Interpretations of the Gradually-Typed Lambda Calculus

(Distilled Tutorial)

Jeremy G. Siek
University of Colorado at Boulder
jeremy.siek@colorado.edu

Ronald Garcia
University of British Columbia
rxg@cs.ubc.ca

Abstract

Gradual typing is an approach to integrating static and dynamic type checking within the same language [Siek and Taha 2006]. Given the name "gradual typing", one might think that the most interesting aspect is the type system. It turns out that the dynamic semantics of gradually-typed languages is more complex than the static semantics, with many points in the design space [Wadler and Findler 2009; Siek et al. 2009] and many challenges concerning efficiency [Herman et al. 2007; Hansen 2007; Siek and Taha 2007; Siek and Wadler 2010; Wrigstad et al. 2010; Rastogi et al. 2012]. In this distilled tutorial, we explore the meaning of gradual typing and the challenges to efficient implementation by writing several definitional interpreters and abstract machines in Scheme for the gradually-typed lambda calculus.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Procedures, functions, and subroutines

General Terms Languages, Theory

Keywords gradual typing, coercions, blame tracking, Scheme

1. Introduction

The early work on gradual typing focused on the type system, but within the first couple years the main pieces fell into place with the creation of the *consistency* relation to govern implicit casts involving the dynamic type [Siek and Taha 2006] and the integration of consistency with subtyping [Siek and Taha 2007]. Ever since then, the primary challenges have been in its dynamic semantics, especially regarding the treatment of casts for which there are many design choices and efficiency issues. This tutorial tells the story of these challenges in the form of several definitional interpreters and abstract machines, all written in Scheme.

We create definitional interpreters for the four variants of the gradually-typed lambda calculus that arise from the choices between eager and lazy cast checking [Herman et al. 2007] and between the D and UD blame tracking strategies [Siek et al. 2009]. The interpreter for the lazy UD variant (Section 3.2) is a straightforward adaptation of reduction semantics of Wadler and Findler [2009]. The interpreter for lazy D (Section 3.1) differs from the reduction semantics for lazy D of Siek et al. [2009] in that it does not rely on the Coercion Calculus [Henglein 1994] but expresses casts simply using source and target types. The eager interpreters are interesting in that we are, in some sense, forced to use the Coercion Calculus to get the correct blame assignment (Section 6). However, unlike the earlier work of Siek et al. [2009], we define a simple and efficient normalizer for coercions (Figures 5 and 13) that is inspired by threesomes [Siek and Wadler 2010] and more recent work on eager threesomes [Garcia 2012].

Turning to the efficiency issues, we present space-efficient abstract machines for all four variants (Section 4). Lastly, we improve the abstract machines by reducing function call overhead (Section 5). All of the code presented in this tutorial is tested with Petite Chez Scheme Version 8.4 from Cadence Research Systems.

The following grammar defines the syntax for the gradually-typed lambda calculus. We write the dynamic type as dyn. The type annotation on the parameter of a lambda expression is optional and defaults to dyn. The expression $(e : T\,\ell)$ is an explicit cast.

variables x integers n blame labels ℓ

basic types	B	$::=$	$\mathsf{int} \mid \mathsf{bool}$
types	T	$::=$	$B \mid (\rightarrow T\; T) \mid \mathsf{dyn}$
constants	k	$::=$	$n \mid \#\mathsf{t} \mid \#\mathsf{f}$
operators	op	$::=$	$\mathsf{inc} \mid \mathsf{dec} \mid \mathsf{zero?}$
expressions	e	$::=$	$k \mid (op\, e\, \ell) \mid (\mathsf{if}\, e\, e\, e\, \ell) \mid x \mid (e\, e\, \ell) \mid$ $(\lambda(x)\, e) \mid (\lambda(x{:}T)\, e) \mid (e : T\,\ell)$

Many of the syntactic forms include blame labels, which we treat as symbols, but a production-quality implementation would use the source location (line and character position) as the blame label.

2. The Gradual Type System and Cast Insertion

The type system of the gradually-typed lambda calculus is similar to the simply-typed lambda calculus, with the main differences in the addition of the type dyn and in the rules for function application and conditional expressions. In function application, instead of requiring the argument's type to be identical to the function's parameter type, we only require that the types be *consistent*, often written $\sim$, and defined by the following procedure[1].

```
(define consistent?
  (lambda (T1 T2)
    (match `(,T1 ,T2)
      [(,T1 dyn) #t] [(dyn ,T2) #t]
      [(int int) #t]  [(bool bool) #t]
      [((→ ,T11 ,T12) (→ ,T21 ,T22))
       (and (consistent? T11 T21) (consistent? T12 T22))]
      [,other #f])))
```

[1] We make use of a **match** macro that implements pattern matching, developed by Friedman, Hilsdale, Ganz, and Dybvig. The implementation of *match* is included with the auxiliary materials for this paper.

Scheme and Functional Programming 2012 September, Copenhagen Denmark.
Copyright © 2012 ACM 978-1-4503-1895-2. . . $15.00

The lack of contravariance in how function parameters are handled in the consistency relation not a mistake. Unlike subtyping, the consistency relation is *symmetric*, so it would not matter if we wrote $T_{21} \sim T_{11}$ instead of $T_{11} \sim T_{21}$. Also, unlike subtyping, consistency is not transitive.

For conditionals we need to give a type to the entire expression that is consistent with the type of each branch. We choose to use the most specific type that is consistent with both branches, that is, we compute the meet of the two types, defined below.

```
(define meet
  (lambda (T1 T2)
    (match `(,T1 ,T2)
      [(,T1 dyn) T1] [(dyn ,T2) T2]
      [(int int) 'int]  [(bool bool) 'bool]
      [((→ ,T11 ,T12) (→ ,T21 ,T22))
       `(→ ,(meet T11 T21) ,(meet T12 T22))]
      [,other (error 'meet "types are not consistent")])))
```

In mathematics, a meet operator computes the greatest lower bound with respect to some ordering relation. In this case, the ordering relation is the "less or equally dynamic" relation, known as naive subtyping in the literature [Wadler and Findler 2009]. The meet operator returns a type exactly when the input types are consistent.

Next we begin to consider the dynamic semantics of the gradually-typed lambda calculus. However, the dynamic semantics is not defined in terms of the surface syntax, but instead in terms of an intermediate language that makes casts explicit. We also change the syntax of primitive application and function application to make them more explicit.

$$\begin{array}{lrcl} \text{expressions} & e & ::= & \ldots \mid (\mathsf{cast}\,\ell\, e : T \Rightarrow T) \mid \\ & & & (\mathsf{prim}\; op\; e) \mid (\mathsf{call}\; e\; e) \end{array}$$

We use a non-standard notation for casts so that they are easier to read, so that they go left to right. The casts are annotated with a single blame label without any notion of polarity because these casts are just casts, not contracts between two parties [Findler and Felleisen 2002]. Gronski and Flanagan [2007] show how to implement contracts in terms of casts.

Cast insertion is a type-directed translation, so it is convenient to define both the type system and cast insertion as just one procedure, here named typecheck and shown in Figure 1. This procedure takes as input an environment (an association list mapping variables to their types) and an expression. The typecheck procedure returns the translation of the expression and its type. The main idea of this translation is that whenever the type system uses the consistency relation, the translation inserts a cast if the source and target types are not equal. In particular, the two cases for function application apply a cast to either the function expression or the argument, depending on whether the function has type dyn or $(\rightarrow T_{11}\, T_{12})$.

The typecheck procedure uses several auxiliary procedures which we describe next. The constant? and operator? functions identify literals and primitive operators.

```
(define constant? (lambda (k) (or (integer? k) (boolean? k))))
(define operator? (lambda (op) (memq op '(inc dec zero?))))
```

The typeof function computes the type of a constant or operator.

```
(define typeof
  (lambda (k)
    (match k
      [,n (guard (integer? n)) 'int]
      [,b (guard (boolean? b)) 'bool]
      [inc '(→ int int)]
      [dec '(→ int int)]
      [zero? '(→ int bool)])))
```

The mk-cast function takes as input an expression, a source type, and a target type. If the source and target type are the same, then mk-cast returns the input expression. Otherwise mk-cast wraps a run-time cast around the expression.

```
(define mk-cast
  (lambda (label e T1 T2)
    (cond [(equal? T1 T2) e]
          [else `(cast ,label ,e : ,T1 ⇒ ,T2)])))
```

Before moving on, we apply typecheck to an example program. For convenience, the following macro encodes a let-expression using a λ definition and application in the standard way.

```
(define-syntax letT
  (syntax-rules () [(letT [x : T label = e] b)
                    `((λ (x : T) ,b) e label)]))
```

In the following example, we cast a function to the dynamic type and then to a type that is inconsistent with the type of the function.

```
(define eg1
  (letT [f0 : dyn 0 = (λ (x : int) (inc x 2))]
    (letT [f1 : (→ bool bool) 1 = f0]
      `(f1 #t 3))))
```

The result of typecheck is shown below. Two casts are inserted: the cast labeled 0 coerces the lambda function from $(\rightarrow \mathsf{int}\,\mathsf{int})$ to dyn and the cast labeled 1 coerces f0 from dyn to $(\rightarrow \mathsf{bool}\,\mathsf{bool})$.

```
'((call (λ (f0 : dyn)
          (call (λ (f1 : (→ bool bool)) (call f1 #t))
                (cast 1 f0 : dyn ⇒ (→ bool bool))))
        (cast 0 (λ (x : int) (prim inc x))
              : (→ int int) ⇒ dyn))
  bool)
```

3. Design Choices Regarding the Dynamics

In this section we look at several design choices regarding the dynamic semantics of casts. Consider what should happen when we run the eg1 example.

- Should a runtime cast error occur when one of the casts is evaluated? Or should an error occur when f1 is applied to #t?
- If a runtime cast error occurs, which cast should be blamed, 0 or 1? More generally, we want to provide programmers with a simple set of rules for when casts are safe (never fail). We define a subtyping relation that does just this, with different rules corresponding to different blame tracking strategies.

