ValleyScript: It's Like Static Typing

Cormac Flanagan

November 20, 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.
- 20 Nov: rework based on allocated and current types
- todo: add typedefs, rework verifier and optimizer

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

$e ::=$		*Terms:*
	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 `is` T	dynamic type check
	e `wrap` T	wrapping a value
$c ::=$		*Constants:*
	n	integer constant
	b	boolean constant
$S, T ::=$		*Types:*
	`Int`	integers
	`Bool`	booleans
	$S \to T$	function type
	X	type variable
	$\forall X.\, T$	generic function type
	$\{\bar{l} : \bar{T}\}$	object types
	$*$	dynamic type
	`like` T	like types
	`wrap` T	wrap types

3 Compatible Types

The judgement $S \sim: T$ (S *is compatible with* T) checks if values of type S can be assigned to a variable of type T. It is analogous to the usual subtype relation, but somewhat more complex because of `like` and $*$. The type $*$ is a top type, and we have width-subtyping on opjects.

The compatibility relation is intransitive since `Int` $\sim: *$ and $* \sim:$ `like Bool`, but `Int` $\sim:$ `like Bool` does not hold.

It is similar to the consistency relation of Siek and Taha, but differs in that whenever a down-conversion is performed via [CL-Dyn], the type constructor `like` is inserted to record this fact. Thus, in general, a type T denotes only values of type T, but `like` T denotes a somewhat larger set of values.

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 `like` type constructor is convariant via [CL-Like].

We have (`like` T) $\sim:$ (`like like` T), but not the converse, so `like like` T is somewhat useless. We could add extra rules to covert `like like` T to `like` T, but it seems unnecessary.

$$\frac{\texttt{like}\ T \sim: \texttt{like}\ T}{\texttt{like}\ T \sim: \texttt{like like}\ T}$$

Figure 2: Compatible Types

Compatible Types $\boxed{S \sim: T}$

$$\frac{}{T \sim: T} \quad \text{[C-Refl]}$$

$$\frac{T_1 \sim: S_1 \qquad S_2 \sim: T_2}{(S_1 \to S_2) \sim: (T_1 \to T_2)} \quad \text{[C-Arrow]}$$

$$\frac{T_1 \sim: \texttt{like}\ S_1 \qquad S_2 \sim: \texttt{like}\ T_2}{(S_1 \to S_2) \sim: \texttt{like}\ (T_1 \to T_2)} \quad \text{[CL-Arrow]}$$

$$\frac{S \sim: T}{(\forall X.\, S) \sim: (\forall X.\, T)} \quad \text{[C-Generic]}$$

$$\frac{S \sim: \texttt{like}\ T}{(\forall X.\, S) \sim: \texttt{like}\ (\forall X.\, T)} \quad \text{[CL-Generic]}$$

$$\frac{T_i \sim: S_i \qquad S_i \sim: T_i \qquad \text{for } i \in 1..n}{\{l_i : S_i^{i \in 1..n+m}\} \sim: \{l_i : T_i^{i \in 1..n}\}} \quad \text{[C-Obj]}$$

$$\frac{T_i \sim: \texttt{like}\ S_i \qquad S_i \sim: \texttt{like}\ T_i \qquad \text{for } i \in 1..n}{\{l_i : S_i^{i \in 1..n+m}\} \sim: \texttt{like}\ \{l_i : T_i^{i \in 1..n}\}} \quad \text{[CL-Obj]}$$

$$\frac{}{T \sim: *} \quad \text{[C-Dyn]}$$

$$\frac{}{* \sim: \texttt{like}\ T} \quad \text{[CL-Dyn]}$$

$$\frac{S \sim: T}{S \sim: \texttt{like}\ T} \quad \text{[C-Like]}$$

$$\frac{S \sim: T}{\texttt{like}\ S \sim: \texttt{like}\ T} \quad \text{[CL-Like]}$$

4 Evaluation

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

$v ::=$		*Values:*
	c	constant
	$\lambda x{:}S.\ e : T$	abstraction
	$\Lambda X.\, t : T$	generic abstraction
	v `wrapped` T	wrapped value
	a	object address
$o ::=$		*Object value:*
	$\{\bar{l} = \bar{v}\} : T$	object value

A *object store* σ maps object addresses a to object values of the form $\{\bar{l} = \bar{v}\} : T$. 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.

