Copy-on-Write in the PHP Language

Akihiko Tozawa Michiaki Tatsubori
Tamiya Onodera
IBM Research, Tokyo Research Laboratory
atozawa@jp.ibm.com,
mich@acm.org,tonodera@jp.ibm.com

Yasuhiko Minamide
Department of Computer Science
University of Tsukuba
minamide@cs.tsukuba.ac.jp

Abstract

PHP is a popular language for server-side applications. In PHP, assignment to variables copies the assigned values, according to its so-called *copy-on-assignment* semantics. In contrast, a typical PHP implementation uses a *copy-on-write* scheme to reduce the copy overhead by delaying copies as much as possible. This leads us to ask if the semantics and implementation of PHP coincide, and actually this is not the case in the presence of sharings within values. In this paper, we describe the copy-on-assignment semantics with three possible strategies to copy values containing sharings. The current PHP implementation has inconsistencies with these semantics, caused by its naïve use of copy-on-write. We fix this problem by the novel *mostly copy-on-write* scheme, making the copy-on-write implementations faithful to the semantics. We prove that our copy-on-write implementations are correct, using bisimulation with the copy-on-assignment semantics.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General

General Terms Design, Languages

1. Introduction

Assume that we want to maintain some data locally. This local data is mutable, but any change to it should not affect the global, master data. So, we may want to create and maintain a copy of the master data. However such copying is often costly. In addition, the copied data may not be modified after all, in which case the cost of copy is wasted. This kind of situation leads us to consider the *copy-on-write* technique.

Copy-on-write is a classic optimization technique, based on the idea of delaying the copy until there is a write to the data. The name of the technique stems from the copy of the original data being forced by the time of the write. One example of copy-on-write is found in the UNIX *fork*, where the process-local memory corresponds to the local data, which should be copied from the address space of the original process to the space of the new process by the fork operation. In modern UNIX systems, this copy is usually delayed by copy-on-write.

Another example is found in the PHP language, a popular scripting language for server-side Web applications. Here is an example with PHP's associative arrays.

```
$r["box"] = "gizmo";
$l = $r;           // assignment from $r to $l
$l["box"] = "gremlin";
echo $r["box"];   // prints out gizmo
```

The change of `$l` at Line 3, following the assignment `$l = $r`, only has local effects on `$l` which cannot be seen from `$r`. The behavior or semantics in PHP is called *copy-on-assignment*, since the value of `$r` seems to be copied before it is passed to `$l`. We can consider the copy-on-write technique to implement this behavior. Indeed, the by far dominant PHP runtime, called the Zend runtime[1], employs copy-on-write and delays the above copy until the write at Line 3.

For readers in the functional or declarative languages community, the semantics of PHP arrays may first sound like a familiar one, e.g., PHP arrays are similar to *functional arrays*. However their similarity becomes less clear as we learn how we can *share* values in PHP. In PHP, we have the reference assignment statement, `=&`, with which we can declare a sharing between two variables. Such a sharing breaks the locality of mutation. For example, the write to `$y` is visible from `$x` in the following program.

```
$x[0] = "shares me";
$y =& $x;                 // creates sharing
$y[0] = "shared you";
echo $x[0];               // shared you
```

Now, our question is as follows. The copy-on-write is considered as a runtime optimization technique to reduce useless copies. Then, does the use of copy-on-write preserve the equivalence to the original semantics, in which we did not delay copying? This equivalence might be trivial without a sharing mechanism as above, but is not clear when we turn to PHP. In PHP, we can even share a location *inside a value*. This is where the problem gets extremely difficult.

```
$r["box"] = "gizmo";
$x =& $r["box"];      // creates sharing inside $r
$l = $r;              // copies $r
$l["box"] = "gremlin";
echo $r["box"];       // what should it be ?
```

The result of this program should reflect how exactly PHP copies arrays when they contain sharings. Our discussion will start from clarifying such PHP's copy semantics.

In this paper, we investigate the semantics and implementation of PHP focusing on the copy-on-write technique and its problems. Our contributions in this paper are as follows.

[1] Available at `http://www.php.net`.

- We develop three *copy-on-assignment* models of PHP, each differing in the copy semantics, i.e., how sharings inside arrays are copied. Three copy semantics are called *shallow copy*, *graphical copy*, and *deep copy*. To capture sharings inside values, our formal model uses *graphs* and their rewriting.
- We identify several problems in the current PHP implementation, including the inconsistency from the copy-on-assignment semantics. In particular, we point out the *inversion of execution order* problem, which is caused by the copy-on-write optimization.
- We propose copy-on-write implementations of PHP based on the novel *mostly copy-on-write* scheme, which fixes the inversion problem by adding moderate overhead to the implementation. This fix works for all three copy strategies.
- We prove that the corresponding copy-on-assignment and mostly copy-on-write models do coincide. We use a *bisimulation* to state this coincidence. We develop proof techniques for our graph rewriting semantics using a domain theoretic approach.
- We report that our experimental implementation of the mostly copy-on-write scheme gives a desirable performance without significant overhead.

1.1 Outline

Let us outline the remainder of this paper. Section 2 covers the basics and the copy-on-assignment models. Section 3 explains the copy-on-write, its problem, and our solution by mostly copy-on-write. Section 4 discusses the correctness of our solution. Section 5 reports the experimental results. Section 6 surveys the related work, and Section 7 concludes.

2. Copy-On-Assignment Model

In this section, we discuss the semantics of PHP, called copy-on-assignment, focusing on how array values are copied when they contain sharings.

2.1 Preliminaries

In this paper, we use graph representations to capture the data and store in PHP. In a graph G, each node and edge of G is labeled. Edges of G are labeled by $c \in \mathit{Const}$. Each node is labeled by an element of $\mathit{Const} \uplus \{\Box\}$ where $\Box$ denotes array nodes. Each array node has a finite number of outgoing edges with non-overlapping labels c reaching nodes representing each element of the array associated with key c. Otherwise nodes are leaves representing constants.

We denote by $G \vdash v \xrightarrow{c} v'$ that there is a c-labeled edge from v to v'. The dynamic environment is modeled by adding the root node Λ to G. The root node is itself labeled by $\Box$, and may have several successor edges labeled by either $\mathtt{\$x} \in \mathit{Var}$ or $c \in \mathit{Const}$ representing the variable environment and constant pool[2], respectively. Each c-labeled edge is associated with a node v^c, which is again labeled by c, such that $G \vdash \Lambda \xrightarrow{c} v^c$.

The subset of PHP we deal with follows the syntax given in Figure 1, in which the overline $\bar{\ }$ is used to denote possibly 0-length sequences. Now the semantics of variable and array lookup expressions are given as follows.

$$\frac{}{G \vdash c \Downarrow v^c} \qquad \frac{G \vdash \Lambda \xrightarrow{\mathtt{\$x}} v}{G \vdash \mathtt{\$x} \Downarrow v} \qquad \frac{G \vdash e \Downarrow v \quad G \vdash e' \Downarrow v^c \quad G \vdash v \xrightarrow{c} v'}{G \vdash e[e'] \Downarrow v'}$$

[2] We introduce such edges from Λ to v^c for technical reasons, i.e., to avoid garbage collection of constant nodes.

$$\begin{aligned}
\mathit{Stmt} \ni \quad & s ::= \mathtt{\$x}\overline{[e]} = e \mid \mathtt{\$x}\overline{[e]} \mathrel{\texttt{=\&}} \mathtt{\$x}\overline{[e]} \mid \mathtt{unset}(\mathtt{\$x}\overline{[e]}) \mid \mathtt{echo}\ e \mid s; s \\
\mathit{Expr} \ni \quad & e ::= \mathtt{\$x} \mid c \mid e[e] \mid \cdots \\
\mathit{Const} \ni \quad & c ::= \mathtt{null} \mid 0 \mid 1 \mid \cdots \mid \text{""} \mid \text{"a"} \mid \text{"b"} \mid \cdots \\
\mathit{Var} \ni \quad & \mathtt{\$x} ::= \mathtt{\$x}, \mathtt{\$y}, \cdots
\end{aligned}$$

Figure 1. Reduced Syntax of PHP

We skip the side effects of expressions since they are irrelevant to the topic of the paper. To simplify the presentation, the syntax in Figure 1 even accepts some expressions rejected by the Zend implementation as syntax error, e.g., $\mathtt{null}[0]$.

2.2 Copy-on-Assignment

The assignment, or store, statement $\mathtt{\$x}\overline{[e]} = e'$ updates the dynamic environment. We model this behavior by a binary relation $G \overset{\mathtt{\$x}\overline{[e]} = e'}{\longmapsto} G'$. Now consider the following program.

```
$r["val"] = 1;
$l = $r;
$l["val"] = 2;
echo $r["val"];   // prints 1
```

As explained in the introduction, `$l = $r` creates a copy of `$r`, and subsequent writes to `$l` do not affect `$r`. This behavior can be illustrated by the following graph rewrites.

$$\begin{tikzcd}
\Lambda \arrow[r, "\mathtt{\$r}"] & \Box \arrow[r, "\text{''val''}"] & \boxed{1} & \overset{\mathtt{\$l = \$r}}{\longmapsto} & \Lambda \arrow[r, "\mathtt{\$r}"] \arrow[dr, bend right, "\mathtt{\$l}"'] & \Box \arrow[r, "\text{''val''}"] & \boxed{1} \\
& & & & & \Box \arrow[r, "\text{''val''}"] & \boxed{1} \\
& & & \overset{\mathtt{\$l[''val''] = 2}}{\longmapsto} & \Lambda \arrow[r, "\mathtt{\$r}"] \arrow[dr, bend right, "\mathtt{\$l}"'] & \Box \arrow[r, "\text{''val''}"] & \boxed{1} \\
& & & & & \Box \arrow[r, "\text{''val''}"] & \boxed{2}
\end{tikzcd}$$

When drawing a graph we elide constant edges under Λ. The rewrite $G \overset{\mathtt{\$x}\overline{[e]} = e'}{\longmapsto} G'$ is broken down into several steps.

- Copying the subgraph at v' such that $G \vdash e' \Downarrow v'$.
- Looking up a path $\Lambda \xrightarrow{\mathtt{\$x}} v_1 \xrightarrow{c_1} \cdots v_n \xrightarrow{c_n} v_{n+1}$ where $\overline{e}$ evaluates to constant nodes for $c_1, \ldots, c_n$.
- The copied graph replacing the node v_{n+1}.

The formal definition will be given later, after we look through PHP references and their problems.

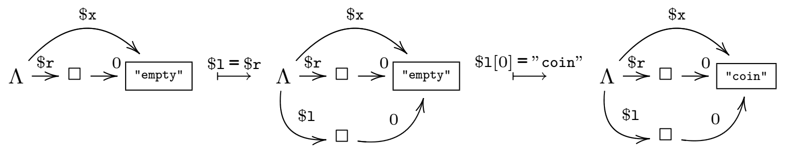
2.3 References in PHP

As long as we use the ordinary assignment $\mathtt{\$x}\overline{[e]} = e'$, the graph representing PHP's dynamic store does not contain any sharing structure. At this moment, programmers are in a safer but restricted world, where updates of values have no destructive effects seen from remote places. However, PHP also provides the reference assignment operator $\mathtt{\$x'}\overline{[e']} \mathrel{\texttt{=\&}} \mathtt{\$x}\overline{[e]}$, by which one can enjoy any advantage of destructive effects as in C or Java, e.g., creating sharing structures, creating back edges or cycles, implementing algorithms otherwise inefficient, and so on.

