Capabilities, Trust, and Risk

- random rant -

Sophia Drossopoulou & James Noble
WG2.16, 13 May 2013

A very powerful program

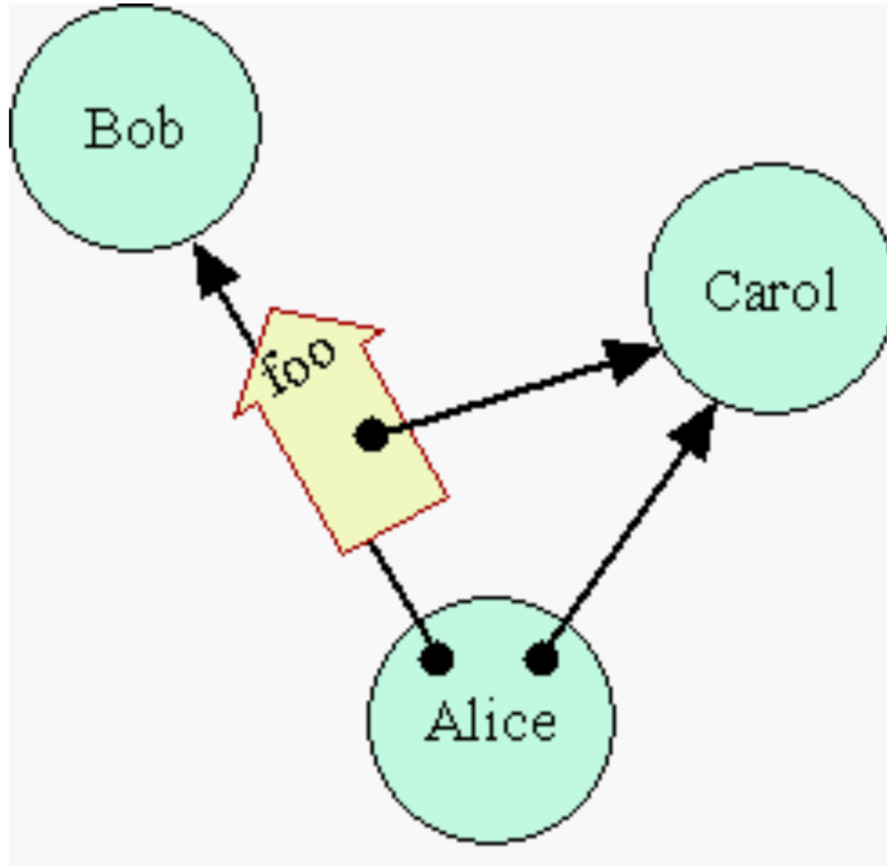
- Stolen shamelessly from David Wagner, http://www.cs.berkeley.edu/~daw/talks/PLAS06.ps

CAP COMPUTER

Object Capabilities

- *Unforgeable* capabilities
  - Possession implies Right
  - No other access control checking
    - Who do you trust?   Who do you *really* trust?
    - Who are you holding on to? Who are you dreaming of?
- Principle of Least Authority
  - No Ambient Authority
- Capabilities + Pure Object-Orientation = Object-Capabilities

Object Capabilities



Aim

Use object capabilities
(and nothing but object capabilities)
to support cooperation / commerce
between mutually untrusting parties

