Distributed Electronic Rights in JavaScript

Mark S. Miller[1], Tom Van Cutsem[2], and Bill Tulloh

[1] Google, Inc.
[2] Vrije Universiteit Brussel

Abstract. Contracts enable mutually suspicious parties to cooperate safely through the exchange of rights. Smart contracts are programs whose behavior enforces the terms of the contract. This paper shows how such contracts can be specified elegantly and executed safely, given an appropriate distributed, secure, persistent, and ubiquitous computational fabric. JavaScript provides the ubiquity but must be significantly extended to deal with the other aspects. The first part of this paper is a progress report on our efforts to turn JavaScript into this fabric. To demonstrate the suitability of this design, we describe an escrow exchange contract implemented in 42 lines of JavaScript code.

Keywords: security, distributed objects, object-capabilities, smart contracts

1 Smart Contracts for the Rest of Us

The fabric of the global economy is held together by contracts. A contract is an agreed framework for the rearrangement of rights between mutually suspicious parties. But existing contracts are ambiguous, jurisdictions-specific, and written, interpreted, and adjudicated only by expensive experts. *Smart contracts* are contract-like arrangements expressed in program code, where the behavior of the program enforces the terms of the "contract"[1]. Though not a substitute for legal contracts, they can provide some of the benefits of contracts for fine-grain, jurisdiction-free, and automated arrangements for which legal contracts are impractical.

To realize this potential, smart contracts need a distributed, secure, persistent, and ubiquitous computational fabric. To avoid merely substituting one set of expensive experts for another, non-experts should be able to write smart contracts understandable by other non-experts. We[3] are working towards turning JavaScript into such a fabric. JavaScript is already understood and used by many non-expert programmers. We call our target JavaScript platform *Dr. SES* for *Distributed Resilient Secure EcmaScript*.[4]

Dr. SES is not specifically tied to electronic rights (erights) or smart contracts per se. Its focus is to make distributed secure programming in JavaScript as effortless as possible. But much of the design of Dr. SES and its predecessors [2,3,4] was shaped by examining what we need to express smart contracts simply. Taking a rights-based approach to local and distributed computing, we believe, has led us to building a better general purpose platform as well as one naturally suited for expressing new kinds of erights and contracts.

[3] Including many collaborators over many years. See the acknowledgements.
[4] The official standards name for JavaScript is "ECMAScript".

The first half of this paper, section 2, explains the design of Dr. SES and our progress building it. After section 2.2, the rest can be skipped on a first read. Section 3 explains how rights help organize complexity in society in a decentralized manner, addressing many of the problems we face building distributed systems. Section 4 examines an implementation of "money". Section 5 examines an escrow exchange contract. Section 6 examines a generic contract host, able to host this contract and others. Together, they demonstrate the simplicity and expressiveness of Dr. SES.

2 Dr. SES: Distributed Resilient Secure EcmaScript

Dr. SES is a platform for distributed, resilient, and secure computing, layered on JavaScript. How do these ingredients support erights and contracts?

The participants in a contract are typically represented by mutually suspicious machines communicating over open networks. JavaScript is not a distributed programming language. In the browser, a large number of APIs are available to scripts to communicate with servers and other frames, but these APIs do not operate at the level of individual objects. Dr. SES builds on the Q library[5] to extend the JavaScript language with a handful of features to support distributed programming at the level of objects and messages.

In an architecture that aims to express erights or contracts, security must play a key role. Dr. SES uses the Q library to support distributed cryptographic capabilities, and builds on the SES library to support local object-capabilities. The latter allows Dr. SES programs to safely execute mobile code from untrusted parties. This is especially relevant in the context of JavaScript, where mobile code is routinely sent from servers to clients. In Section 6, we will show an example that depends on the ability to safely execute third-party code on servers.

Finally, the resilience aspect of Dr. SES deals with the unavoidable issues of failure handling that come up in distributed systems. Server-side Dr. SES programs periodically checkpoint their state, so that in the event of a failure, the program can always recover from a previously consistent state. Such Dr. SES programs can survive failures without effort on the part of the programmer. Dr. SES builds on the *NodeKen* project, which is layering the Node.js server-side JavaScript platform onto the Ken system [6] for distributed orthogonal persistence—resilient against many failures.

2.1 Just Enough JavaScript

JavaScript is a complex language, but this paper depends only on a small subset with two core constructs, *functions* and *records*. As of this writing, the standard and ubiquitous version of JavaScript is ECMAScript 5 (ES5). For the sake of brevity, this paper borrows one syntactic convenience proposed for ES6, *arrow functions* ("`=>`"), and one proposed for ES7, the *eventual-send operator* ("`!`"). Expanding away these conveniences, all the code here is working ES5 code, and is available at `code.google.com/p/es-lab/source/browse/trunk/src/ses/#ses` and its `contract` subdirectory.

[5] Once the `es-lab.googlecode.com/svn/trunk/src/ses/makeQ.js`, [5], and `https://github.com/kriskowal/q` implementations of Q are reconciled.

Arrow functions. The following four lines all define a one parameter function which returns double its parameter. All bind a local variable named "`twice`" to this function. This paper uses only the arrow function syntax of the last three lines.

```
var twice = function(n) { return n+n; };     // old function expr
var twice = (n) => { return n+n; };          // ES6 arrow function
var twice = (n) => n+n;               // non−"{" expr implicitly returned
var twice = n => n+n;                 // parens optional if one param
```

Records. The record syntax `{x: 3, y: 4}` is an expression that evaluates to a record with two named properties initialized to the values shown. Records and functions compose together naturally to give objects:

```
var makePoint = (x, y) => {
  return {
    getX: () => x,
    getY: () => y,
    add: other => makePoint(x + other.getX(), y + other.getY())
  };
};

var pt = makePoint(3, 5).add(makePoint(2, 7));
```

A record of functions hiding variables serves as an object of methods (`getX, getY, add`) hiding instance variables (`x, y`). The `makePoint` function serves as a class-like factory for making new point instances.

2.2 Basic Concepts of Dr. SES

Dr. SES extends this object model across time and space (persistence and distribution), while relieving programmers of many of the typical worries associated with building secure distributed resilient systems. The non-expert programmer can begin with the following oversimplified understanding of Dr. SES:

SES Don't worry about script injection. Mobile code can't do anything it isn't authorized to do. Functions and objects are encapsulated. Objects can invoke objects they have a reference to, but cannot tamper with those objects.