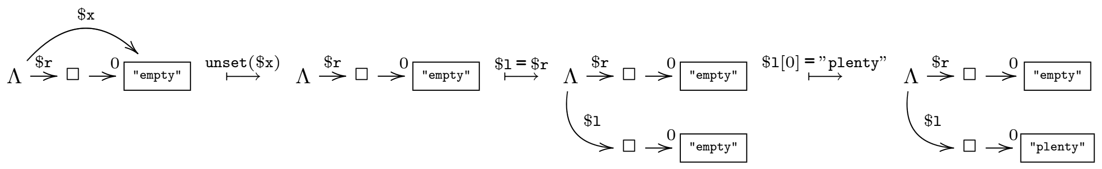
The syntax `=&` indicates the combination of address-of (`&`) and assignment operator (`=`) in C. However, there is an important difference in the meaning between PHP and C.

```
$r["hand"] = "empty";
$x =& $r["hand"];        // creates a sharing
```

```
1: $r[0] = "empty";
2: $x =& $r[0];
3: $l = $r;
4: $l[0] = "coin";
5: echo $r[0];
```



```
1: $r[0] = "empty";
2: $x =& $r[0];
3: unset($x);
4: $l = $r;
5: $l[0] = "plenty";
6: echo $r[0];
```



```
1: $r[0] = "trick";
2: $x =& $r[0];
3: $l = $r;
4: unset($x);
5: $l[0] = "treat";
6: echo $r[0];
```

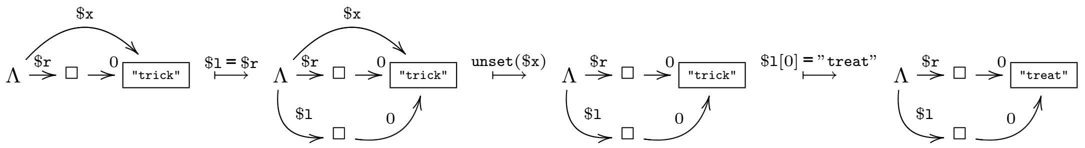
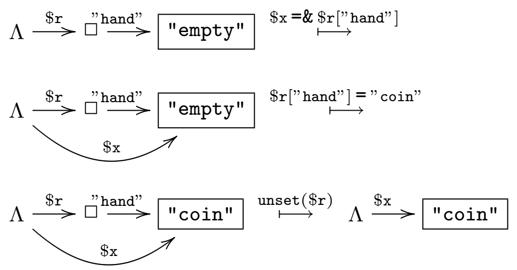


Figure 2. Some examples in copy-on-assignment PHP(s)

```
$r["hand"] = "coin";
unset($r);              // freeing $r
echo $x;                // safe
```

This example demonstrates that in PHP, there is no dangling pointer problem, i.e., reading from a reference whose referent already does not exist. We here model references by shared nodes, i.e., nodes with multiple incoming edges. References in PHP do not indicate any direction from a reference to its referent. Rather, two references, which are symmetric to each other, simply share the same value. So, removing the one side, i.e., $r["hand"], results in a value still seen from the other side $x being no longer shared[3].



Now, by combining two types of assignments = and =&, we may even copy a value, e.g., $l = $r, in which a reference assignment, e.g., $x =& $r["hand"], has introduced a sharing. What will happen then? The right answer is not readily obvious, and actually it seems to be an important design point of the language.

Let's see what happens by running the first program in Figure 2. The Zend runtime interpreting this program prints out "coin", which means that array entries $r[0] and $l[0] are shared before the write at Line 4. Such a sharing should have been created at $l = $r at Line 3. This evidently shows that while copying a value held by $r, the Zend runtime skips the shared node, and instead adds $l[0] to the sharing. We call this behavior *shallow copy*.

2.4 Discussion of PHP's Copy Semantics

Having a specification of a language is important in many respects. Such a specification is necessary for creating new implementations of the language. It also helps us discuss the correctness of optimizations, or design a program analysis, e.g., PHP string analysis by Minamide (Minamide 2005). Unfortunately, in the case of PHP,

[3] PHP references are also close to *hard links* in UNIX file systems.

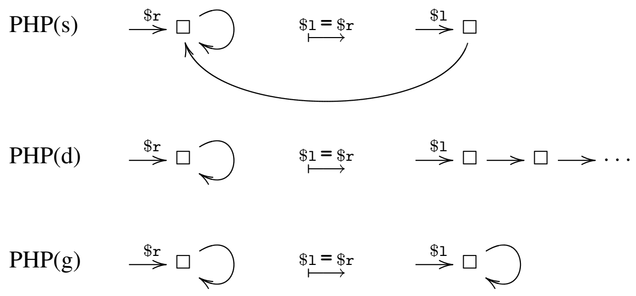


Figure 3. Copying a value in three semantics

we think specifications are not given sufficiently. The official manual[4] explains the copy-on-assignment as follows:

> *In PHP, variables are always assigned by value. That is to say, when you assign an expression to a variable, the entire value of the original expression is copied into the destination variable.*

However, it is not clear from here, or elsewhere in the manual, how values containing references are copied. As a result, the current behavior of the implementation sometimes causes a confusion, and is often criticized, e.g., in the PHP bugzilla mailing list[5].

In this paper, we give a formal copy-on-assignment model of the shallow copy. For comparison, we also discuss two other possible copy semantics classified as variations of deep copy, in the sense that they do not create sharings between the original and copy. Note that our intent here is not to discuss which one of the copy semantics is superior, but is rather to give their better understanding that might improve the current situation.

The following three PHP semantics differ in the way they copy values in assignment statements.

- (s)hallow copy semantics, referred to as PHP(s).

[4] `http://www.php.net/docs.php`

[5] `http://bugs.php.net/`. For example, some classifies the shallow copy in PHP as a bug, and the other argues that the shallow copy may cause a security problem by leaking references to copied values.

- (d)eep copy, or unraveling semantics, referred to as PHP(d).
- (g)raphical copy semantics, referred to as PHP(g).

The first is closest to the Zend runtime. It does not copy shared nodes, which are then further shared by original and copied values. The deep copy semantics may involve an infinite unraveling of a graph when we try to copy cyclic structures. The last alternate semantics is graphical copying, which copies the induced subgraph, preserving the structure. See Figure 3 for their differences when copying a cycle.

Let us give some examples. First, go back to the first example in Figure 2. In PHP(d) and PHP(g), `$l = $r` at Line 3 completely copies an array, so that `$l[0] = "coin"` does not affect `$r`. Hence we get results different from PHP(s).

Then, move to the second program in the same figure. This time, even if we use the shallow copy semantics, `$l = $r` at Line 4 copies the entire array. The difference from the first program is the `unset($x)` at Line 3, which removes variable `$x`, which had shared a node with `$r[0]`. This is another point where PHP references differ from C pointers or ML references. In PHP, a sharing once created can disappear if the other side of the sharing is removed. This poses some difficulty in the implementation of shallow copy, in which copying a node requires the information as to whether or not the node is shared. In other words, how values are copied in PHP(s) relies on *reference counts*. Indeed, the Zend runtime uses a reference counting mechanism here. In contrast, PHP(d) and PHP(g) do not require reference counting, at least for correct implementation.

Let us give another example that might illustrates how the difference in copying strategy has a practical relevance. Look at function calls in PHP.

```
  function foo($a) { $a["passwd"] = ":-X"; }
  $x = null;
  foo($x);
  echo $x["passwd"];    // prints nothing
```

As we see, by default, parameter passing at function calls correspond to the assignment = from actual parameters to formal parameters. So, writes to formal parameters inside the function body should not be seen from the caller. On the other hand, if we really need a function that updates given actual parameter variables, we use the pass-by-reference mechanism, corresponding to =&.

```
  function bar(&$a) { $a["answer"] = ":-)"; }
  $x = null;
  bar($x);
  echo $x["answer"];    // prints :-)
```

One might see the similarity of the pass-by-reference annotation in PHP, to `const` modifiers in C++, `final` fields in Java, or even ML types identifying data structures with destructive references. We think their similarity is the enforcement of the program safety. A program can be safer if it restricts the portion of unwanted updates of values, and is given annotations wherever any such destructive update may or may not occur.

Let us return to the discussion of PHP's copy semantics. If we can measure the safety of program semantics by the possibility of unsafe updates, PHP(d) may be the safest strategy as it creates the least numbers of references by erasing them at each assignment. On the other hand, PHP(s) is less safe than other copying strategies. For example, PHP(s) can cause a leakage of password information from `foo($x)` by doing `$passwd =& $x["passwd"]` before calling `foo($x)`.

2.5 Graph Transformation

In the rest of this section, we define the formal copy-on-assignment semantics of PHP. First, let us summarize the notations and graph operations used throughout. The graphs we deal with are the *rooted graphs* studied in the context of term graph rewriting systems (Barendregt et al. 1987; Barendsen and Smetsers 1992). Formally, a graph G is a tuple $(V^G, \lambda^G, \delta^G)$ where

- V^G is a node set,
- $\lambda^G \in V^G \to C$ is a labeling function on a certain node label set C,
- $\delta^G \in V^G \times L \rightharpoonup V^G$ is a partial edge function on the edge label set L.

We write $G \vdash v \xrightarrow{\ell} \delta^G(v, \ell)$ when $\delta^G(v, \ell)$ is defined. A rooted graph $G = (V^G, \Lambda^G, \lambda^G, \delta^G)$ is a graph with unique root node $\Lambda^G \in V^G$. We denote by E^G the set of *edges* in G, i.e., the domain of the partial function δ^G. Here are some additional definitions.

Definition 1 (BASIC OPERATORS AND RELATIONS OVER GRAPHS).
(i) We denote by $G = \{v \xrightarrow{\ell} w\}$ a graph with two nodes v and w, and a single edge $(v, \ell) \mapsto w$.
(ii) The union $\cup$ and disjoint union $\uplus$ of graphs are the union and disjoint union of their nodes, and edge functions seen as binary relations over $E^G \times V^G$.
(iii) By the notation $G[(v, \ell) \mapsto v']$, we denote a graph $G' = (V^G, \lambda^G, \delta^G[(v, \ell) \mapsto v'])$ whose edge function is updated. Also $G[v \mapsto c]$ denotes a graph $G' = (V^G, \lambda^G[v \mapsto c], \delta^G)$ obtained by relabeling a node.
(iv) We denote by $rc_G(v)$ the reachable node set from v on G.
(v) The induced subgraph of G on node set $V \subseteq V^G$, written $G|_V$, is a subgraph of G with its node set V and containing all edges in G from V to V. We denote by $G|_V^v$ a rooted subgraph with its root $v \in V$. This v can be omitted if $v = \Lambda^G (\in V)$. We define $gc(G) = G|_{rc_G(\Lambda^G)}$.
(vi) We write $in_G(w)$ for a set of edges $(v, \ell) \in E^G$ such that $\delta^G(v, \ell) = w$. A set of outgoing edges, written $out_G(V)$, is defined for a node set $V \subseteq V^G$ as a set such that $(v, \ell) \in out_G(V)$ $(\subseteq E^G)$ iff $v \in V$ and $\delta^G(v, \ell) \notin V$.
(vii) A rooted graph H is homomorphic to G written $G \preceq H$, if there is a label-preserving mapping $\varphi \in V^H \to V^G$ such that 1) $\varphi(\Lambda^H) = \Lambda^G$, 2) $(v, \ell) \in E^H$ iff $(\varphi(v), \ell) \in E^G$, and 3) $\varphi(\delta^H(v, \ell)) = \delta^G(\varphi(v), \ell)$ if $(v, \ell) \in E^H$. We say G and H are isomorphic, written $G \simeq H$, if $G \preceq H$ and $H \preceq G$.

