Mashic: Automated Mashup Sandboxing based on Inter-frame Communication (long version)

Zhengqin Luo
INRIA Sophia Antipolis

Tamara Rezk
INRIA Sophia Antipolis

July 12, 2011

Abstract

Mashups are a prevailing kind of web applications integrating external gadget APIs often written in the Javascript programming language. Writing secure mashups is a challenging task due to the heterogeneity of existing gadget APIs, the priviledges granted to gadgets during mashup executions, and the Javascript's highly dynamic environment.

We propose a new compiler, called Mashic, for the automatic generation of secure Javascript-based mashups from existing mashup code. The Mashic compiler can effortlessly be applied to existing mashups based on a wide-range of gadget APIs. It offers security and correctness guarantees. Security is achieved by using the Same Origin Policy. Correctness is ensured in the presence of benign gadgets, that satisfy confidentiality and integrity constrains with regard to the integrator code. The compiler has been successfully applied to real world mashups based on Google maps, Bing maps, YouTube and Zwibbler APIs.

Contents

1 Introduction 2

2 Running Example 4

3 Mashic Compilation 6

4 Decorated Semantics 10
4.1 Heaps and Objects . . . 10
4.2 Syntax . . . 13
4.3 Configurations and Transition Relation . . . 15
4.4 DOM Semantics Rules . . . 16
4.5 Core Semantics Rules . . . 16

5 Correctness Theorem 22
5.1 Assumptions . . . 22
5.2 Indistinguishability and Correctness . . . 24
5.3 Auxiliary Definitions and Lemmas . . . 27
5.4 Proofs . . . 28

6 Security Theorem 46
6.1 Proofs . . . 48

7 Implementation and Case Studies 51

8 Conclusion 52

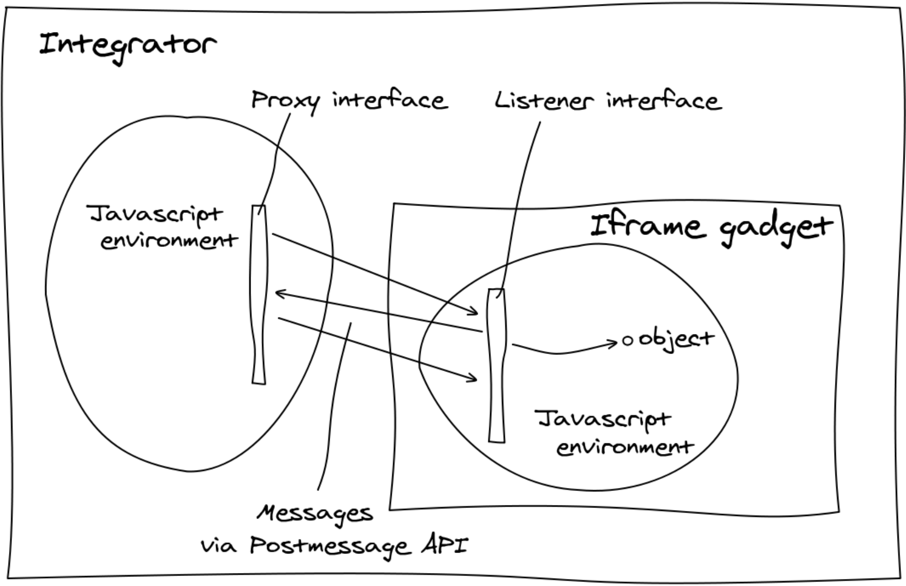


Figure 1: Target Architecture Automatically Generated by Mashic

A Mashic Run-time 55

1 Introduction

Mixing existing online libraries and data into new online applications in a rapid, inexpensive manner, often referred to as mashups, has captured the way of designing web applications. ProgrammableWeb mashup graphs currently report that over 5000 mashup-based web applications and over 3000 gadget APIs currently exist (`http://www.programmableweb.com/`). Since the release of the first major example, `HousingMaps.com` in early 2005, mashups have offered the potential to finally make widespread software reuse a reality.

In a mashup the *integrator* code integrates *gadget* from external code providers, permitting to execute attacker code with otherwise impossible integrator privileges due to browsers security policies. Typically, code is written in the JavaScript (JS) language, the most popular scripting language on the Internet nowadays. JS programs execute on the browser as embedded script nodes in the document object model (DOM) [Hors et al., 2000]. The highly dynamic nature of the language, that is usually combined with the DOM API [Gardner et al., 2008], has led to a prevalence of security vulnerabilities associated with leakage of pointers to sensitive resources in the DOM such as the property "window" of the global object, or rewriting of object properties changing functionality of native functions.

Browsers implement the Same Origin Policy (SOP). An origin is characterized by the protocol, the domain name, and the port used for communication. The SOP does not allow sharing of the global object between different origins in order to isolate untrusted JS code. Two ways of including untrusted JS code are available in the browsers: either including it with the script tag and granting access to all the resources of the integrator, or limiting it within a single frame of a different origin (with the iframe tag) and granting no access to any resource, by the SOP. (For ease of presentation, when we mention frame in the rest of the paper, we mean "frame with a different origin".)

Mashup programmers are thus challenged to provide flexible functionality even if the code consumer is not willing to trust the gadgets that mashups utilize. Often, programmers choose to include gadgets using the script tag and resign security in the name of functionality.

Recently AdJail [Louw et al., 2010] and Postmash [Barth et al., 2009a] propose to use inter-frame communication between integrator and gadgets. AdJail focuses on advertisement scripts by delegating limited DOM interfaces from the integrator. PostMash [Barth et al., 2009a] targets more general inter-

faces to operate on gadgets and propose a promising architecture for mashups depicted in Figure 1. In the PostMash design a programmer must write stub libraries on both the integrator and the gadget. On the integrator side, the stub library must provide an interface similar to the original gadget's interface. The stubbed interface sends corresponding messages by means of the PostMessage API in HTML5. On the gadget side, there is another stub library, listening and decoding incoming messages. Postmash design presents several issues:

- Lack of generality: each gadget requires different interfaces;
- Programmer intervention: the programmer manually writes interfaces for different gadgets. Moreover, the programmer needs to write integrator's code in CPS (Continuation Passing Style) [Danvy and Filinski, 1992] to adapt to the asyncronous nature of PostMessage.

These issues compromise the guarantees offered by the architecture. Crucially, reliability of the mashup depends on the programmer without any enforced guarantees.

We address these issues with a novel compiler called Mashic which inputs existing mashup code, JS code integrated to HTML, to generate reliable mashups using gadget isolation as shown in Figure 1. Automated code transformation with clear correctness and security guarantees can support programmers in dealing with the challenge of writting secure mashups without resigning functionalities. The compiler offers the following features:

Automation and generality: Inter-frame communication and sandboxing code is fully generated by the compiler and can be used with any untrusted gadget without rewriting the gadget's code. Automation is achieved via a proxy and a listener interface (Figure 1), that provide general encoding and operations on gadget objects that are not directly reached by the integrator when the SOP applies. Due to the asynchronous nature of the PostMessage API, integrator's code that initially directly interacts with the gadget is transformed into CPS style in order to interact with the proxy.

Correctness guarantees: We prove a correctness theorem that states that the behavior of the Mashic compiled code is equivalent to the original mashup behavior under the hypothesis that the gadget is *benign*. Precisely defining a benign gadget turns out to be a technical challenge in itself. For that, we instrument the JS semantics extended with HTML constructs by a a generalization of *colored brackets* [Grossman et al., 2000] and resort to equivalences used in information flow security [Sabelfeld and Myers, 2003].

Security guarantees: We prove a security theorem that guarantees integrity and confidentiality for the compiled mashup. These guarantees are essential for the success of the compiler since the programmer can rely on security of compiled mashups using untrusted gadgets without further hypotheses. Indeed, if the gadget is not benign in the original mashup, malicious behavior is neutralized in the compiled mashup. This proof relies on the browsers' SOP, that we formalize by means of iframe DOM elements.

The proposed compiler is directly applicable to real world and widespread mashups. We present evidence that our compiler is effective. We have implemented the compiler in a dialect of Scheme and JS. We have compiled several mashups based on Google and Bing maps, YouTube, and Zwibbler APIs. In summary our contributions are:

1. The Mashic compiler, its design and implementation, that applies to existing mashups that include untrusted gadgets using the script tag.
2. Strong security and correctness guarantees for Mashic compiled code and hence direct guarantees for the mashup end consumer.
3. Case studies based on existing widespread mashups that demonstrate the effectiveness of the compiler.

Related Work: The closest related works to Mashic are AdJail [Louw et al., 2010] and Postmash [Barth et al., 2009a], and are described above. We focus now in other related work.

Jang et al. [2010] study on top of 50000 websites privacy violating information flows in JS based web applications. Their survey shows that top-100 sites present vulnerabilities related to cookie stealing, location hijacking, history sniffing and behavior tracking. Browser implementation vulnerabilities have also been shown to leak JS capabilities between different origins [Barth et al., 2009c]. Many mechanisms to prevent JS based attacks have been deployed. For example the Facebook JS subset (FBJS) [Inc., 2011a] was intended to prevent user-written gadgets to attack trusted code but it did not really succeed in its goals [Maffeis and Taly, 2009]. Google Caja [Inc., 2011b] is similar to FBJS, transforming JS programs to

insert run-time checks to prevent malicious access. Yahoo ADsafe [Crockford, 2011] statically validates JS programs. Maffeis et al. [2009] resort to language-based techniques to find out a subset of JS that can be used to prove an isolation property for JS code. Static analysis is usually not applicable or not sound for large and real world web applications due to the highly dynamic nature of JS programs and because gadgets in general can not be restricted to subsets of JS. As a response to the increasing need to get flexible functionality without resigning to security guarantees, the research community has proposed several communication abstractions [Wang et al., 2007, Crockford, 2010, Jackson and Wang, 2007, Keukelaere et al., 2008]. Specifically, OMASH [Crites et al., 2008] proposes a refined the SOP to enable mashup communication.These abstractions usually require browser modifications and so far have not been adopted in HTML standards [Hickson, 2011].

2 Running Example

In order to provide some background, we illustrate with a mashup different kinds of gadget inclusions and inter-frame communication. We reuse this example thought the rest of the sections.

There are two major types of gadgets in web mashups. The first type *requires* an interface from the integrator to accomplish some tasks. For instance advertisement scripts, which necessarily need to gather information of the integrator page through DOM APIs to implement the advertisement strategy. Another example of the first type of gadgets are user-supplied gadgets in social network platforms such as `facebook.com`. The second type *provides* a set of interfaces to the integrator. For instance the Google maps API is of this kind. It provides various interfaces to visualize and operate a map gadget. In this work, we are interested in the second type of gadget inclusion. We focus on the second type, that is gadget scripts that provide a set of interfaces to enable the integrator to manipulate the gadget.

In the example an integrator `i.com` wants to include a gadget `gadget.js` provided by `untrusted.com`. The integrator usually creates an empty div element to delegate part of the DOM tree. The integrator includes the gadget by using a script tag:

```
1 <div id=gadget_canvas></div>
2 <script src='http://untrusted.com/gadget.js'></script>
```

Listing 1: Code Snippet of http://i.com/integrator.html

The integrator calls methods or functions as interfaces to change the state of the gadget. For example, the following is a code snippet (in the integrator) to manipulate the untrusted gadget via interfaces:

```
1 var mydiv = document.getElementById("gadget_canvas")
2 var instance = new gadget.newInstance(
3                     mydiv, gadget.Type.SIMPLE);
4 instance.setLevel(9);
```

Listing 2: Code Snippet of http://i.com/integrator.html

The gadget defines a global variable `gadget` to provide interfaces to the integrator. The `gadget.newInstance` is used to create a new gadget instance that binds to the div; and `instance.setLevel` is a method used to change state at the gadget instance.

Assume that the integrator stores a secret in global variable `secret` and a global variable `price` holding certain information with an important integrity requirement:

```
1 var secret = document.getElementById("secret_input");
2 var price = 42
```

Listing 3: Code Snippet of http://i.com/integrator.html

If the gadget contains the following code, then the secret flows to an untrusted source and the price is modified at the gadgets will:

```
1 var steal;
2 steal = secret;
3 price = 0;
```

Listing 4: Non-benign Gadget

If the gadget is isolated using the iframe tag with a different origin, variables secret and price cannot be directly accessed by the gadget. We can modify the example in the following way:

```
1 <iframe src='http://u-i.com/gadget.html'></iframe>
```

Listing 5: Code Snippet of http://i.com/integrator-msg.html

```
1 <div id=gadget_canvas></div>
2 <script src='http://untrusted.com/gadget-msg.js'>
3 </script>
```

Listing 6: Code Snippet of http://u-i.com/gadget.html

Instead of directly including the script, the integrator invents a new origin `u-i.com` to be used as an untrusted gadget container, and puts the gadget code in a frame belonging to this origin. By doing this, the JS execution environment between integrator and gadget is isolated, as guaranteed by the browser's SOP. Limited communication between frames and integrator is possible through the PostMessage API in the browser[1] if there is an event listener for the 'message' event. To register a listener one provides a callback function as parameter and treats messages in a waiting queue, asynchronously. With PostMessage, only strings can be sent. However it is possible to marshal objects that do not point to themselves (as e.g. the global object), via the JSON stringify method.

Code in (`gadget-msg.js` and `integrator-msg.html`) needs to adapt to the asynchronous behaviour. Instead of calling methods or functions, the integrator must send messages to manipulate the untrusted gadget in iframe as shown in the following example:

```
1 PostMessage(stringify({action : "newInstance",
2              container : "gadget_div",
3              type : "SIMPLE"}),
4             "http://u-i.com");
5 PostMessage(stringify({action : "setLevel",
6              container : "gadget_div"}),
7              "http://u-i.com");
```

Listing 7: PostMessage Example

Compilation with Mashic will not preserved behavior of Listing 4 but will only preserved behavior that does not represent a confidentiality or integrity violation to the integrator.

The compiler relieves the programmer from rewriting code. Instead of rewriting gadget's code, our compiler inserts a proxy and listerner library that implement a communication protocol for manipulating gadgets independently of untrusted gadgets sandboxed in frames. Instead of rewriting integrators code, our compiler implements a CPS transformation to overcome the asynchronous nature of PostMessage. After compilation of code in Listing 1 and 2 the gadget is modified in the following way:

```
1 <html>
2   <script src="listener.js"></script>
3   <script src="untrusted.com/gadget.js"></script>
4 </html>
```

Listing 8: Compiled Gadget at http://u-i.com/gadget.html

The integrator is modified in the following way:

```
1 <html>
2   <script src="proxy.js"></script>
3   <iframe src="http://u-i.com/gadget.html"></iframe>
4   <script src="integrator_cps.js"></script>
5 </html>
```

Listing 9: Compiled Integrator

[1] Inter-frame communication is also possible via e.g navigation policies Barth et al. [2009b] but this kind of communication is now obsolete.

Notice that the gadget code used in the compiled gadget is not modified from the original one. The proxy library and the listener library provide general ways to encode gadget operations, and the programmer does not need to manually write the stub library to operate on confined gadgets. The integrator code, as we mentioned above is transformed to CPS code `integrator_cps.js` to perform the same task as in the code shown in Listing 2.

3 Mashic Compilation

In this section we describe in detail how proxy and listener libraries work. For that, we need to define opaque object handles.

Opaque Object Handle We sandbox gadgets inside an iframe tag with a different origin. By the SOP policy, the integrator and the gadget cannot exchange JS references to objects. We provide a way for the integrator to refer to objects that are defined inside the gadget, called *opaque object handles*.

An opaque object handle is essentially an unique number associated with an object in the frame. The following code excerpt demonstrate the data type for an opaque object handle:

```
1 function OHandle(id){
2    if (id == undefined) id = handle_id_gen();
3    this._id = id;
4    this._is_ohandle = true;}
```

Practically an opaque object handle is an object with a field `_is_ohandle` being true and a field `_id` being the corresponding id. The `handle_id_gen` function generates an unique id. Since the data structure for handles only contains primitive values, they can be exchanged via PostMessage and JSON Stringify.

On the listener library side, we keep a list for associating handles and objects:

```
1 var handle_list = {};
2 function add_handle_obj(ohandle,obj){
3   handle_list[ohandle._id] = obj;}
4 function get_obj_by_handle(ohandle){
5   return handle_list[ohandle._id];}
```

Since an object could possibly be an opaque object handle, it is necessary to dynamically check weather the object being operated is an opaque object handle or a local object existing in the integrator. If it is an opaque object handle, we need to proxy the operation to the sandbox; if it is a local object, we can directly operate on this object. We define an `isOpaque` function to do the dynamic check:

```
1 function isOpaque(obj){
2   if ((obj != null) && obj._is_ohandle) return true;
3   return false;
4 }
```

Listing 10: isOpaque Function

We model the interface provided by a given gadget as a set $\mathcal{V}$ of global variables in the gadget.

Example 1. For instance in our running example, $\mathcal{V} = \{$`gadget`$\}$, since `gadget` is the only global variable defined by the gadget. Another example is the interface provided by Google Maps API, that contains only the global variable `google`.

The Mashic compiler inserts bootstrapping scripts on both sides, integrator and gadget. The bootstrapping script for the integrator takes a set of variables $\mathcal{V} = \{x_1, \ldots, x_n\}$ and generates opaque object handles for each of them:

```
1 var xi = new OHandle(i);
```

Listing 11: Integrator Bootstrapping

The bootstrapping script for the gadget also generates opaque object handles and add them to a list.

```
1 add_handle_object(new OHandle(i),xi);
```

Listing 12: Gadget Bootstrapping

In the rest of the paper we let $\mathit{Bootstrap}_i^{\mathcal{V}}$ and $\mathit{Bootstrap}_g^{\mathcal{V}}$ be the bootstrapping scripts for variable set $\mathcal{V}$ for the integrator and the gadget respectively.

Proxy and Listener Interface In the rest of the paper we let P_p denote the proxy library, and P_l the listener library. On the proxy library side, we provide a series of interfaces to obtain an opaque object handle, or operate on it. The proxy library P_p and the listener library P_l is defined in Appendix A.

To obtain an opaque object handle from a global variable in the gadget, the `GET_GLOBAL_REF` interface can be used:

```
1 function GET_GLOBAL_REF(global_name,cont){
2   var m_id = gen_id();
3   var msg = {msg_id : m_id,
4              msg_type : 'GET_GLOBAL_REF',
5              global_name : global_name};
6   PostMessage(stringify(msg));
7   set_cont(m_id,cont);}
```

Listing 13: Code Snippet of the Proxy Library

The `GET_GLOBAL_REF` interface takes two parameters, the `global_name`, and a function `cont` to be used as continuation.

The `GET_GLOBAL_REF` function, upon invocation on the proxy side, composes a message with a fresh message id and sends it to the gadget in iframe. Because of the asynchronous nature of the PostMessage communication, the listener library on the gadget side cannot respond to this message immediately. Hence, we register a continuation `cont` with the message id `m_id`.

There are other interfaces that are supported to operate on opaque object handles:

- `GET_PROPERTY`: to obtain an opaque object handle or the primitive value of a property of a given object (opaque object handle);
- `OBJ_PROP_ASSIGN`: to assign a primitive value or an object or an opaque object handle to a property of a given object;
- `CALL_FUNCTION`: to call a function (opaque object handle) with all parameters being primitive values, objects or opaque object handles;
- `CALL_METHOD`: to call a method of an object (opaque object handle) with all parameters being primitive values or objects or opaque object handles;
- `NEW_OBJECT`: to instantiate a function object (that is, an opaque object handle) with all parameters being primitive values or objects or opaque object handles.

Example 2. Recall the mashup from Section 2. The interface to obtain an opaque object handle in the integrator is:

```
GET_GLOBAL_REF("gadget", function(val){...});
```

where "gadget" is the interface provided by the gadget and the second parameter is a callback function. Once the integrator obtains an opaque object handle, it can use other interfaces from the integrator to operate on the opaque object handle. If `opq_instance` corresponds to an instance object inside the gadget, to mimic the code of line 4 in Listing 2 we use:

```
CALL_METHOD(opq_instance, "setlevel",function(val){...}, 9);
```

The interface `CALL_METHOD` sends a message via PostMessage, and waits for a response from the gadget. Once the response arrives, the callback `function(val){...}` is invoked on the returned result. Note that the result might be an opaque object handle as well.

In the listener library, there are interfaces to generate a response as the following function:

```
1 function GET_GLOBAL_REF_L(recv){
2   var obj = window[recv.global_name];
3   return make_resp_msg(recv,obj);
4 }
5 function make_resp_msg(recv,obj){
6   var ohandle, msg;
7   if (obj != null &&
8       (typeof(obj) == "object" ||
9        typeof(obj) == "function"))
```

```
10      {ohandle = new OHandle();
11       add_handle_obj(ohandle,obj);
12       msg = {msg_id : recv.msg_id,
13              msg_type:'EXE_CONT',
14              return_val : ohandle};}
15   else {msg = {msg_id: recv.msg_id,
16             msg_type:'EXE_CONT',
17             return_val : obj};}
18   return msg;
19 }
```

The function GET_GLOBAL_REF_L gets the real object by the global name, and generates an opaque object handle if the object is not a primitive value. Then the opaque object handle is sent back to the integrator via PostMessage as a response for the previous sent message. Finally, the associated continuation `cont` will be applied on the response (possibly an opaque object handle).

Here we give details on how interface CALL_METHOD works.

1. The integrator invokes `CALL_METHOD(opq_obj,method,cont,args)`, where `opq_obj` stands for the object inside the iframe on which we want to invoke the `method`;`cont` is the continuation; the `args` is possibly a list of arguments.
2. The proxy sends the following message to the listener in iframe:

```
1 { msg_id : m_id,
2   msg_type: 'CALL_METHOD',
3   object : opq_obj,
4   method_name: method,
5   arguments : args }
```

3. The proxy library associates `m_id` with the continuation `cont`.
4. When the listener receives the message, it first obtains the real `object` corresponding to the handle `obj`; and then it converts the opaque object handles in `arguments` to corresponding objects; and finally it invokes `object[method_name]` on the `arguments`.
5. Once the invocation is finished, it sends back a message:

```
1 { msg_id: m_id,
2   msg_type:'EXE_CONT',
3   return_val : val}
```

   where `val` is either a primitive value or an opaque object handle.
6. Upon receiving the response, the proxy library applies the continuation `cont` with the received result `val`.

CPS Transformation JS does not support Scheme-style call/cc (Call-with-Current-Continuation) for suspending and resuming an execution. Demanding the programmer to write in CPS style would turn the proposal impractical.

Example 3. Recall the example in Section 2. In order to obtain the property `gadget.Type.SIMPLE`, the programmer should write the following code (using the proxy interface):

```
1 GET_GLOBAL_REF("gadget",
2   function(opq_gadget){
3     GET_PROPERTY(opq_gadget,"Type",
4       function(opq_Type){
5         GET_PROPERTY(opq_Type,"SIMPLE",
6           function(val_SIMPLE){...});});});
```

This style is similar to the CPS , where one needs to explicitly specify continuations for the rest of a computation. In order to reuse the legacy code that operates on a gadget, we propose an automated CPS transformation of legacy code in such a way that programmers *do not need* to rewrite their code. Legacy code in the integrator will be CPS-transformed, and inserted with dynamic checks for opaque object handles when necessary.

```
C⟨s0; s1⟩ :
  function(_k){
    C⟨s0⟩(function(_v){C⟨s1⟩(_k); });
  }

C⟨if (e) s0 else s1⟩ :
  function(_k){
    C⟨e⟩(function(_b){
            if (_b) C⟨s0⟩(_k)
              else C⟨s1⟩(_k);
        });
  }

C⟨while (e) s⟩ :
  function(_k){
    var _c;
    _c = function(_v){
           C⟨e⟩(function(_b){
                   if (_b) C⟨s⟩(_c)
                     else _k(undefined);
                 });
         };
    _c(undefined);
  }

C⟨return e⟩ :
  function(_k){C⟨e⟩(_fun_cont)}
```

Figure 2: CPS Transformation of Statements

```
C⟨this⟩ :
  function(_k){_k(_this); }

C⟨pv⟩ :
  function(_k){_k(pv); }

C⟨x⟩ :
  function(_k){_k(x); }

x = e
  function(_k){
    C⟨e0⟩(function(_x0){
      _k(x = _x0);
    });
  }

e0 op e1
  function(_k){
    C⟨e0⟩(function(_x0){
      C⟨e1⟩(function(_x1){
        _k(_x0 op _x1);
      });
    });
  }

typeof e
  function(_k){
    C⟨e0⟩(function(_x0){
      _k(typeof _x0);
    });
  }

{m0 : e0, m1 : e1}
  function(_k){
    C⟨e0⟩(function(_x0){
      C⟨e1⟩(function(_x1){
        _k({m0 : _x0, m1 : _x1});
      });
    });
  }

function(x){s}
  function(_k){
    _k(function(_fun_cont, x){
      var _this;
      _this = this;
      C⟨s⟩(_fun_cont);
    });
  }
```

Figure 3: CPS Transformation of Expressions, Non-Message-Passing part

We formally define the CPS transformation of an integrator code s, denoting $\mathcal{C}\langle s\rangle$. The function $\mathcal{C} : s \mapsto s$ transforms JS code into CPS style. CPS-transformed programs are functions that take as parameter another function as an explicit continuation of the computation. The transformation rules are defined in Figure 2, 3 and 4. For each operation, the compilation inserts dynamic checks to check whether the object is an opaque object handle or not.

