Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 879–886

ISBN 978-83-60810-22-4

A Type and Effect System for Implementing Functional Arrays with Destructive Updates

Georgios Korfiatis
Email: gkorf@softlab.ntua.gr

Michalis Papakyriakou
Email: mpapakyr@softlab.ntua.gr

Nikolaos Papaspyrou
Email: nickie@softlab.ntua.gr

School of Electrical and Computer Engineering
National Technical University of Athens
Polytechnioupoli, 15780 Zografou, Athens, Greece

***Abstract*—It can be argued that some of the benefits of purely functional languages are counteracted by the lack of efficient and natural-to-use data structures for these languages. Imperative programming is based on manipulating data structures destructively, e.g., updating arrays in-place; however, doing so in a purely functional language violates the language's very nature. In this paper, we present a type and effect system for an eager purely functional language that tracks array usage, i.e., read and write operations, and enables the efficient implementation of purely functional arrays with destructive update.**

I. Introduction

ARRAYS are ubiquitous data structures in imperative programming, offering constant-time storing and retrieval of data. On the contrary, their use in the purely functional programming paradigm is not equally natural. A key property in purely functional programs is referential transparency, which guarantees that an expression has always the same value in any context. Referential transparency facilitates reasoning about program properties and also enables many compiler optimizations. Yet this entails that an operator intended to update the contents of a given array should actually yield a fresh (updated) copy of the array, leaving the original untouched, and thus adding a significant time and space complexity overhead to the program.

Naïve implementations of such an update operator would require an additional $O(n)$ complexity for each update, both in time and space, where n is the size of the array. Better implementations could be based on building a set of differences between the original and the updated array, in the same spirit more or less as Haskell's `DiffArray`. In principle, such implementations could be tailored to specific use patterns for arrays, (e.g., single-threaded updates); however, as Haskell's bug reports reveal, the overall performance is very poor.

Pippenger has shown that every algorithm using strict impure data structures that runs in $O(n)$ can be translated to an algorithm using pure data structures that runs in $O(n \log n)$ time, simulating random access memory with appropriate algebraic data structures such as balanced binary trees [12]. He has also shown that there are algorithms for which this is the best one can do; in other words, there are algorithms for which an impure language performs asymptotically better than a pure language. This result has caused a significant stir in the functional programming community. It has been discussed whether it is valid for lazy pure languages [2], or for algorithms without "on line" requirements.

On the other hand, two practical questions have been raised on this subject:

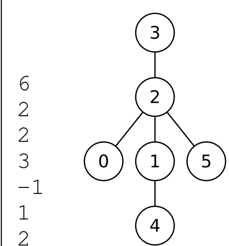
- What good does this study of asymptotic behavior make? Even if the book-keeping cost of using purely functional data structures can be amortized throughout the program and does not increase the overall program complexity, it still induces a constant factor slowdown that may not be negligible.
- How easy is it to work with (and analyze the performance of) purely functional data structures? In other words, how easily can one reuse a long-standing imperative algorithm in a purely functional language and will it be the same algorithm after all?

Let us consider a motivating example. Suppose we are given as input a tree structure which we need to subsequently process, e.g., perform a depth-first traversal. Suppose also that the input is given in the following form: the first line contains the number of nodes n (numbered from 0 to $n-1$) and the following n lines contain the nodes' parents. The root of the tree is given a parent of -1. The following program snippet in C++ can be used to read such a tree. For each node `i`, the list `c[i]` contains the node's children. Variable `r` contains the tree's root.

```
int n, r;
scanf("%d\n", &n);
list<int> c[n];
for (int i=0; i<n; i++) {
  int p;
  scanf("%d\n", &p);
  if (p >= 0)
    c[p].push_back(i);
  else
    r = i;
}
```

Input

```
6
2
2
3
-1
1
2
```



Now consider an equivalent program snippet in a purely functional programming language. We use here OCaml syntax, but we assume purely functional arrays with the following signature. Function `upd` returns the updated array; semantically, the result of `upd a i x` can be thought of as a new array

978-83-60810-22-4/$25.00 © 2011 IEEE

whose contents equal those of `a`, with the exception of value `x` occupying position `i`.

```
type 'a array
val newArray : int -> 'a -> 'a array
val get : 'a array -> int -> 'a
val upd : 'a array -> int -> 'a -> 'a array
```

We skip the reading of data and assume that both n and the list of the nodes' parents l have already been read. Our goal is to build a data structure of type `tree`, which we can then process in a purely functional fashion.

```
type tree = T of int * tree list
```

Function `build` does precisely this. It walks down the list `l`, using array `c` to collect for each node a list of its children, exactly as the imperative program does. Argument `r` propagates the tree's root node. Finally, function `mkTree` builds the tree structure using the information that has been collected in `c`.

```
let build n l =
  let rec walk i c r = function
    | [] ->
        (c, r)
    | p :: ps when p >= 0 ->
        let c = upd c p (i :: get c p) in
        walk (i+1) c r ps
    | _ :: ps ->
        walk (i+1) c i ps in
  let c = newArray n [] in
  let (c, r) = walk 0 c 0 l in
  let rec mkTree u =
    T (u, map mkTree (get c u)) in
  mkTree r
```

With a naïve implementation of functional arrays, copying the array every time it is updated, function `build` will require $O(n^2)$ time, while its imperative counterpart takes just $O(n)$. Even smarter implementations, e.g., using a balanced binary tree for `c`, will take $O(n \log n)$. There doesn't seem to be a natural way to bring the complexity down to $O(n)$ without updating the array in-place. Notice that, in the program snippet above, the array `c` is updated and used only in a single-threaded way; after the updating takes place, the original array is never used again. In this program, all array updates can be done destructively without altering its semantics, thus obtaining an $O(n)$ time complexity. This, however, need not be the case, and this is the real difficulty in optimizing purely functional arrays.