Siek et al. [2009] characterize the different answers to the above questions in terms of the Coercion Calculus [Henglein 1994]. One can choose to check higher-order casts in either a *lazy* or *eager* fashion [Herman et al. 2007] and one can assign blame to only downcasts (D) or the one can share blame between upcasts and downcasts (UD). The semantics of casts with lazy checking is straightforward whereas eager checking is not, so we first discuss lazy checking. Also, the semantics of the D approach is slightly simpler than UD, so we start with Lazy D. We delay discussing the Coercion Calculus until we really need it.

3.1 The Lazy D Semantics

This subsection culminates with a definition of procedure eval-ld that maps a program to an observable for the Lazy D variant. The eval-ld procedure is defined in terms of a recursive interp-ld procedure that maps an expression and environment to a result. The

```
(define typecheck
  (lambda (env e)
    (match e
      [,k (guard (constant? k)) ‘(,k ,(typeof k))]
      [(,op ,e ,label) (guard (operator? op))
       (match ‘(,(typecheck env e) ,(typeof op))
         [((,new-e ,T) (→ ,T1 ,T2)) (guard (consistent? T T1))
          ‘((prim ,op ,(mk-cast label new-e T T1)) ,T2)]
         [,other (error ’typecheck "primitive operator")])]
      [(if ,cnd ,thn ,els ,label)
       (match ‘(,(typecheck env cnd) ,(typecheck env thn) ,(typecheck env els))
         [((,new-cnd ,cnd-T) (,new-thn ,thn-T) (,new-els ,els-T))
          (cond [(and (consistent? cnd-T ’bool) (consistent? thn-T els-T))
                 (let ([if-T (meet thn-T els-T)])
                   ‘((if ,(mk-cast label new-cnd cnd-T ’bool) ,(mk-cast label new-thn thn-T if-T)
                         ,(mk-cast label new-thn els-T if-T)) ,if-T))]
                [else (error ’typecheck "ill-typed if expression")])])]
      [,x (guard (symbol? x)) ‘(,x ,(cdr (assq x env)))]
      [(λ (,x) ,e) (typecheck env ‘(λ (,x : dyn) ,e))]
      [(λ (,x : ,T1) ,e)
       (match (typecheck ‘((,x . ,T1) . ,env) e)
         [(,new-e ,ret-T) ‘((λ (,x : ,T1) ,new-e) (→ ,T1 ,ret-T))])]
      [(,e : ,T ,label)
       (match (typecheck env e)
         [(,new-e ,e-T)
          (cond [(consistent? e-T T) ‘(,(mk-cast label new-e e-T T) ,T)]
                [else (error ’typecheck "cast between inconsistent types")])])]
      [(,e1 ,e2 ,label)
       (match ‘(,(typecheck env e2) ,(typecheck env e1))
         [((,new-e2 ,T2) (,new-e1 dyn))
          ‘((call ,(mk-cast label new-e1 ‘dyn ‘(→ ,T2 dyn)) ,new-e2) dyn)]
         [((,new-e2 ,T2) (,new-e1 (→ ,T11 ,T12)))
          (cond [(consistent? T2 T11) ‘((call ,new-e1 ,(mk-cast label new-e2 T2 T11)) ,T12)]
                [else (error ’typecheck "arg/param mismatch")])]
         [((,new-e2 ,T2) (,new-e1 ,other-T))
          (error ’typecheck "call to non-function")])])))
```

Figure 1. Type checking and cast insertion

values, results, and observables are defined as follows:

procedures	F	$\in$	$V \to R$
injectables	I	$::=$	$B \mid T \to T$
values	$v \in V$	$::=$	$k \mid F \mid (\mathsf{cast}\,\ell\, v : I \Rightarrow \mathsf{dyn}) \mid$
			$(\mathsf{cast}\,\ell\, v : (\to T_1\, T_2) \Rightarrow (\to T_3\, T_4))$
results	$r \in R$	$::=$	$v \mid (\mathsf{blame}\,\ell)$
observables	o	$::=$	$k \mid \mathsf{function} \mid \mathsf{dynamic} \mid (\mathsf{blame}\,\ell)$

The definition of interp-ld relies on one macro and several auxiliary functions, which we discuss next.

To handle the short-circuiting of evaluation on a cast error, signaled by $(\mathbf{blame}\,\ell)$, we use the following monadic let-expression:

```
(define-syntax letB
  (syntax-rules ()
    [(letB [x e1] e2)
     (match e1
       [(blame ,label) ‘(blame ,label)]
       [,v (let ((x v)) e2)])]))
```

The primitive operators are given meaning by the delta function.

```
(define delta
  (lambda (op v)
    (match op
      [inc (+ 1 v)] [dec (− 1 v)] [zero? (= 0 v)])))
```

In lazy cast checking, when determining whether to report a cast error, we only compare for shallow consistency:

```
(define shallow-consistent?
  (lambda (T1 T2)
    (match ‘(,T1 ,T2)
      [(,T1 dyn) #t] [(dyn ,T2) #t]
      [(int int) #t] [(bool bool) #t]
      [((→ ,T11 ,T12) (→ ,T21 ,T22)) #t]
      [,other #f])))
```

The next auxiliary function, named apply-cast-ld and shown in Figure 2, is the main event. It performs a run-time cast on a value. If the heads of the source and target type are inconsistent, then this function returns $(\mathsf{blame}\, l_1)$. If the source type T_1 is dyn, then we cast the underlying value to the target type, using the blame label l_1 associated with the down-cast from dyn to T_2. This gives the semantics its "D" flavor. For the other cases, including when v is a function, we wrap a new cast around the value v. Alternatively, we could create a new function instead of wrapping a cast around the function. (We discuss this alternative in more depth in Section 5) However, the cast-wrapping approach enables the compression of casts in the abstract machines of Section 4.

The apply-lazy procedure performs function application. The parameter F may simply be a procedure, or as just described, it may

```
(define apply-cast-ld
  (lambda (label1 v T1 T2)
    (cond [(shallow-consistent? T1 T2)
           (match T1
             [dyn (match v
                    [(cast ,label2 ,v2 : ,T3 ⇒ dyn)
                     (apply-cast-ld label1 v2 T3 T2)])]
             [,other (mk-cast label1 v T1 T2)])]
          [else ‘(blame ,label1)])))
```

Figure 2. Apply a cast to a value using the Lazy D approach.

be a procedure wrapped in one or more casts. Thus, apply-lazy is defined by recursion on the first parameter.

```
(define apply-lazy
  (lambda (apply-cast)
    (lambda (F v)
      (let ([recur (apply-lazy apply-cast)])
        (match F
          [(cast ,label ,F1 : (→ ,T1 ,T2) ⇒ (→ ,T3 ,T4))
           (letB [x3 (apply-cast label v T3 T1)]
           (letB [x4 (recur F1 x3)]
              (apply-cast label x4 T2 T4)))]
          [,proc (guard (procedure? proc)) (proc v)])))))
```

With these auxiliary procedures and the monadic-let in hand, the definition of the interpreter is straightforward. We define a generic interp procedure that is parameterized over the behavior of function application and casting so that we can reuse it for Lazy UD and the eager calculi. The definition of interp and its instantiation for Lazy D is given in Figure 3.

The semantics for the Lazy D Gradually-Typed Lambda Calculus is defined by the eval-ld procedure, also defined in Figure 3. The eval-ld procedure is defined in terms of the generic eval together with interp-ld and the procedure observe-lazy which turns a result into an observable as follows.

```
(define observe-lazy
  (lambda (r)
    (match r
      [,k (guard (constant? k)) k]
      [,proc (guard (procedure? proc)) 'function]
      [(cast ,label ,v : ,T1 ⇒ (→ ,T3 ,T4)) 'function]
      [(cast ,label ,v : ,T ⇒ dyn) 'dynamic]
      [(blame ,label) ‘(blame ,label)])))
```

Similar to object-oriented languages, we can define a subtyping relation that characterizes when a cast is safe, that is, when it will never fail. The following defines subtyping for the D semantics.

```
(define subtype
  (lambda (T1 T2)
    (match ‘(,T1 ,T2)
      [(,T1 dyn) #t]
      [(int int) #t] [(bool bool) #t]
      [((→ ,T1 ,T2) (→ ,T3 ,T4))
       (and (subtype T3 T1) (subtype T2 T4))]
      [,other #f])))
```

This subtyping relation is what one would expect to see. The dynamic type plays the role of the top element of this ordering and the rule for function types has the usual contravariance in the parameter type. The following Subtyping Conjecture connects the dynamic semantics of blame tracking with the subtyping relation.

Conjecture 1 (Subtyping, Lazy D)**.** *If the unique cast labeled with ℓ in program e respects subtyping, then eval-ld$(e) \neq \mathbf{blame}\,\ell$.*

```
(define interp
  (lambda (cast apply)
    (lambda (env e)
      (let ([recur (interp cast apply)])
        (match e
          [,k (guard (constant? k)) k]
          [(prim ,op ,e) (letB [v (recur env e)] (delta op v))]
          [(if ,cnd ,thn ,els)
            (letB [v (recur env cnd)]
              (cond [v (recur env thn)]
                    [else (recur env els)]))]
          [,x (guard (symbol? x)) (cdr (assq x env))]
          [(call ,e1 ,e2)
            (letB [x1 (recur env e1)] (letB [x2 (recur env e2)]
            (apply x1 x2)))]
          [(λ (,x : ,T1) ,e)
            (lambda (v) (recur ‘((,x . ,v) . ,env) e))]
          [(cast ,label ,e : ,T1 ⇒ ,T2)
            (letB [x (recur env e)] (cast label x T1 T2))])))))

(define interp-ld (interp apply-cast-ld (apply-lazy apply-cast-ld)))

(define eval
  (lambda (interp observe)
    (lambda (e)
      (match (typecheck '() e)
        [(,new-e ,T) (observe (interp '() new-e))]))))

(define eval-ld (eval interp-ld observe-lazy))
```

Figure 3. A generic interpreter and instantiation for Lazy D.