We transform $e_0[e_1]$ to a function taking a parameter $_k$ as continuation. In the body of the function, we apply the CPS-transformed code of e_0 to a continuation where the CPS-transformed code of e_1 is applied to an inner-most continuation. In the inner-most continuation $_x_0$ and $_x_1$ bind to the results of evaluating e_0 and e_1 respectively. We dynamically check if $_x_0$ is an opaque object object handle to decide whether to use the proxy interface or to apply $_k$ to $_x_0[_x_1]$ directly. Notice that $_x_1$ can only hold a string, otherwise the execution blocks since in our simplified JS semantics we do not consider type-casting.

The transformation of the expression new $e_0(e_1)$ is trickier. The semantics of new is similar to calling a function except that the returning value is not the result of evaluating the function but the newly created object. In the inner-most continuation, we first create a dummy function $_x_3$ which duplicates the prototype of $_x_0$, then we create an empty object $_x_2$ with this dummy function (with the same prototype as in $_x_0$). Next we create a warp continuation $_x_4$ where $_k$ is always applied to $_x_2$, no matter what is the parameter $_v$. Finally, we apply $_x_0$ to $_x_1$, using $_x_4$ as continuation and binding $_x_2$ to *this* keyword (to initialize properties in $_x_2$). Notice that the continuation will be always applied to $_x_2$, as new $e_0(e_1)$ will return the created object rather than the result of the function invocation.

```
C⟨e0[e1]⟩ :
  function(_k){
    C⟨e0⟩(function(_x0){
      C⟨e1⟩(function(_x1){
        if (isOpaque(_x0)){
          GET_PROPERTY(_x0, _x1, _k);
        } else {
          _k(_x0[_x1]);
        }
      });});}

C⟨new e0(e1)⟩ :
  function(_k){
    C⟨e0⟩(function(_x0){
      C⟨e1⟩(function(_x1){
        if (isOpaque(_x0)){
          NEW_OBJECT(_x0, _x1, _k);
        } else {
          var _x2, _x3, _x4;
          _x3 = function(x){};
          _x3["prototype"] = _x0["prototype"];
          _x2 = new _x3();
          _x4 = function(_v){_k(_x2);};
          _x0["apply"](_x2, _x4, _x1);
        }
      });});}

C⟨e0(e1)⟩ :
  function(_k){
    C⟨e0⟩(function(_x0){
      C⟨e1⟩(function(_x1){
        if (isOpaque(_x0)){
          CALL_FUNCTION(_x0, _x1, _k);
        } else {
          _x0(_k, _x1);
        }
      });});}

C⟨e0[e1](e2)⟩ :
  function(_k){
    C⟨e0⟩(function(_x0){
      C⟨e1⟩(function(_x1){
        C⟨e2⟩(function(_x2){
          if (isOpaque(_x0[_x1])){
            CALL_METHOD(_x0, _x1, _x2, _k);
          } else {
            _x0[_x1](_k, _x2);
          }
        });});});}

e0[e1] = e2
  function(_k){
    C⟨e0⟩(function(_x0){
      C⟨e1⟩(function(_x1){
        C⟨e2⟩(function(_x2){
          if (isOpaque(_x0[_x1])){
            PROPERTY_ASSIGN(_x0, _x1, _x2, _k);
          } else {
            _k(_x0[_x1] = _x2);
          }
        });});});}
```

Figure 4: CPS Transformation of Expressions, Message-Passing Part

4 Decorated Semantics

In this section we define JS (integrated to HTML) decorated semantics to formally present the correctness and security results of the compiler in following sections.

Corresponding heaps in the executions of original and compiled mashups have different structures and, in particular, objects in the original heap are composed by properties created by the gadget or the integrator, whereas in the compiled mashup objects are not shared between different principals.

We propose a JS decorated semantics to partition a heap at the granularity of object properties according to three different colors. We denote the *Integrator Principal* with ♠, the *Gadget Principal* with V, and we use W to denote neutral principal. We use □ or △ to denote any of them.

For the sake of brevity, we do not include an origin parameter in the primitive PostMessage since there is only two possibilities: either the integrator communicates with the frame or viceversa. We also simplify AddListener for the formal presentation: we assume that the only events it listens to is the event "message".

Our Javascript semantics rules are mostly compliant with Javascript semantics of Maffeis et al. [2008], that conforms to the ECMAScript (ECMA262-3) specification.

4.1 Heaps and Objects

Objects are the main protagonists in JS semantics. An object o is a tuple $\{i_{1\{\square\}} : v_1, \ldots i_{n\{\triangle\}} : v_n\}$ associating decorated properties $i_{\{\square\}}$ (internal identifiers or strings) to values. We use i instead of $i_{\{\square\}}$ whenever the decoration is not important.

We distinguish internal properties that cannot be changed by programs with the symbol "@" in front

of an identifier. We do not model attributes of properties that may indicate access controls as for example "do-not-delete" attributes [Maffeis et al., 2008].

We present a series of auxiliary definitions used in the operational semantics. For an object o and a property i, we use $i_{\{\square\}} \in o$ to denote o has property i with decoration $\square$, and use $i \notin o$ to denote o does not have property i with any decoration.

Objects are stored in heaps. A heap h is a partial mapping from locations in a set $\mathcal{L}$ to objects. We use the notation $h(\ell) = o$, to retrieve the object o stored in location ℓ; and the notation $o.i_{\{\square\}} = v$ to retrieve the value store in property $i_{\{\square\}}$. We also use a shortcut $h(\ell).i_{\{\square\}}$ whenever possible. To update (or create) a property $i_{\{\square\}}$ of an object at location ℓ in the heap, we use the notation $h(\ell.i_{\{\square\}} = v) = h'$, where h' is the updated heap. We also use $\text{Alloc}(h, o) = h', \ell'$, where $\ell' \notin \mathsf{dom}(h)$, for allocating a fresh location for an object in the heap. The new heap after adding the location is h'.

JS heaps contain two important chains of objects. The *scope chain* in a JS execution keep track of the dynamic chains of function calls via the @*scope* property. To resolve a scope of a variable name, one starts from the bottom of the chain, until reaching a scope object which contains the searched variable name. The scope look-up process $\text{Scope}(h, \ell, m)$ function is defined by the following rules. It take 3 parameters: a current heap, a heap location for a scope object (as the bottom of the scope chain), and a variable name as string to look up.

$$\textsc{Scope-null}\quad \text{Scope}(h, \mathit{null}, m) = \mathit{null} \qquad \textsc{Scope-ref}\quad \frac{m \in h(\ell)}{\text{Scope}(h, \ell, m) = \ell} \qquad \textsc{Scope-lookup}\quad \frac{m \notin h(\ell) \qquad \text{Scope}(h, h(\ell).@\mathit{scope}, m) = \ell_n}{\text{Scope}(h, \ell, m) = \ell_n}$$

Example 4. To lookup for name x from scope object ℓ in h, we use $\text{Scope}(h, \ell, \text{"}x\text{"})$.

Similarly, the *prototype chain* represents the hierarchy between objects. A property that is not present in the current object, will be searched in the prototype chain, via the @*prototype* property. The helper function $\text{Prototype}(h, \ell, m)$ looks for the m property of the object $h(\ell)$ via the prototype chain.

$$\textsc{Prototype-null}\quad \text{Prototype}(h, \mathit{null}, m) = \mathit{null} \qquad \textsc{Prototype-ref}\quad \frac{m \in h(\ell)}{\text{Prototype}(h, \ell, m) = \ell}$$

$$\textsc{Prototype-lookup}\quad \frac{m \notin h(\ell) \qquad \text{Prototype}(h, h(\ell).@\mathit{prototype}, m) = \ell_n}{\text{Prototype}(h, \ell, m) = \ell_n}$$

In the top of a scope chain, there is a distinguished object called the global object. We define (a simplified version of) an intial global object below for the integrator (we use the form #*addr* to represent an unique heap location):

$$\mathit{global}_i = \left\{ \begin{array}{ll} @\mathit{this}_{\{\mathrm{W}\}} : & \#\mathit{global}_i \\ @\mathit{scope}_{\{\mathrm{W}\}} : & \mathit{null} \\ \text{"}\mathit{Stringify}\text{"}_{\{\mathrm{W}\}} : & \#\mathit{stringify}_i \\ \text{"}\mathit{Parse}\text{"}_{\{\mathrm{W}\}} : & \#\mathit{parse}_i \\ \text{"}\mathit{PostMessage}\text{"}_{\{\mathrm{W}\}} : & \#\mathit{postmessage}_i \\ \text{"}\mathit{Addlistener}\text{"}_{\{\mathrm{W}\}} : & \#\mathit{addlistener}_i \\ \text{"}\mathit{window}\text{"}_{\{\mathrm{W}\}} : & \#\mathit{global}_i \end{array} \right\}$$

Global variables are defined as properties in the global object. For example *window* is a global variable holding the location $\#\mathit{global}_i$ of the global object. Notice that properties in the initial global object are decorated with W, which are not considered as heap locations created neither by the integrator nor the gadget.

Since by SOP the integrator and the frame does not have shared objects in the heap, we define similarly an initial global object global_f for the frame, in which the properties hold locations $\#\mathit{global}_f$,

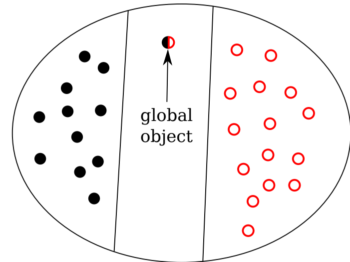


Figure 5: Uniformly Colored Heap

$\#stringify_f, \ldots,$ and $\#addlistener_f$.

$$global_f = \left\{ \begin{array}{ll} @this_{\{W\}}: & \#global_f \\ @scope_{\{W\}}: & null \\ \text{"}Stringify\text{"}_{\{W\}}: & \#stringify_f \\ \text{"}Parse\text{"}_{\{W\}}: & \#parse_f \\ \text{"}PostMessage\text{"}_{\{W\}}: & \#postmessage_f \\ \text{"}Addlistener\text{"}_{\{W\}}: & \#addlistener_f \\ \text{"}window\text{"}_{\{W\}}: & \#global_f \end{array} \right\}$$

Heap locations of the form $\#addr_f$ with a subscript f, as in $\#global_f$, denote native objects that resides in the frame reserved part of the heap, as described by the semantics rules shown later.

Native functions in a heap are represented by locations (e.g. $\#postmessage_i$) as abstract function objects. We use *NativeFuns* to denote the set of locations of native functions. We give definition for pre-defined native objects existing in an initial heap when initializing an integrator or a frame. These native objects are defined below:

Object Prototype
$objprot = \{@prototype : null\}$

Function Prototype
$funprot = \{@prototype : null\}$

Stringify Function
$$stringify = \left\{ \begin{array}{ll} @prototype: & \#funprot \\ @call: & true \end{array} \right\}$$

Parse Function
$$parse = \left\{ \begin{array}{ll} @prototype: & \#funprot \\ @call: & true \end{array} \right\}$$

PostMessage Function
$$postmessage = \left\{ \begin{array}{ll} @prototype: & \#funprot \\ @call: & true \end{array} \right\}$$

AddListener Function
$$addlistener = \left\{ \begin{array}{ll} @prototype: & \#funprot \\ @call: & true \end{array} \right\}$$

The prototype objects of object and function are used as default prototypes. We have 4 native functions defined for marshaling/demarshaling objects to strings, posting messages, and setting event listener.

We assume that $\text{Alloc}(h, o)$ will never allocate those pre-defined heap locations mentioned above. We also use $\oplus$ to denote the union of two disjoint heap (with non-overlapping addresses).

It is useful to define an initial heap. An initial heap for the integrator (resp. for the frame) denoted by h_{in} (resp. h^f_{in}), such that $h_{in}(\#global_i) = global_i$ (for the case of frame $h^f_{in}(\#global_f) = \#global_f$). We give the definition of the intial heaps below:

$$h_{in} = \left\{ \begin{array}{ll} \#global & \mapsto global, \\ \#objprot & \mapsto objprot, \\ \#funprot & \mapsto funprot, \\ \#stringify & \mapsto stringify, \\ \#parse & \mapsto parse, \\ \#postmessage & \mapsto postmessage, \\ \#addlistener & \mapsto addlistener \end{array} \right\} \qquad h^f_{in} = \left\{ \begin{array}{ll} \#global_f & \mapsto global, \\ \#objprot_f & \mapsto objprot, \\ \#funprot_f & \mapsto funprot, \\ \#stringify_f & \mapsto stringify, \\ \#parse_f & \mapsto parse, \\ \#postmessage_f & \mapsto postmessage, \\ \#addlistener_f & \mapsto addlistener \end{array} \right\}$$

We say that a decorated object o is single-colored if and only if all properties of o are decorated with the same color. We say that a decorated heap h is uniformly colored if and only if for all $\ell \in \mathsf{dom}(h)$ such

that $h(\ell)$ is not a global object, then $h(\ell)$ is a single-colored object (see Figure 5, where solid black dots are ♠-colored objects, hollow red dots are V-colored objects). Notice that the only object in the heap that may contain different colors is the global object, since global variables can be created with different decorations as properties of the global object in JS.

Definition 1 (Object Inclusion)**.** $o' \subseteq o$ if and only if

1. $\mathsf{dom}(o') \subseteq \mathsf{dom}(o)$;
2. $\forall\ i_{\{\square\}} \in \mathsf{dom}(o'), o'.i_{\{\square\}} = o.i_{\{\square\}}$.

Definition 2 (Heap Inclusion)**.** $h' \subseteq h$ if and only if

1. $\mathsf{dom}(h') \subseteq \mathsf{dom}(h)$;
2. $\forall\ \ell \in \mathsf{dom}(h'), h'(\ell) \subseteq h(\ell)$.

We also define well-formedness of a decorated heap.

Definition 3 (Well-formed Decorated Heap)**.** Let h be a decorated heap, h is a well-formed decorated heap, if and only if h is a uniformly colored heap and $h_{in} \subseteq h$.

The projection $o\lfloor_{\square}$ for a decorated object o is defined by eliminating non-$\square$ colored properties of o. If there is no property in o with color $\square$ then the projection is undefined and denoted by $\perp$. We define heap projections in order to reason about the portion of the heap owned by a given principal.

Definition 4 (Heap Projection)**.** Given a heap h, projection $h\lfloor_{\square}$ is either undefined if there are no objects with property of color $\square$ or it is a heap h' such that: $\forall \ell \in \mathsf{dom}(h), h(\ell)\lfloor_{\square} \neq \perp \Leftrightarrow \ell \in \mathsf{dom}(h')\ \&\ h'(\ell) = h(\ell)\lfloor_{\square}$.

Remark 1. If h is a uniformly colored heap, and $h' = h\lfloor_{\square}$, then for all $\ell \in \mathsf{dom}(h)$ such that $\ell \neq \#global$ or $\ell \neq \#global_f$, $h'(\ell) = h(\ell)$.

We define $h = h'$ as equality on heaps. We denote $h' =_{\square} h$ for $h'\lfloor_{\square} = h\lfloor_{\square}$. We also denote $h' \subseteq_{\square} h$ for $h'\lfloor_{\square} \subseteq h\lfloor_{\square}$.

4.2 Syntax

We present in Figure 6 a simplified syntax of the language extended with HTML constructs, where we assume a set of URLs or origins $\mathcal{U}rl$, and $u \in \mathcal{U}rl$.

A program in the language is an HTML page M with embedded scripts and frames. Frames are important to reason about the SOP and untrusted code. For simplicity, we choose to restrict the language with at most one frame in HTML pages. Inclusion of many frames does not add any insights to the technical results but adds confusion. (This restriction does not apply to the Mashic compiler.) We assume given an environment $\mathsf{Web} : \mathcal{U}rl \mapsto J$ that maps URLs to gadgets code. When a program M contains an iframe, we model with $\mathsf{Web}(u)$ (implicit in the semantics rules) a gadget from a different origin $u \in \mathcal{U}rl$. In the syntax, scripts are decorated with a color to denote the principal owner of the script. Statements and expressions ranged over by P, s, and e are standard. Notice that f ranges over native functions, as the native PostMessage function.

Before a JS program in a script node is executed, or before a body of a function is evaluated, all variable declarations are added to the current scope object in the heap. To that end, we use a function VD that takes as parameters a heap h, a location ℓ of the current scope object, a statement s, and a color $\square$ to bind variables with proper decorations to the scope object ℓ:

$$\mathrm{VD}(h,\ell,s,\square) = \begin{cases} h & \text{if } s = e \\ \mathrm{VD}(\mathrm{VD}(h,\ell,s_0,\square),\ell,s_1,\square) & \text{if } s = s_0; s_1 \\ h(\ell.x_{\{\square\}} = \mathit{undefined}) & \text{if } s = \mathsf{var}\ x \\ \mathrm{VD}(h,\ell,s_0;s_1,\square) & \text{if } s = \mathsf{if}\ (e)\ s_0\ \mathsf{else}\ s_1 \\ \mathrm{VD}(h,\ell,s,\square) & \text{if } s = \mathsf{while}\ (e)\ s \\ \mathrm{VD}(h,\ell,s,\square) & \text{if } s = \mathsf{return}\ e \end{cases}$$

M	$::=$	<html> F J </html>	HTML page
F	$::=$	<iframe src=u></iframe> $\mid \epsilon$	a frame or empty
J	$::=$	<script□> s </script> $J \mid \epsilon$	sequence of scripts
P, s	$::=$	e	expression
	$\mid$	$s; s$	block
	$\mid$	var x	variable declaration
	$\mid$	if (e) s else s	conditional
	$\mid$	while (e) s	while loop
	$\mid$	return e	return
e	$::=$	this	special property
	$\mid$	x	identifier
	$\mid$	f	native functions
	$\mid$	pv	primitive values
	$\mid$	$\{m_0 : e_0, \ldots, m_n : e_n\}$	object literal
	$\mid$	$e_0[e_1]$	member selector
	$\mid$	new $e_0(e_1)$	constructor invocation
	$\mid$	$e_0(e_1)$	function invocation
	$\mid$	function$(\vec{x})\{s\}$	function expressions
	$\mid$	e_0 bin e_1	binary operations
	$\mid$	typeof e	typeof expression
f	$::=$	$PostMessage \mid AddListener$ $Stringify \mid Parse$	native functions
pv	$::=$	m	string
	$\mid$	n	number
	$\mid$	b	boolean
	$\mid$	$null$	null
bin	$::=$	$+ \mid - \mid < \mid > \mid === \mid =$	binary operators

Figure 6: JS Language Syntax

Finally we define a helper function GetType(v), to return a string as the type of a primitive value.

$$\text{GetType}(h, v) = \begin{cases} \text{“}\textit{number}\text{”} & \text{if } v = n \\ \text{“}\textit{string}\text{”} & \text{if } v = m \\ \text{“}\textit{boolean}\text{”} & \text{if } v = b \\ \text{“}\textit{undefined}\text{”} & \text{if } v = \textit{null} \text{ or } v = \textit{undefined} \\ \text{“}\textit{object}\text{”} & \text{if } v = l \text{ and } @\textit{call} \notin h(\ell) \\ \text{“}\textit{function}\text{”} & \text{if } v = l \text{ and } @\textit{call} \in h(\ell) \end{cases}$$

4.3 Configurations and Transition Relation

Instrumented global configurations feature a decoration component that denotes the owner principal of the program being executed. Decorations are propagated via semantics rules and, importantly, do not affect the normal semantics of JS programs (they can be erased without further changes in the state). A global configuration is a 5-tuple $\langle \Box, h, \ell, R, Q \rangle_I$ that features:

- A subscript I identifying the execution context of current code, (I denotes that the current context is the *integrator*, and F denotes that the current context is the *frame*. We use the subscript x to denote a wildcard symbol for both I or F. This subscript is used to identify destinations of a message being sent.)
- A decoration $\Box$ that denotes the principal of the current program in the configuration.
- A heap h.
- A location $\ell \in \mathcal{L}$ pointing to the current scope object (or *null* only for the initial configuration).
- A run-time program R currently being executed, and a waiting queue Q in order to give semantics to PostMessage mechanism.
- A waiting queue that is of the form $\langle \ell_i, mq_i \rangle \parallel \langle \ell_f, mq_f \rangle$, where ℓ_i and ℓ_f are locations for event listeners and mq_i and mq_f are message queues for both, the integrator and the frame, respectively. The syntax for defining a message queue is :

$$mq \quad ::= \quad m \; mq \;\mid\; \varepsilon$$

  where m is a string. We use $mq_1 + mq_2$ to denote the concatenation of two message queues.

An initial configuration is of the form $\langle \Box, \varepsilon, \textit{null}, M, Q_{init} \rangle_I$ where $Q_{init} = \langle \textit{null}, \varepsilon \rangle \parallel \langle \textit{null}, \varepsilon \rangle$. We use a transition system to define the semantics of our language, via the $\rightarrow$ relation between global configurations. We denote by $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$.

We also define a configuration for the core JS semantics

$$(\Box, h, \ell, s)$$

to be a 4-tuple featuring:

- A decoration $\Box$ that denotes the principal of the current program in the configuration;
- A heap h;
- A location ℓ of the current scope object;
- A current statement s being evaluated.

the transitions between core JS configurations is featured with a label ℓmq: $(h, \ell, s) \xrightarrow{\ell mq} (h', \ell, s')$. The label ℓmq carries the side-effect of PostMessage and event listeners added by the native function addlistener, and is defined by the following syntax:

$$\ell mq \quad ::= \quad mq \;\mid\; \ell + mq$$

where mq is a message queue and ℓ is a mark that denotes that an event listener is added and holds in location ℓ of the heap. We denote $\xrightarrow{\ell mq}^*$ to be a transitive closure of the transition relation, where ℓmq denotes the accumulated side-effect. A trace of transitions is *valid* only if there is at most one side-effect for *addlistener*.

4.4 DOM Semantics Rules

We extend the syntax with run-time expressions as the following.

$$
\begin{array}{rcll}
R & ::= & M \mid F\ J \mid F_{RT}\ J \mid J \mid s\ J & \text{run-time programs} \\
F_{RT} & ::= & \texttt{<iframe>}\ J\ \texttt{</iframe>} & \text{run-time frames} \\
 & \mid & \texttt{<iframe>}\ s\ J\ \texttt{</iframe>} & \\
e & ::= & \ldots \mid @\mathsf{FunExe}(\ell, s, \square) \mid @\mathsf{NewExe}(\ell_o, \ell, s, \square) & \text{run-time expressions} \\
v & ::= & pv \mid \ell \mid \mathit{undefined} & \text{run-time values} \\
i & ::= & @x \mid m & \text{properties of objects}
\end{array}
$$

In the run-time syntax, R denotes run-time programs being executed, extended by run-time frame F_{RT}. The run-time expression e is extended with two types of function execution $@\mathsf{FunExe}(\ell, s, \square)$ and $@\mathsf{NewExe}(\ell_o, \ell, s, \square)$. v denotes run-time values which consist of primitive values pv, heap locations ℓ, undefined value *undefined*.

We define semantics rules for the DOM, *i.e.* the global transitions, in Figure 7. Now we comment on the semantics rules.

DInit A mashup execution initializes the heap of the configuration to the initial heap of the integrator h_{in}. The scope object is set to the global object $\#global$.

DScript A $\triangle$-decorated script starts by $\mathrm{VD}(h, \ell, s, \triangle)$ to initialize variables defined in s to the current scope object ℓ in h. The new configuration has color $\triangle$.

DScriptFini When an execution of a statement terminates, we continue with the rest of the computation.

DScript-I-1 This is a contextual rule: if the core JS configuration can advance by 1 step with label mq as the integrator, then the global configuration will accordingly update the message queue mq_f for the frame. If the listener for the frame ℓ_f is not *null*, then we append mq to mq_f, otherwise we do nothing since no listener will respond to incoming messages.

DScript-I-2 This rule is similar to DScriptFini except it set the listener of the integrator to ℓ' appeared in the label of core JS transition.

DScript-F-1 Similar to rule DScript-I-1.

DScript-F-2 Similar to rule DScript-I-2.

DFrameInit A frame fetches the content $\mathsf{Web}(u)$ and joins the initial frame heap h^f_{in} to the current heap. Addresses in h do not overlap with addresses in h^f_{in} by the SOP. Notice that the current scope object is set to the frame's global object.

DFrameFini This is similar to rule DScriptFini when the execution of a frame is terminated.

DFrameExec This is a contextual rule for the execution of the program inside a frame.

DCallback-I When no program is executing, we can apply the event listeners to pending messages in the queues (this rule and rule DCallback-F). For example, if the integrator's event listener ℓ_i is not *null* and the message queue mq_i is not empty, then we can apply the listener to the first message in the queue. Note that the only non-determinism comes from these two rules for event listeners.

DCallback-F See explanation above.