Meta-Aim

Understand how you could do this
Understand code that claims to do this

Distributed Electronic Rights in JavaScript

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

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

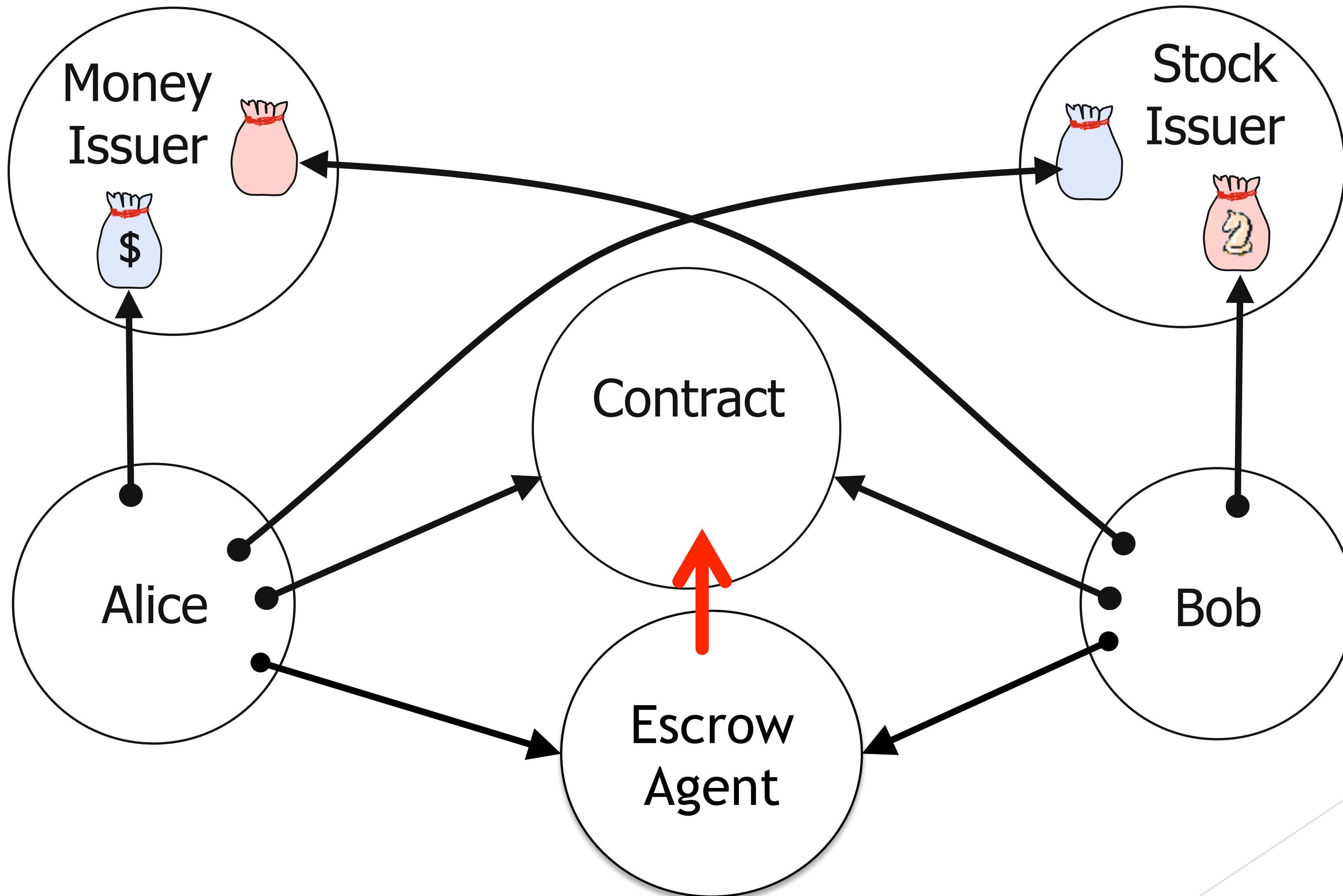
bstract. Contracts enable mutually suspicious parties to cooperate safely rough the exchange of rights. Smart contracts are programs whose behavior nforces the terms of the contract. This paper shows how such contracts can be pecified elegantly and executed safely, given an appropriate distributed, secure,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

