$$f(\sigma) = \Big(\bigsqcap_{\sigma' \preceq \sigma, \sigma' \in A/\{\mu\}} \{\sigma - \sigma'\} * f(\sigma') \Big) \ \sqcap \ (\{\sigma - \mu\} * f(\mu))$$

It remains to show that the right hand side of this intersection contains the left hand side:

$\{\sigma - \mu\} * f(\mu) = \{\sigma - \mu\} * L_f(\mu)$ (because μ is not a footprint of f)

$$= \{\sigma - \mu\} * \bigsqcap_{\sigma' \prec \mu} \{\mu - \sigma'\} * f(\sigma')$$

$$= \{\sigma - \mu\} * \bigsqcap_{\sigma' \prec \mu} \Big(\{\mu - \sigma'\} * \bigsqcap_{\sigma'' \preceq \sigma', \sigma'' \in A/\{\mu\}} \{\sigma' - \sigma''\} * f(\sigma'') \Big)$$

(case 1 applies because Σ is non-negative, so $\sigma' \prec \mu \Rightarrow \mu \npreceq \sigma'$)

$$= \bigsqcap_{\sigma' \prec \mu} \bigsqcap_{\sigma'' \preceq \sigma', \sigma'' \in A/\{\mu\}} \{\sigma - \mu\} * \{\mu - \sigma'\} * \{\sigma' - \sigma''\} * f(\sigma'') \quad \text{(by precision)}$$

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

$$= \bigsqcap_{\sigma'' \prec \mu, \sigma'' \in A/\{\mu\}} \{\sigma - \sigma''\} * f(\sigma'')$$

$$\sqsupseteq \bigsqcap_{\sigma'' \preceq \sigma, \sigma'' \in A/\{\mu\}} \{\sigma - \sigma''\} * f(\sigma'') \qquad\qquad \square$$

Corollary 5.9 (Small Specification)**.** *If Σ 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 5.10 (Permissions)**.** The fractional permissions model [4] is non-well-founded and non-negative. It can be represented by the separation algebra $HPerm = L \rightharpoonup_{fin} Val \times P$ where L

and Val are as in example 2.8, 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. The operator $\bullet$ 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; 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 4.7 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 5.11 (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 $f : \mathbb{Z} \to \mathcal{P}(\mathbb{Z})^\top$ be defined as $f(n) = \{n + c\}$ for some constant c, as in example 2.8. 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 Safety Footprints

In the introduction we discussed how the notion of footprints as the smallest safe states - the *safety footprint*- is inadequate for giving complete specifications, as illustrated by the *AD* example. For this reason, so far in this paper we have investigated the general notion of footprint for arbitrary local functions on arbitrary separation algebras. Equipped with this general theory, we now investigate how the regaining of safety footprints may be achieved with different resource modelling choices. We start by presenting an alternative model of RAM, based on an investigation of why the AD phenomenon occurs in the standard model. We then demonstrate that the footprints of the AD command in this new model do correspond to the safety footprints. In the final section we identify, for arbitrary separation algebras, a condition on local functions which guarantees the equivalence of the safety footprint and the actual footprint. We then show that if this condition is met by all the primitive commands of a programming language then the safety footprints are regained for every program in the language, and finally show that this is indeed the case in our new RAM model.

6.1. **An alternative model.** In this section we explore an alternative heap model in which the safety 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 2.8.2. Consider an application of the allocation command in this model:

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

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 $l \mapsto v$ will represent a local allocated region of the heap at address l with value v. However, unlike before, this state will say nothing about the allocation status any locations other than l. This information about the allocation status of other locations will be represented explicitly in a *free* set, which will contain every location that is not allocated in the *global heap*. The model can be interpreted from an ownership point of view, where the free set is to be thought of as a unique, atomic piece of resource, ownership of which needs to be obtained by a command if it wants to do allocation or deallocation. An analogy is with the permissions model: a command that wants to read or write to a cell needs ownership of the appropriate permission on that cell. In the same way, in our new model, a command that wants to do allocation or deallocation needs to have ownership of the free set: the 'permission' to see which cells are free in the global heap so that it can choose one of them to allocate, or update the free set with the address that it deallocates. On the other hand, commands that only read or write to cells shall not require ownership of the free set.