4.5 Core Semantics Rules

The semantics rules of core JS is defined in a context-redex style in Figure 8 and 9 and 10. The evaluation contexts of the core JS are defined below, where $op \in \{\texttt{<},\texttt{>},\texttt{+},\texttt{-},\texttt{===}\}$:

$$
\begin{array}{lcl}
\mathbf{C} & ::= & _ \mid \mathbf{C_i}[\mathbf{C}] \mid \mathbf{C}\texttt{=}e \\
\mathbf{C_i} & ::= & \mathbf{C_v} \mid _(e) \\
\mathbf{C_v} & ::= & _[e] \mid l[_] \mid \mathsf{new}\ _(e) \mid \mathsf{new}\ l(_) \\
 & \mid & l(_) \mid l[m](_) \mid _\ op\ e \mid v\ op\ _ \\
 & \mid & \mathsf{typeof}\ _ \mid x = _ \mid l[m] = _
\end{array}
$$

Evaluation context $\mathbf{C_i}$ is special for a redex in the form of x (an identifier) to be evaluate to a value. For example, in the expression $x = 3$, x should not be evaluated to a value. Therefore $_\texttt{=}e$ is not a $\mathbf{C_i}$

DInit

$$\langle \Box, \varepsilon, null, \texttt{<html>}\ FJ\ \texttt{</html>}, Q_{init}\rangle_I \rightarrow \langle \Box, h_{in}, \#global, FJ, Q_{init}\rangle_I$$

DScript

$$\frac{\mathrm{VD}(h,\ell,s,\triangle)=h'}{\langle \Box, h, \ell, \texttt{<script}\triangle\texttt{>}\ s\ \texttt{</script>}J, Q\rangle_x \rightarrow \langle \triangle, h', \ell, s\ J, Q\rangle_x}$$

DScriptFini

$$\frac{}{\langle \Box,h,\ell,v\ J,Q\rangle_x \rightarrow \langle \Box,h,\ell,J,Q\rangle_x}$$

DScript-I-1

$$\frac{(\Box,h,\ell,s)\xrightarrow{mq}(\Box,h',\ell,s') \qquad mq'_f = \begin{cases} mq_f & \text{if } \ell_f = null \\ mq_f + mq & \text{otherwise}\end{cases}}{\langle \Box,h,\ell,s\ J,\langle \ell_i, mq_i\rangle \parallel \langle \ell_f, mq_f\rangle\rangle_I \rightarrow \langle \Box,h',\ell,s'\ J,\langle \ell_i,mq_i\rangle \parallel \langle \ell_f, mq'_f\rangle\rangle_I}$$

DScript-I-2

$$\frac{(\Box,h,\ell,s)\xrightarrow{\ell'+mq}(\Box,h',\ell,s') \qquad mq'_f = \begin{cases} mq_f & \text{if } \ell_f = null \\ mq_f + mq & \text{otherwise}\end{cases}}{\langle \Box,h,\ell,s\ J,\langle null, \varepsilon\rangle \parallel \langle \ell_f, mq_f\rangle\rangle_I \rightarrow \langle \Box,h',\ell,s'\ J,\langle \ell',\varepsilon\rangle \parallel \langle \ell_f, mq'_f\rangle\rangle_I}$$

DScript-F-1

$$\frac{(\Box,h,\ell,s)\xrightarrow{mq}(\Box,h',\ell,s') \qquad mq'_i = \begin{cases} mq_i & \text{if } \ell_i = null \\ mq_i + mq & \text{otherwise}\end{cases}}{\langle \Box,h,\ell,s\ J,\langle \ell_i, mq_i\rangle \parallel \langle \ell_f, mq_f\rangle\rangle_F \rightarrow \langle \Box,h',\ell,s'\ J,\langle \ell_i,mq'_i\rangle \parallel \langle \ell_f, mq_f\rangle\rangle_F}$$

DScript-F-2

$$\frac{(\Box,h,\ell,s)\xrightarrow{\ell'+mq}(\Box,h',\ell,s') \qquad mq'_i = \begin{cases} mq_i & \text{if } \ell_i = null \\ mq_i + mq & \text{otherwise}\end{cases}}{\langle \Box,h,\ell,s\ J,\langle \ell_i, mq_i\rangle \parallel \langle null, \varepsilon\rangle\rangle_F \rightarrow \langle \Box,h',\ell,s'\ J,\langle \ell_i,mq'_i\rangle \parallel \langle \ell', \varepsilon\rangle\rangle_F}$$

DFrameInit

$$\frac{\mathsf{Web}(u) = J' \qquad J' \neq \varepsilon}{\langle \Box, h, \#global, \texttt{<iframe src=}u\texttt{></iframe>}\ J, Q\rangle_I \rightarrow \langle \Box, h \oplus h_f, \#global_f, \texttt{<iframe>}\ J'\ \texttt{</iframe>}\ J, Q\rangle_F}$$

DFrameFini

$$\langle \Box, h, \#global_f, \texttt{<iframe>}\ v\ \texttt{</iframe>}\ J, Q\rangle_F \rightarrow \langle \Box, h, \#global, J, Q\rangle_I$$

DFrameExec

$$\frac{\langle \Box, h, \#global_f, P, Q\rangle_F \rightarrow \langle \triangle, h', \#global_f, P', Q'\rangle_F}{\langle \Box, h, \#global_f, \texttt{<iframe>}\ P\ \texttt{</iframe>}\ J, Q\rangle_F \rightarrow \langle \triangle, h', \#global_f, \texttt{<iframe>}\ P'\ \texttt{</iframe>}\ J, Q'\rangle_F}$$

DCallback-I

$$\frac{\ell_i \neq null}{\langle \Box, h, \ell, \varepsilon, \langle \ell_i, m + mq_i\rangle \parallel \langle \ell_f, mq_f\rangle\rangle_x \rightarrow \langle \Box, h, \#global, \ell_i(m), \langle \ell_i, mq_i\rangle \parallel \langle \ell_f, mq_f\rangle\rangle_I}$$

DCallback-F

$$\frac{\ell_f \neq null}{\langle \Box, h, \ell, \varepsilon, \langle \ell_i, mq_i\rangle \parallel \langle \ell_f, m + mq_f\rangle\rangle_x \rightarrow \langle \Box, h, \#global_f, \ell_f(m), \langle \ell_i, mq_i\rangle \parallel \langle \ell_f, mq_f\rangle\rangle_F}$$

Figure 7: Decorated Semantics Rules (DOM)

DThis

$$\frac{h(\ell).@this = v}{(\square, h, \ell, \mathbf{C}[\mathsf{this}]) \xrightarrow{\varepsilon} (\square, h, \ell, \mathbf{C}[v])}$$

DObj-Literal

$$\frac{o = \{@prototype_{\{\square\}} : \#objprot\} \qquad \text{Alloc}(h, o) = h_1, \ell_o \\ (\square, h_i, \ell, e_i) \xrightarrow{mq_i}{}^* (\square, h'_i, \ell, v_1) \qquad h'_i(\ell_o.m_{i\{\square\}} = v_i) = h_{i+1} \qquad mq = mq_1 + mq_2 + \cdots mq_n}{(\square, h, \ell, \mathbf{C}[\{m_1 : e_1, \ldots, m_n : e_n\}]) \xrightarrow{mq} (\square, h_{n+1}, \ell, \mathbf{C}[\ell_o])}$$

DCallFunc

$$\frac{\ell_1 \notin NativeFuns \qquad h(\ell_1).@body_{\{\triangle\}} = \mathsf{function}(x)\{s\} \qquad \ell_g = \text{GetGlobal}(h, \ell) \\ o_s = \left\{ \begin{array}{ll} @scope_{\{\triangle\}} : & h(l_1).@fscope \\ @prototype_{\{\triangle\}} : & null \\ @this_{\{\triangle\}} : & \ell_g \\ \text{“}x\text{”}_{\{\triangle\}} : & v \end{array} \right\} \qquad \text{Alloc}(h, o_s) = h_1, \ell_s \qquad \text{VD}(h_1, \ell_s, s, \triangle) = h_2}{(\square, h, \ell, \mathbf{C}[\ell_1(v)]) \rightarrow (\triangle, h_2, \ell_s, \mathbf{C}[@\mathsf{FunExe}(\ell, s, \square)])}$$

DCallMethod

$$\frac{\text{Prototype}(h, \ell_1, m) = \ell_2 \neq null \qquad h(\ell_2).m = \ell_3 \qquad \ell_3 \notin NativeFuns \qquad h(\ell_3).@body_{\{\triangle\}} = \mathsf{function}(x)\{s\} \\ o_s = \left\{ \begin{array}{ll} @scope_{\{\triangle\}} : & h(\ell_3).@fscope \\ @prototype_{\{\triangle\}} : & null \\ @this_{\{\triangle\}} : & \ell_1 \\ \text{“}x\text{”}_{\{\triangle\}} : & v \end{array} \right\} \qquad \text{Alloc}(h, o_s) = h_1, \ell_s \qquad \text{VD}(h_1, \ell_s, s, \triangle) = h_2}{(\square, h, \ell, \mathbf{C}[\ell_1[m](v)]) \rightarrow (\triangle, h_2, \ell_s, \mathbf{C}[@\mathsf{FunExe}(\ell, s, \square)])}$$

DCallContext

$$\frac{(\square, h, \ell_s, s) \rightarrow (\square, h', \ell'_s, s')}{(\square, h, \ell_s, \mathbf{C}[@\mathsf{FunExe}(\ell, s, \triangle)]) \rightarrow (\square, h', \ell'_s, \mathbf{C}[@\mathsf{FunExe}(\ell, s', \triangle)])}$$

DCallFini

$$(\square, h, \ell_s, \mathbf{C}[@\mathsf{FunExe}(\ell, v, \triangle)]) \rightarrow (\triangle, h, \ell, \mathbf{C}[undefined])$$

DCallRet

$$(\square, h, \ell_s, \mathbf{C}[@\mathsf{FunExe}(\ell, \mathsf{return}\ v, \triangle)]) \rightarrow (\triangle, h, \ell, \mathbf{C}[v])$$

DNew

$$\frac{o = \{@prototype_{\{\triangle\}} : h(\ell_1).\text{“}prototype\text{”}\} \\ \text{Alloc}(h, o) = h_1, \ell_o \qquad \ell_1 \notin NativeFuns \qquad h_1(\ell_1).@body_{\{\triangle\}} = \mathsf{function}(x)\{s\} \\ o_s = \left\{ \begin{array}{ll} @scope_{\{\triangle\}} : & h_1(\ell_1).@fscope \\ @prototype_{\{\triangle\}} : & null \\ @this_{\{\triangle\}} : & l_o \\ \text{“}x\text{”}_{\{\triangle\}} : & v \end{array} \right\} \qquad \text{Alloc}(h_1, o_s) = h_2, \ell_s \qquad \text{VD}(h_2, \ell_s, s, \triangle) = h_3}{(\square, h, \ell, \mathbf{C}[\mathsf{new}\ \ell_1(v)]) \rightarrow (\triangle, h_3, \ell_s, \mathbf{C}[@\mathsf{NewExe}(\ell_o, \ell, s, \square)])}$$

DNewContext

$$\frac{(\square, h, \ell_s, s) \rightarrow (\square, h', \ell'_s, s')}{(\square, h, \ell_s, \mathbf{C}[@\mathsf{NewExe}(\ell_o, \ell, s, \triangle)]) \rightarrow (\square, h', \ell'_s, \mathbf{C}[@\mathsf{NewExe}(\ell_o, \ell, s', \triangle)])}$$

DNewFini

$$(\square, h, \ell_s, \mathbf{C}[@\mathsf{NewExe}(\ell_o, \ell, v, \triangle)]) \rightarrow (\triangle, h, \ell, \mathbf{C}[\ell_o])$$

Figure 8: Decorated Semantics Rules (Core JS)

DFun

$$\dfrac{\begin{array}{l} p = \{@prototype_{\{\square\}} : \#objprot\} \qquad \text{Alloc}(h, p) = h_1, \ell_1 \\ o = \left\{\begin{array}{ll} \text{“}prototype\text{”}_{\{\square\}} : & \ell_1 \\ @prototype_{\{\square\}} : & \#funprot \\ @call_{\{\square\}} : & true \\ @fscope_{\{\square\}} : & \ell \\ @body_{\{\square\}} : & \mathsf{function}(x)\{s\} \end{array}\right\} \qquad \text{Alloc}(h_1, o) = h', \ell' \end{array}}{(\square, h, \ell, \mathbf{C}[\mathsf{function}(x)\{s\}]) \xrightarrow{\varepsilon} (\square, h', \ell, \mathbf{C}[\ell'])}$$

DTypeof

$$\dfrac{\text{GetType}(h, v) = m}{(\square, h, \ell, \mathbf{C}[\mathsf{typeof}\ v]) \xrightarrow{\varepsilon} (\square, h, \ell, \mathbf{C}[m])}$$

DOP

$$\dfrac{v_1\ op\ v_2 = v}{(\square, h, \ell, \mathbf{C}[v_1\ op\ v_2]) \xrightarrow{\varepsilon} (\square, h, \ell, \mathbf{C}[v])}$$

DAsgnIdent

$$\dfrac{\begin{array}{c} \text{Scope}(h, \ell, \text{“}x\text{”}) = \ell_n \qquad \ell_g = \text{GetGlobal}(h, \ell) \\ h_1 = \begin{cases} h(\ell_g.\text{“}x\text{”}_{\{\square\}} = v) & \text{if } \ell_n = null \\ h(\ell_n.\text{“}x\text{”} = v) & \text{otherwise} \end{cases} \end{array}}{(\square, h, \ell, \mathbf{C}[x = v]) \xrightarrow{\varepsilon} (\square, h_1, \ell, \mathbf{C}[v])}$$

DAsgn-New-Property

$$\dfrac{m \notin h(\ell_1) \qquad h(\ell_1.m_{\{\square\}} = v) = h_1}{(\square, h, \ell, \ell_1[m] = v, Q) \xrightarrow{\varepsilon} (\square, h_1, \ell, v, Q)}$$

DModify-Property

$$\dfrac{m_{\{\square\}} \in h(\ell_1) \qquad h(\ell_1.m_{\{\square\}} = v) = h'}{(\triangle, h, \ell, \ell_1[m] = v, Q) \xrightarrow{\varepsilon} (\triangle, h', \ell, v, Q)}$$

DGetVProp

$$\dfrac{\begin{array}{c} \text{Prototype}(h, \ell, m) = \ell_2 \\ v = \begin{cases} undefined & \text{if } \ell_2 = null \\ h(\ell_2).\text{“}x\text{”} & \text{otherwise} \end{cases} \end{array}}{(\square, h, \ell, \mathbf{C}[\mathbf{C_v}[\ell_1[m]]]) \xrightarrow{\varepsilon} (\square, h, \ell, \mathbf{C}[\mathbf{C_v}[v]])}$$

DGetVIdent

$$\dfrac{\text{Scope}(h, \ell, \text{“}x\text{”}) = \ell_1 \neq null \qquad v = h(\ell_1).\text{“}x\text{”}}{(\square, h, \ell, \mathbf{C}[\mathbf{C_i}[x]]) \xrightarrow{\varepsilon} (\square, h, \ell, \mathbf{C}[\mathbf{C_i}[v]])}$$

DParse

$$\dfrac{\ell_1 = \#Parse \text{ or } \ell_1 = \#Parse_f \qquad o = \mathsf{parse}(m) \qquad \text{Alloc}(h, o) = h_1, \ell_o}{(\square, h, \ell, \mathbf{C}[\ell_1(m)]) \xrightarrow{\varepsilon} (\square, h, \ell, \mathbf{C}[\ell_o])}$$

DStringify

$$\dfrac{\begin{array}{c} \ell_1 = \#Stringify \text{ or } \ell_1 = \#Stringify_f \\ m = \mathsf{stringify}(h, v) \end{array}}{(\square, h, \ell, \mathbf{C}[\ell_1(v)]) \xrightarrow{\varepsilon} (\square, h, \ell, \mathbf{C}[m])}$$

DPostMsg

$$\dfrac{\ell_1 = \#Postmessage \text{ or } \ell_1 = \#Postmessage_f}{(\square, h, \ell, \mathbf{C}[\ell_1(m)]) \xrightarrow{m} (\square, h, \ell, \mathbf{C}[undefined])}$$

DAddListener

$$\dfrac{\ell_1 = \#Addlistener \text{ or } \ell_1 = \#Addlistener_f}{(\square, h, \ell, \mathbf{C}[\ell_1(\ell_i)]) \xrightarrow{\ell_i} (\square, h_1, \ell, \mathbf{C}[undefined])}$$

Figure 9: Decorated Semantics Rules (core JS, continued)

DVar
$$(\Box, h, \ell, \mathsf{var}\ x) \xrightarrow{\varepsilon} (\Box, h, \ell, \mathit{undefined})$$

DBlockNext
$$(\Box, h, \ell, v\ s*) \xrightarrow{\varepsilon} (\Box, h, \ell, s*)$$

DBlockContext
$$\frac{(\Box, h, \ell, s_0) \xrightarrow{lmq} (\Box, h', \ell, s_1)}{(\Box, h, \ell, s_0; s) \xrightarrow{lmq} (\Box, h, \ell, s_1; s)}$$

DIfTrue
$$\frac{(\Box, h, \ell, e) \xrightarrow{lmq}{}^{*} (\Box, h', \ell, \mathit{true})}{(\Box, h, \ell, \mathsf{if}\ (e)\ s_0\ \mathsf{else}\ s_1) \xrightarrow{lmq} (\Box, h', \ell, s_0)}$$

DIfFalse
$$\frac{(\Box, h, \ell, e) \xrightarrow{lmq}{}^{*} (\Box, h', \ell, \mathit{false})}{(\Box, h, \ell, \mathsf{if}\ (e)\ s_0\ \mathsf{else}\ s_1) \xrightarrow{lmq}{}^{*} (\Box, h, \ell, s_1)}$$

DWhileTrue
$$\frac{(\Box, h, \ell, e) \xrightarrow{lmq} (\Box, h', \ell, \mathit{true})}{(\Box, h, \ell, \mathsf{while}\ (e)\ s) \xrightarrow{lmq} (\Box, h', \ell, s\ \mathsf{while}\ (e)\ s)}$$

DWhileFalse
$$\frac{(\Box, h, \ell, e) \xrightarrow{lmq} (\Box, h', l, \mathit{false})}{(\Box, h, \ell, \mathsf{while}\ (e)\ s) \xrightarrow{lmq} (\Box, h', \ell, \mathit{undefined})}$$

DReturn
$$\frac{(\Box, h, \ell, e) \xrightarrow{lmq} (\Box, h', \ell, v)}{(\Box, h, \ell, \mathsf{return}\ e; s) \xrightarrow{\varepsilon} (\Box, h', \ell, \mathsf{return}\ v)}$$

Figure 10: Decorated Semantics Rules (core JS, continued)

context. Evaluation context $\mathbf{C_v}$ is special for a redex in the form of $\ell[m]$ (a property accessor) to be evaluate to a value. For example, in the expression $\ell[m](e)$, $\ell[m]$ should not be evaluated into a value since it is a method invocation. Therefore $_(e)$ is not a $\mathbf{C_v}$ context.

Now we explain the transition rules in detail.

DThis To resolve the this keyword, we return the @*this* property of the current scope object ℓ in h.

DObj-Literal We first allocate a new empty object in h, represented by ℓ_o. Then we evaluate each e_i separately, adding the results v_i as properties m_i of the object in ℓ_o[2]. The result is the location ℓ_o of created object. The properties of the created object is all decorated with $\Box$.

DCallFunc To invoke a function, we create a new scope object ℓ_s as current scope object in which the @*scope* property is set to the function's closure scope $h_1(\ell_1)$.@*fscope*. Furthermore, the @*this* property of ℓ_s is set to ℓ_g, the global scope object of the current scope chain. $\mathrm{VD}(h_1, \ell_s, s, \triangle)$ initializes local variables defined in the body of the function in ℓ_1 with the decoration of the function object rather than the current decoration in the configuration. The resulting expression @FunExe($\ell, s, \Box$) keeps record of the scope object ℓ to return, and the decoration $\Box$ to recover when the function execution finishes.For simplicity, we present functions with one parameter only. Function GetGlobal look up in the scope chain to get the address of the global object via the window property.

DCallMethod This rule is similar to DCallFunc, the different is that we look up thought the prototype chain to obtain the function object ℓ_3 from $\ell_1[m]$, and we set the @*this* property of ℓ_s is set to ℓ_1, since it is a method call rather than a function call.

DCallContext This is a contextual rule for evaluating a body of a function.

DCallFini When a function invocation is finished and no value is returned, we restore the scope to ℓ and return *undefined* as result.

DCallRet When a function invocation is finished and value v is returned, we restore the scope to ℓ and return v as result.

DNew The new construct uses a function as a constructor to initialize an object. It behaves method invocation. We first creates an empty object ℓ_o in which the internal @*prototype* property is set to the "*prototype*" property of the function $h\ell_1$. Then we proceed as in method invocation. The result expression @NewExe($\ell_o, \ell, s, \Box$) keep records of both ℓ_o and ℓ.

[2]We do not explicit mention the heap h when there is no ambiguity.

DNewContext This is a contextual rule for evaluating a body of a function as object initialization.

DNewFini When an execution of @NewExe(ℓ_o, ℓ, v) finished, we restore the scope object to ℓ and return ℓ_o as the result of creating an object.

DFun To create a new function, we first create an empty prototype object ℓ_p, then we create ℓ' as the function object, where the "*prototype*" property is set to ℓ_p. We keep the current scope object ℓ in the @*fscope* property of ℓ' as the closure captured by the function definition. We finally returnℓ' as result. The function object is decorated with $\square$, keeping track the owner of the function.

DTypeof We use the $GetTypeh, v$ to obtain a string representing the type of v.

DOP We use a conventional interpretation of *op*.

DAsgnIdent We first look up through the scope chain to check if x is defined in the chain. If it exist in scope object ℓ_n , then we update the property of "x" in ℓ_n, otherwise we create a property "x" in the global object ℓ_g.

DAsgn-New-Property To create a property m of ℓ_1, we directly update ℓ_1 in h without following the prototype chain. A decoration is created only when a new property is created and cannot be changed afterward.

Example 5 (Decoration of a new property). Suppose a location ℓ in variable x points to object o. A program, with decoration V, $x[\text{“}b\text{”}] = 3$ results in the decorated object on the right side:

$$o = \left\{ \begin{array}{ll} a_{\{V\}}: & 2 \\ c_{\{\spadesuit\}}: & 4 \end{array} \right\} \Rightarrow \left\{ \begin{array}{ll} a_{\{V\}}: & 2 \\ b_{\{V\}}: & 3 \\ c_{\{\spadesuit\}}: & 4 \end{array} \right\}$$

DModify-Property It is similar to DAsgn-New-Property. The color of the property in the heap is not changed.

DGetVProp To access a property of an object, we look up through the prototype chain. The value v could possibly be an location. When the property m does not exist we return *undefined*.

DGetVIdent To resolve a variable name, we look up through the scope chain.

DParse To de-marshal an object we use parse(m) to reconstruct an object o. Note that it is a special rule for a native function.

DStringify To marshal an object we use stringify(o) to return the string (in JSON format) representation of the object.

DPostMsg To post a message m, we use a label m to indicate the side-effect.

DAddListener To set a event listener ℓ_i, we use a label ℓ_i to indicate the side-effect.

DVar Since var x has already been treated by VD before statement execution, we just skip this statement.

DBlockNext When a statement evaluates to a value, we continue with the next one.

DBlockContext It is a contextual rule for evaluating sequential composition of statements (block).

DIfTrue If the condition expression e evaluates to *true* then we execute the "*then*" branch.

DIfFalse If the condition expression e evaluates to *false* then we execute the "*else*" branch.

DWhileTrue If the condition expression e evaluates to *true* then we unfold the while body s once.

DWhileFalse If the condition expression e evaluates to *true* then we skip the while body.

DReturn For return statement, we skip all the rest of the statement s.

Example 6 (Decorated Global Object). Recall variables `secret` and `steal` in Listing 4 and 3 of Section 2. Assuming that the secret input is "yes", and after execution of the non-benign gadget, the shared global object has the following form:

$$h(\#global) = \left\{ \begin{array}{ll} & \vdots \\ \text{“}price\text{”}_{\{\spadesuit\}}: & 0 \\ \text{“}secret\text{”}_{\{\spadesuit\}}: & \text{“}yes\text{”} \\ \text{“}steal\text{”}_{\{V\}}: & \text{“}yes\text{”} \end{array} \right\}$$

If the gadget is sandboxed as in Listing 5, the gadget code gets stuck by the semantics when trying to read secret since the variable has not been defined. (In practice, however, the program raises an exception that we do not model in the semantics.)

5 Correctness Theorem

In this section we formally present the correctness theorem and its hypotheses. In order to state the theorem, we define decorations for original and compiled mashups. In the original mashup we decorate the integrator as ♠ and the gadget as V.

Definition 5 (Decorated Original Mashup)**.** Let P_i be an integrator script and P_g be a gadget script. We define the original mashup $\tilde{M}(P_i, P_g)$ to be:

```
<html>
    <scriptV> P_g </script>
    <script♠> P_i </script>
</html>
```

In the compiled mashup we decorate the run-time libraries as W. The run-time libraries are marked as neutral colour W since we show with the correctness theorem that the integrator's heap is preserved in the original and compiled version. The runtime libraries do not appear in the original heap.

Definition 6 (Mashic Compilation)**.** Let P_i be an integrator script, P_g be a gadget script, $\mathcal{V}$ be a set of variables denoting global names exported by the gadget script, we define the Mashic compilation $\tilde{M}_c(P_i, P_g, \mathcal{V})$ to be:

```
<html>
    <iframe src=u></iframe>
    <scriptW> P_p </script>
    <scriptW> Bootstrap_i^V </script>
    <script♠> C⟨P_i⟩(function(_x){_x}) </script>
</html>
```

where

```
           <scriptW> P_l </script>
Web(u) =   <scriptV> P_g </script>
           <scriptW> Bootstrap_g^V </script>
```

We formally define the bootstrapping script that appears in Section 2.

Definition 7 (Bootstrapping Script)**.** Given a set of variables $\mathcal{V} = \{x_0, \ldots, x_n\}$, the Mashic bootstrapping script for an integrator $Bootstrap_i^{\mathcal{V}}$ is defined, for $i \in \{0 \ldots n\}$, as

$$\mathsf{var}\ x_i; x_i = \mathsf{new}\ OHandle(i);$$

and the bootstrapping script for a gadget $Bootstrap_g^{\mathcal{V}}$ is:

$$add_handle_obj(\mathsf{new}\ OHandle(i), x_i);$$

5.1 Assumptions

It is useful for the assumptions to define the notion of simple object. The native stringify function in HTML5 converts an object into JSON-formatted string, ignoring the graph structure of the object. For example the following object o in h:

$$o = \{a : \ell, b : \ell\} \qquad\qquad h(\ell) = \{c : 2\}$$

will result in the below JSON-formatted string:

`{a:{c:2},b:{c:2}}`

in which the original graph structure is not preserved.

