Checks and Balances - Constraint Solving without Surprises in Object-Constraint Programming Languages: Full Formal Development

Tim Felgentreff, Todd Millstein, Alan Borning and Robert Hirschfeld

VPRI Technical Report TR-2015-001
1 Babelsberg/UID and Two Key Theorems

This technical report is intended to accompany the conference paper “Checks and Balances – Constraint Solving without Surprises in Object-Constraint Programming Languages,” and presents the full formalism and proofs for two theorems that capture key properties of the Babelsberg/UID core language. These theorems are stated (without the proofs) in Section 3.3 of that paper.

1.1 Syntax

We use the following syntax for Babelsberg/UID:

\[
\text{Statement} \quad s ::= \text{skip} | \text{L := e} | \text{x := new o} | \text{always C} | \text{once C} \\
\quad \quad \quad | \text{s;s} | \text{if e then s else s} | \text{while e do s}
\]

\[
\text{Constraint} \quad C ::= \rho e | C \land C
\]

\[
\text{Expression} \quad e ::= v | \text{L} | e \oplus e | \text{L=}L | D
\]

\[
\text{Object Literal} \quad o ::= \{l_1:e_1,...,l_n:e_n\}
\]

\[
\text{L-Value} \quad L ::= x | L.l
\]

\[
\text{Constant} \quad c ::= \text{true} | \text{false} | \text{nil} | \text{base type constants}
\]

\[
\text{Variable} \quad x ::= \text{variable names}
\]

\[
\text{Label} \quad l ::= \text{record label names}
\]

\[
\text{Reference} \quad r ::= \text{references to heap records}
\]

\[
\text{Dereference} \quad D ::= \mathbb{H}(e)
\]

\[
\text{Value} \quad v ::= c | r
\]

Metavariable \(c\) ranges over the \text{nil} value, booleans, and primitive type constants. A finite set of operators on primitives is ranged over by \(\oplus\). We assume that \(\oplus\) includes an equality operator for each primitive type; for convenience we use the symbol \(=\) to denote each of these operators. We also assume it includes an operator \(\land\) for boolean conjunction. The operator \(\text{=}\) tests for identity — for primitive values this behaves the same as \(=\). The symbol \(\rho\) ranges over constraint priorities and is assumed to include a bottom element \text{weak} and a top element \text{required}. The syntax requires the priority to be explicit; for simplicity we sometimes omit it in the rules and assume a \text{required} priority.
In the syntax, we treat $\mathsf{H}$ as a keyword used for dereferencing. Source programs will not use expressions of the form $\mathsf{H}(e)$, but they are introduced as part of constraints given to the solver, which we assume will treat $\mathsf{H}$ as an uninterpreted function. We also assume that the solver supports records and record equality, which we also denote with the = operator.

### 1.2 Operational Semantics

The semantics includes an environment $\mathsf{E}$ and a heap $\mathsf{H}$. The former is a function that maps variable names to values, while the latter is a function that maps mutable references to “objects” of the form $\{l_1:v_1,\ldots,l_n:v_n\}$. When convenient, we also treat both $\mathsf{E}$ and $\mathsf{H}$ as a set of pairs $(\{x,v\},\ldots)$ and $(\{r,o\},\ldots)$, respectively. The currently active value constraints are kept as a compound constraint $\mathsf{C}$; identity constraints are kept as a single conjunction referred to as $\mathsf{I}$.

$\mathsf{E};\mathsf{H} \vdash e \downarrow v$

“Expression $e$ evaluates to value $v$ in the context of environment $\mathsf{E}$ and heap $\mathsf{H}.”

The rules for evaluation are mostly as expected in an imperative language. We do not give rules for expressions of the form $\mathsf{H}(e)$, because they are not meant to appear in source. For each operator $\oplus$ in the language we assume the existence of a corresponding semantic function denoted $\mathsf{J}_\oplus$.

\[
\begin{align*}
\mathsf{E};\mathsf{H} & \vdash c \downarrow c & (\mathsf{E-Const}) \\
\mathsf{E} & \vdash x = v \\
\mathsf{E};\mathsf{H} & \vdash x \downarrow v & (\mathsf{E-Var}) \\
\mathsf{E};\mathsf{H} & \vdash L \downarrow r \\
\mathsf{H}(r) & = \{l_1:v_1,\ldots,l_n:v_n\} & 1 \leq i \leq n \\
\mathsf{E};\mathsf{H} & \vdash L.1_i \downarrow v_i & (\mathsf{E-Field}) \\
\mathsf{E};\mathsf{H} & \vdash r \downarrow r & (\mathsf{E-Ref}) \\
\mathsf{E};\mathsf{H} & \vdash e_1 \downarrow v_1 \\
\mathsf{E};\mathsf{H} & \vdash e_2 \downarrow v_2 \\
\mathsf{v}_1 & \mathbb{J}_\oplus v_2 = v & (\mathsf{E-Op}) \\
\mathsf{E};\mathsf{H} & \vdash e_1 \oplus e_2 \downarrow v \\
\mathsf{E};\mathsf{H} & \vdash L_1 \downarrow v \\
\mathsf{E};\mathsf{H} & \vdash L_2 \downarrow v \\
\mathsf{E};\mathsf{H} & \vdash L_1 = L_2 \downarrow \mathsf{true} & (\mathsf{E-IdentityTrue}) \\
\mathsf{E};\mathsf{H} & \vdash L_1 \downarrow v_1 \\
\mathsf{E};\mathsf{H} & \vdash L_2 \downarrow v_2 \\
v_1 & \neq v_2 & (\mathsf{E-IdentityFalse}) \\
\mathsf{E};\mathsf{H} & \vdash e : T \\
\mathsf{E};\mathsf{H} & \vdash C \\
\end{align*}
\]

“Expression $e$ has type $T$ in the context of environment $\mathsf{E}$ and heap $\mathsf{H}.”

