Technical Report: A Type-and-Effect System for Object Initialization

FENGYUN LIU, EPFL
ONDŘEJ LHOTÁK, University of Waterloo
AGGELOS BIBOUDIS, EPFL
MARTIN ODERSKY, EPFL

1 A CLASS-BASED LANGUAGE

In this section, we introduce a class-based language which adopts the design of class parameters. We introduce a semantic property called scoped reachability, which we take advantage of in the design of our type-and-effect system.

1.1 Syntax

Our language resembles a subset of Scala having only top-level classes, mutable fields and methods.

\[
\begin{align*}
\mathcal{P} & \in \text{Program} \quad ::= \quad (\bar{C}, e) \\
C & \in \text{Class} \quad ::= \quad \text{class } C(\hat{f}:D) \{ \bar{F} \bar{M} \} \\
\bar{F} & \in \text{Field} \quad ::= \quad \text{var } f:C = e \\
e & \in \text{Exp} \quad ::= \quad x \mid \text{this} \mid e.f \mid e.m(\bar{e}) \mid \\
& \quad \text{new } C(\bar{e}) \mid e.f = e; e \\
\bar{M} & \in \text{Method} \quad ::= \quad \text{def } m(x:C) : D = e \\
x, y, z & \in \text{Variable} \quad \text{variable names} \\
f, \hat{f}, \tilde{f} & \in \text{FieldName} \quad \text{field names} \\
m & \in \text{MethodName} \quad \text{method names} \\
C, D, E & \in \text{ClassName} \quad \text{class names}
\end{align*}
\]

The language has two salient features compared to conventional class-based programming languages:

- there are no constructors, which are replaced by class parameters,
- in field declaration ($\bar{F}$), field initializer is mandatory.

With this syntax, we get stackability for free, as all fields must be assigned at the end of the class body. Meanwhile, it enables us to give a different typing rule to field reassignment from field initialization to enforce monotonicity of initialization.

A program $\mathcal{P}$ is a pair of a list of class definitions and a term representing the execution entry point. A class is composed of class parameters ($\hat{f}:D$), body fields (\texttt{var } $f:C = e$) and methods ($\texttt{def } m(x:C) : D = e$). A class parameter $\hat{f}$ is also a field of its defining class. By default, we use...
to range over all fields, and \( \hat{f} \) only range over class parameters. The symbol \( \hat{f} \) refers to class parameters that may accept a value that is not transitively initialized. These class parameters are called cold class parameters. The tilde annotation on \( \hat{f} \) is only used in the type-and-effect system, it does not have any special meaning at runtime. That is the only annotation that is required in source code in our system.

### 1.2 Semantics

The following constructs are used in defining the semantics:

\[
\begin{align*}
ct & \in \text{ClassTable} = \text{ClassName} \rightarrow \text{Class} \\
\sigma & \in \text{Store} = \text{Loc} \rightarrow \text{Obj} \\
\rho & \in \text{Env} = \text{Name} \rightarrow \text{Value} \\
o & \in \text{Obj} = \text{ClassName} \times (\text{Name} \rightarrow \text{Value}) \\
l, \psi & \in \text{Value} = \text{Loc}
\end{align*}
\]

A class table \( ct \) is a partial function mapping class names to class definitions. A store \( \sigma \) is a partial function from locations to objects. An environment \( \rho \) is a partial function mapping names to values, representing values of method parameters. The only values are references to objects \( \text{Loc} \) and each object is a pair of a class name and a mapping from field names to values.

Our definitional interpreter appears in Figure 1, which defines the big-step operational semantics for the experimental language. When evaluating a program \((\mathcal{C}, e)\), the interpreter first turns the lists of class definitions into a class table, and then evaluates the top-level expression \( e \) in an empty environment. Additionally, the interpreter passes a dummy instance of class \text{Null} for this. The type system ensures that the class \text{Null} is defined and has no fields or methods.

Evaluating an expression requires the quadruple \((ct, \sigma, \rho, \psi)\), consisting of a class table \(ct\), a store \(\sigma\), an environment \(\rho\) and a value \(\psi\) for this. The evaluation returns the result value and the updated store.

Note that non-initialized fields are represented by missing keys in the object, instead of a null value. Newly initialized objects have no fields, and new fields are gradually inserted during initialization until all fields defined by the class have been assigned.

### 1.3 Scoped Reachability

Lexical scoping is adopted by almost all modern programming languages, which is assumed to be more friendly for programming compared to dynamic scoping. Syntactically, we know that lexical scoping means that only variables in scope may be used, which is usually enforced in a typing judgment like \( \Gamma \vdash t : T \), where \( \Gamma \) holds all lexically available variables.

An interesting question is, what does lexical scoping guarantee in an evaluation \( [e] (ct, \sigma, \rho, \psi) = (l, \sigma') \), where \( \sigma \) is the heap, \( \rho \) is the environment and \( \psi \) is the value for this? Intuitively, as the expression \( e \) may only reach values that are in \( \sigma \) via variables in \( \text{dom}(\rho) \) or this, if the result value \( l \) reaches any values which pre-exist in \( \sigma \), then the values must be reachable from this and \( \text{dom}(\rho) \) in \( \sigma \). To some extent, the environment is like a gate to the heap. Lexical scoping ensures that if the resulting value of an expression reaches any values that pre-exist in the heap, then the expression must have made use of variables from that environment. More formally, the following property is a direct result of lexical scoping:

\[
[e] (ct, \sigma, \rho, \psi) = (l, \sigma') \implies \text{scoped}(\sigma', \sigma, \{ l \}, \text{codom}(\rho) \cup \{ \psi \})
\]

With the scoped relation we define that a set of addresses \( L_2 \subseteq \text{dom}(\sigma_2) \) is scoped by a set of addresses \( L_1 \subseteq \text{dom}(\sigma_1) \):
Program evaluation

\[ [(\overline{C}, e)] = [e] (ct, l \mapsto (\text{Null}, \emptyset), \emptyset, l) \]

where \( ct = \overline{C} \rightarrow \overline{C} \) and \( l \) is a fresh location

Expression evaluation

\[ [x] (ct, \sigma, \rho, \psi) = (\rho(x), \sigma) \]
\[ [\text{this}] (ct, \sigma, \rho, \psi) = (\psi, \sigma) \]
\[ [e.f] (ct, \sigma, \rho, \psi) = (\rho(f), \sigma_1) \text{ where } (l_0, \sigma_1) = [e] (ct, \sigma, \rho, \psi) \]
and \( (_-, o) = \sigma_1(l_0) \)
\[ [e_0.m(\overline{e})] (ct, \sigma, \rho, \psi) = [e_1] (ct, \sigma_2, \rho_1, l_0) \]
where \((l_0, \sigma_1) = [e_0] (ct, \sigma, \rho, \psi) \)
and \((C, _, \psi) = \sigma_1(l_0) \)
and \( \text{lookup}(ct, C, m) = \text{def } m(x:D) : E = e_1 \)
and \((\overline{l}, \sigma_2) = [\overline{e}] (ct, \sigma_1, \rho, \psi) \)
and \( \rho_1 = x \mapsto \overline{l} \)
\[ [\text{new } C(\overline{e})] (ct, \sigma, \rho, \psi) = (l, \sigma_3) \]
where \((\overline{l}, \sigma_1) = [\overline{e}] (ct, \sigma, \rho, \psi) \)
and \( \sigma_2 = l \mapsto (C, \emptyset); \sigma_1 \) where \( l \) is fresh
and \( \sigma_3 = \text{init}(l, \overline{l}, C, ct, \sigma_2) \)
\[ [e_1.f = e_2; e] (ct, \sigma, \rho, \psi) = [e] (ct, \sigma_3, \rho, \psi) \]
where \((l_1, \sigma_1) = [e_1] (ct, \sigma, \rho, \psi) \)
and \((l_2, \sigma_2) = [e_2] (ct, \sigma_1, \rho, \psi) \)
and \( \sigma_3 = \text{assign}(l_1, f, l_2, \sigma_2) \)

Initialization

\[ \text{init}(\psi, \overline{l}, C, ct, \sigma) = \overline{F} (ct, \sigma_1, \psi) \]
where \( \text{lookup}(ct, C) = \text{class } C(\overline{f}; D) \{ \overline{F} \overline{M} \} \)
and \( \sigma_1 = \text{assign}(\psi, \overline{f}, \overline{l}, \sigma) \)
\[ [\text{var } f : D = e] (ct, \sigma, \psi) = \text{assign}(\psi, f, l_1, \sigma_1) \text{ where } (l_1, \sigma_1) = [e] (ct, \sigma, \emptyset, \psi) \]

Helpers

\[ [\overline{e}] (ct, \sigma, \rho, \psi) = \text{fold } \overline{e} (\text{Nil}, \sigma) f \text{ where } f (l_0, \sigma_0) e \mapsto \text{let } (l_0, \sigma_2) = [e] (ct, \sigma_1, \rho, \psi) \text{ in } (l :: l_0, \sigma_2) \]
\[ [\overline{F}] (ct, \sigma, \psi) = \text{fold } \overline{F} \sigma \text{ where } f \sigma_1 \overline{F} = [\overline{F}] (ct, \sigma_1, \psi) \]
\[ \text{assign}(\psi, f, l, \sigma) = \{ \psi \mapsto (C, [f \mapsto l] \text{fields})\sigma \text{ where } (C, \text{fields}) = \sigma(\psi) \}
\[ \text{assign}(\psi, \overline{f}, \overline{l}, \sigma) = \{ \psi \mapsto (C, \overline{f} \mapsto \overline{l} \text{fields})\sigma \text{ where } (C, \text{fields}) = \sigma(\psi) \} \]

Fig. 1. Semantics, defined as a definitional interpreter.

\[
\text{scoped}(\sigma_2, \sigma_1, L_2, L_1) \triangleq \\
\forall l_2 \in L_2. \forall l \in \text{dom}(\sigma_1). \text{reachable}(\sigma_2, l_2, l) \implies \\
\exists l_1 \in L_1. \text{reachable}(\sigma_1, l_1, l)
\]

, Vol. 1, No. 1, Article . Publication date: January 2020.
This is because in a language with mutation, Fengyun Liu, Ondřej Lhoták, Aggelos Biboudis, and Martin Odersky, Vol. 1, No. 1, Article. Publication date: January 2020.

It means if the result object 8 reaches any object which pre-exists in the heap $\sigma$, then it must be reachable from object 2 in the heap $\sigma$. The object 7 which is reachable from this in the heap $\sigma$ is no longer reachable from object 2 in the heap $\sigma'$ after the execution due to the removal of the link from object 1 to object 7 caused by reassignment.

The $\text{reachable}(\sigma, l_1, l_2)$ relation denotes that there is a path from one object $l_1$ to another, $l_2$, in the memory graph $\sigma$. Note that in the above, we use $\text{reachable}(\sigma_1, l_1, l)$ instead of $\text{reachable}(\sigma_2, l_1, l)$. This is because in a language with mutation, $l$ may no longer be reachable from $l_1$ in $\sigma_2$ due to reassignment. This can be seen in Figure 2.

However, the definition of $\text{scoped}$ is only part of the story. In an evaluation $[e](ct, \sigma, \rho, \psi) = (l, \sigma')$, not only the result $l$ is scoped by $\text{codom}(\rho) \cup \psi$, but also existing scoped relations should continue to hold. In other words, existing scoped relations are $\text{preserving}$. More formally, the following property is also a result of lexical scoping:

$$[e](ct, \sigma, \rho, \psi) = (l, \sigma') \implies \text{preserving}(\sigma', \sigma, \text{codom}(\rho) \cup \{ \psi \})$$ (2)

The predicate $\text{preserving}$ is defined as follows:

$$\text{preserving}(\sigma_2, \sigma_1, L_1) \triangleq \forall \sigma_0, \forall L \subseteq \text{dom}(\sigma_1), \forall L_0 \subseteq \text{dom}(\sigma_0).
\text{scoped}(\sigma_1, \sigma_0, L_1, L_0) \land \text{scoped}(\sigma_1, \sigma_0, L, L_0) \implies \text{scoped}(\sigma_2, \sigma_0, L, L_0)$$

Note that in the definition above, $\text{preserving}$ is a triple relation among $\sigma_2$, $\sigma_1$, and $L_1$. Intuitively, the set $L_1$ can be thought as the current stack frame: it is needed because not all lexical scoping are preserving due to reassignment; only those scoping by a stack frame that is still on the stack. In the definition, we may think $L_0$ as the previous stack frame, $\text{scoped}(\sigma_1, \sigma_0, L_1, L_0)$ ensures $L_0$ is still on the stack. The definition says that any set $L$ which is scoped by $L_0$ continues to be scoped by $L_0$ in the heap migration from $\sigma_1$ to $\sigma_2$. In particular, $L$ can be $L_1$.