Simple objects are objects that have a tree structure preserved by stringify. We further simplify this definition to an object with all properties being primitive values for ease of presentation:

Definition 8 (Simple Object)**.** An object o is simple if:

1. $o.@prototype = null$;
2. $\forall i \in \mathsf{dom}(o)\ o.i_{\{\square\}} = pv$ for some primitive value pv.

Example 7 (Simple Object). o_1 is a simple object but o_2 is not.

$$o_1 = \left\{ \begin{array}{ll} a: & 3 \\ b: & \mathit{true} \\ @\mathit{prototype}: & \mathit{null} \end{array} \right\} \qquad o_2 = \left\{ \begin{array}{ll} c: & 3 \\ d: & \ell \\ @\mathit{prototype}: & \ell' \end{array} \right\}$$

Mashic compilation may produce different results from the original mashup if a non-simple object is used as parameter when calling to gadget-defined functions, as shown below:

Example 8 (Simple object). Consider a non-simple object $h(\ell)$:

$$h(\ell) = \left\{ \begin{array}{ll} \text{“}a\text{”} & \ell' \\ \text{“}b\text{”} & \ell' \\ \text{“}c\text{”} & \mathit{true} \end{array} \right\} \qquad h(\ell') = \{a:3\}$$

By semantics of parse and stringify, in the Mashic compilation, the following object is obtained on the gadget side:

$$g(\ell) = \left\{ \begin{array}{ll} \text{“}a\text{”} & \ell_1 \\ \text{“}b\text{”} & \ell_2 \\ \text{“}c\text{”} & \mathit{true} \end{array} \right\} \qquad g(\ell_1) = \{a:3\} \qquad g(\ell_2) = \{a:3\}$$

If the gadget compares $\ell[\text{“}a\text{”}] === \ell[\text{“}b\text{”}]$, it gets true in the original mashup and false in the compiled one.

We now turn to the hypotheses required for correctness.

Benign Gadget Intuitively, a benign gadget P_g does not modify the integrator's portion (marked by ♠) and the neutral portion (marked by W) of the heap. Furthermore the evaluation of P_g does not depend on any part of the heap except for the initial heap.

In order to state the assumption we first define a benign gadget heap as a heap that contains gadget functions with confidentiality and integrity properties.

Definition 9 (Benign Gadget Heap)**.** A heap h_g is benign if and only if for any h_i $(i \in \{0,1\})$ such that $h_i\lfloor_{\mathrm{V}} = h_g$, for any function located in $\ell \in \mathsf{dom}(h_g)$, for any ℓ' such that $h_0(\ell') = h_1(\ell')$ is a simple object, and $(\spadesuit, h_i, \ell_i, \ell(\ell')) \rightarrow^* (\spadesuit, h_i', \ell_i', v_i')$, the following conditions holds:

1. $v_0' = v_1'$;
2. (integrity) $h_i =_{\spadesuit} h_i'$ and $h_i =_{\mathrm{W}} h_i'$;
3. (confidentiality) $h_0'\lfloor_{\mathrm{V}} = h_1'\lfloor_{\mathrm{V}}$;
4. (preservation of benignity) $h_1'\lfloor_{\mathrm{V}}$ is benign

Example 9 (Benign Heap). Recall the integrator's code in Listing 3 in Section 2. If the gadget contains the following code, then the gadget will not produce a benign gadget heap:

```
1 var rungadget;
2 rungadget = function(x){
3              var steal;
4              steal = secret;
5              price = 0;
6 };
```

Listing 14: Non-benign Gadget Heap

The gadget defines a function in the heap which tries to read from the global variable `secret` and tries to write into the global variable `price`. Calling the function from the integrator will violate the integrity and confidentiality requirement.

Assumption 2 (Benign Gadget)**.** *Program P_g is benign if and only if for any heaps h_i $(i \in \{0,1\})$ such that $h_i\lfloor_{\mathrm{V}} = \emptyset$ and $(\mathrm{V}, h_i, \ell, P_g) \rightarrow^* (\mathrm{V}, h_i', \ell, v_i)$, the following conditions hold:*

1. *(integrity) $h_i =_{\spadesuit} h_i'$ and $h_i =_{\mathrm{W}} h_i'$;*
2. *(confidentiality) $h_0' =_{\mathrm{V}} h_1'$;*

3. *$h'_0\lfloor_{\mathrm{V}}$ is benign.*

Example 10 (Benign Gadget Example). Recall the example in Section 2, Listing 4. The gadget is not benign since it tries to read from the global variable `secret` and tries to write into the global variable `price`.

In the benign gadget assumption we explicitly require that the initialization phase (adding functions to the heap) and execution of all functions (that are defined in the heap) always terminate.

It is possible to relax this assumption by not requiring termination of benign gadgets (by using indistinguishability invariants for intermediate running expressions) but we consider more appropriate to see non-terminating behaviour in gadgets as non benign behaviour since the gadget will never give the hand to the integrator to execute. Hence if the gadget is non-terminating we do not offer any correctness guarantees (security guarantees still apply).

Notice that the termination requirement on gadgets does not imply termination of the mashup. The mashup might never terminate if gadget and integrator continously run listener continuations and this is independent of termination of functions in gadgets (see e.g. fair termination Boudol [2010]).

Simple Integrator For correctness, we impose some reasonable restrictions on the integrators's code. Intuitively, a simple integrator does not modify directly a non-W-colored property; and does not use objects defined by gadgets in the prototype chain. This restriction is not very limiting in practice since integrator's usually operate on gadgets via the interfaces provided by it and not directly modifying its properties. We also require a simple integrator only sends *simple objects* to functions defined by gadgets. This assumption is a conservative hypothesis to get correctness using the native stringify function in the compiler.

Assumption 3 (Simple Integrator). *Let P_i be a JS program. P_i is a simple integrator, if and only if, for any benign heap h_g such that $(\spadesuit, h_{in} \oplus h_g, \#global, P_i) \rightarrow^* (\spadesuit, h, \ell, \mathbf{C}[e])$, the following conditions hold:*

1. *If e is of the form $x = v$ and $\mathrm{Scope}(h, \ell, \text{“}x\text{”}) = \ell_n$, then either $\ell_n = null$ or “x” is a $\spadesuit$-colored property of $h(\ell_n)$.*
2. *If e is of the form $\ell'[m] = v$, then $h(\ell')$ is a $\spadesuit$-single-colored object.*
3. *For any ℓ such that $\ell \in \mathsf{dom}(h\lfloor_{\spadesuit})$ and $h(\ell).@prototype = \ell_n$, either $\ell_n = null$ or $h(\ell_n)$ is a $\spadesuit$-colored object.*
4. *If e is of the form $\ell_f(\ell')$ and $h(\ell_f)$ is a V-colored function, then $h(\ell')$ is a simple object.*

Example 11 (Simple integrator prototype chain). We illustrate why an integrator's object can not have a gadget's object as its prototype object (bullet 3). Assume that in the heap of the original mashup, $h(\ell_i)$ is a $\spadesuit$-colored object, and $h(\ell_g)$ is a V-colored object such that

$$h(\ell_i) = \{@prototype : \ell_g\} \qquad\qquad h(\ell_i) = \{a : 3\}$$

By reading the property “a” of ℓ_i in the original mashup we get 3. The heap of the compiled code will contain a pointer to a handle:

$$h(\ell_i) = \{@prototype : \ell_o\} \qquad\qquad h(\ell_o) = \left\{ \begin{array}{ll} \text{“_id”} & n \\ \text{“_is_o”} & true \end{array} \right\}$$

Hence, by reading the property “a” of ℓ_i in the compiled mashup we do not get 3.

5.2 Indistinguishability and Correctness

To define indistinguishability between the original heap and compiled heap, the structure of the scope chain in the heap must be preserved. We start by defining the notion of scope object and scope chain. We use $\#global$ as the address of the original global object.

Definition 10 (Scope Object). Let h be a heap, and ℓ be a location for a scope object. We say ℓ is a scope object in h if one of the following conditions is satisfied:

1. $\ell = \#global$, and $h(\ell).@scope = null$;
2. $\ell \neq \#global$, $h(\ell).@scope = \ell' \neq null$, and ℓ' is also a scope object in h.

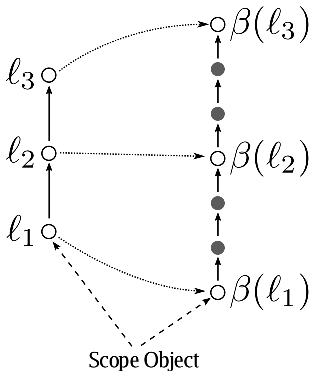


Figure 11: Scope Indistinguishability

Definition 11 (Scope Chain)**.** Let h be a heap, and ℓ_1 be a scope object in h, we say that $\ell_1\ell_2\dots\ell_n$ is the scope chain of ℓ_1 in h, if

1. For $i < n$, $h(\ell_i).@scope = \ell_{i+1}$;
2. $h(l_n).@scope = null$

We use $\ell \in \ell_1\ell_2\dots\ell_n$ to denote that scope object ℓ is included in the scope chain $\ell_1\ell_2\dots\ell_n$ w.r.t some heap h. We define the β-indistinguishability $\sim_\beta$ on values, objects, and scope chains, where $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ is a partial injective function between heap locations $\mathcal{L}$.

Definition 12 (Scope Chain Indistinguishability)**.** Let ℓ_1 be a scope object in h and ℓ'_1 be a scope object in h', and $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ be a partial injective function. Let $\ell_1\ell_2\dots\ell_n$ be the scope chain of ℓ_1 in h, let $\ell'_1\ell'_2\dots\ell'_m$ be the scope chain of ℓ'_1 in h', we say that the two scope chains are indistinguishable, denoted $(h, \ell_1)\approx_\beta(h', \ell'_1)$ if and only if:

1. $\beta(\ell_1)\beta(\ell_2)\dots\beta(\ell_n)$ is a sub-sequence of $\ell'_1\ell'_2\dots\ell'_m$;
2. for $\ell \notin \beta(\ell_1)\beta(\ell_2)\dots\beta(\ell_n)$, and $\ell \in \ell'_1\ell'_2\dots\ell'_m$, $\forall i \in \mathsf{dom}(h'(\ell))$,
   $i \in \{@scope, @prototype, @this, \text{"_k"}, \text{"_l"}, \text{"_m"}, \text{"_}x_i\text{"}\}$

The intuition of scope indistinguishability is that even if scope chains do not have a one to one correspondence, the structure of scope chains is preserved by the CPS transformation, as illustrated in Figure 11. In the figure, scope objects are represented by round points, and the solid arrows represent the scope chain. The scope chain on the left is obtained by a normal execution of an integrator code. The scope chain on the right is obtain by an execution of the corresponding CPS-transformed code, where there are more CPS-administrative scope objects (gray-colored in the figure). The scope indistinguishability does not take into consideration those CPS-administrative scope objects. The only constraint is that administrative scope objects should only contain properties listed in bullet 2.

Two values are indistinguishable if they are equal or if they are both locations related by β. Even assuming a deterministic allocator, we need β to relate two heaps since objects created in the original mashup and compiled mashup will be necessarily different since the compiled heap will contain more objects.

Definition 13 (Value Indistinguishability)**.** Let v_1 and v_2 be two values, and $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ be a partial injective function, value indistinguishability is defined as follows:

$$\frac{v \notin \mathcal{L}}{v \sim_\beta v} \qquad\qquad \frac{v_1, v_2 \in \mathcal{L} \qquad \beta(v_1) = v_2}{v_1 \sim_\beta v_2}$$

Objects are related if they have the same properties with the same values. Exceptions are properties $\{@scope, @fscope, @this\}$ and function objects. Properties $\{@scope, @fscope\}$ are related via the scope chain indistinguishability as explained above. Function objects are indistinguishable if the $@body$ property contains the same code in its original and compiled form.

Definition 14 (Object indistinguishability)**.** Let o_1 and o_2 be two objects, and $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ be a partial injective function, we say $o_1 \simeq_\beta o_2$, if for every $i \in \mathsf{dom}(o_1)$ one of the following holds:

1. $i \in \{@scope, @fscope, @this\}$;
2. $i \notin \{@body, @scope, @fscope, @this\}$ and if $o_1.i \in \mathsf{dom}(\beta)$ then $o_1.i \sim_\beta o_2.i$;
3. $i = @this$ then $o_1.@this \sim_\beta o_2.$"_this";
4. $i = @body$ then $o_1.@body = \mathsf{function}(x)\{s\}$, then $@body \in \mathsf{dom}(o_2)$ and $o_2.@body = \mathsf{function}(_fun_cont, x)\{s_{cps}\}$, where

$$s_{cps} \;=\; \begin{array}{l} \mathsf{var}\ _this; \\ _this = \mathsf{this}; \\ (\mathcal{C}'\langle s\rangle)(_fun_cont) \end{array}$$

We give an examples illustrating the object indistinguishability.

Example 12 (Object indistinguishability). Let o_1, o_2 o_3 be:

$$o_1 = \left\{ \begin{array}{ll} a: & 2 \\ b: & \ell_1 \\ @scope: & \ell_2 \end{array} \right\} \qquad o_2 = \left\{ \begin{array}{ll} a: & 2 \\ b: & \beta(\ell_1) \\ @scope: & \ell'_2 \end{array} \right\} \qquad o_3 = \left\{ \begin{array}{ll} a: & 2 \\ b: & \ell'_1 \\ @scope: & \ell'_2 \end{array} \right\}$$

If $\ell'_1 \neq \beta(\ell_1)$ and $\ell_2 \neq \ell'_2$, then we have $o_1 \simeq_\beta o_2$ and $o_1 \not\simeq_\beta o_3$. We do not compare $@scope$ property between o_1 and o_2; but we do compare property b between o_2 and o_3.

Finally, heaps are indistinguishable if all objects are indistinguishable and respect scope chains are indistinguishable.

Definition 15 (Heap indistinguishability)**.** We say that (h_1, ℓ_1) and (h_2, ℓ_2), are indistinguishable with respect to $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ with $\mathsf{dom}(\beta) = \mathsf{dom}(h_1)$ and $\mathsf{rng}(\beta) \subseteq \mathsf{dom}(h_2)$, denoted $(h_1, \ell_1) \simeq_\beta (h_2, \ell_2)$, if and only if:

1. $h_1(\ell) \simeq_\beta h_2(\beta(\ell))$ for every $\ell \in \mathsf{dom}(\beta)$
2. if $\ell \in \mathsf{dom}(\beta)$ and $h_1(\ell)$ has the $@body$ property, then $(h_1, h_1(\ell).@fscope) \approx_\beta (h_2, h_2(\beta(\ell)).@fscope)$
3. $(h_0, \ell_0) \approx_\beta (h_1, \ell_1)$.

The correctness theorem gives strong guarantees if the gadget is benign: behavior of original and compiled mashup are equivalent in terms of the integrator's heap. If the gadget is not benign there are no correctness guarantees but only security guarantees described in the following section. We use in the hypothesis that integrator and gadget do not declare the same variables $\mathsf{var}(P_i) \cap \mathsf{var}(P_g) = \emptyset$, where var is defined by:

$$\mathsf{var}(s) = \begin{cases} \emptyset & \text{if } s = e \text{ or } s = \mathsf{return}\ e \\ \mathsf{var}(s_0) \cup \mathsf{var}(s_1) & \text{if } s = s_0; s_1 \text{ or } s = \mathsf{if}\ (e)\ s_0\ \mathsf{else}\ s_1 \\ \mathsf{var}(s) & \text{if } s = \mathsf{while}\ (e)\ s \\ \{x\} & \text{if } s = \mathsf{var}\ x \end{cases}$$

Notice that this definition of var refers only to declared variables, and the hypothesis does not assume that integrator and gadget do not share variables.

Theorem 1 (Correctness)**.** *Let P_i be a simple integrator and P_g be a benign gadget such that $\mathsf{var}(P_i) \cap \mathsf{var}(P_g) = \emptyset$. If $\langle \spadesuit, \varepsilon, null, \tilde{M}(P_i, P_g), Q_{init}\rangle_I \rightarrow^* \langle \Box, h_0, \ell_0, \varepsilon, Q_{init}\rangle_x$ then,*

$$\langle \spadesuit, \varepsilon, null, \tilde{M}_c(P_i, P_g, \mathcal{V}), Q_{init}\rangle_I \rightarrow^* \langle \Box, h_1, \ell_1, \varepsilon, Q_1\rangle_x$$

where Q_1 has no message waiting, and there exists β such that

$$(h_1\lfloor_\spadesuit, \ell_1) \simeq_\beta (h_0\lfloor_\spadesuit, \ell_0)$$

The proof proceeds by the help of an intermediate compilation where the integrator is CPS-transformed but the gadget is not sandboxed. We show, by structural induction on statements and expressions, that the behaviour of the original mashup is indistinguishable from that of the intermediate compilation; and the intermediate compilation behaves indistinguishably from the mashic compilation.

```
new  e0(e1)
--------------------------------------------------
function(_k){
  C'⟨e0⟩(function(_x0){
    C'⟨e1⟩(function(_x1){
      if (_x0["_is_cps"]){
        var _x2, _x3, _x4;
        _x3 = function(x){};
        _x3["prototype"] = _x0["prototype"];
        _x2 = new _x3();
        _x4 = function(_v){_k(_x2);};
        _x0["apply"](_x2, _x4, _x1);
      } else {
        _k(new _x0(_x1));
      }});});}
```

```
e0[e1](e2)
--------------------------------------------------
function(_k){
  C'⟨e0⟩(function(_x0){
    C'⟨e1⟩(function(_x1){
      C'⟨e2⟩(function(_x2){
        if (_x0[_x1]["_is_cps"]){
          _x0[_x1](_k, _x2);
        } else {
          _k(_x0[_x1](_x2));
        }});});});}
```

```
e0[e1]
--------------------------------------------------
function(_k){
  C'⟨e0⟩(function(_x0){
    C'⟨e1⟩(function(_x1){
      _k(_x0[_x1]);
    });});}
```

```
e0(e1)
--------------------------------------------------
function(_k){
  C'⟨e0⟩(function(_x0){
    C'⟨e1⟩(function(_x1){
      if (_x0["_is_cps"]){
        _x0(_k, _x1);
      } else {
        _k(_x0(_x1));
      }});});}
```

```
e0[e1] = e2
--------------------------------------------------
function(_k){
  C'⟨e0⟩(function(_x0){
    C'⟨e1⟩(function(_x1){
      C'⟨e2⟩(function(_x2){
        _k(_x0[_x1] = _x2);
      });});});}
```

Figure 12: CPS Transformation (Intermediate)

5.3 Auxiliary Definitions and Lemmas

In this section we give some auxiliary definitions and some useful lemmas to prove the main theorem. First we define the notion of intermediate compilation which the gadget is not sandboxed by an iframe and the integrator is compiled to CPS-only code without using the proxy and listener interface. The intermediate compilation will be used in the proofs.

Definition 16 (Decorated Intermediate Compilation)**.** We define decorated intermediate compilation $\tilde{M}_i(P_i, P_g)$ as the follows:

```
<html>
    <script♥> Pg </script>
    <script♠> C'⟨Pi⟩(function(_x){_x}) </script>
</html>
```

where $\mathcal{C}'\langle\rangle$ is an intermediate CPS-transformation.

The intermediate CPS transformation are identical to the Mashic CPS transformation except for the rules shown in Figure 12, where the proxy and the listener library are not used, since the gadget is not sandboxed.

The following lemma shows that the Mashic compilation and the intermediate compilation preserve simpleness of the integrator, since they will not introduce more behavior.

Lemma 4. *If P_i is a simple integrator, then $\mathcal{C}\langle P_i\rangle$ and $\mathcal{C}'\langle P_i\rangle$ are both simple integrators.*

The following lemma shows that the added decoration does not affect the semantics.

Lemma 5 (Erasing Decoration)**.** *The decorated semantics coincide with the original semantics by simply erasing all the decorations and operations on decorations.*

We define the notion of *strong object indistinguishability*, denoting $\sim_\beta$, where we have a stronger condition in item 2 comparing to object indistinguishability.

Definition 17 (Strong Object Indistinguishability)**.** Let o_1 and o_2 be two objects, and $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ be a partial injective function, we say $o_1 \sim_\beta o_2$, if for every $i \in \mathsf{dom}(o_1)$ one of the following holds:

1. $i \in \{@scope, @fscope, @this\}$;

2. $i \notin \{@body, @scope, @fscope, @this\}$ and if $o_1.i \in \mathsf{dom}(\beta)$ then $o_1.i \sim_\beta o_2.i$ otherwise $o_1.i = o_2.i$.
3. $i = @this$ then $o_1.@this \sim_\beta o_2.$"$_this$";
4. $i = @body$ then $o_1.@body = \mathsf{function}(x)\{s\}$, then $@body \in \mathsf{dom}(o_2)$ and $o_2.@body = \mathsf{function}(_fun_cont, x)\{s_{cps}\}$, where

$$s_{cps} = \begin{array}{l} \mathsf{var}\ _this; \\ _this = \mathsf{this}; \\ (\mathcal{C}'\langle s\rangle)(_fun_cont) \end{array}$$

Accordingly, we update the definition of *strong heap indistinguishability.*

Definition 18 (Strong Heap indistinguishability)**.** Two pairs of heap and scope object (h_1, ℓ_1) and (h_2, ℓ_2), are indistinguishable with respect to a partial injective function $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\mathsf{dom}(\beta) = \mathsf{dom}(h_1)$ and $\mathsf{rng}(\beta) \subseteq \mathsf{dom}(h_2)$, denoted $(h_1, \ell_1) \sim_\beta (h_2, \ell_2)$, if and only if:

1. for every $\ell \in \mathsf{dom}(\beta)$ with $o_1 = h_1(\ell)$ and $o_2 = h_2(\beta(\ell))$:
   (a) $o_1 \sim_\beta o_2$
   (b) if o_1 has the $@body$ property, then $(h_1, o_1.@fscope) \approx_\beta (h_2, o_2.@fscope)$
2. $(h_0, \ell_0) \approx_\beta (h_1, \ell_1)$.

The following lemma shows given two indistinguishable scope chain, the scope looking-up process will return indistinguishable heap location for any normal variable. By normal variable we mean variables that do not start with a "_". Special case applies to resolving the $@this$ identifier.

Lemma 6 (Scope look-up)**.** *Let h, g be two heaps, and ℓ, j be two locations for scope objects, if $h \sim_\beta g$ and $(h, \ell) \sim_\beta (g, j)$, then the following holds:*

1. *if $i \neq @this$, $\mathrm{Scope}(h, \ell, i) = \ell_1$ then $\mathrm{Scope}(g, j, i) = j_1$ and $\beta(\ell_1) = j_1$;*
2. *if $\mathrm{Scope}(h, \ell, @this) = \ell_1$ then $\mathrm{Scope}(g, j, \text{"}_this\text{"}) = j_1$ and $\beta(\ell_1) = j_1$;*

Proof. Straightforward by Definition 12 and Definition of $\mathrm{Scope}(_, _, _)$ in semantics rules. □

We formally define the shape of an opaque object handle.

Definition 19 (Opaque Object Handle)**.** Let o be an object, o is an opaque object handle with id n if and only if

$$o = \left\{ \begin{array}{ll} \text{"}_id\text{"} & n \\ \text{"}_is_o\text{"} & true \end{array} \right\}$$

We define a relation between two heaps up to a mapping from id of opaque object handles to heap locations. The intuition is that if in one heap, a property points to an opaque object handle, then in the other heap, it must points to a location corresponding to the opaque object handle by the mapping.

Definition 20. Let $f : \mathcal{N} \mapsto \mathcal{L}$ be a partial injective function from numbers to locations. We say that $h_c \stackrel{f}{=} h_i$ if

1. $h_c =_{\mathsf{V}} h_i$;
2. $\mathsf{dom}(h_c\lfloor_\spadesuit) = \mathsf{dom}(h_i\lfloor_\spadesuit)$;
3. $\forall \ell \in \mathsf{dom}(h_c\lfloor_\spadesuit), o_c = h_c(\ell)$ and $o_i = h_i(\ell)$, such that
   (a) if $o_c.i_{\{\spadesuit\}} = \ell_o$, and $h_c(\ell_o)$ is an opaque object handle with id n, then $o_i.i = f(n)$;
   (b) otherwise $o_c.i_{\{\spadesuit\}} = o_i.i_{\{\spadesuit\}}$.

5.4 Proofs

The proof of the correctness theorem can be divided into two steps. First we prove the original mashup behave indistinguishable with the intermediate compilation, then we prove the intermediate compilation behave indistinguishable with the Mashic compilation.

Proof of Theorem 1. By Lemma 7,

$$\langle \spadesuit, \varepsilon, null, \tilde{M}_i(P_i, P_g), Q_{init}\rangle_I \to^* \langle \Box, h_i, \ell_i, \varepsilon, Q_{init}\rangle_x \tag{1a}$$

and there exist β such that

$$(h_i\lfloor_{\spadesuit}, \ell_i) \sim_\beta (h_0\lfloor_{\spadesuit}, \ell_0) \tag{1b}$$

By Lemma 12 and (1a),

$$\langle \spadesuit, \varepsilon, null, \tilde{M}_c(P_i, P_g, \mathcal{V}), Q_{init}\rangle_I \to^* \langle \Box, h_1, \ell_1, \varepsilon, Q_1\rangle_x$$

where Q_1 has no message waiting, and there exists f, and $h_1 \stackrel{f}{=} h_i$. Therefore by Definition 15 and (1b),

$$(h_1\lfloor_{\spadesuit}, \ell_1) \simeq_\beta (h_0\lfloor_{\spadesuit}, \ell_0)$$

$\Box$

In the following lemma we show that the original mashup behave indistinguishable with the intermediate compilation. Note that we actually prove a stronger statement by using the strong heap indistinguishability in the lemma.

