Stores as Homomorphisms and their Transformations*

Egidio Astesiano – Gianna Reggio – Elena Zucca

Università di Genova – Dipartimento di Informatica e Scienze dell'Informazione
Viale Benedetto XV,3 16132, Genova, Italy
astes, reggio, zucca @ disi.unige.it

Introduction

In the classical denotational model of imperative languages (see e.g. [7], Chap. 7.3) handling structured types, like arrays, requires an ad-hoc treatment for each data type, including e.g. an ad-hoc allocation and deallocation mechanism. Our aim is to give a homogeneous approach that can be followed whichever is the data structure of the language.

We start from the traditional model for Pascal-like languages, which uses a notion of store as a mapping from left values (containers for values, usually called *locations*), into right values; we combine this idea with the well-known algebraic approach for modelling data types. More precisely, we consider an algebraic structure both for the right and the left values; consequently, the store becomes a homomorphic mapping of the left into the right structure.

Seeing a store as a homomorphism has a number of interesting consequences. First of all, the transformations over a store can be uniformly and rigorously defined on the basis of the principle that they are minimal variations compatible with some basic intended effect (e.g., some elementary substitution). Thus semantic clauses too, which rely on these transformations as auxiliary functions, can be given uniformly; for example, we can give a unique clause for assignment for any data type in Pascal and Ada-like languages.

In Sect. 1 we present the problem and outline the solution, while in Sects. 2 and 3 we give the formal model; in the conclusion we mention some related work.

An extended presentation of the ideas of this paper, including proofs, together with many examples of application, is given in [2].

1 A Motivating Example

As stated in [7], Chap. 5, *imperative languages* can be informally defined as languages that utilize the *store*, which is an abstraction of the computer's memory, and include syntactic constructs (usually called *commands*) whose semantics is, roughly speaking, a store transformation. The most simple example is an assignment like e.g. `x:=y+1` .

* This work has been supported by IS-CORE-Esprit-BRA-W.G. n. 6071 and the project MURST 40% "Metodi e specifiche per la concorrenza".

In the classical denotational model (here and in the sequel we follow in the essence [7] Chap. 7) the store is formalized as a mapping from containers for values, usually called *locations*, into values (like integers). Formally

$Store = [Loc \to Val]_{\text{fin}}$.[2]

This definition is slightly different from the traditional one for distinguishing the case of a location which is unused in the current store ($l \notin Dom(\sigma)$) from the case of a location which is in use, but not yet initialized ($l \in Dom(\sigma)$, $\sigma(l)$ undefined).

The effect of an assignment like above is, roughly speaking, to add to the current store an association from the location corresponding to `x` to the value obtained evaluating `y+1`. The formalization depends on the overall semantics of the language. For example, the typical model based on environment and store formalizes the effect of a command as a function which, for a given environment, returns a store transformation.

$Env = [Id \to Den], \qquad Den = Loc + \ldots$

$\mathcal{C}: Com \to [Env \to [Store \to Store]]$.

The semantics of the above assignment is as follows:

$\mathcal{C}[\![\, \mathtt{x} := \mathtt{y} + \mathtt{1} \,]\!]\rho\sigma = \sigma[\sigma(\rho(\mathtt{y})) +^{\mathbb{Z}} \mathtt{1}/\rho(\mathtt{x})]$.

Introducing two different semantic functions for left and right expressions, i.e. expressions which may appear in the left-hand (resp. right-hand) side of an assignment, the above clause can be obtained as an instance of the following general clause (+) for assignment:

$$(+)\ \mathcal{C}[\![\, lexpr := expr \,]\!]\rho\sigma = \sigma[\mathcal{E}[\![\, expr \,]\!]\rho\sigma/\mathcal{LE}[\![\, lexpr \,]\!]\rho\sigma]$$

where

$\mathcal{LE}: LExpr \to [Env \to [Store \to Loc]], \qquad \mathcal{E}: Expr \to [Env \to [Store \to Val]]$;

of course in the case of an identifier we have

$\mathcal{LE}[\![\, id \,]\!]\rho\sigma = \rho(id), \qquad \mathcal{E}[\![\, id \,]\!]\rho\sigma = \sigma(\rho(id))$.