“Constraint $C$ is well formed in the context of environment $\mathsf{E}$ and heap $\mathsf{H}.”
We use a notion of typechecking to prevent undesirable non-determinism in constraints. Specifically, we want constraint solving to preserve the structure of the values of variables, changing only the underlying primitive data as part of a solution. We formalize our notion of structure through a simple syntax of types:

\[
\text{Type } T ::= \text{PrimitiveType} \mid \{l_1:T_1, \ldots, l_n:T_n\}
\]

The typechecking rules are mostly standard. We check expressions dynamically just before constraint solving, so we typecheck in the context of a runtime environment. Note that we do not include type rules for identities. This ensures that constraints involving them do not typecheck, so identity checks cannot occur in ordinary constraints.

\[
\begin{align*}
E;H \vdash c : \text{PrimitiveType} & \quad \text{(T-CONST)} \\
H(r) = \{l_1:v_1, \ldots, l_n:v_n\} & \quad E;H \vdash v_1 : T_1 \ldots E;H \vdash v_n : T_n & \quad \text{(T-REF)} \\
E;H \vdash x : T & \quad \text{(T-VAR)} \\
E;H \vdash L : \{l_1:T_1, \ldots, l_n:T_n\} & \quad 1 \leq i \leq n & \quad \text{(T-FIELD)} \\
E;H \vdash e_1 : \text{PrimitiveType} & \quad E;H \vdash e_2 : \text{PrimitiveType} & \quad \text{(T-OP)} \\
E;H \vdash e_1 \oplus e_2 : \text{PrimitiveType} & \quad \text{(T-PRIORITY)} \\
E;H \vdash e : T & \quad \text{(T-CONST)
} \\
E;H \vdash C_1 & \quad E;H \vdash C_2 & \quad \text{(T-CONJUNCTION)}
\end{align*}
\]

\[E;H \models C\]

This judgment represents a call to the constraint solver, which we treat as a black box. The proposition \(E;H \models C\) denotes that environment \(E\) and heap \(H\) are an optimal solution to the constraint \(C\), according to the solver’s semantics.

We assume several well-formedness properties about a solution \(E\) and \(H\) to constraints \(C\):

- any object reference appearing in the range of \(E\) also appears in the domain of \(H\)
- any object reference appearing in the range of \(H\) also appears in the domain of \(H\)
- for all variables \(x\) in the domain of \(E\) there is some type \(T\) such that \(E;H \vdash x : T\)
- \(E;H \vdash C\)

\[\text{stay}(x=v, \rho) = C\]
As in Kaleidoscope, the semantics ensure that each variable has a stay constraint to keep it at its current value, if possible. The stay rules take a priority as a parameter. When solving value constraints, this priority is set to \texttt{required}, to ensure that the structures of objects and the relationship between l-values and object references cannot change. When solving identity constraints as part of executing an assignment statement, the priority is set to \texttt{weak} to allow structural changes.

To properly account for the heap in the constraint solver, we employ an uninterpreted function \( H \) that maps references to objects (i.e., records). The rules below employ this function in order to define stay constraints for references.

\[ \begin{align*}
\text{stay}(x=c, \rho) &= \texttt{weak} \ x=c & \text{(StayConst)} \\
\text{stay}(x=r, \rho) &= \rho \ x=r & \text{(StayRef)} \\
\text{stay}(r = \{l_1:v_1, \ldots, l_n:v_n\}, \rho) &= \texttt{(required)} H(r) = \{l_1:x_1, \ldots, l_n:x_n\} \wedge C_1 \wedge \cdots \wedge C_n & \text{(StayObject)} \\
\text{stay}(E, \rho) &= C_1 \wedge \cdots \wedge C_n & \text{(StayEnv)} \\
\text{stay}(H, \rho) &= C_1 \wedge \cdots \wedge C_n & \text{(StayHeap)}
\end{align*} \]

These judgments are another form of stay constraints that ensure that the “prefix” \( L \) of an l-value \( L.l \) is unchanged; this is necessary to ensure that updates to the value of \( L.l \) are deterministic.

\[ \begin{align*}
\text{stayPrefix}(E, H, L) &= C & \text{(StayPrefixVar)} \\
\text{stayPrefix}(E, H, I) &= C & \text{(StayPrefixIdent)}
\end{align*} \]
We use these judgments to translate a constraint into a constraint suitable for the solver. Specifically, each occurrence of an expression of the form \(L.l\), where \(L\) refers to a heap reference \(r\), is translated into \(H(L).l\) (recursively, as required), and each occurrence of the identity operator \(==\) is replaced by ordinary equality. We do not give these rules, because they are straightforward.

\[
\text{solve}(E, H, C, \rho) = E'; H'
\]

“This solving constraint \(C\) in the context of \(E\) and \(H\) using stay constraints with priority \(\rho\) produces the new environment and heap \(E'\) and \(H\).”

This judgment represents one phase of constraint solving – either solving “value” constraints or identity constraints.

\[
\begin{align*}
\text{stay}(E, \rho) &= C_E \\
\text{stay}(H, \rho) &= C_H \\
E; H &\vdash C \leadsto C' \\
\text{solve}(E, H, C_L \land C_I \land L=v \land I, \text{weak}) &= E'; H' \\
\text{solve}(E', H', C_L \land L=v, \text{required}) &= E''; H''
\end{align*}
\]

\((\text{Solve})\)

“A execution starting from configuration \(<E|H|C|I|s>\) ends in state \(<E'|H'|C'|I'>\).”

