A Simple Semantics for Destructive Updates

Ján Kľuka
Institute of Informatics, Comenius University, Mlynská dolina, 842 48 Bratislava, Slovakia
jan.kluka@fmph.uniba.sk

Abstract. Many algorithms intended to destructively update arrays can be implemented in declarative programming languages by programs operating on lists. These programs are easy to understand and prove, but possibly inefficient. We sketch a semantic method ensuring that a suitably annotated program operating on lists can be correctly transformed into an efficient program that destructively updates arrays.

1 Introduction

Many algorithms on arrays can be presented in declarative languages in terms of list constructions and concatenations. Such declarative programs operating on lists are often simpler to understand and easier to prove than naive translations of array algorithms to declarative programs that use array indexing and updates. On the other hand, programs that use indexing are easy to compile to a lower level code that actually indexes and destructively updates arrays.

The note [Vod02] has suggested how to transform a declarative program that operates on lists into one that uses array indexing and updates. In this paper, we describe how a programmer should control the transformation by annotating the program that operates on lists. Moreover, we outline a method for ensuring correctness of the program resulting from the transformation. The method is based on checking certain semantic properties of an intermediate form of the original program. We use the algorithm of quicksort partitioning as an example.

Various approaches to the problem of destructive updates in declarative languages have been proposed. Static analysis enables the compiler to turn some updates into destructive. It is based either on specialized nonstandard semantics [Hud87, Blo94], or on purely syntactic properties of programs [Sha99]. In other approaches, the programmer marks some updates as destructive, and the program is then checked by typing [Wad90], or

Proceedings of the Eighth ESSLLI Student Session
Balder ten Cate (editor)

abstract interpretation [Ode91]. Our approach lets the programmer annotate the program in a way that does not interfere with syntax, and provides semantic conditions in standard semantics necessary for the correctness of desired destructive updates. For most programs, these conditions can be proved automatically by a proof assistant. We define programs and express their properties in the same language—Peano arithmetic—in the style of the programming language and proof assistant *CL* [KV99, CL].

2 Two Declarative Implementations of Quicksort Partitioning

2.1 Preliminaries: Pairs, Lists, and Clausal Definitions. All functions and predicates in this paper are definable in Peano arithmetic as functions and predicates over natural numbers. We use the pairing function written in infix form as x, y to construct both lists and pairs. This function is best thought of as the `(cons x y)` operation of Lisp. The pairing function is defined so that no pair equals the number 0, which thus serves as the *empty list* (`()` in Lisp).

Throughout the paper, we use the *singleton list* function $[x]$ which yields the list $x, 0$ (an equivalent of Lisp's `(list x)`), and the *concatenation* function $x \oplus y$ (analogous to `(append x y)` in Lisp) yielding a list of all elements of x followed by all elements of y. We also use the predicate $x \varepsilon y$ which holds if x is an element of the list y. The *modified subtraction* $x \dot{-} y$, mentioned in the next paragraph, equals $x - y$ if $x \geq y$, and 0 otherwise.

Functions and predicates are defined either *explicitly* (in which case the definition is preceded by the symbol $\vdash_{\mathrm{PAx}}$) or *clausally*. Clausal definitions resemble the clausal definitions of Prolog. Unlike in Prolog, we define both predicates and functions clausally in our language, which is why we strictly demand the clauses to be mutually exclusive. We allow pattern matching in both clause heads (within formal arguments) and clause bodies (pattern matching may occur only on the right-hand side of an identity).

More information on clausal definitions and the pairing function can be found in [KV99].

2.2 A Naive Declarative Partitioning Operating on an Array. In Peano arithmetic, we can represent arrays as lists. If a is an array, let $(a)_i$ denote its element at index i, and $a[i \leftrightarrow j]$ a new array which is just like a, but with elements at indices i and j swapped. If our compiler can compile these operations into actual indexing and destructive array updates, they take constant time. Then we can define an efficient quicksort partitioning function $Part_n$ as shown in Fig. 1.1.