Let us consider the case of compound data structures ([7] 7.3), for example an array declaration like

```
type a = array [1..10] of int.
```

A variable identifier of this type, say `arr`, has a denotation in the environment which is, accordingly with the intuition, a partial function from indexes into integer locations. However, the store remains a mapping from locations of basic types (like `int`) into basic values; no associations are introduced in the store for compound types.

Hence, an assignment to `arr` cannot be modelled using the general schema (+), but is actually expanded to ten assignments, one for each of the components:

$\mathcal{C}[\![\, \mathtt{arr} := \mathtt{expr} \,]\!]\rho\sigma = \sigma[z_1/\rho(\mathtt{arr})[\mathtt{1}]] \ldots [z_{10}/\rho(\mathtt{arr})[10]]$,
if $(\mathcal{E}[\![\, \mathtt{expr} \,]\!]\rho\sigma)(i) = z_i$, for $i = 1, \ldots, 10$.

Analogously, allocation and deallocation for a variable identifier of an array type cannot be modelled following a general schema; allocation for `arr` is expanded to ten allocations of integer locations (see [7] for the details).

[2] Here and in the following $[A \to B]_{\text{fin}}$ denotes the set of the partial functions from a finite subset of A into B.

What we look for in this paper is a general and more abstract model, which allows to treat assignment, allocation and deallocation in a uniform way for any data structure, thus providing a basis for a systematic approach to proving semantic properties.

We start from the idea of modelling a data type as a (many-sorted partial) algebra. That means that, for example, a language operator like the array selector $\_[\_]$, which combines an expression of type `a` and an expression of type `[1..10]`, giving an expression of type `int`, has two different semantic counterparts: an operation which takes a location of type `a`, an index in $\{1, \ldots, 10\}$ and gives an integer location, and an operation which takes a value of type `a`, an index in $\{1, \ldots, 10\}$ and gives an integer. That is formalized by giving a signature Σ_a in which we distinguish left and right sorts, and a corresponding algebra A, as shown in Fig. 1.

sig Σ_a =
  sorts $L\text{-}int, R\text{-}int, L\text{-}a, R\text{-}a, ind$
  opns
    $\_ + \_,\ \_ \times \_ \colon R\text{-}int\ R\text{-}int \to R\text{-}int$
    $\_[\_]_L \colon L\text{-}a\ ind \to L\text{-}int$
    $\_[\_]_R \colon R\text{-}a\ ind \to R\text{-}int$

$A_{L\text{-}int} = Loc_{int} \cup \{l[i] \mid l \in Loc_a, i \in A_{ind}\}, \qquad A_{R\text{-}int} = \mathbb{Z}$
$A_{L\text{-}a} = Loc_a, \qquad A_{R\text{-}a} = [A_{ind} \to A_{R\text{-}int}]$
$A_{ind} = \{1, \ldots, 10\}$
where Loc_a, Loc_{int} are two infinite denumerable sets
$\_ + \_^A$, $\_ \times \_^A$ are the usual sum and product in $\mathbb{Z}$
$\_[\_]_L^A(l, i) =_\perp l[i]$, for each $l \in A_{L\text{-}a}$, $i \in A_{ind}$
$\_[\_]_R^A$ is the usual function application.

Fig. 1. Left-right algebra for the type `a`

Consider now the store σ. Having different sorts, σ is a sort-indexed family of maps $\{\sigma_a, \sigma_{int}\}$, from left to right values of the corresponding sorts; moreover we have some consistency requirements. First, whenever a location of type `a` is in use, all its subcomponents of type `int` are in use, too, and conversely. Formally:

(*) for each $l \in Loc_a$, $i \in \{1, \ldots, 10\}$, $l[i] \in Dom(\sigma)$ iff $l \in Dom(\sigma)$.

