Ultra Fast Change Detection

Status: Draft

Authors: Misko Hevery

This document is published to the web as part of the public Angular Design Docs folder

Objective

Describe the issues with the current Record based Change Detection and propose a new faster and smaller (memory) change detection.

Background

Current Record based change detection suffers from:

1. High algorithmic complexity. It is hard to understand and maintain.
2. Moderate memory pressure
3. Presence of megamorphic call which dominates/limits the maximum speed.
4. Difficulty in understanding the stack traces.
5. Limiting the tree shaking capability of minifiers due to high level of dynamism.

This proposal alleviates all of the above issues by making the change detection more static.

Prior Art

- Current record based change detection: design doc

Detailed Design

Let's assume the following template:

```
class MyController {
  user:User;
  items:List;
}
```

```
Hello {{user.name}}!
<ul [hidden]="items.length == 0">
  <li template="ng-repeat: #item in items">{{item}}</li>
</ul>
```

The above example would generate at a minimum 5 Records (user, name, items, length, ==) for the base template and additional 1 record for each item in the ng-repeat. Each record has a cost of about 16 words (4 byets to work). The total memory consumption is about 80 words plus 5 words per row.

A better way to process the changes is to generate a ChangeDetection class for each View.

```
// Memory Cost: 8 WORDS;
class MyController_ChangeDetector {
  dispatcher:Dispatcher;
  context:MyController;
  // Store last values here
  _user;
  _user_name;
  _items;
  _items_lenth;
```

```
_items_length_EQEQ_0;
children = [];

constructor(dispatcher:Dispatcher) {
  this.dispatcher = dispatcher;
}

attach(context:MyController) {
  this.context = context;
  _user = INITIAL;
  _user_name = INITIAL;
  _items = INITIAL;
  _items_lenth = INITIAL;
  _items_length_EQEQ_0 = INITIAL;
}

detectChanges() {
  var _user;
  var _user_name;
  var _items;
  var _items_lenth;
  var _items_length_EQEQ_0;

  if (this._user !== (_user = context.user)) {
    this._user = securityCheck(_user);
  }
  if (this._user_name !== (_user_name = _user.name)) {
    this._user_name = securityCheck(_user_name);
    this.dispatcher.notify(USER_NAME_ID, _user_name);
  }
  if (this._items !== (_items = context.items)) {
    this._items = securityCheck(_items);
  }
```

```
    if (this._items_length !== (_items_length = _items.length)) {
      this._items_length = securityCheck(_items_length);
    }
    if (this._items_length_EQEQ_0 !== (_items_length_EQEQ_0 = _items_length == 0)) {
      this._items_length_EQEQ_0 = securityCheck(_items_length_EQEQ_0);
      this.dispatcher.notify(ITEM_LENGETH_EQEQ_0_ID, _items_length_EQEQ_0);
    }
    // guard for disabling change detection on hidden bindings
    if (_items_length_EQEQ_0) {
      var _itemsChanges = compareCollection(_items, this._items);
      if (_itemsChanges != null) {
        this.dispatcher.notify(ITEMS_ID, _itemsChanges);
      }

      // now process the child view change detections
      var children = this.guarded_items_length_EQEQ_0_children;
      for(var i = 0; i < children.length; i++) {
        children[i].detectChanges();;
      }
    }
  }
}


// Memory Cost: 3 WORDS;
class MyController_sub_1_ChangeDetector {
  dispatcher:Dispatcher;
  context:MyController;
  // Store last values here
  _item;

  constructor(dispatcher:Dispatcher) {
    this.dispatcher = dispatcher;
```

```
  }

  attach(context:MyController) {
    this.context = context;
    _item = INITIAL;
  }

  detectChanges() {
    var _item;

    if (this._item !== (_item = context.item)) {
      this._item = securityCheck(_item);
      this.dispatcher.notify(ITEM_ID, _item);
    }
  }
}
```

The above code could trivially be generated from the parsed AST of expressions which already exists in Angular2. The benefits are:

- More memory efficient storage
- No megamorphic calls, which means that VM can inline cache field reads for faster execution
- Can be fully generated at compile time for Dart and CSP environments.
- Allows tree shaker to properly analyze and throw away unused code.

Caveats

You may need to describe what you did not do or why simpler approaches don't work. Mention other things to watch out for (if

any).

Security Considerations

How you'll be secure

Performance Considerations / Test Strategy

How you'll be fast.

Work Breakdown

Description of development phases and approximate time estimates.