Let $a[i\,..\,j)$ denote the *segment* of the array a between indices i (including) and j (excluding). The function takes three arguments—an ar-

$$
\begin{array}{lll}
Part_n(a,l,u) = a, l \dot- 1 & \leftarrow l = u & \\
Part_n(a,l,u) = Part_n(a,l,u \dot- 1) & \leftarrow l < u \wedge (a)_{u \dot- 1} \geq (a)_{l \dot- 1} & \\
Part_n(a,l,u) = a[l \dot- 1 \leftrightarrow l], l & \leftarrow l < u \wedge (a)_{u \dot- 1} < (a)_{l \dot- 1} \wedge l+1 = u & \\
Part_n(a,l,u) = Part_n(a_2, l+1, u) & \leftarrow l < u \wedge (a)_{u \dot- 1} < (a)_{l \dot- 1} \wedge l+1 < u \wedge & \\
& \quad a[l \dot- 1 \leftrightarrow l] = a_1 \wedge a_1[l \dot- 1 \leftrightarrow u \dot- 1] = a_2 &
\end{array}
$$

Figure 1.1: A naive declarative implementation of quicksort partitioning.

ray a and indices l, u—, partitions the segment $a[l \dot- 1 \,..\, u)$ according to the pivot $(a)_{l \dot- 1}$, and returns a pair consisting of the modified array a_1 and the new position l_1 of the pivot. Elements of the segment $a[l \dot- 1 \,..\, u)$ that are less than the pivot are placed in $a_1[l \dot- 1 \,..\, l_1)$, those greater than or equal to the pivot are in $a_1[l_1 + 1 \,..\, u)$. This is expressed by the following property:

$$
\begin{array}{l}
Part_n(a,l,u) = a_1, l_1 \wedge 0 < l \wedge l \leq u \wedge u \leq L(a) \rightarrow \\
\quad a_1 \sim a \wedge l \dot- 1 \leq l_1 \wedge l_1 < u \wedge \\
\quad a_1[0 \,..\, l \dot- 1) = a[0 \,..\, l \dot- 1) \wedge a_1[u \,..\, L(a)) = a[u \,..\, L(a)) \wedge \\
\quad \forall x(x \,\varepsilon\, a_1[l \dot- 1 \,..\, l_1) \rightarrow x < (a)_{l \dot- 1}) \wedge (a_1)_{l_1} = (a)_{l \dot- 1} \wedge \\
\quad \forall x(x \,\varepsilon\, a_1[l_1 + 1 \,..\, u) \rightarrow x \geq (a)_{l \dot- 1})
\end{array}
$$

where $L(a)$ denotes the length of the array a, and $a_1 \sim a$ is a predicate expressing that a_1 is a permutation of a. The property and consequently its proof rely heavily on indexing and arithmetic which makes them rather complicated. Besides, the property must contain explicit statement that elements at indices below $l \dot- 1$ and above u remain unchanged.

2.3 An Inefficient Implementation Operating on Lists. If correctness is our primary concern, we can define a partitioning function operating on two lists r and x as shown in Fig. 1.2 (for the time being, ignore the arrows). The list r represents the array segment containing only the pivot, the list x represents the segment to be partitioned. The function $Part$ yields a triple consisting of a list of elements of x less than the pivot, a singleton (atomic) list containing only the pivot, and a list of elements of x greater than or equal to the pivot.

This function is quite inefficient—it performs pattern matching at the end of its second argument and several list concatenations. On the other hand, its main property

$$
\begin{array}{l}
Part([p], x) = l, r, u \rightarrow \\
\quad [p] \oplus x \sim l \oplus r \oplus u \wedge \forall v(v \,\varepsilon\, l \rightarrow v < p) \wedge r = [p] \wedge \forall v(v \,\varepsilon\, u \rightarrow v \geq p)
\end{array}
$$

$$Part([p], 0) = 0, [p], 0$$

$$Part([p], x \oplus [v]) = l, r, u_1 \qquad \leftarrow v \geq p \wedge Part([p], x) = l, r, u \wedge u \oplus [v] = u_1$$

$$Part([p], 0 \oplus [v]) = [v], [p], 0 \qquad \leftarrow v < p$$

$$Part([p], [w] \oplus x \oplus [v]) = l_1, r, u \leftarrow v < p \wedge Part([p], x \oplus [w]) = l, r, u \wedge [v] \oplus l = l_1$$

Figure 1.2: A partitioning function operating on lists.

is expressed in terms of operations on lists, and is considerably easier to prove than the corresponding property of $Part_n$.

In the following paragraphs, we shall sketch how, given some additional information provided by the programmer, we can mechanically transform this function into one that operates destructively on an array similarly to $Part_n$. We also outline how the correctness of the transformed function is ensured.