Without the second property $\text{preserving}$, in an evaluation $[e](ct, \sigma, \rho, \psi) = (l, \sigma')$, we cannot even easily conclude that $\text{scoped}(\sigma', \sigma, L_1, L_1)$ where $L_1 = \text{codom}(\rho) \cup \{ \psi \}$. With property (2), we may prove $\text{scoped}(\sigma', \sigma, L_1, L_1)$ by choosing $\sigma_0 = \sigma$ and $L_0 = L = L_1$, as $\text{scoped}(\sigma, \sigma, L_1, L_1)$ holds trivially. In the meta-theory (Section 3.3), the properties (1) and (2) are interdependent, thus they are proved together.

We call the properties (1) and (2) $\text{scoped reachability}$, which is a semantic consequence of lexical scoping. This property is related to separate logic [7], where the part of the heap that a command
actually uses is called its **footprint**. Here, we over-approximate the footprint of an expression by the set of objects reachable from \(\text{codom}(\rho)\) and \(\psi\), and we may think unreachable heap regions from \(\text{codom}(\rho)\) and \(\psi\) are valid frames for an expression.

### 1.4 Scoped Initialization

Scoped reachability, when combined with monotonicity and stacked initialization, gives the following nice property:

\[
\text{In an evaluation } [e] (ct, \sigma, \rho, \psi) = (l, \sigma'), \text{ if the values } \text{codom}(\rho) \text{ and the value } \psi \text{ for this are transitively initialized, then the result } l \text{ must be transitively initialized.}
\]

By **transitively initialized**, we mean all values reachable from the current object are initialized. To give an intuition of the statement, let us enumerate the transitively reachable values from the resulting value \(l\):

- if \(l\) reaches a value \(l'\) which exists on the heap before the evaluation, by lexical scoping it must be reachable from \(\text{codom}(\rho)\) or \(\psi\). Since \(\text{codom}(\rho)\) and \(\psi\) are transitively initialized, \(l'\) must be initialized too. And by monotonicity it continues to be initialized after the evaluation.
- if \(l\) reaches a fresh value on the resulting heap (including itself), all fields must be assigned thanks to stackability.

As a conclusion, the resulting value \(l\) does not reach any value whose fields are not fully assigned, thus semantically we may take the value as transitively initialized. Similarly, the following is true as well:

\[
\text{In a new-expression, if the arguments are transitively initialized, then the result must be transitively initialized.}
\]

The two insights are exploited in the type-and-effect system. The claims informally stated above are made precise and proved as part of the meta-theory (see the technical report for more details).

### 2 FORMALIZATION

The type-and-effect system is composed of the following components:

- definition of potentials and effects (Figure 3)
- expression typing (Figure 4)
- definition typing (Figure 5)
- effect checking (Figure 6)

#### 2.1 Effects and Potentials

As seen from Figure 3, the definition of potentials \((\pi)\) and effects \((\phi)\) depends on roots \((\beta)\). Roots are the shortest extendable path that represents an alias of a value that may not be transitively initialized. There are two roots in the system:

- \(C\text{.this}\) represents aliasing of \texttt{this} inside class \(C\).
- \(\text{warm}[C]\) represents aliasing of a value of class \(C\), all fields of which are assigned, but it may not be transitively initialized.

Potentials \((\pi)\) represent aliasing information. They extend roots with field aliasing \(\beta.f\) and method aliasing \(\beta.m\). Field aliasing \(\beta.f\) represents aliasing of the field \(f\) of \(\beta\), while method aliasing \(\beta.m\) represents aliasing of the return value of method \(m\) with the receiver \(\beta\). The potential cold refers to a value that may not be initialized, which is used to represent the potential of cold class parameters.

Effects \((\phi)\) include field accesses, method calls and leakings of possibly uninitialized values. A leaking effect is represented with \(\pi\uparrow\), which means the leaking of the potential \(\pi\). The field access
effect $\beta.f!$ means that the field $f$ is accessed on $\beta$. The method call effect $\beta.m!$ means the method $m$ is called on $\beta$.

To simplify our presentation, we use the syntax $\Pi\uparrow$ to denote the set $\{\pi\uparrow | \pi \in \Pi\}$.

<table>
<thead>
<tr>
<th>Potentials and Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T ::= C \mid D \mid E \mid \cdots$</td>
</tr>
<tr>
<td>$\beta ::= C.this \mid warm[C]$</td>
</tr>
<tr>
<td>$\pi ::= cold \mid \beta \mid \beta.f \mid \beta.m$</td>
</tr>
<tr>
<td>$\Pi ::= {\pi_1, \pi_2, \cdots}$</td>
</tr>
<tr>
<td>$\phi ::= \pi\uparrow</td>
</tr>
<tr>
<td>$\Phi ::= {\phi_1, \phi_2, \cdots}$</td>
</tr>
<tr>
<td>$\Omega ::= {f_1, f_2, \cdots}$</td>
</tr>
<tr>
<td>$\Delta ::= f_i \mapsto (\Phi_i, \Pi_i)$</td>
</tr>
<tr>
<td>$S ::= m_i \mapsto (\Phi_i, \Pi_i)$</td>
</tr>
<tr>
<td>$E ::= C \mapsto (\Delta, S)$</td>
</tr>
</tbody>
</table>

Select

| $select(\Pi, f) = \Pi.map(\pi \Rightarrow select(\pi, f)).reduce(\oplus)$ |
| $select(\beta, f) = (\emptyset, \{cold\})$                         |
| $select(\beta, f) = (\emptyset, \emptyset)$                        |
| $select(\beta, f) = (\{\beta.f!\}, \{\beta.f\})$                  |
| $select(\pi, f) = (\{\pi\uparrow\}, \emptyset)$                   |

Call

| $call(C.m, \Pi) = \Pi.map(\pi \Rightarrow call(m, \pi)).reduce(\oplus)$ |
| $call(m, \beta) = (\{\beta.m!\}, \{\beta.m\})$                      |
| $call(m, \pi) = (\{\pi\uparrow\}, \emptyset)$                        |

Init

| $init(C, \beta_i = \Pi_i) = (\cup\Pi_{k \neq j} \uparrow, \{warm[C]\}) \exists j, \Pi_j \neq \emptyset$ |
| $init(C, x_i = \Pi_i) = (\cup \Pi_{j \uparrow}, \emptyset)$            |

Union

| $(A_1, A_2) \oplus (B_1, B_2) = (A_1 \cup B_1, A_2 \cup B_2)$ |

Fig. 3. Type and Effect Definition

### 2.2 Expression Typing

Expression typing (Figure 4) has the form $\Xi; \Gamma; C \vdash e : T! (\Phi, \Pi)$, it means that the expression $e$ in class $C$ under the environment $\Gamma$, can be typed as $T$, it produces effects $\Phi$ and has the potential $\Pi$. Generally, when typing an expression, the effects of sub-expressions will accumulate, while potentials may be refined (via selection), leaked (used as arguments to methods) or absorbed (when used as arguments to cold class parameters).

, Vol. 1, No. 1, Article . Publication date: January 2020.
The definitions assume helper methods $\text{fieldType}(\Xi, C, f)$, $\text{methodType}(\Xi, C, m)$ and $\text{constrType}(\Xi, C)$ to look up in class table $\Xi$ the type, respectively, of field $C.f$, of method $C.m$ and of the constructor of $C$.

In the typing rule T-Var, the effects are empty as accessing a variable cannot cause any runtime error. The potential is empty because the design of the system ensures that variables are transitively initialized, thus they do not need to be tracked in the system.

In the typing rule T-This, the effect is empty as expected, and the potential is $C.this$, as it aliases this in class $C$.

In the typing rule T-Sel, it first computes the effects $\Phi$ and potentials $\Pi$ of the expression $e$. Then it calls $\text{select}(\Pi, f)$ to produce the final potentials $\Pi'$ and accompanied effects $\Phi'$. The helper method select is defined in Figure 3. There are several cases:

- selection of cold class parameter $\hat{f}$ on $\beta$
- selection of non-cold class parameter $\hat{f}$ on $\beta$
- selection of body field $f$ on $\beta$
- selection of a field on $\pi$ where $\pi.f$ is too long

In the first case, the field $\hat{f}$ may hold a value that is not transitively initialized, thus the potential is represented as cold. The effect is empty, as class parameters are always initialized before the class body is executed.
For the same reason, in the second case, the effects are empty. The potentials are empty because a non-cold class parameter may only hold a value that is transitively initialized, thus we do not need to track it in the system.

In the third case, selecting a body field on $\beta$ produces the effect $\beta.f!$ due to field access, and the potential $\beta.f$ due to the fact that the field $f$ may hold a value which is not transitively initialized.

In the last case, if the length of $\pi.f$ exceeds the maximum length of potentials, the system just leaks the potential $\pi$, which is equivalent to say that $\pi$ is a transitively initialized value, thus the potential of the selection is empty. The system restricts the length of potentials to make the domain finite. In our formalization, we set the length to 2. In implementation, the maximum length of potentials may be parameterized.

The typing rule $T\text{-}Call$, first checks the receiver $e_0$ and the arguments $e_i$. Then it calls the helper function $call(T_0.m, \Pi)$. The definition of $call$ (defined in Figure 3) distinguishes two cases:

- the receiver is $\beta$
- the receiver is $\pi$ where $\pi.m$ is too long

In the first case, it produces the effect $\beta.m!$ and potential $\beta.m$ — remember $\beta.m!$ is a placeholder to say all effects associated with the method $m$, and $\beta.m$ to all potentials associated with the return value of the method $m$.

In the second case, the system just leaks the potential $\pi$, just as the case of selection. The resulting potential is empty, because both the receiver and arguments are fully initialized. The semantic justification for this rule is based on a property called scoped reachability, which we explain and formalize in the technical report.

Note that in the current system, method arguments must be transitively initialized, this fact is expressed in the method $call$ by leaking all potentials of the arguments as effects.

To type check $new$–expressions, the typing rule $T\text{-}New$ first type checks all arguments, then it calls the helper method $init(C, \overset{\not\exists}{f}_i = \Pi_i)$. The helper method (defined in Figure 3) distinguishes two cases:

- the class parameters of $C$ accept values under initialization and there exists at least one corresponding argument whose potential is non-empty.
- either class $C$ does not accept values under initialization or all potentials for cold class parameters $\overset{\not\exists}{f}$ are empty.

In the first case, it leaks all potentials that do not correspond to cold class parameters, which is equivalent to say that these arguments are transitively initialized. The potentials corresponding to cold class parameters $\overset{\not\exists}{f}$ are absorbed by the fields $\overset{\not\exists}{f}$ in the resulting potential $\text{warm}[C]$.

In the second case, it leaks all potentials of the arguments to ensure that they are transitively initialized. The result potential is empty, as it must be a transitively initialized value.

Finally, to type check a block expression $e_0.f = e_1; e$, the typing rule $T\text{-}Block$ first type checks $e_0, e_1$ and $e$ separately. Then in the final effect, it leaks the potentials $\Pi_1$ for $e_1$, which ensures that the value of $e_1$ is transitively initialized. This is how monotonicity of initialization is enforced in the system.

### 2.3 Definition Typing

Definition typing (Figure 5) defines how programs, classes, fields and methods are checked. The check happens in two phases:

1. **first phase**: conventional type checking is performed and effect summaries are computed;
2. **second phase**: effect checking is performed to ensure initialization safety.
The two-phase checking is reflected in the typing rule T-Prog. When type checking a program \((\overline{C}, e)\), first each class is checked separately for well-typing and the effect summary for fields \(\Delta_c\) and methods \(S_c\) is computed. Then effect checking is performed modularly on each class with the help of the effect table \(\overline{E}\).

The typing rule T-Prog also checks that the top-level expression \(e\) is well-typed with the empty environment and the type \(\text{Null}\) for this. The effect signature means that the expression may not use this, otherwise the potentials and effects of \(e\) cannot be both empty. The usage of \(\text{Null}\) for the type of this for the top-level expression \(e\) corresponds to the semantic trick to use a dummy value of the type \(\text{Null}\) for this at the top-level. It unifies the semantics and typing rules for top-level expressions and expressions inside classes, which simplifies the meta-theory.