It is desirable to combine the best of the two worlds, imperative and purely functional, in a language that implements array operations efficiently while complying with the referential transparency principle of purely functional programming. In this paper, we present a type and effect system for a purely functional language that enables such an optimization in a safe manner. In particular, the language includes an update operator that can be implemented destructively, but is semantically equivalent to the pure `upd` function that we have just seen. In the sections that follow, we present our approach in an informal way through examples (Section II), then the formalization of the type and effect system (Section III). Section IV discusses related work. We finish with some concluding remarks and directions for future work.

$$
\begin{array}{rcl}
e & ::= & w \mid \mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2 \mid w_1\, w_2 \mid \mathbf{if}\, w \,\mathbf{then}\, e_1 \,\mathbf{else}\, e_2 \\
& \mid & \mathbf{newArray}\, w_1\, w_2 \mid \mathbf{get}\, w_1\, w_2 \mid \mathbf{upd}\, w_1\, w_2\, w_3 \mid \ldots \\
w & ::= & x \mid v \\
v & ::= & f \mid \ell \mid \mathbf{true} \mid \mathbf{false} \mid i \\
f & ::= & \lambda x : \tau . e \mid \mathbf{fix}\, x : \tau . f \\
\tau_g & ::= & \mathbf{Bool} \mid \mathbf{Int} \\
\tau & ::= & \tau_g \mid \mathbf{Array}\, \tau_g \mid (x : \tau_1) \xrightarrow{\gamma \& \delta} \tau_2
\end{array}
$$

Fig. 1. Syntax.

II. An Informal Account of our Approach

In Fig. 1 we define a simple eager functional language, whose main deviation from conventional languages is that evaluation order is explicit: all operators take evaluated arguments (w), which are values (v) or variables that contain values (x). More complex expressions must be explicitly "linearized", forming essentially a sequence of let-bindings with primitive operations. Making the evaluation order explicit is not an important feature of our language, but it facilitates the presentation of our approach. Programs in a traditional eager functional language, like the last example of Section I, can be straightforwardly transformed to this linear form. For example, $x\, y\, z$ is translated to $\mathbf{let}\, y' = x\, y \,\mathbf{in}\, y'\, z$. Notice also that our language is explicitly typed, but a more-or-less standard type inference can be used to fill in most of the type annotations.

The language includes integer values (i), boolean values and standard conditional expressions, function values (f) which can be recursive, and location constants (ℓ) that do not appear in the original program. Although we do not show them, we assume the existence of operators for manipulating integer and boolean values. The language also includes operators for array manipulation:

- $\mathbf{newArray}\, i\, v$ creates an array of size i whose cells are all initialized with value v and returns it;
- $\mathbf{get}\, a\, i$ returns the ith element of array a; and
- $\mathbf{upd}\, a\, i\, v$ returns an array equal to a except that position i is now occupied by value v.

A type (τ) is either a ground type (τ_g), which can be $\mathbf{Bool}$ or $\mathbf{Int}$, an array type whose element is necessarily of ground type, or a function type. The non-standard annotations in function types will be explained shortly. We assume that arrays are 0-indexed and we do not care to use the type system to avoid array out-of-bounds errors.

The main idea is to maintain an effect for each expression, which records the uses of variables that represent arrays within this expression. Let's start by assuming that an effect consists only of a *qualifier assignment* (ξ), which assigns a qualifier q to each of the array variables used in an expression. (In a short while we will extend our notion of effect.) A variable is

assigned qualifier $\mathbf{W}$ (written) if it has been used in an **upd** operation, or qualifier $\mathbf{R}$ (read) if it has been used in a **get** operation.

For example, assuming that a is of array type, consider the following expression:

$$\begin{array}{lr} \mathbf{let}\, v = \mathbf{get}\, a\, 0\ \mathbf{in} & \{a\!:\!\mathbf{R}\} \\ \quad \mathbf{upd}\, a\, 1\, v & \{a\!:\!\mathbf{W}\} \end{array}$$

Each subexpression produces the qualifier assignment shown to its right. The expression gets its overall effect by combining the effects of its subexpressions in the order of execution. In this example it yields the assignment $\{a\!:\!\mathbf{R}, a\!:\!\mathbf{W}\}$.

It is crucial that an array be not used, either for writing or for reading, after it has been written. Therefore, computing the overall effect of the following expression should fail.

$$\begin{array}{lr} \mathbf{let}\, a' = \mathbf{upd}\, a\, 1\, 0\ \mathbf{in} & \{a\!:\!\mathbf{W}\} \\ \quad \mathbf{get}\, a\, 0 & \{a\!:\!\mathbf{R}\} \end{array}$$

However, due to the possibility of aliasing, it is not guaranteed that two variables in the effect of an expression refer to distinct arrays. It is thus necessary to keep track of the order in which the effects appear when combining two (seemingly unconnected) effects. In particular, we maintain a *constraint set* (κ) which records every pair of variables $x < y$ such that $x\!:\!\mathbf{W}$ has appeared before $y\!:\!q$, for any qualifier q. In order to keep track of aliased variables, we additionally annotate each expression with a *propagation set* (δ) which records variables of array type that may be the result of evaluating this expression.

To summarize all this, we relate each expression with a tuple $\gamma \,\&\, \delta$, where the *effect* γ is itself a tuple $\langle \xi, \kappa \rangle$, consisting of an assignment of qualifiers to variables and a set of constraints. Apart from the standard typing environment, we also keep an aliasing environment A mapping variables to all their possible aliases; this environment is updated in every **let** binding, using the propagation set δ of the bound expression.