The redirections are the basic steps of our transformation. We use two kinds of redirections: *edge redirection* and *node redirection*, denoted by $[\epsilon \hookrightarrow v]$ and $[v \hookleftarrow w]$, respectively. Here are examples.

$$\begin{tikzcd}
\Lambda \arrow[rr] \arrow[dr] & & v & {[(w,\ell) \hookrightarrow v] =} & \Lambda \arrow[rr] \arrow[dr] & & v \\
& w \arrow[r, "\ell"] & \cdots & & & w \arrow[ur, "\ell"] &
\end{tikzcd}$$

$$\begin{tikzcd}
\Lambda \arrow[rr, bend left] \arrow[rr] \arrow[dr] & & v \cdots & {[v \hookleftarrow w] =} & \Lambda \arrow[r, bend left] \arrow[r] \arrow[r, bend right] & w \arrow[r] & \square \\
& w \arrow[r] & \square & & & &
\end{tikzcd}$$

The edge redirection redirects a single edge ϵ $(= (w, \ell))$ to a node v. Note that some nodes in G may become unreachable after this rewrite, and will be removed. We can also simply remove an edge ϵ by writing $[\epsilon \hookrightarrow \bot]$. The node redirection means that all incoming edges to v are redirected to w. The edge redirection is formally defined as $G[\epsilon \hookrightarrow v] = gc(G[\epsilon \mapsto v])$. We can extend this rewrite

$$\frac{G \vdash \Lambda \xrightarrow{c} v}{G \vdash c \Downarrow v}\ [\textsc{Const}] \quad \frac{G \vdash \Lambda \xrightarrow{\$\mathtt{x}} v}{G \vdash \$\mathtt{x} \Downarrow v}\ [\textsc{Var}] \quad \frac{G \vdash e \Downarrow v \quad G \vdash e' \Downarrow v' \quad G \vdash v \xrightarrow{\lambda_G(v')} v''}{G \vdash e[e'] \Downarrow v''}\ [\textsc{Lookup}]$$

$$\frac{G \vdash \overline{e} \Downarrow \overline{v} \quad G \vdash e' \Downarrow v' \quad (G', \epsilon) = \mathit{find}(\mathit{copy}_{(x)}(G, v'), \Lambda, \$\mathtt{x}; \lambda^G(\overline{v}))}{G \overset{\$\mathtt{x}\overline{[e]}\,\texttt{=}\,e'}{\longmapsto}_{(x)} G'[\delta^{G'}(\epsilon) \longleftrightarrow \delta^{G'}(\Lambda, \$\$)][(\Lambda, \$\$) \hookrightarrow \bot]}\ [\textsc{Assign}]$$

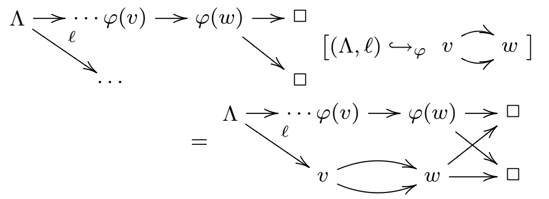
$$\frac{G \vdash \overline{e} \Downarrow \overline{v} \quad (G_1, \epsilon_1) = \mathit{find}(G, \Lambda, \$\mathtt{x}; \lambda^G(\overline{v})) \quad G_1 \vdash \overline{e}' \Downarrow \overline{v}' \quad (G_2, \epsilon_2) = \mathit{find}(G_1, \Lambda, \$\mathtt{x}'; \lambda^{G_1}(\overline{v}'))}{G \overset{\$\mathtt{x}'\overline{[e']}\,\texttt{=\&}\,\$\mathtt{x}\overline{[e]}}{\longmapsto}_{(x)} G_2[\epsilon_2 \hookrightarrow \delta^{G_2}(\epsilon_1)]}\ [\textsc{AssignRef}]$$

$$\frac{G \vdash \overline{e} \Downarrow \overline{v} \quad (G', \epsilon) = \mathit{find}'(G, \Lambda, \$\mathtt{x}; \lambda^G(\overline{v}))}{G \overset{\mathtt{unset}(\$\mathtt{x}\overline{[e]})}{\longmapsto}_{(x)} G'[\epsilon \hookrightarrow \bot]}\ [\textsc{Unset}] \quad \frac{G \vdash e \Downarrow v}{G \overset{\mathtt{echo}\ e}{\longmapsto}_{(x)} \lambda^G(v), G}\ [\textsc{Echo}] \quad \frac{G \overset{s}{\longmapsto}_{(x)} \overline{c}, G'}{\langle G, s; \overline{s}\rangle \overset{\overline{c}}{\Longmapsto}_{(x)} \langle G', \overline{s}\rangle}\ [\textsc{Seq}]$$

Figure 4. Copy-on-assignment PHP(x)

to $G[\mathcal{E} \hookrightarrow v]$ for a set of edges $\mathcal{E} \subseteq E^G$, so as to simultaneously redirect all edges $\epsilon \in \mathcal{E}$ to v. Then the node redirection $G[v \longleftrightarrow w]$ is defined as $G[in_G(v) \hookrightarrow w]$.

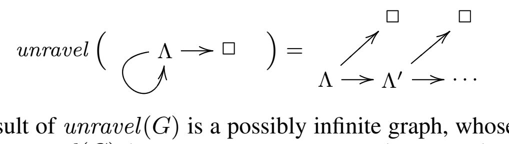
Slightly more difficult, but a useful graph rewrite is the *graph extension*, written $G[\epsilon \hookrightarrow_\varphi G']$ where φ is a mapping from $V^{G'}$ to V^G. Here is an example.



The idea is to extend G by G' by first redirecting an edge ϵ to G''s root, and then by redirecting G''s outgoing edges to G. More precisely, we look at the counterpart $G|_{\varphi(V^{G'})}$ of G', and for each outgoing edge $(\varphi(v), \ell) \in out_G(\varphi(V^{G'}))$ from there, we create a copy (v, ℓ) of this edge and redirect it to G. Note that in graph extension $G[\epsilon \hookrightarrow_\varphi G']$, we always assume that G and G' are disjoint graphs. Otherwise, we create a fresh copy of G' and perform the extension using this copy. Formally, $G[\epsilon \hookrightarrow_\varphi G'] = (G \uplus G')[\epsilon \hookrightarrow \Lambda^{G'}][(v_1, \ell_1) \hookrightarrow \delta^G(\varphi(v_1), \ell_1)] \cdots [(v_k, \ell_k) \hookrightarrow \delta^G(\varphi(v_k), \ell_k)]$ where $(\varphi(v_i), \ell_i) \in out_G(\varphi(V^{G'}))$.

We sometimes consider *path-preserving* φ in the definition $G[\epsilon \hookrightarrow_\varphi G']$. In that case we write $G[\epsilon \hookrightarrow_v G']$ where v corresponds to $\varphi(\Lambda^{G'})$. Here we say φ is path preserving if $\varphi(\delta^{G'}(w, \ell)) = \delta^G(\varphi(w), \ell)$ wherever $\varphi(w)$ and $\delta^{G'}(w, \ell)$ are defined. Note that a path-preserving mapping is close to homomorphism, but it disregards node labels and graphs roots, and also does not preserve the number of edges from a node. Such a mapping φ, if exists, is uniquely determined from $v (= \varphi(\Lambda^{G'}))$.

Finally, the graph unraveling is illustrated as follows.



The result of $\mathit{unravel}(G)$ is a possibly infinite graph, whose node set of $\mathit{unravel}(G)$ has one-to-one correspondence to the set of paths $\overline{\ell}$ such that $\Lambda \xrightarrow{\overline{\ell}} v$ in G. We give the edge function by $\delta^{\mathit{unravel}(G)}(\overline{\ell}, \ell) = \overline{\ell}; \ell$ if $\overline{\ell}$ and $\overline{\ell}; \ell$ are in $V^{\mathit{unravel}(G)}$. Here we write $\overline{\ell}; \ell$ for a concatenation. An important property of $\mathit{unravel}$ is that for any G, H such that $G \preceq H$, we have $H \preceq \mathit{unravel}(H) \simeq \mathit{unravel}(G)$.

2.6 Copy-on-Assignment PHP(x)

The copy-on-assignment operational semantics of PHP(x) is given in Figure 4. Let us first explain several utility functions used in the definition.

The function $\mathit{copy}_{(x)}(G, v)$, parameterized by the copying strategy $x \in \{\mathrm{s}, \mathrm{d}, \mathrm{g}\}$ defines how we copy values at assignment.

Definition 2 (Copy Operation)**.**

$$\mathit{copy}_{(x)}(G, v) = G[(\Lambda, \$\$) \hookrightarrow_{\varphi_{(x)}} G_{(x)}]$$

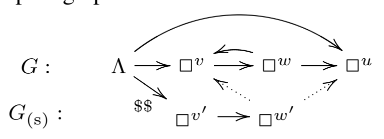
Given a graph G and target node v, this returns a new graph, in which the copy of a subgraph of G is created and stored in the temporary variable $\$\$$. We define each $G_{(x)}$ as follows.

- $G_{(\mathrm{g})}$ is $G|^v_{rc_G(v)}$.
- $G_{(\mathrm{d})}$ is $\mathit{unravel}(G|^v_{rc_G(v)})$.
- $G_{(\mathrm{s})}$ is $G|^v_{\{v\} \cup \{w \mid G \vdash v \xrightarrow{c_1} w_1 \cdots \xrightarrow{c_k} w_k (=w) \text{ s.t. } |in_G(w_i)| = 1\}}$.

The intuition here is as follows: The graphs $G_{(\mathrm{g})}$, $G_{(\mathrm{d})}$, and $G_{(\mathrm{s})}$ are G's induced subgraph for the reachable node set from v, the unraveling of G's induced subgraph for nodes reachable from v, and G's induced subgraph for nodes reachable from v without passing references, respectively.

We also need to specify a mapping $\varphi_{(x)}$ used in the extension operator. For $x \in \{\mathrm{d}, \mathrm{g}\}$, we define $\varphi_{(x)}$ as a path-preserving mapping, explained earlier, that maps a node in $G_{(x)}$ to the other node in G reached by the same path[6]. For PHP(s), we define this $\varphi_{(\mathrm{s})}$ similarly except that we leave $\varphi_{(\mathrm{s})}(\Lambda^{G_{(\mathrm{s})}})$ undefined.

The definition of PHP(s) requires an explanation. Let us give one example. In PHP(s), we should repair the sharing between the original and copied graphs.



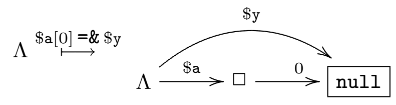
Here we create a copy $G_{(\mathrm{s})}$ of the subgraph of G at v excluding references. Assume that copied nodes are v' and w'. According to the definition of $\varphi_{(\mathrm{s})}$, in this case, $\varphi_{(\mathrm{s})}(V^{G_{(\mathrm{s})}}) = \{w\}$. So, the edge $w \longrightarrow u$ is an outgoing edge from $\varphi(V^{G_{(\mathrm{s})}})$, and is copied as an edge $w' \longrightarrow u$ to the shared node u. Now, we similarly add an edge $w' \longrightarrow v$. Note that any edge to the root node v' in

[6] We also write $G[(\Lambda, \$\$) \hookrightarrow_{\varphi_{(x)}} G_{(x)}]$ as $G[(\Lambda, \$\$) \hookrightarrow_v G_{(x)}]$ for $x \in \{\mathrm{g}, \mathrm{d}\}$.

