Gradual typing is morally incorrect; we're all monsters now

Timothy Jones and Michael Homer

Victoria University of Wellington

{tim,mwh}@ecs.vuw.ac.nz

October 27, 2015

Meaning in the gradually typed world

A type assertion should be meaningful

What do expect this to mean in the context of gradual typing?

method foo(x : A) $\rightarrow$ B

Gradual typing is morally incorrect

The level of knowledge the system has can change behaviour

Morally correct behaviour:

- Raise an error when we know an assertion is not satisfied
- Place the blame on ill-typed code

Gradual typing is morally incorrect

The level of knowledge the system has can change behaviour

Morally correct behaviour:

- Raise an error when we know an assertion is not satisfied
- Place the blame on ill-typed code
- Know as much as possible

Gradual typing is morally incorrect

The level of knowledge the system has can change behaviour

Morally correct behaviour:

- Raise an error when we know an assertion is not satisfied
- Place the blame on ill-typed code
- Know as much as possible
- Prevent interaction with objects which are known to be ill-typed

We're all monsters now

Concession to the pragmatists: we're not moral either

It is necessary that:

- Much more information is retained
- All type errors are fatal

Gradual typing

Typed and untyped worlds can interact

- Macro and micro interpretations of worlds

Runtime enforcement of type assertions

- Refinement of optional typing

Well-typed programs can't be blamed

- Provides a standard for soundness

Languages

$\lambda^{?}_{\rightarrow}$ and $\mathbf{Ob}^{?}_{<:}$ (Siek and Taha)

The Blame Calculus (Wadler and Findler)

Typed Racket (PLT, Tobin-Hochstadt *et al.*)

Reticulated Python (Vitousek *et al.*)

Thorn (Wrigstad *et al.*), SafeScript (Richards *et al.*), etc.

Languages

$\lambda^{?}_{\rightarrow}$ and $\mathbf{Ob}^{?}_{<:}$ (Siek and Taha)

The Blame Calculus (Wadler and Findler)

Typed Racket (PLT, Tobin-Hochstadt *et al.*)

Reticulated Python (Vitousek *et al.*)

Thorn (Wrigstad *et al.*), SafeScript (Richards *et al.*), etc.

Grace

Semantics

Basic checking is easy in a simple nominal world

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

foo(12) // Error: 12 does not satisfy the type String.
```

Semantics

Higher-order types cannot be conclusively checked

```
method foo(f : Function.from(Number) to(String)) {}

foo({ x → if (x ≥ 10) then { "big" } else { x } })
```

Structural types

Structural types are just sets of these function types

- We can check the functions exist, but not if they satisfy the type

```
let Bar = type { bar → Number }

method foo(x : Bar) → Number {
  x.bar // Raises an error here...
}

// ... Blaming this call site
foo(object { method bar { "12" } })
```

Semantics

How do we remember to check these constraints?

- Transient: rewrite the code to check method calls
- Guarded: indirect reference through a first-class contract
- Monotonic: permanently insert the contract into the object

Semantics

How do we remember to check these constraints?

- Transient: rewrite the code to check method calls
- Guarded: indirect reference through a first-class contract
- Monotonic: permanently insert the contract into the object

Each of these semantics has different behaviour

- Both spatial and temporal meanings differ between them

The Gradual Guarantee

Recent refinement of what it means to be ‘gradual’

- (Implied intent made explicit)

Type assertions don’t affect program behaviour

- Correct programs behave the same when any assertions are removed

Meaning what we say

What does it mean when I say, “You must give me an A”

What does it mean when I say, “I will give you a B”

- (Given that assumptions may be invalidated)

```
method foo(x : A) → B {
  e // Type-checked
}
```

Requirements

"You must give me an A":

- Transient: Must behave as A in the scope of the definition
- Guarded: Reference must behave as A
- Monotonic: Object must behave as A

Guarantees

"I will give you a B":

- Transient: If you gave me an A, you will get a B
- Guarded: Reference will behave as B (or blame the A)
- Monotonic: Object will behave as B (or blame the A)

Saying what we mean

Transient semantics cannot perform blame

Monotonic presented as more performant than guarded

- Both are sound up to blame
- But which maps more closely to our desired (intuitive) meaning?

Requirements

Guarded: *My view* of the object must behave as A

- It doesn't matter if the object doesn't actually satisfy A

Monotonic: The object must behave as A

- Interactions with the object *anywhere in the program* now perform checks

Transparent proxies

Guarded semantics wrap objects in transparent proxies

- Different views of the same object can have different behaviour

Consider when "untyped.rkt" defines y as an alias of x:

```
(define-type FooA (Instance (Class [foo (→ A)])))
(define-type FooB (Instance (Class [foo (→ B)])))
(require/typed "untyped.rkt" [x FooA] [y FooB])

(define (bar obj) (send obj foo))

(bar x) ; Fine: x satisfied FooA
(bar y) ; Type error
```

Mutating objects

Monotonic semantics can blame unrelated code

```
def foo(f : Function(Int, Int)):
  f(2)

def bar(f):
  f(6)

def cap(x):
  x if x < 5 else "Too big"

foo(cap) # Fine
bar(cap) # Type error, blaming call to foo
```

Moral correctness

Which of these behaviours is more surprising?

Discovering type information

Information about types can be discovered in many places

- Type assertions

Guarded and monotonic

Discovering type information

Information about types can be discovered in many places

- Type assertions
- Aliases of the same object ascribed different types

Monotonic only

Discovering type information

Information about types can be discovered in many places

- Type assertions
- Aliases of the same object ascribed different types
- Collapsing unions or generic types

Keil and Theimann

Discovering type information

Information about types can be discovered in many places

- Type assertions
- Aliases of the same object ascribed different types
- Collapsing unions or generic types
- Calling methods: what they accept and return

Only after an assertion

Union collapsing

Information about unions of types must collapse

type { foo → A } ∪ **type** { foo → B } ≠ **type** { foo → A ∪ B }

One will invalidate the other

x.foo // *If this is not a B...*

x.foo // *... returning a B must be a type error in the future*

Future behaviour

Future behaviour can invalidate contracts

```
let Foo = type { foo → Number }

method id(x : Foo) → Foo { x }

var z := id(y)
z.foo // Returns a String: blame call to id
```

We know that y does not satisfy the type Foo

Past behaviour

Past behaviour should also invalidate contracts

```
let Foo = type { foo → Number }

method id(x : Foo) → Foo { x }

y.foo // Returns a String
var z := id(y) // Type error?
```

We *should* know that y does not satisfy the type Foo

Catching exceptions considered harmful

Catching type errors leads to strange behaviour

- Invalidates the Gradual Guarantee
- Permits interacting with code known to be ill-typed

Practical implementations concerned with error compatibility

Probing type annotations

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

method fooTakesStrings → Boolean {
  try {
    foo(12)
    return false
  } catch { e : TypeError →
    return true
  }
}

if (fooTakesStrings) then { print(1) } else { print(2) }
```

Probing type annotations

```
method foo(x) {}

method fooTakesStrings → Boolean {
  try {
    foo(12)
    return false
  } catch { e : TypeError →
    return true
  }
}

if (fooTakesStrings) then { print(1) } else { print(2) }
```

Using invalidated objects

Should we be allowed access to known ill-typed objects?

```
(require/typed "untyped.rkt"
  [x (Instance (Class [foo (→ A)] [bar (→ B)]))])

(with-handlers
    [exn:fail:contract? (λ (e)



    )]
  (send x foo)) ; Raises a type error if foo does not return A
```

Using invalidated objects

Should we be allowed access to known ill-typed objects?

```
(require/typed "untyped.rkt"
  [x (Instance (Class [foo (→ A)] [bar (→ B)]))])

(with-handlers
    [exn:fail:contract? (λ (e)
      ; We can use x, even though we know that it is ill-typed
      (send x bar)
    )]
  (send x foo))
```

Using invalidated objects

What about in the monotonic semantics?

```
def foo(f : Function(A, B)) → B:
   return f(a)

try:
   foo(f) # f is permanently modified to ensure B when given A
except CastError:
   f(a) # f fails its permanent contract: can we pass it A now?
```

Achieving perfection

Record everything

- Types of every value that methods accept and return

Respond to everything

- Check all relevant contracts whenever anything happens

Achieving perfection

Record everything

- Types of every value that methods accept and return

Respond to everything

- Check all relevant contracts whenever anything happens

Typed/untyped interaction is no longer a bottleneck

Achieving perfection

Do away with blame

Achieving perfection

Do away with blame

Just travel back in time to the code which was at fault

- No more issues with try-catch, without requiring fatal errors
- Undoing dynamic typing (Benton)

Achieving perfection

Do away with blame

Use flow analysis to propagate all possible assertions into the past

Achieving perfection

Do away with blame

Use flow analysis to propagate all possible assertions into the past

- (that is, just infer conservative types on all untyped code)

Moral correctness

Anything less makes me uncomfortable, so must be wrong