BraidCore Type System and Semantics

supplement to “Static Stages for Heterogeneous Programming,” in OOPSLA 2017

Adrian Sampson   Kathryn S McKinley   Todd Mytkowicz

We formalize a core language to precisely describe the semantics of static staging. Our formalization differs from most traditional semantics for multi-stage programming by incorporating the limitations of static staging, which rule out many of the subtle problems that can arise in dynamically staged languages.

Recall that static staging’s key goal is to generate efficient code for all stages in a program ahead of time. This design requires that code in all stages is statically *meaningful:* the compiler must statically associate each variable reference with its definition. In many traditional staging languages, the semantics allow open code where quotations can refer to a variable x when x is currently undeclared; the program can then dynamically splice this expression into a context where x is defined. This “unhygienic” property of dynamic staging can be useful for code generation, but it is inefficient for Braid’s purposes because the compiled code must dynamically resolve some references.

We formalize BraidCore to demonstrate the simpler type system and dynamic semantics that are possible under static staging. The original safe multi-stage programming language, MetaML, uses environments that mix together values from all stages and tags each value with an integer for its stage. Type rules let code at stage n refer to values tagged with a stage $n' \leq n$. Instead, BraidCore keeps the variables from each stage separate in a stack of per-stage environments—code can only refer to variables defined at the current “top” stage. This approach dramatically simplifies the type system: expressions interact with the top of the type-context stack in the same way that an ordinary language interacts with its monolithic, untagged type context. In our research, the semantics also clarified the meaning of cross-stage references as simple syntactic sugar for materialization.

Figure 1 lists the syntax for BraidCore programs. The formalism language augments a tiny imperative language with quotation, a run operator, splicing, and materialization. It omits the full language’s functions, control flow state-

$$
\begin{aligned}
e ::= {} & c \mid x \mid \textbf{var}\ x = e \mid e\ ;\ e \\
& \langle e \rangle \mid !e \mid {}_n[e] \mid \%[e] \mid {}^{\$}_{n}[e] \mid \$\langle e \rangle \\
& x \in \text{variables},\ c \in \text{constants},\ n \in 1, 2, \ldots
\end{aligned}
$$

Figure 1. Syntax for BraidCore.

ments, and arithmetic. Splicing escapes use a parameter $n \geq 1$ to indicate the number of levels to skip. Materialization escapes, on the other hand, may only skip a single level because multi-level materialization adds no expressive power over multiply-nested materialization brackets. It also excludes macros, which are syntactic sugar for splicing escapes, and cross-stage references, which can be semantically replaced with materialization escapes. The remaining forms focus on the way staging constructs control the order of evaluation and how variables are made available to code at the appropriate time.

1. Type System

Types in BraidCore are either primitive types or code types:

$$\begin{aligned} \tau &::= t \mid \langle \tau \rangle \\ t &::= \texttt{Int} \mid \texttt{Float} \mid \cdots \end{aligned}$$

We define a type context Γ as a stack of per-stage contexts, γ. Each per-stage context maps variable names to types:

$$\begin{aligned} \Gamma &::= \cdot \mid \gamma, \Gamma \\ \gamma &::= \cdot \mid x : \tau, \gamma \end{aligned}$$

We will use $\gamma(x)$ to denote type lookup in a particular stage. To denote a context stack separated into a prefix and a suffix, we write $\overline{\gamma}, \Gamma$ where $\overline{\gamma}$ is the prefix and Γ, the tail, is itself a context. We use a helper judgment $\text{len}(\overline{\gamma}) = n$ to determine the size of a prefix.

The typing judgment for BraidCore imperatively builds up a context Γ from each expression. Complete programs must type check in a context that consists a single, empty per-stage context. The rules for the non-staging syntax forms are standard:

$$\boxed{\Gamma_1 \vdash s : \tau ; \Gamma_2}$$

$$\textsc{type-const-int}\ \frac{c \text{ is an integer}}{\Gamma \vdash c : \texttt{Int}; \Gamma} \qquad \textsc{type-const-float}\ \frac{c \text{ is floating point}}{\Gamma \vdash c : \texttt{Float}; \Gamma} \qquad \textsc{type-seq}\ \frac{\Gamma_1 \vdash e_1 : \tau_1, \Gamma_2 \qquad \Gamma_2 \vdash e_2 : \tau_2, \Gamma_3}{\Gamma_1 \vdash e_1 \ ; \ e_2 : \tau_2; \Gamma_3}$$