$G_{(\mathrm{s})}$ implies that v is shared, so that such an edge to v' should be redirected to v. This is why we exclude v from $\varphi_{(\mathrm{s})}(V^{G_{(\mathrm{s})}})$.

In other copy semantics, the node set $\varphi_{(x)}(V^{G_{(x)}})$ does not have any outgoing edge, so we never create sharings between the original and copied graphs.

The first step of the assignment statement $\mathtt{\$x}\overline{[e]}\,\mathtt{=}\,e'$ is to copy the value of e'. The second step is called the *find* step which is to search inside arrays for the target node corresponding to $\mathtt{\$x}\overline{[e]}$. We also use *find* in the reference assignment $\mathtt{\$x'}\overline{[e']}\,\mathtt{=\&}\,\mathtt{\$x}\overline{[e]}$ to find the targets of both $\mathtt{\$x}\overline{[e]}$ and $\mathtt{\$x'}\overline{[e']}$. In PHP, not only missing target array entries, but also new arrays that are not found on the path to the target node are created, as shown in the following example.



The assignment statement performs the *copy* operation earlier than the *find* operation. This order is crucial, since the latter rewrites the graph for creating the path to the target node, e.g., consider $\mathtt{\$x}[0] = \mathtt{\$x}$. Also during the *find* operation, PHP raises an error if nodes other than `null` and $\Box$ already exist where we need arrays. The following function $\mathit{find}(G, v, \overline{c})$ is undefined in such a case. Note also that the function does not return the target node, but rather the edge reaching the target node.

Definition 3 (FIND OPERATION). *(i)*

$$\mathit{find}(G,v,c) = \begin{cases} \mathit{undefined} & \mathit{If}\ \lambda^G(v) \notin \{\mathtt{null}, \Box\} \\ (G,(v,c)) & \mathit{If}\ G \vdash v \xrightarrow{c} v' \\ (\hat{G},(v,c)) & \mathit{Otherwise}\ (*) \end{cases}$$

$$\mathit{find}(G,v,c;\overline{c}) = \begin{cases} \mathit{undefined} & \mathit{If}\ \lambda^G(v) \notin \{\mathtt{null}, \Box\} \\ \mathit{find}(G,v',\overline{c}) & \mathit{If}\ G \vdash v \xrightarrow{c} v' \\ \mathit{find}(\hat{G},\hat{v},\overline{c}) & \mathit{Otherwise}\ (*) \end{cases}$$

where $\hat{v}$ is a fresh node, and $\hat{G} = (G \cup \{v \xrightarrow{c} \hat{v}\})[v \mapsto \Box, \hat{v} \mapsto$ `null`$]$. (ii) $\mathit{find}'(G,v,\overline{c})$ is defined similarly, but is undefined for the cases (*).

Now we can define the semantics of PHP(x) as in Figure 4. The last rewrite step for $\mathtt{\$x}\overline{[e]}\,\mathtt{=}\,e'$, called the *redirection* step, first redirects all edges to the location $\mathtt{\$x}\overline{[e]}$, to the value at the temporary variable, and then we remove the temporary variable. The last step for $\mathtt{\$x'}\overline{[e']}\,\mathtt{=\&}\,\mathtt{\$x}\overline{[e]}$ is a single edge-redirection, i.e., we redirect an edge to $\mathtt{\$x'}\overline{[e']}$ to the node at $\mathtt{\$x}\overline{[e]}$. The small-step semantics $\overset{\overline{c}}{\Longmapsto}_{(x)}$ is a binary relation over pairs of dynamic environment graphs G and sequences of statements $\overline{s}$. In $\langle G, s; \overline{s}\rangle \overset{\overline{c}}{\Longmapsto}_{(x)} \langle G', \overline{s}\rangle$, each step corresponds to the execution $\overset{s}{\longmapsto}_{(x)}$ of the atomic statement s. The side-effect $\overline{c}$ is either an empty sequence, or a singleton sequence of a literal value, corresponding to the output of `echo` e.

3. Copy-On-Write Model

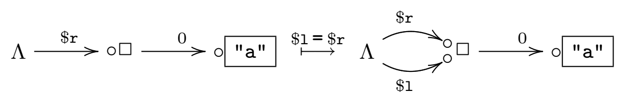
The copy-on-assignment models of PHP defined in Section 2 give the semantics or specification of the language. In this section, we first informally introduce a new graph-rewriting model for the *copy-on-write* scheme, corresponding to the implementation. We then explain that a straightforward copy-on-write model, which coincides with the Zend runtime, exposes a discrepancy from the copy-on-assignment model by inverting the execution of some statements in a program. Since there is no complete specification of PHP, this does not imply that the implementation is *wrong*. However, this discrepancy is still very unintuitive for users, whose understanding should be based on copy-on-assignment.

The latter half of this section discusses our refined copy-on-write scheme, and gives its formal treatment.

3.1 Sharing and Splitting

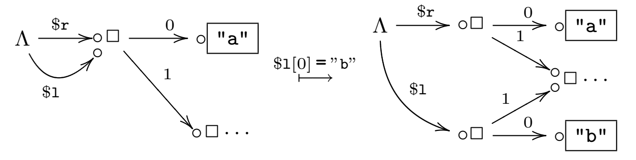
In copy-on-assignment graphs, sharings are by references through which we write to the same location from different places. In the copy-on-write graph, there are two kinds of sharings: one which corresponds to references, and the other for delaying copies.

In copy-on-write, the assignment $\mathtt{\$l} = \mathtt{\$r}$ also creates a sharing:



We use circles $\circ$ within graphs to indicate *ports*, which are means to distinguish sharings created by assignment `=` from those created by reference assignment `=&`. Multiple edges to the same port represent sharing by reference, while multiple edges to the same node, with distinct ports, represent temporary sharing for a copy-on-write operation. We denote by *reference* ports those with multiple incoming edges, and by *copying* ports those on nodes with multiple ports.

If a write occurs on a path containing copying ports, we *split* the graph.



The split operation above is similar to the *copy* operation. The difference is that we do not duplicate the entire subgraph, but only the path to the node to be written.

3.2 Copy-on-Write Inverts Execution Order

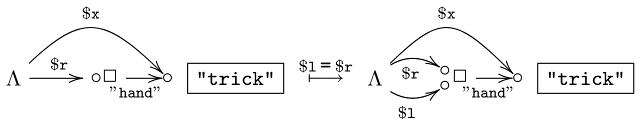
We have to be careful here. The copy-on-write delays copying, and we are not sure if such a delay never exposes itself, e.g., as an inversion of the expected execution order.

Unfortunately, it indeed does. We take the third program in Figure 2 as an example. Let us use the shallow copy semantics, PHP(s), to interpret this program:

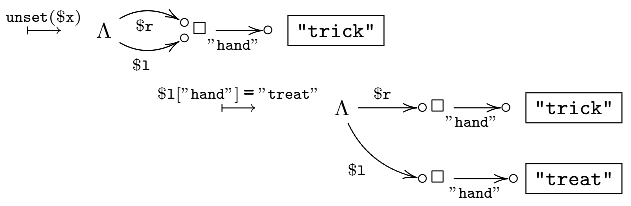
```
1: $r["hand"] = "trick";
2: $x =& $r["hand"];
3: $l = $r;              // <-
4: unset($x);            // <-
5: $l["hand"] = "treat";
6: echo $r["hand"];      // "treat" expected
```

We expect here the same result as the case without `unset($x)`, since this `unset($x)` comes after `$l = $r` which already added `$l["hand"]` to the sharing at `$r["hand"]`. However the Zend runtime gives a different answer – it answers `"trick"`, which is the result expected if `unset($x)` was executed before `$l = $r`. In other words, the execution order was inverted between the copy at Line 3 and the unset at Line 4.

This inversion can be explained by the copy-on-write model, which we believe is not far from the implementation of the Zend runtime.



The difference from the copy-on-assignment graph is the reference counts. Since this time we share the node at `$r` by `$l = $r`, the reference count of the `"trick"` node remains as 2, while it was 3 in the copy-on-assignment graph. However this is already problematic, because in the graph after `unset($x)`, the "`trick`" node is no longer considered as shared.

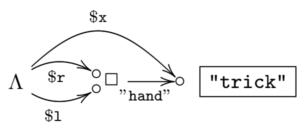


Now the delayed copy is resumed at $1["hand"] = "treat", copying the port on the "`trick`" node before the write, and giving the same result for `echo` as the Zend runtime.

Note that what we see here is not a problem inherent to shallow copy. Other copy semantics also have similar, and rather obvious, inversion problems. For example, consider the following program:

```
1: $r["hand"] = "trick";
2: $x =& $r["hand"];
3: $1 = $r;            // <-
4: $x = "treat";       // <-
5: echo $1["hand"];    // "trick" expected
```

When a value is copied deeply (or graphically), references inside it are also copied rather than skipped, so that we expect that the write to $x at Line 4 is only visible from $r["hand"]. However in the copy-on-write model, we are in the following state before this write.

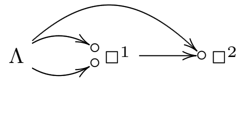


Here it is difficult for $x = "treat" to modify only $r["hand"] but not $1["hand"]. If we modified both, we see the inversion of the execution of Line 3 and Line 4.

3.3 Mostly Copy-on-Write Scheme

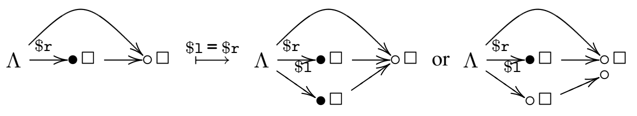
We propose a novel solution for the inversion problem. The key idea is to force a split at the assignment, if the value at the right hand side is *dirty*, i.e., containing references. We assume that PHP programs do not create dirty arrays very often, so that the cost of additional copies is not significant. We can still safely delay copying in other common cases. We call our solution *mostly copy-on-write*.

Let us look into the problem again. It seemed that we went wrong with states such as this:



We think that the above state is already *too late*. At this state, nothing prevents us from, without splitting the node 1, modifying the node 2, either by an `unset` reducing the reference counts, or by a write updating the content of the node. The root of the inversion problem comes from this unset or write, which should have taken place after we split the node 1. It is also difficult to force such a split at the time of the unset or write, which requires the reverse traversal of the graph to find the targets of splits, adding additional complexity and significant costs to the implementation.

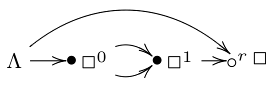
Instead, our scheme forces a split one step before the *too late* state, i.e., when we try to share the node 1. In fact, we do not allow such a sharing. This node 1 is called dirty, which we indicate with a marker •.



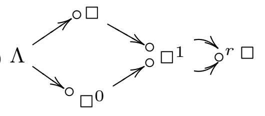
This time the statement $1 = $r forces a split of the graph by copying the array node. This results in the left side graph in PHP(s)[7], and the right side graph in PHP(g). PHP(d) also creates the right side graph with a minor change explained later. Confirm that the subsequent unset or write on the result graphs then will not cause inversion problems.