Every value has an *allocated type* according to the function $aty_\sigma(v)$:

$$\begin{array}{rcll} aty_\sigma(n) & = & \texttt{Int} & \\ aty_\sigma(b) & = & \texttt{Bool} & \\ aty_\sigma(\lambda x{:}S.\ e : T) & = & (S \to T) & \\ aty_\sigma(\Lambda X.\, t : T) & = & (\forall X.\, T) & \\ aty_\sigma(v\ \texttt{wrapped}\ T) & = & T & \\ aty_\sigma(a) & = & T & \text{if } \sigma(a) = \{\ldots\} : T \end{array}$$

In addition, every value has a *current type* $cty_\sigma(v)$, whose definition is identical to $aty_\sigma(v)$, except for objects:

$$cty_\sigma(a) \quad = \quad \{l_i : T_i\} \qquad \text{if } \sigma(a) = \{l_i = v_i\} : S \text{ and } T_i = cty_\sigma(v_i)$$

The allocated type of an object is invariant and never changes; the current type of an object changes over time but is more precise in the case of objects.

A *type tag* is a type that can be returned by $aty_\sigma(v)$ or $cty_\sigma(v)$; in particular, it excludes $*$, `like`, and `wrap` types, and type variables.

An evaluation context is:

$$\begin{array}{rcl} C & ::= & \bullet \mid C\ t \mid v\ C \mid C[T] \mid C\ \texttt{is}\ T \mid C\ \texttt{wrap}\ T \\ & & \mid \{\bar{l} = \bar{v}, l = C, \bar{l} = \bar{e}\} : T \mid C.l \mid C.l() \mid C.l := e \mid v.l := C \end{array}$$

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.)

Several rules refer to the function $convert_\sigma(v, T)$, which checks if the value v can be converted to the type T: see Figure 4.

Figure 3: Evaluation Rules

$$
\begin{array}{rcll}
\sigma, C[(\lambda x{:}S.\ t : T)\ v] & \longrightarrow & \sigma, C[t[x := v']\ \texttt{is}\ T] \quad \text{if } v' = \mathit{convert}_\sigma(v, S) & \text{[E-Beta1]} \\
\sigma, C[(w\ \texttt{wrapped}\ (S \to T))\ v] & \longrightarrow & \sigma, C[(w\ (v\ \texttt{wrap}\ S))\ \texttt{wrap}\ T] & \text{[E-Beta2]} \\
\sigma, C[(\Lambda X.\, t : T)[S]] & \longrightarrow & \sigma, C[t[X := S]\ \texttt{is}\ T[X := S]] & \text{[E-Generic1]} \\
\sigma, C[(w\ \texttt{wrapped}\ (\forall X.\, S))[T]] & \longrightarrow & \sigma, C[w[T]\ \texttt{is}\ S[X := T]] & \text{[E-Generic2]} \\
\sigma, C[v\ \texttt{is}\ T] & \longrightarrow & \sigma, C[v] \quad \text{if } \mathit{aty}_\sigma(v) \sim: T & \text{[E-As]} \\
\sigma, C[v\ \texttt{wrap}\ T] & \longrightarrow & \sigma, C[v'] \quad \text{if } v' = \mathit{convert}_\sigma(v, \texttt{wrap}\ T) & \text{[E-Wrap]} \\
\sigma, C[\{l_i = v_i^{i \in 1..n}\} : T] & \longrightarrow & \sigma[a := (\{l_i = v_i'\} : T)], C[a] & \text{[E-Alloc]} \\
 & & \text{where } T = \{l_i : T_i^{i \in 1..n}\},\ v_i' = \mathit{convert}_\sigma(v_i, T_i) & \\
 & & \text{and } a \text{ fresh} & \\
\sigma, C[a.l] & \longrightarrow & \sigma, C[v] \quad \text{if } \sigma(a) = \{l = v, \ldots\} : T & \text{[E-Get1]} \\
\sigma, C[(w\ \texttt{wrapped}\ \{l : T, \ldots\}).l] & \longrightarrow & \sigma, C[(w.l)\ \texttt{wrap}\ T] & \text{[E-Get2]} \\
\sigma, C[a.l()] & \longrightarrow & \sigma, C[(a.l)\ a] & \text{[E-Call]} \\
\sigma, C[a.l := v] & \longrightarrow & \sigma[a, l := v'], C[v] & \text{[E-Assign1]} \\
 & & \text{where if } \sigma(a) = \{\ldots\} : \{l : T, \ldots\} & \\
 & & \text{then } v' = \mathit{convert}_\sigma(v, T) & \\
 & & \text{else } v' = v & \\
\sigma, C[(w\ \texttt{wrapped}\ \{l : T, \ldots\}).l := v] & \longrightarrow & \sigma, C[w.l := (v\ \texttt{wrap}\ T)] & \text{[E-Assign2]}
\end{array}
$$

