ValleyScript: It's Like Static Typing

Cormac Flanagan

December 20, 2007

Abstract

We formalize the ES4 notion of gradual typing for a small language with functions and objects, and with `like`, `wrap`, and dynamic types. We present a "strict mode" verifier for this language. We also present a type-based analysis that statically identifies which run-time type checks are redundant, and prove the soundness of this analysis. This soundness proof provides a crucial sanity check.

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. The language extends the lambda calculus (variables, abstractions, and application) with object allocation, dereference, and update. It also includes `as` expressions, which check that a value has a particular type. Object addresses a_T should not occur in source programs.

Some expression forms (variable references, function definitions and applications, and object allocation, dereference, and update, and cast expressions) are annotated with a *check mark* $c \in \{\circ, \bullet\}$, where $\bullet$ means that that expression has been proven *safe* and that no run-time type check is required. These check marks are inferred/verified by the analysis of Section 4; for now we mostly ignore these check marks, and omit them if they are irrelevant.

The type language includes integers, function types, object types, and the type $*$, which indicates that no static type information is known.

2 Type Relations

The type system is based in part on the usual subtype relation $S < T$. We write $S = T$ to mean that $S < T$ and $T < S$. For technical reasons, function subtyping is *invariant* in the argument position.

In addition, to support a notion of gradual typing, we also have a compatibility relation $S \rightsquigarrow T$, which holds provided that S and T are identical except that T may include $*$ in places where S does not. The compatibility relation is asymmetric since $\texttt{Int} \rightsquigarrow *$ but $* \not\rightsquigarrow \texttt{Int}$, but is transitive.

The *compatible-subtyping* relation $S \lesssim T$ is then the composition of these two relations: $S \lesssim T$ holds if there exists U such that $S < U$ and $U \rightsquigarrow T$.

Lemma 1 *Subtyping is transitive.*

Lemma 2 *Compatibility is transitive.*

Lemma 3 *Subtyping and compatibility commute, that is,* $(< \circ \rightsquigarrow) = (\rightsquigarrow \circ <)$.

PROVEME.

Lemma 4 *Compatible-subtyping is transitive.*

Figure 1: Syntax

$e ::=$		*Terms:*
	n	integer constants
	x^c	variable
	$\lambda^c\, x{:}S.\ e : T$	abstraction
	$(e\ e)^c$	application
	$\{\bar{l} = \bar{e}\}^c : T$	object allocation
	$e.l^c$	member selection
	$e.l^c := e$	member update
	$e\ \texttt{as}^c\ T$	runtime type check
	a_T	object address
$S, T ::=$		*Types:*
	$\texttt{Int}$	integers
	$S \to T$	function type
	$\{\bar{l} : \bar{T}\}$	object types
	$*$	dynamic type

We have:

$$\begin{array}{rcl} (S \to T) & \lessdot\!\!\sim & (* \to *) \\ \{l : T, \ldots\} & \lessdot\!\!\sim & \{l : *\} \\ \{\ldots\} & \lessdot\!\!\sim & \{\} \end{array}$$

3 Evaluation

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

$v ::=$		*Values:*
	n	integer constant
	$\lambda^c\, x{:}S.\ e : T$	abstraction
	a_T	object address with type T

Each object address a is annotated with the type T of the object it points to. A *object store* σ maps object addresses a_T to object values of the form $\{\bar{l} = \bar{v}\}$. Values and object values in the store are *closed* in that they do not contain free program variables x; though they may contain object addresses.

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

$$\begin{array}{rcl} ty(n) & = & \texttt{Int} \\ ty(\lambda^c\, x{:}S.\ e : T) & = & (S \to T) \\ ty(a_T) & = & T \end{array}$$

The allocated type of an object is invariant and independent of the store σ. A *type tag* is a type that can be returned by $ty(v)$; it includes $\texttt{Int}$, function, and object types, but excludes $*$. We use the notation $T.l$ to denote S if $T = \{l : S, \ldots\}$, and to denote $*$ if T is a different object type; in all other cases $T.l$ is undefined.