Then, when should a node be called dirty? A simplistic answer is "if it represents a value containing references". However this definition can be refined by carefully examining when an update of a reference is seen from the other nodes. We say a graph is *well-colored* if it correctly marks ports on dirty nodes. Here are examples of well-colored graphs according to our definition.

(A)



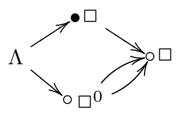
(B)



In graph (A), either unset of an upper edge to port $\circ^r$, or a write through it has an effect, which can be seen from either node 0 or 1, and thus nodes 0 and 1 are dirty. On the other hand, the well-coloredness of graph (B) says that node 0 is clean. This is because any update through the upper path to the reference port $\circ^r$ first triggers the split of this path, which blinds the effect of the update from node 0. For example, even if we try to unset one of the edges to $\circ^r$ in PHP(s), the reference count of $\circ^r$ does not drop to 1, since the split at node 1 raises the reference count of $\circ^r$ to 4 before its drop. Likewise, in PHP(d) or PHP(g), we cannot see from node 0 any write to the node at $\circ^r$ through the upper path, since the split at node 1 duplicates port $\circ^r$ before the write.

Let us also look at the following graph (C).

(C)



The graph (C) is not safe in PHP(g) and PHP(d), but is safe in PHP(s), e.g., unset of an upper edge does not turn a reference to non-reference, so that the effect of unset is not visible from node 0. In this paper, we define the well-coloredness for PHP(s) differently from that for other copy semantics. We first explain PHP(g) and PHP(d), and then will later repeat the discussion for PHP(s). In what follows, we say *well-colored* to denote the graph coloring property for PHP(g) and PHP(d), and use *shallowly well-colored* to denote the property for PHP(s).

The copy-on-write graphs consist of nodes and ports. Let us use $p, q, r, \dots$ to range over ports, $v, w, \dots$ to range over nodes, and $n, m, \dots$ to range over both ports and nodes. For example, each port p which itself is a successor of a certain node v, always has a single node successor w. We say a node v *dominates* n if all paths from Λ to n pass this v, and in particular, v *strictly dominates* n if v dominates n and $v \neq n$. We denote by a *dominance frontier* of v a set of m such that i) m is not strictly dominated by v, and ii) m is a successor of a certain port or node n dominated by v.

[7] So in the mostly copy-on-write model of PHP(s), we keep the color of the entry port of the copied subgraph, while such port becomes clean in other copy semantics. This is because some references may be still shared by the original and copied values in PHP(s).

Definition 4. *1) We say a non-port node v is dirty iff its dominance frontier include ports.*
2) A graph G is well-colored iff i) the ports on dirty nodes are marked dirty, and ii) the copying ports are marked clean.

In graph (A), node 0's dominance frontier is $\{\circ^r\}$, while in graph (B), it only contains node 1. Hence node 0 is dirty in graph (A) but not in graph (B). Note that the well-coloredness does not require the minimality of marking, e.g., we can mark the port on node 0 in graph (B) dirty, since it is not a copying port.

3.4 Maintaining the Well-Coloredness

Before giving formal models of mostly copy-on-write, we discuss the *coloring* process required to maintain well-coloredness. The key idea of mostly copy-on-write was to force splits of the graph one step before the *too late* states. That means the coloring process should take place two steps before such states, i.e., at the reference assignment $x =& $r["hand"].

$$\begin{tikzcd}
\Lambda \arrow[r, "\$r"] & \circ\Box \arrow[r, "0"'] & \circ \; \boxed{\texttt{"trick"}} \arrow[r, mapsto, "\$x\,=\&\,\$r[0]"] & \Lambda \arrow[r, "\$r"] \arrow[rr, bend left, "\$x"] & \circ^{*}\Box \arrow[r, "0"'] & \circ \; \boxed{\texttt{"trick"}}
\end{tikzcd}$$

We call the port $\circ^*$ a *false-negative*, which should be marked dirty after the reference assignment. For PHP(g) and PHP(d), the only chances for introducing false negatives are at =&, while this is not the case in PHP(s). Fortunately, we have linear time algorithms to identify false negatives in either case. Here, we give an algorithm for PHP(g) and PHP(d) that finds false negatives after the execution of =&.

Algorithm 1 (Marking False-Negatives at =&). *In the mostly copy-on-write models given later, rewrites for* x'\overline{[e']}$ =& x\overline{[e]}$ *consist of first finding two paths, as illustrated below, reaching the ports to be shared as a reference, and then redirecting the edge drawn as* $\rightsquigarrow$ *to port p. Assume that during the find steps, we split the paths to the ports to be shared, i.e., none of $p_1, .., p_{k-1}$ and $q_1, .., q_{h-1}$ are copying ports.*[8]

$$\begin{tikzcd}
\Lambda \arrow[r, "\$x"] \arrow[dr, "\$x'"'] & \circ^{p_1}\Box \arrow[r, dashed] & \circ^{p_I=q_J}\Box \arrow[r, dashed] \arrow[dr, dashed] & \bullet^{p_{k-1}}\Box \arrow[r] & \circ^{p}\Box \arrow[d, dotted, no head] \\
& \circ^{q_1}\Box \arrow[ur, dashed] & & \bullet^{q_{h-1}}\Box \arrow[r, squiggly] & \circ^{q}
\end{tikzcd}$$

The coloring algorithm then proceeds as follows. We identify the largest I and J such that p_I appearing in $q_1, .., q_{h-1}$ and q_J appearing in $p_1, .., p_{k-1}$. This process is done in $O(k + h)$-time. Let $I = J = 0$ if there are no such p_I and q_J. Then we mark all p_i and q_j such that $i \in I + 1, .., k - 1$ and $j \in J + 1, .., h - 1$.

The correctness of this algorithm follows from the following remark.

Remark 1. *If the graph is well-colored, the above redirection introduces a false negative port r only when either $r = p_i$ such that $i \in I + 1, .., k - 1$, or $r = q_j$ such that $j \in J + 1, .., h - 1$. This is proved as follows. Assume that r is on node v. If the redirection is about to turn v dirty, either p or q is reachable from v. If q is reachable, q is dominated by v, since v is clean before the rewrite. So, v appears on any path from Λ to q, and thus $r = q_j$ for some $j \in 1, .., h - 1$. The other case is the same. Furthermore, we have $r \neq p_i$ for $i \in 1, .., I$ and $r \neq q_j$ for $j \in 1, .., J$. Otherwise, v dominated both p and q, in which case p is still dominated by v after the rewrite.*

[8] The figure might indicate that all ports p_i ($i \in 1, \ldots, I - 1$) and q_j ($j \in 1, \ldots, J - 1$) are dirty according to the definition of well-coloredness. However, this is not the case if some of them overlap, e.g., $p_1 = q_1$. Also we later introduce shallowly well-colored graphs, in which some of p_i and q_j can be non-dirty even if there is no overlap.

While it is enough to remove false-negatives to guarantee safety, we can also consider the reverse problem, i.e., unmarking *false positives*. By reducing false positives, we can reduce the chance of unnecessary copies during assignments. Unfortunately, this reverse problem is difficult. For example, rewrites like `unset($x)` introduce false-positives on ports not visited during the rewrite.

$$\begin{tikzcd}
\Lambda \arrow[r] \arrow[rrr, bend left, dashed, "\$x"] & \bullet\Box \arrow[r] & \bullet\Box \arrow[r] & \circ\Box
\end{tikzcd}$$

In general, the removal of all false-positives seems to require the traversal of the entire graph. However we may still remove some false-positives by using several techniques. For example, when copying an array, it may be possible to traverse all its elements. If none of the elements inside the array is dirty or a reference, then the array is clean. Alternatively, we can use the the algorithm by Cytron et al. (Cytron et al. 1991), also known as the minimal SSA algorithm, which computes dominance frontiers in super-linear time to the size of the graph. The cost is high but the algorithm can guarantee the minimality of markings. For example, we may run this algorithm periodically according to the estimated numbers of false positives.

To summarize, compared to the original copy-on-write scheme, the overhead added by the mostly copy-on-write scheme comes from

- Additional splits at assignment x\overline{[e]}$ = e when e evaluates to a dirty array.
- Linear-time maintenance of the well-coloredness invariant after each execution of statement, in particular at x\overline{[e]}$ =& x\overline{[e]}$ in PHP(d) and PHP(g).

The latter overhead is linear with the depth of the nested array accesses. We think this overhead is small.

3.5 Mostly Copy-on-Write PHP(g)

A copy-on-write graph is a tuple $G = (P^G \uplus W^G, \lambda^G, \delta^G)$ whose node set is partitioned into a set of ports P^G, and the set of non-port nodes W^G. λ^G is a node labeling that maps $v \in W^G$ to $\mathit{Const} \uplus \{\Box\}$, and port $p \in P^G$ to $\{\circ, \bullet, \diamond\}$. Note that $\diamond$ is used in PHP(d). The edge function δ^G maps $W^G \times (\mathit{Var} \uplus \mathit{Const})$ to P^G, and $P^G \times \{-\}$ to W^G. We write $G \vdash v \xrightarrow{c} pv'$ for an edge from v to v' going through port p.

We construct mostly-copy-on-write models for all three PHP(x). We start with PHP(g). The other copying semantics will be covered next as variations.

In copy-on-write schemes, the *split* operation is a basic operation used in several places. In case of the mostly copy-on-write scheme, a single split may duplicate a certain subgraph, not just a single node as with the original copy-on-write scheme. We cannot place multiple ports on dirty nodes, which would break our invariant. Hence a single split step forces, rather than delays, splits of all dirty nodes reachable from a given port. The rewrite $\mathit{split}_{(g)}(G, p)$ is defined on copy-on-write graphs G and port p.

$$\mathit{split}_{(g)}(G, p) = G[(p, -) \hookrightarrow G|_{\tilde{V}}^{\delta^G(p,-)}]$$

If the subscript of $\hookrightarrow_v$ is omitted, it means $v = \delta^G(p, -)$. Here $\tilde{V}$ is the minimal set satisfying

- $\delta^G(p, -) \in \tilde{V}$,
- $\delta^G(w, c) \in \tilde{V}$ if $w \in \tilde{V}$, and
- $\delta^G(p', -) \in \tilde{V}$ if $p' \in \tilde{V}$ such that p' is marked dirty.

$$\frac{G \vdash \Lambda \xrightarrow{c} pv}{G \vdash c \Downarrow v}\ [\textsc{Const}] \qquad \frac{G \vdash \Lambda \xrightarrow{\$\mathtt{x}} pv}{G \vdash \$\mathtt{x} \Downarrow v}\ [\textsc{Var}] \qquad \frac{G \vdash e \Downarrow v \quad G \vdash e' \Downarrow v' \quad G \vdash v \xrightarrow{\lambda_G(v')} pv''}{G \vdash e[e'] \Downarrow v''}\ [\textsc{Lookup}]$$

$$\frac{\begin{array}{c} G \vdash \overline{e} \Downarrow \overline{v} \quad G \vdash e' \Downarrow v' \quad (G', \epsilon) = \mathit{find2}_{(x)}(\mathit{share}_{(x)}(G, v'), \Lambda, \$\mathtt{x}; \lambda^G(\overline{v})) \\ G'' \approx G'[\delta^{G'}(\epsilon) \leftrightarrow \delta^{G'}(\Lambda, \$\$)][(\Lambda, \$\$) \hookrightarrow \bot] \quad G'' \text{ is } (x)\text{-well-colored} \end{array}}{G \overset{\$\mathtt{x}\overline{[e]} = e'}{\longmapsto_{(x)}} G''}\ [\textsc{Assign}]$$