Moreover, it is easy to see that each store σ must satisfy the condition

(**) $\sigma_{int}(l[i]) =_\perp \sigma_a(l)[i]$, for each $i \in \{1, \ldots, 10\}$,

where $=_\perp$ denotes strong equality (either the two sides are defined and equal or both are undefined).

The properties (*) and (**) can be expressed in a general way for any data type, by structuring the domain of the store $Dom(\sigma)$ as an algebra, and seeing σ as a structure preserving mapping, i.e. a homomorphism.

In this way, the property (*) above can be generalized requiring that $Dom(\sigma)$ is a strong subalgebra of A; the property (**) corresponds to requiring that by extending the store by the identity over right values, we get a partial homomorphism from $Dom(\sigma)$ into the restriction of A to only right sorts and operations (refer to Sect. 2 for the detailed definitions).

A major consequence of seeing the store as homomorphism is the possibility of qualifying the store transformations that can occur in a program execution in a way that it is independent of the particular data structure. Considering for example substitution: the basic intended effect is that a new association is added from a used location, say l of type a, into a right value, removing any preceding association with l. As a consequence, in order to keep the homomorphic structure of the store, a new association is added also for each location which is a subcomponent of l, say $l[1]$, $\ldots$, $l[10]$. Now we can define substitution essentially as the minimal variation of the store which has the above intended effect and is compatible with its homomorphic structure, i.e. gives a new store which is still a homomorphism. Analogously for allocation, deallocation and alike (the definitions are in Sect. 3).

Then we get immediately two important applications.

- We can provide uniform semantic clauses, independently of the data type, since we can use the global definition of the store transformations as auxiliary functions. For example the assignment clause takes the general form
  $\mathcal{C}[\![\, lexpr := expr \,]\!]\rho\sigma = \sigma[\mathcal{E}[\![\, expr \,]\!]\rho\sigma // \mathcal{LE}[\![\, lexpr \,]\!]\rho\sigma]$
  where now $\_[\_//\_]$ denotes substitution (in the sense described before and formally given in Sect. 3).
- For every data type we can check whether the explicit definition of the store transformations is correct, in the sense that the resulting transformations, usually given by a series of detailed clauses, are the same as that given by general definition.

2 Stores as Homomorphisms

Before formally introducing stores, we have to define the overall algebraic structure for left and right values, that we call a *left-right algebra*. That is an algebra over a particular kind of signature, that we call *left-right static signature*, which, informally, provides two different kinds of value types: the types of the values which can be stored, called *left-right types* (modelled by two sorts, one for the actual values, called right, and one for the corresponding locations, called left), and the types of the values which cannot be stored, called *right types* (modelled by only the right sort).

Correspondingly, there are three different kinds of operations: pairs of operations returning a left value and a right value in a "corresponding" way (e.g. array selectors); operations returning left values for which there is no right analogous (e.g. an operation which, given a location, returns the next location in the store); operations returning right values for which there is no left analogous (e.g. integer sum, product and so on).

Definition 1. A *(left-right) static signature* is a 5-tuple $ST\Sigma = (T, RT, OP, LOP, ROP)$ where:

- T (left-right types), RT (right types) are two sets of symbols; let
  $LSorts(ST\Sigma) = \{L\text{-}t \mid t \in T\}$,
  $RSorts(ST\Sigma) = RT \cup \{R\text{-}t \mid t \in T\}$,
  $Sorts(ST\Sigma) = LSorts(ST\Sigma) \cup RSorts(ST\Sigma)$
  be the sets of the *left sorts*, *right sorts* and *sorts* of $ST\Sigma$, respectively;
- $OP = \{OP_{w,L\text{-}t}\}_{w \in Sorts(ST\Sigma)^*, t \in T}$;
- $LOP = \{LOP_{w,L\text{-}t}\}_{w \in Sorts(ST\Sigma)^*, t \in T}$;
- $ROP = \{ROP_{w,rs}\}_{w \in RSorts(ST\Sigma)^*, rs \in RSorts(ST\Sigma)}$.