The output of eval-ld on eg1 is (blame 0), so the blame went to the downcast from dyn to (→ bool bool), as expected. The result of eval-ld should always be the same as that of the Lazy D reduction semantics of Siek et al. [2009].

Conjecture 2 (Agreement with Siek et al. [2009])**.** *For any well-typed program e, eval-ld$(e) = o$ if and only if $\langle\!\langle e\rangle\!\rangle \longmapsto^*_{\mathbf{LUD}} r$ and $observe(r) = o$.*

3.2 The Lazy UD Semantics

One interpretation of the dynamic type dyn is to view it as the following recursive type:

$$\mathsf{dyn} \equiv \mu\, d.\, \mathsf{int} + \mathsf{bool} + (\rightarrow d\, d)$$

(See, for example, the chapter on Dynamic Typing in *Practical Foundations for Programming Languages* [Harper 2012].) In such an interpretation, one can directly convert from (→ dyn dyn) to dyn, but not, for example, from (→ int int) to dyn. Instead, one must first convert from (→ int int) to (→ dyn dyn), then to dyn. The UD blame tracking strategy takes this interpretation to heart and changes the definition of injectable types, restricting the function types to just (→ dyn dyn). This in turn affects the definition of values for Lazy UD.

injectables	I	$::=$	$B \mid (\rightarrow \mathsf{dyn}\,\mathsf{dyn})$
values	$v \in V$	$::=$	$k \mid F \mid$
			$(\mathsf{cast}\,\ell\, v : I \Rightarrow \mathsf{dyn}) \mid$
			$(\mathsf{cast}\,\ell\, v : T_1 \rightarrow T_2 \Rightarrow T_3 \rightarrow T_4)$

This change necessitates some changes in the semantics of casts. In the cast application function, shown in Figure 4, casts from function types to dyn are expanded into casts that go through (→ dyn dyn).

```
(define apply-cast-lud
  (lambda (label1 v T1 T2)
    (cond [(shallow-consistent? T1 T2)
           (match `(,T1 ,T2)
            [(dyn ,T2)
             (match v
              [(cast ,label2 ,v2 : ,T3 ⇒ dyn)
               (apply-cast-lud label1 v2 T3 T2)])]
            [((→ ,T11 ,T12) dyn)
             (guard (not (eq? T11 'dyn)) (not (eq? T12 'dyn)))
             `(cast ,label1 (cast ,label1 ,v
                                  : (→ ,T11 ,T12) ⇒ (→ dyn dyn))
                : (→ dyn dyn) ⇒ dyn)]
            [,other (mk-cast label1 v T1 T2)])]
          [else `(blame ,label1)])))
```

Figure 4. Apply a cast to a value in the Lazy UD approach.

The definitions of interp-lud and eval-lud are otherwise the same as those for Lazy D.

```
(define apply-lud (apply-lazy apply-cast-lud))
(define interp-lud (interp apply-cast-lud apply-lud))
(define eval-lud (eval interp-lud observe-lazy))
```

The following procedure defines the subtyping relation for Lazy UD. With this subtyping relation, the type dyn does not play the role of the top element. Instead, a type T is a subtype of dyn if it is a subtype of some injectable type I.

```
(define subtype-ud
  (lambda (T1 T2)
    (match `(,T1 ,T2)
      [(dyn dyn) #t] [(int int) #t] [(bool bool) #t]
      [(int dyn) #t] [(bool dyn) #t]
      [((→ ,T1 ,T2) dyn)
       (subtype-ud `(→ ,T1 ,T2) `(→ dyn dyn))]
      [((→ ,T1 ,T2) (→ ,T3 ,T4))
       (and (subtype-ud T3 T1) (subtype-ud T2 T4))]
      [,other #f])))
```

Conjecture 3 (Subtyping, Lazy UD). *If the cast labeled with ℓ in program e respects subtyping, then eval-lud$(e) \neq$* $\mathbf{blame}\,\ell$.

The output of eval-lud on eg1 is (blame g1), so the blame went to the upcast from (→ int int) to dyn, as expected for Lazy UD.

Conjecture 4 (Agreement with Siek et al. [2009]). *For any well-typed program e, eval-lud$(e) = o$ if and only if $\langle\!\langle e\rangle\!\rangle \longmapsto^*_{\mathbf{LUD}} r$ and $observe(r) = o$.*

The next logical step would be to present the definitional interpreters for the eager variants of the gradually-typed lambda calculus. However, we instead defer the treatment of eager cast checking to Section 6, by which point we will have introduced the necessary machinery (the Coercion Calculus). Next we look at space-efficient implementations of the lazy calculi.

4. Space-Efficient Abstract Machines

The introduction briefly mentioned that there are efficiency challenges regarding gradual typing. One of those challenges regards space efficiency. Herman et al. [2007] observe two circumstances in which function casts can lead to unbounded space consumption. First, some programs repeatedly apply casts to the same function, resulting in a build-up of casts around the function. In the following example, each time the function bound to k is passed between even? and odd? a cast is added, causing a leak proportional to n.

```
(letrec ([even? (λ (n : int) (λ (k : (→ dyn bool))
                  (if (zero? n)
                      (k #t)
                      ((odd? (dec n)) k))))]
         [odd? (λ (n : int) (λ (k : (→ bool bool))
                 (if (zero? n)
                     (k #f)
                     ((even? (dec n)) k))))])
  ((even? 88) (λ (x : bool) x)))
```

Second, some casts break proper tail recursion. Consider the following example in which the return type of even? is dyn and the return type of odd is bool.

```
(letrec ([even? (λ (n : int)
                  (if (zero? n)
                      (cast 2 #t : bool ⇒ dyn)
                      (odd? (dec n))))]
         [odd? (λ (n : int)
                 (if (zero? n)
                     #f
                     (cast 3 (even? (dec n)) : dyn ⇒ bool)))])
  (even? 88))
```

Assuming tail call optimization, cast-free versions of these even? and odd? functions require only constant space, but because the call to even? is not a tail call, the run-time stack grows with each call and space consumption is proportional to n.

4.1 The Coercion Calculus

The solution of Herman et al. [2007] to this space-efficiency problem relies on the Coercion Calculus [Henglein 1994]. This calculus defines a set of combinators, called *coercions*, that can be used to express casts. The key advantage of coercions is that an arbitrarily long sequence of coercions reduces to a sequence of at most three coercions. The following defines the syntax of coercions. We use the semi-colon ; for left-to-right composition, instead of the right-to-left ∘ operator, to be consistent with the syntax of casts.

$$\text{coercions} \quad c \quad ::= \quad \iota \mid I! \mid I?^{\ell} \mid c \to c \mid c; c \mid \mathit{fail}^{\ell}$$

In the Scheme code we use the following s-expression representation for coercions, which includes a few type annotations to ease the implementation of the typeof-coercion function, given below.

$$\begin{array}{lll}\text{coercions} \quad c & ::= & (\mathit{id}\,T) \mid (\mathit{inj}\,I) \mid (\mathit{proj}\,I\,\ell) \mid (\to c\,c) \mid \\ & & (\mathit{seq}\,c\,c) \mid (\mathit{fail}\,\ell\,T\,T)\end{array}$$

The two most important coercions are the injection coercion $I!$ from I to dyn and the projection coercion $I?^{\ell}$ from dyn to I. Recall that the definition of I depends on the blame tracking strategy:

$$\begin{array}{ll} I ::= B \mid (\to \text{dyn dyn}) & (UD) \\ I ::= B \mid (\to T\,T) & (D) \end{array}$$

The procedure typeof-coercion maps a coercion to its source and target type.

```
(define typeof-coercion
  (lambda (c)
    (match c
      [(id ,T) `(,T ,T)] [(inj ,I) `(,I dyn)] [(proj ,I ,label) `(dyn ,I)]
      [(→ ,c1 ,c2)
       (match `(,(typeof-coercion c1) ,(typeof-coercion c2)_)
         [((,T21 ,T11) (,T12 ,T22))
          `((→ ,T11 ,T12) (→ ,T21 ,T22))])]
      [(seq ,c1 ,c2)
       (match `(,(typeof-coercion c1) ,(typeof-coercion c2))
         [((,T1 ,T2) (,T2 ,T3)) `(,T1 ,T3)])]
      [(fail ,label ,T1 ,T2) `(,T1 ,T2)])))
```

The way in which casts are translated into coercions depends on whether we are using the D or UD blame tracking strategy. Here is the translation for Lazy D.

```
(define mk-arrow-lazy (lambda (c1 c2) `(→ ,c1 ,c2)))
(define mk-coerce-d
  (lambda (mk-arrow)
    (lambda (T1 T2 label)
      (let ([comp (mk-coerce-d mk-arrow)])
        (match `(,T1 ,T2)
          [(dyn dyn) `(id dyn)]
          [(bool bool) `(id bool)] [(int int) `(id int)]
          [(dyn ,I) `(proj ,I ,label)]
          [(,I dyn) `(inj ,I)]
          [((→ ,T1 ,T2) (→ ,T3 ,T4))
           (mk-arrow (comp T3 T1 label) (comp T2 T4 label))]
          [,other `(fail ,label ,T1 ,T2)])))))
(define mk-coerce-ld (mk-coerce-d mk-arrow-lazy))
```