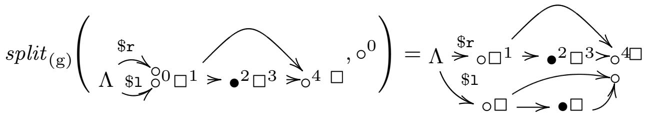
$$\frac{\begin{array}{c} G \vdash \overline{e} \Downarrow \overline{v} \quad (G_1, \epsilon_1) = \mathit{find2}_{(x)}(G, \Lambda, \$\mathtt{x}; \lambda^G(\overline{v})) \quad G_1 \vdash \overline{e}' \Downarrow \overline{v}' \quad (G_2, \epsilon_2) = \mathit{find2}_{(x)}(G_1, \Lambda, \$\mathtt{x}'; \lambda^{G_1}(\overline{v}')) \\ G'' \approx G_2[\epsilon_2 \hookrightarrow \delta^{G_2}(\epsilon_1)] \quad G'' \text{ is } (x)\text{-well-colored} \end{array}}{G \overset{\$\mathtt{x}'\overline{[e']} = \&\, \$\mathtt{x}\overline{[e]}}{\longmapsto_{(x)}} G''}\ [\textsc{AssignRef}]$$

$$\frac{G \vdash \overline{e} \Downarrow \overline{v} \quad (G', \epsilon) = \mathit{find2}'_{(x)}(G, \Lambda, \$\mathtt{x}; \lambda^G(\overline{v})) \quad G'' \approx G'[\epsilon \hookrightarrow \bot] \quad G'' \text{ is } (x)\text{-well-colored}}{G \overset{\mathtt{unset}(\$\mathtt{x}\overline{[e]})}{\longmapsto_{(x)}} G''}\ [\textsc{Unset}]$$

$$\frac{G \vdash e \Downarrow v \quad G'' \approx G \quad G'' \text{ is } (x)\text{-well-colored}}{G \overset{\mathtt{echo}\ e}{\longmapsto_{(x)}} \lambda^G(v), G''}\ [\textsc{Echo}] \qquad \frac{G \overset{s}{\longmapsto_{(x)}} \overline{c}, G'}{\langle G, s; \overline{s}\rangle \overset{\overline{c}}{\Longmapsto_{(x)}} \langle G', \overline{s}\rangle}\ [\textsc{Seq}]$$

Figure 5. Mostly copy-on-write PHP(x)

The set $\tilde{V}$ includes all nodes reachable from $\delta^G(p, -)$ without going through ∘-ports. For example, consider the following graph. When we split this graph at port $\circ^0$, we have $\tilde{V} = \{\square^1, \bullet^2, \square^3, \circ^4\}$. Then the induced subgraph of $\tilde{V}$ is copied and extends the graph at port $\circ^0$, giving the following result.



Let us discuss the assignment statement $G \overset{\$\mathtt{x}\overline{[e]} = e'}{\longrightarrow_{(\mathrm{g})}} G'$ in the mostly copy-on-write model. The definition is given as the rule Assign in Figure 5. The split operation is used here in two places; (1) at *sharing*, when the target is dirty and not sharable, and (2) in the process of *finding* the target node of the write.

- The first step in the copy-on-assignment was the copying of the assigned value. However this time, this occurs only when v' such that $G \vdash e' \Downarrow v'$ is under a dirty port, in which case we split the graph. Otherwise we place a fresh clean copying port on v'. In either case, the root port of the copied value is put in the temporary variable \$\$. The graph after this step is given by $\mathit{share}_{(\mathrm{g})}(G, v')$.
- Similarly to the copy-on-assignment model, we find the following path.

$$\Lambda \xrightarrow{\$\mathtt{x}} p_1 v_1 \xrightarrow{c_1} p_2 v_2 \cdots v_n \xrightarrow{c_n} p_{n+1} (= p)$$

  where $\overline{e}$ evaluates to a sequence of constants $c_1 \cdots c_n$. In this step, defined by $(G', (v_n, c_n)) = \mathit{find2}_{(\mathrm{g})}(\mathit{share}_{(\mathrm{g})}(G, v), \Lambda, \$\mathtt{x}; c_1; ..; c_n)$, any time we encounter a node with multiple ports, we *split* the graph.
- The last redirection step is the same. We switch the incoming edges directed to p to the port at the temporary variable \$\$.

The formal definitions of $\mathit{share}_{(x)}$ and $\mathit{find2}_{(x)}$ are given in Figure 6.

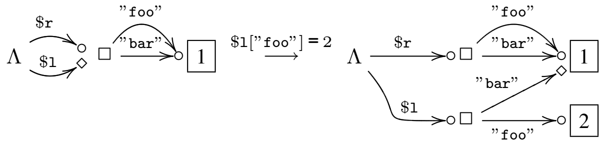
Rewrites for other statements are given similarly. Note that the rule for each statement in Figure 5 contains non-deterministic coloring processes. By saying G is (x)-well-colored, we denote that G is well-colored if $x \in \{\mathrm{d}, \mathrm{g}\}$, and that G is shallowly well-colored, which we define later, if $x = \mathrm{s}$. We write $G'' \approx G'$ if two graphs are isomorphic by ignoring the differences between •-ports and ∘-ports. The definition in Figure 5 does not mention any fixed algorithm, e.g., Algorithm 1, for choosing well-colored G'' as the result of $\overset{s}{\longmapsto}_{(x)}$. Rather, the definition is non-deterministic and chooses arbitrary such G''. The execution should not stop at the coloring process, which is guaranteed by the following lemma.

Lemma 1 (Progress in Coloring)**.** *If G is (x)-well-colored, the rewrite $G \overset{s}{\longmapsto}_{(x)} G'$ never fails in its coloring process.*

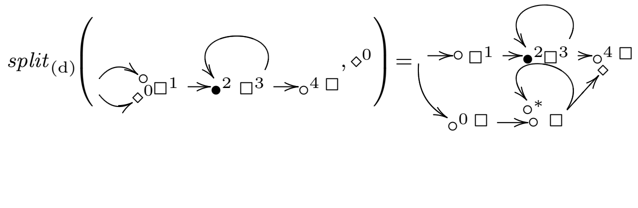
3.6 Other Copy Semantics

We move to mostly copy-on-write models for other two semantics, PHP(d) and PHP(s), by giving minor changes to the model of PHP(g).

Recall that the deep copy semantics distinguishes the copied value at `$1` from the original `$r`, at `$1=$r`. The copied value is treated as a pure value without any references inside. We indicate this pureness by introducing ◇-port. For example,



This rewrite is again implemented by $\mathit{split}_{(\mathrm{d})}(G, p)$ operation. Here one idea to define $\mathit{split}_{(\mathrm{d})}(G, p)$ is to use an unraveling of the subgraph $G|^{v}_{\tilde{V}}$ in the definition of $\mathit{split}_{(\mathrm{g})}(G, p)$. However the problem of this idea, with a graph below for example, is that unraveling of a copied subgraph for $\{\square^1, \bullet^2, \square^3, \circ^4\}$ results in an infinite tree. If we consider a copy-on-write model as an implementation model, such an infinite unraveling is not desirable. Instead, in our scheme, we delay such an unraveling:



Definition 5 (FIND2 AND SHARE OPERATION). *1) Function* $\mathit{find2}_{(x)}(G, v, \overline{c})$ *is defined as follows.*

$$\mathit{find2}_{(x)}(G, v, c) = \begin{cases} \mathit{undefined} & \text{If } \lambda^G(v) \notin \{\mathtt{null}, \Box\} \\ (G, (v, c)) & \text{If } G \vdash v \xrightarrow{c} pv' \\ (\hat{G}, (v, c)) & \text{Otherwise (*)} \end{cases}$$

$$\mathit{find2}_{(x)}(G, v, c; \overline{c}) = \begin{cases} \mathit{undefined} & \text{If } \lambda^G(v) \notin \{\mathtt{null}, \Box\} \\ \mathit{find2}_{(x)}(G, v', \overline{c}) & \\ \quad \text{If } G \vdash v \xrightarrow{c} pv',\ \lambda^G(p) \neq \diamond \text{ and } |in_G(v')| = 1 & \\ \mathit{find2}_{(x)}(G', \delta^{G'}(p, -), \overline{c}) & \\ \quad \text{If } G \vdash v \xrightarrow{c} pv' \text{ and } G' = \mathit{split}_{(x)}(G, p) & \\ \mathit{find2}_{(x)}(\hat{G}, \hat{v}, \overline{c}) & \text{Otherwise (*)} \end{cases}$$

where $\hat{G} = (V^G \uplus \{\hat{p}, \hat{v} \mid \hat{p}, \hat{v} \text{ are fresh}\}, \Lambda^G, \lambda^G[\hat{p} \mapsto \circ, \hat{v} \mapsto \mathtt{null}, v \mapsto \Box], \delta^G[(v, c) \mapsto \hat{p}, (\hat{p}, -) \mapsto \hat{v}])$.

2) $\mathit{find2}'_{(x)}(G, v, \overline{c})$ *is defined similarly except that it is undefined for the cases (*).*

3) We define $\mathit{share}_{(x)}(G, v)$ *as follows.*

$$\mathit{share}_{(x)}(G, v) = \begin{cases} G \cup \{\Lambda \xrightarrow{\$\$} \hat{p}v\} & \text{If } v \text{ has } \circ\text{- or } \diamond\text{-port} \\ \mathit{split}_{(x)}(G \cup \{\Lambda \xrightarrow{\$\$} \hat{p}v\}, \hat{p}) & \text{If } v \text{ has } \bullet\text{-port} \end{cases}$$

where $\hat{p}$ *is a fresh* $\circ$*-port if* $x = \mathrm{g}$*, and is* $\diamond$*-port if* $x = \mathrm{d}$*. In case of* $x = \mathrm{s}$*, it is* $\circ$*-port or* $\bullet$*-port depending on the color of* v*'s ports.*

Figure 6. Definition of $\mathit{find2}_{(x)}$ and $\mathit{share}_{(x)}$

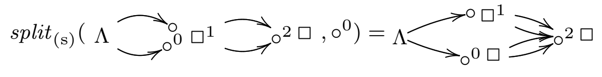
To delay unraveling, we supply sufficient ports, e.g., the port $\circ^*$ above, corresponding to each edge in the copied graph. We define $\mathit{split}_{(\mathrm{d})}(G, p)$ as follows.

$$\mathit{split}_{(\mathrm{d})}(G, p) = \begin{cases} \mathit{split}_{(\mathrm{g})}(G, p) & \text{if } \lambda^G(p) \neq \diamond \\ (G[(p, -) \hookrightarrow \tilde{G}])[p \mapsto \circ] & \text{otherwise} \end{cases}$$

Let $\tilde{V}$ be the set appearing in the definition of $\mathit{split}_{(\mathrm{g})}(G, p)$, from which the above $\tilde{G}$ is defined as follows.

- The node set $V^{\tilde{G}}$ contains all non-port nodes in $\tilde{V}$ and a set of ports p^ϵ each having one-to-one correspondence to an edge ϵ in $G|_{\tilde{V}}$.
- For each $p^{(v,c)} \in V^{\tilde{G}}$ and $v' = \delta^G(v, c; -)$, we have an edge $v \xrightarrow{c} p^{(v,c)}v'$.
- Each port $p^{(v,c)}$ is $\diamond$-port if $\delta^G(v, c; -) \notin \tilde{V}$, and is $\circ$-port otherwise.