Lemma 7 (Original Mashup to Intermediate Compilation)**.** *Let P_i a simple integrator and P_g be a benign gadget such that P_i does not conflict with P_g and $\mathcal{V} = \mathsf{var}(P_g)$, if $\langle \spadesuit, \varepsilon, null, \tilde{M}(P_i, P_g), Q_{init}\rangle_I \to^* \langle \spadesuit, h_0, \ell_0, \varepsilon, Q_{init}\rangle_x$, then $\langle \spadesuit, \varepsilon, null, \tilde{M}_i(P_i, P_g), Q_{init}\rangle_I \to^* \langle \spadesuit, h_i, \ell_i, \varepsilon, Q_{init}\rangle_x$ and there exists β such that $(h_0\lfloor_{\spadesuit}, l_0) \sim_\beta (h_i\lfloor_{\spadesuit}, l_i)$.*

Proof. Since in $\tilde{M}(P_i, P_g)$ and $\tilde{M}_i(P_i, P_g)$ the first script P_g is identical, by semantics rules we have

$$\langle \spadesuit, \varepsilon, null, \tilde{M}(P_i, P_g), Q_{init}\rangle_I \to^* \langle \spadesuit, h, \#global, P_i, Q_{init}\rangle_x$$
$$\langle \spadesuit, \varepsilon, null, \tilde{M}_i(P_i, P_g), Q_{init}\rangle_I \to^* \langle \spadesuit, h, \#global, \mathcal{C}'\langle P_i\rangle(\mathsf{function}(_x)\{_x\}), Q_{init}\rangle_x$$

where

$$(\mathrm{V}, h_{in}, \#global, P_g) \to^* (\mathrm{V}, h, \#global, v)$$

Since P_g is a benign gadget, by Assumption 2, we have

$$h\lfloor_{\spadesuit} = \emptyset \tag{2a}$$
$$h\lfloor_{\mathrm{W}} = h_{in} \tag{2b}$$
$$h\lfloor_{\mathrm{V}} \text{ is a benign gadget's heap.} \tag{2c}$$

By (2a), we have

$$(h\lfloor_{\spadesuit}, \#global) \sim_\beta (h\lfloor_{\spadesuit}, \#global) \tag{2d}$$

Without loss of generality, we assume that

$$\forall l \in \mathsf{dom}(h\lfloor_{\mathrm{V}}), h(l).@body_{\{\mathrm{V}\}} = \mathsf{function}(x)\{S\} \Leftrightarrow h(l).\text{``_is_o''} = true \tag{2e}$$

By (2c),(2d), (2e), Assumption 2, Assumption 3, and Lemma 9, we have

$$(\spadesuit, h, \#global, P_i) \to^* (\spadesuit, h_0, \ell_0, v_0) \tag{2f}$$
$$(\spadesuit, h, \#global, \mathcal{C}'\langle P_i\rangle(\mathsf{function}(_x)\{_x\})) \to^* (\spadesuit, h_1, \ell_1, \ell_f(v_1)) \tag{2g}$$
$$\exists \beta, (h_0\lfloor_{\spadesuit}, \ell_0) \sim_\beta (h_1\lfloor_{\spadesuit}, \ell_1) \tag{2h}$$

By semantics rules, Definition 15, (2g), and (2h), we have

$$(\spadesuit, h_1, \ell_1, \ell_f(v_1)) \to^* (\spadesuit, h_2, \ell_2, v_1) \tag{2i}$$
$$(h_0\lfloor_{\spadesuit}, \ell_0) \sim_\beta (h_2\lfloor_{\spadesuit}, \ell_2) \tag{2j}$$
$$(h_0\lfloor_{\spadesuit}, \#global) \sim_\beta (h_2\lfloor_{\spadesuit}, \#global) \tag{2k}$$

By semantics rules and (2j) we can conclude with

$$\langle \spadesuit, h, \#global, P_i, Q_{init}\rangle_x \rightarrow^* \langle \spadesuit, h_0, \#global, v_0, Q_{init}\rangle_x$$
$$\langle \spadesuit, h, \#global, \mathcal{C}'\langle P_i\rangle(\mathsf{function}(_x)\{_x\}), Q_{init}\rangle_x \rightarrow^* \langle \spadesuit, h_2, \#global, v_1, Q_{init}\rangle_x$$

□

We define a useful notion of compatibility between heaps to relate a normal heap with a corresponding heap generated by the CPS transformation.

Definition 21. Let h and g be two decorated heaps , ℓ and j be two heap locations, $(h, \ell) \preceq_\beta (g, j)$ if and only if:

1. $(h\lfloor_\spadesuit, \ell) \sim_\beta (g\lfloor_\spadesuit, j)$;
2. $h\lfloor_{\mathrm{V}} = g\lfloor_{\mathrm{V}}$, and they are both benign gadget's heaps;
3. $h\lfloor_{\mathrm{W}} = g\lfloor_{\mathrm{W}} = h_{in}$;

We use the syntactic sugar $\mathrm{Lookup}(h, \ell, x) = v$ in the proof to denote the following facts:

$$\mathrm{Scope}(h, \ell, x) = \ell' \neq null$$
$$h(\ell').\text{“}x\text{”} = v$$

Remark 8. We can always simplify transitions in the form of $(\Box, h, \ell, s) \rightarrow^* (\triangle, h', \ell', \mathbf{F}[s'])$ to the form of $(\Box, h, \ell, s) \rightarrow^* (\triangle, h', \ell', s')$.

The following lemma shows that, under certain conditions, for any statement, the intermediate CPS-transformation behaves the same as the original statement.

The proof of the above two lemmas follows a mutual structural inducntion on the definition of statements and definition of expressions.

Lemma 9 (CPS Transformation Lemma - Statements)**.** *Let h and g be two decorated heaps, ℓ and j be two heap locations, s be a statement. If there exists a β such that $(h, \ell) \preceq_\beta (g, j)$, then the following conditions hold.*

1. *(a) If $(\spadesuit, h, \ell, s) \xrightarrow{\varepsilon}{}^* (\spadesuit, h', \ell', v)$, then $(\spadesuit, g, j, \mathcal{C}'\langle s\rangle(\ell_f)) \xrightarrow{\varepsilon}{}^* (\spadesuit, g', j', \ell_f(v'))$;*
   (b) Otherwise, if $(\spadesuit, h, \ell, s) \xrightarrow{\varepsilon}{}^* (\spadesuit, h', \ell', \mathsf{return}\ v)$, and $\mathrm{Lookup}(g, j, _fun_cont) = \ell_c$ then $(\spadesuit, g, j, \mathcal{C}'\langle s\rangle(\ell_f)) \xrightarrow{\varepsilon}{}^* (\spadesuit, g', j', \ell_c(v'))$;
2. *there exists β' such that $(h', \ell') \preceq_{\beta'} (g', j')$ and $v \sim_{\beta'} v'$.*

Proof. The proof is by induction on the definition of CPS transformation $\mathcal{C}'\langle s\rangle$. For all cases, we have some common hypothesis:

$$(h, \ell) \preceq_\beta (g, j) \tag{3a}$$

By definition 21, (3a) implies

$$(h\lfloor_\spadesuit, \ell) \sim_\beta (g\lfloor_\spadesuit, j) \tag{3b}$$
$$h\lfloor_{\mathrm{V}} = g\lfloor_{\mathrm{V}} \text{ are benign gadget's heaps.} \tag{3c}$$
$$h\lfloor_{\mathrm{W}} = g\lfloor_{\mathrm{W}} = h_{in} \tag{3d}$$

Case $s = e$: Since $s = e$ is an expression, by semantics rules

$$(\spadesuit, h, \ell, e) \xrightarrow{\varepsilon}{}^* (\spadesuit, h', \ell', v) \tag{4a}$$

By Lemma 10, (3a) and (4a),

$$(\spadesuit, g, j, \mathcal{C}'\langle e\rangle(\ell_f)) \xrightarrow{\varepsilon}{}^* (\spadesuit, g', j', \ell_f(v')) \tag{4b}$$
$$\exists \beta'.(h', \ell') \preceq_{\beta'} (g', j') \tag{4c}$$
$$v \sim_{\beta'} v'. \tag{4d}$$

from which we can conclude this case.

Case $s = s_0; s_1$: By the definition of the CPS compiler,

$$\mathcal{C}'\langle s\rangle = \mathsf{function}(_k)\{\mathcal{C}'\langle s_0\rangle(f_0)\}$$
$$f_0 = \mathsf{function}(_v)\{\mathcal{C}'\langle s_1\rangle(_k)\}$$

Let us consider two different cases :

1. The evaluation of s_0 does not involve a return statement,that is

$$(\spadesuit, h, \ell, s_0) \xrightarrow{\varepsilon}{}^* (\spadesuit, h_0, \ell_0, v_0) \tag{5a}$$

On the CPS-transformation side, by semantics rules

$$(\spadesuit, g, j, \mathcal{C}'\langle s\rangle(\ell_f)) \xrightarrow{\text{Fun, Fun-Call, Fun}}{}^* (\spadesuit, g_0, j_0, \mathcal{C}'\langle s_0\rangle(\ell_{f_0})) \tag{5b}$$

$$\text{and } g_0(j_0) = \left\{\begin{array}{ll} @scope_{\{\spadesuit\}}: & j \\ @prototype_{\{\spadesuit\}}: & null \\ @this_{\{\spadesuit\}}: & \#global \\ "_k"_{\{\spadesuit\}}: & \ell_f \end{array}\right\} \tag{5c}$$

By Definition 21, (5b) and (5c),

$$(h, \ell) \preceq_\beta (g_0, j_0) \tag{5d}$$

By inductive hypothesis, (5d) and (5a),

$$(\spadesuit, g_0, j_0, \mathcal{C}'\langle s_0\rangle(\ell_{f_0})) \to^* (\spadesuit, g_1, j_1, \ell_{f_0}(v_0'))$$
$$(h_0, \ell_0) \preceq_{\beta'} (g_1, j_1)$$
$$v_0 \sim_{\beta'} v_0'$$

Let us consider two different sub-cases again:

(a) The evaluation of s_1 does not involve a return statement:

$$(\spadesuit, h_0, \ell_0, s_1) \xrightarrow{\varepsilon}{}^* (\spadesuit, h_1, \ell_1, v_1) \tag{5e}$$

On the CPS-transformation side, by semantics rules

$$(\spadesuit, g_1, j_1, \ell_{f_0}(v_0')) \xrightarrow{\text{Fun-Call}}{}^* (\spadesuit, g_2, j_2, \mathcal{C}'\langle s_1\rangle(_k)) \tag{5f}$$

$$\xrightarrow{\text{GetV-Var}}{}^* (\spadesuit, g_2, j_2, \mathcal{C}'\langle s_1\rangle(\ell_f)) \tag{5g}$$

$$\text{and } g_2(j_2) = \left\{\begin{array}{ll} @scope_{\{\spadesuit\}}: & j_0 \\ @prototype_{\{\spadesuit\}}: & null \\ @this_{\{\spadesuit\}}: & \#global \\ "_v"_{\{\spadesuit\}}: & v_0' \end{array}\right\} \tag{5h}$$

By Definition 21, (5f) and (5h),

$$(h_0, \ell_0) \preceq_\beta (g_2, j_2) \tag{5i}$$

By inductive hypothesis, (5i) and (5e),

$$(\spadesuit, g_2, j_2, \mathcal{C}'\langle s_1\rangle(\ell_f)) \to^* (\spadesuit, g_3, j_3, \ell_f(v_1'))$$
$$(h_1, \ell_1) \preceq_{\beta''} (g_3, j_3)$$
$$v_1 \sim_{\beta''} v_1'$$

(b) The evaluation of s_1 does involve a return statement:

$$(\spadesuit, h_0, \ell_0, s_1) \xrightarrow{\varepsilon}{}^* (\spadesuit, h_1, \ell_1, \mathsf{return}\ v_1) \tag{5j}$$

$$\text{Lookup}(g, j, _fun_cont) = \ell_c \tag{5k}$$

By (5h) and (5k)

$$\text{Lookup}(g_2, j_2, _fun_cont) = \ell_c \tag{5l}$$

By inductive hypothesis, (5i), (5j) and (5l),

$$(\spadesuit, g_2, j_2, \mathcal{C}'\langle s_1 \rangle(\ell_f)) \to^* (\spadesuit, g_3, j_3, \ell_c(v_1'))$$
$$(h_1, \ell_1) \preceq_{\beta''} (g_3, j_3)$$
$$v_1 \sim_{\beta''} v_1'$$

2. The evaluation of s_0 does involve a return statement

$$(\spadesuit, h, \ell, s_0) \xrightarrow{\varepsilon}^* (\spadesuit, h_0, \ell_0, \mathsf{return}\ v_1) \tag{5m}$$
$$\text{Lookup}(g, j, _fun_cont) = \ell_c \tag{5n}$$

By (5c) and (5n)

$$\text{Lookup}(g_0, j_0, _fun_cont) = \ell_c \tag{5o}$$

By inductive hypothesis, (5d), (5m) and (5o),

$$(\spadesuit, g_0, j_0, \mathcal{C}'\langle s_0 \rangle(\ell_f)) \to^* (\spadesuit, g_1, j_1, \ell_c(v_0'))$$
$$(h_0, \ell_0) \preceq_{\beta''} (g_1, j_1)$$
$$v_0 \sim_{\beta''} v_0'$$

Case $s = \mathsf{if}\ (e)\ s_0\ \mathsf{else}\ s_1$: By the definition of the CPS compiler,

$$\mathcal{C}'\langle s \rangle = \mathsf{function}(_k)\{\mathcal{C}'\langle e \rangle(f_0)\}$$
$$f_0 = \mathsf{function}(_v)\{\mathsf{if}\ (_v)\ \mathcal{C}'\langle s_0 \rangle(_k)\ \mathsf{else}\ \mathcal{C}'\langle s_1 \rangle(_k)\}$$

Let us consider two different cases:

1. The condition expression e evaluates to *true*:

$$(\spadesuit, h, \ell, e) \xrightarrow{\varepsilon}^* (\spadesuit, h_0, \ell_0, true) \tag{6a}$$

By semantics rules,

$$(\spadesuit, h, \ell, \mathsf{if}\ (e)\ s_0\ \mathsf{else}\ s_1) \xrightarrow{\varepsilon}^* (\spadesuit, h_0, \ell_0, s_0) \tag{6b}$$
$$\xrightarrow{\varepsilon}^* (\spadesuit, h_1, \ell_1, v) \tag{6c}$$

On the CPS-transformation side, by semantics rules

$$(\spadesuit, g, j, \mathcal{C}'\langle s \rangle(\ell_f)) \xrightarrow{\text{Fun, Fun-Call, Fun}}^* (\spadesuit, g_0, j_0, \mathcal{C}'\langle e \rangle(\ell_{f_0})) \tag{6d}$$

$$\text{and } g_0(j_0) = \left\{ \begin{array}{ll} @scope_{\{\spadesuit\}} : & j \\ @prototype_{\{\spadesuit\}} : & null \\ @this_{\{\spadesuit\}} : & \#global \\ "_k"_{\{\spadesuit\}} : & \ell_f \end{array} \right\} \tag{6e}$$

By Definition 21, (6d) and (6e),

$$(h, \ell) \preceq_\beta (g_0, j_0) \tag{6f}$$

By inductive hypothesis, Lemma 10, (6f) and (6b),

$$(\spadesuit, g_0, j_0, \mathcal{C}'\langle e \rangle(\ell_{f_0})) \to^* (\spadesuit, g_1, j_1, \ell_{f_0}(true)) \tag{6g}$$
$$(h_0, \ell_0) \preceq_{\beta'} (g_0, j_0) \tag{6h}$$
$$true \sim_{\beta'} true \tag{6i}$$

By semantics rules:

$$(\spadesuit, g_1, j_1, \ell_{f_0}(true)) \xrightarrow{\text{Fun-Call}}{}^{*} (\spadesuit, g_2, j_2, \mathsf{if}\ (_v)\ \mathcal{C}'\langle s_0\rangle(_k)\ \mathsf{else}\ \mathcal{C}'\langle s_1\rangle(_k)) \tag{6j}$$

$$\to^{*} (\spadesuit, g_2, j_2, \mathcal{C}'\langle s_0\rangle(_k)) \tag{6k}$$

$$\to^{*} (\spadesuit, g_2, j_2, \mathcal{C}'\langle s_0\rangle(\ell_f)) \tag{6l}$$

$$\text{and } g_2(j_2) = \left\{ \begin{array}{ll} @scope_{\{\spadesuit\}}: & j_0 \\ @prototype_{\{\spadesuit\}}: & null \\ @this_{\{\spadesuit\}}: & \#global \\ "_v"_{\{\spadesuit\}}: & true \end{array} \right\} \tag{6m}$$

By Definition 21, (6j) and (6m),

$$(h_0, \ell_0) \preceq_\beta (g_2, j_2) \tag{6n}$$

By inductive hypothesis, (6n) and (6c),

$$(\spadesuit, g_2, j_2, \mathcal{C}'\langle s_0\rangle(\ell_f)) \to^{*} (\spadesuit, g_3, j_3, \ell_f(v'))$$
$$(h_1, \ell_1) \preceq_{\beta''} (g_3, j_3)$$
$$v \sim_{\beta''} v'$$

2. The condition expression e evaluates to *false*. The proof of this sub-case is similar to the proof of previous sub-case. We omit the detail.

We omit the other case where the evaluation of the conditional involves a return statement.
Case $s = \mathsf{while}\ (e)\ s$: By the definition of the CPS compiler,

$$\mathcal{C}'\langle s\rangle = \mathsf{function}(_k)\{\mathsf{var}_c;\ _c = f_0;\ _c(undefined)\}$$
$$f_0 = \mathsf{function}(_v)\{\mathcal{C}'\langle e\rangle(f_1)\}$$
$$f_1 = \mathsf{function}(_b)\{\mathsf{if}\ (_b)\ \mathcal{C}'\langle s\rangle(_c)\ \mathsf{else}\ _k(_b)\}$$

By semantics rules, if the evaluation of the while loop terminates,

$$\begin{aligned} (\spadesuit, h, \ell, \mathsf{while}\ (e)\ s) &\to^{*} (\spadesuit, h_0, \ell_0, s; \mathsf{while}\ (e)\ s) \\ &\to^{*} (\spadesuit, h'_0, \ell'_0, \mathsf{while}\ (e)\ s) \\ &\to^{*} (\spadesuit, h_1, \ell_1, s; \mathsf{while}\ (e)\ s) \\ &\to^{*} (\spadesuit, h'_1, \ell'_1, \mathsf{while}\ (e)\ s) \\ &\ \ \vdots \\ &\to^{*} (\spadesuit, h'_{n-1}, \ell'_{n-1}, \mathsf{while}\ (e)\ s) \\ &\to^{*} (\spadesuit, h_n, \ell_n, false) \end{aligned}$$

On the CPS-transformation side, by semantics rules

$$(\spadesuit, g, j, \mathcal{C}'\langle s\rangle(\ell_f)) \xrightarrow{\text{Fun, Fun-Call}}{}^{*} (\spadesuit, g_0, j_0, \mathsf{var}_c;\ _c = f_0;\ _c(undefined)) \tag{7a}$$

$$\xrightarrow{\text{Var, Assign, GetV, Fun-Call}}{}^{*} (\spadesuit, g_1, j_1, \mathcal{C}'\langle e\rangle(l_{f_1})) \tag{7b}$$

$$\text{and } g_0(j_0) = \left\{ \begin{array}{ll} @scope_{\{\spadesuit\}}: & j \\ @prototype_{\{\spadesuit\}}: & null \\ @this_{\{\spadesuit\}}: & \#global \\ "_k"_{\{\spadesuit\}}: & \ell_f \\ "_c"_{\{\spadesuit\}}: & \ell_c \end{array} \right\} \tag{7c}$$

$$\text{and } g_1(j_1) = \left\{ \begin{array}{ll} @scope_{\{\spadesuit\}}: & j_0 \\ @prototype_{\{\spadesuit\}}: & null \\ @this_{\{\spadesuit\}}: & \#global \\ "_v"_{\{\spadesuit\}}: & undefined \end{array} \right\} \tag{7d}$$

$$\text{and } g_1(\ell_c).@body = \mathcal{C}'\langle e\rangle(f_1) \tag{7e}$$

By Definition 21, (7a) to (7d),

$$(h,\ell) \preceq_{\beta} (g_1, j_1) \tag{7f}$$

Let us prove by induction on the length of the execution of the body of while-loop:

1. **Case** n=0 : The loop body is not executed. We have,

$$(\spadesuit, h, \ell, e) \to^* (\spadesuit, h_0, \ell_0, \mathit{false}) \tag{8a}$$

By inductive hypothesis, Lemma 10, (8a) and (7f),

$$(\spadesuit, g_1, j_1, \mathcal{C}'\langle e\rangle(\ell_{f_1})) \to^* (\spadesuit, g_2, j_2, \ell_{f_1}(false)) \tag{8b}$$

$$(h_0, \ell_0) \preceq_{\beta'} (g_2, j_2) \tag{8c}$$

$$\mathit{false} \sim_{\beta'} \mathit{false} \tag{8d}$$

By semantics rules and (7d),

$$(\spadesuit, g_2, j_2, \ell_{f_1}(false)) \to^* (\spadesuit, g_3, j_3, \mathsf{if}\ (_b)\ \mathcal{C}'\langle s\rangle(_c)\ \mathsf{else}\ _k(_b)) \tag{8e}$$

$$\to^* (\spadesuit, g_3, j_3, _k(_b)) \tag{8f}$$

$$\to^* (\spadesuit, g_3, j_3, \ell_f(false)) \tag{8g}$$

$$\text{and } g_3(j_3) = \left\{ \begin{array}{ll} @\mathit{scope}_{\{\spadesuit\}} : & j_1 \\ @\mathit{prototype}_{\{\spadesuit\}} : & \mathit{null} \\ @\mathit{this}_{\{\spadesuit\}} : & \#\mathit{global} \\ "_b"_{\{\spadesuit\}} : & \mathit{false} \end{array} \right\} \tag{8h}$$

(8i)

Furthermore, by (8e) and (8h) we have

$$(h_n, \ell_n) \preceq_{\beta'} (g_3, j_3) \tag{8j}$$

2. **Case** $n > 0$:

$$(\spadesuit, h, \ell, e) \to^* (\spadesuit, h_0, \ell_0, \mathit{true}) \tag{9a}$$

$$(\spadesuit, h, \ell, s) \to^* (\spadesuit, h_0', \ell_0', v) \tag{9b}$$

By inductive hypothesis, Lemma 10, (9a) and (7f),

$$(\spadesuit, g_1, j_1, \mathcal{C}'\langle e\rangle(\ell_{f_1})) \to^* (\spadesuit, g_2, j_2, \ell_{f_1}(\mathit{true})) \tag{9c}$$

$$(h_0, \ell_0) \preceq_{\beta'} (g_2, j_2) \tag{9d}$$

$$\mathit{true} \sim_{\beta'} \mathit{true} \tag{9e}$$

By semantics rules and (7d),

$$(\spadesuit, g_2, j_2, \ell_{f_1}(\mathit{true})) \to^* (\spadesuit, g_3, j_3, \mathsf{if}\ (_b)\ \mathcal{C}'\langle s\rangle(_c)\ \mathsf{else}\ _k(_b)) \tag{9f}$$

$$\to^* (\spadesuit, g_3, j_3, \mathcal{C}'\langle s\rangle(\ell_c)) \tag{9g}$$

$$\text{and } g_3(j_3) = \left\{ \begin{array}{ll} @\mathit{scope}_{\{\spadesuit\}} : & j_1 \\ @\mathit{prototype}_{\{\spadesuit\}} : & \mathit{null} \\ @\mathit{this}_{\{\spadesuit\}} : & \#\mathit{global} \\ "_b"_{\{\spadesuit\}} : & \mathit{false} \end{array} \right\} \tag{9h}$$

(9i)

Furthermore, by (9f) and (9h) we have

$$(h_0, \ell_0) \preceq_{\beta'} (g_3, j_3) \tag{9j}$$

By inductive hypothesis, (9b) and (9j),

$$(\spadesuit, g_3, j_3, \mathcal{C}'\langle s\rangle(\ell_c)) \rightarrow^* (\spadesuit, g_4, j_4, \ell_c(v')) \tag{9k}$$

$$(h'_0, \ell'_0) \preceq_{\beta''} (g_4, j_4) \tag{9l}$$

$$v \sim_{\beta''} v' \tag{9m}$$

By semantics rule, we have

$$(\spadesuit, g_4, j_4, \ell_c(v')) \rightarrow^* (\spadesuit, g_5, j_5, \mathcal{C}'\langle e\rangle(f_1))$$

$$\text{and } g_5(j_5) = \left\{ \begin{array}{ll} @scope_{\{\spadesuit\}}: & j_0 \\ @prototype_{\{\spadesuit\}}: & null \\ @this_{\{\spadesuit\}}: & \#global \\ "_v"_{\{\spadesuit\}}: & v' \end{array} \right\}$$

$$(h'_0, \ell'_0) \preceq_{\beta''} (g_5, j_5)$$

Now the configuration $(\spadesuit, h'_0, \ell'_0, \mathsf{while}\ (e)\ s)$ has $n-1$ execution of the while-loop body. By inductive hypothesis,

$$(\spadesuit, g_5, j_5, \mathcal{C}'\langle e\rangle(f_1)) \rightarrow^* (\spadesuit, g_m, j_m, \ell_f(\mathit{false}))$$

$$(h_n, \ell_n) \preceq_{\beta'''} (g_m, j_m)$$

We omit the other case where the evaluation of the while-loop involves a return statement.