ESOP'13

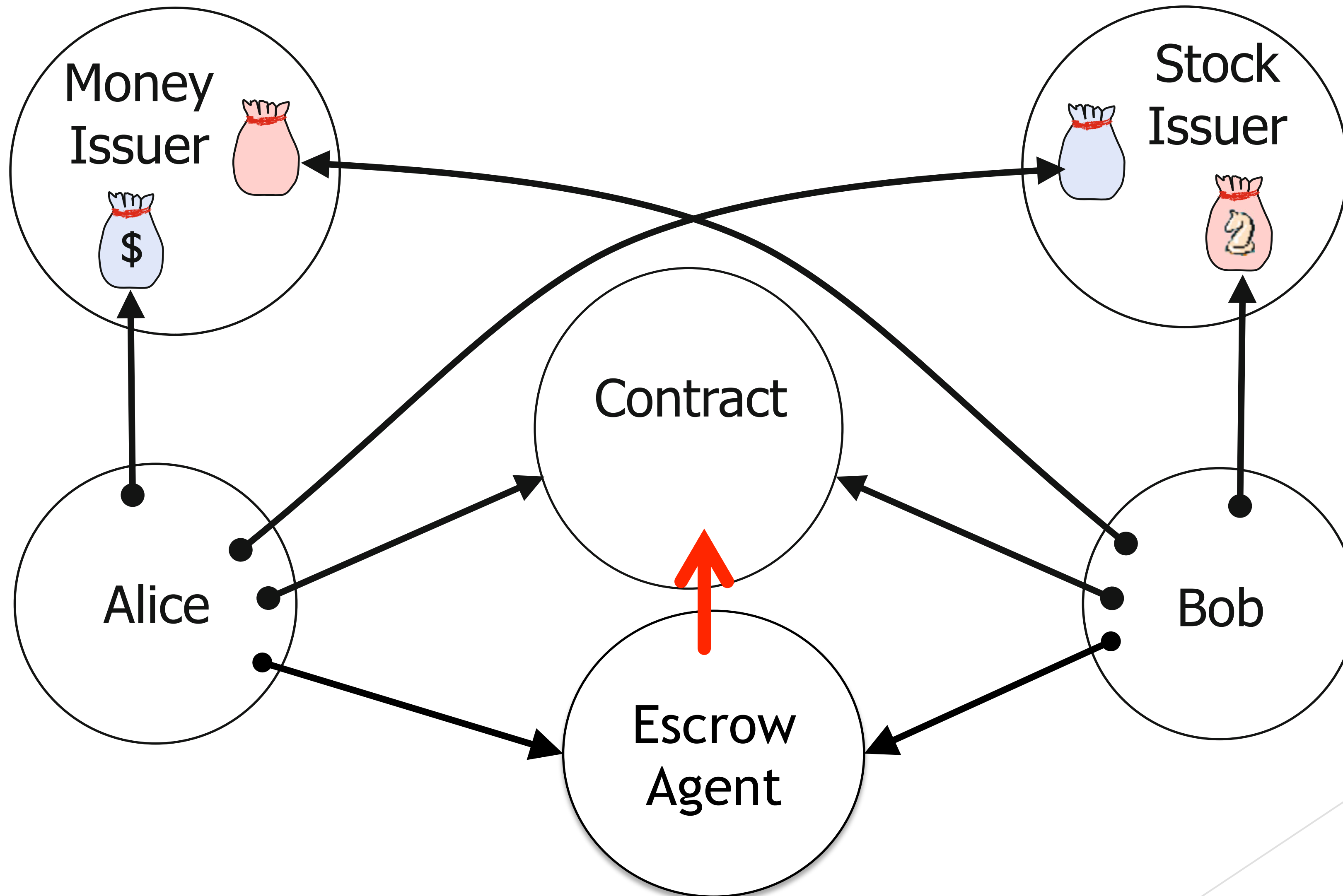
- Smart Contracts
  - "understandable by non-experts"
- Real JavaScript
- Distributed, Concurrent,
- Generic, Symmetrical

```
1  var transfer = (decisionP, srcPurseP
2    var makeEscrowPurseP = Q.join(srcPu
3                                  dst
4    var escrowPurseP = makeEscrowPurseP
5    Q(decisionP).then(
6      _ => { dstPurseP ! deposit(amount
7      _ => { srcPurseP ! deposit(amount
8    return escrowPurseP ! deposit(amour
9  };
10 var failOnly = cancellationP => Q(ca
11   cancellation => { throw cancellatic
12 var escrowExchange = (a, b) => {
13   var decide;
14   var decisionP = Q.promise(resolve =
```

Exchange Contract

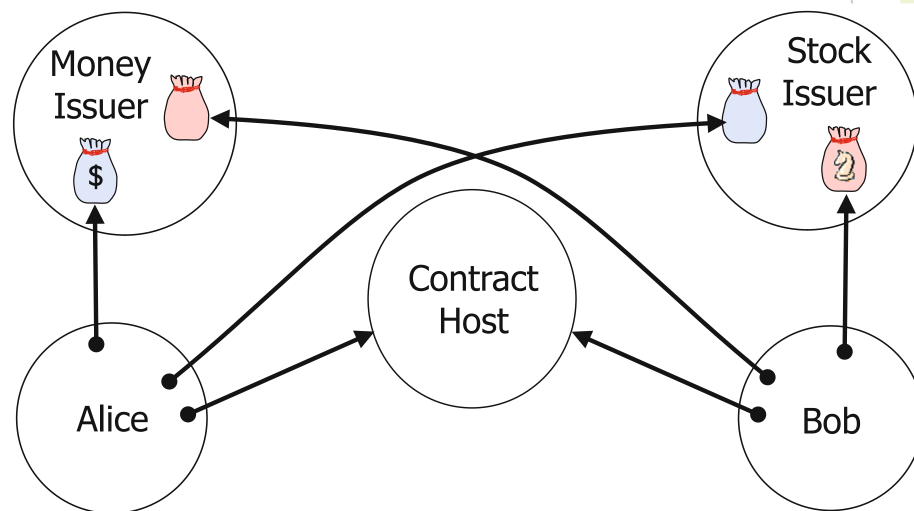


Exchange Contract



Escrow Agent

gives out contracts




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

3    return def({
4      setup: contractSrc => {
5        contractSrc = ''+contractSrc;
6        var tokens = [];
7        var argPs = [];
8        var resolve;
9        var resultP = Q.promise(r => { resolve = r; });
10       var contract = confine(contractSrc, {Q: Q});

