ValleyScript: Its Like Static Typing

Cormac Flanagan

September 27, 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 con-
cepts and to clarify what guarantees can be provided by the verifier in
strict mode. We also formalize Self.

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,
derference, 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.)

The type Self may be used only within an object type T, and refers to
the allocated type of the current object. The type rules for Self are somewhat
subtle, since it may denote any subtype of the object type T.
Figure 1: Syntax

\[
\begin{align*}
\text{Terms:} & \\
\text{\rule{0pt}{2ex} } & \\
\text{\textit{e ::=}} & \\
\text{\textit{c}} & \text{constant} \\
\text{\textit{x}} & \text{variable} \\
\text{\textit{\lambda x:S. e : T}} & \text{abstraction} \\
\text{\textit{e e}} & \text{application} \\
\text{\textit{\{l = \bar{e}\} : T}} & \text{object expression} \\
\text{\textit{e.l}} & \text{member selection} \\
\text{\textit{e.l()}} & \text{member invocation} \\
\text{\textit{e.l := e}} & \text{member update} \\
\text{\textit{e is T}} & \text{dynamic type check} \\
\text{\textit{e wrap T}} & \text{wrapping a value} \\
\end{align*}
\]

\[
\begin{align*}
\text{Constants:} & \\
\text{\textit{\rule{0pt}{2ex} } } & \\
\text{\textit{c ::=}} & \\
\text{\textit{n}} & \text{integer constant} \\
\text{\textit{b}} & \text{boolean constant} \\
\end{align*}
\]

\[
\begin{align*}
\text{Types:} & \\
\text{\textit{\rule{0pt}{2ex} } } & \\
\text{\textit{\textit{S, T ::=}} } & \\
\text{\textit{Int}} & \text{integers} \\
\text{\textit{Bool}} & \text{booleans} \\
\text{\textit{S \rightarrow T}} & \text{function type} \\
\text{\textit{\{l: T\}}} & \text{object types} \\
\text{\textit{* \textit{like T}}} & \text{dynamic type} \\
\text{\textit{Self}}} & \text{like types} \\
\end{align*}
\]
2 Evaluation

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

\[
v ::= \begin{align*}
& c, \text{ constant} \\
& \lambda x : S. \ e : T, \text{ abstraction} \\
& v \ \text{wrapped} \ T, \text{ wrapped value} \\
& a, \text{ object address}
\end{align*}
\]

\[
o ::= \begin{align*}
& \{ \bar{l} = \bar{v} \} : T, \text{ object value}
\end{align*}
\]

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

\[
\begin{align*}
& t_y \sigma(n) = \text{Int} \\
& t_y \sigma(b) = \text{Bool} \\
& t_y \sigma(\lambda x : S. \ e : T) = (S \to T) \\
& t_y \sigma(v \ \text{wrapped} \ T) = T \\
& t_y \sigma(a) = T \quad \text{if } \sigma(a) = \{ \ldots \} : T
\end{align*}
\]

An evaluation context is:

\[
C ::= \begin{align*}
& \bullet \mid \bullet \ t \mid v \ \bullet \mid \bullet \ \text{wrap} \ T \\
& \mid \{ \bar{l} = \bar{v}, l = \bullet, l = \bar{e} \} : T \mid \bullet . l \mid \bullet . l() \mid \bullet . l := e \mid v . l := \bullet
\end{align*}
\]

A \textit{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.

In general, states never contain types that have free occurrences of \textit{Self}. Instead, whenever we pull a component type \( S \) out of an object with allocated type \( T = \{ l : S, \ldots \} \), we replace all occurrences of \textit{Self} in \( S \) with the type \( T \).

Several rules refer to the judgement \( v \ \text{i}s_{\sigma} \ T \), which checks if the value \( v \) matches the type \( T \); see Figure 3. The type \textit{Self} should not occur free in \( T \). This judgement relies on two subtype-like relations on types.

The judgement \( S <: T \) (\( S \) is a \textit{subtype} of \( T \)) checks if every value of type \( S \) can be assigned to a variable of type \( T \). The type \textit{Self} can occur free in both \( S \) and \( T \). The type \( * \) is a top type.

The judgement \( S \sim T \) (\( S \) is \textit{compatible} with \( T \)) extends the subtyping 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 \([S-Like]\) for \textit{like} types switches from checking subtyping to checking compatibility.
Lemma 1 The subtyping judgement is reflexively-transitively closed.

Lemma 2 (Preservation under subtyping) If \( v \ is_\sigma S \) and \( S <: T \) then \( v \ is_\sigma T \).