Consider the following example, which is equivalent to the previous one and should therefore also be rejected:

$$\begin{array}{lr} \mathbf{let}\, b = a\ \mathbf{in} & \langle \emptyset, \emptyset \rangle \,\&\, \{a\} \\ \mathbf{let}\, a' = \mathbf{upd}\, a\, 1\, 0\ \mathbf{in} & \langle \{a\!:\!\mathbf{W}\}, \emptyset \rangle \,\&\, \emptyset \\ \quad \mathbf{get}\, b\, 0 & \langle \{b\!:\!\mathbf{R}\}, \emptyset \rangle \,\&\, \emptyset \end{array}$$

The first **let** binding leads to nothing more than the aliasing $A(b) = \{a\}$. The subsequent operators **upd** and **get** result in the qualifier assignments for a and b, respectively. What is important, however, is the combination of all these effects in the result of this expression. The overall effect is the qualifier assignment $\{a : \mathbf{W}, b : \mathbf{R}\}$, along with the constraint set $\{a < b\}$, which is produced because $a\!:\!\mathbf{W}$ is executed before $b : \mathbf{R}$. When getting out of its scope, b is replaced by its propagation set $\{a\}$ both in the qualifier assignment and in the constraint set. This leads to the constraint set $\{a < a\}$, which is unsatisfiable, and the program is rejected.

Lambda abstractions have no effects themselves. Nevertheless, they need to remember the possible effects of their body. This is done by annotating the function type with the effect and the propagation set. When applying a function, the effect on the arrow should be checked for compatibility with respect to the effects collected so far. Thus typing the application $f\, 4$ in the following example will fail, because a has been updated before being read by the function.

$$\begin{array}{lr} \mathbf{let}\, f = \lambda x : \mathbf{Int}.\, \mathbf{get}\, a\, x\ \mathbf{in} & f : \mathbf{Int} \xrightarrow{\langle \{a:\mathbf{R}\}, \emptyset \rangle \,\&\, \emptyset} \mathbf{Int} \\ \mathbf{let}\, a' = \mathbf{upd}\, a\, 0\, 0\ \mathbf{in} & \langle \{a : \mathbf{W}\}, \emptyset \rangle \,\&\, \emptyset \\ \quad f\, 4 & \langle \{a : \mathbf{R}\}, \emptyset \rangle \,\&\, \emptyset \end{array}$$

Annotations on arrow types can also contain references to the formal parameters of the function. To this end, we name the type of each formal parameter, as in the following example. (Of course, only annotations for array types are of any use; we thus omit naming parameters of a different type in the following examples.) When applying a function, the formal parameters mentioned in its effect are substituted with the variable (and its aliases) that correspond to the actual parameter. Typechecking the following example will fail, because application $f\, r$ updates r before this is read by the last **get** operation.

$$\begin{array}{lr} \mathbf{let}\, f = \lambda a : \mathbf{Array\, Int}.\, \mathbf{upd}\, a\, 0\, 0\ \mathbf{in} & \\ \qquad f : (a : \mathbf{Array\, Int}) \xrightarrow{\langle \{a:\mathbf{W}\}, \emptyset \rangle \,\&\, \emptyset} \mathbf{Array\, Int} & \\ \mathbf{let}\, r' = f\, r\ \mathbf{in} & \langle \{r : \mathbf{W}\}, \emptyset \rangle \,\&\, \emptyset \\ \quad \mathbf{get}\, r\, 0 & \langle \{r : \mathbf{R}\}, \emptyset \rangle \,\&\, \emptyset \end{array}$$

The following example demonstrates how aliasing can be detected in our system. Function f, which takes two array parameters x and y, is applied to the same array r.

$$\begin{array}{lr} \mathbf{let}\, r = \mathbf{newArray}\, 5\, 0\ \mathbf{in} & \\ \mathbf{let}\, f = & \\ \quad \lambda x : \mathbf{Array\, Int}. & \\ \quad \lambda y : \mathbf{Array\, Int}. & \\ \qquad \mathbf{let}\, a = \mathbf{upd}\, x\, 3\, 4\ \mathbf{in} & \langle \{x\!:\!\mathbf{W}\}, \emptyset \rangle \,\&\, \emptyset \\ \qquad \mathbf{let}\, b = \mathbf{get}\, y\, 3\ \mathbf{in} & \langle \{y\!:\!\mathbf{R}\}, \emptyset \rangle \,\&\, \emptyset \\ \qquad\quad \mathbf{upd}\, a\, 2\, b & \langle \{a\!:\!\mathbf{W}\}, \emptyset \rangle \,\&\, \emptyset \\ \mathbf{in} & \\ \mathbf{let}\, f' = f\, r\ \mathbf{in} & \\ \quad f'\, r & \end{array}$$

Function f has the following type:

$$\begin{array}{l} (x : \mathbf{Array\, Int}) \xrightarrow{\emptyset \,\&\, \emptyset} \\ (y : \mathbf{Array\, Int}) \xrightarrow{\langle \{x:\mathbf{W}, y:\mathbf{R}\}, \{x<y\} \rangle \,\&\, \emptyset} \mathbf{Array\, Int} \end{array}$$

Notice that the first arrow carries no effect, since it only yields a closure. Application $f\, r$ has type:

$$(y : \mathbf{Array\, Int}) \xrightarrow{\langle \{r:\mathbf{W}, y:\mathbf{R}\}, \{r<y\} \rangle \,\&\, \emptyset} \mathbf{Array\, Int}$$

and the last application $f'\, r$ produces the effect

$$\langle \{r\!:\!\mathbf{W}, r\!:\!\mathbf{R}\}, \{r < r\} \rangle$$

which contains an unsatisfiable constraint.