Q Don't worry about memory races or deadlocks, they can't happen. Objects can be local or remote. The familiar infix dot ("`.`") in `pt.getX()` accesses the `pt` object *immediately*. Q adds the bang "`!`" to access an object *eventually*. Anywhere you can write a dot, you can use "`!`" as in `pt ! getX()`. Eventual operations return promises for what the answer will be. If the object is remote or a promise, you can only use "`!`" on it.

NodeKen Don't worry about network partitions or machine crashes. Once the machine comes back up, everything keeps going, so a crash and restart is just a very long (possibly infinite) pause. Likewise, a partitioned network is just a slow network waiting to heal. Once things come back up, every message ever sent will be delivered in order exactly once.

The above should be adequate to understand the functionality of the smart contract code when things go well. Of course, much of the point of erights and smart contracts is to limit the damage when things go badly. Understanding these risks does require a careful reading of the following sections.

2.3 SES: Securing JavaScript

In a memory-safe object language with unforgeable object references (protected pointers) and encapsulated objects, an object reference grants the right to invoke the public interface of the object it designates. A message sent on a reference both exercises this right and grants to the receiving object the right to invoke the passed arguments.

In an *object-capability* (ocap) language [7], an object can cause effects on the world outside itself *only* by using the references it holds. Some objects are transitively immutable or *powerless* [8], while others might cause effects. An object must not be given any powerful references by default; any references it has implicit access to, such as language-provided global variables, must be powerless. Under these rules, granted references are the *sole* representation of permission.

Secure EcmaScript (SES) is an ocap subset of ES5. SES is lexically scoped, its functions are encapsulated, and only the global variables on its whitelist (including all globals defined by ES5) are accessible. Those globals are unassignable, and all objects transitively reachable from them are immutable, rendering all implicit access powerless.

SES supports *defensive consistency* [7]. An object is *defensively consistent* when it can defend its own invariants and provide correct service to its well behaved clients, despite arbitrary or malicious misbehavior by its other clients. SES has a formal semantics supporting automated verification of some security properties of SES code [9]. The code in this paper uses the following functions from the SES library:

`def(obj)` *def*ines a *def*ensible object. To support defensive consistency, the `def` function makes the properties of its argument read-only, likewise for all objects transitively reachable from there by reading properties. As a result, this subgraph of objects is effectively tamper proof. A *tamper-proof* record of *encapsulated* functions hiding *lexical* variables is a *defensible object*. In SES, if `makePoint` called `def` on the points it returns by saying "`return def({...})`", it would make defensively consistent points.

`confine(exprSrc, endowments)` enables safe mobile code. The `confine` function takes the source code string for a SES expression and an `endowments` record. It evaluates the expression in a new global environment consisting of the SES whitelisted (powerless) global variables and the properties of this endowments record. For example, `confine('x + y', {x: 3, y: 6})` returns 9.

`Nat(allegedNumber)` tests whether `allegedNumber` is indeed a primitive number, and whether it is a non-negative integer (a natural number) within the contiguous range of exactly representable integers in JavaScript. If so, it returns `allegedNumber`. Otherwise it throws an error.

`var m = WeakMap()` assigns to `m` a new empty weak map. WeakMaps are an ES6 extension (emulated by SES on ES5 browsers) supporting *rights amplification* [10]. Ignoring space usage, `m` is simply an object-identity-keyed table.

`m.set(obj,val)` associates `obj`'s identity as key with `val` as value, so `m.get(obj)` returns `val` and `m.delete(obj)` removes this entry. These methods use only `obj`'s identity without interacting with `obj`.

2.4 Q: Distributed JavaScript Objects

To realize erights, we need a distributed, secure, and persistent computational fabric. We have just seen how SES can secure a local JavaScript environment. Here, we focus on how to link up multiple secured JavaScript environments into a distributed system.

Communicating Event-Loop Concurrency JavaScript's de-facto concurrency model, on both the browser and the server, is "shared nothing" *communicating event loops*. In the browser, every frame of a web page has its own event loop, which is used both for updating the UI (i.e. rendering HTML) and for executing scripts. Node.js, the most widespread server-side JavaScript environment, is based on a similar model, although on the server the issue is asynchronous networking and file I/O rather than UI.

In its most general form, an event loop consists of an event queue and a set of event handlers. The event loop processes events one by one from its queue by dispatching to the appropriate event handler. In JavaScript, event handlers are usually functions registered as callbacks on certain events (e.g. button clicks or incoming XHR responses).

The processing of a single event is called a *turn* of the event loop. Processing an event usually entails calling a callback function, which then runs to completion without interruption. Thus, turns are the smallest unit of interleaving.

A system of communicating event loops consists of multiple event loops (in the same or distributed address spaces) that communicate with each other solely by means of asynchronous message passing. The Web Workers API enables such communication among multiple isolated event loops within the same browser. A JavaScript webpage communicating with a Node.js server using asynchronous XHR requests is an example of two distributed communicating event loops.

Communicating event loop concurrency makes it manageable for objects to maintain their invariants in the face of concurrent (interleaved) requests made by multiple clients [11]. While JavaScript environments already support event loop concurrency, the JavaScript language itself has no support for concurrent or distributed programming. Q thus extends JavaScript with a handful of features that enable programmers to more directly express distributed interactions between individual objects.

Promises We introduce a new type of object, a promise, to represent both the outcome of asynchronous operations as well as remote references [12]. A normal JavaScript direct reference may only designate an object within the same event loop. Only promises designate objects in other event loops. A promise may be in one of several states:

pending when it is not yet determined what object the promise designates,
resolved when it is either fulfilled or rejected,
- **fulfilled** when it is resolved to successfully designate some object,
- **rejected** when it will never designate an object, for an alleged reason represented by an associated error.

`var tP = Q(target)` assigns to `tP` a promise for `target`. If `target` is already a promise, that same promise is assigned. Otherwise, `tP` is a fulfilled promise designating `target`.

`Q.promise( (resolve,reject) => (...) )` returns a fresh promise which is initially pending. It immediately calls the argument function with two functions, conventionally named `resolve` and `reject`, that can be used to either resolve or reject this new promise explicitly.