When type checking a class, the rule T-Class checks that the body fields and methods are well-typed, and the associated effects and potentials are computed. The effects and potentials associated with a field are the effects and fields of the right-hand-side expression. The effects and potentials of the method are computed by the potentials and effects of the arguments in the right-hand-side expression.

---

**Fig. 5. Definition Typing**

<table>
<thead>
<tr>
<th>Program Typing</th>
<th>(\Xi \vdash P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Xi = C \mapsto C)</td>
<td>(\Xi(\text{Null}) = \text{class Null} )</td>
</tr>
<tr>
<td>(\Xi; \emptyset; \text{Null} \vdash e : D ! (\emptyset, \emptyset))</td>
<td>(T-PKG)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class Checking</th>
<th>(\Xi; \overline{E} \vdash C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Delta_c) (\mapsto) (\overline{E}{f_1, \cdots, f_i-1}; C \vdash \Delta(f_i).f st)</td>
<td>(T-CHECK)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class Typing</th>
<th>(\Xi \vdash C ! (\Delta, S))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Xi; \overline{E} \vdash \text{class } C(f_i:D){\overline{F} \overline{M}})</td>
<td>(T-CLASS)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Typing</th>
<th>(\Xi; \emptyset; C \vdash F ! (\Phi, \Pi))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Xi; C \vdash \text{var } x : D \vdash e ! (\Phi, \Pi))</td>
<td>(T-FIELD)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Typing</th>
<th>(\Xi; C \vdash M ! (\Phi, \Pi))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Xi; C \vdash \text{def } m(x:T) : E \vdash e ! (\Phi, \Pi))</td>
<td>(T-METHOD)</td>
</tr>
</tbody>
</table>
potentials associated with a method are the effects and fields of the body expression of the method. The effect summaries are used during the second phase in T-CHECK, where it checks that given the already initialized fields, the effects on the right-hand-side of each field initialization are allowed.

The typing rule T-FIELD checks the right-hand-side expression $e$ in an empty typing environment, as there are no variables in a class body (class parameters are fields of their defining class). In the typing rule T-METHOD, the method parameters ($x : T$) are used as a typing environment to check the method body. They correspond to the semantics for field initialization and method calls respectively.

### 2.4 Effect Checking

The effect checking judgment $E; \Omega; C \vdash \Phi$ (Figure 6) means that the effects $\Phi$ are permitted inside class $C$ when the fields in $\Omega$ are initialized. In the checking, it first computes the fixed-point of $\Phi$ with the helper function $fix$. Then it checks that there is no leaking of this, warm values, or cold values which may not be fully initialized. Finally, it checks that each accessed field is in the set $\Omega$, i.e., only initialized fields are used.

The fixed-point computation is relatively simple: it just propagates the effects recursively until a fixed-point is reached. The fixed-point always exists as the domain of effects $\Phi$ is finite. In Figure 6,
we only show fixed-point computation for effects, the fixed-point computation for potentials looks similar, it suffices to replace \( \Phi \) with \( \Pi \). For simplicity, we use the notation \( E \vdash \Phi \sim \Phi' \) to mean for each \( \phi \in \Phi \), perform the propagation and then union all the results for each \( \phi \).

The main step in fixed-point computation is the propagation of effects and potentials. In effect propagation \( E \vdash \phi \sim \Phi \), field access \( \beta.f! \) is an atomic effect, thus it propagates to the empty set. For leaking effect \( \pi \uparrow \), we first propagate the potential \( \pi \) to a set of potentials \( \Pi \), and then leak each potential in \( \Pi \). For a method call effect \( C.this.m! \), it looks up the effects \( \Phi \) associated with the method \( m \), and then replace \( C.this \) with \( warm[C] \) in \( \Phi \).

In potentials propagation \( E \vdash \pi \sim \Pi \), \( \beta \) and \( cold \) propagates to empty, as they do not contain proxy aliasing information in the effect table. For a field potential like \( C.this.f \), it just looks up the potentials associated with the field \( f \) from the effect table. For a method potential \( C.this.m \), it looks up the potentials associated with the method \( m \) from the effect table. For potentials \( warm[C].f \) and \( warm[C].m \), first it looks up the potential associated with \( f \) and \( m \) respectively, then it replaces \( C.this \) in the potentials with \( warm[C] \).

2.5 Extensions

The value of a formal system depends crucially on its extensibility. We describe two extensions that are implemented in the prototype (Section ??).

**Functions.** Nowadays most languages combine object-oriented programming with functional programming, such as Java, Scala, Swift. To support functions, we add a new potential \( Fun(\Phi, \Pi) \), where \( \Phi \) is the set of effects to be triggered when the function is called, while \( \Pi \) is the set of potentials for the result of the function call. The effect domain is still finite, as the set of function potentials is constrained by the number of function literals in a given program.

The addition improves expressiveness. For example, it enables the following code, which is rejected in Swift:

```plaintext
1 class Rec {
2   val even = (n: Int) => n == 0 || odd(n - 1)
3   val odd = (n: Int) => n == 1 || even(n - 1)
4   val flag: Boolean = odd(6)
5 }
```

In functional programming, the recursive binding construct `letrec` may introduce similar initialization patterns as the code above. With the latest checker [6], OCaml still does not support the code above in the same `let rec`.

**Dependent Potentials.** Inner classes [3] introduce more complexity. A reasonable initialization pattern found in Scala code is the interaction between inner and outer classes [4], as illustrated with the example from the Dotty compiler:

```plaintext
1 class Trees {
2   private var counter: Int = 0
3   class ValDef { counter += 1 }
4   val theEmptyValDef = new ValDef
5 }
```

The code above is semantically equivalent to the following after lowering:

```plaintext
1 class Trees {
2   var counter: Int = 0
3   val theEmptyValDef = new ValDef(this)
4 }
```
class ValDef(outer: Trees) { outer.counter += 1 }

In the formal system, the code above has to be rejected even if we automatically mark outer as @cold, because we cannot access the field outer.counter of the cold object outer. To support the code, we would like to record the aliasing information outer = Trees.this in the system. That is the idea of dependent potentials.

The dependent potential \( C[\tilde{f}_i = \Pi_i] \) denotes a warm object of the type \( C \), where the cold class parameters \( \tilde{f}_i \) are bound to the potentials \( \Pi_i \). We also need to introduce object construction effects \( C[\tilde{f}_i = \Pi_i].init! \) to check the effects that may happen on the actual arguments to cold class parameters. For example, in the code above, summarizing the class ValDef will associate the effect ValDef.this.outer.counter! with the constructor. Now in checking the class Trees, the effect expands as follows:

\[
\text{ValDef[outer = Trees.this].init!} \\
\Rightarrow \text{ValDef[outer = Trees.this].outer.counter!} \\
\Rightarrow \text{Trees.this.counter!}
\]

As the field counter is already initialized at the point, the code above will be accepted by the checker.

Naive addition of dependent potentials will make the analysis non-terminating, as the following example shows:

class C(c: C @cold) { val c2 = new C(this) }

The non-termination can be seen from the expansion:

\[
C[c = C.this].init! \\
\Rightarrow C[c = C[c = C.this]].init! \\
\Rightarrow C[c = C[c = C[c = C.this]]].init!
\]

We resort to a standard technique in abstract interpretation, widening [2]. As a dependent potential is always warm, it suffices to widen a dependent potential to warm if it exceeds some size limit, e.g.:

\[
C[c = C.this].init! \\
\Rightarrow C[c = C[c = C.this]].init! \\
\Rightarrow C[c = warm[c]].init!
\]

This guarantees that the expansion of effects always reaches a fixed point.

3 META-THEORY

To show that the type system is sound, we follow the approach of definitional interpreters [1, 8]. In order to reason about the definitional interpreter formally as a function, it has to be total. It means we need to deal with non-termination and errors explicitly. The standard approach is to introduce a fuel \( k \) to deal with non-termination, and handle errors with an option monad. The skeleton of the adapted definitional interpreter looks as follows (for exposition in Scala):

```scala
def evalProg(p: Prog)(k: Int): Option[Option[(Addr, Store)]] = 
  eval(p.expr)(p.ct, Map(0 -> Obj(Null, Map.empty)), Map.empty, 0, k)

  Option[Option[(Addr, Store)]] = k match {
  case 0 => None // timeout
  case n =>
```
As seen from the code above, we use None to represent timeouts, Some(None) to represent errors, and Some(Some(l, s)) to represent the successful result. Apart from the handling of timeouts and errors, the adapted interpreter is the same as the semantics given in Figure 1, thus we omit the details.

The soundness statement says that well-typed programs do not go wrong, which is formulated as follows:

**Proposition 3.1 (Soundness).** If $\vdash P$, then $\forall k. \text{evalProg}(P)(k) = \text{Some}(r) \implies r \neq \text{None}.$

Proof for soundness requires some semantic definitions and a list of lemmas. Eventually, it follows from the main lemma 3.27.

### 3.1 Definitions

As memory may form cycles, we introduce store typing to type check memory store, which is a standard technique to avoid coinduction [5, Chapter 13].

$$\Sigma \in \text{StoreTyping} = \text{Loc} \rightarrow \text{Type} \times \text{Mode}$$

$$\Omega \in \text{Fields} = \{ f_1, f_2, \cdots \}$$

$$\mu \in \text{Mode} = \text{hot} \mid \text{warm} \mid \Omega$$

We augment the conventional store typing which is a mapping from locations to types with initialization information $\mu \in \text{Mode}$. The mode $\mu$ may take the following values:

- **hot**: the value is transitively initialized
- **warm**: all fields of the object are assigned, but some fields are not hot
- **$\Omega$**: the fields in $\omega$ are assigned, whose value can be in any mode

We show the definition of subtyping for type-modes, store typing, store typing ordering and environment typing in Figure 7. Some of the definitions (here and below) will require class table $\Xi$ and effect table $\mathcal{E}$. As they never change for a given program, we implicitly assume they are available when needed instead of thread through them everywhere. Most of the definitions are standard, thus we omit detailed discussion. The ordering of store typing $\Sigma_1 \leq \Sigma_2$ gives us an easy way to define monotonicity of initialization. The principle stacked initialization is formally defined by the predicate stacked.

The semantic typing for values and objects are presented in Figure 8. The value typing judgment $\Sigma \vdash l : C^{\mu}$ is standard, it observes subtyping of type-mode.

The semantics for modes hot, warm and $\Omega$ is defined by object typing. The object typing judgment has the form $\Sigma; l \vdash o : C^{\mu}$, where $l$ is the address of the object $o$. The address of the object is required to give semantic meaning of potentials, which represents aliasing information.

As expected, in the typing judgment $\Sigma; l \vdash o : C^{\mu}$, all fields are required to be assigned, and in $\Sigma \vdash o : C^{\text{hot}}$ the fields are required to be transitively hot. A noticeable element in the definition is the usage of potential typing judgment $\Sigma; l \vdash \omega(f) : \text{potential}(C, f)$, which checks that the values stored in the fields of the object agree with the aliasing information.

The semantics for potentials and effects are defined in Figure 9. Note that both potential typing and effect typing are defined relative to the value $\psi$ for this — that is partly expected, as potentials and effects are in essence tracking aliasing and field accesses relative to this respectively.
Type-Mode Subtyping

\[ C^{μ_1} <: C^{μ_2} \]
\[ C^{hot} <: C^{warm} \]
\[ C^{μ_1} <: C^{μ_2} \quad C^{μ_2} <: C^{μ_3} \]
\[ Ω \subseteq \text{fields}(C) \quad Ω_2 \subseteq Ω_1 \]
\[ C^{warm} <: C^{Ω} \quad C^{Ω_1} <: C^{Ω_2} \]

Store typing

\[ ∀ l \in \text{dom}(Σ). \; Σ ; l \not\equiv σ(l) : Σ(l) \]
\[ Σ \not\equiv σ \]

Store typing ordering

\[ \text{dom}(Σ_1) \subseteq \text{dom}(Σ_2) \quad ∀ l \in \text{dom}(Σ_1). \; Σ_2 \not\equiv l : Σ_1(l) \]
\[ Σ_1 \not\equiv Σ_2 \]

Environment typing

\[ \emptyset ; Σ \not\equiv \emptyset \]
\[ Γ ; Σ \not\equiv ρ \quad Σ \not\equiv l : C^{μ} \]
\[ Γ ; x:C ; Σ \not\equiv ρ , x;l \]

Stacked

\[ \text{stacked}(Σ', Σ) \triangleq ∀ l \in \text{dom}(Σ'). \; Σ' \not\equiv l : C^{warm} \lor l \in \text{dom}(Σ) \]

Fig. 7. Definitions of type-mode subtyping, environment typing, store typing and its ordering

In potential typing Σ;ψ ; l : Π, a location l may take any potential Π if l is hot. In particular, Π can be an empty set. Otherwise, there should exist π ∈ fix(E, Π), such that one of the following cases hold:

- π = C,this: l should be equal to ψ
- π = warm[C]: l should be warm
- π = cold: l must be well typed

An immediate result of this definition is that if Σ;ψ ; l : Π, then either l is hot or there exists π in the set fix(E, Π) such that Σ;ψ ; l : Π. That is exactly what we want the type-and-effect to do: if an expression may take a value that is not fully initialized, then it should have a potential that aliases a potentially uninitialized value through β or cold.