As a bigger example, we adapt the core of the motivating example from Section I. Function *walk* collects in array c the list of children of each node, taking as input the list of nodes' parents (l). We assume a ground type **List Int** of lists of integers but, as our language does not support tuples, we skip

the calculation of the tree's root, which can easily be done by a separate function. We also assume the usual list operators: **nil** and **cons**, **isnil**, **head**, and **tail**.

$$
\begin{array}{l}
\mathbf{let}\ walk\ = \\
\quad \mathbf{fix}\, f : \mathbf{Int} \xrightarrow{\emptyset\,\&\,\emptyset} (c : \mathbf{Array}\,(\mathbf{List\,Int})) \xrightarrow{\emptyset\,\&\,\emptyset} \\
\qquad\quad \mathbf{List\,Int} \xrightarrow{\langle\{c{:}\mathbf{W}, c{:}\mathbf{R}\},\emptyset\rangle\,\&\,\emptyset} \mathbf{Array}\,(\mathbf{List\,Int}). \\
\quad \lambda i : \mathbf{Int}. \\
\quad \lambda c : \mathbf{Array}\,(\mathbf{List\,Int}). \\
\quad \lambda l : \mathbf{List\,Int}. \\
\qquad \mathbf{let}\, b = \mathbf{isnil}\, l\ \mathbf{in} \\
\qquad\ \ \mathbf{if}\, b\, \mathbf{then} \\
\qquad\qquad c \\
\qquad\ \ \mathbf{else} \\
\qquad\qquad \mathbf{let}\, p = \mathbf{head}\, l\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\, ps = \mathbf{tail}\, l\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\, c' = \\
\qquad\qquad\quad \mathbf{let}\, b = (p >= 0)\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{if}\, b\, \mathbf{then} \\
\qquad\qquad\qquad\quad \mathbf{let}\, v = \mathbf{get}\, c\, p\ \mathbf{in} \\
\qquad\qquad\qquad\quad \mathbf{let}\, v' = \mathbf{cons}\, i\, v\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{upd}\, c\, p\, v' \\
\qquad\qquad\qquad \mathbf{else} \\
\qquad\qquad\qquad\quad c\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\, i' = i + 1\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\, f_1 = f\, i'\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\, f_2 = f_1\, c'\ \mathbf{in} \\
\qquad\qquad\quad f_2\, ps
\end{array}
$$

III. Formalization

In this section we formally present the operational semantics of our language and the type and effect system. The syntax of the language has already been presented in Fig. 1.

A. Operational semantics

In Fig. 2, we define a standard eager operational semantics of expressions with respect to a memory μ, mapping locations ℓ to memory blocks. A memory block $\{\,\overline{v_i}^{\,i<n}\,\}$ has size n and contains the values $v_0, \ldots, v_{n-1}$. The linearized nature of the language ensures that there is only one propagation rule in the semantics (for **let**), with every other rule corresponding to a particular sort of redex. Note that $\mathbf{upd}\,\ell\,j\,v$ updates the memory block in-place and returns the original location.

$$\boxed{\mu; e \longrightarrow \mu'; e'}$$

$$\frac{\mu; e_1 \longrightarrow \mu'; e_1'}{\mu; \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \longrightarrow \mu'; \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2}$$

$$\frac{}{\mu; \mathbf{let}\ x = v\ \mathbf{in}\ e \longrightarrow \mu; e[v/x]}$$

$$\frac{}{\mu; (\lambda x : \tau.e)\ v \longrightarrow \mu; e[v/x]}$$

$$\frac{}{\mu; \mathbf{if\ true\ then}\ e_1\ \mathbf{else}\ e_2 \longrightarrow \mu; e_1}$$

$$\frac{}{\mu; \mathbf{if\ false\ then}\ e_1\ \mathbf{else}\ e_2 \longrightarrow \mu; e_2}$$

$$\frac{}{\mu; (\mathbf{fix}\ x : \tau.f)\ v \longrightarrow \mu; f[(\mathbf{fix}\ x : \tau.f)/x]\ v}$$

$$\frac{\begin{array}{l}\ell\ \mathbf{fresh\ in}\ \mu \\ \forall j.(0 <= j < i \Rightarrow v_j = v)\end{array}}{\mu; \mathbf{newArray}\ i\ v \longrightarrow \mu, \ell \mapsto \{\,\overline{v_j}^{\,j<i}\,\}; \ell}$$

$$\frac{0 <= j < n}{\mu, \ell \mapsto \{\,\overline{v_i}^{\,i<n}\,\}; \mathbf{get}\,\ell\,j \longrightarrow \mu, \ell \mapsto \{\,\overline{v_i}^{\,i<n}\,\}; v_j}$$

$$\frac{\begin{array}{l}0 <= j < n \\ \forall j.(0 <= i < n \wedge i \neq j \Rightarrow v_i' = v_i) \\ v_j' = v\end{array}}{\mu, \ell \mapsto \{\,\overline{v_i}^{\,i<n}\,\}; \mathbf{upd}\,\ell\,j\ v \longrightarrow \mu, \ell \mapsto \{\,\overline{v_i'}^{\,i<n}\,\}; \ell}$$

Fig. 2. Operational semantics.

$$
\begin{array}{lcl}
q & ::= & \mathbf{W} \mid \mathbf{R} \\
\xi & ::= & \emptyset \mid \xi, x : q \\
\kappa & ::= & \emptyset \mid \kappa, x < y \\
\gamma & ::= & \langle \xi, \kappa \rangle \\
\delta & ::= & \emptyset \mid \delta, x \\
A & ::= & \emptyset \mid A, x \mapsto \delta \\
\Gamma & ::= & \emptyset \mid \Gamma, x : \tau
\end{array}
$$

Fig. 3. Qualifiers, effects, and environments.