2.4 Annotation and Additional Requirements. *Annotations* represent a programmer's intention to place certain values at specific indices in an array. We represent such an intention by an arrow leading from a pattern matching of a singleton list to a construction of another singleton list. The value used in the destination singleton is intended to be placed at the index of the source singleton. The annotation of function *Part* displayed in Fig. 1.2 closely follows the pattern of indexing from the function $Part_n$.

Apart from annotating the reuse of indices, the programmer should state additional requirements for the transformation of the *Part* function:

1. Its two arguments are supposed to be adjacent segments of an input array, and the members of the result triple are supposed to be adjacent segments of an output array.

2. The arguments reside at the same indices of the input array as the indices occupied by the members of the result triple in the output array.

3. The input array is the same as the output array, which is thus updated *destructively*.

$$
\begin{array}{ll}
Lln(0) & \\
Lln((v,\ell),w) \leftarrow Lln(w) & [v]^\ell = (v,\ell),0 \\
Labels(0) = 0 & Unl(0) = 0 \\
Labels((v,\ell),w) = \ell, Labels(w) & Unl((v,\ell),w) = v, Unl(w)
\end{array}
$$

$$
\begin{gathered}
Lln(x \oplus y) \leftrightarrow Lln(x) \land Lln(y) \\
Unl([v]^\ell) = [v] \\
Lln(x \oplus y) \rightarrow Unl(x \oplus y) = Unl(x) \oplus Unl(y)
\end{gathered}
$$

Figure 1.3: Labeled lists.

$$
\begin{array}{ll}
Part_\ell([p]^{\ell_1}, 0) = 0, [p]^{\ell_1}, 0 & \\
Part_\ell([p]^{\ell_1}, x \oplus [v]^{\ell_2}) = l, r, u_1 & \leftarrow v \geq p \land Part_\ell([p]^{\ell_1}, x) = l, r, u \land \\
& \quad u \oplus [v]^{\ell_2} = u_1 \\
Part_\ell([p]^{\ell_1}, 0 \oplus [v]^{\ell_2}) = [v]^{\ell_1}, [p]^{\ell_2}, 0 & \leftarrow v < p \\
Part_\ell([p]^{\ell_1}, [w]^{\ell_3} \oplus x \oplus [v]^{\ell_2}) = l_1, r, u & \leftarrow v < p \land Part_\ell([p]^{\ell_3}, x \oplus [w]^{\ell_2}) = l, r, u \land \\
& \quad [v]^{\ell_1} \oplus l = l_1
\end{array}
$$

Figure 1.4: The labeled version of the function *Part*.

3 From Lists to Destructively Updated Arrays

3.1 Labeled Lists. The intention to put a value at a certain index can be expressed semantically in terms of labeled lists. A *labeled list* is a list of pairs v, ℓ where v is a value, and ℓ is a label representing an array index where the value resides.

The predicate *Lln* (cf. Fig. 1.3) is defined to hold for such lists. We shall construct labeled lists by concatenation and the labeled-singleton function $[v]^\ell$. We also define two projection functions: *Labels* yielding a list of labels of a labeled list, and *Unl* yielding an unlabeled version of such a list, i.e., its list of values.

3.2 Transforming a Function Operating on Lists to a Function Operating on Labeled Lists. Given the annotated function, we now mechanically construct a function operating on labeled lists such that it manipulates the values of list elements just like the function *Part*, and manipulates their labels as prescribed by the annotation. We call the new function $Part_\ell$, or the *labeled version* of the function *Part*. Its clausal definition is shown in Fig. 1.4.

$$\vdash_{\mathrm{PAx}} x \boxdot y \leftrightarrow \forall p \forall v_1 \forall \ell_1 \forall v_2 \forall \ell_2 \forall q (x = p \oplus [v_1]^{\ell_1} \wedge [v_2]^{\ell_2} \oplus q = y \rightarrow \ell_1 + 1 = \ell_2)$$

$$\vdash_{\mathrm{PAx}} \mathit{Vect}(x) \leftrightarrow \forall y \forall z (y \oplus z = x \rightarrow y \boxdot z)$$

$$\vdash_{\mathrm{PAx}} x \approx y \leftrightarrow \mathit{Lln}(x) \wedge \mathit{Lln}(y) \wedge \mathit{Labels}(x) = \mathit{Labels}(y)$$