In an effect typing judgment Σ;ψ ; Φ, we ensure that all effects Φ are allowed given the value ψ for this and the store typing Σ. This is enforced by first computing the fixed point of Π, and then ensure that each effect φ in Φ′ is allowed. There are several cases depending on φ:

- φ = π↑: if π is β or cold, then ψ must be hot
- φ = C,this.f!: f must be an initialized field of ψ
- φ = warm[C].f!: field access on warm values are fine
- φ = β.m!: placeholder effects are ignored
### Value Typing

\[ \Sigma(I) = C^{\mu'} \quad C^{\mu'} <; C^{\mu} \]

**Object Typing**

\[ (C, \omega) = o \quad \text{classParameters}(C) \subseteq \text{dom}(\omega) \]
\[ \forall f \in \text{dom}(\omega). \ D = \text{fieldType}(C, f) \implies \Sigma; l \models \omega(f) : \text{potential}(C, f) \]
\[ \Sigma \models o : C^{\text{dom}(\omega)} \]

\[ (C, \omega) = o \quad \forall f \in \text{fields}(C). \ D = \text{fieldType}(C, f) \implies \Sigma; l \models \omega(f) : \text{potential}(C, f) \]
\[ \Sigma \models o : C^{\text{warm}} \]

\[ (C, \omega) = o \quad \forall f \in \text{fields}(C). \ D = \text{fieldType}(C, f) \implies \Sigma \models \omega(f) : D^{\text{hot}} \]
\[ \Sigma \models o : C^{\text{hot}} \]

### Field Potential

- \( \text{potential}(C, \tilde{f}) = \{ \text{cold} \} \)
- \( \text{potential}(C, \hat{f}) = \emptyset \)
- \( \text{potential}(C, f) = \{ C\.this.f \} \)

Fig. 8. Value and object typing

### 3.2 Lemmas

**Lemma 3.2 (Value Typing Monotonicity).** For all \( \Sigma_1, \Sigma_2, A \) and value \( l \), if \( \Sigma_1 \preceq \Sigma_2 \) and \( \Sigma_1 \models l : C^{\mu} \), then \( \Sigma_2 \models l : C^{\mu} \).

**Proof.** By the definition of store typing ordering and value typing. \( \square \)

**Lemma 3.3 (Environment Typing Monotonicity).** For all \( \Sigma_1, \Sigma_2, \Gamma \) and environment \( \rho \), if \( \Sigma_1 \preceq \Sigma_2 \) and \( \Gamma ; \Sigma_1 \models \rho \), then \( \Gamma ; \Sigma_2 \models \rho \).

**Proof.** By induction on environment typing and Lemma 3.2. \( \square \)

**Lemma 3.4 (Potential Typing Monotonicity).** For all \( \Sigma, \Sigma', \Pi, \psi, l \), if \( \Sigma \preceq \Sigma' \) and \( \Sigma, \psi \models l : \Pi \), then \( \Sigma', \psi \models l : \Pi \).

**Proof.** By the definition of potential typing there are two cases.

- **Case** \( \Sigma \models l : C^{\text{hot}} \).
  
  By Lemma 3.2, we have \( \Sigma' \models l : C^{\text{hot}} \). Use the definition of potential typing we have \( \Sigma'; \psi \models \Pi \).

- **Case** \( \Sigma; \psi \models l : \pi \).
  
  There are four cases depending on the shape of \( \pi \):

  - **Case** \( \pi = C\.this \)
    
    We have \( l = \psi \), use the same rule we get \( \Sigma'; \psi \models l : C\.this \). Now use the definition of potential typing.
Potential Typing

\[
\begin{align*}
\Sigma; \psi \not\vdash l : C^{hot} & \quad \exists \pi \in \text{fix}(E, \Pi). \Sigma; \psi \not\vdash l : \pi \\
\Sigma; \psi \not\vdash l : C.this & \quad \Sigma; \psi \not\vdash l : C^{warm} \\
\Sigma; \psi \not\vdash l : \text{warm}[C] & \quad \Sigma; \psi \not\vdash l : D^{\mu} \\
l = \psi & \quad \Sigma; \psi \not\vdash l : \text{cold}
\end{align*}
\]

Effect Typing

\[
\begin{align*}
\Phi' = \text{fix}(E, \Phi) & \quad \forall \phi \in \Phi'. \Sigma; \psi \not\vdash \phi \\
\Sigma; \psi \not\vdash \Phi & \quad \Sigma; \psi \not\vdash C^{\Omega} \\
f \in \Omega & \quad \Sigma; \psi \not\vdash C.this.f! \\
\pi = \beta \lor \pi = \text{cold} & \Rightarrow \Sigma; \psi \not\vdash C^{hot} \\
\Sigma; \psi \not\vdash \pi\uparrow & \quad \Sigma; \psi \not\vdash \beta.m!
\end{align*}
\]

Fig. 9. Definition of potential typing and effect typing

- case \( \pi = \text{warm}[C] \)
  We have \( \Sigma \vdash l : C^{\text{warm}} \). Use Lemma 3.2, we have \( \Sigma' \not\vdash l : C^{\text{warm}} \). Now use the definition of potential typing.
- case \( \pi = \text{cold} \)
  We have \( \Sigma \vdash l : D^{\mu} \). Use Lemma 3.2, we have \( \Sigma' \not\vdash l : C^{\mu} \). Now use the definition of potential typing.

\[\square\]

Lemma 3.5 (Object Typing Monotonicity). For all \( \Sigma, \Sigma', T, \Pi, l \) and object \( o \), if \( \Sigma \preceq \Sigma' \) and \( \Sigma; l \vdash o : C^{\mu} \), then \( \Sigma' ; l \vdash o : C^{\mu} \).

Proof. By definition of object typing, Lemma 3.2 and Lemma 3.4. \[\square\]

Lemma 3.6 (Effect Typing Monotonicity). For all \( \Sigma, \Sigma', \Phi \) and value \( \psi \), if \( \Sigma \preceq \Sigma' \) and \( \Sigma, \psi \vdash \Phi \), then \( \Sigma' , \psi \vdash \Phi \).

Proof. By the definition of effect typing and Lemma 3.2. \[\square\]

Lemma 3.7 (Hot Transitivity). If \( \Sigma \vdash l : C^{hot} \) and \( \Sigma \vdash \sigma \), then for all \( l' \), reachable(\( \sigma, l, l' \)) and \( (D, \_ ) = \sigma(l') \) implies \( \Sigma \vdash l' : D^{\text{hot}} \).

Proof. By induction on the definition of reachable and the definition of value typing. \[\square\]

Lemma 3.8 (Environment Regularity). If \( \Gamma; \Sigma \not\vdash \rho \) and \( \Xi; \Gamma; C \vdash \rho(x) : D^{\mu} \).

Proof. By induction on \( \Gamma \). The case \( \Gamma = \emptyset \) trivially holds as the precondition \( \Xi; \emptyset; C \vdash x : D \) does not hold.

, Vol. 1, No. 1, Article . Publication date: January 2020.
In the case $\Gamma = \Gamma', y:T$ and $\rho = \rho', y:l$, we have $\Sigma \vdash l : T^\rho$. If $x = y$, then it must be $T = D$ ¹, we get the result immediately. Otherwise, we must have $\Xi;\Gamma' ; C \vdash x : D$, now use the induction hypothesis and we get $\Sigma \vdash \rho'(x) : D^\rho$. The conclusion follows immediately due to $\rho'(x) = \rho(x)$. □

**Lemma 3.9 (Potential Typing Regularity).** If $\Sigma; \psi \vdash l : \Pi$, then $\Sigma \vdash l : D^{\text{hot}}$ or $\beta \in \text{fix}(E, \Pi)$ or $\text{cold} \in \text{fix}(E, \Pi)$.

**Proof.** By the definition of potential typing, there are two cases. If $\Sigma \vdash l : D^{\text{hot}}$, which completes the proof goal immediately. Otherwise, there exists $\pi \in \text{fix}(E, \Pi)$ such that $\Sigma; \psi \vdash l : \pi$. There are three cases, for each case we have $\pi = \beta$ or $\pi = \text{cold}$.

**Lemma 3.10 (Empty Potential Regularity).** If $\Sigma; \psi \vdash l : \emptyset$, then $\Sigma \vdash l : D^{\text{hot}}$.

**Proof.** By definition of potential typing, the only possibility is $\Sigma \vdash l : D^{\text{hot}}$.

**Lemma 3.11 (Hot Effect Regularity).** If $\Sigma \vdash \psi : C^{\text{hot}}$, then $\Sigma; \psi \vdash \Phi$.

**Proof.** By the definition of effect typing.

**Lemma 3.12 (Field Check Regularity).** If $E; \Omega; C \vdash \Phi$ and $\Sigma \vdash \psi : C^{\Omega}$, then $\Sigma; \psi \vdash \Phi$.

**Proof.** Let $\Phi' = \text{fix}(E, \Phi)$, we need to prove that for all $\phi \in \Phi'$, we have $\Sigma; \psi \vdash \phi$. If $\Phi' = \emptyset$, then the conclusion holds trivially by the definition of effect typing.

The only cases that do not hold trivially is when $\phi = \pi \dagger$ or $\phi = C\.this.f!$. In the first case, from the definition of $E; \Omega; C \vdash \Phi$, we know $\beta \dagger \notin \Phi'$ and $\text{cold} \dagger \notin \Phi'$. Thus, $\pi \notin \beta$ and $\pi \notin \text{cold}$, $\Sigma; \psi \vdash \pi \dagger$ holds trivially. In the second case $\phi = C\.this.f!$, and from effect checking we have $f \in \Omega$. Now use the definition of effect typing, the result follows immediately. □

**Lemma 3.13 (Effect Potential Cancellation).** Given

- $\Sigma \sqsubseteq \Sigma'$
- $\Sigma; \psi \vdash \Pi \dagger$
- $\Sigma'; \psi \vdash l : \Pi$
- $\Sigma \vdash \psi : C^{\text{hot}} \implies \Sigma' \vdash l : D^{\text{hot}}$

Then we have

- $\Sigma'; \psi \vdash l : D^{\text{hot}}$

**Proof.** From Lemma 3.9, we have either $\beta$ or $\text{cold} \in \text{fix}(E, \Pi)$, or $\Sigma; \psi \vdash l : D^{\text{hot}}$. We only need to consider $\beta$ or $\text{cold} \in \text{fix}(E, \Pi)$. From the definition of $\Sigma; \psi \vdash \Pi \dagger$, we know $\Sigma; \psi \vdash \beta \dagger$ or $\Sigma; \psi \vdash \text{cold} \dagger$. In each case, we have $\Sigma \vdash \psi : C^{\text{hot}}$. Now use the premise, we get $\Sigma'; \psi \vdash l : D^{\text{hot}}$. □

**Lemma 3.14 (Potential Weakening).** If $\Sigma; \psi \vdash l : \Pi$ and $\Pi \sqsubseteq \text{fix}(E, \Pi')$, then $\Sigma; \psi \vdash l : \Pi'$.

**Proof.** From the definition of potential typing, there are two cases: either $\Sigma \vdash l : D^{\text{hot}}$ or $\exists \pi \in \text{fix}(E, \Pi).\Sigma; \psi \vdash l : \pi$. In the first case, apply the definition of potential typing, we get the result immediately. We only need to consider the second case, where $\Sigma; \psi \vdash l : \pi$.

