Questioning Gradual Typing

Timothy Jones <tim@montoux.com>

November 4, 2018

Montoux

Last Time

- Gradual typing is morally incorrect

Last Time

- Gradual typing is morally incorrect
- We’re all monsters now

This Time

- The Gradual Guarantee

- Dynamic Type Errors

- Gradual checks in Grace

This Time

- The Gradual Guarantee
    - Is it a useful property?
- Dynamic Type Errors
- Gradual checks in Grace

This Time

- The Gradual Guarantee
    - Is it a useful property?
- Dynamic Type Errors
    - What determines if a value satisfies a type assertion?
- Gradual checks in Grace

This Time

- The Gradual Guarantee
    - Is it a useful property?
- Dynamic Type Errors
    - What determines if a value satisfies a type assertion?
- Gradual checks in Grace
    - How should we interpret types?

The Gradual Guarantee

If an expression e_1 evaluates without error in one step to e_2, then any e_1' where $e_1' \sqsubseteq e_1$ also evaluates in zero or more steps to e_2' where $e_2' \sqsubseteq e_2$.

Grace

```
method assertString(x : String) {}

method classify(o : Unknown) → String {
    try {
        assertString(o)
        return “string”
    } catch { e : TypeError →
        return “not string”
    }
}
```

Dart

```
assertString(String x) {}

classify(o) {
    try {
        assertString(o);
        return “string”;
    } catch(e) {
        return “not string”;
    }
}
```

Typed Racket

```
(require/typed racket
    [(identity assertString) (→ Any String)])

(define (classify o)
    (with-handlers ([exn:fail:contract? (λ (e) “not string”)])
        (assertString o)
        “string”))
```

Reticulated Python

```
def assertString(x: str):
    pass

def classify(o):
    try:
        assertString(o)
        return “string”
    except:
        return “not string”
```

Higher-order Casts

```
def assertFloatList(l: List(float)):
    for x in l:
        pass

def classify(o):
    try:
        assertFloatList(o)
        return “float list”
    except CastError:
        return “checked, it’s not a float list”
    except RuntimeCheckError:
        return “oops, it’s not a float list”

classify([1, “x”])
```

Hack

```
function errorhandler($errno, $errstr, $errfile, $errline) {
    if ($errno == E_RECOVERABLE_ERROR) {
        print “not ”
        return true;
    }
    return false;
}

function assertString(string $x) {}
function classify(o) {
    set_error_handler(‘errorhandler’);
    assertString(o);
    print “string”
}
```

A New Gradual Guarantee?

If an expression e_1 *containing no traps for failed typecasts* evaluates without error in one step to e_2, then any e_1' where $e_1' \sqsubseteq e_1$ also evaluates in zero or more steps to e_2' where $e_2' \sqsubseteq e_2$.

Another Solution

- Ensure that type errors are irrevocably fatal

- Maybe calculi can get away with this…

Gradual Guarantee

- How important is the guarantee?

- What other language constructs interfere with it?

The Source of Truth

Who Decides What Fails?

- When should a dynamically well-typed program fail?

Who Decides What Fails?

- When should a dynamically well-typed program fail?
- What if every object satisfies every assertion?

Grace(ish)

```
method forget(x : Object) → Unknown { x }

method remember⟦T⟧(x : Unknown) → T { x }

type Sized = interface { size → Number }
def sized = object { method size → Number { 5 } }

remember⟦Sized⟧(forget(sized))
```

Reticulated Python

```
def forget(x: {}) → any:
    return x

def remember(x: any) → {“size”: int}:
    return x

class Sized(object):
    def size(self):
        return 5

remember(forget(Sized()))
```

Typed Racket

```
(require/typed racket
   [(identity remember) (→ Any Sized)])

(define-type Sized
   (Object [size (→ Integer)]))

(define sized : Sized
   (make-object (class object%
                  (super-new)
                  (define/public (size) 5))))

(define (forget [x : (Object)]) : Any x)

(remember (forget sized))
```

In Practice

- The object’s interface determines if a cast fails

In Practice

- The object’s interface determines if a cast fails
- What does the theory say?