A “configuration” defining the state of an execution includes a concrete context, represented by the environment and heap, a symbolic context, represented by the constraint and identity constraint stores, and a statement to be executed. The environment, heap, and statement are standard, while the constraint stores are not part of the state of a computation in most languages. Intuitively, the environment and heap come from constraint solving during the evaluation of the immediately preceding statement, and the constraint records the always constraints that have been declared so far during execution. Note that our execution implicitly gets stuck if the solver cannot produce a model.

The rule below describes the semantics of assignments. We employ a two-phase process. First the identity constraints are solved in the context of the new assignment. This phase propagates the effect of the assignment through the identities, potentially changing the structures of objects as well as the relationships among objects in the environment and heap. In the second phase, the value constraints are typechecked against the new environment and heap resulting from the first phase. If they are well typed, then we proceed to solve them. This phase can change the values of primitives but will not modify the structure of any object.

Implicitly this rule gets stuck if either a) the identity constraints cannot be solved, b) the value constraints do not typecheck, or c) the value constraints cannot be solved. A practical implementation would add explicit exceptions for these cases that the programmer could handle.

\[
\begin{align*}
E; H \vdash e \downarrow v &\quad \text{stayPrefix}(E, H, L) = C_L \\
\text{solve}(E, H, C_L \land C_I \land L=v \land I, \text{weak}) &= E'; H' \\
\text{solve}(E', H', C_L \land L=v, \text{required}) &= E''; H''
\end{align*}
\]

\((\text{S-Asgn})\)

The next rule describes the semantics of object creation, which is straightforward. For simplicity we require a new object to be initially assigned to a fresh variable, but this is no loss of expressiveness.
The next two rules describe the semantics of identity constraints. The rules require that identity constraint already be satisfied when it is asserted; hence the environment and heap are unchanged.

\[
\begin{align*}
E;H \vdash L_0 \downarrow v & \quad E;H \vdash L_1 \downarrow v \\
\langle E|H|C|I \mid \text{once } L_0 == L_1 \rangle & \rightarrow \langle E'|H'|C|I' \rangle
\end{align*}
\]  

\[
\begin{align*}
\langle E|H|C|I \mid \text{once } C_0 \rangle & \rightarrow \langle E'|H'|C|I' \rangle \\
\langle E|H|C|I \mid \text{always } C_0 \rangle & \rightarrow \langle E'|H'|C'|I' \rangle
\end{align*}
\]  

The following two rules describe the semantics of value constraints. Recall that these constraints cannot contain identity constraints in them (because identity constraints do not typecheck). As we show later, solving value constraints cannot change the structure of any objects on the environment and heap.

\[
\begin{align*}
E;H \vdash C_0 \quad \text{solve}(E, H, C \land C_0, \text{required}) &= E';H' \\
\langle E|H|C|I \mid \text{once } C_0 \rangle & \rightarrow \langle E'|H'|C|I' \rangle
\end{align*}
\]  

\[
\begin{align*}
\langle E|H|C|I \mid \text{always } C_0 \rangle & \rightarrow \langle E'|H'|C'|I' \rangle
\end{align*}
\]  

The remaining rules are standard for imperative languages, only augmented with constraint stores, and are only given for completeness.

\[
\begin{align*}
\langle E|H|C|I \mid \text{skip} \rangle & \rightarrow \langle E|H|C|I \rangle
\end{align*}
\]  

\[
\begin{align*}
\langle E|H|C|I \mid s_1 \rangle & \rightarrow \langle E'|H'|C'|I' \rangle \\
\langle E'|H'|C'|I' \mid s_2 \rangle & \rightarrow \langle E''|H''|C''|I'' \rangle \\
\langle E|H|C|I \mid s_1; s_2 \rangle & \rightarrow \langle E''|H''|C''|I'' \rangle
\end{align*}
\]  

\[
\begin{align*}
E;H \vdash e \downarrow \text{true} & \quad \langle E|H|C|I \mid s_1 \rangle \rightarrow \langle E'|H'|C'|I' \rangle \\
\langle E|H|C|I \mid \text{if } \text{then } s_1 \text{ else } s_2 \rangle & \rightarrow \langle E'|H'|C'|I' \rangle
\end{align*}
\]  

\[
\begin{align*}
E;H \vdash e \downarrow \text{false} & \quad \langle E|H|C|I \mid s_2 \rangle \rightarrow \langle E'|H'|C'|I' \rangle \\
\langle E|H|C|I \mid \text{if } \text{then } s_1 \text{ else } s_2 \rangle & \rightarrow \langle E'|H'|C'|I' \rangle
\end{align*}
\]  

\[
\begin{align*}
E;H \vdash e \downarrow \text{true} & \quad \langle E|C|H|I \mid s \rangle \rightarrow \langle E'|H'|C'|I' \rangle \\
\langle E'|H'|C'|I' \mid \text{while } e \text{ do } s \rangle & \rightarrow \langle E''|H''|C''|I'' \rangle
\end{align*}
\]  

\[
\begin{align*}
\langle E|H|C|I \mid \text{while } e \text{ do } s \rangle & \rightarrow \langle E''|H''|C''|I'' \rangle
\end{align*}
\]
1.3 Properties

Here we state and prove two key theorems about our formalism.

We assume that a given configuration \( E|H|C|I \) is \emph{well formed}, meaning that it satisfies these sanity conditions:

- any object reference appearing in the range of \( E \) also appears in the domain of \( H \)
- any object reference appearing in the range of \( H \) also appears in the domain of \( H \)
- for all variables \( x \) in the domain of \( E \) there is some type \( T \) such that \( E;H \vdash x : T \)
- \( E;H \vdash C \)
- \( E;H \vdash I \downarrow \text{true} \)

Well-formedness follows from the assumptions made on the constraint solver described earlier.