The translation function for UD is more complicated because the definition of injectable type is more restrictive.

```
(define mk-coerce-ud
  (lambda (mk-arrow)
    (lambda (T1 T2 label)
      (let ([comp (mk-coerce-ud mk-arrow)])
        (match `(,T1 ,T2)
          [(dyn dyn) `(id dyn)]
          [(bool bool) `(id bool)] [(int int) `(id int)]
          [(dyn int) `(proj int ,label)]
          [(dyn bool) `(proj bool ,label)]
          [(dyn (→ ,T3 ,T4))
           `(seq (proj (→ dyn dyn) ,label)
                (→ ,(comp T3 'dyn label) ,(comp 'dyn T4 label)))]
          [(int dyn) `(inj int)] [(bool dyn) `(inj bool)]
          [((→ ,T1 ,T2) dyn)
           `(seq (→ ,(comp 'dyn T1 label) ,(comp T2 'dyn label))
                (inj (→ dyn dyn)))]
          [((→ ,T1 ,T2) (→ ,T3 ,T4))
           (mk-arrow (comp T3 T1 label) (comp T2 T4 label))]
          [,other `(fail ,label ,T1 ,T2)])))))
(define mk-coerce-lud (mk-coerce-ud mk-arrow-lazy))
```

In the Coercion Calculus, coercions are considered equal up to associativity:

$$c_1; (c_2; c_3) = (c_1; c_2); c_3$$

The following are the reduction rules for lazy coercions.

$$\begin{aligned}
\iota; c &\longrightarrow_{\mathbf{L}} c \\
c; \iota &\longrightarrow_{\mathbf{L}} c \\
I_1!; I_2?^{\ell} &\longrightarrow_{\mathbf{D}} \textsf{mk-coerce-d}(I_1, I_2, \ell) \\
I_1!; I_2?^{\ell} &\longrightarrow_{\mathbf{UD}} \textsf{mk-coerce-ud}(I_1, I_2, \ell) \\
(c_{11} \to c_{12}); (c_{21} \to c_{22}) &\longrightarrow_{\mathbf{L}} (c_{21}; c_{11}) \to (c_{12}; c_{22}) \\
\mathit{fail}^{\ell}; c &\longrightarrow_{\mathbf{L}} \mathit{fail}^{\ell} \\
I!; \mathit{fail}^{\ell} &\longrightarrow_{\mathbf{L}} \mathit{fail}^{\ell} \\
(c_{11} \to c_{12}); \mathit{fail}^{\ell} &\longrightarrow_{\mathbf{L}} \mathit{fail}^{\ell}
\end{aligned}$$

The grammar below characterizes the coercions in normal form $\hat{c}$.

$$\begin{array}{llll}
\text{wrapper coercions} & \overline{c} & ::= & I! \mid \hat{c} \to \hat{c} \mid (\hat{c} \to \hat{c}); I! \\
\text{normal coercions} & \hat{c} & ::= & \overline{c} \mid \iota \mid \mathit{fail}^{\ell} \mid \\
 & & & I?^{\ell} \mid I?^{\ell}; \mathit{fail}^{\ell} \mid I?^{\ell}; I! \mid \\
 & & & I?^{\ell}; (\hat{c} \to \hat{c}) \mid I?^{\ell}; (\hat{c} \to \hat{c}); I!
\end{array}$$

```
(define seq-lazy
  (lambda (mk-coerce)
    (lambda (c1 c2)
      (let ([recur (seq-lazy mk-coerce)])
        (match `(,c1 ,c2)
          [((id ,T) ,c2) c2]
          [(,c1 (id ,T)) c1]
          [((inj ,I1) (proj ,I2 ,label)) (mk-coerce I1 I2 label)]
          [((→ ,c11 ,c12) (→ ,c21 ,c22))
           `(→ ,(recur c21 c11) ,(recur c12 c22))]
          [((fail ,label ,T1 ,T2) ,c2) `(fail ,label ,T1 ,T2)]
          [((inj ,I) (fail ,label ,T1 ,T2)) `(fail ,label ,T1 ,T2)]
          [((→ ,c11 ,c12) (fail ,label ,T1 ,T2))
           `(fail ,label ,T1 ,T2)]
          [((seq ,c11 ,c12) ,c2) (recur c11 (recur c12 c2))]
          [((proj ,I ,label) (seq (→ ,c21 ,c22) ,c23))
           `(seq (proj ,I ,label) (seq (→ ,c21 ,c22) ,c23))]
          [(,c1 (seq ,c21 ,c22)) (recur (recur c1 c21) c22)]
          [(,c1 ,c2) `(seq ,c1 ,c2)])))))
```

Figure 5. The composition operator for lazy coercions.

The wrapper coercions are the coercions that may appear around a value. From the above grammar, it is clear that coercions in normal form have a length of at most three.

Coercion reduction also has the nice property that reduction always terminates.

Theorem 5 (Strong Normalization for Lazy Coercions)**.** *There is a $\hat{c}$ such that $c \longmapsto_X^* \hat{c}$ for $X = \mathbf{L} \cup \mathbf{UD}$ or $X = \mathbf{L} \cup \mathbf{D}$.*

While it is nice to know what coercions always reduce to a normal form, how can we implement a procedure that efficiently performs this reduction? Siek and Wadler [2010] discover an alternative to coercions, called threesomes, and defined an efficient composition operator on threesomes. [Garcia 2012] develops a variant of threesomes for eager cast checking. In this paper we instead develop a composition operator that works directly on coercions.

The procedure seq-lazy, defined in Figure 5, takes two coercions in normal form, c_1 and c_2, and computes the result $\hat{c_3}$ of sequencing them and reducing to normal form according to the lazy cast checking semantics: $c_1; c_2 \longrightarrow^* \hat{c_3}$. The procedure presented here is inspired by the composition operator of threesomes, but it is much simpler than what one would obtain from the one-to-one correspondence between threesomes and coercions. The author came up with this procedure while writing up these notes, as part of a desire to find the most simple and elegant procedure. The author is almost embarrassed that the result is so simple.

The seq-lazy procedure is parameterized by mk-coerce so that it can be used for both the D and UD variants. The first seven cases of seq-lazy are just a direct transcription of the above reduction rules. The last four cases concern sequences and deal with associativity. In our normal forms, we choose to associate sequences to the right, so (seq (seq c_{11} c_{12}) c_2) is not a normal form and we proceed by recursively normalizing c_{12} and c_2, then combining the result with c_{11}. The next case is to prevent a recursive call (via the following case) when we have a normal form of length three. The last case handles length-three coercions that are not yet in normal form because there is a redex in the combination $(c_1; c_{21})$.

Conjecture 6 (Correctness of seq-lazy)**.** *Given two well-typed coercions in normal form, c_1 and c_2, we have*

1. *$((\textsf{seq-lazy mk-coerce-ld})\ c_1 c_2) = \hat{c_3}$ and $(c_1; c_2) \longmapsto_{\mathbf{L} \cup \mathbf{D}}^* \hat{c_3}$.*
2. *$((\textsf{seq-lazy mk-coerce-lud}) c_1 c_2) = \hat{c_3}$ and $(c_1; c_2) \longmapsto_{\mathbf{L} \cup \mathbf{UD}}^* \hat{c_3}$.*

4.2 An Abstract Machine for Lazy Casts

With the Coercion Calculus in hand, we define a space-efficient abstract machine. This machine is a variant of the ECD machine [Siek 2012] that works on programs in administrative normal form (ANF) [Danvy 1991; Flanagan et al. 1993], given by the grammar below. Our choice of the ECD machine simplifies how we achieve space efficiency. Herman et al. [2007] take an alternative approach that relies on mutually-recursive evaluation contexts.

$$\begin{array}{llll}
\text{simple expressions} & \tilde{e} & ::= & k \mid x \mid (\lambda(x{:}T)\,\overline{s}) \\
\text{expressions} & e & ::= & \tilde{e} \mid (\mathsf{prim}\; op\, \tilde{e}) \\
\text{assign. statements} & s_a & ::= & (\mathsf{assign}\, x\, e) \mid (\mathsf{call}\, x\, e\, e) \mid \\
 & & & (\mathsf{cast}\, x\, e : \hat{c}) \mid (\mathsf{if}\, e\, \overline{s_a}\, \overline{s_a}) \\
\text{tail statements} & s_t & ::= & (\mathsf{return}\, e) \mid (\mathsf{tail\text{-}call}\, e\, e) \mid \\
 & & & (\mathsf{tail\text{-}call}\, e\, e : \hat{c}) \mid (\mathsf{if}\, e\, \overline{s}\, \overline{s}) \\
\text{seq. of assign. stmt.} & \overline{s_a} & ::= & (s_a) \mid (s_a \,.\, \overline{s_a}) \\
\text{block of statements} & \overline{s} & ::= & (s_t) \mid (s_a \,.\, \overline{s})
\end{array}$$

The main characteristic of ANF is that intermediate computations are assigned to temporary variables: the x's in the assign, call, and cast statements. The tail-call forms play a special role in making tail calls efficient. The first tail-call form is the usual one [Danvy 1992; Danvy and Filinski 1992] whereas the second form includes an outstanding cast. Both forms of tail call receive special treatment from the abstract machine.

Figure 6 defines the translation into ANF. In addition to the conversion to ANF, we translate casts into coercions. The procedure to-ANF has three parameters: an expression to be translated, a Boolean indicating whether the expression is in tail position, and a procedure for making coercions (e.g., mk-coerce-ud would be a suitable argument for this parameter). The result of to-ANF is either a block of statements (if in tail position) or a sequence of assignment statements and an expression (if not in tail position). (The sequence of assignment statements may be empty, unlike the sequences within the "then" and "else" branches of the if.)