Case $s = \mathsf{return}\ e$: By the definition of the CPS compiler,

$$\mathcal{C}'\langle s\rangle = \mathsf{function}(_k)\{\mathcal{C}'\langle e\rangle(_fun_cont)\}$$

By semantics rules,

$$(\spadesuit, h, \ell, e) \rightarrow^* (\spadesuit, h', \ell', v) \tag{10a}$$

$$(\spadesuit, h, \ell, \mathsf{return}\ e) \rightarrow^* (\spadesuit, h', \ell', \mathsf{return}\ v) \tag{10b}$$

$$\tag{10c}$$

By assumption,

$$\text{Lookup}(g, j, _fun_cont) = \ell_c \tag{10d}$$

On the CPS-transformation side, by semantics rules

$$(\spadesuit, g, j, \mathcal{C}'\langle s\rangle(\ell_f)) \xrightarrow{\text{FUN, FUN-CALL, FUN}}{}^{*} (\spadesuit, g_0, j_0, \mathcal{C}'\langle e\rangle(\ell_c)) \tag{10e}$$

$$\text{and } g_0(j_0) = \left\{ \begin{array}{ll} @scope_{\{\spadesuit\}}: & j \\ @prototype_{\{\spadesuit\}}: & null \\ @this_{\{\spadesuit\}}: & \#global \\ "_k"_{\{\spadesuit\}}: & \ell_f \end{array} \right\} \tag{10f}$$

By Definition 21, (10e) and (10f),

$$(h, \ell) \preceq_{\beta} (g_0, j_0) \tag{10g}$$

By inductive hypothesis, Lemma 10, (10g) and (10a),

$$(\spadesuit, g_0, j_0, \mathcal{C}'\langle e\rangle(\ell_x)) \rightarrow^* (\spadesuit, g_1, j_1, \ell_c(v')) \tag{10h}$$

$$(h', \ell') \preceq_{\beta'} (g_1, j_1) \tag{10i}$$

$$v \sim_{\beta'} v' \tag{10j}$$

from which we can conclude this case. □

The following lemma shows that, under certain conditions, for any expression, the intermediate CPS-transformation behaves the same as the original expression.

Lemma 10 (CPS Transformation Lemma - Expressions)**.** *Let h and g be two decorated heaps, ℓ and j be two heap locations such that $(h, \ell) \preceq_\beta (g, j)$ for some β. If $(\spadesuit, h, \ell, e) \xrightarrow{\varepsilon}^* (\spadesuit, h', \ell', v)$, then the following conditions hold:*

1. $(\spadesuit, g, j, \mathcal{C}'\langle e\rangle(\ell_f)) \xrightarrow{\varepsilon}^* (\spadesuit, g', j', \ell_f(v'))$
2. *there exists β' such that $(h', \ell') \preceq_{\beta'} (g', j')$ and $v \sim_{\beta'} v'$.*

Proof. The proof is by induction on the definition of the CPS transformer $\mathcal{C}'\langle e\rangle$. For all cases, we have some common hypothesis:

$$(h, \ell) \preceq_\beta (g, j) \tag{11a}$$

By Definition 21, (11a) implies

$$(h\lfloor_\spadesuit, \ell) \sim_\beta (g\lfloor_\spadesuit, j) \tag{11b}$$

$$h\lfloor_{\mathrm{V}} = g\lfloor_{\mathrm{V}} \text{ are benign gadget's heaps.} \tag{11c}$$

$$h\lfloor_{\mathrm{W}} = g\lfloor_{\mathrm{W}} = h_{in} \tag{11d}$$

Case $e = \mathsf{this}$ **:** By definition of the CPS compiler, we have

$$\mathcal{C}'\langle\mathsf{this}\rangle = \mathsf{function}(_k)\{_k(_this)\}$$

By the semantics rules, we have

$$\mathrm{Lookup}(h, \ell, @this) = \ell_t \tag{12a}$$

$$(\spadesuit, h, \ell, \mathsf{this}) \xrightarrow{\text{THIS}} (\spadesuit, h, \ell, \ell_t)$$

By the semantics rules,

$$\begin{aligned}(\spadesuit, g, j, (\mathcal{C}'\langle\mathsf{this}\rangle)(\ell_f)) &\xrightarrow{\text{FUN}} (\spadesuit, g_0, j, \ell_{cps}(\ell_f)) \\ &\xrightarrow{\text{FUN-CALL}} (\spadesuit, g_1, j_0, _k(_this)) \\ &\xrightarrow{\text{GETV}} (\spadesuit, g_1, j_0, \ell_f(_this)) \\ &\xrightarrow{\text{GETV}} (\spadesuit, g_1, j_0, \ell_f(\ell'_t))\end{aligned}$$

By semantics rules FUN and FUN-CALL, we have

$$g_1 = g \cup \left\{\begin{array}{ll} \ell_{cps\{\spadesuit\}} \mapsto & \left\{\begin{array}{ll} \text{"}prototype\text{"}: & \ell_{p1} \\ @prototype: & \#funprot \\ @call: & true \\ @fscope: & j \\ @body: & \mathsf{function}(_k)\{_k(_this)\} \end{array}\right\}, \\ \ell_{p1\{\spadesuit\}} \mapsto & \{@prototype: \#objprot\}, \\ j_{0\{\spadesuit\}} \mapsto & \left\{\begin{array}{ll} @scope: & j \\ @prototype: & null \\ @this: & \#global \\ \text{"}_k\text{"}: & \ell_f \end{array}\right\} \end{array}\right\} \tag{12b}$$

(12c)

By Definition 12, Definition 21, (11b) and (12b),

$$(h\lfloor_\spadesuit, \ell) \sim_\beta (g_2\lfloor_\spadesuit, j_0) \tag{12d}$$

$$(h, \ell) \preceq_{\beta(} g_2, j_0)$$

By semantics rule GetV, (12a), (12d)

$$\text{Lookup}(g_2, j_0, \text{``_}this\text{''}) = \ell'_t \tag{12e}$$
$$\ell_t \sim_\beta \ell'_t \tag{12f}$$

from which we can conclude this case.

Case $e = x$: By definition of the CPS compiler, we have

$$\mathcal{C}'\langle x \rangle = \ \mathsf{function}(_k)\{_k(x)\}$$

By the semantics rules, we have

$$\text{Scope}(h, \ell, x) = l_1 \tag{13a}$$
$$h(\ell_1).\text{``}x\text{''}_{\{\square\}} = v$$
$$(\spadesuit, h, \ell, x) \xrightarrow{\text{GetV}} (\spadesuit, h, \ell, v) \tag{13b}$$

By the semantics rules,

$$\begin{aligned}(\spadesuit, g, j, (\mathcal{C}'\langle x \rangle)(\ell_f)) &\xrightarrow{\text{Fun}} (\spadesuit, g_0, j, \ell_{cps}(\ell_f))\\ &\xrightarrow{\text{Fun-Call}} (\spadesuit, g_1, j_0, _k(x))\\ &\xrightarrow{\text{GetV}} (\spadesuit, g_2, j_0, \ell_f(x))\\ &\xrightarrow{\text{GetV}} (\spadesuit, g_2, j_0, \ell_f(v'))\end{aligned}$$

By similar reasoning from (12b),

$$(h\lfloor_\spadesuit, \ell) \sim_\beta (g_2\lfloor_\spadesuit, j_0) \tag{13c}$$
$$(h, \ell) \preceq_{\beta(} g_2, j_0) \tag{13d}$$

By semantics rule GetV,

$$\text{Scope}(g_2, j_0, \text{``}x\text{''}) = j_1 \tag{13e}$$
$$g_2(j_1).\text{``}x\text{''}_{\{\square\}} = v'$$

By Definition 12, Lemma 6, (13a), (13e), (13c) and (13d), we have the following two cases to discuss:

1. $\ell_1 = j_1 = \#global$:
   (a) "x" is a W or V-colored property, then $v = v'$;
   (b) "x" is a $\spadesuit$-colored property, then we have $v \sim_\beta v'$;
2. $\ell_1 = j_1 \neq \#global$, we have $v \sim_\beta v'$.

therefore in both case we have

$$v \sim_\beta v'$$

from which we can conclude this case.

Case $e = pv$: Similar to the previous case.

Case $e = \{m_0 : e_0, m_1 : e_1\}$:

We prove the case for an object literal that contains two properties. Other cases can be easily extended. By definition of the CPS compiler, we have

$$\begin{aligned}\mathcal{C}'\langle\{m_0 : e_0, m_1 : e_1\}\rangle &= \ \mathsf{function}(_k)\{(\mathcal{C}'\langle e_0\rangle)(f_0)\}\\ f_0 &= \ \mathsf{function}(_x_0)\{(\mathcal{C}'\langle e_1\rangle)(f_1)\}\\ f_1 &= \ \mathsf{function}(_x_1)\{_k(\{m_0 : _x_0 \ op \ m_1 : _x_1\})\}\end{aligned}$$

By the semantics rules, we have

$$(\spadesuit, h, \ell, e_0) \to^* (\spadesuit, h_0, \ell, v_0) \tag{14a}$$

$$(\spadesuit, h_0, \ell, e_1) \to^* (\spadesuit, h_1, \ell, v_1) \tag{14b}$$

$$(\spadesuit, h, \ell, \{m_0 : e_0, m_1 : e_1\}) \xrightarrow{\text{OBJ-LITERAL}} (\spadesuit, h_2, \ell, \ell_0) \tag{14c}$$

By the rule OBJ-LITERAL, and (14c), we have

$$h_2 = h_1 \cup \{l_o \mapsto o\} \tag{14d}$$

By the semantics rules, we also have

$$\begin{aligned}
(\spadesuit, g, j, (\mathcal{C}'\langle e\rangle)(f)) &\xrightarrow{\text{FUN}} (\spadesuit, g_0, j, \ell_{cps}(\ell_f)) \\
&\xrightarrow{\text{FUN-CALL}} (\spadesuit, g_1, j_0, (\mathcal{C}'\langle e_0\rangle)(f_0)) \\
&\xrightarrow{\text{FUN}} (\spadesuit, g_2, j_0, (\mathcal{C}'\langle e_0\rangle)(\ell_{f_0})) \\
&\xrightarrow{\text{BY (14G),FUN-CALL}}{}^{*} (\spadesuit, g_3, j_1, (\mathcal{C}'\langle e_1\rangle)(f_1)) \\
&\xrightarrow{\text{FUN}} (\spadesuit, g_4, j_1, (\mathcal{C}'\langle e_1\rangle)(\ell_{f_1})) \\
&\xrightarrow{\text{BY (14L),FUN-CALL}}{}^{*} (\spadesuit, g_5, j_2, _k(\{m_0 : _x_0, m_1 : _x_1\})) \\
&\xrightarrow{\text{FUN-CALL}} (\spadesuit, g_6, j_2, \ell_f(\ell'_o))
\end{aligned}$$

By similar reasoning from (12b), Definition 12, 21, and (11a),

$$(h\lfloor_{\spadesuit}, \ell) \sim_{\beta} (g_2\lfloor_{\spadesuit}, j_0) \tag{14e}$$

$$(h, \ell) \preceq_{\beta(} g_2, j_0) \tag{14f}$$

By inductive hypothesis, (14a) and(14f),

$$(\spadesuit, g_2, j_0, (\mathcal{C}'\langle e_0\rangle)(\ell_{f_0})) \to^* (\spadesuit, g_3, j'_1, \ell_{f_0}(v'_0)) \tag{14g}$$

$$v_0 \sim_{\beta'} v'_0 \tag{14h}$$

$$(h_0, \ell) \preceq_{\beta'} (g_3, j'_1) \tag{14i}$$

By semantics rules, similar reasoning from (12b), Definition 12 and (14i), we have

$$g_4(j_1) = \left\{ \begin{array}{ll} @\mathit{scope}: & j_0 \\ @\mathit{prototype}: & \mathit{null} \\ @\mathit{this}: & \#\mathit{global} \\ ``_x_0" : & v'_0 \end{array} \right\} \tag{14j}$$

$$(h, \ell) \preceq_{\beta(} g_4, j_1) \tag{14k}$$

By inductive hypothesis and (14b), we have

$$(\spadesuit, g_4, j_1, (\mathcal{C}'\langle e_1\rangle)(\ell_{f_1})) \to^* (\spadesuit, g_5, j'_2, \ell_{f_1}(v'_1)) \tag{14l}$$

$$v_1 \sim_{\beta''} v'_1 \tag{14m}$$

$$(h_1, \ell) \preceq_{\beta''} (g_5, j'_2) \tag{14n}$$

By similar reasoning from (12b), Definition 12 and (14l), we have

$$g_5(j_2) = \left\{ \begin{array}{ll} @\mathit{scope}: & j_1 \\ @\mathit{prototype}: & \mathit{null} \\ @\mathit{this}: & \#\mathit{global} \\ ``_x_1" : & v'_1 \end{array} \right\} \tag{14o}$$

By semantics rules, we have

$$g_6 = g_5 \cup \{l'_{o\{\spadesuit\}} \mapsto o'\} \tag{14p}$$

$$o = \left\{ \begin{array}{ll} @prototype_{\{\spadesuit\}}: & \#objprot \\ m_{1\{\spadesuit\}}: & v_0 \\ m_{2\{\spadesuit\}}: & v_1 \end{array} \right\} \tag{14q}$$

$$o' = \left\{ \begin{array}{ll} @prototype_{\{\spadesuit\}}: & \#objprot \\ m_{1\{\spadesuit\}}: & v'_0 \\ m_{2\{\spadesuit\}}: & v'_1 \end{array} \right\} \tag{14r}$$

$$\tag{14s}$$

By definition 17,

$$o \sim_{\beta''} o' \tag{14t}$$

Let $\beta''' = \beta'' \cup \{l_o \mapsto l'_o\}$, by (14o), Definition 15 and Definition 21 we have

$$l_o \sim_{\beta'''} l'_o \tag{14u}$$

$$(h_2, \ell) \preceq_{\beta'''} (g_6, j_2) \tag{14v}$$

Case $e = e_0[e_1]$: By definition of the CPS compiler, we have

$$\begin{aligned} \mathcal{C}'\langle e_0[e_1]\rangle =\ & \mathsf{function}(_k)\{(\mathcal{C}'\langle e_0\rangle)(f_0)\} \\ f_0 =\ & \mathsf{function}(_x_0)\{(\mathcal{C}'\langle e_1\rangle)(f_1)\} \\ f_1 =\ & \mathsf{function}(_x_1)\{_k(_x_0[_x_1])\} \end{aligned}$$

By the semantics rules, we have

$$(\spadesuit, h, \ell, e_0[e_1]) \to^* (\spadesuit, h_0, \ell, v_0[e_1]) \tag{15a}$$

$$\to^* (\spadesuit, h_1, \ell, v_0[v_1]) \tag{15b}$$

$$\xrightarrow{\text{GetV}} (\spadesuit, h_1, \ell, v) \tag{15c}$$

By semantics rules and similar reasoning from (14l),

$$(g, j, \mathcal{C}'\langle e\rangle(\ell_f)) \to^* (g_0, j_0, l_f(v'_1[v'_2])) \tag{15d}$$

$$(g_0, j_0, l_f(v')) \tag{15e}$$

$$\tag{15f}$$

such that the following conditions holds,

$$(h_1, \ell) \preceq_{\beta'} (g_0, j_0) \tag{15g}$$

$$v_1 \sim_{\beta'} v'_1 \tag{15h}$$

$$v_2 \sim_{\beta'} v'_2 \tag{15i}$$

Let us analyze by the case of v_1 and v'_1, since $v_1 \sim_{\beta'} v'_1$, we have the following cases:

1. $v_1 \notin \mathsf{dom}(\beta')$ and $v_1 = v'_1$, thus $h_1(v_1)$ is a W or V-colored object. By (15g) we have $h_1(v_1) = g_0(v_1)$, therefore we have $v = v'$;
2. $v_1 = v'_1 = \#global$,
   (a) If v_2 is a $\spadesuit$-colored property, by Definition 15 we have $v \sim_{\beta'} v'$;
   (b) If v_2 is a W or V-colored property, by similar reasoning we have $v = v'$.
3. $\beta'(v_1) = v'_1$, then by Definition 15 we have $v \sim_{\beta'} v'$.

Therefore in all sub-cases we have

$$v \sim_{\beta'} v' \tag{15j}$$

The rest reasoning follows the same strategy of the previous case.

Case $e = e_0(e_1)$: By definition of the CPS compiler, we have

$$\begin{aligned}
\mathcal{C}'\langle e_0(e_1)\rangle = \;& \mathsf{function}(_k)\{(\mathcal{C}'\langle e_0\rangle)(f_0)\} \\
f_0 = \;& \mathsf{function}(_x_0)\{(\mathcal{C}'\langle e_1\rangle)(f_1)\} \\
f_1 = \;& \mathsf{function}(_x_1)\{\mathsf{if}\ (_x_0._is_ohandle)\ _k(_x_0(_x_1))\ \mathsf{else}\ _x_0(_k, _x_1)\}
\end{aligned}$$

By the semantics rules, we have

$$\begin{aligned}
(\spadesuit, h, \ell, e_0(e_1)) &\to^* (\spadesuit, h_0, \ell, v_0(e_1)) && (16a)\\
&\to^* (\spadesuit, h_1, \ell, v_0(v_1)) && (16b)\\
&\to^* (\spadesuit, h_2, \ell, v) && (16c)
\end{aligned}$$

By semantics rules and similar reasoning from (14l),

$$\begin{aligned}
&(\spadesuit, g, j, (\mathcal{C}'\langle e\rangle)(\ell_f)) \xrightarrow{\text{FUN}} (\spadesuit, g_0, j_0, \mathsf{if}\ (v_0'._is_ohandle)\ _k(_x_0(_x_1))\ \mathsf{else}\ _x_0(_k, _x_1)) && (16d)\\
&(h_1, \ell) \preceq_{\beta'} (g_0, j_0) && (16e)\\
&v_0 \sim_{\beta'} v_0' && (16f)\\
&v_1 \sim_{\beta'} v_1' && (16g)
\end{aligned}$$

Let us discuss by two different sub-cases:

1. $g_0(v_0).``_is_ohandle" = true$: then v_0 is a V-colored function object. Therefore by Assumption 3

$$\begin{aligned}
v_0 &= v_0' \\
h_1(v_1) &= g_0(v_1') = o \text{ is a simple object} \\
h_1\lfloor_{\mathrm{V}} &= g_0\lfloor_{\mathrm{V}} \text{ is a benign gadget's heap}
\end{aligned}$$

   then by Assumption 9,

$$\begin{aligned}
(\spadesuit, g_0, j_0, \mathsf{if}\ (v_0'._is_ohandle)\ _k(_x_0(_x_1))\ \mathsf{else}\ _x_0(_k, _x_1)) &\to^* (\spadesuit, g_0, j_0, \ell_f(v_0'(v_1'))) \\
&\to^* (\spadesuit, g_1, j_0, \ell_f(v'))
\end{aligned}$$

   such that

$$\begin{aligned}
h_2\lfloor_{\mathrm{V}} &= g_1\lfloor_{\mathrm{V}} \text{ is a benign gadget's heap} \\
(h_2, \ell) &\preceq_{\beta'} (g_1, j_0) \\
v &= v'
\end{aligned}$$

2. $g_0(v_0).``_is_ohandle" = true$: then v_0 is a $\spadesuit$-colored function object. We have

$$\begin{aligned}
(\spadesuit, h_1, \ell, v_0(v_1)) &\to^* (\spadesuit, h_1', \ell_1, \text{@FunExe}(\ell, s)) && (16h)\\
&\to^* (\spadesuit, h_2', \ell_1, \text{@FunExe}(\ell, v)) && (16i)\\
&\to^* (\spadesuit, h_2, \ell, v) && (16j)
\end{aligned}$$

$$h_1(v_0) = \left\{ \begin{array}{ll} \text{"}prototype\text{"}: & \ell_{p_0} \\ @prototype: & \#funprot \\ @call: & true \\ @fscope: & \ell_0 \\ @body: & \mathsf{function}(x)\{s'\} \end{array} \right\} \tag{16k}$$

$$\begin{aligned}
h_1(\ell_1).@scope &= \ell_0 && (16l)\\
h_1(\ell_1).``x" &= v_1 && (16m)
\end{aligned}$$

By Definition 15 and (16e), we have

$$g_0(v_1') = \left\{ \begin{array}{ll} \text{"}prototype\text{"}: & j_{p_0'} \\ @prototype: & \#funprot \\ @call: & true \\ @fscope: & j_1 \\ @body: & \mathsf{function}(x)\{s''\} \end{array} \right\} \tag{16n}$$

$$\begin{array}{rcl} s'' & = & \mathsf{var}\ _this; \\ & & _this = \mathsf{this}; \\ & & (\mathcal{C}'\langle s' \rangle)(_fun_cont) \end{array} \tag{16o}$$

By semantics rules,

$$(\spadesuit, g_0, j_0, v_1'(l_f, v_2')) \xrightarrow{\text{FUN-CALL},\ldots}{}^{*} (\spadesuit, g_1, j_2, (\mathcal{C}'\langle s' \rangle)(l_f)) \tag{16p}$$

Let $\beta' = \beta \cup \{\ell_1 \mapsto j_2\}$, by Definition 15,

$$(h_1', \ell_1) \preceq_{\beta'} (g_1, j_2) \tag{16q}$$

$$g_1(j_2).\text{"}_fun_cont\text{"} = \ell_f \tag{16r}$$

By Lemma 9,

$$\begin{array}{c} (\spadesuit, g_1, j_2, (\mathcal{C}'\langle s' \rangle)(l_f)) \rightarrow^{*} (\spadesuit, g_2, j_3, l_f(v')) \\ (h_2', \ell_1) \preceq_{\beta''} (g_2, j_3) \\ v \sim_{\beta''} v' \end{array}$$

Case $e = e_0[e_1](e_2)$: Similar to previous case.
Case $e = \mathsf{function}(x_0)\{s_0\}$:
By definition of the CPS compiler, we have

$$\mathcal{C}'\langle \mathsf{function}(x_0)\{s_0\} \rangle = \mathsf{function}(_k)\{_k(\mathsf{function}(_fun_cont, x_0)\{s'\})\}$$

$$\begin{array}{rcl} s' & = & \mathsf{var}\ _this; \\ & & _this = \mathsf{this}; \\ & & (\mathcal{C}'\langle s_0 \rangle)(_fun_cont) \end{array}$$

We assume that e is a simple function, which has the following properties:

s_0 does not contain variable declaration.

By the semantics rules, we have

$$(\spadesuit, h, \ell, \mathsf{function}(x_0)\{s_0\}) \xrightarrow{\text{FUN}} (\spadesuit, h_0, \ell, l_{f_0}) \tag{17a}$$

where

$$h_0 = h \cup \left\{ \begin{array}{ll} \ell_{f_0\{\spadesuit\}} \mapsto & \left\{ \begin{array}{ll} \text{"}prototype\text{"}: & \ell_p \\ @prototype: & \#funprot \\ @call: & true \\ @fscope: & \ell \\ @body: & \mathsf{function}(x_0)\{s_0\} \end{array} \right\}, \\ \ell_{p\{\spadesuit\}} \mapsto & \{@prototype: \#objprot\} \end{array} \right\} \tag{17b}$$

By the semantics rules, we also have

$$(\spadesuit, g, j, (\mathcal{C}'\langle e \rangle)(\ell_f)) \xrightarrow{\text{FUN}} (\spadesuit, g_0, j, l_{cps}(l_f)) \tag{17c}$$

$$\xrightarrow{\text{FUN-CALL}} (\spadesuit, g_1, j_0, _k(\mathsf{function}(_fun_cont, x_0)\{s'\})) \tag{17d}$$

$$\xrightarrow{\text{GETV}} (\spadesuit, g_1, j_0, l_f(\mathsf{function}(_fun_cont, x_0)\{s'\})) \tag{17e}$$

$$\xrightarrow{\text{GETV}} (\spadesuit, g_2, j_0, l_f(l_{f_0}')) \tag{17f}$$

By similar reasoning from (12d), we have

$$(h\lfloor_\spadesuit, l) \sim_\beta (g_1\lfloor_\spadesuit, j_0) \tag{17g}$$
$$(h, \ell) \preceq_{\beta(} g_1, j_0)$$

and we also have

$$g_2 = g_1 \cup \left\{ \begin{array}{ll} \ell'_{f_0\{\spadesuit\}} \mapsto & \left\{ \begin{array}{ll} \text{"}prototype\text{"}: & \ell'_p \\ @prototype: & \#funprot \\ @call: & true \\ @fscope: & j_0 \\ @body: & \mathsf{function}(_fun_cont, x_0)\{s'\} \end{array} \right\}, \\ \ell'_{p\{\spadesuit\}} \mapsto & \{@prototype: \#objprot\} \end{array} \right\} \tag{17h}$$

Let $\beta' = \beta \cup \{\ell_{f_0} \mapsto \ell'_{f_0}, \ell_p \mapsto \ell'_p\}$, by Definition 12, we have.

$$(h_0\lfloor_\spadesuit, \ell) \sim_{\beta'} (g_2\lfloor_\spadesuit, j_0) \tag{17i}$$
$$(h_0, \ell) \preceq_{\beta(} g_2, j_0)$$
$$\ell_{f_0} \sim_{\beta'} \ell'_{f_0}$$

Case $e = e_0 \; op \; e_1$: This case can be reasoned by similar strategy from the case of object literal.
Case $e = x = e_0$: By definition of the CPS compiler, we have

$$\mathcal{C}'\langle x = e_0\rangle = \mathsf{function}(_k)\{(\mathcal{C}'\langle e_0\rangle)(f_0)\} \tag{18a}$$
$$f_0 = \mathsf{function}(_x_0)\{k(x = _x_0)\} \tag{18b}$$

By the semantics rules, we have

$$(\spadesuit, h, l, x = e) \rightarrow^* (\spadesuit, h_0, l, x = v_0) \tag{18c}$$
$$\tag{18d}$$

By semantics rules, inductive hypothesis and similar reasoning from (14l),

$$(\spadesuit, g, j, (\mathcal{C}'\langle e\rangle)(f)) \rightarrow^* (\spadesuit, g_0, j_0, l_f(x \;=\; v'_0)) \tag{18e}$$
$$\rightarrow^* (\spadesuit, g_1, j_0, l_f(v'_0)) \tag{18f}$$

such that the following holds.

$$(h_0, \ell) \preceq_{\beta'} (g_0, j_0) \tag{18g}$$
$$v_0 \sim_{\beta'} v'_0 \tag{18h}$$
$$\tag{18i}$$

By the semantics we have

$$\text{Scope}(h_0, \ell, x) = \ell_0 \tag{18j}$$
$$\text{Scope}(g_0, j_0, x) = j_2 \tag{18k}$$

By Definition 12, Lemma 6, we have the following two cases to discuss:

1. $\ell_0 = j_2 = \#global$: it must be the case that "x" is a $\spadesuit$-colored property of the global object, otherwise the simple-integrator assumption us violated.
2. $\beta(\ell_0) = j_2$, then $h(\ell_0)$ and $g_0(j_2)$ are both $\spadesuit$-colored.

therefore, for both case, by Definition 15, Definition 12 and 18h, we have

$$(h_1\lfloor_\spadesuit, \ell) \sim_{\beta'} (g_1\lfloor_\spadesuit, j_0) \tag{18l}$$
$$(h_1, \ell) \preceq_{\beta'} (g_1, j_0) \tag{18m}$$

Case $e = e_0[e_1] = e_2$: By similar reasoning of the previous case.
Case $e = \mathsf{typeof}\; e_0$: This case can be reasoned by similar strategy from the case of object literal. □

Remark 11. If there is a frame, we can always separate the heap into two part where $h = h_i \oplus h_f$, and h_i and h_f does not contain any locations to each other.