The first theorem formalizes the idea that any solution to a value constraint preserves the structures of the objects on the environment and heap:

\textbf{Theorem 1} (Structure Preservation) If \( \langle E|H|C|I|s \rangle \rightarrow \langle E'|H'|C'|I'|s' \rangle \) and \( s \) either has the form \( \text{once} \) \( C_0 \) or always \( C_0 \) and \( E;H \vdash C_0 \) and \( E;H \vdash L : T \), then \( E';H' \vdash L : T \).

\textbf{Proof.} If \( s \) has the form \( \text{once} \) \( C_0 \) then the result follows by Lemma \[1\]. If \( s \) has the form \( \text{always} \) \( C_0 \), then since \( \langle E|H|C|I|s \rangle \rightarrow \langle E'|H'|C'|I'|s' \rangle \) and \( C_0 \) is not an identity test, by rule \( \text{S-Always} \) we have \( \langle E|H|C|I|\text{once} \ C_0 \rangle \rightarrow \langle E'|H'|C'|I'| \rangle \). Then again the result follows from Lemma \[1\]. \( \square \)

\textbf{Lemma 1} If \( \langle E|H|C|I|\text{once} \ C_0 \rangle \rightarrow \langle E'|H'|C'|I'| \rangle \) and \( E;H \vdash C_0 \) and \( E;H \vdash L : T \), then \( E';H' \vdash L : T \).

\textbf{Proof.} Since \( \langle E|H|C|I|\text{once} \ C_0 \rangle \rightarrow \langle E'|H'|C'|I'| \rangle \) and \( C_0 \) is not an identity test, by \( \text{S-Once} \) we have that \( E;H \vdash C_0 \) and \( \text{solve}(E, H, C \land C_0, \text{required}) = E';H' \). By the assumption of well-formedness we have \( E;H \vdash C \) so by \( \text{T-Conjunction} \) also \( E;H \vdash C \land C_0 \). Therefore the result follows from Lemma \[2\]. \( \square \)

\textbf{Lemma 2} If \( E;H \vdash C \) and \( \text{solve}(E, H, C, \text{required}) = E';H' \) and \( E;H \vdash L : T \), then \( E';H' \vdash L : T \).

\textbf{Proof.} Since \( \text{solve}(E, H, C, \text{required}) = E';H' \) by \( \text{Solve} \) we have that \( \text{stay}(E, \text{required}) = C_{E_h} \) and \( \text{stay}(H, \text{required}) = C_{H_h} \) and \( E;H \vdash C \Rightarrow C' \) and \( E';H' \models (C' \land C_{E_h} \land C_{H_h}) \). We prove this lemma by structural induction on \( T \).

By Lemma \[3\] there exists a value \( v \) such that \( E;H \vdash L \Downarrow v \) and \( E;H \vdash v : T \). Then by Lemma \[4\] there exists a value \( v' \) such that \( E';H' \vdash L \Downarrow v' \). Furthermore, if \( v \) is an object reference \( r \), then also \( v' = r \). Case analysis on the form of \( v' \):

- Case \( v' \) is a constant \( c' \). Then also \( v \) is a constant \( c \). Since \( E;H \vdash v : T \), the last rule in this derivation must be \( \text{T-Const} \) so \( T \) is \( \text{PrimitiveType} \) and the result follows from \( \text{T-Const} \).

- Case \( v' \) is a reference \( r' \). Since \( E';H' \vdash L \Downarrow v' \) by Lemma \[5\] there is a type \( T' \) such that \( E';H' \vdash L : T' \) and \( E';H' \vdash r' : T' \). So it suffices to show that \( T = T' \). Case analysis on the form of \( v \):
Lemma 3 If \( E;H \vdash L : T \), then there exists a value \( v \) such that \( E;H \vdash L \downarrow v \) and \( E;H \vdash v : T \).
\textbf{Proof.} By structural induction on \(L:\)

- Case \(L\) is a variable \(x\): Then by T-VAR we have that \(E(x) = v\) and \(E ; H \vdash x : T\). Finally by E-VAR we have \(E ; H \vdash x \downarrow v\).
- Case \(L\) has the form \(L' . l_i\): By T-FIELD we have that \(E ; H \vdash L' : \{ l_1 : T_1, \ldots, l_n : T_n \}\) and \(1 \leq i \leq n\) and \(T = T_i\). By induction there exists a value \(v'\) such that \(E ; H \vdash L' \downarrow v'\) and \(E ; H \vdash v' : \{ l_1 : T_1, \ldots, l_n : T_n \}\).

  Case analysis of the derivation of \(E ; H \vdash v' : \{ l_1 : T_1, \ldots, l_n : T_n \}:\)
  - Case T-CONST: Then \(\{ l_1 : T_1, \ldots, l_n : T_n \} = \text{PrimitiveType}\) and we have a contradiction.
  - Case T-REF: Then we have \(v' = r\) and \(H(r) = \{ l_1 : v_1, \ldots, l_n : v_n \}\) and \(E ; H \vdash v_i : T_i\). Finally, by E-FIELD we have \(E ; H \vdash L' . l_i \downarrow v_i\).

\[\square\]

\textbf{Lemma 4} \textit{If} \(E ; H \vdash C\) \textit{and solve}(\(E, H, C, \text{required}\)) = \(E' ; H'\) \textit{and} \(E ; H \vdash L \downarrow v\), \textit{then there exists a value} \(v'\) \textit{such that} \(E' ; H' \vdash L \downarrow v'\). \textit{Furthermore, if} \(v\) \textit{is an object reference} \(r\), \textit{then also} \(v' = r\).