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

\[
\begin{align*}
\sigma, C[(\lambda x : S. t : T) \; v] & \longrightarrow \sigma, C[t[x := v] \; is \; T] \quad \text{if } v \ is_\sigma S \quad \text{[E-Beta1]} \\
\sigma, C[(w \; \text{wrapped} \; (S \rightarrow T)) \; v] & \longrightarrow \sigma, C[(w \; (v \; \text{wrap} \; S)) \; \text{wrap} \; T] \quad \text{[E-Beta2]} \\
\sigma, C[v \; \text{is} \; T] & \longrightarrow \sigma, C[v] \quad \text{if } v \ is_\sigma T \quad \text{[E-As]} \\
\sigma, C[v \; \text{wrap} \; T] & \longrightarrow \sigma, C[w] \quad \text{[E-Wrap]}
\end{align*}
\]

where \( w = \begin{cases} v & \text{if } \text{type}(v) <: T \\ v \; \text{wrapped} \; T & \text{if } v \ is_\sigma T \end{cases} \)

\[
\begin{align*}
\sigma, C[(l_i = v^{i \in 1..n}) : T] & \longrightarrow \sigma[a := \{(l_i = v_i) : T\}, C[a] \quad \text{[E-Alloc]}
\end{align*}
\]

where \( T = \{l_i : T^{i \in 1..n}\} \)

and \( v_i \ is_\sigma T_i[\text{Self} := T] \)

and \( a \) fresh

\[
\sigma, C[a.l] & \longrightarrow \sigma, C[v] \quad \text{if } \sigma(a) = \{l = v, \ldots\} : T \quad \text{[E-Get1]} \\
\sigma, C[(w \; \text{wrapped} \; T).l] & \longrightarrow \sigma, C[(w.l) \; \text{wrap} \; S[\text{Self} := T]] \quad \text{[E-Get2]}
\]

where \( T = \{l : S, \ldots\} \)

\[
\sigma, C[a.l()] & \longrightarrow \sigma, C[(a.l) \; a] \quad \text{[E-Call]} \\
\sigma, C[a.l := v] & \longrightarrow \sigma[a, l := v, C[v] \quad \text{[E-Assign1]}
\end{align*}
\]

where \( \sigma(a) = \{\ldots\} : T \)

and if \( T = \{l : S, \ldots\} \)

then \( v \ is_\sigma S[\text{Self} := T] \)

\[
\begin{align*}
\sigma, C[(w \; \text{wrapped} \; T).l := v] & \longrightarrow \sigma, C[w.l := (v \; \text{wrap} \; S[\text{Self} := T])] \quad \text{[E-Assign2]}
\end{align*}
\]

where \( T = \{l : S, \ldots\} \)
Figure 3: Relations on Types

Dynamic type check

\[ ty_\sigma(v) <: T \]
\[ v \ is_\sigma T \]

\[ \sigma(a) = \{l_i = v_i^{i \in 1..n+m} : S \} \]
\[ v_i \ is_\sigma \ like \ T_i \ for \ i \in 1..n \]
\[ a \ is_\sigma \ like \ \{l_i : T_i^{i \in 1..n}\} \]

\[ ty_\sigma(v) \sim: T \]
\[ v \ not \ an \ object \ address \]
\[ v \ is_\sigma \ like \ T \]

Subtyping

\[ T <: T \]

\[ T_1 <: S_1 \]
\[ S_2 <: T_2 \]
\[ (S_1 \rightarrow S_2) <: (T_1 \rightarrow T_2) \]

\[ S_i <: T_i \]
\[ S_i <: T_i \ for \ i \in 1..n \]
\[ \{l_i : S_i^{i \in 1..n+m}\} <: \{l_i : T_i^{i \in 1..n}\} \]

\[ T <: * \]

\[ S \sim: T \]
\[ S <: \ like \ T \]

\[ S <: T \]
\[ like \ S <: \ like \ T \]

Compatible Types

\[ S \sim: T \]

Includes all of the above rules, and also:

\[ * \sim: T \]
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 \( \{ f : \text{Int} \} \), then \( x.f \) could actually return a value of any type. Nevertheless, \( x.f \) would be expected to produce values of type \( \text{Int} \), and so the expression \( x.f.g \) would yield a compile-time type error. Thus, we say that \( x.f \) has type \( \text{Int} \), 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. However, this intent can be still used to detect type errors at compile time.
Figure 4: Type Rules for Strict Mode

Type rules

\[ E \vdash t : T \]

\[ (x : \text{like}^* T) \in E \]

\[ E \vdash x : T \]  \hspace{1cm} \text{[T-Var]} 

\[ E \vdash e : t y(e) \]  \hspace{1cm} \text{[T-Const]} 

\[ E, x : S \vdash e : T' \quad T' \ll T \]

\[ E \vdash (\lambda x : S. e : T) : (S \to T) \]  \hspace{1cm} \text{[T-Fun]} 

\[ E \vdash t_1 : (S \to T) \quad E \vdash t_2 : S' \quad S' \ll S \]

\[ E \vdash (t_1 \ t_2) : T \]  \hspace{1cm} \text{[T-App1]} 

\[ E \vdash t_1 : * \quad E \vdash t_2 : S' \]

\[ E \vdash (t_1 \ t_2) : * \]  \hspace{1cm} \text{[T-App2]} 

\[ E \vdash t : S \]

\[ E \vdash \text{isT} : T \]  \hspace{1cm} \text{[T-As]} 

\[ E \vdash t : S \]

\[ E \vdash \text{wrapT} : T \]  \hspace{1cm} \text{[T-Wrap]} 

\[ E \vdash t_i : S_i \quad S_i \ll T \]