Figure 4: Dynamic Type Checks

$$
\begin{array}{rcll}
\mathit{convert}_\sigma(v, T) & = & v & \text{if } \mathit{aty}_\sigma(v) \sim: T \\
\mathit{convert}_\sigma(v, \texttt{like}\ T) & = & v & \text{if } \mathit{cty}_\sigma(v) \sim: \texttt{like}\ T \\
\mathit{convert}_\sigma(0, \texttt{Bool}) & = & \texttt{false} & \\
\mathit{convert}_\sigma(n, \texttt{Bool}) & = & \texttt{true} & \text{if } n \neq 0 \\
\mathit{convert}_\sigma(v, \texttt{wrap}\ T) & = & v' & \text{if } v' = \mathit{convert}_\sigma(v, T) \\
\mathit{convert}_\sigma(v, \texttt{wrap}\ T) & = & v\ \texttt{wrapped}\ T & \text{if } \mathit{cty}_\sigma(v) \sim: \texttt{like}\ T
\end{array}
$$

5 Strict Mode Type System

Figure 5: Type Relations

Convertible Types $\boxed{S \prec T}$

$$\frac{S \sim: T}{S \prec T} \quad \text{[Con-Sub]}$$

$$\frac{}{\texttt{Int} \prec \texttt{Bool}} \quad \text{[Con-IntBool]}$$

$$\frac{S \sim: T}{S \prec \texttt{wrap}\ T} \quad \text{[Con-Wrap]}$$

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` $\{f : \texttt{Int}\}$, 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 $T = \texttt{like}\ T'$, 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 6: Type Rules for Strict Mode

Type rules $\boxed{E \vdash t : T}$

$$\frac{(x : (\texttt{like} \mid \texttt{wrap})^* \; T) \in E}{E \vdash x \; : \; T} \quad \text{[T-Var]}$$

$$\frac{}{E \vdash c \; : \; ty(c)} \quad \text{[T-Const]}$$

$$\frac{E, x : S \vdash e \; : \; T' \qquad T' \prec T}{E \vdash (\lambda x{:}S.\; e : T) \; : \; (S \to T)} \quad \text{[T-Fun]}$$

$$\frac{E \vdash t_1 \; : \; (S \to T) \qquad E \vdash t_2 \; : \; S' \qquad S' \prec S}{E \vdash (t_1 \; t_2) \; : \; T} \quad \text{[T-App1]}$$

$$\frac{E \vdash t_1 \; : \; * \qquad E \vdash t_2 \; : \; S'}{E \vdash (t_1 \; t_2) \; : \; *} \quad \text{[T-App2]}$$

$$\frac{E \vdash t \; : \; S}{E \vdash t \; \texttt{is} \; T \; : \; T} \quad \text{[T-As]}$$

$$\frac{E \vdash t \; : \; S}{E \vdash t \; \texttt{wrap} \; T \; : \; T} \quad \text{[T-Wrap]}$$

$$\frac{E \vdash t_i \; : \; S_i \qquad S_i \prec T_i \qquad T = \{l_i : T_i^{i \in 1..n}\}}{E \vdash (\{l_i = t_i^{i \in 1..n}\} : T) \; : \; T} \quad \text{[T-Alloc]}$$

