CMSC 631 – Program Analysis and Understanding

Dynamic Typing, Contracts, and Gradual Typing

Static vs. Dynamic Typing

- Languages with Static Typing
  - Examples: Ocaml, Java, C#, Scala, Haskell
  - Typechecker proves lack of certain errors
  - Runtime refuses to run ill-typed program (usually)
  - Types also provide some level of documentation

- Languages with Dynamic Typing
  - Examples: Ruby, Python, JavaScript, Racket, Bash
  - Runtime protects against data misuses by raising errors
  - Dynamic languages often used for scripting, prototyping
  - Focus on testing and specification (e.g., RSpec)

Checking Dynamically Typed Programs

Adding assertions about the current program state

Adding contracts on program behavior

Transitioning programs piece-by-piece to a typed sister language

Assertions

- Recall from the symbolic execution project
- Often used to protect against unrecoverable errors
  - E.g., protecting against allocation failure in C
- Only describes where error happened, not why
  - No indication of what code caused the error
  - Backtrace may help...
  - ... but cause of error may not still be on the stack

Design by Contract

- Introduced by Bertrand Meyer in Eiffel
- Contracts part of class and method definitions
- Three type of contract clauses:
  - Preconditions – what must hold when a method is called
  - Postconditions – what must hold when a method returns
  - Invariants – what must hold at each method entry and exit
- Design by Contract libraries available for many object-oriented languages

Contracts in Eiffel

```
class PRIME_STACK
create
  make
feature
  is_empty : BOOLEAN is
    do
      Result := intlist.is_empty
    end
  push(new_item : INTEGER) is
    require
      item_is_prime: is_prime(new_item)
    do
      intlist.add(new_item)
    end
  pop : INTEGER is
    require
      stack_is_nonempty : is_empty = False
    do
      Result := intlist.remove
    ensure
      is_prime(Result)
    end
feature {NONE}
  make
    do
      create intlist
    end
  intlist : LIST [ INTEGER ]
invariant
  intlist_not_void : intlist /= Void
end
```

Design by Contract

- Focused on what's true at method entry and exit
  - E.g., invariants can be temporarily broken within a method
- Caller responsible for preconditions and invariants
- Callee responsible for postconditions and invariants

Issues with Design by Contract

- Does not handle higher-order behavior well
  - Contract clauses only check first-order properties
  - Functions represented as class with an apply method
    - Need a different class for each desired contract
    - Higher-order functions require class for each order as well
    - Burden on user to provide function of the appropriate class
- Does not handle duck typing well
  - "If it walks, swims, sounds like a duck..."
  - Focuses on behavior of objects, not lineage
- Discovers failures only on method entry/exit
- Every method entry/exit checked
  - Spec# – Explicitly mark regions where contracts not checked

Higher-Order Contracts

- Contracts for Higher-Order Functions
  - Findler, Felleisen – ICFP 2002
  - Most Influential ICFP Paper (ICFP 2012)
- Contracts added separately to existing values
- Contracts can specify higher-order behavior
  - Such behavior cannot be checked immediately
- Views contracts more as legal contracts
  - Contracts are between two parties: a provider and a user
  - Parties that enter into a contract are blamed for misbehavior
  - Each party only responsible for interactions with other party
    - In particular, not responsible for own use of provided values

Higher-Order Contracts in Racket

server

```
#lang racket

(define (deriv f dx)
  (lambda (x)
    (/ (- (f (+ x dx)) (f x)) dx)))

(provide/contract
  [deriv
   (-> (-> real? real?) real?
       (-> real? real?))])
```

client

```
#lang racket

(require server)

(define (sin x) ...)

(define cos (deriv sin 0.001))

(cos #f)
;; client broke the contract on
;; deriv: expected real?, got #f
;; for first argument of result
;; in (-> (-> real? real?) real?
;;        (-> real? real?))
```

Higher-Order Contract Semantics

$e ::= \ldots \mid \text{blame } x \mid e \rightarrow e \mid e^{e,x,x}$

$v ::= \ldots \mid v \rightarrow v \mid v^{v \rightarrow v,x,x}$

$E ::= \ldots \mid E \rightarrow e \mid v \rightarrow E \mid e^{E,x,x} \mid E^{v,x,x}$

$E[v^{\lambda x.e,p,n}] \longmapsto E[\text{if } (\lambda x.e)\ v \text{ then } v \text{ else blame } p]$

$E[v1^{v3 \rightarrow v4,p,n}\ v2] \longmapsto E[(v1\ v2^{v3,n,p})^{v4,p,n}]$

- Notice exchange of p and n on the function argument (v2)

More Higher-Order Contract Work

- Semantics of higher-order contracts
  - Findler and Blume. Contracts as Pairs of Projections. FLOPS 2006.
  - Dimoulas et al. Complete Monitors for Behavioral Contracts. ESOP 2012.
- Contracts for other higher-order behavior
  - Strickland and Felleisen. Contracts for First-Class Classes. DLS 2010.
  - Takikawa et al. Constraining Delimited Control with Contracts. ESOP 2013.
- Contracts for mutable values
  - Strickland et al. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. OOPSLA 2012.

