

ValleyScript: It's Like Static Typing

Cormac Flanagan

November 16, 2007

Abstract

We formalize the ES4 notions of **like** types and **wrap** operators for a lambda-calculus with ES4-style objects, to better understand these concepts and to clarify what guarantees can be provided by the verifier in strict mode.

1 Change Log

- 6 Oct: extended with generic function, as a warm-up for implementing all this in the verifier.
- 8 Nov: extended with conversion from **Int** to **Bool**.
- 14 Nov: extended with generic functions.
- 14 Nov: added wrap as a type constructor.
- todo: typedefs, generics in type system, wrap in optimization system

2 Language Overview

We consider the implementation of a *gradual typed* language that supports both typed and untyped terms, which interoperate in a flexible manner. We begin by defining the syntax of terms and types in the language: see Figure 1. In addition to the usual terms of the lambda calculus (variables, abstractions, and application), the language also includes constants and expressions to create, dereference, and update objects. It also includes **is** expressions, which check that a value has a particular type, and **wrap** expressions, which, if necessary, wrap the given value to ensure that it behaves like it has that type.

The type language is fairly rich. In addition to base types (**Int** and **Bool**), function types, and object types, the language includes additional types related to gradual typing. The type ***** is (roughly) a top type, and indicates that no static type information is known. The type **like** *T* describes values whose value components match *T*, but whose type components may be more vague than *T*, due to the presence of the type *****. (Due to imperative constructs, that matching-value guarantee does not persist, and so **like** types are helpful for debugging but do not provide strong guarantees.)

We include generic function definition, generic function application, and the associated polymorphic types and type variables.

Figure 1: Syntax

$e ::=$	<i>Terms:</i>
c	constant
x	variable
$\lambda x:S. e : T$	abstraction
$e\ e$	application
$\Lambda X. e : T$	generic function
$e[T]$	generic application
$\{\bar{l} = \bar{e}\} : T$	object expression
$e.l$	member selection
$e.l()$	member invocation
$e.l := e$	member update
$e\ \text{is}\ T$	dynamic type check
$e\ \text{wrap}\ T$	wrapping a value
$c ::=$	<i>Constants:</i>
n	integer constant
b	boolean constant
$S, T ::=$	<i>Types:</i>
Int	integers
Bool	booleans
$S \rightarrow T$	function type
X	type variable
$\forall X. T$	generic function type
$\{\bar{l} : \bar{T}\}$	object types
$*$	dynamic type
$\text{like}\ T$	like types
$\text{wrap}\ T$	wrap types

3 Type Relations

The judgement $S <: T$ (S is a subtype of T) checks if every value of type S can be assigned to a variable of type T (some caveats for **like** types are needed here). The type $*$ is a top type, and we have width-subtyping on objects. Although the subtyping judgement does not include an explicit environment, the rule [S-GENERIC] means that the types may include free type variables. This rule applies implicit alpha-renaming so that the same type variable can be interpreted as the same on both sides.

The judgement $S \sim: T$ (S is compatible with T) extends the subtype relation with a more flexible interpretation of dynamic types; in particular, the type $*$ is compatible with any type. The compatibility judgement is not transitively closed. In particular, we have that $\text{Int} \sim: *$ and $* \sim: \text{Bool}$, but the judgement $\text{Int} \sim: \text{Bool}$ does not hold. The rule [SUB-LIKE] for **like** types switches from checking subtyping to checking compatibility.

The **like** type constructor is covariant, in that the following rule is admissible:

$$\frac{S \sim: T}{\text{like } S <: \text{like } T}$$

Lemma 1 *The subtyping judgement is transitively closed.*

Proof: TBW

The constructor **like** is “idempotent”, in that we have that $(\text{like } \text{like } T) <: (\text{like } T) <: (\text{like } \text{like } T)$.

$$\frac{\frac{T \sim: T}{\text{like } T \sim: T}}{\text{like } \text{like } T <: \text{like } T} \qquad \frac{\text{like } T \sim: \text{like } T}{\text{like } T <: \text{like } \text{like } T}$$