Example 6.1 (Heap model with free set). Formally, we work with a separation algebra $(H, \bullet, u_H)$. Let L, Var and Val be locations, variables and values, as before. 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 $F \in \mathcal{P}(L)$. The operator $\bullet$ 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. 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 any h' and F'. We assume $\bullet$ is associative and commutative with unit u_H.

In this model, the allocation command requires ownership of the free set for safe execution, since 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
$$new[x](h) = \left\{ \begin{array}{ll} \{h' \bullet x \mapsto l \bullet l \mapsto w \bullet F \backslash \{l\} \mid w \in Val, l \in F\} & h = h' \bullet x \mapsto v \bullet F \\ \top & \text{otherwise} \end{array} \right.$$
Note that the output states $h' \bullet x \mapsto l \bullet l \mapsto w \bullet F \backslash \{l\}$ are defined, since we have $l \notin F \backslash \{l\}$ and the input state $h' \bullet x \mapsto v \bullet F$ implies that $loc(h')$ is disjoint from $F \backslash \{l\}$. The deallocation command

also requires the free set, as it updates the set with the 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 model, only the allocation and deallocation commands require ownership of the free set, since commands such as mutation and lookup are completely independent of the allocation status of other cells, and they are defined exactly as in example 2.8.2:

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

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

Lemma 6.2. *The functions $new[x]$, $dispose[x]$, $mutate[x,v]$ and $lookup[x,y]$ are all local in the separation algebra $(H, \bullet, u_H)$ from example 6.1.*

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

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

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

6.2. Safety footprints for *AD*.

We consider the footprint of the *AD* command in the new model. In this model the sequential composition $new[x]; dispose[x]$ 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 $\{x \mapsto v \bullet F \mid v \in Val, F \in \mathcal{P}(L)\}$. By lemma 4.4, 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 $h \bullet x \mapsto v \bullet F$ for non-empty h. 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 $L_{AD}(h \bullet x \mapsto v \bullet F) \sqsubseteq \{h\} * AD(x \mapsto v \bullet F)$ by definition, we have by proposition 4.2 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.

Thus the footprints of AD in this model do not include any non-empty heaps. By corollary 5.6, in this model the AD command has a smallest complete specification in which the pre-condition only describes the empty heap. This specification is

$$\{(\{x \mapsto v \bullet F\}, \{x \mapsto l \bullet F\}) \mid v \in Val, F \in \mathcal{P}(L), l \in F\}$$

Intuitively, it says that if initially the heap is empty, the variable x is present in the stack, and we know which cells are free in the global heap, then after the execution, the heap will still be empty,

$$\llbracket c \rrbracket \in LocFunc \qquad \llbracket \mathtt{skip} \rrbracket(\sigma) = \{\sigma\}$$

$$\llbracket C_1; C_2 \rrbracket = \llbracket C_1 \rrbracket; \llbracket C_2 \rrbracket \qquad \llbracket C_1 + C_2 \rrbracket = \llbracket C_1 \rrbracket \sqcup \llbracket C_2 \rrbracket \qquad \llbracket C^\star \rrbracket = \bigsqcup_n \llbracket C^n \rrbracket$$

Figure 2: Denotational semantics for the imperative programming language

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 using the frame rule. For example, we get the complete specification on the larger state in which 42 is allocated:

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

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.

Notice that in order to check that we have 'regained' safety footprints, we only needed to check that the footprint definition (definition 4.3) corresponds to the smallest safe states. The desired properties such as essentiality, sufficiency, and small specifications then follow by the results established in previous sections.

