Target Langage

The target language (TL) is built around a parred down statically typed object calculus. The calculus has non-recursive structural substyping[1] without inheritance.[2] The calculus does not have null values.[3] Fields are accessed by automatically provided getter and setter methods, respectively denoted $\mathsf{x.f()}$ and $\mathsf{x.f(e)}$.[4] Fields appear in type signatures only through the presence of the getter and setter methods. Method calls are denoted $\mathsf{x.m(e)}$.[5] Following C# 4.0 [] the TL allows a dynamic type, denoted $\star$, and dynamically-typed method invocation, denoted $\mathsf{x@m(e)}$. Dynamically-typed method invocation treats arguments and return value as $\star$. A new object is created with a class name and a sequence of arguments in the order of definition of fields, denoted $\textbf{new}\ \mathsf{C}(\bar{\mathsf{e}})$. Meta-variable x ranges over argument names, a over memory locations, f over field names, m over method names, C over class names. The language has four different kinds of cast expressions: it has two structural casts and two generative casts. The structural subtype cast, denoted $<\mathsf{t}>\ \mathsf{a}$, asserts that the object at location a is of type t. The structural shallow cast, denoted $\prec \mathsf{t} \succ \mathsf{a}$, asserts that the object at location a has methods with names matching those of t. This does not make any guarantee about the type of arguments. The generative behavioral cast, denoted $\blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{a}$, will ensure that either a behaves as a t or that it get stuck. The generative monotonic cast, denoted $\triangleleft \mathsf{t} \triangleright \mathsf{a}$, is a behavioral cast that, in addition, imposes constraints on fields.

this is a distinguished variable name that denotes the current object. that is a distinguished field name that denotes the target of a wrapper.

D is a meta-variable used to range over dynamically generated class names.

$$
\begin{array}{lllll}
\mathsf{p} & ::= \mathsf{e}\ \overline{\mathsf{k}} & & & \\
\mathsf{e} & ::= \mathsf{x} & \mid \mathsf{this} & \mid \mathsf{that} & \\
 & \mid \mathsf{e.f()} & \mid \mathsf{e.f(e)} & \mid \mathsf{e.m(e)} & \mid \mathsf{e@m(e)} \\
 & \mid \textbf{new}\ \mathsf{C}(\bar{\mathsf{e}}) & & & \\
 & \mid <\mathsf{t}>\ \mathsf{e} & \mid \prec \mathsf{t} \succ \mathsf{e} & \mid \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e} & \mid \triangleleft \mathsf{t} \triangleright \mathsf{e} \\
 & \mid \mathsf{a} & & & \\
\mathsf{M} & ::= \cdot & \mid \overline{\mathsf{t} <: \mathsf{t}} & & \\
\mathsf{K} & ::= \cdot & \mid \overline{\mathsf{C} \mapsto \mathsf{k}} & &
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{k} & ::= \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f}: \mathsf{t}}\ \overline{\mathsf{md}}\,\} \\
\mathsf{md} & ::= \mathsf{m(x: t): t}\ \{\,\mathsf{e}\,\} \\
 & \quad \mid \mathsf{f(x: t): t}\ \{\,\mathsf{e}\,\} \mid \mathsf{f(): t}\ \{\,\mathsf{e}\,\} \\
\mathsf{mt} & ::= \mathsf{m(t): t} \mid \mathsf{f(t): t} \mid \mathsf{f(): t} \\
\Gamma & ::= \mathsf{x: t} \mid \cdot \\
\mathsf{t} & ::= \star \mid \mathsf{C} \mid \{\overline{\mathsf{mt}}\,\} \\
\sigma & ::= \cdot \mid \sigma\overline{[\mathsf{a} \mapsto \mathsf{C}\{\bar{\mathsf{a}}\}]}
\end{array}
$$

The semantics of TL is defined by a small step operational semantics with evaluation contexts. The context are deterministic. Program error is denoted by a stuck term. Meta-variable k ranges over class definitions. The semantics operate over an explicit class table, denoted K, which is a sequence of class definitions. A heap, denoted σ, maps memory locations to objects. We use the notation $\sigma[\mathsf{a} \mapsto \mathsf{C}\{\bar{\mathsf{a}}\}]$ to denote the heap σ extended by the binding of location a to object $\mathsf{C}\{\bar{\mathsf{a}}\}$. A configuration $\mathsf{K}\ \mathsf{e}\ \sigma$ evaluates in one step to a new configuration, denoted $\mathsf{K}\ \mathsf{e}\ \sigma \rightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'$. Execution terminate if e' is a value, a, or if there is no applicable reduction, in which case the program is stuck.[6]

New object creation picks a fresh memory location a' and binds it to the newly created. Operations on fields are require a typed receiver, for example in the expression $\mathsf{this.f()}$, this

---

[1] **I am not sure that this is the right way to describe what we have – check it is. JV**

[2] Structural subtyping is needed for type systems that have a notion of "consistency". Inheritance does not add anything other than code reuse.