Let in what follows $ST\Sigma = (T, RT, OP, LOP, ROP)$ be a (left-right) static signature; then
$OP_L = \{op_L : s_1 \ldots s_n \to L\text{-}t \mid op \in OP_{s_1 \ldots s_n, L\text{-}t}\}$,
$OP_R = \{op_R : rs_1 \ldots rs_n \to R\text{-}t \mid op \in OP_{s_1 \ldots s_n, L\text{-}t}\}$,
where $rs_i = s_i$ if $s_i \in RSorts(ST\Sigma)$, $R\text{-}t_i$ if $s_i = L\text{-}t_i$.

Note that LOP models left operations with no right analogous, while OP_L models the left version of operations having also the right one; the same difference holds between ROP and OP_R.

It is easy to see that $ST\Sigma$ uniquely determines a usual many-sorted signature, denoted by $WholeSig(ST\Sigma)$, defined as the pair

$$<Sorts(ST\Sigma), LOP \cup OP_L \cup OP_R \cup ROP>.$$

Moreover, we associate with $ST\Sigma$ other two signatures (which are subsignatures of $ST\Sigma$) keeping only the operations which must be preserved by the homomorphic structure of the stores, in the left and right version respectively:
$LSig(ST\Sigma) = <Sorts(ST\Sigma), OP_L>$, the *left part* of $ST\Sigma$,
$RSig(ST\Sigma) = <RSorts(ST\Sigma), OP_R>$, the *right part* of $ST\Sigma$.
Finally, we denote by $\phi_{ST\Sigma}$ the signature morphism from $LSig(ST\Sigma)$ into $RSig(ST\Sigma)$ which maps right sorts into themselves, left sorts and operations into corresponding right sorts and operations.

Definition 2. A *left-right $ST\Sigma$-algebra* is a partial algebra A over $WholeSig(ST\Sigma)$.

If A is a left-right $ST\Sigma$-algebra, then the *left* (resp. *right*) part of A, denoted by A^L (resp. A^R), is the restriction of A to $LSig(ST\Sigma)$ (resp. $RSig(ST\Sigma)$).

An element l belonging to $A_{L\text{-}t}$ is called a *principal left value* iff there exist no $op_L : s_1 \ldots s_n \to L\text{-}t$ in OP, $a_1 \in A_{s_1}, \ldots, a_n \in A_{s_n}$ s.t. $l = {op_L}^A(a_1, \ldots, a_n)$. Intuitively, principal left values are left values which are not subcomponents of other left values. For each left-right type t, let ${Loc^A}_t$ denote the set of the principal left values in $A_{L\text{-}t}$.

We define now *stores*. Roughly speaking a store is a mapping from (currently existing) locations into right values, satisfying some consistency requirements. More precisely: existing locations consist in a finite family of principal locations together with all their subcomponents (compare Sect. 1 (*)); the associations from left into right values respect the operations (compare Sect. 1 (**)). These requirements can be formally expressed as below.

Definition 3. If A is a left-right $ST\Sigma$-algebra, $\Sigma = \mathit{WholeSig}(ST\Sigma)$, then a *store* of A is a Σ-homomorphism $\sigma: D \to A^R|_{\phi_{ST\Sigma}}$ which satisfies the following assumptions. Set ${Loc^D}_t = D_{L\text{-}t} \cap {Loc^A}_t$, for all $t \in T$.

1. ${Loc^D}_t$ is finite, for all $t \in T$;
2. D is the subalgebra of A^L generated by $Loc^D \cup \{A_{rs}\}_{rs \in RSorts(ST\Sigma)}$;
3. $\sigma_{rs} = Id_{A_{rs}}$ (the identity of A_{rs}), for all $rs \in RSorts(ST\Sigma)$.

We denote by A_{store} the set of the stores of A.

Here above $A^R|_{\phi_{ST\Sigma}}$ denotes the reduct of A^R w.r.t. the signature morphism $\phi_{ST\Sigma}$.