The below evaluation function valueof, and its auxiliary simple-valueof, map an expression and environment to a value, similar to the interp function, but now expressions are no longer nested and there are fewer cases to handle here. The following is the definition of values for this machine.

$$\begin{array}{llll}
\text{uncasted values} & \tilde{v} & ::= & k \mid (\lambda(x{:}T)\,\overline{s})\,\rho) \\
\text{values} & v & ::= & \tilde{v} \mid (\mathsf{cast}\,\tilde{v} : \overline{c})
\end{array}$$

Function are handled differently compared to interp: here we build a closure as data whereas interp used a Scheme procedure.

```
(define simple-valueof
  (lambda (e env)
    (match e
      [,k (guard (constant? k)) k]
      [,x (guard (symbol? x)) (cdr (assq x env))]
      [(λ (,x : ,T1) ,s) `((λ (,x : ,T1) ,s) ,env)])))

(define valueof
  (lambda (e env)
    (match e
      [(prim ,op ,e) (delta op (simple-valueof e env))]
      [,other (simple-valueof e env)])))
```

With casts expressed in terms of coercions, we must replace the apply-cast procedures with a procedure for applying coercions to a value. In the following apply-coercion-lazy, if the value is already casted, we use seq-lazy to compose the old and new coercions, then call mk-cast-lazy, which looks at the resulting coercion. If it is an identity coercion, mk-cast-lazy returns the underlying value. If the coercion is a failure, then the result is blame. Otherwise, the value is wrapped in a cast. Looking back at apply-coercion-lazy, if the value is not already casted, then the value and new cast are passed to mk-cast-lazy.

```
(define run
  (lambda (e mk-coerce coerce seq-norm)
    (match (typecheck '() e)
     [(,casted-e ,T)
      (let loop ([s (to-ANF casted-e #t mk-coerce)]
                 [env '()]
                 [stack `(stack (id ,T) ())])
        (match (transition s env stack coerce seq-norm)
         [(state ,s ,env ,stack) (loop s env stack)]
         [,r (observe2 r)]))])))

(define observe2
  (lambda (r)
    (match r
      [,k (guard (constant? k)) k]
      [((λ (,x : ,T1) ,s) ,env) 'function]
      [(cast ,v : (→ ,c1 ,c2)) 'function]
      [(cast ,v : (seq (→ ,c1 ,c2) (inj ,l))) 'dynamic]
      [(cast ,v : (inj ,l)) 'dynamic]
      [(blame ,label) `(blame ,label)])))
```

Figure 8. The run and observe2 procedures.

```
(define mk-cast-lazy
  (lambda (v c)
    (match c
     [(id ,T) v]
     [(fail ,label ,T1 ,T2) `(blame ,label)]
     [,other `(cast ,v : ,c)])))

(define apply-coercion-lazy
  (lambda (mk-coerce)
    (lambda (v c)
      (match v
       [(cast ,v1 : ,c1)
        (mk-cast-lazy v1 ((seq-lazy mk-coerce) c1 c))]
       [,other (mk-cast-lazy v c)]))))

(define apply-coercion-ld (apply-coercion-lazy mk-coerce-ld))
(define apply-coercion-lud (apply-coercion-lazy mk-coerce-lud))
```

The machine state has the form $(\mathsf{state}\, s\, \rho\, \kappa)$, where the stack κ is essentially a list of frames. There is a pending coercion associated with every frame and with the empty stack. We define stacks so that there is a uniform way to access the pending coercions.

$$\begin{array}{llll}
 & \sigma & ::= & () \mid (x\,\overline{s}\,\rho\,\kappa) \\
\text{stack} & \kappa & ::= & (\mathsf{stack}\, c\, \sigma)
\end{array}$$

The definition of the transition procedure is given in Figure 7. We parameterize this procedure over the method of applying a coercion to a value (coerce) and the method of normalizing coercions (seq-norm) so that it may be reused for all four variants of the semantics. The run procedure, shown in Figure 8, iterates the transitions until a result is produced (a value or blame).

The space-efficiency of this machine comes from two places. First, the values only ever include a single cast wrapped around a simple value. The apply-coercion-lazy function maintains this invariant by normalizing whenever a cast is applied to an already-casted value. The second place is the transition rule for tail calls. The coercion in tail position c_1 is sequenced with the pending coercion c_2 and normalized to become the new pending coercion.

```
(define to-ANF
  (lambda (e tail? mk-coerce)
    (match e
     [,k (guard (constant? k)) (cond [tail? `((return ,k))] [else `(() ,k)])]
     [(prim ,op ,e)
      (let ([ss-e (to-ANF e #f mk-coerce)])
        (cond [tail? (append (car ss-e) `((return (prim ,op ,(cadr ss-e)))))]
              [else `(,(car ss-e) (prim ,op ,(cadr ss-e)))]))]
     [(if ,cnd ,thn ,els)
      (cond [tail? (match `(,(to-ANF cnd #f mk-coerce) ,(to-ANF thn #t mk-coerce) ,(to-ANF els #t mk-coerce))
                     [((,cnd-ss ,new-cnd) ,thn-ss ,els-ss) (append cnd-ss `((if ,new-cnd ,thn-ss ,els-ss)))])]
            [else (match `(,(to-ANF cnd #f mk-coerce) ,(to-ANF thn #f mk-coerce) ,(to-ANF els #f mk-coerce))
                    [((,cnd-ss ,new-cnd) (,thn-ss ,new-thn) (,els-ss ,new-els))
                     (let ([tmp (gensym "t")])
                       `(,(append cnd-ss `((if ,new-cnd ,(append thn-ss `((assign ,tmp ,new-thn)))
                                                        ,(append els-ss `((assign ,tmp ,new-els)))))) ,tmp))])])]
     [,x (guard (symbol? x)) (cond [tail? `((return ,x))] [else `(() ,x)])]
     [(λ (,x : ,T1) ,e)
      (let ([f `(λ (,x : ,T1) ,(to-ANF e #t mk-coerce))])
        (cond [tail? `((return ,f))] [else `(() ,f)]))]
     [(call ,e1 ,e2)
      (let ([tmp (gensym "t")] [ss1-e1 (to-ANF e1 #f mk-coerce)] [ss2-e2 (to-ANF e2 #f mk-coerce)])
        (cond [tail? (append (append (car ss1-e1) (car ss2-e2)) `((tail-call ,(cadr ss1-e1) ,(cadr ss2-e2))))]
              [else `(,(append (append (car ss1-e1) (car ss2-e2)) `((call ,tmp ,(cadr ss1-e1) ,(cadr ss2-e2)))) ,tmp)]))]
     [(cast ,label (call ,e1 ,e2) : ,T1 ⇒ ,T2) (guard tail?)
      (let ([tmp (gensym "t")] [ss1-e1 (to-ANF e1 #f mk-coerce)] [ss2-e2 (to-ANF e2 #f mk-coerce)])
          (append (append (car ss1-e1) (car ss2-e2)) `((tail-call ,(cadr ss1-e1) ,(cadr ss2-e2) : ,(mk-coerce T1 T2 label)))))]
     [(cast ,label ,e : ,T1 ⇒ ,T2)
      (let∗ ([tmp (gensym "t")] [ss-e (to-ANF e #f mk-coerce)] [new-cast `(cast ,tmp ,(cadr ss-e) : ,(mk-coerce T1 T2 label))])
           (cond [tail? (append (car ss-e) `(,new-cast (return ,tmp)))]
                 [else `(,(append (car ss-e) `(,new-cast)) ,tmp)]))])))
```

Figure 6. The conversion to A-normal form, which handles intra-procedural order of evaluation.

With run complete, we have an implementation of Lazy D and UD that is space efficient and relatively efficient in time as well. However, there is a nagging issue regarding the speed of statically-typed code, which we address in the next section.

Conjecture 7 (Correctness of the abstract machine)**.**
Suppose e is a well-typed program. Let $(e'\, T) = (\mathit{typecheck}'\,()\, e)$ and $s = (\textit{to-ANF}\, e'\, \#t\ \textit{mk-coerce-X})$. Then we have

$$(\textit{run}\, s\ \textit{mk-coerce-X}\ \textit{seq-X}) = o \text{ if and only if } (\textit{eval-X}\, e) = o$$

for $X = \mathit{ld}$ and $X = \mathit{ud}$.

5. Efficient Function Application

Consider this statically-typed function: $(\lambda(f : (\rightarrow \mathsf{int}\,\mathsf{int}))\ (f\,42))$. We would like the execution speed of this function to be the same as if the entire language were statically typed. That is, we do not want statically-typed parts of a gradually-typed program to pay overhead because other parts may be dynamically typed. Or put another way, we want casted functions to pay their own way. Unfortunately, in the abstract machine defined in Section 4, there is some overhead in calling uncasted functions. At the point of a function call, such as the call $(f\,42)$ above, the machine needs to check whether f has evaluated to a closure or to a closure wrapped in a cast. This act of checking is the run-time overhead that we would like to remove.

Taking a step back, there are two approaches in the literature regarding how a cast is applied to a function. One approach is to build a new function that casts the argument, applies the old function, and then casts the result. Here is that reduction rule:

$$\begin{aligned}&(\mathsf{cast}\,v : (\rightarrow T_1\,T_2) \Rightarrow (\rightarrow T_3\,T_4))\\ \longrightarrow\ &(\lambda(x{:}T_3)(\mathsf{cast}\,(v\,(\mathsf{cast}\,x : T_3 \Rightarrow T_1)) : T_2 \Rightarrow T_4))\end{aligned}$$

The nice thing about this approach is that there is only one kind of value of function type, functions! So when it comes to function application, we only need one reduction rule, good old beta:

$$((\lambda(x : T)\,e)\,v) \longrightarrow [x{:=}v]e$$

The other approach is to leave the cast around the function and then add the following reduction rule for function application.

$$\begin{aligned}&((\mathsf{cast}\,v_1 : (\rightarrow T_1\,T_2) \Rightarrow (\rightarrow T_3\,T_4))\,v_2)\\ \longrightarrow\ &(\mathsf{cast}\,(v_1\,(\mathsf{cast}\,v_2 : T_3 \Rightarrow T_1)) : T_2 \Rightarrow T_4)\end{aligned}$$

The nice thing about this approach is that the cast around the function is easy to access, which we took advantage of to compress sequences of such casts. But as we have seen, the two kinds of values at function type induces some run-time overhead, even in parts of the program that are statically typed.