Let us move to the shallow copy semantics PHP(s). In this case, $\mathit{split}_{(\mathrm{s})}$ does not duplicate ports containing multiple incoming edges.



This is explained from the set $\tilde{V}$ for defining $\mathit{split}_{(\mathrm{s})}$ which is this time $\{\Box^1\}$, and does not include $\circ^2$. Here is a definition of $\mathit{split}_{(\mathrm{s})}(G, p)$.

$$\mathit{split}_{(\mathrm{s})}(G, p) = G[(p, -) \hookrightarrow G|_{\tilde{V}}^{\delta^G(p,-)}]$$

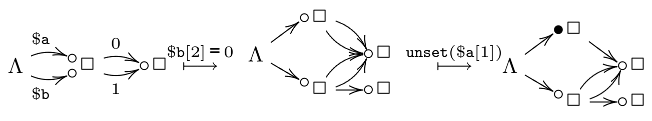
Here $\tilde{V}$ is the minimal set satisfying

- $\delta^G(p, -) \in \tilde{V}$,
- $\delta^G(w, c) \in \tilde{V}$ if $w \in \tilde{V}$ where $|in_G(\delta^G(w, c))| = 1$, and
- $\delta^G(p', -) \in \tilde{V}$ if $p' \in \tilde{V}$ such that p' is marked dirty.

The remainder of the mostly copy-on-write semantics of both PHP(d) and PHP(s) are the same as PHP(g), except that we need a different well-coloredness invariant in PHP(s) case.

3.7 Shallow Well-Coloredness

The original well-coloredness invariant does not fit well for PHP(s). This is due to the difficulty to guarantee the progress property at coloring, i.e., Lemma 1. For instance, some rewrites can break the well-coloredness in a manner that it cannot easily be fixed.



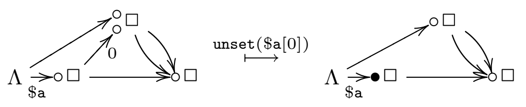
The above graph after the split caused by $\$\mathtt{b}[2] = 0$ is not well-colored in the original sense. It is nor easy to turn this graph well-colored, since it requires marking of the port at $\$\mathtt{a}$. As an alternative solution, we may accept this graph by using a different invariant in the case of PHP(s). We call the graph after $\$\mathtt{b}[2] = 0$ *shallowly well-colored*. By carefully examining this case, we can observe that actual problems do not occur immediately, but rather they occur several steps later, e.g., after we unset $\$\mathtt{a}[1]$, share the node at $\$\mathtt{a}$, and then we unset $\$\mathtt{b}$. During these steps, we have a second chance to mark the port at $\$\mathtt{a}$, e.g., at $\mathtt{unset}(\$\mathtt{a}[1])$.

We have devised a new definition of the shallow well-coloredness as follows. We write $cl(v)$ to denote a set of nodes reachable from v without going through reference ports. We denote by p's reference count from V, a number of edges $(v, c) \in in_G(p)$ such that $v \in V$.

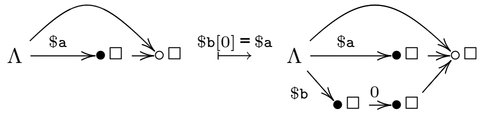
Definition 6. *1) We call a node* v *shallowly dirty iff there is a reference port* p *such that* p*'s reference count from* $cl(v)$ *is equal to 1.*
2) A graph is shallowly well-colored if i) ports on shallowly dirty nodes are marked dirty, and ii) copying ports are marked clean.

Let us discuss the progress property in PHP(s). In PHP(g) and PHP(d), at least for identifying false negatives for the safety, we need the coloring process only after the reference assignment $\$\mathtt{x}'\overline{[e']}$ `=&` $\$\mathtt{x}\overline{[e]}$. On the other hand, PHP(s) is not so simple, in which any statements rewriting the graph may introduce false negatives that should be immediately fixed after rewrites. We can classify the sources of false negatives in PHP(s) into the following three categories.

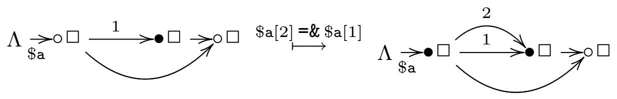
- The first source comes from rewrites like `unset` as shown in the following example. Such false negatives may also result from `=` and `=&`, if the rewrites erase existing edges in the graph.



- The second source is the rewrite by $\$\mathtt{x}\overline{[e]} = e'$. Note that the definition of *share* for PHP(s) says that the port at \$\$ becomes dirty if the port at rvalue e' is dirty. If so, some ports on the path to $\$\mathtt{x}\overline{[e]}$ may become false negative.



- The third source of false negatives comes at reference assignment. The coloring algorithm here is similar to that for PHP(g) and PHP(d), but actually it is more conservative. Look at the following example.



Note that the new coloring invariant for PHP(s) itself is neither weaker nor stronger than the original invariant. For example, the port at `$a` in the third example should be marked only in PHP(s).

For PHP(s), the following algorithm is at least sufficient for fixing the graph after the rewrites.

Algorithm 2 (Marking False-Negatives in PHP(s))**.** *1) The rewrite at* Unset *in Figure 5 removes an existing edge ϵ. In PHP(s), if there is a reference port reached from ϵ, we mark all ports encountered during the "find" step before the rewrite.*
2) The "redirection" step at Assign *replaces the content at the target port with the content of* $$. *If either i) the target port of the redirection is non-reference, but from which we reach a certain reference port, or ii) the port at* $$ *is dirty, we mark any port encountered during the "find" step dirty.*
3) Similarly to Algorithm 1, at AssignRef*, we mark ports found during the "find" steps. The difference is that we cannot skip ports $p_1, \ldots, p_I$ and $q_1, \ldots, q_J$, if either i) there is a reference port reachable from the node under $q(= q_h)$, ii) a reference port is reached from the node under $p(= p_k)$, or iii) one of $p_{I+1}, \ldots, p_{k-1}$ and $q_{J+1}, \ldots, q_{h-1}$ is a reference port.*

Remark 2. *Let us discuss the correctness of Algorithm 2. We say a path is split if it does not go through copying ports.*

Case 1). Assume that a node v is about to turn shallowly dirty after the removal of ϵ, which would drop the reference count of a certain p from $cl(v)$. This means that there is a path from v to p, which goes through ϵ but not any reference port. Now note that the "find" step in Unset *guarantees the existence of a split path from the root to ϵ, which implies that any node which can reach ϵ without passing reference ports dominates ϵ. Hence v dominates ϵ, so that v is on the path found in the "find" step.*

A similar discussion applies to cases 2.i) and 3.i). The case 2.ii) is similar to Remark 1. Cases 3.ii) and 3.iii) come from the fact that a reference introduced by the reference assignment may remove some nodes from $cl(v)$.

Combining Remark 1 and 2, we obtain the first half of the proof of Lemma 1. Note that in any semantics PHP(x), the last redirection steps of the rewrites introduce false negatives only on non-copying ports, so that we can safely fix the graph by marking such ports dirty. The detail is omitted, but we can complete the proof of our progress lemma by showing that other rewrite steps, i.e., sharing and splitting steps, never break well-coloredness or shallow well-coloredness.

4. Correctness of Mostly Copy-On-Write Scheme

The mostly-copy-on-write scheme is correct with respect to the copy-on-assignment semantics. In this section, we use a bisimulation to show this result.

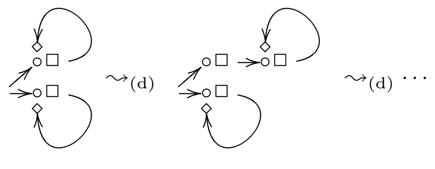
4.1 Equivalence by Bisimulation

If two systems are bisimilar, we cannot distinguish between the two from their behaviors, which in our case, are observed from the output of `echo` statements. Let $\hat{G}^0$ and G^0 be initial copy-on-assignment and copy-on-write graphs, including only constants but no variables.

Theorem 1 (Equivalence of Copy-on-Assignment and Mostly Copy-on-Write)**.** *Let $\overline{s}$ be a PHP program. Two states $\langle \hat{G}^0, \overline{s}\rangle$ and $\langle G^0, \overline{s}\rangle$, in the copy-on-assignment and mostly copy-on-write semantics $\overset{\overline{c}}{\Longmapsto}_{(x)}$, respectively, are bisimilar to each other.*

4.2 Domain-Theoretic Approach

Intuitively speaking, the bisimilarity relation should be obtained by *sufficiently* splitting the given copy-on-write graph, which should give the *canonical* form that has the same shape as the copy-on-assignment graph. However formalizing this intuition requires care. For example, for the following graph, we may split the upper cycle forever without ever reaching the desired canonical form.



To formally define the canonical form, we use a domain-theoretic approach, in which the *sufficient* splitting means to take the *lub* of arbitrary possible results of splitting.

We say a graph G is *regular* if $G' \preceq G$ for some finite graph G'. We use $\mathbb{G}$ to denote the set of all regular graphs, for which the following result is known.

Theorem 2 (Regular Graphs Form Cpo)**.** *(Barendsen and Smetsers 1992) $(\mathbb{G}, \preceq)$ is an unpointed-cpo.*

In other words, any directed subset $D \subseteq \mathbb{G}$ of regular graphs has a lub $\bigsqcup D$ in $(\mathbb{G}, \preceq)$.

In the following, we mean by copying ports either ports on nodes with multiple ports or $\diamond$-ports. We say a path in the copy-on-write graph is *split* if it does not go through copying ports. A port or node n, or an edge ϵ, is *split* if there is a split path from Λ to n, or ϵ. Now we define the binary relation $\rightsquigarrow_{(x)}$ of copy-on-write graphs as follows.

Definition 7 (Definition of $\rightsquigarrow_{(x)}$)**.** *1) A set of (x)-well-colored graphs is denoted by $\mathbb{W}_{(x)}$. 2) A set of weakly well-colored graphs, denoted by $\mathbb{W}^0_{(x)}$, is the set of copy-on-write graphs such that dirty (or shallowly dirty if $x =$ s) nodes have neither non-split clean ports nor copying ports. 3) Let $G, G' \in \mathbb{W}^0_{(x)}$, we write $G \rightsquigarrow_{(x)} G'$, if there is a copying port $p \in V^G$ such that i) p is split, and ii) $G' = split_{(x)}(G, p)$.*

Proposition 1 (Property of $\rightsquigarrow_{(x)}$)**.** *We write $erase(G)$ for a graph obtained by erasing all ports in G. (i) $G \rightsquigarrow_{(x)} G'$ implies $erase(G) \preceq erase(G')$. (ii) $\rightsquigarrow_{(x)}$ is confluent.*

With $\rightsquigarrow_{(x)}$, the canonical form of the copy-on-write graph, $cano_{(x)}(G)$, is defined as follows.

$$cano_{(x)}(G) = \bigsqcup_{G \rightsquigarrow^*_{(x)} G'} erase(G')$$

The bisimulation relation for Theorem 1 relates a copy-on-write graph G and copy-on-assignment graph $\hat{G}$ iff i) $G \in \mathbb{W}_{(x)}$, and ii) $cano_{(x)}(G) \simeq \hat{G}$.