From $\pi \in \text{fix}(E, \Pi)$ and $\Pi \sqsubseteq \text{fix}(E, \Pi')$, we have $\pi \in \text{fix}(E, \Pi')$ from the definition of $\text{fix}$. The result follows immediately from the fact that $\Sigma; \psi \vdash l : \pi$ and $\pi \in \text{fix}(E, \Pi')$. □

**Lemma 3.15 (Potential Strengthening).** If $\Sigma; \psi \vdash l : \Pi$, then there exists $\pi \in \Pi$ such that $\Sigma; \psi \vdash l : \{\pi\}$.

¹Remember that as a convention, we assume variables in the environment $\Gamma$ are unique.
Proof. From the definition of potential typing, there are two cases: either $\Sigma \models l : D^{\text{hot}}$ or $\exists \pi \in \text{fix}(E, \Pi), \Sigma; \psi \models l : \pi$. In the first case, apply the definition of potential typing, we get the result immediately. We only need to consider the second case, where there exists $\pi_0 \in \text{fix}(E, \Pi)$ such that $\Sigma; \psi \not\models l : \pi_0$.

From the definition of $\text{fix}$, we have $\text{fix}(E, \Pi) = \bigcup \text{fix}(E, \pi_i)$ with $\Pi = \{ \pi_i \}$. Thus, there must exist $k$ such that $\pi_0 \in \text{fix}(E, \pi_k)$. Now we have $\Sigma; \psi \not\models l : \{ \pi_k \}$. □

Lemma 3.16 (Effect Weakening). If $\Sigma; \psi \models \Phi$ and $\Phi' \subseteq \text{fix}(E, \Phi)$, then $\Sigma; \psi \not\models \Phi'$.

Proof. From the definition of effect typing, we know $\forall \phi \in \text{fix}(E, \Phi), \Sigma; \psi \models \phi$. From $\Phi' \subseteq \text{fix}(E, \Phi)$, we have $\text{fix}(E, \Phi') \subseteq \text{fix}(E, \Phi)$ from the definition of $\text{fix}$. Thus it follows that $\forall \phi \in \text{fix}(E, \Phi'), \Sigma; \psi \models \phi$. The result follows immediately from the definition of effect typing. □

Lemma 3.17 (Potential View Change - Field). If $\Sigma; l_0 \not\models l : \{ T_0, \text{this}, f \}$ and $\Sigma; \psi \not\models l_0 : \text{warm}[T_0]$, then $\Sigma; \psi \not\models l : \{ \text{warm}[T_0], f \}$.

Proof. From the definition of potential typing, there are two cases: either $\Sigma \models l : D^{\text{hot}}$ or $\exists \pi \in \text{fix}(E, \Pi), \Sigma; \psi \models l : \pi$. In the first case, apply the definition of potential typing, we get the result immediately. We only need to consider the second case. There are following possibilities for $\pi$:

- $\pi = T_0, \text{this}$
  We have $l = l_0$ by the definition of potential typing. As $T_0, \text{this} \in \text{fix}(E, \{ T_0, \text{this}, f \})$, we must have $\text{warm}[T_0] \in \text{fix}(E, \{ \text{warm}[T_0], f \})$ from the definition of $\text{fix}$. The result follows from $\Sigma; \psi \not\models l : \text{warm}[T_0]$.

- $\pi = \text{warm}[T_x]$  
  We have $\Sigma \models l : T^{\text{warm}}$. As $\text{warm}[T_x] \in \text{fix}(E, \{ T_0, \text{this}, f \})$, we must have $\text{warm}[T_x] \in \text{fix}(E, \{ \text{warm}[T_x], f \})$ from the definition of $\text{fix}$. The result follows from $\Sigma; \psi \not\models l : \text{warm}[T_x]$.

- $\pi = \text{cold}$
  We have $\Sigma \not\models l : D^{\mu}$. The result follows from the potential typing for cold. □

Lemma 3.18 (Potential View Change - Method). If $\Sigma; l_0 \not\models l : \{ T_0, \text{this}, m \}$ and $\Sigma; \psi \not\models l_0 : \text{warm}[T_0]$, then $\Sigma; \psi \not\models l : \{ \text{warm}[T_0], m \}$.

Proof. Similar as the Lemma 3.17 above. □

Lemma 3.19 (Collective Maturity). For all $\Sigma, \sigma, l, C$, if the following conditions hold:

1. $\Sigma \models l : C^{\text{warm}}$
2. $\Sigma \models \sigma$
3. if for all $l'$, $(D, _) = \sigma(l')$ and reachable($\sigma, l, l'$), then $\Sigma \not\models l' : D^{\text{warm}}$, 

then there exists $\Sigma'$ such that:

1. $\Sigma' \not\models \sigma$ and $\Sigma \preceq \Sigma'$ and
2. $\Sigma' \not\models l : C^{\text{hot}}$.

Proof. Define $\Sigma'$ as follows:

$$\Sigma'(l') = \begin{cases} D^{\text{hot}} & \text{if reachable}(\sigma, l, l') \text{ and } \Sigma(l') = D^{\mu}, \\ \Sigma(l') & \text{otherwise} \end{cases}$$

It is straightforward to show that $\Sigma \preceq \Sigma'$ and $\Sigma \not\models l : C^{\text{hot}}$ as reachable($\sigma, l, l$). We must then prove $\Sigma' \not\models \sigma$, that is, $\forall l \in \text{dom}(\Sigma'). \Sigma' \not\models \sigma(l) : \Sigma'(l)$.  

, Vol. 1, No. 1, Article . Publication date: January 2020.
In words, we must show that all objects in $\sigma$ are well-typed in $\Sigma'$. For any location $l'$ that is not reachable from $l$, we have $\Sigma'(l') = \Sigma(l')$, so we need to show $\Sigma' \vdash \sigma(l') : \Sigma'(l')$, which follows from object typing monotonicity (Lemma 3.5).

If instead $\text{reachable}(\sigma, l, l')$, we must show $\Sigma' \vdash \sigma(l') : \Sigma'(l')$. That follows because:

- By the definition of $\Sigma'$, all values $l'$ reachable from $l$ have the type $T^{hot}$, thus the fields of the object referred by $l'$ are hot.
- From premises, the value $l'$ is warm, thus all fields that are defined in its class are assigned.

\[\square \]

**Lemma 3.20 (Scope Transitivity).** Given

- $\text{dom}(\sigma_1) \subseteq \text{dom}(\sigma_2)$
- $\text{scoped}(\sigma_2, \sigma_1, L_2, L_1)$
- $\text{scoped}(\sigma_3, \sigma_2, L_3, L_2)$

then

- $\text{scoped}(\sigma_3, \sigma_1, L_3, L_1)$

**Proof.** From the definition of $\text{scoped}$, we need to prove that for any $l_3 \in L_3$ and $l \in \text{dom}(\sigma_1)$, if (A1) holds, then (A2) holds too:

- (A1) $\text{reachable}(\sigma_3, l_3, l)$
- (A2) $\exists l_1 \in L_1. \ \text{reachable}(\sigma_1, l_1, l)$

From $l \in \text{dom}(\sigma_1)$ and $\text{dom}(\sigma_1) \subseteq \text{dom}(\sigma_2)$, we have

- (B1) $l \in \text{dom}(\sigma_2)$

Use (A1), (B1) and $\text{scoped}(\sigma_3, \sigma_2, L_3, L_2)$, we have

- (C1) $\exists l_2 \in L_2. \ \text{reachable}(\sigma_2, l_2, l)$

Pick $l_2$ in (C1), now use (C1) and $\text{scoped}(\sigma_2, \sigma_1, L_2, L_1)$, we arrive at (A2) immediately. \[\square \]

**Lemma 3.21 (Preserving Transitivity).** Given

- $\text{preserving}(\sigma_2, \sigma_1, L_1)$
- $\text{preserving}(\sigma_3, \sigma_2, L_2)$
- $\text{scoped}(\sigma_2, \sigma_1, L_2, L_1)$

then

- $\text{preserving}(\sigma_3, \sigma_1, L_1)$

**Proof.** From the definition of $\text{preserving}$, we need to prove that for any $\sigma_0$, $L \subseteq \text{dom}(\sigma_1)$ and $L_0 \subseteq \text{dom}(\sigma_0)$, if (A1) and (A2) hold, then (A3) holds too:

- (A1) $\text{scoped}(\sigma_1, \sigma_0, L_1, L_0)$
- (A2) $\text{scoped}(\sigma_1, \sigma_0, L, L_0)$
- (A3) $\text{scoped}(\sigma_3, \sigma_0, L, L_0)$

From (A1), (A2) and $\text{preserving}(\sigma_2, \sigma_1, L_1)$, we have

- (B1) $\text{scoped}(\sigma_2, \sigma_0, L, L_0)$

From the premise $\text{scoped}(\sigma_2, \sigma_1, L_2, L_1)$ and (A1), we use Lemma 3.20:

- (C1) $\text{scoped}(\sigma_2, \sigma_0, L_2, L_0)$

From (B1), (C1) and $\text{preserving}(\sigma_3, \sigma_2, L_2)$, we arrive at (A3) immediately. \[\square \]

**Lemma 3.22 (Preserving Regularity).** If $\text{dom}(\sigma_0) \subseteq \text{dom}(\sigma_1)$, $\text{preserving}(\sigma_2, \sigma_1, L_1)$ and $\text{scoped}(\sigma_1, \sigma_0, L_1, L_0)$ then $\text{scoped}(\sigma_2, \sigma_0, L_1, L_0)$. 


**Proof.** From the definition of \( preserving(\sigma_2, \sigma_1, L_1) \), choose \( \sigma_0 = \sigma_1, L = L_1, L_0 = L_1 \), the preconditions hold trivially, thus we have \( scoped(\sigma_2, \sigma_1, L_1, L_1) \). Now by transitivity of scoping (Lemma 3.20), we have \( scoped(\sigma_2, \sigma_1, L_1, L_0) \). □

**Lemma 3.23 (Preserving Regularity - Degenerate Case).**
If \( preserving(\sigma_2, \sigma_1, L_1) \), then \( scoped(\sigma_2, \sigma_1, L_1, L_1) \).

**Proof.** Use Lemma 3.22 by \( \sigma_0 = \sigma_1 \) and \( L_0 = L_1 \), the preconditions hold trivially, thus we have \( scoped(\sigma_1, \sigma_1, L_1, L_1) \). □

**Lemma 3.24 (Preserving Transitivity - Degenerate Case).**
Given
- \( preserving(\sigma_2, \sigma_1, L_1) \)
- \( preserving(\sigma_3, \sigma_2, L_1) \)

then
- \( preserving(\sigma_3, \sigma_1, L_1) \)

**Proof.** Use Lemma 3.21 and Lemma 3.23. □

**Lemma 3.25 (Assignment Preserving).**
Given
- \( preserving(\sigma_2, \sigma_1, L_1) \)
- \( scoped(\sigma_2, \sigma_1, l, L_1) \)
- \( (C, \omega) = \sigma_2(l) \)
- \( \omega' = [f \mapsto l]_\omega \)
- \( \sigma'_2 = [l' \mapsto (C, \omega')]|\sigma_2 \)

then
- \( preserving(\sigma'_2, \sigma_1, L_1) \)

**Proof.** From the definition of \( preserving \), we need to prove that for any \( \sigma_0, L \subseteq dom(\sigma_1), L_0 \subseteq dom(\sigma_0) \), if (A1) and (A2) hold, then (A3) holds:
- (A1) \( scoped(\sigma_1, \sigma_0, L_1, L_0) \)
- (A2) \( scoped(\sigma_1, \sigma_0, L, L_0) \)
- (A3) \( scoped(\sigma'_2, \sigma_0, L, L_0) \)

From (A1), (A2) and \( preserving(\sigma_2, \sigma_1, L_1) \), we have
- (B1) \( scoped(\sigma_2, \sigma_0, L, L_0) \)

From \( scoped(\sigma_2, \sigma_1, l, L_1) \), (A1) and Lemma 3.20, we have
- (C1) \( scoped(\sigma_2, \sigma_0, l, L_0) \)

To prove (A3), we need to show that for any \( l_2 \in L \) and \( l_0 \in dom(\sigma_0) \), if (D1) holds, then (D2) holds too:
- (D1) \( reachable(\sigma'_2, l_2, l_0) \)
- (D2) \( \exists l \in L_0 . reachable(\sigma_0, l, l_0) \)

For (D1), we consider the path that begins from \( l_2 \) to \( l_0 \).
If the path does not contain the segment \( (l', l) \), then the path must exist in \( \sigma_2 \), i.e. \( reachable(\sigma_2, l_2, l_0) \).
Now use (B1) we arrive at (D2) immediately.
If the path contains the segment \( (l', l) \), then we have \( reachable(\sigma_2, l, l_0) \). Now use (C1), we arrive at (D2) too. □

**Lemma 3.26 (Stacked Transitivity).**
If \( stacked(\Sigma_2, \Sigma_1) \) and \( stacked(\Sigma_3, \Sigma_2) \), then \( stacked(\Sigma_3, \Sigma_1) \).

