ValleyScript: Its Like Static Typing

September 12, 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 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.)

Figure 1: Syntax

$e ::=$		*Terms:*
	c	constant
	x	variable
	$\lambda x\!:\!S.\ e : T$	abstraction
	$e\ e$	application
	$\{\bar{l} = \bar{e}\} : T$	object expression
	$e.l$	member selection
	$e.l()$	member invocation
	$e.l := e$	member update
	$e\ \texttt{is}\ T$	dynamic type check
	$e\ \texttt{wrap}\ T$	wrapping a value
$c ::=$		*Constants:*
	n	integer constant
	b	boolean constant
$S, T ::=$		*Types:*
	$\texttt{Int}$	integers
	$\texttt{Bool}$	booleans
	$S \to T$	function type
	$\{\bar{l} : \bar{T}\}$	object types
	$*$	dynamic type
	$\texttt{like}\ T$	like types

2 Evaluation

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

$$
\begin{array}{lll}
v ::= & & \textit{Values:} \\
& c & \text{constant} \\
& \lambda x\!:\!S.\ e : T & \text{abstraction} \\
& v\ \texttt{wrapped}\ T & \text{wrapped value} \\
& a & \text{object address} \\
\\
o ::= & & \textit{Object value:} \\
& \{\bar{l} = \bar{v}\} : T & \text{object value}
\end{array}
$$

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 $ty_\sigma(v)$:

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

An evaluation context is:

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

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 2. 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 $v\ \texttt{is}_\sigma\ T$, which checks if the value v matches the type T: see Figure 3. This judgement relies on two subtype-like relations on types.

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. The type $*$ is a top type.

Lemma 1 *The subtyping judgement is reflexively-transitively closed.*

The judgement $S \sim: T$ (*S is compatible with T*) extends the assignable 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 $\texttt{Int} \sim: *$ and $* \sim: \texttt{Bool}$, but the judgement $\texttt{Int} \sim: \texttt{Bool}$ does not hold. The rule [S-Like] for `like` types switches from checking subtyping to checking compatibility.

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

TBP.

Two types are *indistinguishable* if they are indistinguishable under the subtyping, compatibility, and `is` relations.

Conjecture: The types `like` T and `like like` T are indistinguishable.

Figure 2: Evaluation Rules

$$\sigma, C[(\lambda x\!:\!S.\ t : T)\ v] \longrightarrow \sigma, C[t[x := v]\ \texttt{is}\ T] \quad \text{if } v\ \texttt{is}_\sigma\ S \qquad \text{[E-BETA1]}$$

$$\sigma, C[(w\ \texttt{wrapped}\ (S \to T))\ v] \longrightarrow \sigma, C[(w\ (v\ \texttt{wrap}\ S))\ \texttt{wrap}\ T] \qquad \text{[E-BETA2]}$$

$$\sigma, C[v\ \texttt{is}\ T] \longrightarrow \sigma, C[v] \quad \text{if } v\ \texttt{is}_\sigma\ T \qquad \text{[E-AS]}$$

$$\sigma, C[v\ \texttt{wrap}\ T] \longrightarrow \sigma, C[w] \qquad \text{[E-WRAP]}$$
$$\text{where } w = \begin{cases} v & \text{if } ty_\sigma(v)\ <:\ T \\ v\ \texttt{wrapped}\ T & \text{if } v\ \texttt{is}_\sigma\ T \end{cases}$$

$$\sigma, C[\{l_i = v_i^{i \in 1..n}\} : T] \longrightarrow \sigma[a := (\{l_i = v_i\} : T)], C[a] \qquad \text{[E-ALLOC]}$$
$$\text{where } T = \{l_i : T_i^{i \in 1..n}\}, v_i\ \texttt{is}_\sigma\ T_i \text{ and } a \text{ fresh}$$

$$\sigma, C[a.l] \longrightarrow \sigma, C[v] \quad \text{if } \sigma(a) = \{l = v, \ldots\} : T \qquad \text{[E-GET1]}$$

$$\sigma, C[(w\ \texttt{wrapped}\ \{l : T, \ldots\}).l] \longrightarrow \sigma, C[(w.l)\ \texttt{wrap}\ T] \qquad \text{[E-GET2]}$$