$$\begin{array}{cc}
x \approx x & x \boxdot 0 \\
x \approx y \rightarrow y \approx x & 0 \boxdot x \\
x \approx y \wedge y \approx z \rightarrow x \approx z & x \approx y \rightarrow x \boxdot z \leftrightarrow y \boxdot z \\
x \approx y \leftrightarrow z \oplus x \approx z \oplus y & x \approx y \rightarrow z \boxdot x \leftrightarrow z \boxdot y \\
x \approx y \leftrightarrow x \oplus z \approx y \oplus z & \mathit{Vect}(x \oplus y) \leftrightarrow \mathit{Vect}(x) \wedge \mathit{Vect}(y) \wedge x \boxdot y \\
[v_1]^{\ell_1} \approx [v_2]^{\ell_2} \leftrightarrow \ell_1 = \ell_2 & x \approx y \rightarrow \mathit{Vect}(x) \leftrightarrow \mathit{Vect}(y)
\end{array}$$

Figure 1.5: The Adjacency ($\boxdot$), Vector (*Vect*), and the Same-Sequences-of-Labels ($\approx$) predicates.

3.3 The Basic Correctness Property of the Transformed Function. Before continuing any further, we have to be sure that the results of the function $Part_\ell$ are equivalent to those of the function *Part* except for labeling, which is stated formally as follows:

$$\begin{aligned}&\mathit{Lln}(r \oplus x) \rightarrow \\ &\quad \mathit{Part}_\ell(r, x) = l, r_1, u \leftrightarrow \mathit{Part}(\mathit{Unl}(r), \mathit{Unl}(x)) = \mathit{Unl}(l), \mathit{Unl}(r_1), \mathit{Unl}(u)\,.\end{aligned}$$

This can be proved automatically on object level by a proof assistant from obvious properties of unlabeling mentioned in Fig. 1.3, or we can prove on the meta-level that any function and its labeled version behave the same except for labeling.

3.4 Labeled Lists and Array Segments. Our next step is to show that the function $Part_\ell$ meets requirements 1 and 2 from Par. 2.4. We need some definitions (see Fig. 1.5) to do that.

We say that two labeled lists x and y are *adjacent* ($x \boxdot y$) iff the first label in y immediately succeeds the last label in x. We call a labeled list representing an array segment a *vector*, and define the predicate *Vect* to hold for a list x iff every two lists y and z such that $x = y \oplus z$ are adjacent. Finally, $x \approx y$ means that the labeled lists x and y have *identical lists of labels*.

The first two requirements from Par. 2.4 can now be expressed by the two formulas (proof obligations) below:

$$\mathit{Lln}(r \oplus x) \wedge \mathit{Vect}(r \oplus x) \wedge \mathit{Part}_\ell(r, x) = l, r_1, u \rightarrow \mathit{Vect}(l \oplus r_1 \oplus u) \quad (1)$$

$$\mathit{Lln}(r \oplus x) \wedge \mathit{Vect}(r \oplus x) \wedge \mathit{Part}_\ell(r, x) = l, r_1, u \rightarrow r \oplus x \approx l \oplus r_1 \oplus u \quad (2)$$

Properties of the predicates *Vect* and $\approx$ together with (2) imply (1). It is therefore sufficient to prove the second obligation. A sufficiently sophisticated proof assistant can construct the proof automatically when given the properties from Fig. 1.5.

Nevertheless, there is another problem to be addressed here. A concatenation of array segments can be performed if and only if they are adjacent. It can then be carried out in constant time, since it involves only calculation of limits of the resulting segment. Note that the result of a concatenation of labeled lists is a vector (i.e., a representation of an array segment) iff the concatenated lists are adjacent vectors (Fig. 1.5). We refer to a concatenation whose result is a vector as *admissible*.

To outline how to check that all concatenations are admissible, let $\varphi[\vec{a}]$ be a formula representing the requirements imposed on the arguments $\vec{a}$ of a labeled version f_ℓ of a function, and assume a concatenation occurs in the following context in a clause of that version ($\psi[x,y]$ and ω are possibly empty conjunctions of literals):

$$f_\ell(\vec{a}) = \vec{r} \leftarrow \psi[x,y] \wedge x \oplus y = z \wedge \omega\,.$$

Then to ensure that this concatenation is admissible, it is sufficient to prove

$$\varphi[\vec{a}] \wedge \psi[x,y] \rightarrow \mathit{Vect}(x \oplus y)\,.$$

Formulas of this type are mechanically derivable from the function definition, and if the annotation is correct, they are automatically provable using the properties of vectors, the requirement (2), and the properties of the predicate $\approx$.