**Proof.** From premises, we have
Technical Report: A Type-and-Effect System for Object Initialization

• (A1) \( \forall l_3 \in \text{dom}(\Sigma_3). \Sigma_3 \vdash l_3 : \text{C}^{\text{warm}} \lor l_3 \in \text{dom}(\Sigma_2) \)
• (A2) \( \forall l_2 \in \text{dom}(\Sigma_2). \Sigma_2 \vdash l_2 : \text{C}^{\text{warm}} \lor l_2 \in \text{dom}(\Sigma_1) \)

From (A1) and (A2), it is easy to see
• (B1) \( \forall l_3 \in \text{dom}(\Sigma_3). \Sigma_3 \vdash l_3 : \text{C}^{\text{warm}} \lor l_3 \in \text{dom}(\Sigma_1) \)

(B1) is exactly the definition of \( \text{stacked}(\Sigma_3, \Sigma_1) \).

3.3 Main Lemma

**Lemma 3.27 (Expression Soundness).** Given
(1) \( \Xi \) is well-typed
(2) \( \Xi ; \Gamma ; C \vdash e : D ! (\Phi, \Pi) \)
(3) \( \Gamma ; \Sigma \vdash \rho \)
(4) \( \Sigma \vdash \psi : C^{\mu} \)
(5) \( \forall l \in \text{codom}(\rho). \Sigma ; \psi \vdash l : \emptyset \)
(6) \( \Sigma ; \psi \vdash \Phi \)
(7) \( \text{eval}(e)(\Xi, \rho, \sigma, \psi)(k) = \text{Some}(\text{result}) \)

then there exists \( l, \mu', \sigma', \Sigma' \) such that:
(a) \( \text{result} = \text{Some}(l, \sigma') \)
(b) \( \Sigma \ll \Sigma' \) and \( \Sigma' \vdash \sigma' \)
(c) \( \Sigma' \vdash l : D^{\mu'} \)
(d) \( \Sigma'; \psi \vdash l : \Pi \)
(e) \( \text{scoped}(\sigma', \sigma, l, \psi \cup \text{codom}(\rho)) \)
(f) \( \text{stacked}(\Sigma', \Sigma) \)
(g) \( \text{preserving}(\sigma', \sigma, \psi \cup \text{codom}(\rho)) \)
(h) \( \Sigma \vdash \psi : C^{\text{hot}} \implies \Sigma' \vdash l : D^{\text{hot}} \)

**Proof.** By induction on the typing rule \( \Xi ; \Gamma ; C \vdash e : D ! (\Phi, \Pi) \).

• **case** T-VAR. \( e = x \)

From the typing rule T-VAR, we have the following:
- (A1) \( \Phi = \emptyset \)
- (A2) \( \Pi = \emptyset \)

Choose \( l = \rho(x), \sigma' = \sigma, \Sigma' = \Sigma \):
(b) hold trivially from premises.
(c) holds from Lemma 3.8.
(d) holds from the premises (6) and we know \( \mu' = \text{hot} \) from Lemma 3.10.
(e) holds by choosing the existential to be \( \rho(x) \).
(f) holds trivially due to \( \Sigma' = \Sigma \).
(g) holds trivially due to \( \sigma' = \sigma \).
(h) holds trivially due to \( l \) is hot.

• **case** T-THIS. \( e = \text{this} \)

From the typing rule T-THIS, we have the following:
- (A1) \( \Phi = \emptyset \)
- (A2) \( \Pi = \{C\.this\} \)
- (A3) \( D = C \)

Choose \( l = \psi, \sigma' = \sigma, \Sigma' = \Sigma \):
(b), (c) and (d) hold trivially from premises.
(e) holds by choosing the existential to be \( \psi \).
(f) holds trivially due to $\Sigma' = \Sigma$.
(g) holds trivially due to $\sigma' = \sigma$.
(h) holds trivially due to $l = \psi$.

- **case T-Sel. $e = e_0.f$**

  From the typing rule T-Sel, we have the following:
  - (A1) $\Xi; \Sigma; C \vdash e_0 : E ! (\Phi_0, \Pi_0)$
  - (A2) $D = fieldType(\Xi, E, f)$
  - (A3) $(\Phi', \Pi') = select(\Pi_0, f)$
  - (A4) $\Phi = \Phi_0 \cup \Phi'$
  - (A5) $\Pi = \Pi'$

  From the induction hypothesis on $e_0$, we know there exists $l_0, \sigma'$ and $\Sigma'$ such that:
  - (B1) $\Sigma \lessdot \Sigma'$ and $\Sigma' \vdash \sigma'$
  - (B2) $\Sigma' \vdash l_0 : E^{\mu_0}$
  - (B3) $\Sigma'; \psi \vdash l_0 : \Pi_0$
  - (B4) $scoped(\sigma', \sigma, l_0, \psi \cup dom(\rho))$
  - (B5) $stacked(\Sigma', \Sigma)$
  - (B6) $preserving(\sigma', \sigma, \psi \cup dom(\rho))$
  - (B7) $\Sigma \vdash \psi : C^{\text{hot}} \implies \Sigma \vdash l_0 : E^{\text{hot}}$
  - (B8) $(E, \omega) = \sigma' (l_0)$

  Now do case analysis on $\Pi_0$
  - **case $\Pi_0 = \emptyset$**

    In this case, we know $l_0$ is hot:
    - (a0) $\Sigma \lessdot \Sigma'$ and $\Sigma' \vdash \omega (f) : \{\pi\}$
    
    Choose $\omega (f), \sigma'$ and $\Sigma'$.
    (b) hold from (B1).
    (c) holds from (a2).
    (d) holds from (a3).
    (e) holds from (B4), transitivity of reachability and reachable($\sigma', l_0, \omega (f)$).
    (f) holds from (B5).
    (g) holds from (B6).
    (h) holds trivially due to $\omega (f)$ is hot.
  - **case $\Pi_0 \neq \emptyset$**

    In this case, from Lemma 3.15 we know there exists $\pi \in \Pi_0$ such that
    - (a0) $\Sigma'; \psi \vdash l_0 : \{\pi\}$.

    There are four cases for $\pi$:
    - **case $\pi = E.this$**

      By definition of potential typing and object typing, we have
      - (a1) $l_0 = \psi$
      - (a2) $E = C$
      - (a3) $\forall f \in dom(\omega). D = fieldType(C, f) \implies \Sigma'; \psi \vdash \omega (f) : \{C.this.f\}$

      From (A4) and the definition of $select$, there are three cases. The first two are selection of class parameters, which are always initialized — the selection is safe guaranteed by object typing. We consider the general case $select(\beta, f)$:
      - (b1) $E.this.f! \in \Phi'$
From $\Sigma; \psi \vdash \Phi'$, we know $\Sigma; \psi \vdash \Phi'$ by monotonicity. Combined with (b1), we have

- (c1) $\Sigma'; \psi \vdash E.\text{this}.f$

From (c2) and (a3) we have

- (d1) $\Sigma'; \psi \vdash \omega(f)$

From (c2) and (a9) we have

- (e1) $\Sigma'; \psi \vdash \omega(f)$

Choose $\omega(f)$, $\sigma'$ and $\Sigma'$.

(b) holds from (B1).

(c) and (d) hold trivially because $\omega(f)$ is hot.

(e) holds from (B4), transitivity of reachability and $\text{reachable}(\sigma', l_0, \omega(f))$.

(f) holds from (B5).

(g) holds from (B6).

(h) holds from (B7) and definition of object typing for hot.

* case $\pi = \text{warm}[E]$  
Similar as the case above, except that the field $f$ exists because $l_0$ is warm. The potential typing (e) depends on the Lemma 3.17.

* case $\pi = \beta.f$

From (A4), we have

- (a1) $\beta.f \uparrow \in \Phi'$
- (a2) $\Pi' = \emptyset$

From premise (7), (A4), (a1) and monotonicity of effect typing (Lemma 3.6), we have:

- (b1) $\Sigma'; \psi \vdash \{\beta.f\}$

Now from (a0), (b1), (B2), we use the Lemma 3.13:

- (c1) $\Sigma' \vdash l_0 : E^\text{hot}$

As $l_0$ is hot, by object typing, we know $\omega(f)$ must be hot.

Choose $\omega(f)$, $\sigma'$ and $\Sigma'$.

(b) holds from (B1).

(c) and (d) hold trivially because $\omega(f)$ is hot.

(e) holds from (B4), transitivity of reachability and $\text{reachable}(\sigma', l_0, \omega(f))$.

(f) holds from (B5).

(g) holds from (B6).

(h) holds from (B7) and definition of object typing for hot.

* case $\pi = \beta.m$

Similar as the case above.

• case $T\text{-Call}$, $e = e_0.m(\vec{x})$

From the typing rule $T\text{-Call}$, we have the following:

- (A1) $\Xi; \Sigma; C \vdash e_0 : T_0 ! (\Phi_0, \Pi_0)$
- (A2) $\Xi; \Sigma; C \vdash e_i : T_i ! (\Phi_i, \Pi_i)$
- (A3) $(x_i;T_i, D) = \text{methodType}(\Xi, T_0, m)$
- (A4) $\Phi, \Pi' = \text{call}(T_0.m, \Pi_0)$
- (A5) $\Phi = \Phi_0 \cup \Phi_i \cup \Pi_i \uparrow \cup \Phi'$
- (A6) $\Pi = \Pi'$

From the induction hypothesis on $e_0$, we know there exists $l_0, \sigma_0, \Sigma_0$ such that:
We prepare the environment \( T \).

The method \( T_0,m \) is well typed, thus we have

\( \Xi; x_i:T_i; T_0 \vdash e_m : D ! (\Phi_m, \Pi_m) \)

We prepare the environment \( \rho' \) and \( \psi' \) for the method call as follows:

\(- (H1) \rho' = \overline{x_i:l_i} \)
\(- (H2) \psi' = l_0 \)

From (B4), (B6), (D4), (D6) and Lemma 3.22, we have

\(- (I1) scoped(\sigma_n, \sigma, \psi \cup codom(\rho'), \psi \cup codom(\rho)) \)
\(- (I2) preserving(\sigma_n, \sigma, \psi \cup codom(\rho)) \)

Now do case analysis on \( \Pi_0 \) in (B3):

\(- \text{case } \Pi_0 = \emptyset \)

In this case, we know \( l_0 \) is hot:

\(- (a1) \Sigma_n \vdash l_0 : T_0^{hot} \)

As both the receiver and method parameters are hot, we have:

\(- (c1) \Sigma_n; l_0 \vdash l_0 : T_0 \quad \text{this} \)
\(- (c2) \forall l \in codom(\rho'). \Sigma_n; l_0 \vdash l : \emptyset \)
\(- (c3) \Sigma_n; l_0 \vdash \Phi_m \)

Now we can use the induction hypothesis for (G1):
* (d1) \( \Sigma_n \preceq \Sigma_m \) and \( \Sigma_m \vdash \sigma_m \)
* (d2) \( \Sigma_m \vdash l_m : T^\mu_m \)
* (d3) \( \Sigma_m ; l_0 \vdash l_m : \Pi_m \)
* (d4) scoped(\( \sigma_m, \sigma_n, l_m, l_0 \cup \text{dom}(\rho') \))
* (d5) stacked\((\Sigma_m, \Sigma_n)\)
* (d6) preserving\((\sigma_m, \sigma_n, l_0 \cup \text{dom}(\rho'))\))
* (d7) \( \Sigma_n \vdash l_0 : T^\text{hot}_m \implies \Sigma_m \vdash l_m : T^\text{hot}_m \)

Now we have
* (e1) \( \Sigma'_m \vdash l_m : D^\text{hot} \)
  by (a1) and (d7)

Choose \( l_m, \sigma_m \) and \( \Sigma_m \): (b) holds from (d1), (D1) and (B1).
(c) and (d) holds from (e1).
(e) holds from (d4), (I1) and Lemma 3.20.
(f) holds from (d5), (D5) and (B5).
(g) holds from (d6), (I1), (I2) and Lemma 3.21.
(h) follows from (e1) trivially.

