CM 9 de Compilation The mini-JS machine exceptions and objects

Flavien BREUVART

March 11, 2025

© CC-BY-NC-SA

Exceptions : a control structure

```
try {
    print("x/5 :"+f(x))
} catch (e) {
    print("x%5 :"+(e+5))
}
function f (x) {
    if (x < 0) throw (x+5);
    if (x == 0) return x;
    return 1+f(x-5);
}
```

Remarks

- The `catch (e)` recover from a `throw` inside the `try`.
- The exception is thrown through multiple function calls.
- `throw` interups all calculations.
- The stack is overfilled whenever the exception is thrown.

If `x` is divisible by 5 then we get `x/5`, else we get `res=x%5`.

3 implementations of exceptions

Exception Table

- also called 0-cost (optimal if no exception thrown),
- the table : l. of code → whet to do on exception.
- Compiler dificult to code.
- Bloc some powerful optimisations.

Used by Java, but less and less popular.

Exception Mode

- exception flag,
- tow instruction sets (depending on the flag),
- slow if not optimised,
- difficult to optimize.

Mostly legacy, or can be implemented at sofwear level (library/framework)

Error continuation

- `throw` ∼ `return`,
- if exception, pop until the next continuation,
- difficult to optimise on big stacks, would need an allocated register and static informations.

Trendy (LLVM...)

Error continuations

A continuation contains

- a pointer toward the line that called the function,
- a context
- a "type" : `return` or `erreur`,

(some langages have access to more kind of continuations.)

Instruction `Throw`

Save the head of the stack,
Pop everything until next error continuation,
Restore the state (PC and CC) of the error continuation,
Restore the head of the stack.

Intermezzo : instruction Return

You need a “clean stack”

The instructions may use “Peak” or “Pop”, carefuly used the correct one. The instruction Drop remove residuals, you can also use alternative instructions

Example :

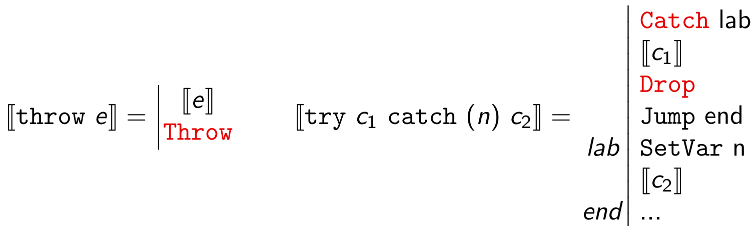
```
42; y = 2 + (x=3);
```

```
#42;          #y=2+(x=3)     SetVaD x
CsteNb 42     CsteNb 2       AddiNb
Drop          CsteNb 3       SetVar y
```

Our (non-optimal) Return instruction automatically cleans the stack

To avoid bugs in the begining of the project, Return behave like Throw : it digs for the next return continuation.

Naif encodaging



Catch off	Push(NewContinuation{cont = CC, code = PC + off + 1, err = true})	pile	#t:pile

Warning: now you have to clean up your stack

The instruction Drop will misbehave if the stack is not cleaned in the "try" block.

Example

```
try {
    res = 2 + f(x)
} catch (e) {
    res = e+5
}
function f (x) {
    if (x < 0) throw x;
    if (x == 0) return x;
    return 1+f(x-5);
}
```

```
NewClo 16
DecArg x
SetVar f
Catch 7
  CsteNb 2
  GetVar f
  StCall
  AddiNb
  SetVar res
Drop
Jump 5
SetVar e
  GetVar e
  CsteNb 5
  AddiNb
  SetVar res
Halt
```

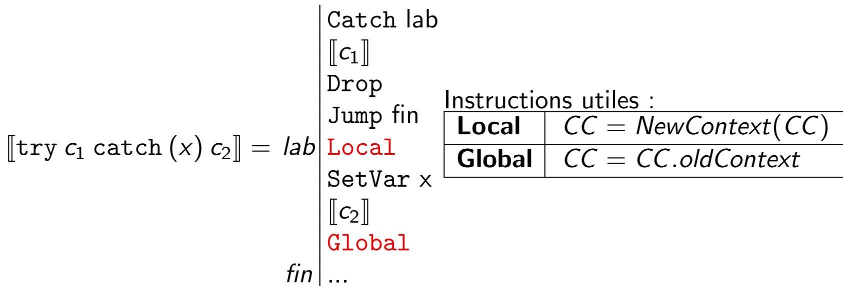
```
#function f
GetVar x
CsteNb 0
LwStNb
ConJmp 2
  GetVar x
  Throw
GetVar x
CsteNb 0
Equals
ConJmp 2
  GetVar x
  Return
CsteNb 1
GetVar f
StCall
```

```
GetVar x
  CsteNb 5
  SubsNb
  SetArg
  Call
  Return
```

Patching our implementation: The variable catching the error should be local

try c_1 catch (x) c_2
x have to be local in c_2

Il serait mieux de rendre x visible uniquement dans c_2 et non dans le contexte global.