\textbf{Proof.} Since \(\text{solve}(E, H, C, \text{required}) = E' ; H'\) by \text{SOLVE} we have that \(\text{stay}(E, \text{required}) = C_{E_r}\) and \(\text{stay}(H, \text{required}) = C_{H_r}\) and \(E ; H \vdash C \leadsto C'\) and \(E' ; H' \vdash (C' \land C_{E_r} \land C_{H_r})\). We proceed by structural induction on \(L:\)

- Case \(L\) is a variable \(x\). Since \(E ; H \vdash L \downarrow v\), by E-VAR we have that \(E(x) = v\). Then since \(\text{stay}(E, \text{required}) = C_{E_r}\), by \text{STAYENV} we have that \(C_{E_r}\) includes a conjunct \(C_x\) such that \(\text{stay}(x = v, \text{required}) = C_x\).

  Case analysis of the rule used in the derivation of \(\text{stay}(x = v, \text{required}) = C_x:\)
  - \text{STAYCONST}: Then \(C_x\) has the form \text{weak} \(x = v\) and \(v\) is a constant \(c\). Therefore the variable \(x\) appears in the constraint \((C' \land C_{E_r} \land C_{H_r})\), so any solution to the constraint must include some value \(v'\) for \(x\) in \(E'\), and the result follows by E-VAR.
  - \text{STAYREF}: Then \(C_x\) has the form \text{required} \(x = v\) and \(v\) is an object reference \(r\). Therefore any solution to \((C' \land C_{E_r} \land C_{H_r})\) must map \(x\) to \(r\) in \(E'\), and the result follows by E-VAR.

- Case \(L\) has the form \(L' . l_i\). Then by E-FIELD we have \(E ; H \vdash L' \downarrow r\) and \(H(r) = \{ l_1 : v_1, \ldots, l_n : v_n \}\) and \(1 \leq i \leq n\) and \(v = v_i\). By induction we have \(E' ; H' \vdash L' \downarrow r\). Since \(\text{stay}(H, \text{required}) = C_{H_r}\), by \text{STAYHEAP} we have that \(C_{H_r}\) includes a conjunct \(C_r\) such that \(\text{stay}(r = \{ l_1 : v_1, \ldots, l_n : v_n \}, \text{required}) = C_r\). By \text{STAYOBJECT} \(C_r\) has the form \((\text{required} \ H(r) = \{ l_1 : x_1, \ldots, l_n : x_n \}) \land C_1 \land \cdots \land C_n\), where \(x_1 \ldots x_n\) are fresh variables and \(\text{stay}(x = v_i, \text{required}) = C_i\). Therefore any solution to \((C' \land C_{E_r} \land C_{H_r})\) must map \(r\) to an object value of the form \(\{ l_1 : v'_1, \ldots, l_n : v'_n \}\) in \(H'\), where \(v'_i\) is the value assigned to variable \(x_i\) in the solution. Therefore by E-FIELD we have \(E ; H \vdash L' . l_i \downarrow v'_i\). Finally suppose \(v_i\) is an object reference \(r_i\). Then by \text{STAYREF} \(C_i\) has the form \text{required} \(x_i = r_i\) so any solution to the constraints must map \(x_i\) to \(r_i\), so also \(v'_i\) is \(r_i\).

\[\square\]

\textbf{Lemma 5} \textit{If} \(E ; H \vdash L \downarrow v\) \textit{then there is some type} \(T\) \textit{such that} \(E ; H \vdash L : T\) \textit{and} \(E ; H \vdash v : T\).

\textbf{Proof.} By induction on the derivation of \(E ; H \vdash L \downarrow v:\)
• Case E-VARIABLE. Then L is a variable x and E(x) = v. Then by well formedness there is some type T such that E;H ⊢ x : T and by T-Var also E;H ⊢ v : T.

• Case E-FIELD. Then L has the form L'.1.1 and E;H ⊢ L' ⊥ r and H(r) = {l1:v1, ..., ln:vn} and 1 ≤ i ≤ n and v = vi. By induction there is some type T' such that E;H ⊢ L' : T' and E;H ⊢ r : T'. Then by T-Ref T' has the form {l1:T1, ..., ln:Tn} where E;H ⊢ vi : Ti. Then by T-FIELD also E;H ⊢ L : T'.

Lemma 6 If E;H ⊢ L ⊥ r then r is in the domain of H.

Proof. Case analysis on the last rule in the evaluation derivation.

• Case E-VARIABLE. Then L is a variable x and E(x) = r. Then the result follows from the assumption that E and H are well formed.

• Case E-FIELD. Then L has the form L'.1.1 and E;H ⊢ L' ⊥ r' and H(r) = {l1:v1, ..., ln:vn} and 1 ≤ i ≤ n and r = vi. Then the result follows from the assumption that E and H are well formed.

The second theorem formalizes the idea that all solutions to an assignment will produce structurally equivalent environments and heaps:

Theorem 2 (Structural Determinism) If <E|H|C|I|L := e> → <E1|H1|C1|I1> and <E|H|C|I|L := e> → <E2|H2|C2|I2> and E;H ⊢ x : T0, then there exists a type T such that E1;H1 ⊢ x : T and E2;H2 ⊢ x : T.