In the particular case of the function Part_ℓ, we have $\varphi[r,x] \equiv \mathit{Lln}(r \oplus x) \wedge \mathit{Vect}(r \oplus x)$. From the last clause of the function, the following proof obligations are generated:

$$\varphi\big[[p]^{\ell_1}, [w]^{\ell_3} \oplus x \oplus [v]^{\ell_2}\big] \rightarrow \mathit{Vect}(x \oplus [w]^{\ell_2})$$

$$\varphi\big[[p]^{\ell_1}, [w]^{\ell_3} \oplus x \oplus [v]^{\ell_2}\big] \wedge v < p \wedge \mathit{Part}_\ell([p]^{\ell_3}, x \oplus [w]^{\ell_2}) = l, r, u \rightarrow \mathit{Vect}([v]^{\ell_1} \oplus l)\,.$$

3.5 Labeled Lists and Destructive Updates. Recall that in Par. 2.4, an arrow leading from a pattern matching of a singleton list to a construction of another singleton list, say $[v]$, declares a programmer's intention to put the value v at the same index where the pattern matching occurred. In a labeled version of a function, the index is represented by a label.

Suppose we have a clause whose body contains a pattern matching $x = [v]^\ell \oplus y$ followed by a construction $[w]^\ell = z$. The construction is intended to destructively update an array at the index ℓ. If x is implemented as a pair of indices limiting a segment of that array, then x cannot be used further in

$$\vdash_{\mathrm{PAx}} x \mathrel{\#} y \leftrightarrow \forall \ell \forall v_1 \forall v_2 (v_1, \ell \not\varepsilon\, x \lor v_2, \ell \not\varepsilon\, y)$$
$$\vdash_{\mathrm{PAx}} x \between y \leftrightarrow \forall \ell \forall v_1 \forall v_2 (v_1, \ell \mathrel{\varepsilon} x \land v_2, \ell \mathrel{\varepsilon} y \to v_1 = v_2)$$

$$\begin{array}{c}
x \mathrel{\#} 0 \\
\mathit{Lln}(x) \to x \mathrel{\#} x \leftrightarrow x = 0 \\
x \mathrel{\#} y \to y \mathrel{\#} x \\
x \mathrel{\#} [v]^{\ell} \leftrightarrow \forall v (v, \ell \not\varepsilon\, x) \\
x \mathrel{\#} y \oplus z \leftrightarrow x \mathrel{\#} y \land x \mathrel{\#} z \\
x \approx y \to x \mathrel{\#} z \leftrightarrow y \mathrel{\#} z \\
\mathit{Lln}(x \oplus y) \land \mathit{Vect}(x \oplus y) \to x \mathrel{\#} y
\end{array}
\qquad
\begin{array}{c}
x \between 0 \\
x \between y \to y \between x \\
x \between y \oplus z \leftrightarrow x \between y \land x \between z \\
x \mathrel{\#} y \to x \between y \\
\mathit{Lln}(x \oplus y) \land \mathit{Vect}(x \oplus y) \to x \oplus y \between x \\
\mathit{Lln}(x \oplus y) \land \mathit{Vect}(x \oplus y) \to x \oplus y \between y
\end{array}$$

Figure 1.6: Definitions and selected properties of the Disjoint-Labels predicate $\#$, and the Correctly-Aliased predicate $\between$.

the clause since the list of values of x may be no longer actually present in the corresponding array segment.

A simple solution to this problem is to allow each list variable to be used only once after it is given a value. Although the function *Part* (and consequently Part_ℓ) complies with this constraint, it is too restrictive—in the above example, using the variable x between the pattern matching and the construction would do no harm.

To deal with this problem in a more permissive semantic manner, we define the predicates shown in Fig. 1.6. Formula $x \mathrel{\#} y$ expresses the fact that the sets of labels of two labeled lists x and y are disjoint. We shall call two lists x and y *correctly aliased* and write $x \between y$ iff for every label ℓ shared by these lists, the value that occurs at label ℓ in the list x is the same as the value occurring at that label in the list y. If these lists are vectors, and they actually share a label, they represent two overlapping array segments.