Figure 2: Type Relations

Subtyping	$\boxed{S <: T}$
$\overline{T <: T}$	[SUB-REFL]
$\frac{T_1 <: S_1 \quad S_2 <: T_2}{(S_1 \rightarrow S_2) <: (T_1 \rightarrow T_2)}$	[SUB-ARROW]
$\frac{S <: T}{(\forall X. S) <: (\forall X. T)}$	[SUB-GENERIC]
$\frac{T_i <: S_i \quad S_i <: T_i \quad \text{for } i \in 1..n}{\{l_i : S_i^{i \in 1..n+m}\} <: \{l_i : T_i^{i \in 1..n}\}}$	[SUB-OBJ]
$\overline{T <: *}$	[SUB-DYN]
$\frac{S \sim: T}{S <: \text{like } T}$	[SUB-LIKE]
Compatible Types	$\boxed{S \sim: T}$
Includes the above subtyping rules (with $<:$ replaced by $\sim:$), and also:	
$\overline{* \sim: T}$	[COM-DYN]
$\frac{S \sim: T}{\text{like } S \sim: T}$	[COM-LIKE]
Convertible Types	$\boxed{S \prec T}$
$\frac{S <: T}{S \prec T}$	[CON-SUB]
$\overline{\text{Int} \prec \text{Bool}}$	[CON-INTBOOL]
$\frac{S \sim: T}{S \prec \text{wrap } T}$	[CON-WRAP]

4 Evaluation

We next describe the evaluation semantics of the language. The set of values in the language is given by:

$v ::=$	<i>Values:</i>
c	constant
$\lambda x : S. e : T$	abstraction
$\Lambda X. t : T$	generic abstraction
$v \text{ wrapped } T$	wrapped value
a	object address
$o ::=$	<i>Object value:</i>
$\{\bar{l} = \bar{v}\} : T$	object value

A *object store* σ maps object addresses a to object values of the form $\{\bar{l} = \bar{v}\} : T$. Every value has an *allocated type* according to the function $ty_\sigma(v)$:

$ty_\sigma(n)$	$=$	Int
$ty_\sigma(b)$	$=$	Bool
$ty_\sigma(\lambda x : S. e : T)$	$=$	$(S \rightarrow T)$
$ty_\sigma(\Lambda X. t : T)$	$=$	$(\forall X. T)$
$ty_\sigma(v \text{ wrapped } T)$	$=$	T
$ty_\sigma(a)$	$=$	T if $\sigma(a) = \{\dots\} : T$

Values and object values in the store are *closed* in that they do not contain free program variables x or type variables X ; though they may contain object addresses.

An evaluation context is:

$$C ::= \bullet \mid C \ t \mid v \ C \mid C[T] \mid C \text{ is } T \mid C \text{ wrap } T \mid \{\bar{l} = \bar{v}, l = C, \bar{l} = \bar{e}\} : T \mid C.l \mid C.l() \mid C.l := e \mid v.l := C$$

A *state* is a pair of an object store and a current expression. The evaluation relation on states is defined by the rules in Figure 3. The rule [E-ALLOC] requires an explicit object type with a type for each field, but those field types could simply be $*$. (Note, fields can never be deleted in our semantics.) Thus, the necessary object type could always be locally inferred from the object expression.

Several rules refer to the judgement $v \text{ is}_\sigma T$, which checks if the value v matches the type T : see Figure 4. This judgement relies on two subtype-like relations on types.

Lemma 2 (Preservation under subtyping) *If $v \text{ is}_\sigma S$ and $S <: T$ then $v \text{ is}_\sigma T$.*

Does not hold, as $\text{true is}_\sigma * <: \text{like Int}$.

Figure 3: Evaluation Rules

$\sigma, C[(\lambda x : S. t : T) v] \longrightarrow \sigma, C[t[x := v'] \text{ is } T] \quad \text{if } v' = \text{convert}_\sigma(v, S)$	[E-BETA1]
$\sigma, C[(w \text{ wrapped } (S \rightarrow T)) v] \longrightarrow \sigma, C[(w (v \text{ wrap } S)) \text{ wrap } T]$	[E-BETA2]
$\sigma, C[(\lambda X. t : T)[S]] \longrightarrow \sigma, C[t[X := S] \text{ is } T[X := S]]$	[E-GENERIC1]
$\sigma, C[(w \text{ wrapped } (\forall X. S))[T]] \longrightarrow \sigma, C[w[T] \text{ is } S[X := T]]$	[E-GENERIC2]
$\sigma, C[v \text{ is } T] \longrightarrow \sigma, C[v] \quad \text{if } v \text{ is}_\sigma T$	[E-AS]
$\sigma, C[v \text{ wrap } T] \longrightarrow \sigma, C[v'] \quad \text{if } v' = \text{convert}_\sigma(v, \text{wrap } T)$	[E-WRAP]
$\sigma, C[\{l_i = v_i^{i \in 1..n}\} : T] \longrightarrow \sigma[a := (\{l_i = v'_i\} : T)], C[a]$ where $T = \{l_i : T_i^{i \in 1..n}\}$, $v'_i = \text{convert}_\sigma(v_i, T_i)$ and a fresh	[E-ALLOC]
$\sigma, C[a.l] \longrightarrow \sigma, C[v] \quad \text{if } \sigma(a) = \{l = v, \dots\} : T$	[E-GET1]
$\sigma, C[(w \text{ wrapped } \{l : T, \dots\}).l] \longrightarrow \sigma, C[(w.l) \text{ wrap } T]$	[E-GET2]
$\sigma, C[a.l()] \longrightarrow \sigma, C[(a.l) a]$	[E-CALL]
$\sigma, C[a.l := v] \longrightarrow \sigma[a, l := v'], C[v]$ where if $\sigma(a) = \{\dots\} : \{l : T, \dots\}$ then $v' = \text{convert}_\sigma(v, T)$ else $v' = v$	[E-ASSIGN1]
$\sigma, C[(w \text{ wrapped } \{l : T, \dots\}).l := v] \longrightarrow \sigma, C[w.l := (v \text{ wrap } T)]$	[E-ASSIGN2]