B. Type system

Effects γ are defined in Fig. 3 as a pair $\langle \xi, \kappa \rangle$, where ξ is an assignment of qualifiers q to variables and κ a set of constraints. A propagation set δ is simply a set of variables, an aliasing environment A maps variables to sets of variables, and a typing environment Γ maps variables to types τ.

We have opted to presented the type system in a form of input and output effects. This allows us to determine a type error at the exact program point that is to blame for the constraint violation. The typing judgment for an expression e

$$\Gamma; A; \gamma_0 \vdash e : \tau \,\&\, \gamma \,\&\, \delta$$

yields a type τ, effect γ, and propagation set δ, given type environment Γ, aliasing environment A, and input effect γ_0. The output effect γ results from a combination of the input effect and the actual effect of the expression in question. The typing judgment for evaluated arguments (w) is similar but lacks the output effect; it will be presented later.

Consider first the rule for **get**:

$$\frac{\begin{array}{l}\Gamma; A; \gamma_0 \vdash w_1 : \mathbf{Array}\,\tau_g \,\&\, \delta_1 \\ \Gamma; A; \gamma_0 \vdash w_2 : \mathbf{Int} \,\&\, \emptyset \\ \gamma = \gamma_0 \ltimes \delta_1 : \mathbf{R}\end{array}}{\Gamma; A; \gamma_0 \vdash \mathbf{get}\ w_1\ w_2 : \tau_g \,\&\, \gamma \,\&\, \emptyset}$$

Arguments w_1 and w_2 are first checked and w_1 corresponds to a propagation set δ_1. In fact, since locations do not appear in the original program that we typecheck, w_1 can only be a

variable, say x, thus δ_1 must be the singleton $\{x\}$, as it will become clear in the rule for variables. The actual effect of the **get** expression is $\langle\{x:\mathbf{R}\},\emptyset\rangle$, since variable x is used for reading. This is expressed concisely, generalized for any δ, using the following notation:

$$\delta : q \equiv \langle\{x : q \mid x \in \delta\}, \emptyset\rangle$$

Input and actual effects are combined according to the following function:

$$\gamma_1 \ltimes \gamma_2 \equiv \langle\xi_1,\kappa_1\rangle \ltimes \langle\xi_2,\kappa_2\rangle \equiv \langle\xi_1 \cup \xi_2, (\kappa_1 \cup \kappa_2) \cup \{x < y \mid (x:\mathbf{W}) \in \xi_1 \wedge (y:q) \in \xi_2\}\rangle$$

which takes the union of the qualifier assignments (ξ_1 and ξ_2) and the union of constraints (κ_1 and κ_2) plus a new set of constraints which results from the combination of every *written* variable in γ_1 with every variable in γ_2.

The rule for **upd** is similar; here the actual effect has a $\mathbf{W}$ assignment instead of $\mathbf{R}$.

$$\frac{\begin{array}{l}\Gamma; A; \gamma_0 \vdash w_1 : \mathbf{Array}\,\tau_g \,\&\, \delta_1 \\ \Gamma; A; \gamma_0 \vdash w_2 : \mathbf{Int} \,\&\, \emptyset \\ \Gamma; A; \gamma_0 \vdash w_3 : \tau_g \,\&\, \emptyset \\ \gamma = \gamma_0 \ltimes \delta_1 : \mathbf{W}\end{array}}{\Gamma; A; \gamma_0 \vdash \mathbf{upd}\, w_1\, w_2\, w_3 : \mathbf{Array}\,\tau_g \,\&\, \gamma \,\&\, \emptyset}$$

In both rules, we have omitted array index out-of-bounds checks. This is orthogonal to our approach and we would not like to diverge from the issue of destructive implementation of arrays to discuss it in this paper.

Creating a new array produces no effect; therefore the rule for **newArray** simply propagates the input effect after checking its arguments.

$$\frac{\begin{array}{l}\Gamma; A; \gamma_0 \vdash w_1 : \mathbf{Int} \,\&\, \emptyset \\ \Gamma; A; \gamma_0 \vdash w_2 : \tau_g \,\&\, \emptyset\end{array}}{\Gamma; A; \gamma_0 \vdash \mathbf{newArray}\, w_1\, w_2 : \mathbf{Array}\,\tau_g \,\&\, \gamma_0 \,\&\, \emptyset}$$

In order to typecheck a **let** expression, we first check the bound expression e_1 given input effect γ_0.

$$\frac{\begin{array}{l}\Gamma; A; \gamma_0 \vdash e_1 : \tau_1 \,\&\, \gamma_1 \,\&\, \delta_1 \\ \Gamma, x : \tau_1; A \oplus x \mapsto \delta_1; \gamma_1 \vdash e_2 : \tau_2 \,\&\, \gamma_2 \,\&\, \delta_2 \\ \tau' = \tau_2[\delta_1/x] \\ \gamma' = \gamma_2[\delta_1/x] \\ \delta' = \delta_2[\delta_1/x]\end{array}}{\Gamma; A; \gamma_0 \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \tau' \,\&\, \gamma' \,\&\, \delta'}$$

This produces type τ_1, effect γ_1 and propagation set δ_1. The body e_2 is checked on input effect γ_1, since this represents the combination of γ_0 with the actual effect of e_1, that is, all the effects that occurred prior to the execution of e_2. Apart from adding the binding variable in the type environment, we also add the aliasing information for e_1 in the map A. Binding variable x maps to all variables contained in the propagation set δ_1 and to all their aliases, transitively, as defined below:

$$A \oplus x \mapsto \delta \equiv A, x \mapsto \delta \cup \bigcup\{A(y) \mid y \in \delta\}$$