Due to the requirements that a store must satisfy, it turns out that it is uniquely determined by fixing which are the currently existing principal locations and their associated right values. Hence it is more convenient to introduce a notion of *store kernel* which consists in a mapping from a finite set of principal locations into right values. In this way a store can be defined as the minimal mapping which extends a store kernel and moreover satisfies the consistency requirements, i.e. conditions 1, 2, 3 above.

Definition 4. If A is a left-right $ST\Sigma$-algebra, then a *(store) kernel of* A is a T-family of partial functions $\kappa = \{\kappa_t\}_{t \in T}$ s.t., for all $t \in T$, $\kappa_t \in [{Loc^A}_t \to A_{R\text{-}t}]_{\text{fin}}$. We denote by A_{kernel} the set of the kernels of A.

Proposition 5. *There exists a bijective correspondence* $\overline{\_}: A_{\text{kernel}} \to A_{\text{store}}$ *which associates with each kernel* κ *the store* $\overline{\kappa}$ *generated by* κ*.*

3 Store Transformations

The purpose of a left-right algebra is to give the algebraic structure of all intermediate configurations in the execution of an imperative program; each store models one configuration. In order to have a complete model of the execution, we must add *dynamic operations*, i.e. operations which model store transformations.

Definition 6. A *left-right signature* is a pair $\Sigma = (ST\Sigma, DOP)$ where:

- $ST\Sigma$ is a left-right static signature;
- DOP is an $S^\star \times S$-family of symbols called *dynamic operation symbols*, where $S = Sorts(ST\Sigma)$; if $dop \in DOP_{s_1 \ldots s_n, s}$, then we write $dop: s_1 \ldots s_n \Rightarrow s$.

Let in what follows $\Sigma = (ST\Sigma, DOP)$ be a left-right signature.

Definition 7. A *left-right structure* over Σ is a pair $LRS = (A, \{dop^{LRS}\}_{dop \in DOP})$ where:

- A is a left-right $ST\Sigma$-algebra;
- for each $dop: s_1 \ldots s_n \Rightarrow s$,
  $dop^{LRS}: A_{\text{store}} \times A_{s_1} \times \ldots \times A_{s_n} \to A_{\text{store}} \times A_s$.

Dynamic operations returning just a store can be obtained by adding a dummy sort whose carrier is a singleton and are denoted by $dop: s_1 \ldots s_n \Rightarrow$.

Note that, due to Prop. 5 above, in order to define a dynamic operation, it is sufficient to define a corresponding operation which acts on kernels. Formally, let $f: A_{\text{kernel}} \times A_{s_1} \times \ldots \times A_{s_n} \to A_{\text{kernel}} \times A_s$ be an operation which acts on kernels; then we define $\overline{f}: A_{\text{store}} \times A_{s_1} \times \ldots \times A_{s_n} \to A_{\text{store}} \times A_s$ by $\overline{f}(\overline{\kappa}) = \overline{f(\kappa)}$.

We define now a family of dynamic operations sufficient for modelling store transformations in Pascal-like languages, by giving the corresponding ones on the kernels.

Let in what follows $LRS = (A, \{dop^{LRS}\}_{dop \in DOP})$ be a left-right structure.

Definition 8. The *empty kernel*, denoted by $\emptyset$, is the kernel with empty domain.

Fact 9. *The store generated by $\emptyset$ is the store with empty domain.*

Definition 10. For each $t \in T$, let new_t denote the predicate over $A_{\text{kernel}} \times Loc^A{}_t$ defined by: $\text{new}_t(\kappa, l)$ holds iff $l \notin Dom(\kappa)$.

Definition 11. For each $t \in T$, the *extension* (or *allocation*) operation of type t is the function

$\_ +_t \_: A_{\text{kernel}} \times Loc^A{}_t \to A_{\text{kernel}}$

defined as follows:

$\kappa +_t l = \kappa'$ if $\text{new}_t(\kappa, l)$ holds, undefined otherwise where:

$Dom(\kappa') = Dom(\kappa) \cup \{l\}$; $\quad Graph(\kappa') = Graph(\kappa)$.