$$\sigma, C[a.l()] \longrightarrow \sigma, C[(a.l)\ a] \qquad \text{[E-CALL]}$$

$$\sigma, C[a.l := v] \longrightarrow \sigma[a, l := v], C[v] \qquad \text{[E-ASSIGN1]}$$
$$\text{where if } \sigma(a) = \{\ldots\} : \{l : T, \ldots\} \text{ then } v\ \texttt{is}_\sigma\ T$$

$$\sigma, C[(w\ \texttt{wrapped}\ \{l : T, \ldots\}).l := v] \longrightarrow \sigma, C[w.l := (v\ \texttt{wrap}\ T)] \qquad \text{[E-ASSIGN2]}$$

Figure 3: Dynamic Type Checks

Dynamic type check $\boxed{v \ \texttt{is}_\sigma \ T}$

$$\frac{ty_\sigma(v) \ <: \ T}{v \ \texttt{is}_\sigma \ T} \quad \text{[Is-Ok]}$$

$$\frac{\begin{array}{c}\sigma(a) = \{l_i = v_i^{i \in 1..n+m}\} : S \\ v_i \ \texttt{is}_\sigma \ \texttt{like} \ T_i \quad \text{for } i \in 1..n\end{array}}{a \ \texttt{is}_\sigma \ \texttt{like} \ \{l_i : T_i^{i \in 1..n}\}} \quad \text{[Is-Obj]}$$

$$\frac{\begin{array}{c}ty_\sigma(v) \sim: T \\ v \text{ not an object address}\end{array}}{v \ \texttt{is}_\sigma \ \texttt{like} \ T} \quad \text{[Is-NonObj]}$$

Subtyping $\boxed{S \ <: \ T}$

$$\frac{}{T \ <: \ T} \quad \text{[S-Refl]}$$

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

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

$$\frac{}{T \ <: \ *} \quad \text{[S-Dyn]}$$

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

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

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

Includes all of the above rules, and also:

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

3 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` $\{f : \texttt{Int}\}$, 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.

Figure 4: Type Rules for Strict Mode

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

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

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

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

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

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

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

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

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

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

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

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

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

4 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
	$\texttt{safe}\ x$	safe variable
	$\texttt{safe}\ \lambda x\!:\!S.\ e : T$	safe abstraction
	$\texttt{safe}\ e\ e$	safe application
	$\texttt{safe}\ \{\bar{l} = \bar{e}\} : T$	safe object allocation
	$\texttt{safe}\ e.l$	safe field selection
	$\texttt{safe}\ e.l := e$	safe field update
$v ::=$		*Values:*
	$\ldots$	values mentioned earlier
	$\texttt{safe}\ \lambda x\!:\!S.\ e : T$	safe abstraction

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

Figure 5 presents *optimization rules*, which verify that `safe` $\ldots$ 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 5: Type Rules for Optimization

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

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

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

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

$$\frac{E, x : S \Vdash e : T' \qquad T' <: T}{E \Vdash \texttt{safe}\ (\lambda x{:}S.\ e : T) : (S \rightarrow T)} \quad \text{[O-FUN-SAFE]}$$

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

$$\frac{E \Vdash t_1 : (S \rightarrow T) \qquad E \Vdash t_2 : S' \qquad S' <: S}{E \Vdash \texttt{safe}\ (t_1\ t_2) : T} \quad \text{[O-APP-SAFE]}$$

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

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

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

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

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

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

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

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

$$\frac{E \Vdash e_1 : * \qquad E \Vdash e_2 : S}{E \Vdash e_1.l := e_2 : S} \quad \text{[O-SET2]}$$

5 Sugar

$$
\begin{array}{rcl}
\lambda x{:}(\texttt{wrap}\ S).\ e : T & = & \lambda x{:}(\texttt{like}\ S).\ \texttt{let}\ x : S = (x\ \texttt{wrap}\ S)\ \texttt{in}\ e : T \\
\lambda x{:}S.\ e : (\texttt{wrap}\ T) & = & \lambda x{:}S.\ (e\ \texttt{wrap}\ T) : T \\
\texttt{wrap}\ (\lambda x{:}S.\ e : T) & = & \lambda x{:}(\texttt{wrap}\ S).\ e : T
\end{array}
$$