The resulting type, effect, and propagation set of the **let** expression are those computed for its body. However, these may contain occurrences of x, which must be replaced by its aliases δ_1. The required substitutions are defined below.

$$\begin{array}{rcl}\delta'[\delta/x] & = & \begin{cases}\delta' - \{x\} \cup \delta & \text{if } x \in \delta \\ \delta' & \text{otherwise}\end{cases} \\ \gamma[\delta/x] & = & \langle\xi[\delta/x], \kappa[\delta/x]\rangle \\ \xi[\delta/x] & = & \{y:q \mid (y:q) \in \xi \wedge y \neq x\} \\ & & \cup \bigcup\{\delta : q \mid (x : q) \in \xi\} \\ \kappa[\delta/x] & = & \{y < z \mid (y < z) \in \kappa \wedge y \neq x \wedge z \neq x\} \\ & & \cup\ \{w < z \mid (y < z) \in \kappa \wedge y = x \wedge w \in \delta\} \\ & & \cup\ \{y < w \mid (y < z) \in \kappa \wedge z = x \wedge w \in \delta\}\end{array}$$

Substituting δ for x in γ amounts to removing any assignment $x:q$ and any constraint containing x, and adding the respective assignments and constraints for all variables in δ. Substitution of δ in a type ($\tau[\delta/x]$) walks through the arrows recursively and applies the substitution for γ and δ.

The rule for application is the most elaborate one.

$$\frac{\begin{array}{l}\Gamma; A; \gamma_0 \vdash w_1 : (x : \tau) \xrightarrow{\gamma \& \delta} \tau' \,\&\, \emptyset \\ \Gamma; A; \gamma_0 \vdash w_2 : \tau \,\&\, \delta_2 \\ A \vdash_{wf} \tau'[\delta_2/x] \\ A \vdash_{wf} \gamma[\delta_2/x] \\ \gamma' = \gamma_0 \ltimes \gamma[\delta_2/x] \\ A \vdash_{wf} \gamma' \\ A; \gamma' \vdash_{compat} \delta[\delta_2/x]\end{array}}{\Gamma; A; \gamma_0 \vdash w_1\, w_2 : \tau'[\delta_2/x] \,\&\, \gamma' \,\&\, \delta[\delta_2/x]}$$

We first typecheck w_1 and w_2. The effect of the application is the effect γ that the function carries on its type. As explained in the related example in Section II, any occurrence of the formal parameter x need to be replaced by the aliases of the actual parameter, δ_2. This substitution may render some constraints inconsistent; thus the resulting effect needs to be checked for well-formedness. Judgment

$$A \vdash_{wf} \gamma$$

checks whether γ is well-formed, i.e., it contains no constraint of the form $x < y$ where x and y are aliases (or the same variable). Similarly, the effect γ' that results from combining the input effect with the computed one need to be checked, since it contains fresh constraints. Likewise, the output propagation set is the result of substituting δ_2 for the formal parameter x in the propagation set of the function, δ. Given the output effect γ', we have to make sure that the resulting propagation set does not include any variable that has already been used for writing. For this purpose we employ the following judgment:

$$A; \gamma \vdash_{compat} \delta$$

which checks whether δ is compatible with γ given A, i.e., it does not contain any variable such that either itself or one of its aliases is assigned qualifier $\mathbf{W}$ in γ.

Finally, the resulting type τ' can itself be an arrow type. Since it can contain occurrences of the formal parameter x, these have to be replaced by d_2, too. This substitution may break some constraints; we have thus to check the well-formedness of the type. Judgment

$$A \vdash_{wf} \tau$$

checks the well-formedness of a type with respect to aliasing environment A by checking the well-formedness of all possible γ's contained in the type as well as the compatibility of all δ's with their respective γ.

The rule for **if** typechecks the two alternatives e_1 and e_2 given the same input effect γ_0.

$$\frac{\begin{array}{l}\Gamma; A; \gamma_0 \vdash w : \mathbf{Bool} \,\&\, \emptyset \\ \Gamma; A; \gamma_0 \vdash e_1 : \tau \,\&\, \gamma_1 \,\&\, \delta_1 \\ \Gamma; A; \gamma_0 \vdash e_2 : \tau \,\&\, \gamma_2 \,\&\, \delta_2 \\ \gamma = \gamma_1 \cup \gamma_2 \\ \delta = (\delta_1 - \mathbf{W}_A(\gamma_2)) \cup (\delta_2 - \mathbf{W}_A(\gamma_1))\end{array}}{\Gamma; A; \gamma_0 \vdash \mathbf{if}\; w \;\mathbf{then}\; e_1 \;\mathbf{else}\; e_2 : \tau \,\&\, \gamma \,\&\, \delta}$$

The output effect γ is the union of the two alternative effects, since we need to conservatively assume that either of the two effects has actually happened. Similarly, the output propagation set δ results from the union of the alternative propagation sets. However, extra care is needed here. We need to make sure that δ be compatible with γ. We thus subtract all written variables (and their aliases) of γ_1 (notation $\mathbf{W}_A(\gamma_1)$) from δ_2, and those of γ_2 from δ_1 respectively. This ensures that if a variable is assigned qualifier $\mathbf{W}$ in either effect, then it cannot be an alias for the **if** expression.

Finally, an evaluated argument w is considered an expression, too. The following rule simply uses the respective judgment and propagates the input effect.

$$\frac{\Gamma; A; \gamma_0 \vdash w : \tau \,\&\, \delta}{\Gamma; A; \gamma_0 \vdash w : \tau \,\&\, \gamma_0 \,\&\, \delta}$$

Evaluated arguments (values and variables) are typechecked with the following judgment.

$$\Gamma; A; \gamma_0 \vdash w : \tau \,\&\, \delta$$

This differs from the judgment for expressions in that there is no output effect. On the other hand, rules expect an input effect γ_0, but only for reasons of checking compatibility, as it will be explained shortly.