Figure 4: Dynamic Type Checks

$\text{convert}_\sigma(v, T) = v \quad \text{if } v \text{ is}_\sigma T$	
$\text{convert}_\sigma(0, \text{Bool}) = \text{false}$	
$\text{convert}_\sigma(n, \text{Bool}) = \text{true} \quad \text{if } n \neq 0$	
$\text{convert}_\sigma(v, \text{wrap } T) = v' \quad \text{if } v' = \text{convert}_\sigma(v, T)$	
$\text{convert}_\sigma(v, \text{wrap } T) = v \text{ wrapped } T \quad \text{if } v \text{ is}_\sigma \text{ like } T$	
Dynamic type check	$\boxed{v \text{ is}_\sigma T}$
$\frac{ty_\sigma(v) <: T \quad (v \neq a \text{ or } T \neq \text{like } \dots)}{v \text{ is}_\sigma T}$	[IS-OK]
$\frac{\sigma(a) = \{l_i = v_i^{i \in 1..n+m}\} : S \quad v_i \text{ is}_\sigma \text{ like } T_i \text{ for } i \in 1..n}{a \text{ is}_\sigma \text{ like } \{l_i : T_i^{i \in 1..n}\}}$	[IS-OBJ]

5 Strict Mode Type System

In a traditional statically typed language, the type system fulfills two goals:

1. It detects certain errors at compile time.
2. It guarantees what kinds of values are produced by certain expressions, which enables run-time check elimination.

The situation in ES4 is somewhat different, because of two reasons. First, in **standard** mode, we would like to eliminate run-time checks where possible, using a type-based analysis, without reporting any compile-time type errors. Second, **like** types weaken the guarantees provided by **strict** mode. For these reasons, we actually present *two* type systems.

For example, if a variable x has type **like** $\{f : \text{Int}\}$, then $x.f$ could actually return a value of any type. Nevertheless, $x.f$ would be expected to produce values of type **Int**, and so the expression $x.f.g$ would yield a compile-time type error. Thus, we say that $x.f$ has type **Int**, but this type is *only* a statement of intent; it does not guarantee what kinds of values are returned by that expression, and so cannot be used for run-time check elimination.

The strict mode type system is based on a judgement $E \vdash e : T$, stating that expression e has type T in environment E . Note that the type T *only* indicates that e is intended to produce values of type T ; there are no guarantees here, due to the use of **like** types. Thus, the strict mode type system If T is not a **like** type, then e will only produce values of that type. If $T = \text{like } T'$, then the intent is that e will only produce values of type T' , but there is no guarantee. However, this intent can be still used to detect type errors at compile time.

Note that the judgement $E \vdash e : T$ means that e is *expected* to only produce values of type T , and the purpose of the strict mode type system is only to heuristically detect errors at verification time. The following section presents a type system with stronger guarantees that are sufficient for removing some run-time type checks.

If $E \vdash e : T$, then T is not a **like** or **wrap** type (at least at the top level).