Let $\varphi[\vec{a}]$ stand for the requirements imposed on the arguments of a labeled version f_ℓ of a function (as in Par. 3.4). Furthermore, assume that labeled-list variables x and y occur in a clause of f_ℓ in the following context (ξ, ψ, and $\omega[x]$ are possibly empty conjunctions of literals, τ and ρ are terms):

$$f_\ell(\vec{a}) = \vec{r} \leftarrow \xi \land \tau = x \land \psi \land \rho = y \land \omega[x] \,.$$

The identity $\tau = x$ gives a value to the variable x, $\rho = y$ gives a value to y. After that, x is used in the rest $\omega[x]$ of the clause.

We claim, that if a property of the form

$$\varphi[\vec{a}] \land \xi \land \tau = x \land \psi \land \rho = y \to y \between x$$

holds, then the contents of the array segment denoted by the variable x will not be affected by a destructive update which implements the identity $\rho = y$ if ρ contains a singleton construction. This holds even if variables x and y denote overlapping array segments. The clause context above shows one of three cases. The other two cases apply when the variable x is one of arguments $\vec{a}$, or a member of the result tuple $\vec{r}$; properties that should hold are similar.

As an example, consider the last clause of $Part_\ell$ with atomized list operations:

$$\begin{aligned} Part_\ell([p]^{\ell_1}, [w]^{\ell_3} \oplus x \oplus [v]^{\ell_2}) = l_1, r_1, u \leftarrow {} & v < p \wedge [p]^{\ell_3} = r \wedge \\ & x \oplus [w]^{\ell_2} = x_1 \wedge \\ & Part_\ell(r, x_1) = l, r_1, u \wedge \\ & [v]^{\ell_1} \oplus l = l_1 \end{aligned}$$

The corresponding proof obligations are:

$$\varphi\big[[p]^{\ell_1}, [w]^{\ell_3} \oplus x \oplus [v]^{\ell_2}\big] \to [p]^{\ell_3} \between x \wedge x \oplus [w]^{\ell_2} \between [p]^{\ell_3}$$

$$\begin{aligned} \varphi\big[[p]^{\ell_1}, [w]^{\ell_3} \oplus x \oplus [v]^{\ell_2}\big] \wedge Part_\ell([p]^{\ell_3}, x \oplus [w]^{\ell_2}) = l, r_1, u \to & \\ [v]^{\ell_1} \oplus l \between r_1 \wedge [v]^{\ell_1} \oplus l \between u\,. & \end{aligned}$$

Again, a suitable proof assistant can automatically construct a proof of these obligations, which is based on the properties from Fig. 1.6 and the property 3.4(2).

3.6 Mechanical Transformation of the Function Operating on Labeled Lists into One Operating on Arrays. Finally, we sketch the transformation of the function $Part_\ell$ to a function operating destructively on an array. Let $a[i := v]$ denote a destructive update of the array a placing the value v at index i. Every vector from the function $Part_\ell$ is transformed into a pair of indices that limit the corresponding array segment. The singleton construction $[v]^\ell$ is transformed into the destructive update $a[\ell := v]$. Concatenations are replaced by calculations of limits of array segments.

After elimination of unnecessary variables, a function $Part_a(a, r, x, t) = a_1, l, r_1, u, t_1$ is obtained, where a is the array to be partitioned, the pair of indices r, x limits the singleton segment containing the pivot, indices x, t limit the segment to be partitioned, l, r_1 limit the segment containing elements less then the pivot, r_1, u limit of the new pivot segment, and indices u, t_1 are limits of the segment of elements greater than the pivot.

Bibliography

[Blo94] A. Bloss. Path analysis and the optimization of nonstrict functional languages. *ACM Transactions on Programming Languages and Systems*, 16(3):328–369, 1994.

[CL] The *CL* homepage. http://www.ii.fmph.uniba.sk/cl .

[Hud87] P. Hudak. A semantic model for reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis Horwood Ltd., 1987.

[KV99] J. Komara and P. Voda. Theorems of Péter and Parsons in computer programming. In G. Gottlob, E. Grandjean, and K. Seyr, editors, *Proceedings of CSL'98*, number 1584 in LNCS, pages 204–223. Springer Verlag, 1999.

[Ode91] M. Odersky. How to make destructive updates less destructive. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 25–26. ACM Press, 1991.

[Sha99] N. Shankar. *Efficiently Executing PVS.* Technical report, SRI International, Menlo Park, CA, 1999. Available through WWW from http://www.csl.sri.com/users/shankar/shankar-drafts.html.

[Vod02] P. Voda. On modification of declarative arrays, 2002. Available through WWW from http://www.diku.dk/undervisning/2002e/213/segm.ps.

[Wad90] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.