\[ E \vdash \{ \{ l_i = T_i \ in \ 1 \ldots n \} \} : T \]  \hspace{1cm} \text{[T-Alloc]} 

\[ E \vdash e : T \quad T = \{ l_i = T_i \ in \ 1 \ldots n \} \]

\[ E \vdash e.l : S \[ \text{Self} := T_i, \text{Self}^{-} := ..T.. \] \]

\[ E \vdash e : * \]

\[ E \vdash e.l : * \]  \hspace{1cm} \text{[T-Get1]} 

\[ E \vdash e : T \quad T = \{ l : \text{Self} \to S_i, \ldots \} \]

\[ E \vdash e.l() : S \[ \text{Self}^{+} := T, \text{Self}^{-} := ..T.. \] \]

\[ E \vdash e : * \]

\[ E \vdash e.l() : * \]  \hspace{1cm} \text{[T-Get2]} 

\[ E \vdash e_1 : T \quad T = \{ l : S_i, \ldots \} \quad E \vdash e_2 : S' \quad S' \ll T \]

\[ E \vdash e_1.l := e_2 : S \]  \hspace{1cm} \text{[T-Call1]} 

\[ E \vdash e_1 : * \quad E \vdash e_2 : S \]

\[ E \vdash e_1.l := e_2 : S \]  \hspace{1cm} \text{[T-Call2]} 

\[ E \vdash e_1 : T \quad T = \{ l : S_i, \ldots \} \quad E \vdash e_2 : S' \quad S' \ll T \]

\[ E \vdash e_1.l := e_2 : S \]  \hspace{1cm} \text{[T-Set1]} 

\[ E \vdash e_1 : * \quad E \vdash e_2 : S \]

\[ E \vdash e_1.l := e_2 : S \]  \hspace{1cm} \text{[T-Set2]}
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.

\[
\begin{align*}
e & ::= \\
& \quad \ldots \\
& \quad \text{expressions mentioned earlier} \\
safe \ x & \quad \text{safe variable} \\
safe \ \lambda x : S. \ e : T & \quad \text{safe abstraction} \\
safe \ e \ e & \quad \text{safe application} \\
safe \ \{ l = \bar{e} \} : T & \quad \text{safe object allocation} \\
safe \ e.l & \quad \text{safe member selection} \\
safe \ e.l() & \quad \text{safe member invocation} \\
safe \ e.l := e & \quad \text{safe member update} \\
\end{align*}
\]

\[
\begin{align*}
v & ::= \\
& \quad \ldots \\
& \quad \text{values mentioned earlier} \\
safe \ \lambda x : S. \ e : T & \quad \text{safe abstraction} \\
\end{align*}
\]

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.

Some rules, such as \([O-Get-Safe]\), replace contravariant occurrences of \texttt{Self} with a minimal type \(\bot\), which is guaranteed to always be a subtype of the current object type. Thus, extracting a field (whose type has no contravariant occurrences of \texttt{Self}) is safe; extracting a method (a function whose argument type is \texttt{Self}) may also be safe, but the static argument type of the resulting function is \(\bot\), so invoking that function is always unsafe, and its argument must be checked dynamically. Via \([O-Set-Safe]\), updating a member is in general safe only if its type does not contain covariant occurrences of \texttt{Self}.

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

\[
\begin{array}{ll}
\text{Type rules} & E \vdash t : T \\
(x : T) \in E & \quad \text{E-Var-Safe} \\
E \vdash \text{safe } x : T & \quad \text{E-Var-Unsafe} \\
E \vdash c : ty(c) & \quad \text{E-Const} \\
E, x : S \vdash e : T & \quad \text{E-Fun-Safe} \\
E, x : S \vdash e : T' & \quad \text{E-Fun-Unsafe} \\
E \vdash t_1 : (S' \rightarrow T) & \quad \text{E-App-Safe} \\
E \vdash t_2 : S' & \quad \text{E-App-Unsafe} \\
E \vdash \text{safe } \{l_i = t \in 1..n\} : T & \quad \text{E-Alloc-Safe} \\
E \vdash \text{safe } e.l : S & \quad \text{E-Get-Safe} \\
E \vdash e : S & \quad \text{E-Get-Unsafe} \\
E \vdash e.l() : S & \quad \text{E-Call-Safe} \\
E \vdash e : S & \quad \text{E-Call-Unsafe} \\
E \vdash e_1 : S & \quad \text{E-Set-Safe} \\
E \vdash e_2 : S' & \quad \text{E-Set2} \\
\end{array}
\]
5 Sugar

\[
\lambda x : (\text{wrap } S) . \ v :: \ T = \ \lambda x : (\text{like } S) . \ \text{let } x : S = (x \text{ wrap } S) \ \text{in } \ v :: \ T
\]

\[
\lambda x : S . \ v :: (\text{wrap } T) = \lambda x : S . \ (v \text{ wrap } T) :: T
\]

\[
\text{wrap } (\lambda x : S . \ v :: T) = \lambda x : (\text{wrap } S) . \ v :: T
\]