Our solution to this conundrum is to combine the best of these two approaches and to take advantage of the indirection that is already present when calling a closure, that is, the target function is determined by the function pointer inside the closure. Instead of having two kinds of values at function type, we have only one: a closure that includes an optional coercion: $((\lambda(x : T)\,\overline{s})\,\rho\,c)$. When a closure is created, the coercion is absent ($\#f$) which indicates that the closure has never been cast.

$$(\mathsf{valueof2}\,(\lambda(x : T_1) \rightarrow T_2\,\overline{s})\,\rho) = ((\lambda(x : T_1)\,\overline{s})\,\rho\,\#f)$$

```
(define transition
  (lambda (s env stack coerce seq-norm)
    (match s
      [((assign ,x ,e) . ,s2)
       (let ([v (valueof e env)])
         `(state ,s2 ((,x . ,v) . ,env) ,stack))]
      [((if ,cnd ,thn-ss ,els-ss) . ,s2)
       (let ([v (valueof cnd env)])
         (cond [v `(state ,(append thn-ss s2) ,env ,stack)]
               [else `(state ,(append else-ss s2) ,env ,stack)]))]
      [((call ,x ,e1 ,e2) . ,s2)
       (let ([v1 (valueof e1 env)] [v2 (valueof e2 env)])
         (match v1
           [((λ (,y : ,T1) ,s3) ,env2) `(state ,s3 ((,y . ,v2) . ,env2) (stack (id _) (,x ,s2 ,env ,stack)))]
           [(cast ((λ (,y : ,T1) ,s3) ,env2) : (→ ,c1 ,c2))
            (letB [v2c (coerce v2 c1)] `(state ,s3 ((,y . ,v2c) . ,env2) (stack ,c2 (,x ,s2 ,env ,stack))))]))]
      [((return ,e))
       (match stack
         [(stack ,c (,x ,s2 ,env2 ,stack2)) (letB [v (coerce (valueof e env) c)] `(state ,s2 ((,x . ,v) . ,env2) ,stack2))]
         [(stack ,c ()) (coerce (valueof e env) c)])]
      [((tail-call ,e1 ,e2)) (transition `((tail-call ,e1 ,e2 : (id _))) env stack coerce seq-norm)]
      [((tail-call ,e1 ,e2 : ,c1))
       (let ([v1 (valueof e1 env)] [v2 (valueof e2 env)])
         (match v1
           [((λ (,y : ,T1) ,s3) ,env2)
            (match stack [(stack ,c2 ,rest) `(state ,s3 ((,y . ,v2) . ,env2) (stack ,(seq-norm c1 c2) ,rest))])]
          [(cast ((λ (,y : ,T1) ,s3) ,env2) : (→ ,c2 ,c3))
            (match stack
              [(stack ,c4 ,rest)
               (let ([c314 (seq-norm c3 (seq-norm c1 c4))])
                 (letB [v2c (coerce v2 c2)] `(state ,s3 ((,y . ,v2c) . ,env2) (stack ,c314 ,rest))))])]))]
      [((cast ,x ,e : ,c) . ,s2)
        (letB [v (coerce (valueof e env) c)] `(state ,s2 ((,x . ,v) . ,env) ,stack))])))
```

Figure 7. The transition function for the abstract machine.

Later, when a closure is cast, the coercion is replaced. The transition rules for calls and tail calls passes the coercion as a special parameter, named $c_{\_\_}$, to the function.

When we apply a coercion to a closure for the first time, we build a wrapper function but using the special variable $c_{\_\_}$ to refer to the coercion instead of hard-coding the cast into the wrapper function. We add *dom* and *cod* operations to the language for accessing the parts of a function coercion. When a coercion is applied to a closure for the second time, the coercions are normalized to save space. The complete definition of the new procedure for applying coercions to a value is shown in Figure 9.

The new definitions of the expression evaluation function valueof2 and its auxiliary simple-valueof2 are shown in Figure 10. The main additions are to handle the dom and cod operators on coercions and the addition of a coercion to every closure.

The transition rules for the faster abstract machine are given in Figure 11. The procedures run-faster and observe3 are shown in Figure 12. Notice that there are fewer cases and the cases are somewhat simpler compared to the first abstract machine.

We now have a machine that does not perform extra dispatching at function calls. There is still a small amount of overhead in the form of passing the $c_{\_\_}$ argument. This overhead can be removed by passing the entire closure to itself (instead of passing the array of free variables and the coercion separately), and from inside the function, accessing the coercion from the closure. Also, there is still a little overhead in a return: we have to apply the pending coercion that is on the stack.

```
(define apply-coercion-faster
  (lambda (seq-norm)
    (lambda (v c)
      (match c
        [(id ,T) v]
        [(fail ,label ,T1 ,T2) `(blame ,label)]
        [,other
         (match v
           [((λ (,x : ,T1) ,s) ,env #f)
            (let ([x1 (gensym "x")][x2 (gensym "x")]
                  [f (gensym "f")])
            (let ([s2 `((cast ,x2 ,x1
                          : (dom c__)) (tail-call ,f ,x2 : (cod c__)))]
                  [env2 `((,f . ,v) . ,env)])
               `((λ (,x1 : ,T1) ,s2) ,env2 ,c)))]
           [((λ (,x : ,T1) ,s) ,env ,c2)
            (match (seq-norm c2 c)
              [(fail ,label ,T3 ,T4) `(blame ,label)]
              [(seq (→ ,c1 ,c2) (fail ,label ,T3 ,T4)) `(blame ,label)]
              [,c3 `((λ (,x : ,T1) ,s) ,env ,c3)])]
           [(cast ,v1 : ,c1) (mk-cast-lazy v1 (seq-norm c1 c))]
           [,other (mk-cast-lazy v c)])]))))
```

Figure 9. Procedure for applying a coercion to a value in the faster abstract machine.

```
(define transition-faster
  (lambda (s env stack coerce seq-norm)
    (match s
     [((assign ,x ,e) . ,s2)
      (let ([v (valueof e env)])
         `(state ,s2 ((,x . ,v) . ,env) ,stack))]
     [((if ,cnd ,thn-ss ,els-ss) . ,s2)
      (let ([v (valueof cnd env)])
         (cond [v `(state ,(append thn-ss s2) ,env ,stack)]
               [else `(state ,(append else-ss s2) ,env ,stack)]))]
     [((call ,x ,e1 ,e2) . ,s2)
      (let ([v1 (valueof2 e1 env)] [v2 (valueof2 e2 env)])
         (match v1 [((λ (,y : ,T1) ,s3) ,env2 ,cf)
                    `(state ,s3 ((c__ . ,cf) . ((,y . ,v2) . ,env2)) (stack (id _) (,x ,s2 ,env ,stack)))]))]
     [((return ,e))
      (match stack
       [(stack ,c (,x ,s2 ,env2 ,stack2))
        (letB [v (coerce (valueof2 e env) c)] `(state ,s2 ((,x . ,v) . ,env2) ,stack2))]
       [(stack ,c ()) (coerce (valueof2 e env) c)])]
     [((tail-call ,e1 ,e2))
      (let ([v1 (valueof2 e1 env)] [v2 (valueof2 e2 env)])
         (match v1 [((λ (,y : ,T1) ,s3) ,env2 ,cf)
                    `(state ,s3 ((c__ . ,cf) . ((,y . ,v2) . ,env2)) ,stack)]))]
     [((tail-call ,e1 ,e2 : ,c1))
      (let ([v1 (valueof2 e1 env)] [v2 (valueof2 e2 env)] [c1 (valueof2 c1 env)])
         (match v1 [((λ (,y : ,T1) ,s3) ,env2 ,cf)
                    (match stack
                     [(stack ,c2 ,rest) `(state ,s3 ((c__ . ,cf) . ((,y . ,v2) . ,env2)) (stack ,(seq-norm c1 c2) ,rest))])]))]
     [((cast ,x ,e : ,c) . ,s2)
       (letB [v (coerce (valueof2 e env) (valueof2 c env))]
          `(state ,s2 ((,x . ,v) . ,env) ,stack))]
     [,other (error 'transition-faster "unmatched ~s" other)])))
```

Figure 11. The transition-faster and run-faster functions for the more efficient abstract machine.

```
(define simple-valueof2
  (lambda (e env)
    (match e
     [,k (guard (constant? k)) k]
     [,x (guard (symbol? x)) (cdr (assq x env))]
     [(λ (,x : ,T1) ,s) `((λ (,x : ,T1) ,s) ,env #f)]
     [(id ,T) e]
     [(proj ,l ,label) e]
     [(inj ,l) e]
     [(→ ,e1 ,e2) e]
     [(seq ,e1 ,e2) e]
     )))

(define valueof2
  (lambda (e env)
    (match e
     [(prim ,op ,e) (delta op (simple-valueof2 e env))]
     [(dom ,e) (match (simple-valueof2 e env) [(→ ,c1 ,c2) c1])]
     [(cod ,e) (match (simple-valueof2 e env) [(→ ,c1 ,c2) c2])]
     [,other (simple-valueof2 e env)])))
```

Figure 10. The valueof2 function for the faster abstract machine.

```
(define run-faster
  (lambda (e mk-coerce coerce seq-norm)
    (match (typecheck '() e)
     [(,casted-e ,T)
      (let loop ([s (to-ANF casted-e #t mk-coerce)] [env '()]
                 [stack `(stack (id ,T) ())])
        (match (transition-faster s env stack coerce seq-norm)
         [(state ,s ,env ,stack)
          (loop s env stack)]
         [,r (observe3 r)]))])))

(define observe3
  (lambda (r)
    (match r
      [,k (guard (constant? k)) k]
      [((λ (,x : ,T1) ,s) ,env ,c)
       (match c
        [(seq (→ ,c1 ,c2) (inj ,l)) 'dynamic]
        [(inj ,l) 'dynamic]
        [,other 'function])]
      [(cast ,v : (inj ,l)) 'dynamic]
      [(blame ,label) `(blame ,label)])))
```

Figure 12. The run-faster and observe3 procedures.