In the following lemma we show that the intermediate compilation behave indistinguishable with the Mashic compilation. The proof proceeds by a mutual structural induction on the definition of statements and expressions.

Lemma 12 (Intermediate to Mashic Compilation)**.** *Let P_i be a simple integrator and P_g be a benign gadget such that P_i does not conflict with P_g and $\mathcal{V} = \mathsf{var}(P_g)$. If $\langle \spadesuit, \varepsilon, null, \tilde{M}_i(P_i, P_g), Q_{init}\rangle_I \to^* \langle \square, h_i, \ell_i, \varepsilon, Q_{init}\rangle_x$, then $\langle \spadesuit, \varepsilon, null, \tilde{M}_c(P_i, P_g, \mathcal{V}), Q_{init}\rangle_I \to^* \langle \square, h_c, \ell_c, \varepsilon, Q_c\rangle_x$ where Q_c has no message waiting, and there exist a mapping $f : \mathcal{N} \mapsto \mathcal{L}$ such that $h_c \stackrel{f}{=} h_i$.*

Sketch of Proof. Without of loss of generality, we assume that the *Alloc* function in the decorated semantics are deterministic up to decorations, which means for three different decorations W, V, $\spadesuit$, the *Alloc* function uses three non-overlap set of heap locations.

By semantics rules, we show transitions for $\tilde{M}_c(P_i, P_g, \mathcal{V})$ until the CPS-transformed code $\mathcal{C}\langle P_i\rangle$ is evaluated.

$$\langle \spadesuit, \varepsilon, null, \tilde{M}_c(P_i, P_g, \mathcal{V}), Q_{init}\rangle_I \to^* \langle \spadesuit, h_c^0, \#global, \mathcal{C}\langle P_i\rangle(\mathsf{function}(_x)\{_x\}), Q\rangle_I$$
$$\text{where } h_c^0 = h_{int} \oplus h_{fra}$$
$$h_{int} = h_{in} \oplus h_p$$
$$h_{fra} = h_f \oplus h_l \oplus h_g$$

where the heap h_c^0 can be separated into a frame part h_{fra} and a non-frame part h_{int}. Furthermore, we have $h_{int}\lfloor_{\mathrm{W}} = h_{int} = h_{in} \oplus h_p$, and h_p is the initial heap and interfaces provided by the proxy library P_p. For the frame part of the heap, we have $h_{fra}\lfloor_{\mathrm{W}} = h_f \oplus h_l$, and h_l which is the initial frame heap and interfaces provided by the listener library P_l. The gadget's heap is $h_{fra}\lfloor_{\mathrm{V}} = h_g$.

Assuming $\mathcal{V} = \{x_1, \ldots x_n\}$, by the definition of bootstrapping code $Bootstrap_i^{\mathcal{V}}$ and $Bootstrap_g^{\mathcal{V}}$, for each x_i, $h_c(\#global).``x_i"_{\{\mathrm{W}\}}$ holds an opaque object handle with an id of $i$. The mapping from opaque object handle to object maintained in $h_l$ by $h_c^0(\#global_f).``handle_list"_{\{\mathrm{W}\}}$ maps i to $h_c^0(\#global_f).``x_i"_{\{\mathrm{V}\}}$. Q is in the following form

$$Q = \langle \ell_p, \varepsilon\rangle \parallel \langle \ell_l, \varepsilon\rangle$$

where $h_c^0(\ell_p)$ is the proxy event handler, and $h_c^0(\ell_l)$ is the listener event handler.

By semantics rules, we show transitions for $\tilde{M}_i(P_i, P_g)$, until the CPS-transformed code is evaluated.

$$\langle \spadesuit, \varepsilon, null, \tilde{M}_i(P_i, P_g), Q_{init}\rangle_I \to^* \langle \spadesuit, h_i^0, \#global, \mathcal{C}'\langle P_i\rangle(\mathsf{function}(_x)\{_x\}), Q_{init}\rangle_I$$
$$h_i^0 = h_{in} \oplus h_g$$

where $h_i^0\lfloor_{\mathrm{W}} = h_{in}$ and $h_i^0\lfloor_{\mathrm{V}} = h'_g$. By Assumption 2, we have

$$h_i^0\lfloor_{\mathrm{V}} = h_c^0\lfloor_{\mathrm{V}} = h_g = h'_g$$

and h_g is a benign gadget heap.

Let $f_0 = \{1 \mapsto h_g(\#global_f).``x_1"_{\{\mathrm{V}\}}, \ldots, n \mapsto h_g(\#global_f).``x_n"_{\{\mathrm{V}\}}\}$, we have

$$h_c^0 \stackrel{f_0}{=} h_i^0$$

By Lemma 4, both $\mathcal{C}\langle P_i\rangle$ and $\mathcal{C}'\langle P_i\rangle$ are simple, which means that they will not try to assign to a heap location not decorated with $\spadesuit$.

We conclude the proof by the following inlined lemma.

Lemma 13. *Let h_c^0 and h_i^0 be two heaps, f be a partial injective function such that $h_c^0 \stackrel{f}{=} h_i^0$, $h_p \subseteq h_c^0$ and $h_l \subseteq h_c^0$. If $\langle \spadesuit, h_i^0, \#global, \mathcal{C}'\langle P_i\rangle(\mathsf{function}(x)\{s_i\}), Q_{init}\rangle_I \to^* \langle \spadesuit, h'_i, \ell, s_i, Q_{init}\rangle_I$, then we have $\langle \spadesuit, h_c^0, \#global, \mathcal{C}\langle P_i\rangle(\mathsf{function}(x)\{s_c\}), Q\rangle_I \to^* \langle \spadesuit, h'_c, \ell, s_c, Q\rangle_I$ and there exist f' such that $h'_c \stackrel{f'}{=} h'_i$, $h_p \subseteq h'_c$ and $h_l \subseteq h'_c$.*

Proof. We prove by induction on the definition of P_i. We exam important cases, other cases are trivial since the CPS compilation for $\mathcal{C}\langle P_i\rangle$ and $\mathcal{C}'\langle P_i\rangle$ are identical.
Case this : By definition of the CPS compiler, we have

$$\mathcal{C}\langle \mathsf{this}\rangle = \mathcal{C}'\langle \mathsf{this}\rangle = \ \mathsf{function}(_k)\{_k(_this)\}$$

Since $h_c^0 \stackrel{f}{=} h_i^0$, therefore we have

$$\ell_s = \text{Scope}(h_c^0, \ell, \text{``}_this\text{''}) = \text{Scope}(h_i^0, \ell, \text{``}_this\text{''})$$

We have to case to analyze here:

1. If $h_c^0(\ell_s) = \ell'$ hold an opaque object handle with id i, then we have $h_i^0(\ell_s) = \ell'' = f(i)$;
2. If $h_c^0(\ell_s) = \ell'$ points to an normal object, then we have $h_i^0(\ell_s) = \ell'$;
3. If $h_c^0(\ell_s) = pv$ for some pv, then we have $h_i^0(\ell_s) = pv$.

In all cases the continuation $_k$ will be applied to either identical values or f-related opaque object handle and corresponding object, therefore the created scope object by applying $_k$ either contains same value for x, or f-related opaque object handle and corresponding object.
Case $e_0[e_1]$: The only difference of the compilation in both side is the then-branch of if-test. By semantics and inductive hypothesis, we have

$$\langle \spadesuit, h_i^0, \#global, \mathcal{C}'\langle P_i\rangle(\mathsf{function}(x)\{s_i\}), Q_{init}\rangle_I \to^* \langle \spadesuit, h_i^1, \ell, s_i', Q_{init}\rangle_I$$
$$\langle \spadesuit, h_c^0, \#global, \mathcal{C}\langle P_i\rangle(\mathsf{function}(x)\{s_c\}), Q\rangle_I \to^* \langle \spadesuit, h_c^1, \ell, s_c', Q\rangle_I$$

such that $h_c^1 \stackrel{f_1}{=} h_i^1$ for some f_0, by inductive hypothesis, and

$$s_i' = _k(_x1[_x2]);$$
$$s_c' = \mathsf{if}\ (_x1[\text{``}_is_ohandle\text{''}])\ \mathtt{_GET_PROPERTY}(_x1, _x2, _k)\ \mathsf{else}\ _k(_x1[_x2])$$

Therefore there are 2 cases.

1. If-test in s_c evaluate to false: meaning that $_x1$ is not an object defined by gadget. Therefore, we have:
$$\langle \spadesuit, h_c^1, \ell, s_c', Q\rangle_I \to^* \langle \spadesuit, h_c^1, \ell, _k(_x1[_x2]), Q\rangle_I$$
By similar reasoning from the case of this, $_x1[_x2]$ will either evaluated to same value or f_1-related object location in both configuration. After the application of the continuation $_k$ in both case,
$$\langle \spadesuit, h_c^1, \ell, _k(_x1[_x2]), Q\rangle_I \to^* \langle \spadesuit, h_c', \ell', s_c, Q\rangle_I$$
$$\langle \spadesuit, h_i^1, \ell, _k(_x1[_x2]), Q\rangle_I \to^* \langle \spadesuit, h_i', \ell', s_i, Q\rangle_I$$
where $h_c' \stackrel{f_1}{=} h_i'$.
2. If-test in s_c evaluate to true, meaning that $_x1$ in h_i^1 is an object defined by gadget. Therefore we must have
$$\begin{aligned}\langle \spadesuit, h_i^1, \ell, s_c', Q_{init}\rangle_I &\to^* \langle \spadesuit, h_i^1, \ell, _k(\ell_o[m]), Q_{init}\rangle_I\\ &\to^* \langle \spadesuit, h_i^1, \ell, _k(v), Q_{init}\rangle_I\\ &\to^* \langle \spadesuit, h_i^2, \ell', s_i, Q_{init}\rangle_I\end{aligned}$$
We now exam the execution for the proxy library `_GET_PROPERTY` and corresponding listener library.
$$\begin{aligned}\langle \spadesuit, h_c^1, \ell, s_c', Q\rangle_I &\to^* \langle \spadesuit, h_c^1, \ell, \mathtt{_GET_PROPERTY}(\ell_o', m, _k), Q\rangle_I &\text{(19a)}\\ &\to^* \langle \text{W}, h_c^2, \ell_c, \varepsilon, \langle \ell_i, \varepsilon\rangle \parallel \langle \ell_l, msg_1\rangle\rangle_I &\text{(19b)}\\ &\to^* \langle \text{W}, h_c^3, \ell_c', \varepsilon, \langle \ell_i, msg_2\rangle \parallel \langle \ell_l, \varepsilon\rangle\rangle_I &\text{(19c)}\\ &\to^* \langle \text{W}, h_c^4, \ell_c'', _cont(v'), \langle \ell_i, \varepsilon\rangle \parallel \langle \ell_l, \varepsilon\rangle\rangle_I &\text{(19d)}\\ &\to^* \langle \spadesuit, h_c^5, \ell', s_c, Q\rangle_I &\text{(19e)}\end{aligned}$$

By semantics, we have (19a). By inductive hypothesis, we have $h_c^1(\ell_o')._id = n$ and $f_1(n) = \ell_o$. Then the proxy library send a msg

$$msg_1 = \{\texttt{"type":"GET_PROPERTY", "obj":n, "prop":m}\}$$

to the frame. Since we have $f_1(n) = l_o$, therefore the listener library, upon receiving, can obtain l_o from `handle_list`, and get the property `m` from $h_c^2(l_o).m = v'$. By hypothesis $h_c^1 \stackrel{f_1}{\equiv} h_i^1$, we have

$$h_c^1 =_{\mathrm{V}} h_i^1$$

therefore we have $v = v'$. If v' is a primitive value, then the listener library sends back a msg

$$msg_2 = \{\texttt{"val":pv}\}.$$

Otherwise it sends back a msg

$$msg_2 = \{\texttt{"val":\{_id:n',_is_ohandle:true\}}\}.$$

where n' is a unique id for new opaque object handle. We can extend f_1 to $f_2 = f_1 \cup \{n' \mapsto v'\}$. Upon receiving this message from the frame, the proxy library now restore the execution by apply the continuation $_cont$ to the received value or opaque object handle. Finally in (19e) $_x$ in the scope is bind to either the primitive value pv or an opaque object handle with id n', where $f_2(n') = v$. We also have

$$h_c^5 \stackrel{f_2}{\equiv} h_i^2$$

Case $e_0(e_1)$: The style of reasoning of this case is similar to the previous case. The only difference of the compilation in both side is the then-branch of if-test. By semantics and inductive hypothesis, we have

$$\langle \spadesuit, h_i^0, \#global, \mathcal{C}'\langle P_i \rangle(\mathsf{function}(x)\{s_i\}), Q_{init}\rangle_I \to^* \langle \spadesuit, h_i^1, \ell, s_i', Q_{init}\rangle_I$$
$$\langle \spadesuit, h_c^0, \#global, \mathcal{C}\langle P_i \rangle(\mathsf{function}(x)\{s_c\}), Q\rangle_I \to^* \langle \spadesuit, h_c^1, \ell, s_c', Q\rangle_I$$

such that $h_c^1 \stackrel{f_1}{\equiv} h_i^1$ for some f_1, by inductive hypothesis, and

$$s_i' = \mathsf{if}\ (_x1[\text{``}_is_ohandle\text{''}])\ _k(_x1(_x2))\ \mathsf{else}\ _x1(_k, _x2)$$
$$s_c' = \mathsf{if}\ (_x1[\text{``}_is_ohandle\text{''}])\ \texttt{_CALL_FUNCTION}(_x1, _x2, _k)\ \mathsf{else}\ _x1(_k, _x2)$$

Therefore there are 2 cases.

1. If-test in s_c evaluate to false, meaning that $_x1$ is not an object defined by gadget. Therefore, we have:

$$\langle \spadesuit, h_c^1, \ell, s_c', Q\rangle_I \to^* \langle \spadesuit, h_c^1, \ell, _x1(_k, _x2), Q\rangle_I$$
$$\langle \spadesuit, h_i^1, \ell, s_i', Q\rangle_I \to^* \langle \spadesuit, h_i^1, \ell, _x1(_k, _x2), Q\rangle_I$$

   By similar reasoning from the case of this, since $_x1$ is a CPS-style function defined by the integrator, then by inductive hypothesis, we have the following transitions. After the application of the continuation $_k$ in both case,

$$\langle \spadesuit, h_c^1, \ell, _x1(_k, _x2), Q\rangle_I \to^* \langle \spadesuit, h_c', \ell', s_c, Q\rangle_I$$
$$\langle \spadesuit, h_i^1, \ell, _x1(_k, _x2), Q\rangle_I \to^* \langle \spadesuit, h_i', \ell', s_i, Q\rangle_I$$

   where $h_c' \stackrel{f_2}{\equiv} h_i'$ for some f_2.

2. If-test in s_c evaluate to true, meaning that $_x1$ in h_i^1 is an function object defined by gadget. Therefore we must have

$$\langle \spadesuit, h_i^1, \ell, s_i', Q_{init}\rangle_I \to^* \langle \spadesuit, h_i^1, \ell, _k(\ell_o(v_0)), Q_{init}\rangle_I \tag{20a}$$

$$\to^* \langle \spadesuit, h_i^2, \ell, _k(v), Q_{init}\rangle_I \tag{20b}$$

$$\to^* \langle \spadesuit, h_i^3, \ell', s_i, Q_{init}\rangle_I \tag{20c}$$

Since $h_i^2\lfloor_{\mathrm{V}}$ is a simple gadget's heap, by Assumption 9 and 3, we have v_0 is either a primitive a value or it points to a simple object. We also have that $h_i^1\lfloor_\spadesuit = h_i^2\lfloor_\spadesuit$. We now only exam the case where v_0 points to a simple object. The other case is straightforwardly similar to this case.

We now exam the execution for the proxy library `_CALL_FUNCTION` and corresponding listener library.

$$\langle \spadesuit, h_c^1, \ell, s_c', Q\rangle_I \to^* \langle \spadesuit, h_c^1, \ell, _\texttt{CALL_FUNCTION}(\ell_o', v_0, _k), Q\rangle_I \tag{20d}$$

$$\to^* \langle \mathrm{W}, h_c^2, \ell_c, \varepsilon, \langle \ell_i, \varepsilon\rangle \parallel \langle \ell_l, msg_1\rangle\rangle_I \tag{20e}$$

$$\to^* \langle \mathrm{W}, h_c^3, \ell_c', \varepsilon, \langle \ell_i, msg_2\rangle \parallel \langle \ell_l, \varepsilon\rangle\rangle_I \tag{20f}$$

$$\to^* \langle \mathrm{W}, h_c^4, \ell_c'', _cont(v'), \langle \ell_i, \varepsilon\rangle \parallel \langle \ell_l, \varepsilon\rangle\rangle_I \tag{20g}$$

$$\to^* \langle \spadesuit, h_c^5, \ell', s_c, Q\rangle_I \tag{20h}$$

By semantics, we have (20d). By inductive hypothesis, we have $h_c^1(\ell_o')._id = n$ and $f_1(n) = \ell_o$. Then the proxy library send a msg

$$msg_1 = \{\texttt{"type":"CALL_FUNCTION", "obj":n, "arg":\{...\}}\}$$

to the frame. Since we have $f_1(n) = l_o$, therefore the listener library, upon receiving, can obtain l_o from **`handle_list`**. By the *stringify-parse* assumption, on the frame side, we can obtain an exactly same simple object. By hypothesis $h_c^1 \overset{f_1}{\equiv} h_i^1$, we have

$$h_c^1 =_{\mathrm{V}} h_i^1 \tag{20i}$$

which are benign gadget's heap. By applying l_o to the same simple object as in (20a). Therefore we would obtain the same result v as in (20b). If v' is a primitive value, then the listener library sends back a messsage

$$msg_2 = \{\texttt{"val":pv}\}.$$

Otherwise it sends back a msg

$$msg_2 = \{\texttt{"val":\{_id:n',_is_ohandle:true\}}\}.$$

where n' is a unique id for new opaque object handle. We can extend f_1 to $f_2 = f_1 \cup \{n' \mapsto v'\}$. Upon receiving this message from the frame, the proxy library now restore the execution by apply the continuation $_cont$ to the received value or opaque object handle. Finally in (20h) $_x$ in the scope is bind to either the primitive value pv or an opaque object handle with id n', where $f_2(n') = v$. We also have

$$h_c^5 \overset{f_2}{\equiv} h_i^2 \tag{20j}$$

The proof of rest cases follows the same style of reasoning. □

We conclude the proof of Lemma 12 with above inlined lemma. □

6 Security Theorem

In this section we present the security theorem. In Mashic compiled code, the integrator has complete access to gadget resources but the gadget only has access to resources offered by the integrator in the proxy library. After Mashic compilation, the malicious gadget cannot scan properties of the integrator, as e.g. in Listing 4, because the SOP policy prevents the iframed gadget from accessing the JS execution environment of the integrator as shown in the DFrameInitrule in Figure 7.

Example 13 (Gadget modifies native functions). For example, a native function that can commonly appear in the integrator code is the `setTimeout` function. This function takes two parameters, the first one is a function that will be executed when the time (in milliseconds) specified in the second parameter has passed:

```
1 setTimeout("alert(timeout!!)",5);
```

In this example, after 5ms a pop-up window with caption "timeout!" appears.

This function, as all native JS functions, is associated as a property with the global object of the document created when the HTML file is loaded. As many native functions the code associated to the `setTimeout` function can be changed at execution time, changing in this way the assumed behavior for `setTimeout`.

Suppose the untrusted gadget owned by the attacker writes a function of its own into the `setTimeout` property:

```
1 setTimeout=function (x,y) { evil code here} ;
```

Then every call to `setTimeout` in the integrator's code will be calling the attacker's code with the integrator privileges.

If instead the gadget is enclosed in a frame, the same code trying to affect the `setTimeout` property of the global object will only affect the property of the global object of the frame, that is in a disjoint part of the heap according to the SOP.

In order to state the security guarantee, we consider that all code coming from origin u is part of the gadget principal V. In contrast to the decorations used for correctness, we now consider the listener library and bootstrapping as gadget's code. This is not surprising since the gadget can modify this code and we want the security theorem to be valid still in this case.

We decorate all code residing in the integrator with ♠. This is also different from the correctness theorem. Essentially, we are now interested in asserting that the gadget cannot change the proxy library or bootstrapping in the integrator, whereas for the correctness theorem we were interested in heap indistinguishability only for the integrator heap in original and compiled mashup. Furthermore, we assume h_{in} is decorated with ♠, and h^f_{in} is decorated with V.

(Notice that decorations do not affect the compiler or semantics of JS code and are only used as technical instrumentation for the theorems and their proofs.)

Definition 22 (Decorated Mashic Compilation (for security theorem)). Given an integrator script P_i and a gadget script P_g, a set of variable $\mathcal{V}$ denoting global names exported by the gadget script, we define the Mashic compilation $\dot{M}_c(P_i, P_g, \mathcal{V})$ to be:

```
<html>
    <iframe src=u></iframe>
    <script♠>
      P_p; Bootstrap_i^V; C⟨P_i⟩(function(_x){_x})
    </script>
</html>
```

where

$$\mathsf{Web}(u) = \texttt{<scriptV>}\ P_l; P_g; \mathit{Bootstrap}_g^{\mathcal{V}}\ \texttt{</script>}$$

Example 14 (Integrity violation). In the example referred just above, the initial heap contains the native function *setTimeout*. Since the initial heap is decorated with ♠, the shape of the global object of the integrator is:

$$h(\#global) = \left\{ \begin{array}{cc} \vdots & \\ \text{“}timeout\text{”}_{\{\spadesuit\}} : & \ell \\ \vdots & \end{array} \right\}$$

By using decoration of Definition 22 and Rule DMODIFY-PROPERTY, we get that the projection $h\lfloor_{\spadesuit}$ of the integrator heap before execution of the gadget and projection $h'\lfloor_{\spadesuit}$ after execution of the gadget do not coincide. The `setTimeout` property of the integrator's global object has been changed by the gadget execution. This represents an integrity violation.

REFLEXIVITY
$$(\square, h, \ell, s) \rightarrow^* (\square, h, \ell, s)$$

TRANSITIVITY
$$\frac{(\square, h, \ell, s) \rightarrow (\cdot, h_0, \ell_0, s_0) \qquad (\cdot, h, \ell, s) \rightarrow^* (\triangle, h', \ell', s')}{(\square, h, \ell, s) \rightarrow^* (\triangle, h', \ell', s')}$$

Figure 13: Semantics Rules for Transitivity

Example 15 (Confidentiality violation). Recall variable `secret` in example of Section 2. Let us assume that the gadget's heap is $h\lfloor_{\mathrm{V}}$.

After execution of the non-benign gadget in Listing 4 with integrator's global object:

$$h(\#global) = \left\{ \begin{array}{cc} & \vdots \\ \text{“}secret\text{”}_{\{\spadesuit\}} : & \text{“}yes\text{”} \\ & \vdots \end{array} \right\}$$

the gadget heap is $h \lfloor_{\mathrm{V}} (\#global_f).\text{“}steal\text{”} = \text{“}yes\text{”}$.

but starting with integrator's global object:

$$h(\#global) = \left\{ \begin{array}{cc} & \vdots \\ \text{“}secret\text{”}_{\{\spadesuit\}} : & \text{“}no\text{”} \\ & \vdots \end{array} \right\}$$

the gadget heap is $h \lfloor_{\mathrm{V}} (\#global_f).\text{“}steal\text{”} = \text{“}no\text{”}$. This difference depends on the integrator's heap and this represents a confidentiality violation.