Alternatives to Pure Static Typing

- Dynamic Types (Cardelli – CFPL 1985)
  - Dynamic-typed values pair typed values with their type
  - Dynamic values in typed positions check type at run-time
- Soft Typing (Cartwright, Fagan – PLDI 1991)
  - Adds explicit run-time checks where typechecker cannot prove type correctness
  - Allows running possibly ill-typed programs
- Gradual Typing
  - Parallel work
    - Tobin-Hochstadt and Felleisen. Interlanguage Migration. DLS 2006.
    - Siek and Taha. Gradual Typing for Objects. ECOOP 2007.
  - Focuses on providing sister typed and untyped languages
  - Allows interaction between typed and untyped modules

Gradual Typing Implementations

- Creating new pairs of typed/untyped languages
  - Thorn
  - Dart
- Creating a typed sister language for an existing untyped language
  - Typed Racket
  - core.typed for Clojure
  - Rtc for Ruby

Higher-Order Contracts in Racket

server

```
#lang racket

(provide/contract
  [deriv
   (-> (-> real? real?) real?
       (-> real? real?))])

(define (deriv f dx)
  (lambda (x)
    (/ (- (f (+ x dx)) (f x)) dx)))
```

client

```
#lang racket

(require server)

(define (sin x) ...)

(define cos (deriv sin 0.001))

(cos #f)
;; client broke the contract on
;; deriv: expected real?, got #f
;; for first argument of result
;; in (-> (-> real? real?) real?
;;        (-> real? real?))
```

Mixed Typed Racket Program

server

```
#lang typed/racket

(provide deriv)

(: deriv (Real -> Real) Real ->
         (Real -> Real))

(define (deriv f dx)
  (lambda (x)
    (/ (- (f (+ x dx)) (f x)) dx)))
```

client

```
#lang racket

(require server)

(define (sin x) ...)

(define cos (deriv sin 0.001))

(cos #f)
;; client broke the contract on
;; deriv: expected real?, got #f
;; for first argument of result
;; in (-> (-> real? real?) real?
;;        (-> real? real?))
```

Key Insight to Gradual Typing

Handle typed/untyped boundary crossings with dynamic checks (e.g., contracts)

Typing Racket Code

- Need to handle certain idioms not (usually) present in statically typed languages
  - “True” unions (instead of disjoint unions)
  - Type/tag checks (e.g., the real? predicate)
  - Assumptions about data based on control flow

Example:

```
(: value->string ((U String Number) -> String))


(define (value->string v)
  (cond
    [(number? v) (number->string v)]
    [(string? v) v])
(string-append (value->string 3) (value->string “foo”))
```

Typing Racket Code

- Need to handle certain idioms not (usually) present in statically typed languages
  - “True” unions (instead of disjoint unions)
  - Type/tag checks (e.g., the real? predicate)
  - Assumptions about data based on control flow

Solution: Occurrence Typing

Tobin-Hochstadt and Felleisen. Logical Types for Untyped Languages. ICFP 2010.

Typing Racket Code

- Need to handle certain idioms not (usually) present in statically typed languages
  - Applying functions to heterogenous rest args

Example:

```
(define (fold-left f a . lss)
  (if (ormap null? lss) a
      (fold-left f (apply f a (map car lss))
                   (map cdr lss))))
(fold-left (lambda (a n s)
             (string-append a (number->string n) s))
           “” (list 1 2 3) (list “foo” “bar” “baz”))
```

Typing Racket Code

- Need to handle certain idioms not (usually) present in statically typed languages
  - Applying functions to heterogenous rest args

Example:

```
(: fold-left (All (A B ...)
               ((A B ... -> A) A (List B) ... -> A)
(define (fold-left f a . lss)
  (if (ormap null? lss) a
      (fold-left f (apply f a (map car lss))
                 (map cdr lss))))
(fold-left (lambda (a n s)
             (string-append a (number->string n) s))
           “” (list 1 2 3) (list “foo” “bar” “baz”))
```

Typing Racket Code

- Need to handle certain idioms not (usually) present in statically typed languages
  - Applying functions to heterogenous rest args

Solution: Types for heterogenous variable-arity functions

Strickland, Tobin-Hochstadt, and Felleisen. Practical Variable-Arity Polymorphism. ESOP 2009.

Gradual Type Soundness

In a gradual typing system, type soundness looks something like the following:

For all programs, if the typed parts are well-typed, then evaluating the program either

1. produces a value,
2. diverges,
3. produces an error that is not caught by the type system (e.g., division by zero),
4. produces a run-time error in the untyped code, or
5. produces a contract error that blames the untyped code.

Typing Your Favorite Dynamic Language

- What are the idioms for your language?
- What types do you need to describe those idioms?
- How do you protect typed modules from untyped modules?