`var resultP = tP.then( (v) => result1, (e) => result2 )` provides eventual access to `tP`'s resolution. The `.then` method takes two callback arguments, a `success` callback and an optional `failure` callback. It registers these callbacks to be called back in a later turn after `tP` is resolved. If `tP` was fulfilled with a value `v`, then `success(v)` is called. If `tP` was rejected with an error `e`, then `failure(e)` is called. `resultP` is a promise for the invoked callback's `result` value.

If the callback invoked by `.then` throws an error, that error is used to reject `resultP`. This propagation of errors along chains of dependent promises is called rejected promise contagion [11], and it is the asynchronous analogue of propagating exceptions up the call stack. If the failure callback is missing, rejecting `tP` will eventually reject `resultP` with the same reason. If `pointP` is a promise for a local `point` object, we may construct a derived point promise as follows:

```
var newP = pointP.then((point) => point.add(makePoint(1,2)));
```

Just like it is useful to compose individual functions into a composite function, it is often useful to compose individual promises into a single promise whose outcome depends on the individual promises. The Q library provides some useful combinator[6] functions we use later in the escrow exchange contract:

`Q.race(answerPs)` takes an array of promises, `answerPs`, and returns a promise for the resolution of whichever promise we notice has resolved first. For example, `Q.race([xP,yP]).then(v => print(v))` will cause either the value of `xP` or `yP` to be printed, whichever resolves first. If neither resolves, then neither does the promise returned by `Q.race`. If the first promise to resolve is rejected, the promise returned by `Q.race` is rejected with the same reason.

`Q.all(answerPs)` takes an array of promises and returns a promise for an array of their fulfilled values. We often need to collect several promised answers, in order to react either when *all* the answers are ready or when *any* of them become rejected. Given `var sumP = Q.all([xP,yP]).then(([x,y]) => x+y),` if both `xP` and `yP` are fulfilled with numbers, `sumP` is fulfilled with their sum. If neither resolves, neither does `sumP`. If either `xP` or `yP` is rejected, `sumP` is rejected with the same reason.

`Q.join(xP,yP)` takes two promises and returns a promise for the one object they both designate. `Q.join` is our eventual equality operation. Any messages sent to the joined promise are only delivered if `xP` and `yP` eventually come to designate

[6] These are easily built from the above primitives. Their implementation can be found at `wiki.ecmascript.org/doku.php?id=strawman:concurrency`.

the same target. In this case, all messages are eventually delivered to that target and the joined promise itself eventually becomes fulfilled to designate that target. Otherwise, all these messages are discarded with the usual rejected promise contagion.

Immediate call and eventual send Promises may designate both local objects, and remote objects belonging to another event loop. If the promise comes to designate a local object (or a primitive value), that value can be accessed via the `.then` method.

However, if the promise comes to designate a remote object, it is not possible to resolve the promise to a local reference. Instead, one must interact with the remote object via the promise. Any such interaction must be asynchronous, to ensure that interaction between the event loops as a whole remains asynchronous.

JavaScript provides many operators to interact with an object. Here, we will focus on only three: method calls, function calls, and reading the value of a property. JavaScript has the familiar dot operator to express local, immediate method calls, such as `point.getX()`. We introduce a corresponding infix "`!`" operator (named the *eventually*) operator, which designates asynchronous, possibly remote interactions.

The `!` operator can be used anywhere the dot operator can be used. If `pointP` is a promise for a `point`, then `pointP ! getX()` denotes an eventual send, which enqueues a request to call the `getX()` method in the event loop of `point`. The syntax `fP ! (x,y)`, where `fP` is a promise designating a function `f`, enqueues a request to call `f(x,y)` in the event loop of `f`. The `!` operator is actually syntactic sugar for calling a method on the promise object itself:

Immediate syntax	Eventual syntax	Expansion
`p.m(x,y)`	`p ! m(x,y)`	`Q(p).send("m",x,y)`
`p(x,y)`	`p ! (x,y)`	`Q(p).fcall(x,y)`
`p.m`	`p ! m`	`Q(p).get("m")`

Remote object references A local reference to an object is guaranteed to be unique and unforgeable, and only grants access to the public interface of the designated object. When a promise comes to designate a remote object, the promise effectively becomes a remote object reference. A remote reference only carries eventual message sends, not immediate method calls. Whereas local references are unforgeable, for remote references over open networks, we use unguessability to approximate unforgeability.

Primitive values such as strings and numbers are *pass-by-copy*—when passed as arguments or returned as results in remote messages, their contents are serialized and unserialized. JavaScript arrays default to pass-by-copy. All other objects and functions default to *pass-by-reference*—when passed as an argument or returned result, information needed to access them is serialized, which is unserialized into a remote reference for sending messages back to this object itself.

Over the RESTful transport [5], we serialize pass-by-reference objects using unguessable HTTPS URLs (also called web-keys [13]). Such a reference may look like `https://www.example.com/app/#mhbqcmmva5ja3`, where the fragment (everything after the `#`) is a random character string that uniquely identifies an object on the `example.com` server. We use unguessable secrets for remote object references because of a key similarity between secrets and object references: If you do not know an unguessable secret, you can only come to know it if somebody else who knows the secret chooses to share it with you.

`Q.passByCopy(record)` will override the pass-by-reference default, marking `record` as pass-by-copy. The record will then be shallow-copied to the destination, making a record with the same property names. The values of these properties get serialized according to these same argument passing rules.

2.5 NodeKen: Distributed Orthogonal Persistence

Rights, to be useful, must persist over time. Since object-references are our representation of rights, object references and the objects they designate must persist as well. We have already covered the distributed and secure aspects of Dr. SES. Here, we cover resilience against failures.

To introduce resilience, Dr. SES builds upon the Ken platform [6]. Ken applications are distributed communicating event loops, which aligns well with JavaScript's de-facto execution model. The event loop of a Ken process invokes application-level code to process incoming messages (one turn, i.e., one event loop iteration, per message). In addition, Ken provides:

Distributed consistent snapshots Ken provides a persistent heap for storing application data. All objects stored in this heap are persistent. Ken ensures that the snapshots of two or more communicating processes cannot grow inconsistent, by recording messages in flight as part of a process' snapshot.

