Introduction

Contract
- Specifies the interface of a software component
- Obligations - Benefits

TreatJS
- Language embedded contract system for JavaScript
- Enforced by run-time monitoring
- Inspired by existing contract systems

Features

- Standard abstractions for higher-order-contracts (base, function, and dependent contracts) [Findler,Felleisen’02]
- Intersection and union contracts
- Side-effect free contract execution
- Contract constructors generalize dependent contracts
Base Contract [Findler,Felleisen'02]

- Base Contracts are built from predicates
- Specified by a plain JavaScript function

```javascript
function isNumber (arg) {
  return typeof arg === 'number';
}
var Number = Contract.Base(isNumber);
assert(1, Number);
assert('a', Number); // blame the subject
```

Subject $v$ gets blamed for Base Contract $B$ iff: $B(v) = \text{false}$

Notizen

Function Contract [Findler,Felleisen'02]

```javascript
// Number × Number → Number
function plus (x, y) {
  return (x + y);
}
var plus = assert(plus,
  Contract.Function([Number, Number], Number));
```

Notizen

Function Contract [Findler,Felleisen'02]

```javascript
// Number × Number → Number
function plus (x, y) {
  return (x + y);
}
plus('a', 'a'); // blame the context
```

Context gets blamed for $C \rightarrow C'$ iff: Argument $x$ gets blamed for $C$ (as subject)
Function Contract [Findler, Felleisen ’02]

```javascript
// Number × Number → Number
function plusBroken(x, y) {
  return (x > 0 && y > 0) ? (x + y) : 'Error';
}
```

```javascript
plusBroken(0, 1); // blame the subject
```

Subject \( f \) gets blamed for \( \mathcal{C} \rightarrow \mathcal{C}' \) if:

\[ \neg (\text{Context gets blamed } \mathcal{C}) \land (f(x) \text{ gets blamed } \mathcal{C}') \]

Roman Matthias Keil, Peter Thiemann

TreatJS

July 8, 2015

6 / 18

New!

Overloaded Operator

- Function `+` works for strings, too
- Requires to model overloading and multiple inheritances

```javascript
// Number × Number → Number
function plus(x, y) {
  return (x + y);
}
```

```javascript
plus('a', 'a'); // blame the context
```

Roman Matthias Keil, Peter Thiemann

TreatJS

July 8, 2015

7 / 18
Intersection Contract

1 // (Number × Number → Number) ∩ (String × String → String)
2 function plus (x, y) {
3     return (x + y);
4 }
5
6 var plus = assert(plus, Contract.Intersection(
7     Contract.Function([Number, Number], Number),
8     Contract.Function([String, String], String)));

Roman Matthias Keil, Peter Thiemann
TreatJS

Notizen

Intersection Contract

1 // (Number × Number → Number) ∩ (String × String → String)
2 function plusBroken (x, y) {
3     return (x > 0 && y > 0) ? (x + y) : 'Error';
4 }
5
6 plusBroken(0, 1); X blame the subject

Subject f gets blamed for C ∩ C' iff:
(f gets blamed for C) ∨ (f gets blamed for C')

Roman Matthias Keil, Peter Thiemann
TreatJS

Notizen

Intersection Contract

1 // (Number × Number → Number) ∩ (String × String → String)
2 function plus (x, y) {
3     return (x + y);
4 }
5
6 plus(true, true); X blame the context

Context gets blamed for C ∩ C' iff:
(Context gets blamed for C) ∧ (Context gets blamed for C')

Roman Matthias Keil, Peter Thiemann
TreatJS

Notizen
Union Contract

1 // (Number × Number → Number) ∪ (Number × Number → String)
2 function plusBroken (x, y) {
3     return (x>0 && y>0) ? (x + y) : 'Error';
4 }
5
6 var plusBroken = assert(plusBroken, Contract.Union/
7     Contract.Function([[Number, Number], Number])
8     Contract.Function([[Number, Number], String]));

Context gets blamed for $C \cup C'$ iff:
(Context gets blamed for $C$) ∨ (Context gets blamed for $C'$)

Subject $f$ gets blamed for $C \cup C'$ iff:
($f$ gets blamed for $C$) ∧ ($f$ gets blamed for $C'$)
Contract Assertion

A failing contract must not signal a violation immediately.
Violation depends on combinations of failures in different sub-contracts.

1. $(\text{Number} \rightarrow \text{Number}) \cap (\text{String} \rightarrow \text{String})$
2. function addOne(x) {
    return (x + 1);
}
3. addOne('a');

Roman Matthias Keil, Peter Thiemann
TreatJS
July 8, 2015 10 / 18
Blame Calculation

- Contract assertion must connect each contract with the enclosing operations.
- Callback implements a constraint and links each contracts to its next enclosing operation.
- Reports a record containing two fields, context and subject.
- Fields range over $B_4 = \{ \bot, f, t, \top \}$ [Belnap'1977]

 Callback Graph

```
(Number -> Number) \cap (String -> String)
```

```javascript
function addOneBroken(x) {
    return (x + '1');
}
```

```
addOneBroken('a'); // blame the subject
```
Non-Interference

- No syntactic restrictions on predicates
- Problem: Contract may interfere with program execution
- Solution: Predicate evaluation takes place in a sandbox

```javascript
function isNumber(arg) {
    type = (typeof arg);
    return type === 'number';
}

var Number = Contract.Base(isNumber);
```

Roman Matthias Keil, Peter Thiemann
TreatJS
July 8, 2015

Sandbox

- All contracts guarantee noninterference
- Read-only access is safe

```javascript
var Array = Contract.Base(function(arg) {
    return (arg instanceof Array);
});
```
Sandbox

- All contracts guarantee noninterference
- Read-only access is safe

```javascript
var _Array_ = Contract.Base(function (arg) {
    return (arg instanceof OutsideArray);
});
var _Array_ = Contract.With([OutsideArray, _Array_], _Array_);
```

Contract Constructor

- Building block for dependent, parameterized, abstract, and recursive contracts
- Constructor gets evaluated in a sandbox, like a predicate
- Returns a contract
- No further sandboxing for predicates

```javascript
var _Type_ = Contract.Constructor(function (type) {
    return Contract.Base(function (arg) {
        return typeof arg === type;
    });
});
var _Number_ = _Type_('number');
```

Contract Abstraction

```javascript
// T × T → T
function plus (x, y) {
    return (x + y);
}
var _Plus_ = Contract.Constructor(function (_Type_) {
    return Contract.Function([_Type_, _Type_], _Type_);
});
var _Plus_ = assert(plus, _Plus_);
Plus(_Number_)(1, 2); // true
```
Dependent Contract

```javascript
// T × T → T
function plus(x, y) {
    return (x + y);
}

var __Type__ = Contract.Constructor(function(x, y) {
    return Contract.Base(function(arg) {
        return ((typeof x) === (typeof y)) &&
        ((typeof x) === (typeof arg));
    });
});

var plus = assert(plus, Contract.Dependent(__Type__));

plus(1, 2);
```

Conclusion

- TreatJS: Language embedded, dynamic, higher-order contract system for full JavaScript
- Support for intersection and union contracts
- Systematic blame calculation
- Composable sandboxing that guarantees non-interference
- Contract constructors with local scope