The rules for variable lookup and assignment use the head of the type environment stack:

$$\textsc{type-lookup}\ \frac{}{\gamma, \Gamma \vdash x : \gamma(x); \gamma, \Gamma} \qquad \textsc{type-var}\ \frac{\Gamma_1 \vdash e : \tau; \gamma, \Gamma_2}{\Gamma_1 \vdash \textbf{var}\, x = e : \tau; (x : \tau, \gamma), \Gamma_2}$$

The typing rules for quotation and run expressions add or remove a level of quotation from the result's type:

$$\textsc{type-quote}\ \frac{\cdot, \Gamma_1 \vdash e : \tau; \gamma, \Gamma_2}{\Gamma_1 \vdash \langle e \rangle : \langle \tau \rangle; \Gamma_2} \qquad \textsc{type-run}\ \frac{\Gamma_1 \vdash e : \langle \tau \rangle; \Gamma_2}{\Gamma_1 \vdash !e : \tau; \Gamma_2}$$

The quotation rule uses a fresh, empty stage context at the head of the context stack. The rule then "pops" the context off again, discarding any updates to the context for the quoted stage.

A splicing escape type-checks its expression in the context n levels "up" and requires the result to be a code type. The resulting type is the unwrapped inner type, i.e., τ when the spliced code value has type $\langle \tau \rangle$:

$$\frac{\Gamma_1 \vdash e : \langle \tau \rangle ; \Gamma_2 \qquad \mathrm{len}(\overline{\gamma}) = n}{\overline{\gamma}, \Gamma_1 \vdash {}_{n}[e] : \tau ; \overline{\gamma}, \Gamma_2}\ \text{TYPE-SPLICE}$$

This rule temporarily extracts the top n stages' contexts as $\overline{\gamma}$, where n is the level-count number from the escape expression, ${}_{n}[e]$. The inner expression e is typed in the context made up of the remaining stages, Γ_1. Finally, the prefix is placed back onto the context stack.

The rule for materialization is similar, but the expression need not evaluate to a code type and the escape only moves by a single stage:

$$\frac{\Gamma_1 \vdash e : \tau ; \Gamma_2}{\gamma, \Gamma_1 \vdash \%[e] : \tau ; \gamma, \Gamma_2}\ \text{TYPE-MATERIALIZE}$$

2. Dynamic Semantics

We define a big-step operational semantics for BraidCore execution. First, we define heaps as the dynamic analog for type contexts. Specifically, a heap H is a stack of per-stage environments h, which each map variable names to values. Values are either constants c or code values $\langle\!\langle e, h \rangle\!\rangle$:

$$\begin{aligned} H &::= \cdot \mid h, H \\ h &::= \cdot \mid x \mapsto v, h \\ v &::= c \mid \langle\!\langle e, h \rangle\!\rangle \end{aligned}$$

A code value consists of an expression e and an environment map h. Our semantics construct code values so that they never contain escapes that go beyond the top level: intuitively, they contain ready-to-execute code with no computations left to perform at earlier stages. Splice escapes are removed by inlining expressions from other code values, and materialization escapes are replaced with variable references into the associated environment h.

The big-step judgment evaluates an expression to a value and updates the heap. As with the type system, we use $h(x)$ to denote variable lookup. The basic rules are standard:

$$\boxed{H ; e \Downarrow H' ; v}$$

$$\frac{}{H ; c \Downarrow H ; c}\ \text{CONST} \qquad \frac{H_1 ; e_1 \Downarrow H_2 ; v_1 \qquad H_2 ; e_2 \Downarrow H_3 ; v_2}{H_1 ; (e_1\ ;\ e_2) \Downarrow H_3 ; v_2}\ \text{SEQ}$$

The rules for assignment and variable lookup work with the top per-stage environment in the heap:

$$\textsc{lookup}\quad \frac{}{h, H; x \Downarrow h, H; h(x)} \qquad \textsc{assign}\quad \frac{H; e \Downarrow h, H'; v}{H; \mathbf{var}\ x = e \Downarrow (x \mapsto v, h), H'; v}$$

To execute a code value, we evaluate its expression in a heap that contains its associated materialization environment. The expression is not allowed to escape beyond the first level of execution, so only this single stage is needed. Any updates to this temporary heap are ignored, and the run expression returns the value of the code value's expression:

$$\textsc{run}\quad \frac{H_1; e \Downarrow H_2; \langle\!\langle e_q, h_q \rangle\!\rangle \qquad h_q, \cdot; e_q \Downarrow H_3, v}{H_1; !e \Downarrow H_2, v}$$

The absence of multi-level materialization escapes avoids the need to "carry through" the materialization environment h_q to later stages: the variables in it can only be referenced at the first stage.

