Gradual Function Application in Set-Theoretic Types:
Design Rationale and Soundness

This note establishes the theoretical basis for the gradual function application algorithm implemented in `descr.ex`. We prove that the implementation is sound: in particular, that applying a function to an argument that partially escapes its domain returns the correct conservative type, and that the extended gradual domain used in the domain check is well-defined and correctly computed.

1 Notation and Setup

We work with *gradual types* that combine static and dynamic components. A type τ decomposes into two *materializations*:

- $\tau^{\Uparrow}$: the *upper (static) materialization*—the precise, non-dynamic part.
- $\tau^{\Downarrow}$: the *lower materialization*—what holds even under dynamic uncertainty.

For a purely dynamic type $? \wedge t$ (where t is static): $(? \wedge t)^{\Uparrow} = t$ and $(? \wedge t)^{\Downarrow} = \texttt{none}$.

We write $\mathbf{F} = \texttt{none} \to \texttt{term}$ for the top function type, $\mathrm{dom}(t)$ for the static domain of a function type $t \leq \mathbf{F}$, and $\circ$ for the standard set-theoretic function application operator.

Key invariant. Static inputs yield static outputs; dynamic uncertainty propagates but is never silently promoted to a static guarantee.

2 Extended Gradual Domain

To decide whether a function application can proceed, we need the *domain* of a gradual function type: the set of arguments it may accept. The static domain $\mathrm{dom}(\tau)$ is only defined when both $\tau^{\Uparrow}, \tau^{\Downarrow} \leq \mathbf{F}$, which excludes the practically important case $\tau = ?$ (where $\tau^{\Uparrow} = \texttt{term}$).

Definition 1 (Extended gradual domain)**.** For a gradual type τ with $\tau^{\Downarrow} \leq \mathbf{F}$:

$$\mathrm{dom}^{?}(\tau) \;=\; \mathrm{dom}\big(\tau^{\Uparrow} \wedge \mathbf{F}\big) \;\vee\; ? \wedge \mathrm{dom}\big(\tau^{\Downarrow}\big)$$

The $\wedge\,\mathbf{F}$ projection discards the non-function part of $\tau^{\Uparrow}$ before computing the domain. When $\tau^{\Uparrow} \leq \mathbf{F}$, this reduces to the formula of Lanvin (2021).

Example 1.

$$\begin{aligned}
\mathrm{dom}^{?}(?) &= \mathrm{dom}(\texttt{term} \wedge \mathbf{F}) \vee ? \wedge \mathrm{dom}(\texttt{none}) = \mathrm{dom}(\mathbf{F}) \vee ? \wedge \texttt{term} = \texttt{none} \vee ? = ? \\
\mathrm{dom}^{?}\big(? \wedge (\texttt{int} \to \texttt{int})\big) &= \mathrm{dom}(\texttt{int} \to \texttt{int}) \vee ? \wedge \mathrm{dom}(\texttt{none}) = \texttt{int} \vee ? \\
\mathrm{dom}^{?}\big((\texttt{bool} \to \texttt{bool}) \vee ? \wedge (\texttt{int} \to \texttt{int})\big) &= \mathrm{dom}\big((\texttt{bool} \to \texttt{bool}) \vee (\texttt{int} \to \texttt{int})\big) \vee ? \wedge \mathrm{dom}(\texttt{bool} \to \texttt{bool}) \\
&= \texttt{none} \vee ? \wedge \texttt{bool} = ? \wedge \texttt{bool}
\end{aligned}$$

In the implementation, `fun_normalize/3` computes the domain of $\tau^{\Uparrow} \wedge \mathbf{F}$ implicitly by inspecting only the `:fun` component of the internal type representation (the projection by $\wedge\,\mathbf{F}$ is performed structurally). The function `fun_normalize_both/3` assembles $\mathrm{dom}^{?}(\tau)$ as:

```
domain = union(dynamic_domain, dynamic(static_domain))
```

corresponding to $\mathrm{dom}(\tau^{\Uparrow} \wedge \mathbf{F}) \vee ? \wedge \mathrm{dom}(\tau^{\Downarrow})$.

3 The Application Operator

Definition 2 (Gradual function application)**.** Given $f : \tau$ and $e : \tau'$, application $f\ e$ is typed as follows.

Preconditions. Reject if $\tau' = \texttt{none}$. If $\tau' \not\leq \mathrm{dom}^{?}(\tau)$, reject unless τ' is *compatible* with $\mathrm{dom}^{?}(\tau)$, in which case return ?.