We show that for any gadget code P_g, and any integrator code P_i, the Mashic compilation $\dot{M}_c(P_i, P_g, \mathcal{V})$ provides integrity and confidentiality guarantees:

Theorem 2 (Security Guarantee of Integrator)**.** *Let P_g and P_i be gadget and integrator code respectively, and let $\mathcal{V}$ be a set of variables. For any configuration reachable from initial Mashic compilation $\dot{M}_c(P_i, P_g, \mathcal{V})$:*

$$\langle \spadesuit, \varepsilon, null, \dot{M}_c(P_i, P_g, \mathcal{V}), Q_{init} \rangle_I \rightarrow^* \langle \mathrm{V}, h, \ell, s, Q \rangle_F$$

if

$$\langle \mathrm{V}, h, \ell, s, Q \rangle_F \rightarrow \langle \mathrm{V}, h', \ell', s', Q' \rangle_F$$

then we have

1. *(integrity.)* $h\lfloor_{\spadesuit} = h'\lfloor_{\spadesuit}$ *;*
2. *(confidentiality.) For any h_0 such that $h_0\lfloor_{\mathrm{V}} = h\lfloor_{\mathrm{V}}$, we have $\langle \mathrm{V}, h_0, \ell, s, Q \rangle_F \rightarrow \langle \mathrm{V}, h'_0, \ell', s', Q' \rangle_F$, and $h'_0\lfloor_{\mathrm{V}} = h'\lfloor_{\mathrm{V}}$.*

The proof of security proceeds by induction on the length of the execution and is simpler than the one of the correctness theorem. It uses the SOP modeled in Rule DFRAMEINITof the semantics. Although the ultimate goal of the Mashic compiler is to provide the security guarantees listed in the theorem, the real technical challenge resided in providing this security result by preserving functionality under hypotheses not affecting current practices in existing mashups code (correctness theorem).

6.1 Proofs

We define the transitivity rules of the semantics explicitly in Figure 13.

The following lemma shows the same-origin policy guaranteed by our semantics, that is, the execution of the integrator and the frame does not interfere with each other.

Lemma 14 (Same-Origin Policy)**.** *Given a global configuration $\langle h, \ell, s\ J, Q \rangle_x$ reachable from any initial configuration, the following facts hold:*

1. $h = h_i \oplus h_f$: *The heap h can be separated into two non-overlapping part.* h_i *is the heap of the integrator,* h_f *is the heap of the frame of different origin.*
2. $\langle h_i \oplus h_f, \ell, s\ J, Q\rangle_I \to \langle h'_i \oplus h_f, \ell', s'\ J, Q'\rangle_I$ *or* $\langle h_i \oplus h_f, \ell, s\ J, Q\rangle_F \to \langle h_i \oplus h'_f, \ell', s'\ J, Q'\rangle_I$ *holds;*

The following lemma shows that if a heap is uniformly colored with $\square$, and the configuration is decorated with the same color $\square$, then after one semantics step the heap is still uniformly colored.

Lemma 15 (Color Preservation). *Given a heap h such that* $h\lfloor_\square = h$, *for any* ℓ, s,

$$(\square, h, \ell, s) \xrightarrow{lmq} (\square, h', \ell', s')$$

and $h'\lfloor_\square = h'$. *Further if* $\ell mq = \ell + mq$ *or* ℓ, *then* $h(\ell).@body_{\{\square\}} = \mathsf{function}(x)\{s\}$,

Proof. We prove by the height t of the derivation tree for the transition relation $\to$ and $\to^*$.

Case $t = 0$: The possible cases are rules Reflexivity, New-Finish, Call-Finish, Call-Return, Var, Block-Next, Return. By the semantics rule we have $h = h'$ therefore $h'\lfloor_\square = h'$.

Case $t = 1$: For rules This,Typeof, GetV-Identifier, Op-Num, GetV-Property, Stringify-Function, PostMsg-Function, by the semantics rule we have $h = h'$ therefore $h'\lfloor_\square = h'$.

For rule AddListener, we have $s = \mathbf{C}[\ell_1(\ell_l)]$ such that $(\square, h, \ell, s = \mathbf{C}[\ell_1(\ell_l)]) \xrightarrow{\ell_l} (\square, h, \ell, undefined)$. By semantics rules we have $\ell_1 = \#Addlistener$ or $\#Addlistener_f$, and $h(\ell_l).@body_{\{\square\}} = \mathsf{function}(x)\{s\}$.

For rule New, we have

$$h' = h \cup \left\{ \begin{array}{ll} \ell_{o\{\square\}} \mapsto & \{@prototype : \ell_p\}, \\ \ell_{s\{\square\}} \mapsto & \left\{ \begin{array}{ll} @scope : & \ell'_s \\ @prototype : & null \\ @this : & \ell_o \\ "x" : & v \end{array} \right\} \end{array} \right\}$$

therefore we have $h'\lfloor_\square = h'$.

For rule Call-Method, Call-Function, Asgn-Identifier, Asgn-Property, Parse-Function we reason similarly.

Case $t > 1$: For rule Obj-Literal, we have

$$s = \mathbf{C}[\{m_1 : e_1, \ldots, m_n : e_n\}]$$

By the semantics rule, we have

$$\begin{array}{c} (\square, h, \ell, e_1) \xrightarrow{mq_1}{}^* (\square, h_1, \ell, v_1) \\ \vdots \\ (\square, h_{n-1}, \ell, e_n) \xrightarrow{mq_n}{}^* (\square, h_n, \ell, v_n) \end{array}$$

The height of derivation tree of those transitions are at most $t - 1$, by inductive hypothesis we have $h_i\lfloor_\square = h_i$ for $i \in \{1, \ldots, n\}$. By semantics rule, we also have

$$h' = h_n \cup \left\{ \ell_{o\{\square\}} \mapsto \left\{ \begin{array}{ll} @prototype : & \#objprot \\ "m_1" : & v_1 \\ \vdots & \vdots \\ "m_n" : & v_n \end{array} \right\} \right\}$$

Therefore $h'\lfloor_\square = h'$.

Other cases are similar by applying the inductive hypothesis. $\square$

The following lemma shows that, by the decorated Mashic compilation defined in this section, the integrator's heap is always uniformly ♠-colored. The frame's heap is always uniformly V-colored. The proof proceeds by examing the execution of a mashic compilation result.

Lemma 16 (Heap Separation)**.** *Given a gadget code P_g, an integrator code P_i, a set of variable $\mathcal{V}$, for any configuration reachable from initial Mashic compilation $\dot{M}_c(P_i, P_g, \mathcal{V})$:*

$$\langle \spadesuit, \varepsilon, null, \dot{M}_c(P_i, P_g, \mathcal{V}), Q_{init}\rangle_I \rightarrow^* \langle \square, h, \ell, s, Q\rangle_F$$

we have $h = h_i \oplus h_f$, $h\lfloor_{\spadesuit} = h_i$ and $h\lfloor_{\mathrm{V}} = h_f$.

Proof. By semantics rules, we have

$$\begin{aligned}\langle \spadesuit, \varepsilon, null, \texttt{<html>}\ FJ\ \texttt{</html>}, Q_{init}\rangle_I &\xrightarrow{\text{rule INIT}} \langle \spadesuit, h_0, \#global, FJ, Q_{init}\rangle_I \\ &\xrightarrow{\text{rule FRAME}} \langle \spadesuit, h_1, \#global_f, F'J, Q_{init}\rangle_F \\ &\xrightarrow{\text{rule SCRIPT, FRAME-EXEC}} \langle \mathrm{V}, h_1, \#global_f, F''J, Q_{init}\rangle_F\end{aligned}$$

where

$$\begin{aligned} h_0 =& h_{in} \\ h_1 =& h_{in} \oplus h^f_{in} \\ F =& \texttt{<iframe src=}u\texttt{></iframe>} \\ F' =& \texttt{<frame><script}\mathrm{V}\texttt{>}\ P_l; P_g; Bootstrap^{\mathcal{V}}_g\ \texttt{</script></frame>} \\ F'' =& \texttt{<frame>}P_l; P_g; Bootstrap^{\mathcal{V}}_g\texttt{</frame>} \\ J =& \texttt{<script}\spadesuit\texttt{>}\ P_p; Bootstrap^{\mathcal{V}}_i; \mathcal{C}\langle P_i\rangle(\mathsf{function}(_x)\{_x\}\ \texttt{</script>} \end{aligned}$$

By Lemma 14 and Lemma 15, we have

$$\begin{gathered}\langle \mathrm{V}, h^f_{in}, \#global_f, F'J, Q_{init}\rangle_F \rightarrow^* \langle \mathrm{V}, h^f_2, \#global_f, \texttt{<iframe>}\ v\ \texttt{</iframe>}J, Q_0\rangle_F \\ \langle \mathrm{V}, h_{in} \oplus h^f_{in}, \#global_f, F'J, Q_{init}\rangle_F \rightarrow^* \langle \mathrm{V}, h_{in} \oplus h^f_2, \#global_f, \texttt{<iframe>}\ v\ \texttt{</iframe>}J, Q_0\rangle_F \\ h^f_2\lfloor_{\mathrm{V}} = h^f_2\end{gathered}$$

and $Q_0 = \langle \varepsilon, \varepsilon\rangle \parallel \langle \ell_f, \varepsilon\rangle$ and $\ell_f \in \mathsf{dom}(h^f_2)$. By semantics rules,

$$\langle \mathrm{V}, h_{in} \oplus h^f_2, \#global_f, \texttt{<iframe>}\ v\ \texttt{</iframe>}J, Q_0\rangle_F \rightarrow^* \langle \spadesuit, h_{in} \oplus h^f_2, \#global, s, Q_0\rangle_I$$

By Lemma 14 and Lemma 15, we have

$$\begin{gathered}\langle \spadesuit, h_{in} \oplus h^f_2, \#global, s, Q_0\rangle_I \rightarrow^* \langle \spadesuit, h_2 \oplus h^f_2, \#global, v, Q_1\rangle_I \\ h_2\lfloor_{\spadesuit} = h_2\end{gathered}$$

and $Q_1 = \langle \ell_i, \varepsilon\rangle \parallel \langle \ell_f, mq_f\rangle$, and $\ell_i \in \mathsf{dom}(h_2)$. We now prove that for any $Q = \langle \ell_i, m+mq_i\rangle \parallel \langle \ell_f, mq_f\rangle$, for any $h_i \oplus h^f_i$ such that $h_i\lfloor_{\spadesuit} = h_i$, $h^f_i\lfloor_{\mathrm{V}} = h^f_i$, $\ell_i \in \mathsf{dom}(h_i)$, and $\ell_f \in \mathsf{dom}(h^f_i)$, we have

$$\begin{aligned}\langle \square, h_i \oplus h^f_i, \#global, \varepsilon, Q\rangle_x &\rightarrow \langle \spadesuit, h_i \oplus h^f_i, \#global, \ell_i(m), Q'\rangle_I \\ &\rightarrow^* \langle \spadesuit, h'_i \oplus h^f_i, \#global, v, Q''\rangle_I\end{aligned}$$

By Lemma 14 and Lemma 15, we have $h'_i\lfloor_{\spadesuit} = h'_i$ and $Q'' = \langle \ell'_i, mq_i\rangle \parallel \langle \ell_f, mq'_f\rangle$ such that $\ell'_i \in \mathsf{dom}(h'_i)$. The reasoning of the symmetric case where the frame's callback is executed is similar. □

Now we can prove the main security theorem.

Proof of Theorem 2. By Lemma 16, we have

$$h = h_i \oplus h_f \tag{23a}$$
$$h\lfloor_{\spadesuit} = h_i \tag{23b}$$
$$h\lfloor_{\mathrm{V}} = h_f \tag{23c}$$

By Remark 14, we have

$$h' = h_i \oplus h'_f \tag{23d}$$
$$\langle \mathrm{V}, h_f, \ell, s, Q\rangle_F \rightarrow \langle \mathrm{V}, h'_f, \ell', s', Q'\rangle_F \tag{23e}$$

It remains to show that $h'\lfloor_{\mathrm{V}} = h'_f$, which is equivalent to $h'_f\lfloor_{\mathrm{V}} = h'_f$. By semantics rule, we have

$$(\mathrm{V}, h_f, \ell, s) \rightarrow (\mathrm{V}, h'_f, \ell', s')$$

By Lemma 15, we have $h'_f\lfloor_{\mathrm{V}} = h'_f$. □

7 Implementation and Case Studies

In this section we discuss practical issues regarding the implementations of the Mashic compiler and cases studies with well-known gadget APIs. The Mashic compiler is written in Bigloo [3] (a dialect of Scheme) and JS. It has 3.3k lines of Bigloo code and 0.8k lines of JS code.

Optimizations Since JS does not support any tail-recursive call optimization, CPS-transformed code can easily run out of call stacks. In order to deal with this, we implement a trampoline mechanism as proposed by Loitsch [2009]. We define a global variable `counter` to count the depth of current call stacks. If the counter exceeds a certain limit (in the following example it is 30)a tail call will return a trampoline object instead of invoking the function.

This is shown in Listing 15.

```
1 if (counter > 30)
2    return new Trampoline(fun, arg);
3    return fun(arg);
```

Listing 15: Trampoline of Tail Call

A guard loop, on the top level, detects if a trampoline object is returned, as shown in Listing 16. If a trampoline object is detected, the loop restarts the execution of the tail call.

```
1 res_or_tramp=fun(arg);
2 while (res_or_tramp instanceof Trampoline)
3    res_or_tramp = res_or_tramp.restart();
```

Listing 16: Guard Loop of Trampoline Execution

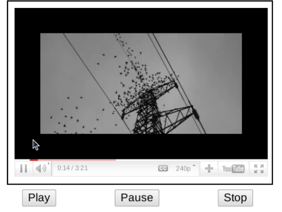
Event Handler In mashups, we also find demands for registering integrator-defined functions as event handlers of gadgets' DOM objects. For example, the Google Maps API provides an interface to set an integrator's function as a handler of the event of clicking on the map. Every time the map is clicked, the corresponding function will be invoked, to notify the integrator of the event. By SOP, the integrator and the gadget in a Mashic compilation cannot exchange function references. Hence we design and implement a mechanism called *Opaque Function Handle* to achieve the same functionality of event handler. Similar to the opaque object handle, we associate opaque function handles with function objects on the integrator side. When an iframe-sandboxed gadget receives a function handle, it creates a warp function by using the function shown in Listing 17.

```
1 function wrap_fun(fhandle){
2    return function(arg){
3       var msg = { fun : fhandle,
4                   msg_type : 'CALLBACK',
5                   argument : arg};
6       PostMessage(stringify(msg));
7       return;
8    };
9 }
```

Listing 17: Wraping Function Handle

[3] `http://www-sop.inria.fr/mimosa/fp/Bigloo/`

(a) Map Directions



(b) Youtube Player Controls

Figure 14: Case Studies

The wrapped function, upon each invocation, sends a message to notify the integrator to invoke the function associated with the function handle.

Case Studies We have successfully applied our compiler to mashups which use well-known gadget APIs, such as Google Maps API, Bing Maps API, Youtube API, and Zwibbler API[4]. Those examples involve non-trivial interactions between the integrator and the gadget. In Figure 14 we show two concrete examples. The first example is a mashup using Google Maps API to calculate driving directions between two cities. The map gadget is sandboxed by the Mashic compiler in an iframe, as indicated by a black box in the figure. The compiled integrator, as in the original integrator, permits to choose a starting point and an ending point to display a route in the map. The gadget's response displayed by the integrator, is the distance between the two points. The latter example shows a sandboxed Youtube player, where one can control the behavior of the player though buttons in the integrator.

From an end-user point of view, one can barely feel the latencies introduced by Mashic compilation, compared to the original mashup.

For the 25 examples of Google Maps API we have studied, we have successfully compiled 23 of them. The other 2 examples not supported by the Mashic compiler are `overlay-remove` and `overlay-simple`. The example of `overlay-remove` uses the for-in construct which are not currently supported by our compiler. In the `overlay-simple` example, the integrator uses some gadget's object as the prototype of an integrator's object, which not allowed by Assumption 3 (The simple integrator assumption).

Unsupported Constructs Our CPS transformer in the Mashic compiler currently supports the full JS language [ECMA, 2009] except for a few programming constructs. Specifically, the *for-in* construct and *exception* construct are not supported.

8 Conclusion

We have proposed the Mashic compiler as an automatic process to secure existing real world mashups. The Mashic compiler can offer a significant practical advantage to developers in order to effortlessly write secure mashups without giving up on functionality.

Compiled code is guaranteed to satisfy integrity and confidentiality constraints of integrator's sensitive resources. We do not address in this paper analysis to prevent security vulnerabilities introduced by the integrator's code. Consider the following silly code:

```
1 CALL_METHOD(eval,opq_obj,"foo",{});
```

This integrator will `eval` the result from calling the `foo` method of the opaque object handle `opq_obj`. The gadget might return some string representing a malicious JS program. Then the integrator will execute the malicious code with its own privilege.To avoid this kind of vulnerabilities, analysis of the integrator's code is required. This is orthogonal to the current Mashic compilation and we leave it as future work.

[4]For instance, we have compiled several examples obtained from `http://code.google.com/apis/maps/documentation/javascript/examples/index.html`

Mashic offers correctness guarantess *only if* the untrusted gadget is benign. This is a goal of the compiler and not a disadvantage: mashup behavior should *not* be the same if the gadget is malicious. If the gadget is malicious the programmer does not get any alert that the compiled secured mashup does not behave as the original mashup. Another interesting future direction will be to provide JS code analyses that will conservatively detect non-benign gadgets in order to alert the programmer.

References

Adam Barth, Collin Jackson, and William Li. Attacks on JavaScript Mashup Communication. In *Proceedings of Web 2.0 Security and Privacy 2009 (W2SP 2009)*, 2009a.

Adam Barth, Collin Jackson, and John C. Mitchell. Securing Frame Communication in Browsers. *Commun. ACM*, 52(6):83–91, 2009b.

Adam Barth, Joel Weinberger, and Dawn Song. Cross-origin Javascript Capability Leaks: Detection, Exploitation, and Defense. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 187–198, Berkeley, CA, USA, 2009c. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1855768.1855780`.

Gérard Boudol. Typing termination in a higher-order concurrent imperative language. *Inf. Comput.*, 208:716–736, June 2010. ISSN 0890-5401. doi: http://dx.doi.org/10.1016/j.ic.2009.06.007. URL `http://dx.doi.org/10.1016/j.ic.2009.06.007`.

Steven Crites, Francis Hsu, and Hao Chen. OMash: Enabling Secure Web Mashups via Object Abstractions. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 99–108. ACM, 2008. ISBN 978-1-59593-810-7.

Douglas Crockford. The `<module>` Tag , 2010. http://www.json.org.

Douglas Crockford. ADsafe, 2011. http://www.adsafe.org/.

Olivier Danvy and Andrzej Filinski. Representing control: A study of the cps transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

ECMA. ECMAScript Language Specification. Technical report, ECMA International, December 2009. http://www.ecma-international.org/.

P. Gardner, G. Smith, M.J. Wheelhouse, and U. Zarfaty. DOM: Towards a Formal Specification. In *Proceedings of the ACM SGIPLAN workshop on Programming Language Technologies for XML (PLAN-X)*, California, USA, January 2008. ACM Press.

Dan Grossman, J. Gregory Morrisett, and Steve Zdancewic. Syntactic Type Abstraction. *ACM Trans. Program. Lang. Syst.*, 22(6):1037–1080, 2000.

Ian Hickson. HTML5. Technical report, W3C, May 2011.

Arnaud Le Hors, Philippe Le Hegaret, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document Object Model (DOM) level 2 Core Specification. Technical report, W3C, November 2000.

Facebook Inc. Facebook Javascript Subset, 2011a. http://facebook.com.

Google Inc. Google Caja Project, 2011b. http://google.com.

Collin Jackson and Helen J. Wang. Subspace: Secure Cross-domain Communication for Web Mashups. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *WWW*, pages 611–620. ACM, 2007. ISBN 978-1-59593-654-7.

Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An Empirical Study of Privacy-violating Information Flows in JavaScript Web Applications. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 270–283. ACM, 2010. ISBN 978-1-4503-0245-6.

Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. SMash: Secure Component Model for Cross-domain Mashups on Unmodified Browsers. In Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang, editors, *WWW*, pages 535–544. ACM, 2008. ISBN 978-1-60558-085-2.

Florian Loitsch. *Scheme to JavaScript Compilation*. PhD thesis, Université de Nice - Sophia Antipolis, March 2009.

Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *USENIX Security Symposium*, pages 371–388. USENIX Association, 2010.

S. Maffeis and A. Taly. Language-based Isolation of Untrusted Javascript. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 77–91. IEEE, 2009.

S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. of APLAS'08*, volume 5356 of *LNCS*, pages 307–325, 2008. See also: Dep. of Computing, Imperial College London, Technical Report DTR08-13, 2008.

S. Maffeis, J.C. Mitchell, and A. Taly. Isolating JavaScript with Filters, Rewriting, and Wrappers. In *Proc of ESORICS'09*. LNCS, 2009.

A. Sabelfeld and A.C. Myers. Language-based information-flow security. 21(1):5–19, 2003.

Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *SOSP '07*, pages 1–16, 2007. ISBN 978-1-59593-591-5.

A Mashic Run-time

Proxy Library

```
var _cont;

var _message_handler;
_message_handler = function(m){
    _cont(parse(m));
    _cont = function(){};
};

addeventlistener(_message_handler);

var _GET_GLOBAL_REF;
_GET_GLOBAL_REF = function(name,cont){
    var msg;
    msg = {"type" : "GET_GLOBAL_REF", "name" : name};
    postMessage(msg);
    _cont = cont;
};

var _GET_PROPERTY;
_GET_PROPERTY = function(id,prop,cont){
    var msg;
    msg = {"type" : "GET_PROPERTY", "obj" : id, "prop" : prop};
        postMessage(msg);
    _cont = cont;
};

var _PROPERTY_ASSIGN;
_PROPERTY_ASSIGN = function(id, prop, val, cont){
    var msg;
    msg = {"type" : "PROPERTY_ASSIGN", "obj" : id, "prop" : prop, "val" : val};
    postMsg(msg);
    _cont = cont;
};


var _CALL_FUNCTION;
_CALL_FUNCTION = function(id, arg, cont){
    var msg;
    msg = {"type" : "CALL_FUNCTION", "obj" : id, "arg" : arg};
    postMsg(msg);
    _cont = cont;
};

var _CALL_METHOD;
_CALL_METHOD = function(id, prop, arg, cont){
    var msg;
    msg = {"type" : "CALL_METHOD", "obj" : id, "prop" : prop, "arg" : arg};
    postMsg(msg);
    _cont = cont;
};

var _NEW_OBJECT;
_NEW_OBJECT = function(id, arg, cont){
    var msg;
    msg = {"type" : "_NEW_OBJECT", "obj" : id, "arg" : arg};
    postMsg(msg);
    _cont = cont;
};
```

Listing 18: Proxy Library

Listener Library

```
var OHandle;

OHandle = function(){
    var id;
    id = handle_id_gen();
    this._id = id;
    this._is_ohandle = true;
};

var handle_list = {};
function add_handle(ohandle,obj){
  handle_list[ohandle["_id"]] = obj;
}

var get_obj;
get_obj=function(id){
  return handle_list[id];
}

var message_handler;
message_handler = function(m){
    var recv;
    recv = parse(m);
    var resp;
    dispatch(recv);
};

function dispatch(recv){
    if (recv["type"] === "GET_GLOBAL_REF")
        get_global_ref(recv);
    if (recv["type"] === "GET_PROPERTY")
        get_property(recv);
    if (recv["type"] === "CALL_METHOD")
        call_method(recv);
    if (recv["type"] === "CALL_FUNCTION")
        call_function(recv);
    if (recv["type"] === "NEW_OBJECT")
        new_object(recv);
    if (recv["type"] === "OBJ_PROP_ASSIGN")
        obj_prop_assign(recv);
}

function get_global_ref(recv){
    var val;
    val = window[recv["name"]];
    return respond(val);
}

function get_property(recv){
    var obj;
    obj = get_obj(recv["obj"]);
    var prop;
    prop = recv["prop"];
    return respond(obj[prop]);
}

function call_method(recv){
    var obj;
    obj = get_obj(recv["obj"]);
    var prop;
    prop = recv["prop"];
    var arg;
    arg = recv["arg"];
```

```
64     val = obj[prop](arg);
65     return respond(val);
66 }
67
68 function call_function(recv){
69     var obj;
70     obj = get_obj(recv["obj"]);
71     var arg;
72     arg = recv["arg"];
73     val = obj(arg);
74     return respond(val);
75 }
76
77 function new_object(recv){
78     var obj;
79     obj = get_obj(recv["obj"]);
80     var arg;
81     arg = recv["arg"];
82     val = new obj(arg);
83     return respond(val);
84 }
85
86 function obj_prop_assign(recv){
87     var obj;
88     obj = get_obj(recv["obj"]);
89     var prop;
90     prop = recv["prop"];
91     var arg;
92     arg = recv["arg"];
93     val = obj[prop] = arg;
94     return respond(val);
95 }
96
97 var respond;
98 respond = function(val){
99     var handle;
100    var msg;
101    if (typeof val == "object"){
102        handle = new OHandle();
103        add_handle(handle,val);
104        msg = {"val" : ohandle};
105    } else if (typeof val == "function"){
106        handle = new OHandle();
107        add_handle(handle,val);
108        msg = {"val" : ohandle};
109    } else{
110        msg = {"val" : val};
111    };
112    postMessage(stringify(msg));
113 }
```

Listing 19: Listener Library