Instructions utiles :

Local	$CC = NewContext(CC)$
Global	$CC = CC.oldContext$

var vs let keywords

var x

Hoisted to the header of the current function.

let x

Hoisted at the start of the block (the {...})

Issue with `let x`: x available only inside a block

On utilise encors les instruction `Local` et `Global`

$$[\![\{p_1 \texttt{let}\ x = 3; p_2\}]\!] = \left| \begin{array}{l} \texttt{Local} \\ \texttt{DecVar x} \\ [\![p_1]\!] \\ \texttt{CstNb 3} \\ \texttt{SetVar x} \\ [\![p_2]\!] \\ \texttt{Global} \end{array} \right.$$

Optimizing locality: host variable in the stack

Local and Global can sometime be optimized

Use `DecStk`, `GetStk n`, `SetStk n` in place of `DecVar x`, `GetVar x`, `SetVar n`

$[\![\{\texttt{let}\ x = 5; \texttt{let}\ y = 7 + x;\ x + y;\}]\!] =$

```
DecStk      GetStk 3    AddiNb
DecStk      AddiNb      Swap
CstNb 5     SetStk 1    Drop
SetStk 2    GetStk 2    Swap
CstNb 7     GetStk 2    Drop
```

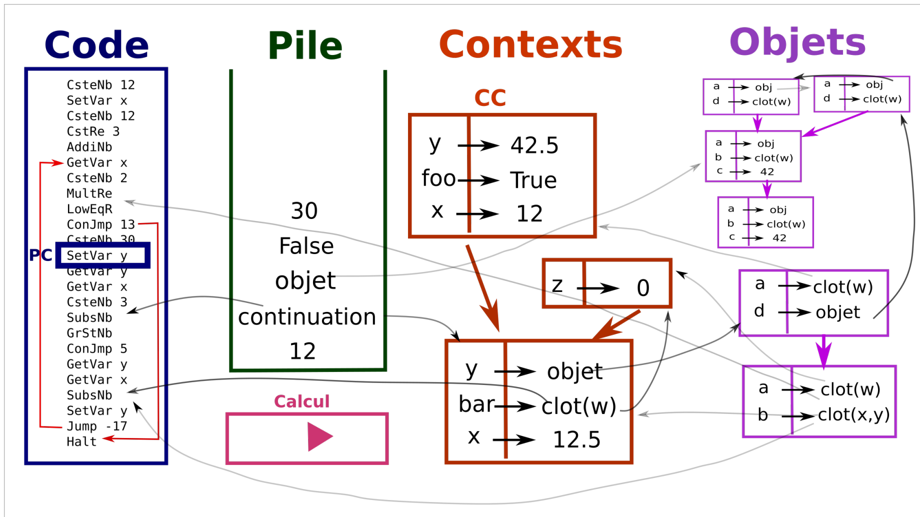
DecStk	Push(undefined)	stk	u:stk
GetStk n	Push(Stk[n])	$v_1 : \cdots : v_n$:stk	$v_n : v_1 : \cdots : v_n$:stk
SetStk n	Stk[n] = Pop	$v' : v_1 : \cdots : v_n$:stk	$v_1 : \cdots : v'$:stk

Dificulty: knowing variable's depth in the stack

Impossibility of closure

Closures will not see variables in the stack.

Objects (JS objects ≠ Java objects)



Objects are values

An object is also implemented as a dictionary

Attributs and methodes are both values associated to a name.

```
{ attr1: 42,
  attr2: 256,
  meth1: function (x)
             {return x+1;}
}
```

```
NewObj             # function meth1
CsteNb 42          GetVar x
SetObj attr1       CsteNb 1
CsteNb 256         AddiNb
SetObj attr2       Return
NewClo 3
DecArg x
SetObj meth1
Halt
```

NewObj	Push(NewObject{})	stack	o:stack
SetObj nom	Pull.cont.Set(nom, Pop)	v:o:stack	o:stack

Assignation et récupération dans un object

```
z = {
  a: 42
  b: 0
  }
z.b = 1 + z.a;
```

```
#créer z    #z.b = z.a+1;
NewObj      GetVar z
CsteNb 42   CsteNb 1
SetObj a    GetVar z
CsteNb 0    GetObj a
SetObj b    AddiNb
SetVar z    SetObj b
```

GetObj nom	Push(Pop.Get(nom))	o:stack	v:stack
SetObj nom	Pull.cont.Set(nom, Pop)	v:o:stack	o:stack

GetObj and SetObj

Those are similar to GetVal or SetVal but look at an object on the stack rather than CC.