Conjecture 8 (Correctness of the faster abstract machine)**.**
Suppose e is a well-typed program. Let $(e'\ T) = (\textsf{typecheck}'\,()\ e)$ and $s = (\textsf{to-ANF}\, e'\ \texttt{\#t}\ \textsf{mk-coerce-X})$. Then we have

$(\textsf{run-faster}\, s\ \textsf{mk-coerce-X}\, \textsf{seq-X}) = o$ if and only if $(\textsf{eval-X}\, e) = o$

for $\textsf{X} = \textsf{ld}$ and $\textsf{X} = \textsf{ud}$.

6. Eager Cast Checking

Section 3 mentioned that there is a design choice between eager and lazy cast checking. Recall the following example.

```
(define eg1
  (letT [f0 : dyn 0 = (λ (x : int) (inc x 2))]
    (letT [f1 : (→ bool bool) 1 = f0]
      ‘(f1 #t 3))))
```

With eager cast checking, the cast from dyn to (→ bool bool) fails at the moment when it is applied to a value. Whereas with lazy cast checking, the casts initially succeeds, but then later, when the function is applied at (f1 #t), the cast fails. Eager cast checking has the advantage that it tells the programmer as soon as possible that something is amiss. Further, when using the space-efficient implementations, eager and lazy checking are about the same regarding run-time overhead.

We gave the specification for lazy cast checking in Section 3, but the specification for eager checking was postponed. The reason for the postponement was that specifying the semantics of eager cast checking requires more machinery than for lazy cast checking. (I'll expand on this claim in the next paragraph.) Thankfully, in the meantime we have acquired the necessary machinery: the Coercion Calculus. The Eager Coercion Calculus was first developed by Herman et al. [2007] and was extended to include blame labels in by Siek et al. [2009]. Here we flesh out more of the theory of eager coercions: characterizing the normal forms and defining an efficient method of composing eager coercions.

Before discussing the eager coercion calculus, let us see why the eager coercion calculus is needed for the specification of the semantics, not just for an efficient implementation. After all, there was no mention of coercions in the definitional interpreters of the lazy variants. Instead, those interpreters just talked about casts (a pair of types and a blame label). Recall that the main events in those semantics were the apply-cast-ld and apply-cast-lud procedures that that apply a cast to a value.

It is instructive to see where naive definitions of an apply-cast-ed function for eager checking break down. The most obvious thing to try is to modify the apply-cast-ld function to check for (deep) type consistency instead of shallow consistency. So we change (shallowconsistent? T1 T2) to (consistent? T1 T). Let us see what happens on an example a little different from the previous example. In this example, the casts go through (→ dyn dyn) instead of dyn.

```
(define eg1c
  (letT [f0 : (→ dyn dyn) 0 = (λ (x : int) (inc x 2))]
    (letT [f1 : (→ bool bool) 1 = f0]
      ‘(f1 #t 3))))
```

With the naive approach, both casts succeed, producing the value

```
’(cast 1 (cast 0 (λ (x:int) (inc x))
                 : (→ int int) ⇒ (→ dyn dyn))
         : (→ dyn dyn) ⇒ (→ bool bool))
```

However, that is not what an eager cast checking should do. The above should not be a value, it should have already failed.

So the apply-cast-ed function not only needs to check whether the target type is consistent with the source type, it also needs to check whether the target type is consistent with all of the casts that are wrapping the value. One way we could try to do this is to compute the greatest lower bound (with respect to naive subtyping) of all the types in the casts on the value, and then compare the target type to the greatest lower bound (meet). The meet operator operator was defined in Section 2, though we would introduce a bottom type $\bot$ and change the meet operator to be a total function. Next we would define a function that computes the meet of all the casts wrapping a value. Then we would have apply-cast-id use the this meet operator and check whether the result is consistent with the target type.

How does this version fare on our example? An error would be triggered when the value flows into the cast from (→ dyn dyn) to (→ bool bool), so that is good, but the blame goes to 1. But the prior work on eager checking based on coercions says that 0 should be blamed instead! The problem is that the $\sqcap$ operator forgets about all the blame labels that are in the casts wrapping the value. In this example, it is dropping the label 0 which ought to be blamed. With these other avenues exhausted, let us turn to the eager variant of the Coercion Calculus.

6.1 The Eager Coercion Calculus

In the context of the Coercion Calculus, one needs to add the following two reduction rules to obtain eager cast checking. What these rules do is make sure that failure coercions immediately bubble up to the top of the coercion where they trigger a cast failure.

$$\begin{aligned} \mathit{fail}^{\ell} \to c &\longrightarrow_{\mathbf{E}} \mathit{fail}^{\ell} \\ \hat{c} \to \mathit{fail}^{\ell} &\longrightarrow_{\mathbf{E}} \mathit{fail}^{\ell} \end{aligned}$$

In the second rule, we require the domain to be in normal form, thereby imposing a left-to-right ordering on coercion failures.

To ensure confluence, we also need to make two changes to existing reduction rules. In the rule for composing function coercions, we require that the two coercions be in normal form. (The notation $\tilde{c}$ is new and will be explained shortly.)

$$(\tilde{c}_{11} \to \tilde{c}_{12}); (\tilde{c}_{21} \to \tilde{c}_{22}) \longrightarrow_{\mathbf{E}} (\tilde{c}_{21}; \tilde{c}_{11}) \to (\tilde{c}_{12}; \tilde{c}_{22})$$

Here is the counter-example to confluence if the above restriction is not made.

$$\begin{aligned} (\mathit{fail}^{\ell_1} \to c_1); (\mathit{fail}^{\ell_2} \to c_2) &\longrightarrow \mathit{fail}^{\ell_1}; (\mathit{fail}^{\ell_2} \to c_2) \longrightarrow \mathit{fail}^{\ell_1} \\ (\mathit{fail}^{\ell_1} \to c_1); (\mathit{fail}^{\ell_2} \to c_2) &\longrightarrow (\mathit{fail}^{\ell_2}; \mathit{fail}^{\ell_1}) \to (c_1; c_2) \\ &\longrightarrow \mathit{fail}^{\ell_2} \to (c_1; c_2) \\ &\longrightarrow \mathit{fail}^{\ell_2} \end{aligned}$$

There is also a confluence problem regarding the following rule.

$$(c_{11} \to c_{12}); \mathit{fail}^{\ell} \longrightarrow \mathit{fail}^{\ell}$$

The counter-example is

$$\begin{aligned} (\iota \to \mathsf{bool!}); (\iota \to \mathsf{int?}^{\ell_2}); \mathit{fail}^{\ell_1} &\longrightarrow (\iota; \iota) \to (\mathsf{bool!}; \mathsf{int?}^{\ell_2}); \mathit{fail}^{\ell_1} \\ &\longrightarrow^* \iota \to (\mathit{fail}^{\ell_2}); \mathit{fail}^{\ell_1} \\ &\longrightarrow^* \mathit{fail}^{\ell_2} \\ (\iota \to \mathsf{bool!}); (\iota \to \mathsf{int?}^{\ell_2}); \mathit{fail}^{\ell_1} &\longrightarrow (\iota \to \mathsf{bool!}); \mathit{fail}^{\ell_1} \\ &\longrightarrow \mathit{fail}^{\ell_1} \end{aligned}$$

We fix this problem by removing the reduction rule that allows function coercions to be consumed on the left of a failure. Here is the complete set of reduction rules for the Eager Coercion Calculus.

$$
\begin{aligned}
\iota; c &\longrightarrow_{\mathbf{E}} c \\
c; \iota &\longrightarrow_{\mathbf{E}} c \\
(\tilde{c}_{11} \to \tilde{c}_{12}); (\tilde{c}_{21} \to \tilde{c}_{22}) &\longrightarrow_{\mathbf{E}} (\tilde{c}_{21}; \tilde{c}_{11}) \to (\tilde{c}_{12}; \tilde{c}_{22}) \\
\mathit{fail}^{\ell}; c &\longrightarrow_{\mathbf{E}} \mathit{fail}^{\ell} \\
I!; \mathit{fail}^{\ell} &\longrightarrow_{\mathbf{E}} \mathit{fail}^{\ell} \\
(\mathit{fail}^{\ell} \to c) &\longrightarrow_{\mathbf{E}} \mathit{fail}^{\ell} \\
(\tilde{c} \to \mathit{fail}^{\ell}) &\longrightarrow_{\mathbf{E}} \mathit{fail}^{\ell}
\end{aligned}
$$

These additions and changes to the reduction rules cause changes in the normal forms of coercions. First, fail^{ℓ} cannot appear under a function coercion. We therefore introduce another category, called "normal parts" and written $\tilde{c}$, that excludes fail^{ℓ} (but still includes $I?^{\ell_1}; \mathit{fail}^{\ell_2}$ because the ℓ_1 projection could still fail and take precedence over ℓ_2). Also, $(\tilde{c}_1 \to \tilde{c}_2); \mathit{fail}^{\ell}$ is now a normal form. The following grammar defines the normal coercions for eager cast checking.

$$
\begin{array}{llcl}
\text{wrappers} & \bar{c} & ::= & I! \mid \tilde{c} \to \tilde{c} \mid (\tilde{c} \to \tilde{c}); I! \\
\text{normal parts} & \tilde{c} & ::= & \bar{c} \mid \iota \mid I?^{\ell} \mid I?^{\ell}; \mathit{fail}^{\ell} \mid \\
 & & & I?^{\ell}; I! \mid I?^{\ell}; (\tilde{c} \to \tilde{c}) \mid \\
 & & & I?^{\ell}; (\tilde{c} \to \tilde{c}); I! \mid (\tilde{c} \to \tilde{c}); \mathit{fail}^{\ell} \mid \\
 & & & I?^{\ell}; (\tilde{c} \to \tilde{c}); \mathit{fail}^{\ell} \\
\text{normal coercions} & \hat{c} & ::= & \tilde{c} \mid \mathit{fail}^{\ell}
\end{array}
$$