- case \( \Pi_0 \neq \emptyset \)

In this case, from Lemma 3.15 we know there exists \( \pi \in \Pi_0 \) such that
* (a0) \( \Sigma_0 ; \psi \vdash l_0 : \{ \pi \} \)

There are four cases for \( \pi \):
* case \( \pi = C\text{.this} \)

By definition of potential typing for (e0), we have
  · (a1) \( l_0 = \psi \)
  · (a2) \( T_0 = C \)

From (A4), we have
  · (b1) \( C\text{.this}.m \in \Pi' \)
  · (b2) \( C\text{.this}.m! \in \Phi' \)

From \( \Sigma : \psi \vdash \Phi, \Phi' \subseteq \Phi, \Phi_m \subseteq \text{fix}(\mathcal{E}, C\text{.this}.m!) \), (b2) and monotonicity of potential typing, we know
  · \( \Sigma_n ; \psi \vdash \Phi_m \)

We prepare the environment \( \rho' \) for the method call as follows:
  · (c1) \( \rho' = x_i : l_i \)

We have:
  · (d1) \( \Sigma_n ; \psi \vdash \psi : C\text{.this} \)
  · (d2) \( \forall l \in \text{dom}(\rho'), \Sigma_n ; \psi \vdash l : \emptyset \)

Now we can use the induction hypothesis for (G1):
  · (e1) \( \Sigma_n \preceq \Sigma_m \) and \( \Sigma_m \vdash \sigma_m \)
  · (e2) \( \Sigma_m \vdash l_m : T^\mu_m \)
  · (e3) \( \Sigma_m ; \psi \vdash l_m : \Pi_m \)
  · (e4) scoped\((\sigma_m, \sigma_n, l_m, l_0 \cup \text{dom}(\rho'))\))
  · (e5) stacked\((\Sigma_m, \Sigma_n)\)
  · (e6) preserving\((\sigma_m, \sigma_n, \psi \cup \text{dom}(\rho'))\))
  · (e7) \( \Sigma_n \vdash \psi : C^\text{hot}_m \implies \Sigma_m \vdash l_m : T^\text{hot}_m \)

From (e3) and Lemma 3.14, we have
  · (f1) \( \Sigma_m ; \psi \vdash l_m : \{ C\text{.this}.m \} \)
  · (f2) \( \Sigma_m ; \psi \vdash l_m : \Pi' \)
  · from (b1) and (f1)
(b) holds from (e1), (D1) and (B1).
(c) holds from (e2).
(d) holds from (f2).
(e) holds from (e4), (I1) and Lemma 3.20.
(f) follows from (e5), (D5) and (B5).
(g) holds from (e6), (I1), (I2) and Lemma 3.21.
(h) follows from (B7), (D7) and (e7).

* case $\pi = \text{warm}[T_0]$

From (A4), we have
- (b1) $\text{warm}[T_0].m \in \Pi'$
- (b2) $\text{warm}[T_0].m \in \Phi'$

Let’s consider whether $\beta \uparrow$ or $\text{cold} \uparrow \in \text{fix}(E, \Phi_m)$. If that is the case, from $\Sigma, \psi \models \Phi, (b2)$ and (A5), it must be $\Sigma \models \psi : C^{\text{hot}}$. Use (B7), we know $l_0$ is hot. Now we can follow the safe proof for the case $\Pi_0 = \emptyset$.

So we only need to consider the case where $\Phi_m$ does not leak neither $\beta$ nor $\text{cold}$. The only non-trivial effect is $T_0\text{.this.f}$. Given that $l_0$ is warm, all field access effects are fine, thus we have
- (b3) $\Sigma_n; l_0 \models \Phi_m$

We prepare the environment $\rho'$ for the method call as follows:
- (c1) $\rho' = x_i : l_i$

We have:
- (d1) $\Sigma_n; l_0 \models l_0 : T_0\text{.this}$
- (d2) $\forall l \in \text{dom}(\rho'), \Sigma_n; l_0 \models l : \emptyset$

Now we can use the induction hypothesis for (G1):
- (e1) $\Sigma_n \preceq \Sigma_m$ and $\Sigma_m \models \sigma_m$
- (e2) $\Sigma_m \models l_m : T_m^{\mu_m}$
- (e3) $\Sigma_m; l_0 \models l_m : \Pi_m$
- (e4) scoped($\sigma_m, \sigma_n, l_m, l_0 \cup \text{dom}(\rho')$)
- (e5) stacked($\Sigma_m, \Sigma_n$)
- (e6) preserving($\sigma_m, \sigma_n, l_0 \cup \text{dom}(\rho')$)
- (e7) $\Sigma_n \models l_0 : C^{\text{hot}} \implies \Sigma_m \models l_m : T_m^{\text{hot}}$

From (e3), we have
- (f1) $\Sigma_m; \psi \models l_m : \{T_0\text{.this.m}\}$ ▷ Lemma 3.14
- (f2) $\Sigma_m; \psi \models l_m : \text{warm}[T_0].m$ ▷ Lemma 3.18
- (f3) $\Sigma_m; \psi \models l_m : \Pi'$ ▷ from (b1), (f2) and Lemma 3.14

(b) holds from (e1), (D1) and (B1).
(c) holds from (e2).
(d) holds from (f3).
(e) holds from (e4), (I1) and Lemma 3.20.
(f) follows from (e5), (D5) and (B5).
(g) holds from (e6), (I1), (I2) and Lemma 3.21.
(h) follows from (B7), (D7) and (e7).

* case $\pi = \beta.f$

From (A4), we have
- (a1) $\beta.f \uparrow \in \Phi'$
- (a2) $\Phi' = \emptyset$

From premise (7), (A5), (a1) and monotonicity of effect typing (Lemma 3.6), we have:
- (b1) $\Sigma_0; \psi \not\models \beta.f \uparrow$
With \((a0), (b1), (B7)\), we use the Lemma 3.13:
\[ (c1) \Sigma_0 \vdash l_0 : T_0^{hot} \]
Now we can follow the case \(\Pi_0 = \emptyset\) to complete the proof.

\* **case** \(\pi = \beta.m\)
Similar as above.

- **case** \(T-New\). \(e = new D(\overline{v})\)
From the typing rule T-New, we have the following:
\(- (A1) \overline{f_i}: T_i = constrType(\Xi, D)\)
\(- (A2) \Xi; D \vdash e_i : T_i \land (\Phi_i, \Pi_i)\)
\(- (A3) (\Phi', \Pi') = init(D, \overline{f_i} = \Pi_i)\)
\(- (A4) \Phi = \cup \Phi_i \cup \Phi'\)
\(- (A5) \Pi = \Pi'\)

Let \(\Sigma_0 = \Sigma, \sigma_0 = \sigma\), use induction hypothesis on arguments consecutively:
\(- (B1) \Sigma_{l-1} \subseteq \Sigma_l\) and \(\Sigma_l \vdash \sigma_i\)
\(- (B2) \Sigma_l \vdash l_i : T_i^{\mu_i}\)
\(- (B3) \Sigma_l; \psi \vdash l_i : \Pi_i\)
\(- (B4) \text{scoped}(\sigma_i, \sigma_{l-1}, l_i, \psi \cup \text{codom}(\rho))\)
\(- (B5) \text{stacked}(\Sigma_i, \Sigma_{l-1})\)
\(- (B6) \text{preserving}(\sigma_i, \sigma_{l-1}, \psi \cup \text{codom}(\rho))\)
\(- (B7) \Sigma_{l-1} \vdash \psi : C^{hot} \implies \Sigma_l \vdash l_i : T_i^{hot}\)

From (B4), (B6) and Lemma 3.22, we have
\(- (C1) \text{scoped}(\sigma_n, \sigma, \{l_i, \cdots, l_n\}, \psi \cup \text{codom}(\rho))\)
\(- (C2) \text{preserving}(\sigma_n, \sigma, \psi \cup \text{codom}(\rho))\)  \(\triangleright\) By Lemma 3.24

We define \(\sigma'_0\) and \(\Sigma'_0\) with a fresh location \(l\):
\(- (D1) \sigma'_0 = \sigma_n \cup \{l \mapsto (D, \overline{f_i} = l_i)\}\)
\(- (D2) \Sigma'_0 = \Sigma_n \cup \{l \mapsto D(\overline{f_i})\}\)  \(\triangleright\) class parameters

We have the following:
\(- (E1) \text{scoped}(\sigma'_0, \sigma, l, \psi \cup \text{codom}(\rho))\)  \(\triangleright\) from (C1) and (D1)
\(- (E2) \text{preserving}(\sigma'_0, \sigma_n, \psi \cup \text{codom}(\rho))\)  \(\triangleright\) from (C1), (D1) and definition
\(- (E3) \text{preserving}(\sigma'_0, \sigma, \psi \cup \text{codom}(\rho))\)  \(\triangleright\) by (C2), (E2) and Lemma 3.24

From the definition of \(init\), there are two cases:

- **case** \(\Pi = \emptyset\)
In this case we have \(\Pi_I \subseteq \Phi\), thus
\* (a1) \(\Sigma; \psi \not\vdash \Pi_I\)
\* (a2) \(\Sigma_n \vdash l_i : T_i^{hot}\)  \(\triangleright\) from (a1), (B4) and Lemma 3.13

As all current fields of \(l\) is hot, we have
\* (b1) \(\Sigma \subseteq \Sigma'_0\) and \(\Sigma'_0 \not\vdash \sigma'_0\)  \(\triangleright\) by definition and (B1)
\* (b2) \(\Sigma'_0 \vdash l : D(\overline{f_i})\)  \(\triangleright\) by (B3) and Lemma 3.9 and 3.4

The following invariant is true at this step:
\* (c1) \(\forall \alpha \in \text{dom}(\sigma'_0).\text{reachable}(\sigma'_0, l, \alpha') \implies \Sigma'_0 \not\vdash \alpha' : T^{warm} \lor \alpha' = l\)
\* (c2) \(\forall \alpha \in \text{dom}(\sigma'_0).\Sigma' \vdash \alpha' : T^{warm} \lor \alpha' \in \text{dom}(\sigma) \lor \alpha' = l\)  \(\triangleright\) from (B5) and (D1)

We will show that there exists \(\sigma', \Sigma'\) after all fields are initialized:
\* (d1) \(\forall \alpha \in \text{dom}(\sigma').\text{reachable}(\sigma', l, \alpha') \implies \Sigma' \not\vdash \alpha' : T^{warm} \lor \alpha' = l\)
\* (d2) \(\Sigma \subseteq \Sigma'\) and \(\Sigma' \not\vdash \sigma'\)
\* (d3) \(\Sigma' \vdash l : D(\overline{f_i} : \cdots . f_n, f_i : \cdots . f_m)\)
We perform induction on \( m \) — the number of fields in the class body.

The basic case \( m = 0 \) is trivially true as shown in (b1), (b2), (c1), (c2) and (E1) - (E3).