Proof. By S-Asgn we have E;H ⊢ e ⊥ v and stayPrefix(E, H, L) = CL and stayPrefix(E, H, I) = CI and solve(E, H, CL ∧ CI ∧ L=v ∧ I, weak) = E′1;H′1 and E′1;H′1 ⊢ C and solve(E′1, H′1, C ∧ L=v, required) = E1;H1. Also by S-Asgn and Lemma [3] we have solve(E, H, CL ∧ CI ∧ L=v ∧ I, weak) = E′2;H′2 and E′2;H′2 ⊢ C and solve(E′2, H′2, C ∧ L=v, required) = E2;H2. By Lemma [1] we have that E′2 = E′2 and H′1 = H′2. Since E;H ⊢ x : T0, by T-Var x is in the domain of E, so it is also in the domain of E′1 and by E-Var we have E′1;H′1 ⊢ x ⊥ v where E′1(x) = v. Then by Lemma [3] there is some type T' such that E′1;H′1 ⊢ v : T' and E′2;H′2 ⊢ x : T'. Finally by Lemma [2] we have that E1;H1 ⊢ x : T' and E2;H2 ⊢ x : T', so the result is shown with T = T'.

Definition 1 We say that L1 and L2 are aliases given E and H if either:

• L1 and L2 are the same variable x

• L1 has the form L1'.1 and L2 has the form L2'.1 and there is a reference r such that E;H ⊢ L1' ⊥ r and E;H ⊢ L2' ⊥ r

Definition 2 We say that L and L' are the operands of the constraint L == L'.
Definition 3 We define the induced graph of $I$ and $L$ given $E$ and $H$ as follows. Let $S$ be the set that includes $L$ as well as all operands of identity tests in $I$. Partition $S$ into equivalence classes defined by the alias relation: $L_1$ and $L_2$ are in the same equivalence class if and only if they are aliases given $E$ and $H$. Then the induced graph has one node per equivalence class and an undirected edge between nodes $N_1$ and $N_2$ if there is a conjunct $L_1 = L_2$ in $I$ such that $L_1$ belongs to node $N_1$ and $L_2$ belongs to node $N_2$.

Definition 4 We say that a node in the induced graph of $I$ and $L$ given $E$ and $H$ is relevant if it is reachable from $L$’s node in the graph; an l-value is relevant if its node in the graph is relevant.

Definition 5 The relevant update of $E$ and $H$ for $I$ and $L=v$ is the environment $E'$ and heap $H'$ that are identical to $E$ and $H$ except that for each relevant l-value $L_0$ in the induced graph of $I$ and $L$ given $E$ and $H$:

- If $L_0$ is a variable $x$, then $E'(x) = v$.
- If $L_0$ has the form $L_1.L$ and $E;H \vdash L_1 \Downarrow r$, then $E';H' \vdash r.L \Downarrow v$.

Lemma 7 If stayPrefix($E$, $H$, $L$) = $C_L$ and stayPrefix($E$, $H$, $I$) = $C_I$ and solve($E$, $H$, $C_L \land C_I \land L=v \land I$, weak) = $E_1;H_1$ and solve($E$, $H$, $C_L \land C_I \land L=v \land I$, weak) = $E_2;H_2$, then $E_1 = E_2$ and $H_1 = H_2$.

Proof. By Solve we have stay($E$, weak) = $C_E$ and stay($H$, weak) = $C_H$ and $E;H \vdash (C_L \land C_I \land L=v \land I) \rightarrow C'$ and $E';H' \models (C' \land C_E \land C_H)$ and $E'';H'' \models (C' \land C_E \land C_H)$. By Lemma 6, the relevant update of $E$ and $H$ for $I$ and $L=v$ is a solution to the constraint $C' \land C_E \land C_H$. Therefore if $E'$ and $H'$ is not the relevant update, then by Definition 5, either:

- there is a variable $x$ such that $E_0(x) = E(x)$ but $E'(x)$ has a different value
- there is a reference $r$ and field label $l$ such that $H_0(r).l = H(r).l$ but $H'(r).l$ has a different value.

Consider the former. Since stay($E$, weak) = $C_E$, by StayEnv, StayConst, and StayRef we have that $C_E$ includes a weak constraint $x=v_x$, where $E(x) = v_x$. Since $E'$ and $H'$ include all the updates of the relevant update, $E'$ and $H'$ satisfy strictly fewer weak constraints than the relevant update, contradicting the optimality of $E'$ and $H'$.

Similarly, consider the latter. Since stay($H$, weak) = $C_H$, by StayHeap and StayObject $C_H$ includes a required constraint $H(r) \{l_1:x_{i_1},\ldots,l_n:x_{i_n}\}$ where the $x_i$ variables are fresh and $l$ is some $l_i$. Then by StayConst and StayRef there is a weak constraint of the form $x_i = v_i$ where $H(r).l_i = v_i$. Since $E'$ and $H'$ include all the updates of the relevant update, $E'$ and $H'$ satisfy strictly fewer weak constraints than the relevant update, contradicting the optimality of $E'$ and $H'$.

Lemma 8 If stay($E$, weak) = $C_E$ and stay($H$, weak) = $C_H$ and stayPrefix($E$, $H$, $L$) = $C_L$ and stayPrefix($E$, $H$, $I$) = $C_I$ and $E;H \vdash (C_L \land C_I \land L=v \land I) \rightarrow C'$ and the constraint $C' \land C_E \land C_H$ is satisfiable, then the relevant update $E'$ and $H'$ of $E$ and $H$ for $I$ and $L=v$ is a solution to the constraint $C' \land C_E \land C_H$. 

VPRI Technical Report TR-2015-001
Proof. It suffices to show that all required constraints in \( C' \land C_E \land C_H \) are satisfied in \( E' \) and \( H' \). We consider the various constraints in turn:

- \( C_E \): Since \( \text{stay}(E, \text{weak}) = C_E \), by \text{STAYENV}, \text{STAYCONST}, and \text{STAYREF} there are no required constraints in \( C_E \), so all required constraints are satisfied vacuously.