An evaluation context is:

$$C \quad ::= \quad \bullet \mid (C\ t) \mid (v\ C) \mid C\ \texttt{as}\ T \mid \{\bar{l} = \bar{v}, l = C, \bar{l} = \bar{e}\} : T \mid C.l \mid C.l := e \mid v.l := C$$

Figure 2: Subtyping and Compatibility

Subtyping $\boxed{S < T}$

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

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

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

Compatibility $\boxed{S \rightsquigarrow T}$

$$\frac{}{T \rightsquigarrow T} \quad \text{[Com-Refl]}$$

$$\frac{}{T \rightsquigarrow *} \quad \text{[Com-Dyn]}$$

$$\frac{S_1 \rightsquigarrow T_1 \qquad S_2 \rightsquigarrow T_2}{(S_1 \to S_2) \rightsquigarrow (T_1 \to T_2)} \quad \text{[Com-Arrow]}$$

$$\frac{S_i \rightsquigarrow T_i \qquad \text{for } i \in 1..n}{\{l_i : S_i^{i \in 1..n}\} \rightsquigarrow \{l_i : T_i^{i \in 1..n}\}} \quad \text{[Com-Obj]}$$

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. For now, these rules ignore the check marks, and always perform dynamic type checks by calling the function $convert_\sigma(v, T)$, which checks if the value v can be converted to the type T.

The notation $\sigma[a_T, l := v]$ denotes the store that is identical to σ, except that the l field of the object at address a_T is updated with the value v.

4 Check Optimization

In a traditional statically typed language, the type system both detects errors and eliminates dynamic checks. This presentation separates these two components, primarily so that the check elimination phase can be run even in `standard` mode.

We first extend the operational semantics to omit *convert* checks on operations labelled with the check mark •: see Figure 4. Thus, even though the original rule [E-Beta] can evaluate all applications, [E-Beta-Safe] provides an optimized evaluation rule for a safe application.

We now sketch a type-based analysis that statically verifies that the check mark • is only used in places where the corresponding dynamic type check is redundant: see Figure 5. (It is straightforward to rephrase this process to infer check marks.) These rules rely on the relation $S \Rightarrow T$, which is defined to hold if $S < T$

Figure 3: Operational Semantics

Evaluation Rules

$$\sigma, C[((\lambda^c\, x{:}S.\ t : T)\ v)^d] \longrightarrow \sigma, C[t[x := v']\ \mathtt{as}^c\ T] \quad \text{if } v' = convert_\sigma(v, S) \quad \text{[E-Beta]}$$

$$\sigma, C[v\ \mathtt{as}^c\ T] \longrightarrow \sigma, C[v'] \quad \text{if } v' = convert_\sigma(v, T) \quad \text{[E-As]}$$

$$\sigma, C[\{l_i = v_i^{i\in 1..n}\}^c : T] \longrightarrow \sigma[a_T := (\{l_i = v_i'\})], C[a_T] \quad \text{if } v_i' = convert_\sigma(v_i, T.l_i),\ a_T \text{ fresh},\ T = \{\ldots\} \quad \text{[E-Alloc]}$$

$$\sigma, C[a_T.l^c] \longrightarrow \sigma, C[v] \quad \text{if } \sigma(a_T) = \{l = v, \ldots\} \quad \text{[E-Get]}$$

$$\sigma, C[a_T.l^c := v] \longrightarrow \sigma[a_T, l := v'], C[v] \quad \text{if } v' = convert_\sigma(v, T.l) \quad \text{[E-Assign]}$$

Dynamic Type Checks

$$convert_\sigma(v, T) = v \quad \text{iff} \quad ty(v) \lessdot\!\!\sim T$$

Figure 4: Extended Operational Semantics

Evaluation Rules (in addition to those of Figure 3)

$$\sigma, C[((\lambda^c\, x{:}S.\ t : T)\ v)^\bullet] \longrightarrow \sigma, C[t[x := v]\ \mathtt{as}^c\ T] \quad \text{[E-Beta-Safe]}$$

$$\sigma, C[v\ \mathtt{as}^\bullet\ T] \longrightarrow \sigma, C[v] \quad \text{[E-As-Safe]}$$

$$\sigma, C[\{l_i = v_i^{i\in 1..n}\}^\bullet : T] \longrightarrow \sigma[a_T := \{l_i = v_i\}], C[a_T] \quad a_T \text{ fresh} \quad \text{[E-Alloc-Safe]}$$

$$\sigma, C[a_T.l^\bullet := v] \longrightarrow \sigma[a_T, l := v], C[v] \quad \text{[E-Assign-Safe]}$$

and T is $*$-free.

$$\frac{S < T \qquad T \text{ is } *\text{-free}}{S \Rightarrow T}$$

TODO: example.

The following lemma states that if a value of allocated type V is in a variable of static type S, and is passed to a context with a static requirement T but underlying dynamic requirement U, then if $S \Rightarrow T$ then we know that the value also has type U.

Lemma 5 *If $V \lessdot\!\!\sim S$ and $S \Rightarrow T$ and $U > U'$ and $U' \rightsquigarrow T$ then $V \lessdot\!\!\sim U$.*

Lemma 6 *The $\Rightarrow$ relation is transitive.*

Lemma 7 (No Failure) *For any term e with no $\bullet$ annotations, there exists T such that $\emptyset \Vdash e : T$.*

Proof: By induction on the structure of e.