$\mathrm{Ob}_{<:}^{\langle\cdot\rangle}$

$$\langle[\mathrm{size} : \mathbb{Z}] \Leftarrow ?\rangle\, \langle ? \Leftarrow [\mathrm{size} : \mathbb{Z}]\rangle\, [\mathrm{size} = 5]$$

$\mathrm{Ob}^{?}_{<:}$

$$\mathrm{forget}(t) = [\mathrm{id} = ?\ \varsigma(x : [])\, x].\mathrm{id}(t)$$

$$\mathrm{remember}(t, T) = [\mathrm{id} = T\ \varsigma(x : ?)\, x].\mathrm{id}(t)$$

$$\mathrm{remember}(\mathrm{forget}([\mathrm{size} = 5]), [\mathrm{size} : \mathbb{Z}])$$

$\mathrm{Ob}^{?}_{<:}$

$$\mathrm{forget}(t) = [\mathrm{id} = ?\ \varsigma(x : [])\, x].\mathrm{id}(t)$$

$$\mathrm{remember}(t) = [\mathrm{id} = [\mathrm{size} : \mathbb{Z}]\ \varsigma(x : ?)\, x].\mathrm{id}(t)$$

$$\mathrm{remember}(\mathrm{forget}([\mathrm{size} = 5]))$$

Cast Insertion

$$\mathrm{forget}(t) = [\mathrm{id} = ?\ \varsigma(x : [])\ \langle ? \Leftarrow [] \rangle\ x].\mathrm{id}(t)$$

$$\mathrm{remember}(t) = [\mathrm{id} = [\mathrm{size} : \mathbb{Z}]\ \varsigma(x : ?)\ \langle [\mathrm{size} : \mathbb{Z}] \Leftarrow ? \rangle\ x].\mathrm{id}(t)$$

$$\mathrm{remember}(\mathrm{forget}([\mathrm{size} = 5]))$$

Stuck Cast

$$\langle[\mathsf{size} : \mathbb{Z}] \Leftarrow ?\rangle\, \langle ? \Leftarrow []\rangle\, [\mathsf{size} = 5]$$

What Happened?

- Subsumption

What Happened?

- Subsumption
- There is no path to a fully-typed program

What Happened?

- Subsumption
- There is no path to a fully-typed program
- Is this a desirable property of gradual typing?

Object-Oriented Types

Extensible Contracts

```
type Contract⟦T⟧ = interface {
    matches(value) → MatchResult⟦T⟧
}

type MatchResult⟦T⟧ = MatchFailure ∪ MatchSuccess⟦T⟧

type MatchSuccess⟦T⟧ = true ∩ interface {
    result → T
}
```

Extensible Contracts

```
method m(x : A) → B {
    . . .
}
```

Extensible Contracts

```
method m(x) {
    def pre = A.matches(x)
    pre.assert
    def x = pre.result
    def post = B.matches(···)
    post.assert
    post.result
}
```

Extensible Contracts

- Flat
    - result is the tested object

Extensible Contracts

- Flat
    - result is the tested object
- Chaperone
    - result is a transparent proxy around the object

Extensible Contracts

- Flat
  - result is the tested object
- Chaperone
  - result is a transparent proxy around the object
- Impersonator
  - result is whatever the match wants

Extensible Contracts

- Flat
    - result is the tested object
- Chaperone
    - result is a transparent proxy around the object
- Impersonator
    - result is whatever the match wants
    - Except it must satisfy T: MatchSuccess is a chaperone

Erasing Checks

- A dialect might be free to erase checks
- Does it need to prove the erasure is behaviour-preserving?

Questioning Gradual Typing

Questions for Grace

- Is it appropriate for MatchResult to be a boolean?
- What class of exception is a TypeError?
- Should we consider subsumption during type tests?
- How can a dialect communicate what it knows to the runtime?