Result (when preconditions pass and $\tau' \leq \mathrm{dom}^{?}(\tau)$):

$$\tau \circ \tau' \;=\; \begin{cases} \tau \circ \tau' & \text{fully static} \\ ? \wedge \left(\tau^{\Uparrow} \circ \tau'^{\Uparrow}\right) & \tau^{\Downarrow} = \texttt{none} \\ \left(\tau^{\Downarrow} \circ \tau'^{\Uparrow}\right) \;\vee\; ? \wedge \left(\tau^{\Uparrow} \circ \tau'^{\Uparrow}\right) & \text{mixed} \end{cases}$$

Remark 1. The purely dynamic case is a special instance of the mixed formula: $\texttt{none} \circ s = \texttt{none}$ (applying the empty function type yields nothing), so $(\texttt{none} \circ \tau'^{\Uparrow}) \vee ? \wedge (\tau^{\Uparrow} \circ \tau'^{\Uparrow}) = ? \wedge (\tau^{\Uparrow} \circ \tau'^{\Uparrow})$.

Remark 2. Both gradual branches use $\tau'^{\Uparrow}$ (the upper bound of the argument), refining Lanvin's Definition 6.15 which uses $\tau'^{\Downarrow}$ in the dynamic branch. This is sound: both branches are ?-tagged, so widening the input cannot produce unsound static guarantees. It is also more precise: with overloaded (intersected) arrows, a broader argument yields a tighter intersection of return types.

The implementation in `fun_apply_with_strategy/3` realizes exactly these three cases:

```
static? ->
  {:ok, fun_apply_static(arguments, static_arrows)}

static_arrows == [] ->  # purely dynamic: tau_down = none
  arguments = Enum.map(arguments, &upper_bound/1)
  {:ok, dynamic(fun_apply_static(arguments, dynamic_arrows))}

true ->                 # mixed
  arguments = Enum.map(arguments, &upper_bound/1)
  {:ok, union(
    fun_apply_static(arguments, static_arrows),
    dynamic(fun_apply_static(arguments, dynamic_arrows))
  )}
```

4 Soundness: Domain Escape Yields `term`

The domain check $\tau' \leq \mathrm{dom}^{?}(\tau)$ ensures $\tau'^{\Uparrow} \leq \mathrm{dom}(\tau^{\Downarrow})$ (the static branch is well-defined), but does *not* guarantee $\tau'^{\Uparrow} \leq \mathrm{dom}(\tau^{\Uparrow})$. We must verify that the dynamic branch remains sound when $\tau'^{\Uparrow}$ escapes $\mathrm{dom}(\tau^{\Uparrow})$.

The implementation computes not the standard $\circ$ but a *generalized* operator $\circ^*$ that is total (defined for any input, whether or not it is in the domain):

Definition 3 (Generalized application)**.** Let $t \simeq \bigvee_{i \in I^+} \left(\bigwedge_{(s_j,t_j) \in P_i} (s_j \to t_j) \wedge \bigwedge_{N_i} \neg(\cdots)\right)$ be the function DNF of t.

$$t \circ^* s \;\coloneqq\; \bigvee_{i \in I^+} \bigvee_{\substack{Q \subseteq P_i \\ s \not\leq \bigvee_{j \in Q} s_j}} \bigwedge_{j \in P_i \setminus Q} t_j$$

where $\bigwedge_{\emptyset} t_j \coloneqq \texttt{term}$ (empty intersection is top). When $s \leq \mathrm{dom}(t)$, we have $t \circ^* s = t \circ s$.

Proposition 1 (Domain escape yields `term`)**.** *Let $t \leq \mathbf{F}$ be a function type and s any type with $s \neq \texttt{none}$ and $s \not\leq \mathrm{dom}(t)$. Then $t \circ^* s = \texttt{term}$.*

Proof. Write $\mathrm{dom}(t) = \bigwedge_{i \in I^+} D_i$ where $D_i = \bigvee_{(s_j,t_j) \in P_i} s_j$. Since $s \not\leq \mathrm{dom}(t)$, there exists $i_0 \in I^+$ with $s \not\leq D_{i_0}$.

For clause i_0, take $Q = P_{i_0}$. Then $\bigvee_{j \in Q} s_j = D_{i_0}$, and $s \not\leq D_{i_0}$, so the filter condition $s \not\leq \bigvee_{j \in Q} s_j$ is satisfied. The corresponding return-type term is $\bigwedge_{j \in P_{i_0} \setminus P_{i_0}} t_j = \bigwedge_{\emptyset} t_j = \texttt{term}$.