Reliable messaging Under the assumption that all Ken processes eventually recover, all messages transmitted between Ken processes are delivered exactly once, in FIFO order. A permanently crashed Ken process is indistinguishable from a very slow process. To deal with such situations, applications may still want to do their own failure handling using time-outs.

A set of Ken processes can tolerate arbitrary failures in such a way that when a process is restarted after a crash, it is always restored to a previously consistent state. To the crashed process itself, it is as if the crash had never happened. To any of the process's communication partners, the process just seemed slow to respond. A crash will never cause messages to be dropped or delivered twice.

To achieve orthogonal persistence of JavaScript programs, the Ken platform must be integrated with the JavaScript runtime. NodeKen is our attempt at layering the Node.js runtime on top of Ken.[7] NodeKen can then be used as a stand-alone JavaScript environment to run persistent server-side Dr. SES programs. It is not our aim to embed Ken into the browser. This leads to two types of Dr. SES environments: Dr. SES in the browser runs in an ephemeral environment that ceases to exist when the user navigates to a different page, or closes the page. Objects and object references in such environments are not persistent.

[7] At the time of writing, NodeKen does not yet exist. We are actively working on integrating Ken with the v8 JavaScript virtual machine, upon which Node.js is based. See `https://github.com/supergillis/v8-ken`.

By contrast, Dr. SES on NodeKen runs in a persistent environment. JavaScript objects born in such an environment are persistent by default, as are object references spanning two persistent Dr. SES environments. Eventual message sends made using the "!" operator over persistent references are reliable.

Following the philosophy of Waterken [4], the persistent Java web server where the Ken ideas originated, we expect it to be common for ephemeral and persistent Dr. SES environments to communicate with each other, The ephemeral environment (inside the browser) primarily deals with UI and the persistent environment stores durable application state, a distributed form of the Model-View-Controller pattern. In the remainder of this paper, we assume that all Dr. SES code runs in persistent Dr. SES environments.

Implementation Ken achieves distributed consistent snapshots as follows:

- During a turn, accumulate all outgoing messages in an outgoing message queue. These messages are not yet released to the network.
- At the end of each turn, make an (incremental) checkpoint of the persistent heap and of all outgoing messages.
- After the end-of-turn checkpoint is made, release any new outgoing messages to the network and acknowledge the incoming message processed by this turn.
- Number outgoing messages with a sequence number (for duplicate detection and message ordering).
- Periodically retry sending unacknowledged outgoing messages (with exponential back-off) until an acknowledgement is received.
- Check incoming messages for duplicates. When a duplicate message is detected, it is dropped (not processed) and immediately acknowledged.

The key point is that outgoing messages are released, and incoming messages are acknowledged, only *after* the message has been fully processed by the receiver *and* the heap state has been checkpointed. The snapshot of a Ken process consists of both the heap and the outgoing message queue. It does not include the runtime stack (which is always empty between turns) nor the incoming message queue.

Checkpointing a program's entire state after every event loop turn may be considered costly. Ken takes care to only store those parts of the heap to disk that are updated during a turn. Further, the availability of cheap low-latency non-volatile memory (such as solid-state drives) has driven down the cost of writing state to "disk" to the point that making micro-snapshots after every turn becomes practical.

Ken and security The Ken protocol guarantees distributed snapshots even among mutually suspicious machines. An adversarial process cannot corrupt the distributed snapshots of benign processes.

The implementation of Ken underlying NodeKen currently does not use an encrypted communications channel to deliver messages between Ken processes. Hence, the authenticity, integrity or confidentiality of incoming messages cannot be guaranteed. In NodeKen, our plan is to actively secure the communications channels between NodeKen processes using a cryptographic library.[8]

[8] An outline of such a design, due to Brian Warner, is available online: `eros-os.org/pipermail/cap-talk/2012-September/015386.html`

Now that we've seen the elements of Dr. SES, we can proceed to explain how to use it to build erights and smart contracts.

3 Toward Distributed Electronic Rights

The elements of Dr. SES demonstrate how JavaScript can be transformed into a distributed, secure, and resilient system. At its core is the recognition that object references represent a right to perform a set of operations on a specific, designated resource. This emphasis on distributed rights has its counterpart in society: a system of rights is society's answer to creating distributed, secure, and resilient commercial systems.

Global commerce rests on tradeable rights. This system is: "*the product of thousands of years of evolution. It is highly complex and embraces a multitude of actions, objects, and individuals. ... With minor exceptions, rights to take almost all conceivable actions with virtually all physical objects are fixed on identifiable individuals or firms at every instant of time. The books are kept up to date despite the burden imposed by dynamic forces, such as births and deaths, dissolutions, and new technology.*" [14]

Rights help people coordinate plans and resolve conflicts over use of resources. Rights partition the space of actions to avoid interference between separately formulated plans, thus enabling cooperative relationships despite mutual suspicion and competing goals [15]. This rights-based perspective can shed light on the problem of securing distributed computational systems.

All computational systems must address the problem of open access. Global mutable state creates a tragedy of the commons: since anyone can access and change it, no one can safely rely on it. Use conflicts arise from both intentional (malicious) and unintentional (buggy) actions. Preventing use conflicts over shared state is one of the main challenges designers face in building computational systems.

Historically, two broad strategies for avoiding the tragedy of the commons have emerged: a governance strategy and a property rights strategy [16]. The governance approach solves the open access problem by restricting access to members and regulating each member's use of the shared resource. The property rights approach divides ownership of the resource among the individuals and creates abstract rules that govern the exchange of rights between owners. These approaches have their analogues in computational systems: ocap systems pursue a property rights strategy, while access control lists implement a governance strategy.

Access control lists solve the open access problem by denying unauthorized users access, and specifying access rights for authorized users. Great effort is put into perimeter security (firewalls, antivirus, intrusion detection, and the like) to keep unauthorized users out, while detailed access control lists regulate use by authorized users.

Governance regimes have proved successful in managing shared resources in many situations [17]. However, they tend to break down under increasing complexity. As the number of users and types of use increases, the ability of governance systems to limit external access and manage internal use breaks down. Perimeter security can no longer cope with the pressure for increased access, and access control lists cannot keep up with dynamic requests for changes in access rights.