Lemma 8 (Preservation) *If $\Vdash \sigma, e : ok$ and $\sigma, e \longrightarrow \sigma', e'$ then $\Vdash \sigma', e' : ok$.*

PROVEME

A *unsafe operation* r is a term of the form:

$$r \ ::= \ x^\circ \mid (v\ v)^\circ \mid \{\bar{l} = \bar{v}\}^\circ : T \mid v.l^\circ \mid v.l^\circ := v \mid v\ \mathtt{as}^\circ\ T$$

Lemma 9 (Progress) *If $\Vdash \sigma, e : ok$ and $\sigma, e \not\longrightarrow \sigma', e'$ for any σ' and e' then $e = C[r]$ where r is an unsafe operation.*

PROVEME

Lemma 10 (Determinism) *If* $\Vdash \sigma, e : ok$ *and* $\sigma, e \longrightarrow \sigma_1, e_1$ *and* $\sigma, e \longrightarrow \sigma_2, e_2$ *then (modulo consistent address renamings)* $\sigma_1 = \sigma_2$ *and* $e_1 = e_2$.

PROVEME

5 Strict Mode Type System

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; it provides no guarantees, and the sole purpose of the strict mode type system is to heuristically detect errors at verification time.

The type system is based on the *convertibility relation* $S \prec T$ (shown in Figure 6) to see if an expression of static type S can be passed to a context expecting a type T. The relation $S \prec T$ holds if there exists U such that $S < U$ and $U \sim T$. In turn, the *consistency relation* $U \sim T$ checks if types U and T are identical, where $*$ in either U or T matches any type. Thus, consistency extends the compatibility relation, and it is symmetric but not transitive.

TODO: When do we have guarantees?

6 Extensions

We now extend the language and type language with `like` types, `wrap` types, and implicit int-to-boolean conversions:

$e ::=$	$\ldots$	*Terms:*
	b	boolean constants
	e `wrap` T	wrap operation
$v ::=$	$\ldots$	*Values:*
	b	boolean constants
	v `wrapped` T	wrapped value
$C ::=$	$\ldots$	*Evaluation Contexts:*
	C `wrap` T	wrap operation
$S, T ::=$	$\ldots$	*Types:*
	`Bool`	booleans
	`like` T	like type
	`wrap` T	wrap type

We add the following evaluation rules. Note that when operating on a wrapped value, the check mark c on the operation is irrelevant and so ignored.

$$\sigma, C[v \texttt{ wrap } T] \longrightarrow \sigma, C[v \texttt{ wrapped } T] \quad (T \text{ a fn or obj type}) \quad \text{[E-Wrap]}$$

$$\sigma, C[((w \texttt{ wrapped } (S \to T))\ v)^c] \longrightarrow \sigma, C[(w\ (v \texttt{ wrap } S))^\circ \texttt{ wrap } T] \quad \text{[E-Beta-w]}$$

$$\sigma, C[(w \texttt{ wrapped } \{l : T, \ldots\}).l^c] \longrightarrow \sigma, C[(w.l^\circ) \texttt{ wrap } T] \quad \text{[E-Get-w]}$$

$$\sigma, C[(w \texttt{ wrapped } \{l : T, \ldots\}).l^c := v] \longrightarrow \sigma, C[w.l^\circ := (v \texttt{ wrap } T)] \quad \text{[E-Assign-w]}$$

The allocated types of the additional values is given by:

$$\begin{aligned} ty(b) &= \texttt{Bool} \\ ty(v \texttt{ wrapped } T) &= T \end{aligned}$$

Figure 5: Type Rules for Optimization

Optimization rules for expressions $\boxed{E \Vdash e : T}$

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

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

$$\frac{}{E \Vdash n : Int} \quad \text{[O-Int]}$$

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

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

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

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

$$\frac{E \Vdash t : S \qquad S \Rightarrow T}{E \Vdash t\ \mathtt{as}^{\bullet}\ T : T} \quad \text{[O-As-safe]}$$

$$\frac{E \Vdash t : S}{E \Vdash t\ \mathtt{as}^{\circ}\ T : T} \quad \text{[O-As-Unsafe]}$$

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

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

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

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

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

$$\frac{E \Vdash e_1 : S \qquad E \Vdash e_2 : T}{E \Vdash (e_1.l^{\circ} := e_2) : T} \quad \text{[O-Set-Unsafe]}$$