Definition 12. For each $t \in T$, the *restriction* (or *deallocation*) operation (of type t) is the function

$\_\backslash_t \_: A_{\text{kernel}} \times Loc^A{}_t \to A_{\text{kernel}}$

defined as follows:

$\kappa \backslash_t l = \kappa'$, where

$Dom(\kappa') = Dom(\kappa) - \{l\}$; $\quad \kappa'(l') =_\perp \kappa(l')$, for each $l' \in Dom(\kappa')$.

For defining in a general way the substitution operation, we need to assume that, for each non principal location, say l, l can be obtained in a unique way as a subcomponent of a unique principal location, say l' (that implies in particular that each operation in OP has only an argument of left sort); moreover, changing the right value associated with l uniquely determines a corresponding change of the right value associated with l'.

These assumptions allow to uniquely define the store transformation induced by updating whatever location; actually less restrictive assumptions would be sufficient (allowing more arguments of left sorts in operations in OP), but the above version allows a simpler formalization, and is satisfied by all usual imperative languages, as Pascal, Algol, Ada, Common Lisp and so on.

Assumption Upd1. The operations in OP are all of the form

$op: L\text{-}t\ rs_1 \ldots rs_n \to L\text{-}t'$, with $t, t' \in T, rs_i \in RSorts(ST\Sigma)$ for $i = 1, \ldots, n$.

For formally expressing the second assumption, we need some technical definitions.

Definition 13. Let A be a left-right $ST\Sigma$-algebra.

For each operation $op: L\text{-}t\ rs_1 \ldots rs_m \to L\text{-}t' \in OP$, $r_1 \in A_{rs_1}, \ldots, r_m \in A_{rs_m}$, we say that $sel = op[r_1, \ldots, r_m]$ is a *selector* from t into t' and define:

- $sel_L\colon A_{L\text{-}t} \to A_{L\text{-}t'}$, $sel_L(l) = op_L(l, r_1, \ldots, r_m)$;
- $sel_R\colon A_{R\text{-}t} \to A_{R\text{-}t'}$, $sel_R(r) = op_R(r, r_1, \ldots, r_m)$.
- $l \xrightarrow{sel} l'$ iff $l' = sel_L(l)$;
- $l \xrightarrow{sel_1 \cdot \ldots \cdot sel_n} l'$ iff there exist $l_1, \ldots, l_n$ s.t. $l \xrightarrow{sel_1} l_1 \to \ldots \to l_{n-2} \xrightarrow{sel_{n-1}} l_{n-1} \xrightarrow{sel_n} l_n = l'$, $(n \geq 0)$.

Assumption Upd2. For each $t \in T$, $l \in A_{L\text{-}t}$, there exist unique $t' \in T$, $l' \in {Loc^A}_{t'}$, $sel_1 \cdot \ldots \cdot sel_n$ selector list s.t. $l' \xrightarrow{sel_1 \cdot \ldots \cdot sel_n} l$.
Assumption Upd3. For each selector *sel* from t into t', we assume that there exist two functions:

$\text{upd}(sel)\colon A_{R\text{-}t} \times A_{R\text{-}t'} \to A_{R\text{-}t}$ $\quad \text{upd}^{\perp}(sel)\colon A_{R\text{-}t'} \to A_{R\text{-}t}$ such that:

- $sel_R(\text{upd}(sel)(r, r')) = r'$; $\quad sel_R(\text{upd}^{\perp}(sel)(r')) = r'$;
- for each sel' selector from t into t'', $sel' \neq sel$,
  $sel'_R(\text{upd}(sel)(r, r')) =_{\perp} sel'_R(r)$ $\quad sel'_R(\text{upd}^{\perp}(sel)(r'))$ undefined.

The assumption above informally means that for each selector *sel* from t into t' (e.g. _[1] which takes the first element of an array), there exist two corresponding operations:

- $\text{upd}(sel)$ which models the effect of changing the *sel*-component of a value of type t (e.g. _[_/1] which returns the array obtained updating the first element);
- $\text{upd}^{\perp}(sel)$ which constructs a value of type t from a value of type t' (e.g. $\emptyset$[_/1] which returns an array having only the first element).