6.3. **Safety footprints for arbitrary programs.** Now that we have regained the safety footprints for *AD* in the new model, we want to know if this is generally the case for *any program*. We consider the abstract imperative programming language given in [9]:

$$C \quad ::= \quad c \mid \mathtt{skip} \mid C; C \mid C + C \mid C^\star$$

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

Taking the primitive commands to be $new[x]$, $dispose[x]$, $mutate[x, v]$, and $lookup[x, y]$, 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 Σ. If, for $A \in \mathcal{P}(\Sigma)$, we define $f(A) = \bigsqcup_{\sigma \in A} f(\sigma)$, then the locality condition (definition 2.6) can be restated as

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

The $\sqsubseteq$ ordering in this definition allows local functions to be more deterministic on larger states. This sensitivity of determinism to larger states is apparant in the *AD* command in the standard model from example 2.8.2. On the empty heap, the command produces an empty heap, and reassigns variable x to *any* value, while on the singleton cell 1, it disallows the possibility that $x = 1$ afterwards. In the new model, the AD command does not have this sensitivity of determinism in the output states. In this case, the presence or absence of the cell 1 does not affect the outcomes of the AD command, since the command can only assign x to a value chosen from the free set, which does not change no matter what additional cells may be framed in. With this observation, we consider the general class of local functions in which this sensitivity of determinism is not present.

Definition 6.3 (Determinism Constancy). Let f be a local function and $safe(f)$ the set of states on which f does not fault. f has the determinism constancy property iff, for every $\sigma \in safe(f)$,

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

Notice that the determinism constancy property by itself implies that the function is local, and it can therefore be thought of as a form of 'strong locality'. Firstly, we find that local functions that have determinism constancy always have footprints given by the smallest safe states.

Lemma 6.4. *If a local function f has determinism constancy then its footprints are the smallest safe states.*

Proof. Let $min(f)$ be the smallest safe states of f. These are footprints by lemma 4.4. For any larger state $\sigma' \bullet \sigma$ where $\sigma \in min(f)$, $\sigma' \in \Sigma$ and σ is non-empty, we have

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

Since $L_f(\sigma' \bullet \sigma) \sqsubseteq \{\sigma'\} * f(\sigma)$, by proposition 4.2 we have that $L_f(\sigma' \bullet \sigma) = f(\sigma' \bullet \sigma)$, and so $\sigma' \bullet \sigma$ is not a footprint of f. $\qquad \square$

We now demonstrate that the determinism constancy property is preserved by all the constructs of our programming language. This implies that if all the primitive commands of the programming language have determinism constancy, then the footprints of every program are the smallest safe states.

Theorem 6.5. *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 composite command has determinism constancy and the result follows by lemma 6.4. So for commands C_1 and C_2, let $f = [\![C_1]\!]$ and $g = [\![C_2]\!]$ and assume f and g have determinism constancy. For sequential composition we have, for $\sigma \in safe(f;g)$ and $\sigma' \in \Sigma$,

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

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

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

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

$$(f^\star)(\{\sigma'\} * \{\sigma\})$$
$$= \bigsqcup_n f^n(\{\sigma'\} * \{\sigma\})$$
$$= \bigsqcup_n \{\sigma'\} * f^n(\{\sigma\}) \qquad \text{(determinism constancy preserved under sequential composition and}$$
$$\sigma \in safe(f^n))$$
$$= \{\sigma'\} * \bigsqcup_n f^n(\{\sigma\}) \qquad \text{(distributivity)}$$
$$= \{\sigma'\} * (f^\star)(\{\sigma\}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad \square$$

Now that we have shown the general result, it remains to check that all the primitive commands in the new model of section 6.1 do have determinism constancy.

Proposition 6.6. *Let H_1 be the stack and heap model of example 2.8.2 and H_2 be the alternative model of section 6.1. The commands $new[x]$, $mutate[x, v]$ and $lookup[x, y]$ all have determinism constancy in both models. The $dispose[x]$ command has determinism constancy in H_2 but not in H_1.*

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

$$dispose[x](\{l \mapsto v\} * \{x \mapsto l \bullet l \mapsto w\})$$
$$= dispose[x](\emptyset)$$
$$= \emptyset$$
$$\sqsubseteq \{l \mapsto v \bullet x \mapsto l\}$$
$$= \{l \mapsto v\} * dispose[x](x \mapsto l \bullet l \mapsto w)$$

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

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

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

$\emptyset = \{h'\} * new[x](h \bullet x \mapsto v)$. If $h' \bullet h \bullet x \mapsto v$ is defined, then

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

For $dispose[x]$ 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

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

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

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

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

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

$\square$

Thus theorem 6.5 and proposition 6.6 tell us that using the alternative model of example 6.1, the footprint of every program is given by the smallest safe states, and hence we have regained safety footprints for all programs. In fact, the same is true for the original model of example 2.8.2 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. Conclusions