The first rule concerns variables of an array type.

$$\frac{\begin{array}{c}(x : \mathbf{Array}\,\tau_g) \in \Gamma \\ A; \gamma_0 \vdash_{compat} \{x\}\end{array}}{\Gamma; A; \gamma_0 \vdash x : \mathbf{Array}\,\tau_g \,\&\, \{x\}}$$

Using the appropriate compatibility judgment, we check that such a variable has not been previously used for writing. The main duty of this rule is to propagate variable x as an alias.

Likewise, considering a variable of type other than an array, possibly of arrow type:

$$\frac{\begin{array}{c}(x : \tau) \in \Gamma \\ \mathbf{isNotArray}\,\tau \\ A; \gamma_0 \vdash_{compat} \tau\end{array}}{\Gamma; A; \gamma_0 \vdash x : \tau \,\&\, \emptyset}$$

or a lambda abstraction:

$$\frac{\begin{array}{c}\Gamma, x : \tau; A; \emptyset \vdash e : \tau' \,\&\, \gamma \,\&\, \delta \\ A; \gamma_0 \vdash_{compat} (x : \tau) \xrightarrow{\gamma \& \delta} \tau'\end{array}}{\Gamma; A; \gamma_0 \vdash \lambda x : \tau . e : (x : \tau) \xrightarrow{\gamma \& \delta} \tau' \,\&\, \emptyset}$$

we have to check that the γ and δ on the arrow types are compatible with the input effect, in order to rule out closures containing variables that have been previously used for writing.

The rule for **fix** simply checks that the body of **fix**, f, has the same arrow type as its annotated binder x.

$$\frac{\begin{array}{c}\tau = (y : \tau_1) \xrightarrow{\gamma \& \delta} \tau_2 \\ \Gamma, x : \tau; A; \gamma_0 \vdash f : \tau \,\&\, \emptyset\end{array}}{\Gamma; A; \gamma_0 \vdash \mathbf{fix}\, x : \tau . f : \tau \,\&\, \emptyset}$$

Finally, the typing for boolean and numerical constants is standard. Fig. 4 presents the compatibility and well-formedness judgments mentioned in this section.

IV. Related Work

Several type-based solutions to the destructive update problem for functional languages have been proposed. In some of them, as in our work, the language uses "functional" arrays and the goal is to identify updates that can be done destructively, to enable compiler optimizations. The lazy purely functional language Clean [13] employs uniqueness typing in order to add side effects without sacrificing referential transparency [1]; a simplified version of uniqueness typing has also been presented [3]. Clean's type system works by assigning types containing uniqueness information to expressions. Typechecking takes into account annotations assigned to variables by a sharing analysis, which indicates whether a variable has been used only once within its scope or more than once. Related to uniqueness types is also the work of Harrington [8], and Hage *et al.* [7], [10]. The use of uniqueness types instead of monadic arrays in languages like Haskell is also advocated by Diviánszky [5].

Type-based approaches for functional arrays that are more directly related to the classic theory of linearity have also been proposed. Guzmán and Hudak [6] present a type and effect system similar to ours for a non-strict language, which is based on calculating "liabilities", i.e., mutability, shareness, and linearity attributes for each variable. Their system also cannot handle "unnamed" structures (such as the ones created by ML's `ref` constructor) and for this reason they apply several syntactic restrictions (which we avoid by using a linearized language). They assume some strictness analysis and disallow higher-order arguments to strict function application. Contrary to our work, arrays of functional values are allowed. Also, marginally related to our work is Tov and Pucella's programmer-friendly capability-based type system with affine types [17].

In another family of strongly typed functional languages, arrays are used in the "imperative" style. Then, in languages derived from ML, some of the language's purity is sacrificed for improved efficiency and the researchers' goal is to minimize the cost of impurity [11], [9], [16]. An interesting solution is the use of monadic computations in Haskell [18], which save purity and referential transparency by defining a "language inside the language" for performing impure operations.

$$\boxed{A;\gamma \vdash_{compat} \delta}$$

$$\frac{}{A;\gamma \vdash_{compat} \emptyset} \qquad \frac{x \notin \mathbf{W}_A(\gamma) \quad A;\gamma \vdash_{compat} \delta}{A;\gamma \vdash_{compat} \delta, x}$$

$$\boxed{A;\gamma \vdash_{compat} \tau}$$

$$\frac{}{A;\gamma \vdash_{compat} \tau_g}$$

$$\frac{}{A;\gamma \vdash_{compat} \mathbf{Array}\,\tau_g}$$

$$\frac{A;\gamma \vdash_{compat} \tau \quad A;\gamma \vdash_{compat} \tau' \quad A;\gamma \vdash_{compat} \delta' \quad A;\gamma \vdash_{compat} \gamma'}{A;\gamma \vdash_{compat} (x:\tau) \xrightarrow{\gamma'\&\delta'} \tau'}$$

$$\boxed{A;\gamma \vdash_{compat} \gamma'}$$

$$\frac{A;\gamma \vdash_{compat} \{x \mid \exists q : (x\!:\!q) \in \xi\}}{A;\gamma \vdash_{compat} \langle \xi, \kappa \rangle}$$

$$\begin{array}{lcl} \mathbf{aliases}_A\, x & \equiv & \{z \mid y \in (A(x), x) \wedge (A \vdash \mathbf{areAlias}\, y\, z)\} \\ \mathbf{W}_A(\langle \xi, \kappa \rangle) & \equiv & \bigcup\{\mathbf{aliases}_A\, x \mid (x\!:\!\mathbf{W}) \in \xi\} \end{array}$$

