TreatJS
Higher-Order Contracts for JavaScript

Matthias Keil, Peter Thiemann
Institute for Computer Science
University of Freiburg
Freiburg, Germany

June 30, 2014, Dagstuhl Seminar on Scripting Languages and Frameworks: Analysis and Verification
Introduction

TreatJS

- Language embedded contract system for JavaScript
- Enforced by run-time monitoring
- Inspiration similar to *Contracts.js* [Disney]

Contract

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

- Standard abstractions for higher-order-contracts (base, function, and object contracts) [Findler,Felleisen’02]
- Support for boolean combinations (and, or, not contracts) as building blocks for intersection, union, and implication
- Contract constructors generalize dependent contracts
- Non-interference for contract execution
Base Contracts are built from predicates
- Specified by a plain JavaScript function

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

var IsNumber = BaseContract(typeOfNumber, 'IsNumber');

assert(1, IsNumber); ✓
```

- Value $v$ fulfills $\text{BaseContract}(B)$ iff: $B(v) = true$
Base Contracts are predicates
- Specified by a plain JavaScript function

```
function typeOfNumber (arg) {
    return (typeof arg) === 'number';
}
var IsNumber = BaseContract(typeOfNumber, 'IsNumber');

assert('a', IsNumber); X Blame the Value
```

- Value $v$ gets blamed for contract $B$ iff: $B(v) = false$
// Number × Number → Number

function addUnchecked(x, y) {
    return (x + y);
}

var addChecked = assert(addUnchecked, FunctionContract([IsNumber, IsNumber], IsNumber));
Function Contract [Findler,Felleisen’02]

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

addChecked(1, 1); ✓
```

- $x$ is an acceptable argument for contract $C \rightarrow C'$ iff: 
  ($x$ fulfills $C$)

- Function $f$ fulfills contract $C \rightarrow C'$ at argument $x$ iff: 
  ($x$ fulfills $C$) $\rightarrow$ ($f(x)$ fulfills $C'$)
Function Contract [Findler, Felleisen’02]

```javascript
// Number × Number → Number

function addUnchecked(x, y) {
  return (x + y);
}

addChecked('a', 'a'); X Blame the Argument
```

Argument x gets blamed for $C \rightarrow C'$ iff:

$\neg (x$ is an acceptable argument for contract $C \rightarrow C'$) iff:

$\neg (x$ fulfills $C$)
Function Contract [Findler,Felleisen’02]