11       var addParam = (i, token) => {
12         tokens[i] = token;
13         var resolveArg;
14         argPs[i] = Q.promise(r => { resolveArg = r; });
```

Escrow Agent

```
ef escrowAgent = object { // well known singleton

class contract.new(name' : String) { ... } // see fig 3

var terms : String
var currentContract : Contract
var waitingForSeller := true

// called by seller to request a seller−side contract
method getSellerContract(terms': String) −> Contract {
      if (! waitingForSeller)
          then { Error.raise "already has seller" }
      terms := terms'
      waitingForSeller := false // now waiting for a buyer
      currentContract := contract.new(terms)
      return currentContract
}
```

Escrow Agent

```
called by buyer to request a buyer-side contract
ethod getBuyerContract(terms' : String) -> Contract {
   if (waitingForSeller) then {
      Error.raise "waiting for a seller" }
   if (terms != terms') then {
      Error.raise "terms don't match" }
   def thisContract = currentContract
   terms := "invalid terms"
   currentContract := contract.new(terms)
   waitingForSeller := true
   return thisContract
```

```
// Alice the seller moves first
def alice = object {
   def alicesContract =
         escrowAgent.getSellerContract("some terms")
   ...

// Bob the buyer moves second
def bob = object {
   def bobsContract =
         escrowAgent.getBuyerContract("some terms")
```

Contract

```
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; });
```

Contract

```
ss contract.new(name' : String) {

var offered := false
var sellersGoods : m.Purse //P
var amount : Number
var price : Number
var sellersMoney : m.Purse

method offer(sellersGoods' : m.Purse,
             amount' : Number,
             price' : Number,
             sellersMoney' : m.Purse) {
  sellersGoods := sellersGoods'
  amount := amount'
  price := price'
  sellersMoney := sellersMoney'
  offered := true
}
```

```
// Alice the seller
def mDst = mint.newPurse("Alice's mDst", 0)
def gSrc = goods.newPurse("Alice's gSrc", 7)
alicesContract.offer(gSrc, 7, 10, mDst)

// Bob the buyer
def mSrc = mint.newPurse("Bob's mSrc", 10)
def gDst = goods.newPurse("Bob's's gDst", 0)
bobsContract.bid(gDst, 7, 10, mSrc)
```

ontract

```
method bid(buyersGoods : m.Purse,
           amount' : Number,
           price' : Number,
           buyersMoney : m.Purse) -> Done {
  if (!offered) then { Error.raise "Not offered" }
  if ( (amount != amount') || (price != price') ) then
     { Error.raise "Bid/Offer mismatch" }
  if ( (amount < 0) || (price < 0) ) then
     { Error.raise "Bid/Offer fraud" }

  // check purses are from the same mints
  buyersGoods.deposit( 0, sellersGoods )
  buyersMoney.deposit( 0, sellersMoney )

  // here we go
  def moneyEscrow : m.Purse = buyersMoney.makePurse
  moneyEscrow.deposit( price, buyersMoney )
     // exceptions are not caught here, so end the bid

  def goodsEscrow : m.Purse = sellersGoods.makePurse
  try { goodsEscrow.deposit( amount, sellersGoods ) }
   catch { _ -> buyersMoney.deposit( price, moneyEscrow );
                Error.raise "TXN FAILURE" }

  sellersMoney.deposit( price, moneyEscrow )
  buyersGoods.deposit(amount, goodsEscrow )
 }
}
```

Trust

what does trust mean?
who trusts whom?

- Buyer and Sellers
  - Trust their Mints & Purses, and the Escrow Agent
  - Don’t trust each other — cannot have mutual references
- Mints & Purses trust nobody
- Escrow Agent (and Contract)
  - Don’t trust Buyers or Sellers or Mints or Purses…
  - So what guarantees can they supply?

Risk

what's the worst that can happen if I trust?

Buying 10 apples for £10, £1000 in your purse. New purse: £1.

- Hand out main purse: best case 10🍏 £10; worst: 0🍎 £1000
- Temporary purse: 10🍏 £11; 0🍎 £11
- Escrow/w main purse: 10🍏 £15; 0🍎 £1000
- Escrow/ temp purse: 10🍏 £17; 0🍎 £2
- Say we really trust escrow: 10🍏 £15; 0🍎 £0