Figure 5: Type Rules for Strict Mode

Type rules	$E \vdash t : T$
$\frac{(x : (\text{like} \mid \text{wrap})^* T) \in E}{E \vdash x : T}$	[T-VAR]
$\frac{}{E \vdash c : \text{ty}(c)}$	[T-CONST]
$\frac{E, x : S \vdash e : T' \quad T' \prec T}{E \vdash (\lambda x : S. e : T) : (S \rightarrow T)}$	[T-FUN]
$\frac{E \vdash t_1 : (S \rightarrow T) \quad E \vdash t_2 : S' \quad S' \prec S}{E \vdash (t_1 \ t_2) : T}$	[T-APP1]
$\frac{E \vdash t_1 : * \quad E \vdash t_2 : S'}{E \vdash (t_1 \ t_2) : *}$	[T-APP2]
$\frac{E \vdash t : S}{E \vdash t \text{ is } T : T}$	[T-AS]
$\frac{E \vdash t : S}{E \vdash t \text{ wrap } T : T}$	[T-WRAP]
$\frac{E \vdash t_i : S_i \quad S_i \prec T_i \quad T = \{l_i : T_i^{i \in 1..n}\}}{E \vdash (\{l_i = t_i^{i \in 1..n}\} : T) : T}$	[T-ALLOC]
$\frac{E \vdash e : \{l : T, \dots\}}{E \vdash e.l : T}$	[T-GET1]
$\frac{E \vdash e : *}{E \vdash e.l : *}$	[T-GET2]
$\frac{E \vdash e_1 : \{l : T, \dots\} \quad E \vdash e_2 : S \quad S \prec T}{E \vdash e_1.l := e_2 : S}$	[T-SET1]
$\frac{E \vdash e_1 : * \quad E \vdash e_2 : S}{E \vdash e_1.l := e_2 : S}$	[T-SET2]

6 Check Optimization

We now sketch a type-based analysis that statically identifies dynamic type checks that can be eliminated. We introduce the following additional “**safe**” expression forms, for which run-time checks are unnecessary.

$e ::=$	<i>Expressions:</i>
...	expressions mentioned earlier
safe x	safe variable
safe $\lambda x:S. e : T$	safe abstraction
safe $e e$	safe application
safe $\{\bar{l} = \bar{e}\} : T$	safe object allocation
safe $e.l$	safe field selection
safe $e.l := e$	safe field update
$v ::=$	<i>Values:</i>
...	values mentioned earlier
safe $\lambda x:S. e : T$	safe abstraction

It is straightforward to formulate the operational semantics of the extended language.

Figure 6 presents *optimization rules*, which verify that **safe** ... occurs in correct places; it is straightforward to reformulate the analysis to infer these **safe** annotations.

The following lemmas remain to be proven.

Lemma 3 (No Failure) *For any term e , there is some placement of **safe** annotations into e yielding e' such that $\emptyset \Vdash e' : T$ for some T .*

Lemma 4 (Soundness) *If $\emptyset \Vdash e : T$, then **safe** operation in e can never get stuck.*

Figure 6: Type Rules for Optimization

Type rules	$E \Vdash t : T$
$\frac{(x : T) \in E}{E \Vdash \mathbf{safe} \ x : T}$	[O-VAR-SAFE]
$\frac{}{E \Vdash x : *}$	[O-VAR-UNSAFE]
$\frac{}{E \Vdash c : ty(c)}$	[O-CONST]
$\frac{E, x : S \Vdash e : T' \quad T' <: T}{E \Vdash \mathbf{safe} \ (\lambda x : S. e : T) : (S \rightarrow T)}$	[O-FUN-SAFE]
$\frac{E, x : S \Vdash e : T'}{E \Vdash (\lambda x : S. e : T) : (S \rightarrow T)}$	[O-FUN-UNSAFE]
$\frac{E \Vdash t_1 : (S \rightarrow T) \quad E \Vdash t_2 : S' \quad S' <: S}{E \Vdash \mathbf{safe} \ (t_1 \ t_2) : T}$	[O-APP-SAFE]
$\frac{E \Vdash t_1 : S \quad E \Vdash t_2 : S'}{E \Vdash (t_1 \ t_2) : *}$	[O-APP-UNSAFE]
$\frac{E \Vdash t : S}{E \Vdash t \ \mathbf{is} \ T : T}$	[O-AS]
$\frac{E \Vdash t : S}{E \Vdash t \ \mathbf{wrap} \ T : T}$	[O-WRAP]
$\frac{E \Vdash t_i : S_i \quad S_i <: T_i \quad T = \{l_i : T_i^{i \in 1..n}\}}{E \Vdash \mathbf{safe} \ (\{l_i = t_i^{i \in 1..n}\} : T) : T}$	[O-ALLOC-SAFE]
$\frac{E \Vdash t_i : S_i}{E \Vdash (\{l_i = t_i^{i \in 1..n}\} : T) : T}$	[O-ALLOC-UNSAFE]
$\frac{E \Vdash e : \{l : T, \dots\}}{E \Vdash \mathbf{safe} \ e.l : T}$	[O-GET-SAFE]
$\frac{E \Vdash e : S}{E \Vdash e.l : *}$	[O-GET-UNSAFE]
$\frac{E \Vdash e_1 : \{l : T, \dots\} \quad E \Vdash e_2 : S \quad S <: T}{E \Vdash \mathbf{safe} \ (e_1.l := e_2) : S}$	[O-SET-SAFE]
$\frac{E \Vdash e_1 : * \quad E \Vdash e_2 : S}{E \Vdash e_1.l := e_2 : S}$	[O-SET2]