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

addChecked(0, 1); // Blame the Function
```

- Function $f$ gets blamed for $C \rightarrow C'$ at argument $x$ iff:
  $\neg (\text{Function } f \text{ fulfills contract } C \rightarrow C' \text{ at argument } x)$ iff:
  $(x \text{ fulfills } C) \land \neg (f(x) \text{ fulfills } C')$
New!
// Number × Number → Number

function addUnchecked(x, y) {
    return (x + y);
}

addChecked('a', 'a'); X
/* (Number × Number → Number) ∩ (String × String → String) */

function addUnchecked(x, y) {
    return (x + y);
}

var addChecked = assert(addUnchecked, Intersection(FunctionContract([IsNumber, IsNumber], IsNumber), FunctionContract([IsString, IsString], IsString));
// (Number × Number → Number) ∩ (String × String → String)

function addUnchecked(x, y) {
    return (x + y);
}

addChecked('a', 'a'); ✓

- x is an acceptable argument for contract C ∩ C' iff:
  (x is acceptable arg. for C) ∨ (x is acceptable arg. for C')

- Function f fulfills contract C ∩ C' at argument x iff:
  (f fulfills C at x) ∧ (f fulfills C' at x)
// (Number × Number → Number) ∩ (String × String → String)

function addUnchecked(x, y) {
  return (x + y);
}

addChecked(true, true); X Blame the Argument

- Argument x gets blamed for $C \cap C'$ iff:
- $\neg (x$ is an acceptable argument for contract $C \cap C'$)
- $\neg ((x$ is acc. arg. for $C$) ∨ (x is acc. arg. for $C'$))
- $\neg (x$ is acc. arg. for $C$) ∧ $\neg (x$ is acc. arg. for $C'$)
- (x gets blamed for $C$) ∧ (x gets blamed for $C$)
Intersection Contract

1. // (Number × Number → Number) ∩ (String × String → String)

2. function addUnchecked(x, y) {
   return (x > 0 && y > 0) ? (x + y) : 'Error';
}

3. addChecked(0, 1); X Blame the Function

- Function f gets blamed for C ∩ C’ at argument x iff:
  (f gets blamed for C at x) ∨ (f gets blamed for C’ at x)
Union Contract

// (Number × Number → Number) ∪ (Number × Number → String)

function addUnchecked(x, y) {
    return (x > 0 && y > 0) ? (x + y) : 'Error';
}

var addChecked = assert(addUnchecked, Union(
    FunctionContract([IsNumber, IsNumber], IsNumber)
    FunctionContract([IsNumber, IsNumber], IsString)));
// (Number × Number → Number) ∪ (Number × Number → String)

function addUnchecked(x, y) {
    return (x>0 && y>0) ? (x + y) : 'Error';
}

addChecked(0, 1); ✓

- Argument x is an acceptable argument for contract $C \cup C'$ iff:
  $(x$ is acceptable arg. for $C) \land (x$ is acceptable arg. for $C')$

- Function $f$ fulfills contract $C \cup C'$ at argument $x$ iff:
  $(f$ fulfills $C$ at $x) \lor (f$ fulfills $C'$ at $x)$
Union Contract

1. \((\text{Number} \times \text{Number} \to \text{Number}) \cup (\text{Number} \times \text{Number} \to \text{String})\)

2. function addUnchecked(x, y) {
   return (x > 0 && y > 0) ? (x + y) : 'Error';
}

3. addChecked('a', 'a'); \(\times\) Blame the Argument

- Argument \(x\) gets blamed for \(\mathcal{C} \cup \mathcal{C}'\) iff:
  \((x\) gets blamed for \(\mathcal{C}\)) \lor (x\) gets blamed for \(\mathcal{C}'\)\)
// (Number × Number → Number) ∪ (Number × Number → String)

```javascript
function addUnchecked(x, y) {
  return (x > 0 && y > 0) ? (x + y) : undefined;
}
```

```javascript
addChecked(0, 1); // Blame the Function
```

- Function \( f \) gets blamed for \( C \cup C' \) at argument \( x \) iff:
  \[ (f \text{ gets blamed for } C \text{ at } x) \land (f \text{ gets blamed for } C' \text{ at } x) \]
Non-Interference

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

```javascript
function typeOfNumber (arg) {
    type = (typeof arg); \x20 Access\ forbidden
    return type === 'number';
}

var FaultyIsNumber =
    BaseContract(typeOfNumber, 'FaultyIsNumber');
```
Non-Interference

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

```javascript
var isArray = BaseContract(function (arg) {
    return (arg instanceof Array); // Access forbidden
});
```
Non-Interference

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

```javascript
var IsArray = BaseContract(function (arg) {
    return (arg instanceof InsideArray);
});

var IsArrayComplete = With({InsideArray:Array}, IsArray);
```
Contract Constructor

// $T \times T \rightarrow T$

function addUnchecked(x, y) {
    return (x + y);
}

var addChecked = assert(addUnchecked,
    FunctionContract([[CheckType, CheckType], CheckType]));

var CheckType = BaseContract(function (arg) {
    // to be completed
});
Contract Constructor

- Constructor gets evaluated in a sandbox, like a predicate
- Returns a contract
- No further sandboxing for predicates

```javascript
var ctor = Constructor(function() {
  var type = undefined;
  var CheckType = BaseContract(function (arg) {
    type = type || (typeof arg);
    return type === (typeof arg);
  });
  return FunctionContract([CheckType, CheckType], CheckType):
  }, 'SameType');
```
```
var SameTypeCtor = Constructor(function(arg) {
    var type = (typeof arg);
    return BaseContract(function (arg) {
        return (typeof arg) === type);
    });
}

var addChecked = assert(addUnchecked, DependentContract(SameTypeCtor));
```
Conclusion

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