Proposition 2. (Continuity Of Redirection) *Let D be a directed set on $(\mathbb{G}, \preceq)$. Let $\overline{C}$ be a set of paths and $\overline{c}$ be a path such that for each $\hat{G} \in D$, i) $\mathcal{E}^{\hat{G}} = \{(\delta^{\hat{G}}(\Lambda, \overline{c}'), c'') \mid \overline{c}'; c'' \in \overline{C}\}$ is a set of edges to a single node, and ii) $v^{\hat{G}} = \delta^{\hat{G}}(\Lambda, \overline{c}) \in V^{\hat{G}}$.*

$$\bigsqcup_{\hat{G} \in D} \hat{G}[\mathcal{E}^{\hat{G}} \hookrightarrow v^{\hat{G}}] \simeq (\bigsqcup D)[\mathcal{E}^{\bigsqcup D} \hookrightarrow v^{\bigsqcup D}].$$

Lemma 2. (REDIRECTION AND SPLIT COMMUTE) *Let $G \in \mathbb{W}^0{}_{(x)}$. Let p, q, r be ports, and ϵ (= (v, ℓ)) be an edge such that p, q, r, and v are split. We also assume that r is a copying port.*

$$split_{(x)}(G, r)[\epsilon \hookrightarrow \bot] \simeq split_{(x)}(G[\epsilon \hookrightarrow \bot], r)$$
$$split_{(x)}(G, r)[\epsilon \hookrightarrow p] \simeq split_{(x)}(G[\epsilon \hookrightarrow p], r)$$
$$split_{(x)}(G, r)[q \hookleftarrow p] \simeq split_{(x)}(G[q \hookleftarrow p], r)$$

Now by combining the results obtained so far, we can show the following lemma, which states that each rewrite step between two models preserves the bisimulation relation. In particular, Lemma 3(ii) is a direct consequence of Proposition 2 and Lemma 2.

Notation 1. *1) For $n \in V^G$, we write $[n]_G$ for a set of paths to reach n, i.e., $[n]_G = \{\overline{c} \mid G \vdash \Lambda \xrightarrow{\overline{c}} n\}$. 2) Given a path $\overline{c}$ on copy-on-write graphs G, a path $\overline{c}^\star$ on $cano_{(x)}(G)$ is obtained by removing all occurrences of $-$ from $\overline{c}$. 3) Given a split node $n \in G$, we write $n^\star \in V^{cano_{(x)}(G)}$ for a node such that $[n]^\star_G = [n^\star]_{cano_{(x)}(G)}$, and $\epsilon^\star$ for an edge $(v^\star, c)$ such that $\epsilon = (v, c)$.*

Lemma 3. (EACH STEP PRESERVES BISIMULATION) *Let G be a copy-on-write graph in $\mathbb{W}^0_{(x)}$.*
(i) Let $\hat{G} \simeq cano_{(x)}(G)$ and $v \in V^G$ be a split node. 1) $(G', (w, c)) = find2_{(x)}(G, v, \overline{c})$ is defined iff $(\hat{G}', (\hat{w}, \hat{c})) = find(\hat{G}, v^\star, \overline{c})$ is defined. If both are defined, 2) $\hat{G}' \simeq cano_{(x)}(G')$, 3) $[\hat{w}]_{\hat{G}'} = [w]^\star_{G'}$ and $\hat{c} = c$, and 4) w is split in G'.
(ii) $cano_{(x)}(G[\epsilon \hookrightarrow p]) \simeq cano_{(x)}(G)[\epsilon^\star \hookrightarrow p^\star]$, if ϵ and p are split. Similarly, for $[\epsilon \hookrightarrow \bot]$ and $[q \leftarrow p]$ where ϵ, p and q are split.
(iii) $cano_{(x)}(share_{(x)}(G, v)) \simeq copy_{(x)}(cano_{(x)}(G), \hat{v})$, if $G \in \mathbb{W}_{(x)}$ and $[\hat{v}]_{cano_{(x)}(G)} \subseteq [v]_G$.
(iv) $cano_{(x)}(G) \simeq cano_{(x)}(G')$ if $G \approx G'$ and $G' \in \mathbb{W}^0_{(x)}$.

Now Theorem 1 is a straightforward consequence of Lemma 1 and Lemma 3.

5. Experimental Implementation

We implemented the mostly copy-on-write scheme on top of the PHP Runtime, called P9, which is a native compiler-based runtime currently developed in IBM Tokyo Research Lab. In the actual implementation, we need some extensions to the model defined in Section 3.

- Our graph model has a single root, which represents a universal variable table. In reality, PHP has multiple variable tables for global variables, local variables, and so on. Moreover, P9 sometimes performs an optimization to decompose a variable table to separate variables on local frame. In such a case, each single variable becomes a root of the graph. Similarly, PHP also has *objects* which are assigned by pointers[9], rather than by copying. Each object also corresponds to a root of the graph whose successors are field members. In the implementation we extended our scheme to deal with such multi-rooted graphs.
- We have not discussed so far language constructs other than `=&` in PHP for creating references. For example, PHP supports pass-by-reference and return-by-reference mechanisms at function calls. Such mechanisms can be implemented by adaptation of `=&`. PHP also has an array initializer, i.e., `array` expression, which can directly create an array value containing references. In our implementation, arrays created by `array` expression simply inherit the dirtiness information of their elements.

[9] P9 is an implementation of PHP5. Historically, PHP4 objects are treated by values, and this has changed during the move to PHP5.

	array	array-ref	specweb
PHP(g) coa	62.8	95.2	1624.2
PHP(g) m-cow	38.3	95.0	1482.7
PHP(d) m-cow	38.5	95.0	1479.6
näive cow	38.5	39.3	1431.2

Table 1. Benchmark results (elapsed time μsec)

- Recall the explanation of $find2_{(x)}()$ during which we split any node encountered if it has multiple ports. We use approximate reference counting to implement this function. It is easy to see that for PHP(d) and PHP(g), we can perform splits according to such over-approximated reference counts. On the other hand, we do not currently have a faithful implementation of PHP(s). This is because as argued earlier in Section 2, such an implementation requires the exact reference counting.

Table 1 summarizes the preliminary benchmark results of our implementation. In addition to mostly-copy-on-write implementations of PHP(d) and PHP(g), we also prepared copy-on-assignment PHP(g), and a naïve copy-on-write of the shallow copy close to the Zend runtime. In Table 1, the results of first two columns use microbenchmark scripts, which repeat array assignments for clean and dirty arrays, respectively. The third one is a script taken from the SPECweb2005 benchmark[10] whose result is measured using an actual web-server configuration. The behaviors of all scripts are not affected by the difference in copy semantics. The measurements were made on Windows XP, Intel Core2 2.0GHz for array and array-ref, and Linux Pentium (R) 3.4 GHz for specweb.

The use of copy-on-assignment degrades the performance by 13.5 % on specweb. In contrast, the mostly copy-on-write implementations reduce the performance degrade for specweb to around 3%, while guaranteeing the equivalence to the copy-on-assignment semantics.

6. Related Work

PHP arrays are close to functional arrays, whose optimization is a recurring problem in the study of functional languages. Indeed, copy-on-write techniques for functional arrays were already proposed for IFP (Robison 1987) and SETL (Schwartz 1975) in early days. More recently, after the dynamic reference counting becomes less popular as a memory management scheme, optimization techniques using the static approximation of reference counts are proposed (Hudak and Bloss 1985; Bloss 1989; Odersky 1991; Wadler 1990; Turner et al. 1995; Hofmann 2000; Shankar 2001). Such work tries to determine when we can perform *in-place* update on functional data structures. Regarding efficient runtime techniques for functional arrays, alternatives to copy-on-write arrays, called *version tree arrays* or *trailer*, are proposed (Baker 1978; Aasa et al. 1988; O'Neill and Burton 1997; Conchon and Filliâtre 2007).

Term graph rewriting systems (TGRS) (Barendregt et al. 1987) extend the term rewriting system to graph rewriting. Our graph rewriting semantics and proofs are largely inspired by the work on TGRS. Barendsen and Smetsers (Barendsen and Smetsers 1992) discussed the *lazy copying* in this setting, whose idea is close to copy-on-write.

Speaking in a broader context, formal semantics approaches have been identifying and solving intricate problems in real-life programming languages, e.g., co-/contra-variance problems in Eiffel, C++, Java, and other object oriented languages (Castagna 1995; Igarashi and Viroli 2002), or dynamic loading problems in Java

[10] `http://www.spec.org/web2005/`. We used bank/transfer.php.

(Qian et al. 2000), etc. Our work can be considered as one on this line of researches.

7. Conclusion

We have discussed the semantics of PHP focusing on the copy semantics for arrays. We have also presented the efficient mostly copy-on-write implementation faithful to the semantics. Our future work directions include:

- Exploring the implementation of PHP without costly reference counting. For this, we may combine our mostly copy-on-write technique with existing work on functional arrays and in-place update problems.
- Developing compiler optimization techniques for PHP's arrays and references, including deforestation, escape analysis, etc., based on the semantics defined in this paper.

8. Acknowledgment

We would like to thank Rob Nicholson for early discussions on PHP arrays' problems. Minamide's work was partly supported by Grant-in-Aid for Scientific Research 20300001 and 18700018, and CREST of JST (Japan Science and Technology Agency).

References

A. Aasa, Sören Holmström, and Christina Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28(3): 490–503, 1988.

Henry G. Baker. Shallow binding in LISP 1.5. *Communications of the ACM*, 21(7):565–569, 1978.

H. P. Barendregt, M. C. J. D. Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph reduction. In *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*, pages 141–158, 1987.

Erik Barendsen and Sjaak Smetsers. Graph rewriting and copying. Technical Report 92-20, University of Nijmegen, 1992.

Adrienne Bloss. Update analysis and the efficient implementation of functional aggregates. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 26–38, New York, NY, USA, 1989. ACM.

Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Trans. Program. Lang. Syst.*, 17(3):431–447, 1995.

Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In *ML '07: Proceedings of the 2007 workshop on Workshop on ML*, pages 37–46, New York, NY, USA, 2007. ACM.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic J. of Computing*, 7(4):258–289, 2000.

Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 300–314, New York, NY, USA, 1985. ACM.

Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 441–469, London, UK, 2002. Springer-Verlag.

Yasuhiko Minamide. Static approximation of dynamically generated Web pages. In *Proceedings of the 14th International World Wide Web Conference*, pages 432–441. ACM Press, 2005.

Martin Odersky. How to make destructive updates less destructive. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–36, 1991.

Melissa E. O'Neill and F. Warren Burton. A new method for functional arrays. *Journal of Functional Programming*, 7(5):487–514, September 1997.

Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. In *Proc. 15th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, volume 35 of *ACM SIGPLAN Notices*, pages 325–336, October 2000.

A. D. Robison. The Illinois functional programming interpreter. In *SIGPLAN '87: Papers of the Symposium on Interpreters and interpretive techniques*, pages 64–73, 1987.

Jacob T. Schwartz. Optimization of very high level languages, parts I, II. *Comput. Lang.*, 1(2-3):161–218, 1975.

Natarajan Shankar. Static analysis for safe destructive updates in a functional language. In *Proc. of LOPSTER 2001, 11th International Workshop on Logic Based Program Synthesis and Transformation, Paphos, Cyprus, November 28-30, 2001*, LNCS 2372, pages 1–24, 2001.

David N Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Functional Programming Languages and Computer Architecture*, San Diego, California, 1995.

P. Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359, 1990.