To evaluate a quotation, the operational semantics "switches" to a helper judgment, $\Downarrow_i$, to preserve the quoted expression while evaluating its escapes:

$$\textsc{quote}\quad \frac{H_1; \cdot; e \Downarrow_1 H'; h_q; e_q}{H_1; \langle e \rangle \Downarrow H'; \langle\!\langle e_q, h_q \rangle\!\rangle}$$

The escape expressions ${}_n[e]$, $\%[e]$, and $\overset{\$}{n}[e]$ are not allowed in top-level program source, so the main judgment does not have rules for them.

In the quoted interpretation judgment $\Downarrow_i$, the stage number $i \geq 1$ indicates the current level of quotation. The judgment works as a translator from expressions to expressions: it "scans over" the program to find escapes that need to be evaluated eagerly and switches back to the main judgment $\Downarrow$. It threads through a heap H, which may be updated inside escaped expressions, and a materialization environment h, which holds the results of top-level materialization escapes. The rules leave most expressions intact:

$$\boxed{H; h; e \Downarrow_i H'; h'; e'}$$

$$\textsc{quoted-const}\quad \frac{}{H; h; c \Downarrow_i H; h; c} \qquad \textsc{quoted-lookup}\quad \frac{}{H; h; x \Downarrow_i H; h; x} \qquad \textsc{quoted-assign}\quad \frac{H; h; e \Downarrow_i H'; h'; e'}{H; h; \mathbf{var}\ x = e \Downarrow_i H'; h'; \mathbf{var}\ x = e'}$$

$$\textsc{quoted-seq}\quad \frac{H_1; h_1; e_1 \Downarrow_i H_2; h_2; e_1' \qquad H_2; h_2; e_2 \Downarrow_i H_3; h_3; e_2'}{H_1; h_1; (e_1\ ;\ e_2) \Downarrow_i H_3; h_3; (e_1'\ ;\ e_2')} \qquad \textsc{quoted-run}\quad \frac{H; h; e \Downarrow_i H'; h'; e'}{H; h; !e \Downarrow_i H'; h'; !e'}$$

Quotation and escape expressions shift the current quote level number, as long the escape does not go beyond the current level:

$$\textsc{quoted-quote}\quad \frac{H; h; e \Downarrow_{i+1} H'; h'; e'}{H; h; \langle e\rangle \Downarrow_i; H'; h'; \langle e'\rangle} \qquad \textsc{quoted-splice}\quad \frac{H; h; e \Downarrow_{i-n} H'; h'; e' \qquad n < i}{H; h; {}_n[e] \Downarrow_i H'; h'; {}_n[e']}$$

$$\textsc{quoted-materialize}\quad \frac{H; h; e \Downarrow_{i-1} H'; h'; e' \qquad i > 1}{H; h; \%[e] \Downarrow_i H'; h'; \%[e']}$$

The judgment switches back to ordinary $\Downarrow$ interpretation when an escape goes beyond the current quotation level. For splicing, we require the expression to produce a code value and use a $\text{merge}(h, h') = h''$ judgment to combine the variable mappings from two environments:

$$\textsc{quoted-splice-resume}\quad \frac{H; e \Downarrow H'; \langle\!\langle e_q, h_q\rangle\!\rangle \qquad n = i \qquad \text{merge}(h, h_q) = h'}{H; h; {}_n[e] \Downarrow_i H'; h'; e_q}$$

For materialization escapes, the expression need not produce a code value. We add a new variable mapping to the environment h to hold to the materialized value and replace the materialization expression with a reference to the new variable. The judgment $\text{fresh}(x)$ ensures that the name x is not used elsewhere:

$$\textsc{quoted-materialize-resume}\quad \frac{H; e \Downarrow H'; v \qquad i = 1 \qquad \text{fresh}(x)}{H; h; \%[e] \Downarrow_i H'; x \mapsto v, h; x}$$

3. Safety Theorem

We state a safety theorem for BraidCore, which says that evaluation preserves the type of expressions.