$$\boxed{A \vdash_{wf} \tau}$$

$$\frac{}{A \vdash_{wf} \tau_g}$$

$$\frac{}{A \vdash_{wf} \mathbf{Array}\,\tau_g}$$

$$\frac{A \vdash_{wf} \tau \quad A \vdash_{wf} \tau' \quad A \vdash_{wf} \gamma \quad A;\gamma \vdash_{compat} \delta}{A \vdash_{wf} ((x:\tau) \xrightarrow{\gamma\&\delta} \tau')}$$

$$\boxed{A \vdash_{wf} \gamma}$$

$$\frac{A \vdash_{wf} \kappa}{A \vdash_{wf} \langle \xi, \kappa \rangle}$$

$$\boxed{A \vdash_{wf} \kappa}$$

$$\frac{\forall x.\forall y.(x<y) \in \kappa \Rightarrow \neg(A \vdash \mathbf{areAlias}\, x\, y)}{A \vdash_{wf} \kappa}$$

$$\boxed{A \vdash \mathbf{areAlias}\, x\, y}$$

$$\frac{(A(x_1), x_1) \cap (A(x_2), x_2) \neq \emptyset}{A \vdash \mathbf{areAlias}\, x_1\, x_2}$$

Fig. 4. Compatibility and well-formedness judgments.

On the other hand, several proposed solutions to the same problem are based not on a type system but on static analysis. Sastry *et al.* [14] present a static analysis approach for first-order linearized strict languages with flat arrays, based on abstract interpretation. Liveness analysis determines which updates can be done destructively, using reference counts for abstract locations. Based on this work, Wand and Clinger [19] present a compiler optimization for destructive array updates, focusing on a first-order strict functional language with flat arrays; their approach is based on interprocedural aliasing and liveness analysis, using set constraints. Dimoulas and Wand extend this to an untyped higher-order language [4]. A similar approach is followed by Shankar [15].

V. Concluding Remarks

We have presented a type and effect system for a purely functional linearized language that implements array updates destructively. We expect this type system to play an important role in the optimizer of a purely functional language with arrays, deciding whether an array update can be performed destructively. In case it cannot, the update will have to be implemented in a slower way. We have developed a prototypical compiler and type checker for this language.[1] As future work, we plan to extend this prototype implementation to a fuller functional language, such as ML and to eliminate the existing restrictions (e.g., the lack of support for aggregate data structures, such as tuples, and for arrays of non-ground types). Moving to a full programming language will also require a type inference algorithm that is aware of our effects, as well as effect polymorphism. A different line of research is to investigate how this technique works with lazy functional languages, such as Haskell. Also, we are working on a formal proof of type safety.

[1] Available from http://www.softlab.ntua.gr/~gkorf/src/puredest.tar.gz.

References

[1] E. Barendsen and S. Smetsers, "Uniqueness typing for functional languages with graph rewriting semantics," *Mathematical Structures in Computer Science*, vol. 6, pp. 579–612, 1996.

[2] R. Bird, G. Jones, and O. De Moor, "More haste, less speed: lazy versus eager evaluation," *Journal of Functional Programming*, vol. 7, pp. 541–547, Sep. 1997.

[3] E. de Vries, R. Plasmeijer, and D. M. Abrahamson, "Uniqueness typing simplified," in *Implementation and Application of Functional Languages*, O. Chitil, Z. Horváth, and V. Zsók, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 201–218.

[4] C. Dimoulas and M. Wand, "The higher-order aggregate update problem," in *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2009, pp. 44–58.

[5] P. Diviánszky, "Non-monadic models of mutable references," *Central European Functional Programming School*, pp. 146–182, 2010.

[6] J. Guzmán and P. Hudak, "Single-threaded polymorphic lambda calculus," in *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, 1990, pp. 333–343.
[7] J. Hage, S. Holdermans, and A. Middelkoop, "A generic usage analysis with subeffect qualifiers," in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, 2007, pp. 235–246.
[8] D. Harrington, "Uniqueness logic," *Theoretical Computer Science*, vol. 354, no. 1, pp. 24–41, 2006.
[9] B. Lippmeier, "Type inference and optimisation for an impure world," Ph.D. dissertation, Australian National University, 2009.
[10] A. Middelkoop, "Improved uniqueness typing for Haskell," Master's Thesis, INF/SCR-06-08, Universiteit Utrecht, 2006.
[11] M. Odersky, "How to make destructive updates less destructive," in *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1991, pp. 25–36.
[12] N. Pippenger, "Pure versus impure Lisp," *ACM Transactions on Programming Languages and Systems*, vol. 19, pp. 223–238, Mar. 1997.
[13] R. Plasmeijer and M. van Eekelen, "Clean language report, version 2.1," Department of Software Technology, University of Nijmegen, 2002.
[14] A. Sastry, W. Clinger, and Z. Ariola, "Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 1993, pp. 266–275.
[15] N. Shankar, "Static analysis for safe destructive updates in a functional language," in *Selected papers from the 11th International Workshop on Logic Based Program Synthesis and Transformation*. London, UK: Springer-Verlag, 2001, pp. 1–24.
[16] T. Terauchi and A. Aiken, "Witnessing side-effects," in *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, 2005, pp. 105–115.
[17] J. A. Tov and R. Pucella, "Practical affine types," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011, pp. 447–458.
[18] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, Jan. 1992, pp. 1–14.
[19] M. Wand and W. Clinger, "Set constraints for destructive array update optimization," *Journal of Functional Programming*, vol. 11, no. 3, pp. 319–346, 2001.