Let us consider \( m = i + 1 \). As induction hypothesis, we have

* (e1) \( \forall l' \in \text{dom}(\sigma'_i). \text{reachable}(\sigma'_i, l, l') \implies \Sigma'_i \vdash l' : T^{\text{warm}} \lor l' = l \).
* (e2) \( \Sigma \subseteq \Sigma'_i \) and \( \Sigma'_i \vdash \sigma'_i \)
* (e3) \( \Sigma'_i \vdash l : D(f_1, \ldots, f_i) \)
* (e4) \( \text{scoped}(\sigma'_i, \sigma, l, \psi \cup \text{codom}(\rho)) \)
* (e5) \( \text{preserving}(\sigma'_i, \sigma, \psi \cup \text{codom}(\rho)) \)
* (e6) \( \forall l' \in \text{dom}(\sigma'_i). \Sigma'_i \vdash l' : T^{\text{warm}} \lor l' \in \text{dom}(\sigma) \lor l' = l \)

Consider the \( i+1 \)-th field in the class body \( \text{var } f_{i+1} : T_{i+1} = e_{i+1} \). From field typing, we have

* (f1) \( \Xi; 0; C \vdash e_{i+1} : T_{i+1} ! (\Phi_{i+1}, \Pi_{i+1}) \)

From the typing rule T-Check, we have

* (g1) \( \Xi; \{ f_1, \ldots, f_i \}; C \vdash \Phi_{i+1} \)

Now with (e3) we use Lemma 3.12, we get

* (h1) \( \Sigma'_i \vdash l \in \Phi_{i+1} \)

Now use induction hypothesis on (f1), we know there exists \( l_{i+1}, \sigma'_{i+1}, \Sigma'_{i+1} \) such that:

* (i1) \( \Sigma'_i \subseteq \Sigma'_{i+1} \) and \( \Sigma'_{i+1} \vdash \sigma'_{i+1} \)
* (i2) \( \Sigma'_{i+1} \vdash l_{i+1} : T_{i+1} \)
* (i3) \( \Sigma'_{i+1} \vdash l \in l_{i+1} : \Pi_{i+1} \)
* (i4) \( \text{scoped}(\sigma'_{i+1}, \sigma'_i, l_{i+1}, l) \)
* (i5) \( \text{stacked}(\Sigma'_{i+1}, \Sigma'_i) \)
* (i6) \( \text{preserving}(\sigma'_{i+1}, \sigma'_i, l) \)
* (i7) \( \Sigma'_{i+1} \vdash l : \text{hot} \implies \Sigma'_{i+1} \vdash l_{i+1} : T^{\text{hot}}_{i+1} \)

We define \( \sigma_D \) and \( \Sigma_D \) as follows:

* (k1) \( \sigma_D = \sigma'_{i+1} \cup \{ l \mapsto [f_{i+1} \mapsto l_{i+1} \mid \sigma'_{i+1}(l)] \} \)
* (k2) \( \Sigma_D = \Sigma'_{i+1} \cup \{ l \mapsto D(f_1, \ldots, f_i) \} \)
* (k3) \( \Sigma'_{i+1} \subseteq \Sigma_D \) and \( \Sigma_D \vdash \sigma_D \)

\( \triangleright \) trivial by definition

(d1) holds from (e1), (i4) and (i5).
(d2) holds from (k3), (i1), (e2) and transitivity of store typing ordering.
(d3) holds from (k2), (e2) and (i3).
(d4) holds from (e4), (i4) and definition of scoped
(d5) holds from (e4), (e5), (i6), Lemma 3.21, (i4), (k1) and Lemma 3.25.
(d6) holds from (e6), (i5) and (k1).

Now, as all fields of the object \( l \) are assigned, we may define

* (l1) \( \Sigma'_D = \{ l \mapsto D^{\text{warm}} \} \Sigma_D \)
* (l2) \( \Sigma'_D \subseteq \Sigma'_D \) and \( \Sigma'_D \vdash \sigma'_D \)

\( \triangleright \) trivial by definition

Now from (d1) we know all objects reachable from \( l \) are warm, thus we can use the Lemma 3.19 to type all warm objects reachable from \( l \) to hot.

(b) follows from (d2) and (l2).
(c) and (d) holds trivially due to \( l \) is hot.
(e) follows from (d4)
(f) holds from (d6) and the fact that \( l \) is warm.
(g) holds from (d5).
(h) follows from the fact that \( l \) is hot.
Technical Report: A Type-and-Effect System for Object Initialization

We will show that there exists \( \Pi \rightarrow \sigma \) from (a1), (B4) and Lemma 3.13.

We know the arguments \( \overline{T_i} \) are hot, while \( \overline{T_j} \) might not. The latter corresponds to cold class parameters, any values are allowed. Thus, we have

* (b1) \( \Sigma_0' = \Sigma_0 \cup \{ l \mapsto D(f_i,f_j) \} \) \( \triangleright \) class parameters

* (b2) \( \Sigma_n \preceq \Sigma_0' \) and \( \Sigma_0' \models \sigma_0' \) \( \triangleright \) by definition

In (b2), the fields \( f_i \) of the value \( l \) type check because the potential typing for \( C.this.f \) allows any value, other fields type check because they are hot.

Now use induction hypothesis on (f1), we know there exists \( \sigma' \), \( \Sigma' \) after all fields are initialized:

* (d1) \( \Sigma_0' \preceq \Sigma' \) and \( \Sigma' \models \sigma' \)

* (d2) \( \Sigma' \models l : D(f_i,f_j,f_k) \)

* (d3) \( \text{scopel}(\sigma', \sigma, l, \psi \cup \text{cdom}(\rho)) \)

* (d4) \( \text{preservel}(\sigma', \sigma, \psi \cup \text{cdom}(\rho)) \)

* (d5) \( \forall l' \in \text{dom}(\sigma'). \Sigma' \models l' : T_{\text{warm}} \lor l' \in \text{dom}(\sigma) \lor l' = l \)

We perform induction on \( m \) — the number of fields in the class body.

The basic case \( m = 0 \) is trivially true by choosing \( \Sigma' = \Sigma_0' \) and \( \sigma' = \sigma_0' \). The result follows from (b1), (b2), (c1) and (E1) - (E3).

Let us consider \( m = i + 1 \). As induction hypothesis, we have

* (e1) \( \Sigma_i' \preceq \Sigma_i' \) and \( \Sigma_i' \models \sigma_i' \)

* (e2) \( \Sigma_i' \models l : D(f_i,\ldots,f_n,f_i) \)

* (e3) \( \text{scopel}(\sigma_i', \sigma, l, \psi \cup \text{cdom}(\rho)) \)

* (e4) \( \text{preservel}(\sigma_i', \sigma, \psi \cup \text{cdom}(\rho)) \)

* (e5) \( \forall l' \in \text{dom}(\sigma_i'). \Sigma_i' \models l' : T_{\text{warm}} \lor l' \in \text{dom}(\sigma) \lor l' = l \)

Consider the \( i + 1 \)-th field in the class body \( \text{var} f_{i+1} : T_{i+1} = e_{i+1} \). From field typing, we have

* (f1) \( \Sigma_i', l : e_{i+1} \models f_{i+1} \)

We define \( \sigma_D \) and \( \Sigma_D \) as follows:

* (k1) \( \sigma_D = \sigma_{i+1}' \cup \{ l \mapsto [f_{i+1} \mapsto l_{i+1}]\sigma_{i+1}'(l) \} \)

* (k2) \( \Sigma_D = \Sigma_{i+1}' \cup \{ l \mapsto D(f_i,\ldots,f_n,f_i,f_{i+1}) \} \)

* (k3) \( \Sigma_{i+1}' \preceq \Sigma_D \) and \( \Sigma_D \models \sigma_D \) \( \triangleright \) trivial by definition

(d1) holds from (k3), (i1), (e1) and transitivity of store typing ordering.
We define 
\[ \sigma \]

We have the following holds 

From (C3), (D1) and Lemma 3.13, we have 

\[ \Sigma \]

From the premise 

\[ \Sigma \]

From the induction hypothesis on 

\[ e \]

\[ \sigma \]

From the typing rule 

\[ T-Block \]

\[ case \ T-Block. \ e = (e_0, f = e_1; e_2) \]

From the typing rule T-Block, we have the following:

- (A1) \( \Xi; \Gamma; C \vdash e_0 : T_0 ! (\Phi_0, \Pi_0) \)
- (A2) \( T_1 = fieldType(\Xi, T_0, f) \)
- (A3) \( \Xi; \Gamma; C \vdash e_1 : T_1 ! (\Phi_1, \Pi_1) \)
- (A4) \( \Xi; \Gamma; C \vdash e_2 : D ! (\Phi_2, \Pi_2) \)
- (A5) \( \Phi = \Phi_0 \cup \Phi_1 \cup \Pi_1 \uparrow \Phi_2 \)
- (A6) \( \Pi = \Pi_2 \)

From the induction hypothesis on \( e_0 \), we know there exists \( l_0, \sigma_0 \) and \( \Sigma_0 \) such that:

- (B1) \( \Sigma \leq \Sigma_0 \) and \( \Sigma_0 \vdash \sigma_0 \)
- (B2) \( \Sigma_0 \vdash l_0 : T_0^{\mu_0} \)
- (B3) \( \Sigma_0; \psi \vdash l_0 : \Pi_0 \)
- (B4) \( \text{scoped}(\sigma_0, \sigma, l_0, \psi \cup \text{dom}(\rho)) \)
- (B5) \( \text{stacked}(\Sigma_0, \Sigma) \)
- (B6) \( \text{preserving}(\sigma_0, \sigma, \psi \cup \text{dom}(\rho)) \)
- (B7) \( (T_0, \omega_0) = \sigma_0(l_0) \)
- (B7) \( \Sigma \vdash \psi : C^{\text{hot}} \implies \Sigma_0 \vdash l_0 : T_0^{\text{hot}} \)

From the induction hypothesis on \( e_1 \), we know there exists \( l_1, \sigma_1 \) and \( \Sigma_1 \) such that:

- (C1) \( \Sigma_0 \leq \Sigma_1 \) and \( \Sigma_1 \vdash \sigma_1 \)
- (C2) \( \Sigma_1 \vdash l_0 : T_1^{\mu_1} \)
- (C3) \( \Sigma_1; \psi \vdash l_1 : \Pi_1 \)
- (C4) \( \text{scoped}(\sigma_1, \sigma_0, l_1, \psi \cup \text{dom}(\rho)) \)
- (C5) \( \text{stacked}(\Sigma_1, \Sigma_0) \)
- (C6) \( \text{preserving}(\sigma_1, \sigma_0, \psi \cup \text{dom}(\rho)) \)
- (C7) \( \Sigma_0 \vdash \psi : C^{\text{hot}} \implies \Sigma_1 \vdash l_1 : T_1^{\text{hot}} \)

From the premise \( \Sigma; \psi \vdash \Phi \), (A5) and monotonicity of effect typing, we have

- (D1) \( \Sigma_1; \psi \vdash \Pi_1 \uparrow \)

From (C3), (D1) and Lemma 3.13, we have

- (E1) \( \Sigma_1 \vdash l_1 : T_1^{\text{hot}} \)

We define \( \sigma_1' \) as follows

- (F1) \( \sigma_1' = \sigma_1 \cup \{ l \mapsto [f \mapsto l_1]_{\sigma_1}(l_0) \} \)
- (F2) \( \Sigma_1 \vdash \sigma_1' \)

\[ \text{by (E1) and definition} \]

We have the following holds
– (F3) preserving($\sigma'_1, \sigma_1, \psi \cup \text{codom}(\rho)$) \hspace{1cm} \triangleright from (F1), (C4), (C6) and Lemma 3.25
– (F4) preserving($\sigma'_1, \sigma, \psi \cup \text{codom}(\rho)$) \hspace{1cm} \triangleright by (F3), (C6), (B6) Lemma 3.24
– (F5) scoped($\sigma'_1, \sigma, \psi \cup \text{codom}(\rho), \psi \cup \text{codom}(\rho)$) \hspace{1cm} \triangleright by (F4) and Lemma 3.23

Now use induction hypothesis on $e_2$ in (A4), we have

– (G1) $\Sigma_1 \preccurlyeq \Sigma_2$ and $\Sigma_2 \not\vDash \sigma_2$
– (G2) $\Sigma_2 \vDash l_2 : T^\mu_2$
– (G3) $\Sigma_2; \psi \vDash l_2 : \Pi_2$
– (G4) scoped($\sigma_2, \sigma'_1, l_2, \psi \cup \text{codom}(\rho)$)
– (G5) stacked($\Sigma_2, \Sigma_1$)
– (G6) preserving($\sigma_2, \sigma_1, \psi \cup \text{codom}(\rho)$)
– (G7) $\Sigma_1 \vDash \psi : C_{\text{hot}} = \Rightarrow \Sigma_2 \vDash l_1 : T^\mu_{\text{hot}}$

We have the following

– (H1) preserving($\sigma_2, \sigma, \psi \cup \text{codom}(\rho)$) \hspace{1cm} \triangleright by (G6), (F3), (C6), (B6) and Lemma 3.21
– (H2) scoped($\sigma_2, \sigma, l_2, \psi \cup \text{codom}(\rho)$) \hspace{1cm} \triangleright by (G4), (F5) and Lemma 3.20

Choose $l = l_2$, $\sigma' = \sigma$, $\Sigma' = \Sigma_2$;
(b) follows from (G1), (F2), (C1), (B1).
(c) follows from (G2).
(d) follows from (G3).
(e) follows from (H2).
(f) follows from (G5), (C5) and (B5).
(g) follows (H1).
(h) follows from (G7) and monotonicity of value typing.

□

REFERENCES