The keyword this in JS

```
foo = {
  a: 41,
  getA: function (x)
        {return this.a;}
};

foo.a=foo.a+1;
```

The keyword this in JS

```
foo = {
  a: 41,
  getA: function (x)
        {return this.a;}
};
```

```
foo.a=foo.a+1;
```

$\longrightarrow 42$

The keyword this in JS

```
foo = {
  a: 41,
  getA: function (x)
        {return this.a;}
};
```

```
foo.a=foo.a+1;
```

$\longrightarrow$ 42

```
foo.getA();
```

The keyword this in JS

```
foo = {
  a: 41,
  getA: function (x)
        {return this.a;}
};
```

```
foo.a=foo.a+1;
```
$\longrightarrow$ 42

```
foo.getA();
```
$\longrightarrow$ 42

The keyword `this` in JS

```
foo = {
  a: 41,
  getA: function (x)
        {return this.a;}
};
```

```
foo.a=foo.a+1;
```
$\longrightarrow$ 42

```
foo.getA();
```
$\longrightarrow$ 42

```
bar = {
  a: foo.getA()-42,
  getA: foo.getA
}
```

The keyword this in JS

```
foo = {
  a: 41,
  getA: function (x)
        {return this.a;}
};
```

```
foo.a=foo.a+1;
```
$\longrightarrow$ 42

```
foo.getA();
```
$\longrightarrow$ 42

```
bar = {
  a: foo.getA()-42,
  getA: foo.getA
}
```

```
bar.a+bar.a;
```

The keyword this in JS

```
foo = {
  a: 41,
  getA: function (x)
        {return this.a;}
};
```

```
foo.a=foo.a+1;
```
$\longrightarrow 42$

```
foo.getA();
```
$\longrightarrow 42$

```
bar = {
  a: foo.getA()-42,
  getA: foo.getA
}
```

```
bar.a+bar.a;
```
$\longrightarrow 0$

The keyword this in JS

```
foo = {
  a: 41,
  getA: function (x)
        {return this.a;}
};
```

```
foo.a=foo.a+1;
```
$\longrightarrow$ 42

```
foo.getA();
```
$\longrightarrow$ 42

```
bar = {
  a: foo.getA()-42,
  getA: foo.getA
}
```

```
bar.a+bar.a;
```
$\longrightarrow$ 0

```
bar.getA();
```

The keyword `this` in JS

```
foo = {
  a: 41,
  getA: function (x)
        {return this.a;}
};
```

```
foo.a=foo.a+1;
```
$\longrightarrow$ 42

```
foo.getA();
```
$\longrightarrow$ 42

```
bar = {
  a: foo.getA()-42,
  getA: foo.getA
}
```

```
bar.a+bar.a;
```
$\longrightarrow$ 0

```
bar.getA();
```
$\longrightarrow$ 0

In JS, "this" refers to the object from which is extracted the curent closure

```
foo = { a:42, getA: function(){return this.a}};
bar = {a:0, getA:foo.getA}
bar.getA();
```

```
#créer foo     #créer bar     #bar.getA      Halt
NewObj         NewObj         GetVar bar
CsteNb 0       CsteNb 42      Copy           #code f
SetObj a       SetObj a       GetObj getA     GetVar this
NewClo 21      GetVar foo     StCall         GetObj a
SetObj getA    GetObj getA    Swap           return
SetVar foo     SetObj getA    SetIn this
               SetVar bar     Call
```

SetIn n	Push(Pop.Set(n, Pop))	v:c:stack	c:stack

Dur to the keyword this, the operatot "." is polymorph

```
z = {
a: 42,
getA: function (x)
    {return this.a;}
};
z.getA();
```

```
#créer z      #z.getA()     Call
NewObj        GetVar z      Halt
CsteNb 42      Copy          #getA
SetObj a      GetObj getA   GetVar this
NewClo 12     StCall        GetObj a
DecArg x      Swap          return
SetObj getA   SetIn this
SetVar z      StCall
```

The behaviour of "." in ob.var is type dependant

- on a closure, one has to add "this" in the context,
- On any other value, no need to do anything.