- \( C_H \): Since \( \text{stay}(H, \text{weak}) = C_H \), by \text{STAYHEAP}, \text{STAYOBJECT}, \text{STAYCONST}, and \text{STAYREF} the only required constraints in \( C_H \) have the form \( H(r) = \{l_1:x_1, \ldots, l_n:x_n\} \) where the \( x_i \) variables are fresh and \( H \) maps \( r \) to some value of the form \( \{l_1:v_1, \ldots, l_n:v_n\} \). By Definition [5] also \( H' \) maps \( r \) to a value of the form \( \{l_1:v'_1, \ldots, l_n:v'_n\} \) so the constraint is satisfied.

- \( C_I \): By \text{STAYPREFIXFIELD} the conjuncts in \( C_I \) have the form \( x=v \) or \( r.l=v \). Suppose the relevant update fails to satisfy one of these conjuncts. We consider each form in turn:
  - \( x=v \): Then the relevant update maps \( x \) to some \( v' \neq v \) in \( E' \). But by \text{SOLVE} and Lemma [9] any solution to the constraint \( C' \land C_E \land C_H \) must map \( x \) to \( v' \) in the environment, which violates the constraint \( x=v \). So there must be no solution to the constraints, contradicting our assumption of satisfiability.
  - \( r.l=v \): Then the relevant update maps \( r.l \) to some \( v' \neq v \) in \( H' \). But by \text{SOLVE} and Lemma [9] any solution to the constraint \( C' \land C_E \land C_H \) must map \( r.l \) to \( v' \) in the environment, which violates the constraint \( r.l=v \). So there must be no solution to the constraints, contradicting our assumption of satisfiability.

- \( C_I \): By \text{STAYPREFIXIDENT} the conjuncts in \( C_I \) have the form \( x=v \) or \( r.l=v \). So the argument is identical to that above for the \( C_L \) constraint.

- \( L=v \): By Definitions [3] and [5] we have that \( L \) is relevant for \( I \) and \( L \) given \( E \) and \( H \). Then by Lemma [10] we have \( E';H';L \downarrow v \), so by \text{E-OP} and the semantics of equality we have \( E';H';L=v \downarrow \text{true} \).

- \( I \): Consider a conjunct \( L_0 \equiv L_1 \) in \( I \). We have two cases:
  - \( L_0 \) is relevant for \( I \) and \( L \) given \( E \) and \( H \). By Definitions [4] and [3] \( L_1 \) is also relevant. Then by Lemma [10] we have \( E';H';L_0 \downarrow v \) and \( E';H';L_1 \downarrow v \), so by \text{E-IDENTITYTRUE} we have \( E';H';L_0=L_1 \downarrow \text{true} \).
  - \( L_0 \) is not relevant for \( I \) and \( L \) given \( E \) and \( H \). By Definitions [4] and [3] \( L_1 \) is also not relevant. By well formedness we have \( E;H;L_0=L_1 \downarrow \text{true} \), so by \text{E-IDENTITYTRUE} there is some \( v_0 \) such that \( E;H;L_0 \downarrow v_0 \) and \( E;H;L_1 \downarrow v_0 \). Then by Lemma [11] we have \( E';H';L_0 \downarrow v_0 \) and \( E';H';L_1 \downarrow v_0 \), so by \text{E-IDENTITYTRUE} we have \( E';H';L_0=L_1 \downarrow \text{true} \).

\[ \square \]

Lemma 9 If \( \text{stayPrefix}(E, H, L) = C_L \) and \( \text{stayPrefix}(E, H, I) = C_I \) and \( \text{solve}(E, H, C_L \land C_I \land L=v \land I, \text{weak}) = E';H' \) and \( L_0 \) is a relevant \( l \)-value of \( I \) and \( L \) given \( E \) and \( H \), then \( E';H';L_0 \downarrow v \) and:

- If \( L_0 \) is a variable \( x \), then \( E'(x) = v \).
- If \( L_0 \) has the form \( L_0', l \) and \( E;H \downarrow L_0' \downarrow r \), then \( E';H';r.l \downarrow v \).

Proof.

By Definition [4] we know that \( L_0 \)'s node in the induced graph for \( I \) and \( L \) given \( E \) and \( H \) is reachable from \( L \)'s node. The proof proceeds by induction on the length \( k \) of the path between these nodes.
Lemma 10 If \( \text{stayPrefix}(E, H, L) = C_L \) and \( \text{stayPrefix}(E, H, I) = C_I \) and \( E \) and \( H' \) is the relevant update of \( E \) and \( H \) for \( I \) and \( L = v \) and \( E'; H' \vdash C_L \downarrow \text{true} \) and and \( E'; H' \vdash C_I \downarrow \text{true} \) and \( L_0 \) is a relevant l-value for \( I \) and \( L \) given \( E \) and \( H \), then \( E'; H' \vdash L_0 \downarrow v \).

Proof. Case analysis of the structure of \( L_0 \):

- Case \( L_0 \) is a variable \( x \): By Definition 10 \( E'(x) = v \), so the result follows by \( E\text{-VAR} \).