These two functions can be naturally extended to lists of selectors (we omit the formal definition).

Definition 14. For each $t \in T$, the *substitution operation* (of type t) is the function
$\_[\_//\_]_t\colon A_{\text{kernel}} \times A_{R\text{-}t} \times A_{L\text{-}t} \to A_{\text{kernel}}$
defined as follows:
$\kappa[r//l]_t = \kappa'$ if there exist $l' \in Dom(\kappa)$, *sel-list* list of selectors s.t. $l' \xrightarrow{sel\text{-}list} l$,
undefined otherwise, where
$Dom(\kappa') = Dom(\kappa)$;
$\kappa'(l') = \text{upd}(sel\text{-}list)(\kappa(l'), r)$, if $\kappa(l')$ is defined;
$\kappa'(l') = \text{upd}^{\perp}(sel\text{-}list)(r)$, if $\kappa(l')$ is undefined;
$\kappa'(l'') = \kappa(l'')$, for each $l'' \neq l'$.

Note that if there exists l' and *sel-list*, then they are unique by assumption **Upd2**.

Below we give an explicit definition of the three basic dynamic operations as acting on the stores, in the case that assumptions **Upd1**, **Upd2** and **Upd3** hold.

In the following we write $l \to l'$ ($l \not\to l'$) iff there exists (resp. there does not exist) a list of selectors *sel-list* s.t. $l \xrightarrow{sel\text{-}list} l'$.

Fact 15. *For each $t \in T$, $\sigma \in A_{\text{store}}$ and $l \in {Loc^A}_t$ s.t. $\text{new}_t(ker(\sigma), l)$ holds,*
$\overline{ker(\sigma) +_t l} = \sigma'$, *where:*
$Dom(\sigma') = Dom(\sigma) \cup \{l' \mid l \to l'\}$; $Graph(\sigma') = Graph(\sigma)$.

In other words, the store obtained by an allocation includes the new location with all its subcomponents and is unchanged elsewhere.

Fact 16. *For each* $t \in T$, $\sigma \in A_{\mathrm{store}}$, $l \in Loc^A{}_t$,
$\underline{ker(\sigma)\backslash_t\, l} = \sigma'$, *where*
$Dom(\sigma') = Dom(\sigma) - \{l' \mid l \to l'\}$; $\sigma'(l') =_\perp \sigma(l')$, *for each* $l' \in Dom(\sigma')$.

In other words, the store obtained by a deallocation keeps only the locations which are not subcomponents of the deleted location with their associated right values.

Proposition 17. *For each* $t \in T$, $\sigma \in A_{\mathrm{store}}$, $r \in A_{R\text{-}t}$, $l \in A_{L\text{-}t}$ *s.t.* $l \in Dom(\sigma)$,
$\underline{ker(\sigma)[r//l]_t} = \sigma'$, *where* σ' *is inductively defined by*

$Dom(\sigma') = Dom(\sigma)$;
$\sigma'(l) = r$;
for each $l' \in Dom(\sigma)$ *s.t.* $l \not\to l'$, $l' \not\to l$, $\sigma'(l') =_\perp \sigma(l')$;
for each l' *s.t.* $l \to l'' \xrightarrow{sel} l'$, *if* $\sigma'(l'') = r''$, *then* $\sigma'(l') =_\perp sel_R(r'')$;
for each l' *s.t.* $l' \xrightarrow{sel} l'' \to l$;
if $\sigma'(l'') = r''$, $\sigma(l') = r'$, *then* $\sigma'(l') = \mathrm{upd}(sel)(r', r'')$;
if $\sigma'(l'') = r''$, $\sigma(l')$ *is undefined, then* $\sigma'(l') = \mathrm{upd}(sel)^\perp(r'')$.