The property rights strategy deals with increasing complexity by implementing a decentralized system of individual rights. Rights are used to partition the commons into separate domains under the control of specific agents who can decide its use, as long as the use is consistent with the rights of others. Instead of excluding non-members at the perimeter, the property strategy brings all agents under a common set of abstract rules that determine how rights are initially acquired, transferred, and protected [18]. Individual rights define the boundaries within which agents can act free of interference from others. Contracts enable the exchange of rights across these protected domains.

The ocap approach can be seen as analogous to an individual rights approach to coordinating action in society. The local unforgeable object reference and the remote unguessable reference represent one kind of eright—the right to invoke the public interface of the object it designates. In ocap systems, references bundle authority with designation [19]. Like property rights, they are possessory rights: possession of the reference is all that is required for its use, its use is at the discretion of the possessing entity, and the entity holding the reference is free to transfer it to others [20].

The private law system of property, contract, and tort brings resources into a system of rights. Property law determines the initial acquisition of rights; contract law governs the transfer of rights; and tort law protects rights from interference [21]. Ocap systems follow a similar logic: the rules of object creation make it easy to create objects with only the rights they need, the message passing rules govern the transfer of rights, and encapsulation protects rights from interference [7].

While object references represent a kind of eright, they differ in several respects from more familiar rights in society. For example, object references are typically shared. When Alice gives Bob a reference to an object, she is transferring a copy of the reference thereby sharing access to the object. In society, transfers of rights usually take the form of a transfer of exclusive access due to the rivalrous nature of physical objects. I give up my access to my car when I transfer title to you. Exclusive rights is the default in the physical world; complex legal frameworks are needed to enable sharing (partnerships, corporations, easements, and so forth). Computational systems face the opposite tradeoff: sharing is easy, but exclusivity is hard.

In the next sections, we will show how, by building on object references as erights, we can create new kinds of erights at a new level of abstraction. We look first at how money can be implemented as a smart contract. Money differs from other forms of property in several ways [22]. Here, we identify four dimensions in which money differs from object references as rights. Object references are shareable, specific, opaque, and exercisable, whereas money is exclusive, fungible, measurable, and symbolic.

By contrast with object references that are shareable, money needs to be exclusive to serve as medium of exchange. Bob does not consider himself paid by Alice until he knows that he has exclusive access to the funds. Object references are also specific; they designate a particular object. Money, on the other hand, is fungible. You care about having a certain quantity of a particular currency, not having a specific piece of currency. One dollar is as good as another.

Objects are opaque. The clients of an object can invoke it but don't necessarily know how it will react—that information is private to the object. By contrast, money is measurable. Bob must be able to determine that he really has a certain quantity of a particular currency. Finally, money, unlike object references, is never exercisable. The right you have when you have an object reference is the right to do something: the right to invoke the behavior of the object it designates. Money, however, has no direct use value; its value is symbolic. It has value only in exchange.

Contracts manipulate rights. The participants in a contract each bring to it those rights the contract will manipulate [23]. The logic of the contract together with the decisions of the participants determines which derived rights they each walk away with. The simplest example is a direct trade. Since half the rights exchanged in most trades are money, we start with money.

4 Money as an Electronic Right

Figure 1 is our implementation of a money-like rights issuer, using only elements of Dr. SES explained above. To explain how it works, it is best to start with how it is used. Say Alice wishes to buy something from Bob for $10. The three parties involved would be Alice, Bob, and a $ issuer, which we will informally call a bank. The starting assumptions are that Alice and Bob do not trust each other, the bank does not trust either Alice or Bob, and Alice and Bob trust the bank with their money but with nothing else. In this scenario, Alice is willing to risk her $10 on the possibility of Bob's non-delivery. But Bob wants to be sure he's been paid before he releases the good in exchange.

What do these relationships mean in terms of a configuration of persistent objects? Say Alice owns (or is) a set of objects on machine A, Bob on machine B, and the bank on machine C. In order for Alice to make a `buy` request of Bob, we assume one of Alice's objects already has a remote reference to one of Bob's objects. Alice's trust of the bank with her money is represented by a remote reference to an object within the bank representing Alice's account at the bank. We refer to such objects as *purses*. The one for Alice's account is Alice's *main purse*. And likewise for Bob. Where do these initial account purses come from?

For each currency the bank wishes to manage, the bank calls `makeMint()` once to get a `mint` function for making purses holding units of that currency. When Alice opens an account with, say $100 in cash, the bank calls `mint(100)` on its $ mint, to make Alice's main purse. The bank then gives Alice a persistent remote reference to this purse object within the bank.

For Alice to pay Bob, she sets up a *payment purse*, deposits $10 into it from her main purse, and sends it to Bob in a buy request, together with a description of what she wishes to buy.

```
var paymentP = myPurse ! makePurse();
var ackP = paymentP ! deposit(10, myPurse);
var goodP = ackP.then(_ => bobP ! buy(desc, paymentP));
```

On the diagram in Figure 1, each `makeMint` call creates a layer with its own (`mint`, `m`) pair representing a distinct currency. Each `mint` call creates a nested layer with its own (`purse`, `decr`, `balance`) triple. On line 16 of the code, each `purse` to `decr` mapping is also entered into the `m` table shared by all purses of the same currency. Alice's main purse is on the bottom purse layer. Bob's is on the top layer. Alice's payment purse, being sent to Bob in the buy message, is in the middle layer.

```
 1 var makeMint = () => {
 2  var m = WeakMap();
 3  var makePurse = () => mint(0);

 4  var mint = balance => {
 5   var purse = def({
 6    getBalance: () => balance,
 7    makePurse: makePurse,
 8    deposit: (amount, srcP) =>
 9     Q(srcP).then(src => {
10      Nat(balance + amount);
11      m.get(src)(Nat(amount));
12      balance += amount;
13     })
14   });
15   var decr = amount => { balance = Nat(balance - amount); };
16   m.set(purse, decr);
17   return purse;
18  };
19  return mint;
20 };
```