6.2 The Eager Gradually-Typed Lambda Calculus

Taking a step back, recall that we defined the semantics of the Lazy Gradually-Typed Lambda Calculi in terms of the definitional interpreter eval-ld and eval-lud. We can do the same for the Eager variant but using coercions to give the meaning of casts. The following is the definition of values and results for the Eager variant.

$$
\begin{array}{lrcl}
 & F & \in & V \to R \\
\text{values} & v \in V & ::= & k \mid F \mid (\mathsf{cast}\, v : \bar{c}) \\
\text{results} & r \in R & ::= & v \mid (\mathsf{blame}\, \ell)
\end{array}
$$

To implement the cast application function, we need a procedure for applying a coercion to a value, apply-coercion-eager, shown below. Instead of composing coercions using the seq-lazy procedure, it uses the seq-eager procedure, which we discuss in the next paragraph. The auxiliary mk-cast-eager function differs from the lazy version in that another case is needed to handle the new coercion normal form: a function coercion followed by a failure.

```
(define mk-cast-eager
  (lambda (v c)
    (match c
      [(id ,T) v]
      [(fail ,label ,T1 ,T2) ‘(blame ,label)]
      [(seq (→ ,c1 ,c2) (fail ,label ,T1 ,T2)) ‘(blame ,label)]
      [,other ‘(cast ,v : ,c)])))
(define apply-coercion-eager
  (lambda (mk-coerce)
    (lambda (v c)
      (match v
       [(cast ,v1 : ,c1)
        (mk-cast-eager v1 ((seq-eager mk-coerce) c1 c))]
       [,other (mk-cast-eager v c)]))))
```

Figure 13 presents the seq-eager procedure. Only two changes are required relative to seq-lazy. First, in the case for composing function coercions, we use the auxiliary mk-arrow-eager procedure to detect failures and propagate them up to the top. In addition, the case in seq-lazy for dealing with a function coercion followed by a failure is removed, mirroring the removal of the corresponding reduction rule.

```
(define mk-arrow-eager
  (lambda (c1 c2)
    (match ‘(→ ,c1 ,c2)
      [(→ (fail ,label ,T1 ,T2) ,c2)
       (match (typeof-coercion c2)
         [(,T3 ,T4) ‘(fail ,label (→ ,T2 ,T3) (→ ,T1 ,T4))])]
      [(→ ,c1 (fail ,label ,T3 ,T4))
       (match (typeof-coercion c2)
         [(,T1 ,T2) ‘(fail ,label (→ ,T2 ,T3) (→ ,T1 ,T4))])]
      [,other other])))

(define seq-eager
  (lambda (mk-coerce)
    (lambda (c1 c2)
      (let ([recur (seq-eager mk-coerce)])
        (match ‘(,c1 ,c2)
          [((id ,T) ,c2) c2]
          [(,c1 (id ,T)) c1]
          [((inj ,I1) (proj ,I2 ,label)) (mk-coerce I1 I2 label)]
          [((→ ,c11 ,c12) (→ ,c21 ,c22))
           (mk-arrow-eager (recur c21 c11) (recur c12 c22))]
          [((fail ,label ,T1 ,T2) ,c2) ‘(fail ,label ,T1 ,T2)]
          [((inj ,I) (fail ,label ,T1 ,T2)) ‘(fail ,label ,T1 ,T2)]
          [((seq ,c11 ,c12) ,c2) (recur c11 (recur c12 c2))]
          [((proj ,I ,label) (seq (→ ,c21 ,c22) ,c23))
           ‘(seq (proj ,I ,label) (seq (→ ,c21 ,c22) ,c23))]
          [(,c1 (seq ,c21 ,c22)) (recur (recur c1 c21) c22)]
          [(,c1 ,c2) ‘(seq ,c1 ,c2)])))))
```

Figure 13. The composition operator for eager coercions.

Conjecture 9 (Correctness of seq-eager)**.** *Given two well-typed coercions in normal form, c_1 and c_2, we have*

1. $((\textit{seq-eager mk-coerce-ld})\, c_1 c_2) = \hat{c_3}$ *and* $(c_1; c_2) \longmapsto^*_{\mathbf{E} \cup \mathbf{D}} \hat{c_3}$.
2. $((\textit{seq-eager mk-coerce-lud}) c_1 c_2) = \hat{c_3}$ *and* $(c_1; c_2) \longmapsto^*_{\mathbf{E} \cup \mathbf{UD}} \hat{c_3}$.

We can now give the definition of interp-ed and interp-eud (Figure 14). The apply-cast-eager function is defined in terms of coercion application and the mk-coerce function, which creates a coercion from a source and target type and a blame label. The apply-eager function is similar to the lazy variant, but adapted to coercions. The semantics for the D and UD variants of the Eager Gradually-Typed Lambda Calculus are defined by the eval-ed and eval-eud procedures in Figure 14.

6.3 Efficient Eager Machines

To obtain space-efficient machines for the Eager variants, we use the run procedure unchanged, but supply the appropriate eager operations: mk-coerce-ed, apply-coercion-ed, and seq-ed (or the versions with eud suffix to obtain the UD semantics). Similarly, to obtain a machine that improves the efficiency of function calls, we use the run-faster procedure unchanged, again supplying the appropriate eager operations.

7. Conclusion

The dynamic semantics of gradual typing is not as straightforward as one might think. First, there is the question of blame assignment, for which we have seen the D and UD strategies. The UD strategy is the natural way to track blame if one views the dyn type as a recursive type. On the other hand, the D strategy corresponds to a more natural subtyping relation in which dyn is the top type. This

```
(define apply-cast-eager
  (lambda (mk-coerce)
    (lambda (label v T1 T2)
      ((apply-coercion-eager mk-coerce)
       v (mk-coerce T1 T2 label)))))
(define apply-eager
  (lambda (apply-coercion)
    (lambda (F v)
      (let ([recur (apply-eager apply-coercion)])
        (match F
          [(cast ,F1 : (→ ,c1 ,c2))
           (letB [x3 (apply-coercion v c1)]
           (letB [x4 (recur F1 x3)]
              (apply-coercion x4 c2)))]
          [,proc (guard (procedure? proc)) (proc v)])))))

(define mk-coerce-ed (mk-coerce-d mk-arrow-eager))
(define mk-coerce-eud (mk-coerce-ud mk-arrow-eager))
(define apply-coercion-ed (apply-coercion-eager mk-coerce-ed))
(define apply-coercion-eud (apply-coercion-eager mk-coerce-eud))
(define apply-cast-ed (apply-cast-eager mk-coerce-ed))
(define apply-cast-eud (apply-cast-eager mk-coerce-eud))
(define seq-ed (seq-eager mk-coerce-ed))
(define seq-eud (seq-eager mk-coerce-eud))

(define interp-ed
  (interp apply-cast-ed (apply-eager apply-coercion-ed)))
(define interp-eud
  (interp apply-cast-eud (apply-eager apply-coercion-eud)))

(define observe4
  (lambda (r)
    (match r
      [,k (guard (constant? k)) k]
      [,proc (guard (procedure? proc)) 'function]
      [(cast ,v : (→ ,c1 ,c2)) 'function]
      [(cast ,v : (seq (→ ,c1 ,c2) (inj ,l))) 'dynamic]
      [(cast ,v : (inj ,l)) 'dynamic]
      [(blame ,label) `(blame ,label)])))

(define eval-ed (eval interp-ed observe4))
(define eval-eud (eval interp-eud observe4))
```

Figure 14. Interpreters for Eager D and Eager UD.

tutorial presents the semantics for D and UD both in terms of casts between types and in terms of the Coercion Calculus.

There is also the design choice between eager and lazy cast checking. Lazy cast checking is straightforward to express in terms of casts and types, whereas for eager cast checking, even the definitional interpreters needed the Coercion Calculus.

Regarding efficiency, we develop an abstract machine that compresses sequences of casts by representing casts as coercions and by normalizing those coercions. We present new and more elegant procedures for normalizing lazy and eager coercions. To address the function-call overhead in the space-efficient machine, we create another machine that unifies the representation of functions and casted functions, thereby streamlining the calling convention.

Acknowledgments

We thank Olivier Danvy for the invitation to give this tutorial and for his suggestions for improvement of this paper. We thank the NSF for supporting this work in the form of award 0846121.

References

O. Danvy. Three steps for the CPS transformation. Technical Report CIS-92-02, Kansas State University, December 1991.

O. Danvy. Back to direct style. In *European Symposium on Programming*, ESOP, pages 130–150, February 1992.

O. Danvy and A. Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, December 1992.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, ICFP, pages 48–59, October 2002.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Conference on Programming Language Design and Implementation*, PLDI, pages 502–514, June 1993.

R. Garcia. Calculating threesomes with blame. October 2012.

J. Gronski and C. Flanagan. Unifying hybrid types and contracts. In *Trends in Functional Prog. (TFP)*, page XXIX, April 2007.

L. T. Hansen. Evolutionary programming and gradual typing in ECMAScript 4 (tutorial). Technical report, ECMA TG1 working group, November 2007.

R. Harper. Practical foundations for programming languages. Working Draft, Version 1.32, May 2012.

F. Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.

A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *Symposium on Principles of Programming Languages*, POPL, pages 481–494, January 2012.

J. G. Siek. My new favorite abstract machine: ECD on ANF. http://siek.blogspot.com/2012/07/my-new-favorite-abstract-machine-ecd-on.html, July 2012.

J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

J. G. Siek and W. Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, volume 4609 of *LCNS*, pages 2–27, August 2007.

J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Symposium on Principles of Programming Languages*, POPL, pages 365–376, January 2010.

J. G. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In *European Symposium on Programming*, ESOP, pages 17–31, March 2009.

P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, ESOP, pages 1–16, March 2009.

T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Symposium on Principles of Programming Languages*, POPL, pages 377–388, January 2010.