In other words, the store obtained by a substitution is the minimal store which contains the new association and leaves unchanged all the unrelated locations (i.e., which are neither subcomponents of the updated location, nor conversely).

Allocation, deallocation and substitution can be considered the basic store transformations in usual imperative languages, in the sense that the final store obtained as the result of a program can be always obtained starting from the initial store (empty) by applying a finite sequence of these operations. That property is formally expressed here below.

Proposition 18. *Assume that LRS satisfies assumptions* **Upd1**, **Upd2** *and* **Upd3**. *Then the operations empty store, extension and substitution are a generating family for* A_{store}, *in the sense that the stores are the family inductively defined by:*

(1) $\emptyset \in A_{\mathrm{store}}$;
(2) if $\sigma \in A_{\mathrm{store}}$, *then* $\sigma + l \in A_{\mathrm{store}}$, *for each principal left value* l *s.t.* $\mathrm{new}(\sigma, l)$ *holds;*
(3) if $\sigma \in A_{\mathrm{store}}$, *then* $\sigma[r//l] \in A_{\mathrm{store}}$, *for each* $l \in A_{L\text{-}t}$, $r \in A_{R\text{-}t}$, *s.t.* $l \in Dom(\sigma)$.

4 Conclusion

We have defined a mathematical structure, the left-right structure, which extends to the imperative case the usual algebraic framework for data types. That means fixing a structure for right values and for locations (left values); moreover the association from locations into values must respect this structure, i.e. the store is a homomorphism. For what concerns dynamics, the underlying data structure determines also which are the possible basic transformations of the store: these are modelled in turn as operations which involve the store and (either left or right) values, called dynamic operations.

The main result of the paper is to provide an abstract uniform setting for the semantics of programming languages, which in a sense rounds up and completes the well-known denotational approach. Our framework can be used also in the context of the wider approach of inductive semantics, advocated by the first of the authors; actually the first application of left-right structures has been shown in that context (see [1]).

There is an interesting issue that we have not treated here and will be exposed in some further paper, namely the relationship to the approaches whose early representative is the "evolving algebra" framework (see [4]). The distinguishing feature in our approach is the concept of store as homomorphism; however it should be possible to pass from one formalism to the other in some canonical way.

An important paper also dealing with L-values and R-values is [6], where however the main aim is the (functorial) treatment of the locality of variables and the issue of structured data types is not tackled.

We mention also a very recent paper on "mutation algebras" (see [5]), which has some similarity in the aims, but a completely different technique, without reference to the concept of a homomorphic store.

Finally, it is possible to describe also object based languages using left/right structures modelling objects in the same way of values of pointer types; nevertheless, this kind of model looks a too low level for objects. A better model which is a generalization of left/right structures is proposed in [3].

Acknowledgment We thanks the referee for various helpful comments and for reminding us the work in [6].

References

1. E. Astesiano. Inductive semantics. In *Formal Descriptions of Programming Concepts*, Berlin, 1991. Springer Verlag.
2. E. Astesiano, G. Reggio, and E. Zucca. Stores as homomorphisms and their transformations – A uniform approach to structured data types in denotational semantics. Technical report, 1993. Submitted.
3. E. Astesiano and E. Zucca. A semantic model for dynamic systems. In U.W. Lipeck and B. Thalheim, editors, *Fourth International Workshop on Foundations of Models and Languages for Data and Objects – Modelling Database Dynamics*, Volkse (Germany), October 1992.
4. Y. Gurevich. Evolving algebra, a tutorial introduction. *Bulletin of the EATCS*, (43):264–284, 1991.
5. G.T. Leavens and K.K. Dhara. A foundation for the model theory of abstract data types with mutation and aliasing. Technical Report 92-35, Department of Computer Science, Iowa State University, Amsterdam, November 1992.
6. J.C. Reynolds. The essence of Algol. In *Intl. Symp. on Algorithmic Languages*, Amsterdam, 1981. North-Holland.
7. D.A. Schmidt. *Denotational Semantics: a methodology for language development.* Wm. C. Brown Publishers, Duboque, Iowa, 1986.