The overall result is a join containing `term`; since `term` is the maximum type, $t \circ^* s = \texttt{term}$. □

Correspondence with `aux_apply/4`. The implementation computes $\circ^*$ directly. `fun_apply_static/2` reduces over DNF clauses, calling `aux_apply/4` for each with `rets_reached` initialized to `term()`:

```
Enum.reduce(arrows, none(), fn intersection, acc ->
  aux_apply(acc, type_args, term(), intersection)
end)
```

Within a clause, `aux_apply/4` processes arrows one at a time. For each arrow $(s_k \to t_k)$ it splits:

```
dom_subtract = difference(input, args_to_domain(args))   # input \ s_k
ret_refine   = intersection(returns_reached, ret)         # rets meet t_k

# Phase 1: input escapes s_k -- add arrow to Q, rets unchanged
result = if empty?(dom_subtract), do: result,
         else: aux_apply(result, dom_subtract, returns_reached, rest)

# Phase 2: input covered by s_k -- keep arrow in P_i \ Q, refine rets
aux_apply(result, input, ret_refine, rest)
```

Phase 1 corresponds to adding the current arrow to Q (the "missing" set); Phase 2 keeps it in $P_i \setminus Q$. At the base case, `rets_reached` is unioned into the result:

```
if subtype?(rets_reached, result), do: result, else: union(result, rets_reached)
```

When $s \not\leq D_{i_0}$, the pure-escape path is always available: at each arrow (s_k, t_k) in clause i_0, the difference $s \setminus (s_1 \vee \cdots \vee s_k)$ is non-empty (since $s \not\leq D_{i_0}$ and $s_1 \vee \cdots \vee s_k \leq D_{i_0}$), so Phase 1 is always taken and `rets_reached` is never refined. At the leaf, `term()` is unioned into the result; since `term()` is maximal, the clause contributes `term` to the final join.

Theorem 1 (Soundness of the dynamic branch)**.** *When $\tau^{\Downarrow} =$ `none` and $\tau'^{\Uparrow} \not\leq \mathrm{dom}(\tau^{\Uparrow})$, the implementation computes:*

$$? \wedge (\tau^{\Uparrow} \circ^* \tau'^{\Uparrow}) = ? \wedge \mathtt{term} = ?.$$

All arrow type information is lost, which is the correct conservative result for an argument that may not satisfy the function's domain.

Example. Apply $\tau = ? \wedge ((\mathtt{int} \to \mathtt{int}) \wedge (\mathtt{bool} \to \mathtt{bool}))$ to $\tau' = ? \wedge (\mathtt{int} \vee \mathtt{float})$.

Domain check: $\mathrm{dom}^?(\tau) = (\mathtt{int} \vee \mathtt{bool}) \vee ?$, and $? \wedge (\mathtt{int} \vee \mathtt{float}) \leq \mathrm{dom}^?(\tau)$. ✓

Application: $\tau^{\Downarrow} = \mathtt{none}$, so we compute $? \wedge ((\mathtt{int} \to \mathtt{int}) \wedge (\mathtt{bool} \to \mathtt{bool}) \circ^* (\mathtt{int} \vee \mathtt{float}))$. Since $\mathtt{int} \vee \mathtt{float} \not\leq \mathtt{int} \vee \mathtt{bool} = \mathrm{dom}(\tau^{\Uparrow})$, Proposition 1 gives $\tau^{\Uparrow} \circ^* (\mathtt{int} \vee \mathtt{float}) = \mathtt{term}$, and the result is ?.

5 Soundness of the Domain Check

The domain check $\tau' \leq \mathrm{dom}^?(\tau)$ is *necessary and sufficient* to guarantee that the static branch $\tau^{\Downarrow} \circ \tau'^{\Uparrow}$ is well-defined:

Proposition 2. *$\tau' \leq \mathrm{dom}^?(\tau)$ implies $\tau'^{\Uparrow} \leq \mathrm{dom}(\tau^{\Downarrow})$.*

Proof. $\mathrm{dom}^?(\tau) = \mathrm{dom}(\tau^{\Uparrow} \wedge \mathbf{F}) \vee ? \wedge \mathrm{dom}(\tau^{\Downarrow})$. The static part of $\tau' \leq \mathrm{dom}^?(\tau)$ forces $\tau'^{\Uparrow} \leq \mathrm{dom}(\tau^{\Uparrow} \wedge \mathbf{F}) \vee \mathrm{dom}(\tau^{\Downarrow})$. Since $\mathrm{dom}(\tau^{\Uparrow} \wedge \mathbf{F}) \leq \mathrm{dom}(\tau^{\Downarrow})$ (the lower materialization is more restrictive), we get $\tau'^{\Uparrow} \leq \mathrm{dom}(\tau^{\Downarrow})$. □

Together, Propositions 1 and 2 and Theorem 1 establish that the application operator in `descr.ex`:

1. correctly rejects arguments outside the domain,
2. is well-defined on the static component whenever the domain check passes, and
3. conservatively degrades to ? when an argument escapes the dynamic component's domain, rather than producing a spurious type.