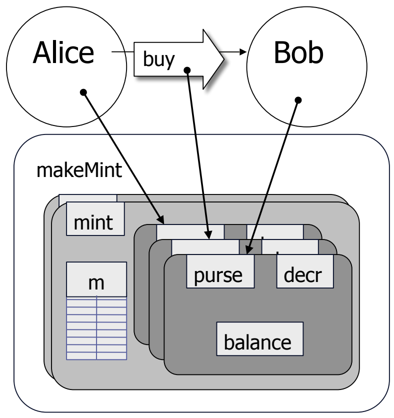


Fig. 1. The Mint Maker

Bob receives this request at the following `buy` method:

```
buy: (desc, paymentP) => {
 // do whatever with desc, look up $10 price
 return (myPurse ! deposit(10, paymentP)).then(_ => good);
}
```

Bob's `buy` method handles a message from untrusted clients such as Alice, and thus it does not know what object Alice actually provided as the payment argument. At this point, the purse provided by Alice is *specific*—it is the specific object Alice designated, but to Bob it also is *opaque*. In particular, Bob has no idea if his `paymentP` parameter actually designates a purse, whether it is a purse at this bank, of this currency, and with adequate funds. Even if he knew all these conditions were true at the moment, due to the *shareable* nature of argument passing, Bob wouldn't know the funds would still be there by the time he deposits it. Alice may have retained a reference to it. He delegates all these problems to the bank with the `deposit` request above.

If the bank's `deposit` method acknowledges a successful deposit, by fulfilling the promise for the result of the deposit, then Bob knows he has obtained *exclusive* access to a *fungible* and *measurable* quantity of a given currency at a given bank. In this case, the success callback of the `.then` above gets called, returning the `good`, fulfilling Alice's pending `goodP` promise.

The interesting work starts on line 11, where `deposit` looks up the alleged payment purse in the `m` table. If this is anything other than a purse of the same currency at the same bank, this lookup will instead return `undefined`, causing the following function call to throw an error, rejecting Bob's promise for the result of the deposit, rejecting Alice's `goodP`. If this lookup succeeds, it finds the `decr` function for decrementing that purse's `balance`, which it calls with the amount to withdraw. If the payment has insufficient funds, `balance - amount` would be negative and `Nat` would throw.

We have now arrived at the commit point. All the tests that might cause failure have already passed, and no side effects have yet happened. Now we perform all side effects, all of which will happen since no locally observable failure possibilities remain. The assignment decrements the payment purse's `balance` by `amount`, and `decr` returns. Line 12 increments the balance of the purse being deposited into.

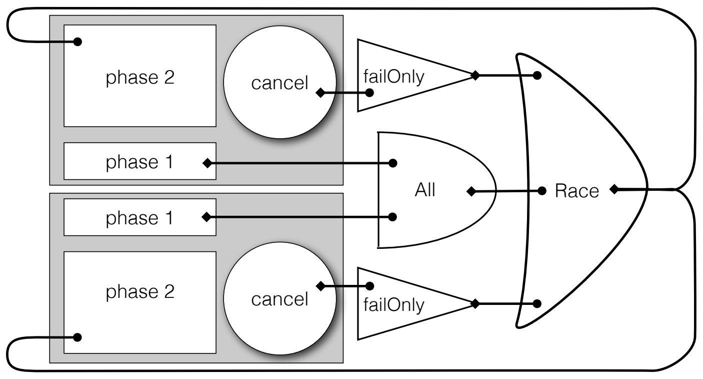
The success callback in the `deposit` method implicitly returns `undefined`, fulfilling Bob's promise for the result of the `deposit` request, triggering Bob to release the good to Alice in exchange.

5 The Escrow Exchange Contract

In the mint maker scenario, Alice must risk her $10 on the possibility of Bob's non-delivery. We now introduce an escrow exchange contract that implements an *all or nothing* trade. We explain the escrow exchange contract in terms of a scenario among five players: Alice, Bob, a money issuer (running the code of Figure 1), a stock issuer (also running the code of Figure 1 but with the units representing shares of some particular stock), and an escrow exchange agent (running the code of Figure 2). The diagram at the top of Figure 3 shows the initial relationships, with the escrow exchange agent in the role of contract host.

Alice and Bob again do not trust each other. They wish to trade $10 of Alice's money for 7 shares of Bob's stock, but in this case, neither is willing to risk their assets on the possibility of the other's non-delivery. They both trust the same money issuer with their money, the same stock issuer with their stock, and the same escrow exchange agent with the rights to be traded. The money issuer, the stock issuer, and the escrow exchange agent each have no prior knowledge or trust in the others. Additionally, none of these trust Alice or Bob. The rest of the scenario as presented below examines only the consequences of Alice or Bob's misbehavior and assumes the other three run the code shown honestly. A full analysis of vulnerabilities should consider all combinations.

Since the situation is now symmetric, we explain the progression of events from Alice's perspective. Alice's prior trust in each issuer is represented as before—Alice holds a persistent reference to her main purse at each issuer. Alice's prior trust in the escrow exchange agent is represented as the ability to provide the first "`a`" argument in the call to `escrowExchange` (Figure 2, line 12) for which Bob is able to provide the second "`b`" argument.



```
var transfer = (decisionP, srcPurseP, dstPurseP, amount) => {
  var makeEscrowPurseP = Q.join(srcPurseP ! makePurse,
                                dstPurseP ! makePurse);
  var escrowPurseP = makeEscrowPurseP ! ();

  Q(decisionP).then(                                // setup phase 2
    _ => { dstPurseP ! deposit(amount, escrowPurseP); },
    _ => { srcPurseP ! deposit(amount, escrowPurseP); });

  return escrowPurseP ! deposit(amount, srcPurseP); // phase 1
};

var failOnly = cancellationP => Q(cancellationP).then(
  cancellation => { throw cancellation; });

var escrowExchange = (a, b) => {       // a from Alice, b from Bob
  var decide;
  var decisionP = Q.promise(resolve => { decide = resolve; });

  decide(Q.race([Q.all([
      transfer(decisionP, a.moneySrcP, b.moneyDstP, b.moneyNeeded),
      transfer(decisionP, b.stockSrcP, a.stockDstP, a.stockNeeded)
    ]),
    failOnly(a.cancellationP),
    failOnly(b.cancellationP)]));
  return decisionP;
};
```

Fig. 2. The Escrow Exchange Contract

Alice might create this argument as follows:

```
var cancel;
var a = Q.passByCopy({
  moneySrcP: myMoneyPurse ! makePurse(),
  stockDstP: myStockPurse ! makePurse(),
  stockNeeded: 7,
  cancellationP: Q.promise(r => { cancel = r; })
});
a.moneySrcP ! deposit(10, myMoneyPurse);
```

By a protocol whose details appear below, Alice sends this "a" object to the escrow exchange agent, for it to use as the first argument in a call to `escrowExchange`, which initiates this specific contract between Alice and Bob. The `escrowExchange` function returns a promise for the outcome of the contract, which the escrow exchange agent returns to Alice.