To state the theorem, we define judgments for well-typed per-stage environments and complete heaps as $\vdash h : \gamma$ and $\vdash H : \Gamma$, respectively. These judgments map the expression typing judgment over the contained values. We also add a typing rule rule for code values $\langle\!\langle e, h\rangle\!\rangle$. Recall that code values do not appear in source programs, but they can be produced as results. The rule checks the wrapped expression in a context containing types for each of the variables in the environment h, which contains materialized values:

$$\textsc{type-code-value}\quad \frac{\gamma, \cdot \vdash e : \tau; \gamma, \cdot \qquad \vdash h : \gamma}{\Gamma \vdash \langle\!\langle e, h\rangle\!\rangle : \langle\tau\rangle; \Gamma}$$

Next, we state a lemma that expresses type preservation for the quoted evaluation judgment, $\Downarrow_i$. The idea is that, when quoted evaluation transforms an expression e_1 to e_2, it can only replace splice and materialization expressions with new expressions that match the types of the originals. These new expressions may be variable references, substituting for materialization expressions, or arbitrary spliced code, substituting for splices.

Lemma 1. (*Quoted type preservation*)
If $\gamma_1, \Gamma_1 \vdash e : \tau; \gamma_2, \Gamma_2$ and $H_1; h_1; e_1 \Downarrow_i H_1; h_1; e_1$ where $\vdash H_1 : \Gamma_1$ and $\vdash h_1 : \gamma_1$, then $\gamma_2, \Gamma_2 \vdash e_2 : \tau; \Gamma_2$ where $\vdash H_2 : \Gamma_2$ and $\vdash h_2 : \gamma_2$.

The lemma uses γ_1 and γ_2 as the types for the contexts that hold the materialized values, h_1 and h_2. The proof is by induction on the typing rules.

We can now state the main safety theorem.

Theorem 1. (*Type preservation*)
If $\cdot \vdash e : \tau; \Gamma$ and $\cdot; e \Downarrow H; v$, then $\cdot \vdash v : \tau; \cdot$.

Proof sketch. We show a stronger property that allows for for arbitrary well-typed initial heaps. Namely, we prove that if $\Gamma_1 \vdash e : \tau; \Gamma_2$ and $H_1; e \Downarrow H_2; v$ where $\vdash H_1 : \Gamma_1$, then $\vdash H_2 : \Gamma_2$ and $\Gamma_1 \vdash v : \tau; \Gamma_1$. The proof is by induction on the rules for the typing judgment.

The cases for literals (TYPE-CONST-INT and TYPE-CONST-FLOAT), sequences (TYPE-SEQ), and variables (TYPE-LOOKUP, and TYPE-VAR) follow from a standard correspondence with the dynamic semantics. We focus on the staging-related rules:

- TYPE-QUOTE. Here, e is a quote expression $\langle e' \rangle$ and τ is a quote type $\langle \tau' \rangle$. By the induction hypothesis, if $H; e' \Downarrow H'; v$, then $\cdot, \Gamma \vdash v : \tau'; \cdot, \Gamma$. This case requires the quoted type preservation lemma, above, which says that the heap h produced from quoted evaluation contains well-typed values. These types satisfy the premise of our TYPE-CODE-VALUE rule, above.
- TYPE-RUN. Here, e is a run expression $!e'$ where we assume $\Gamma_1 \vdash e' : \langle \tau' \rangle; \Gamma_2$. From the derivation of $H_1; e \Downarrow H_2; v$, we know that e' must produce a code value $\langle\!\langle e_q, h_q \rangle\!\rangle$. By the typing rule for code values, we have types γ_q for the values in the environment h_q and that $\gamma_q, \cdot \vdash e_q; \gamma_q, \cdot$. By the induction hypothesis, v has the type τ', as required.

The TEXT-SPLICE and TYPE-MATERIALIZE cases are vacuous because the corresponding expression forms do not have rules in the standard (non-quoted) evaluation semantics.