$$\frac{E \vdash e \; : \; \{l : T, \ldots\}}{E \vdash e.l \; : \; T} \quad \text{[T-Get1]}$$

$$\frac{E \vdash e \; : \; *}{E \vdash e.l \; : \; *} \quad \text{[T-Get2]}$$

$$\frac{E \vdash e_1 \; : \; \{l : T, \ldots\} \qquad E \vdash e_2 \; : \; S \qquad S \prec T}{E \vdash e_1.l := e_2 \; : \; S} \quad \text{[T-Set1]}$$

$$\frac{E \vdash e_1 \; : \; * \qquad E \vdash e_2 \; : \; S}{E \vdash e_1.l := e_2 \; : \; S} \quad \text{[T-Set2]}$$

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.

$e ::=$		*Expressions:*
	$\ldots$	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
$v ::=$		*Values:*
	$\ldots$	values mentioned earlier
	`safe` $\lambda x{:}S.\ e : T$	safe abstraction

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

Figure 7 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 1 (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 2 (Soundness) *If $\emptyset \Vdash e : T$, then* `safe` *operation in e can never get stuck.*

Figure 7: Type Rules for Optimization

Type rules $\boxed{E \Vdash t : T}$

$$\frac{(x : T) \in E}{E \Vdash \texttt{safe}\ x : T} \quad \text{[O-Var-Safe]}$$

$$\frac{}{E \Vdash x : *} \quad \text{[O-Var-Unsafe]}$$

$$\frac{}{E \Vdash c : ty(c)} \quad \text{[O-Const]}$$

$$\frac{E, x : S \Vdash e : T' \qquad T' \sim: T}{E \Vdash \texttt{safe}\ (\lambda x{:}S.\ e : T) : (S \to T)} \quad \text{[O-Fun-Safe]}$$

$$\frac{E, x : S \Vdash e : T'}{E \Vdash (\lambda x{:}S.\ e : T) : (S \to T)} \quad \text{[O-Fun-Unsafe]}$$

$$\frac{E \Vdash t_1 : (S \to T) \qquad E \Vdash t_2 : S' \qquad S' \sim: S}{E \Vdash \texttt{safe}\ (t_1\ t_2) : T} \quad \text{[O-App-Safe]}$$

$$\frac{E \Vdash t_1 : S \qquad E \Vdash t_2 : S'}{E \Vdash (t_1\ t_2) : *} \quad \text{[O-App-Unsafe]}$$

$$\frac{E \Vdash t : S}{E \Vdash t\ \texttt{is}\ T : T} \quad \text{[O-As]}$$

$$\frac{E \Vdash t : S}{E \Vdash t\ \texttt{wrap}\ T : T} \quad \text{[O-Wrap]}$$

$$\frac{E \Vdash t_i : S_i \qquad S_i \sim: T_i \qquad T = \{l_i : T_i^{i \in 1..n}\}}{E \Vdash \texttt{safe}\ (\{l_i = t_i^{i \in 1..n}\} : T) : T} \quad \text{[O-Alloc-Safe]}$$

$$\frac{E \Vdash t_i : S_i}{E \Vdash (\{l_i = t_i^{i \in 1..n}\} : T) : T} \quad \text{[O-Alloc-Unsafe]}$$

$$\frac{E \Vdash e : \{l : T, \ldots\}}{E \Vdash \texttt{safe}\ e.l : T} \quad \text{[O-Get-Safe]}$$

$$\frac{E \Vdash e : S}{E \Vdash e.l : *} \quad \text{[O-Get-Unsafe]}$$

$$\frac{E \Vdash e_1 : \{l : T, \ldots\} \qquad E \Vdash e_2 : S \qquad S \sim: T}{E \Vdash \texttt{safe}\ (e_1.l := e_2) : S} \quad \text{[O-Set-Safe]}$$

$$\frac{E \Vdash e_1 : * \qquad E \Vdash e_2 : S}{E \Vdash e_1.l := e_2 : S} \quad \text{[O-Set2]}$$