If this outcome promise becomes fulfilled, the exchange succeeded, she should expect her `a.moneySrcP` to be drained, and 7 shares of stock to be deposited into her `a.stockDstP` promptly.[9] If this promise becomes rejected, the exchange failed, and she should expect her $10 to reappear in her `a.moneySrcP` promptly. In the meantime, if she gets impatient and would rather not continue waiting, she can call her `cancel` function with her alleged reason for walking away. Once she does so, the exchange will then either succeed or fail promptly.

On lines 13 and 14 of Figure 2 the `escrowExchange` contract makes a `decisionP` promise whose fulfillment or rejection represents its decision about whether the exchange must succeed or fail. It makes this decision by calling `decide` with the outcome of a race between a `Q.all` and two calls to `failOnly`. Until a player cancels the exchange, the `Q.race` can only be won by the `Q.all`, where the exchange is proceeding.

The arguments to `Q.all` are the results of two calls to `transfer`. The first call to `transfer` sets up an arrangement of objects whose purpose is to transfer money from Alice to Bob. The second call's purpose is to transfer stock from Bob to Alice. Each call to transfer returns a promise whose fulfillment or rejection indicates whether it has become confident that this one-way transfer of erights would succeed. If both transfers become confident (before any cancellations win the race), then the overall decision is to proceed. If either transfer indicates failure, by rejecting the promise it has returned, then, via `Q.all`, `decisionP` becomes rejected.[10]

We do not feed the cancellation promises directly into the race, as Alice could then fulfill the cancellation promise, causing the race to signal a decision to proceed with the exchange, even though Alice's money has not been escrowed, potentially giving Bob's stock to Alice for free. Instead, once the cancellation promise has been either fulfilled

[9] By "promptly" we mean, once the relevant machines are up, processes running, and reachable to each other over the network.

[10] This pattern implements two phase commit enhanced with the possibility of cancellation, where the call to `escrowExchange` creates a transaction coordinator, and each of its calls to `transfer` creates a participant.

or rejected, the promise returned by `failOnly` will only become rejected. Only the `Q.all` can win the race with a success.

Since the two calls to `transfer` are symmetric, we examine only the first. The first phase of the transfer, on line 8 of Figure 2, attempts to deposit Alice's money into an escrow purse mentioned only within this transfer. If this deposit succeeds, Alice's money has been escrowed, so the money portion of the exchange is now assured. If this deposit fails, then the exchange as a whole should be cancelled. So `transfer` simply returns the promise for the outcome of this first deposit.

The `transfer` function sets up the second phase on lines 5, 6, and 7. If the overall decision is that the exchange should succeed, the success callback deposits Alice's escrowed money into Bob's account. Otherwise it refunds Alice's money.

Only one mystery remains. How does the escrow agent obtain a fresh escrow purse at this money issuer, in order to be confident that it has obtained exclusive access to the money at stake? Since the escrow exchange agent has no prior knowledge or trust in the money issuer, it cannot become confident that the issuer is honest or even that the money it issues means anything. The question is meaningless. Instead, it only needs to obtain a fresh escrow purse whose veracity is *mutually acceptable* to Alice and Bob.

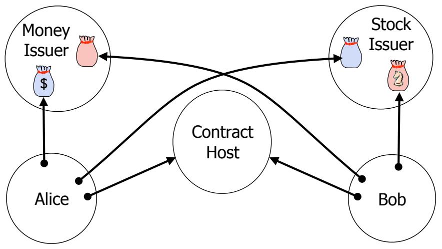
If the escrow contract simply asks Alice's purse for a new empty purse (`srcPurseP ! makePurse()`), Alice could return a dishonest purse that acknowledges deposit without transferring anything. Alice would then obtain Bob's stock for free. If it simply asks Bob's purse, then Bob could steal Alice's money during phase 1. Instead, it checks if their `makePurse` methods have the same object identity by using `Q.join` on promises for these two methods. This is why, on lines 3 and 7 of Figure 1, all purses of the same currency at the same bank share the same function as their `makePurse` method. If the `Q.join` of these two methods fails, then either Alice was dishonest, Bob was dishonest, or they simply didn't have prior agreement on the same currency at the same money issuer.

6 The Contract Host

Once Alice and Bob agree on a contract, how do they arrange for it to be run in a mutually trusted manner?

To engage in the escrow exchange contract, Alice and Bob had to agree on the issuers, which is unsurprising since they need to agree on the nature of rights exchanged by the contract. And they had to agree on an escrow exchange agent to honestly run this specific escrow exchange contract. For a contract as reusable as this, perhaps that is not a problem. But if Alice and Bob negotiate a custom contract specialized to their needs, they should not expect to find a mutually trusted third party specializing in running this particular contract. Rather, it should be sufficient for them to agree on:

- The issuers of each of the rights at stake.
- The source code of the contract.
- Who is to play which side of the contract.
- A third party they mutually trust to run their agreed code, *whatever it is*, honestly.



```
var makeContractHost = () => {
 var m = WeakMap();

 return def({
  setup: contractSrc => {
   contractSrc = ''+contractSrc;
   var tokens = [];
   var argPs = [];
   var resolve;
   var resultP = Q.promise(r => { resolve = r; });
   var contract = confine(contractSrc, {Q: Q});

   var addParam = (i, token) => {
    tokens[i] = token;
    var resolveArg;
    argPs[i] = Q.promise(r => { resolveArg = r; });
    m.set(token, (allegedSrc, allegedI, arg) => {
     if (contractSrc !== allegedSrc) {
      throw new Error('unexpected contract: '+contractSrc);
     }
     if (i !== allegedI) {
      throw new Error('unexpected side: '+i);
     }
     m.delete(token);
     resolveArg(arg);
     return resultP;
    });
   };
   for (var i = 0; i < contract.length; i++) {
    addParam(i, def({}));
   }
   resolve(Q.all(argPs).then(
    args => contract.apply(undefined, args)));
   return tokens;
  },
  play: (tokenP, allegedSrc, allegedI, arg) => Q(tokenP).then(
   token => m.get(token)(allegedSrc, allegedI, arg))
 });
};
```