We have developed a general theory of footprints in the abstract setting of local functions that act on separation algebras. Although central and intuitive concepts in local reasoning, the notion of footprints and small specifications had evaded a formal general treatment until now. The main obstacle was presented by the *AD* problem, which demonstrated the inadequacy of the safety footprint notion in yielding complete specifications. In addressing this issue, we first investigated the notion of footprint which does not suffer from this inadequacy. Based on an analysis of the definition of locality, we introduced the definition of the footprint of a local function, and demonstrated that, according to this definition, the footprints are the only essential elements necessary to obtain a complete specification of the function. For well-founded resource models, we showed that the footprints are also sufficient, and we also presented results for non-well-founded models.

Having established the footprint definition, we then explored the conditions under which the safety footprint does correspond to the actual footprint. We introduced an alternative heap model in which safety footprints are regained for *every* program, including *AD*. We also presented a general condition on local functions in arbitrary models under which safety footprints are regained, and showed that if this condition is met by all the primitive commands of the programming language, then safety footprints are regained for every program. The theory of footprints has proven very useful in exploring the situations in which safety footprints could be regained, as one only needs to check that the smallest safe states correspond to the footprint definition 4.3. This automatically gives the required properties such as essentiality and sufficiency, which, without the footprint definition and theorems, would need to be explicitly checked in the different cases.

Finally, we comment on some related work. 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 recently 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 semantics, as the allocation of cells in an execution prevents the framing of those cells. Interestingly, however, the heap model from example 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. Instead, 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*, Springer, vol. 4590, pp. 178-192, 2007.

[2] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Automatic modular assertion checking with separation logic. In *FMCO*, Springer, vol. 4111, pp. 115-137, 2006.

[3] L. Birkedal and H. Yang. Relational parametricity and separation logic. In *FOSSACS*, Springer, vol. 4423, pp. 93-107, 2007.

[4] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, ACM, vol. 40, pp. 259-270, 2005.

[5] R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. In *MFPS*, Elsevier ENTCS, vol. 155, pp. 247-276, 2005.

[6] S. D. Brookes. A semantics for concurrent separation logic. In *Theoretical Computer Science*, Elsevier, vol. 375, pp. 227-270, 2007.

[7] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *POPL*, ACM, vol. 40, pp. 271-282, 2005.

[8] C. Calcagno, P. Gardner, and U. Zarfaty. Local Reasoning about Data Update. In *Gordon Plotkin's festschrift*, Elsevier ENTCS, vol. 172, pp. 133-175, 2007.

[9] C. Calcagno, P. O'Hearn, and H. Yang. Local Action and Abstract Separation Logic. In *LICS*, IEEE Computer Society, pp. 366-378, 2007.

[10] T. Hoare and P. O'Hearn. Separation Logic Semantics of Communicating Processes. In *FICS*, Elsevier ENTCS, vol. 212, pp. 3-25, 2008.

[11] S. Isthiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, ACM, vol. 36, pp. 14-26, 2001.

[12] C. C. Morgan. The specification statement. In *ACM Transactions on Programming Languages and Systems*, ACM, vol. 10, pp. 403-419, 1988.

[13] P. O'Hearn. Resources, concurrency and local reasoning. In *Theoretical Computer Science*, Elsevier, vol. 375, pp. 271-307, 2007.

[14] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, Springer-Verlag, vol. 2142, pp. 1-19, 2001.

[15] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. In *Bulletin of Symbolic Logic*, ASL, vol. 5, pp. 215-244,1999.

[16] M. Parkinson. Local Reasoning for Java. Ph.D. Thesis (University of Cambridge), 2005.

[17] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *LICS*, IEEE Computer Society, pp. 137-146, 2006.

[18] D. Pym, P. O'Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. In *Theoretical Computer Science*, Elsevier, vol. 315, pp. 257-305, 2004.

[19] D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Kluwer Academic Publishers, Applied Logic Series, vol. 26, 2002.

[20] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, IEEE Computer Society, pp. 55-74, 2002.

[21] H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *FOSSACS*, Springer-Verlag, vol. 2303, pp. 402-416, 2002.