- Case \( L_0 \) has the form \( L'.1 \): First we argue that \( \text{stayPrefix}(E, H, L_0) = C_{L_0} \) and \( E'; H' \vdash C_{L_0} \downarrow \text{true} \). If \( L_0 \) is \( L \), then these follow from the assumptions of the lemma. Otherwise, by Definition 10 \( L_0 \) is an operand in an identity test in \( I \). Then since \( \text{stayPrefix}(E, H, I) = C_I \) by \( \text{STAYPREFIXIDENT} \) we have \( \text{stayPrefix}(E, H, L_0) = C_{L_0} \), where \( C_{L_0} \) is a conjunct in \( C_I \). Then since \( E'; H' \vdash C_I \downarrow \text{true} \) also \( E'; H' \vdash C_{L_0} \downarrow \text{true} \). So by Lemma 10 and Lemma 12 we have \( E'; H' \vdash L' \downarrow r \) and \( E'; H' \vdash L_0 \downarrow r \). Then since \( E'; H' \vdash L_0 \downarrow v \) by \( E\text{-FIELD} \) we have \( H'(r) \) is a record whose \( 1 \) field has value \( v \). Then by \( E\text{-FIELD} \) also \( E'; H' \vdash r.1 \downarrow v \) and \( E'; H' \vdash L_0 \downarrow v \).

\[\square\]
Lemma 11  If stayPrefix(E, H, L) = $C_L$ and stayPrefix(E, H, I) = $C_I$ and $E'$ and $H'$ is the relevant update of $E$ and $H$ for I and $L = \nu$ and $E'; H' \Downarrow C_I \Downarrow$ true and and $E'; H' \Downarrow C_I \Downarrow$ true and $L_0$ is an operand of an identity test in I and $L_0$ is not relevant for I and $L$ given $E$ and $H$ and $E; H \Downarrow L_0 \Downarrow v_0$, $E'; H' \Downarrow L_0 \Downarrow v_0$.

Proof.  Case analysis on the structure of $L_0$:

- Case $L_0$ is a variable $x$. Since $x$ is not relevant, by Definition 5 the value of $x$ in $E'$ is the same as that in Euid. Since $E; H \Downarrow L_0 \Downarrow v_0$, by E-Var we have $E(x) = v_0$, so also $E'(x) = v_0$ and the result follows by E-Var.

- Case $L_0$ has the form $L'.1$. Since $L_0$ is an operand in an identity test in I and stayPrefix(E, H, I) = $C_I$, by STAYPREFIXIDENT we have stayPrefix(E, H, $L_0$) = $C_{L_0}$, where $C_{L_0}$ is a conjunct in $C_I$. Then since $E'; H' \Downarrow C_I \Downarrow$ true also $E'; H' \Downarrow C_{L_0} \Downarrow$ true.

Therefore by Lemma 12 there is some $x'$ such that $E; H \Downarrow L' \Downarrow x'$ and $E'; H' \Downarrow L' \Downarrow x'$. Since $E; H \Downarrow L_0 \Downarrow v_0$ by E-FIELD and Lemma 13 we have that $H(x').1$ is $v_0$. By Definition 5 $H(x')$ also has a field with label 1, so by E-FIELD we are done if that field’s value is $v_0$. Suppose not. Then by Definition 5 there is some relevant l-value $L_1$ of the form $L'1.1$ such that $E; H \Downarrow L'1 \Downarrow x'$. But then by Definition 1 we have that $L_0$ and $L_1$ are aliases so they correspond to the same node in the induced graph of I and L given E and H by Definition 3. But then since $L_1$ is relevant, by Definition 4 so is $L_0$ and we have a contradiction.

Lemma 12  If stayPrefix(E, H, L.f) = $C$ and $E'; H' \Downarrow C \Downarrow$ true, then there is some reference $r$ such that $E; H \Downarrow L \Downarrow r$ and $E'; H' \Downarrow L \Downarrow r$.

Proof.  By STAYPREFIXFIELD we have $L.f = x.1_1 \ldots 1_n$ and $n > 0$ and $E; H \Downarrow x \Downarrow r$ and $E; H \Downarrow x.1_1 \Downarrow r_1 \cdots E; H \Downarrow x.1_1 \ldots 1_{n-1} \Downarrow r_{n-1}$ and $C$ is $x=r \land x.1_1=r_1 \land \cdots \land x.1_1 \ldots 1_{n-1}=r_{n-1}$.

- Case $n = 1$. Then $L$ is $x$ and $C$ is $x=r$. Since $E'; H' \Downarrow C \Downarrow$ true, by E-OP and the semantics of equality we have $E'; H' \Downarrow x \Downarrow v_1$ and $E'; H' \Downarrow r \Downarrow v_2$ and $v_1 = v_2$. By E-Ref we have that $v_2 = r$, so the result follows.

- Case $n > 1$. Then $L$ is $x.1_1 \ldots 1_{n-1}$. Since $E'; H' \Downarrow C \Downarrow$ true, by E-OP and the semantics of equality we have $E'; H' \Downarrow L \Downarrow v_1$ and $E'; H' \Downarrow r_{n-1} \Downarrow v_2$ and $v_1 = v_2$. By E-Ref we have that $v_2 = r_{n-1}$, so the result follows.

Lemma 13 (Determinism)

- If $E; H \Downarrow e \Downarrow v_1$ and $E; H \Downarrow e \Downarrow v_2$ then $v_1 = v_2$.

- If stay(E, $\rho$) = $C_1$ and stay(E, $\rho$) = $C_2$ then $C_1 = C_2$.

- If stay(H, $\rho$) = $C_1$ and stay(H, $\rho$) = $C_2$ then $C_1 = C_2$.

- If stayPrefix(E, H, L) = $C_1$ and stayPrefix(E, H, L) = $C_2$ then $C_1 = C_2$.  

14
\* If \( \text{stayPrefix}(E, H, I) = C_1 \) and \( \text{stayPrefix}(E, H, I) = C_2 \) then \( C_1 = C_2 \).

**Proof.** Straightforward. \( \square \)