Fig. 3. The Contract Host

Figure 3 shows the code for a generic contract host. It is able to host any contract formulated, as our escrow exchange contract is, as a function, taking one argument from each player and returning the outcome of the contract as a whole. Setting up a contract involves a necessary asymmetry among the players. One of the players, say Bob, must initiate a new live contract instance by sending the contract's code to the contract host. At this point, only Bob knows both this contract instance and that he'd like to invite Alice to participate in *this* instance. If Bob simply sent to Alice references to those objects on the contract host that enable Alice to play, Alice would not know what she's received, since she received it from Bob whom she does not trust. She does trust the contract host, and these objects are on the contract host, but so are the objects corresponding to other contracts this host is initiating or running. Only Bob can connect Alice to this contract instance, but Alice's confidence that she's playing the contract she thinks she is must be rooted in her prior trust in the contract host.

Our contract host is an object with two methods, `setup` and `play`. Bob sets up the contract instance by calling `setup` with the source code for the contract function in question, e.g., `escrowExchange`. At line 32, `setup` returns an array of unique unforgeable tokens, one for each contract parameter. Bob's invitation to Alice includes this token, the source for the contract he wishes Alice to play, the argument index indicating what side of the contract Alice is to play, and the contract host in question.

If Alice decides she'd like to play this contract, she formulates her argument object as above, and sends it in a `play` request to the contract host along with the token, the alleged contract source code, and the alleged side she is to play. If all of this checks out and this token has not previously been redeemed, then this token gets used up, Alice's argument is held until the arguments for the other players arrive, and Alice receives a promise for the outcome of the contract. Once all arguments arrive, the contract function is called and its result is used to resolve the previously returned promise.

By redeeming the token, Alice obtains the exclusive right to play a specific contract whose logic she knows, and whose play she expects to cause external effects. This eright is exclusive, specific, measurable, and exercisable.

Conclusions

In human society, rights are a scalable means for organizing the complex cooperative interactions of decentralized agents with diverse interests. This perspective is helping us shape JavaScript into a distributed resilient secure programming language. We show how this platform would enable the expression of new kinds of rights and smart contracts *simply*, supporting new forms of cooperation among computational agents.

Acknowledgements Many people contributed to the progress reported here, including the e-lang community for refining ocap-based smart contracting, the Google Caja group for SES's growth and deployment, TC39 for making ES5 and successors friendly to ocaps, Tyler Close for the first Ken and Q, Terence Kelly for the new Ken, and Kris Kowal for the new Q.

Tom Van Cutsem is a post-doctoral fellow of the Research Foundation, Flanders. Thanks to Terry Stanley, Kevin Reid, and Terence Kelly for suggestions improving this paper.

References

1. Szabo, N.: Formalizing and securing relationships on public networks. First Monday **2**(9) (1997)
2. Tribble, E.D., Miller, M.S., Hardy, N., Krieger, D.: Joule: Distributed Application Foundations. Technical Report ADd03.4P, Agorics Inc., Los Altos (December 1995) `erights.org/history/joule/`.
3. Miller, M.S., Morningstar, C., Frantz, B.: Capability-based Financial Instruments. In: Proc. Financial Cryptography 2000, Anguila, BWI, Springer-Verlag (2000) 349–378 `www.erights.org/elib/capability/ode/index.html`.
4. Close, T.: Waterken Server: capability-based security for the Web (2004) `waterken.sourceforge.net`.
5. Close, T.: web_send `waterken.sourceforge.net/web_send/`.
6. Yoo, S., Killian, C., Kelly, T., Cho, H.K., Plite, S.: Composable reliability for asynchronous systems. In: Proceedings of the 2012 USENIX conference on Annual Technical Conference. USENIX ATC'12, Berkeley, CA, USA, USENIX Association (2012) 3–3
7. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA (May 2006)
8. Mettler, A.: Language and Framework Support for Reviewably-Secure Software Systems. PhD thesis, EECS Department, University of California, Berkeley (Dec 2012)
9. Taly, A., Erlingsson, U., Mitchell, J.C., Miller, M.S., Nagra, J.: Automated analysis of security-critical javascript apis. In: Security and Privacy (SP), 2011 IEEE Symposium on, IEEE (2011) 363–378
10. Jones, A.K.: Protection in Programmed Systems. PhD thesis, Department of Computer Science, Carnegie-Mellon University (June 1973)
11. Miller, M.S., Tribble, E.D., Shapiro, J.S.: Concurrency Among Strangers: Programming in E as Plan Coordination. In: Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers. (2005) 195–229
12. Liskov, B., Shrira, L.: Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In: PLDI '88: Proc. ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, New York, NY, USA, ACM Press (1988) 260–267
13. Close, T.: Web-key: Mashing with permission. In: W2SP'08. (2008)
14. Jensen, M.C., Meckling, W.H.: Specific and general knowledge and organizational structure. Journal of Applied Corporate Finance (8:2) (1995) 4–18
15. Steiner, H.: An Essay on Rights. Wiley-Blackwell (1994)
16. Smith, H.E.: Exclusion versus governance: two strategies for delineating property rights. Journal of Legal Studies **31** (2002)
17. Ostrom, E.: Governing the Commons: The Evolution of Institutions for Collective Action. Cambridge University Press (1990)
18. Hayek, F.A.: Law, Legislation and Liberty, Volume 1: Rules and Order. University Of Chicago Press (1973)
19. Hardy, N.: The Confused Deputy. Operating Systems Review (October 1988)
20. Mossoff, A.: What is property-putting the pieces back together. Arizona Law Review **45** (2003) 371
21. Epstein, R.A.: Simple Rules for a Complex World. Harvard University Press (1995)
22. Fox, D.: Property rights in money. Oxford University Press (2008)
23. Barnett, R.E.: A consent theory of contract. Columbia Law Review **86** (1986) 269