Optimization rules for states $\boxed{E \Vdash \sigma, e : ok}$

$$\frac{\begin{array}{l}\text{if } \sigma(a_T) = \{l_i = v_i^{i \in 1..m}\} \text{ and } T = \{l_i : T_i^{i \in 1..n}\} \\ \text{then } n \leq m \text{ and } \forall i \in 1..n.\ \emptyset \Vdash v_i : S_i \text{ and } S_i \lessdot\!\!\sim T_i\end{array} \qquad \emptyset \Vdash e : S}{\Vdash \sigma, e : ok} \quad \text{[O-State]}$$

Figure 6: Convertibility and Consistency

Convertibility $\boxed{S \prec T}$

$$\frac{S < U \qquad U \sim T}{S \prec T} \quad [\textsc{Con-Refl}]$$

Consistency $\boxed{S \sim T}$

$$\frac{}{T \sim T} \quad [\textsc{Con-Refl}]$$

$$\frac{}{T \sim *} \quad [\textsc{Con-Dyn}]$$

$$\frac{}{* \sim T} \quad [\textsc{Con-Dyn}]$$

$$\frac{S_1 \sim T_1 \qquad S_2 \sim T_2}{(S_1 \to S_2) \sim (T_1 \to T_2)} \quad [\textsc{Con-Arrow}]$$

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

For subtyping, `like` S is a supertype of S (describes more values) and is covariant. `wrap` T is a subtype of T, since it describes certain kinds of T values.

$$\frac{S < T}{S < \texttt{like}\ T} \quad [\textsc{Sub-Like-Inc}]$$

$$\frac{S < T}{\texttt{like}\ S < \texttt{like}\ T} [\textsc{Sub-Like-Cov}]$$

$$\frac{S < T}{\texttt{wrap}\ S < T} \quad [\textsc{Sub-Wrap}]$$

Figure 7: Type Rules for Strict Mode

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

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

$$\frac{}{E \vdash n : \texttt{Int}} \quad \text{[T-Int]}$$

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

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

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

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

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

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

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

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

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

We extend compatibility and consistency to `like` and `wrap` types.

$$\frac{S \rightsquigarrow T}{\texttt{like}\ S \rightsquigarrow \texttt{like}\ T}\text{[Com-Like]}$$

$$\frac{S \rightsquigarrow T}{\texttt{wrap}\ S \rightsquigarrow \texttt{wrap}\ T}\text{[Com-Like]}$$

$$\frac{S \sim T}{\texttt{like}\ S \sim \texttt{like}\ T}\ \text{[Con-Like]}$$

$$\frac{S \sim T}{\texttt{wrap}\ S \sim \texttt{wrap}\ T}\ \text{[Con-Like]}$$

We extend the convertibility relation to allow int-to-boolean conversions:

$$\frac{}{\texttt{Int} \prec \texttt{Bool}}\text{[Conv-Int-Bool]}$$

We extend *convert* as follows:

$$\begin{array}{rcll} \mathit{convert}_\sigma(v, T) & = & v & \text{if } v\ \texttt{is}_\sigma\ T \\ \mathit{convert}_\sigma(0, \texttt{Bool}) & = & \texttt{false} & \\ \mathit{convert}_\sigma(n, \texttt{Bool}) & = & \texttt{true} & n \neq 0 \\ \mathit{convert}_\sigma(v, \texttt{wrap}\ T) & = & v & \text{if } v\ \texttt{is}_\sigma\ T \\ \mathit{convert}_\sigma(v, \texttt{wrap}\ T) & = & v\ \texttt{wrap}\ T & \text{if } v\ \texttt{is}_\sigma\ \texttt{like}\ T \end{array}$$

The new relation $v\ \texttt{is}_\sigma\ T$ checks if the value v matches the type T:

$$\frac{\mathit{ty}(v) \ll\!\!\rightsquigarrow T}{v\ \texttt{is}_\sigma\ T}\qquad\text{[Is-Ok]}$$

$$\frac{\sigma(a_S) = \{l_i = v_i^{i \in 1..n+m}\} \quad v_i\ \texttt{is}_\sigma\ \texttt{like}\ T_i \ \text{ for } i \in 1..n}{a_S\ \texttt{is}_\sigma\ \texttt{like}\ \{l_i : T_i^{i \in 1..n}\}}\text{[Is-Like]}$$

We add the type checking and optimization rules for the wrap operation and boolean constants:

$$\frac{}{E \Vdash b : \texttt{Bool}}\qquad\text{[O-Bool]}$$

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

$$\frac{}{E \vdash b : \texttt{Bool}}\qquad\text{[T-Bool]}$$

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

We conjecture (but have not proved!) that the resulting system then behaves as intended.