→ we need dynamic typing mecanisms...

Is the operator "." realy an operator ?

In JS keywords can be use as atributes

```
z = {
  if: 42,
  else: 0
};
z.if;
```

Posible solution : modify the LEXER !

"if:" and ".if" can be considered as lexems vastly diferent from "if"

Remark: to consider "if" and "if :" as diferent lexems, the maximum much is needed.

History of JS classes: Starts as syntactic sugar hiding a constructor

```
class MaClasse {
  attr1 = 42;
  attr2;
  meth1(){return 128;}
}
new Maclasse();
```

$\simeq$

```
function new_MaClasse (){
  return {
    attr1 : 42,
    attr2 : undefined,
    meth1 : function (){
              return 128;
  }         }
}
new_MaClasse();
```

History of JS classes: Starts as syntactic sugar hiding a constructor

```
class MaClasse {
  attr1 = 42;
  attr2;
  meth1(){return 128;}
  constructor(x){
    this.attr2 = x+a;
    }
}
new MaClasse(64);
```

$\simeq$

```
function new_MaClasse (x){
  rez = {
    attr1 : 42,
    attr2 : undefined,
    meth1 : function (){
              return 128;
  }          }
  rez.attr2 = x+a;
  return rez;
}
new_MaClasse(64);
```

Consequence: environement binded to the point where the class is created

```
function f(x) {
  var y=42
  class MaClass{
    x = y;
    g() {return y++;}
  }
  return MaClass;
}
maclasse = f(42);
ob1 = new maclasse();
ob1.g();
ob2 = new maclasse();
ob1.x;
ob2.x;
```

Consequence: environement binded to the point where the class is created

```
function f(x) {
  var y=42
  class MaClass{
    x = y;
    g() {return y++;}
  }
  return MaClass;
}
maclasse = f(42);
ob1 = new maclasse();
ob1.g();
ob2 = new maclasse();
ob1.x;
ob2.x;
```

$\longrightarrow$ 42

$\longrightarrow$ 43

History of JS classes: Need for factorisation in classes whith many methodes

Slow initialisation

For each execution of the constructor, the closures are reinitialised.

Inneficient space management

Methodes closures are integrates in each objects.

Example: For chained lists, each node would contains more than 70 closures which have to be initialised at each insertion !

History of JS classes: Prototype containing a constructor

```
class MaClasse {
  attr1 = 42;
  attr2;
  meth1(){
    return 128;
  }
  meth2(){
    this.attr1++;
  }
}
new Maclasse();
```

$\simeq$

```
MaClasse = {
  meth1 : function (){return 128;}
  meth2 : function (){this.attr1++;}
  constructor : function () {
    return {
      __proto__ : MaClasse,
      attr1 : 42,
      attr2 : undefined
    };
  }
}
MaClass.constructor();
```

JS object : attributs + prototype (⊇methodes)

objet1

nom	"Compilation"
mcc	function (pr,p2){....}

objet1

nom	"Calculabilité"
mcc	function (p1,p2){....}

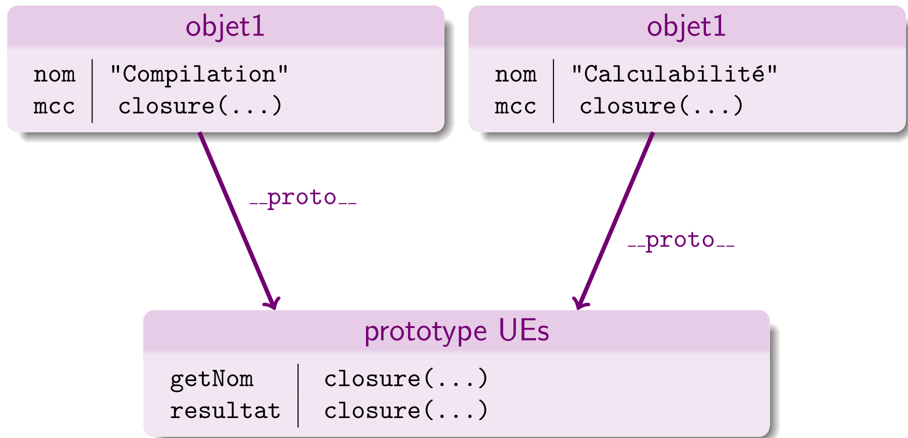
__proto__ → prototype UEs

__proto__ → prototype UEs

prototype UEs

getNom	function(){return this.nom}
resultat	function(eleve){....}

JS object : attributs + prototype (⊇methodes)



Inheritence = Prototypes imbrication