[3] Null adds a source of error but otherwise should not be too interesting. We should mention what is the type of null.

[4] This is because we want an easy way to interpose on field access in wrappers.

[5] The single argument limitation is purely to lighten the notation.

[6] Intuitively we would expect that the program can only get stuck at dynamic calls or at structural casts. But generative casts add some complexity to that statement.

is always of the type of the current class. Field access through a getter method, works as follows. If the receiver's class has a getter method, that method is evaluated, otherwise the field corresponding the getter's name is updated. Methods are segregated into typed method (methods whose argument is not $\star$) and untyped method (methods whose argument is $\star$). The former can be called by statically resolved method, the latter must be called by dynamically resolved methods.

The auxiliary function `names` return the list of function names in a class or type, functions that have $\star$ as return values are prefixed with the symbol $\star$. The auxiliary function `typeof` return the type of the object at the location a in the heap σ. The auxiliary function `read` return the location a' pointing to the field f of the object at location a. The auxiliary function `write` return the heap $\sigma'$with the field f of the object at location a updated to the location a'.

$\boxed{\mathsf{K}\ \mathsf{e}\ \sigma \rightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'}$ $\mathsf{e}\ \sigma$ evaluates to e' in a step

$$\begin{array}{lll}
\mathsf{K}\ \textbf{new}\ \mathsf{C}(\bar{\mathsf{a}})\ \sigma & \rightarrow \mathsf{K}\ \mathsf{a}'\ \sigma' & \textit{if}\ \ \sigma' = \sigma[\mathsf{a}' \mapsto \mathsf{C}\{\bar{\mathsf{a}}\}] \wedge\ \mathsf{a}'\ \text{fresh} \\
\mathsf{K}\ \mathsf{a.f}()\ \sigma & \rightarrow \mathsf{K}\ [\mathsf{a}/\mathsf{this}]\mathsf{e}\ \sigma & \textit{if}\ \ \mathsf{f}()\colon \mathsf{t}\ \{\,\mathsf{e}\,\} \in \mathsf{method}(\sigma, \mathsf{a}, \mathsf{K}) \\
\mathsf{K}\ \mathsf{a.f}(\mathsf{a}')\ \sigma & \rightarrow \mathsf{K}\ [\mathsf{a}/\mathsf{this}\ \mathsf{a}'/\mathsf{x}]\mathsf{e}\ \sigma & \textit{if}\ \ \mathsf{f}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}\ \{\,\mathsf{e}\,\} \in \mathsf{method}(\sigma, \mathsf{a}, \mathsf{K}) \\
\mathsf{K}\ \mathsf{a.f}()\ \sigma & \rightarrow \mathsf{K}\ \mathsf{a}'\ \sigma & \textit{if}\ \ \mathsf{a}' = \mathsf{read}(\sigma, \mathsf{a}, \mathsf{f}) \\
\mathsf{K}\ \mathsf{a.f}(\mathsf{a}')\ \sigma & \rightarrow \mathsf{K}\ \mathsf{a}'\ \sigma' & \textit{if}\ \ \sigma' = \mathsf{write}(\sigma, \mathsf{a}, \mathsf{f}, \mathsf{a}') \\
\mathsf{K}\ \mathsf{a.m}(\mathsf{a}')\ \sigma & \rightarrow \mathsf{K}\ [\mathsf{a}/\mathsf{this}\ \mathsf{a}'/\mathsf{x}]\mathsf{e}\ \sigma & \textit{if}\ \ \mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}'\ \{\,\mathsf{e}\,\} \in \mathsf{method}(\sigma, \mathsf{a}, \mathsf{K}) \wedge \mathsf{t} \neq \star \\
\mathsf{K}\ \mathsf{a@m}(\mathsf{a}')\ \sigma & \rightarrow \mathsf{K}\ [\mathsf{a}/\mathsf{this}\ \mathsf{a}'/\mathsf{x}]\mathsf{e}\ \sigma & \textit{if}\ \ \mathsf{m}(\mathsf{x}\colon \star)\colon \star\ \{\,\mathsf{e}\,\} \in \mathsf{method}(\sigma, \mathsf{a}, \mathsf{K}) \\
\mathsf{K}\ <\mathsf{t}>\ \mathsf{a}\ \sigma & \rightarrow \mathsf{K}\ \mathsf{a}\ \sigma & \textit{if}\ \ \cdot \vdash \mathsf{mtypes}(\sigma, \mathsf{a}, \mathsf{K}) <: \mathsf{t}\ \vee\ \mathsf{t} = \star \\
\mathsf{K}\ \prec \mathsf{t} \succ \mathsf{a}\ \sigma & \rightarrow \mathsf{K}\ \mathsf{a}\ \sigma & \textit{if}\ \ \mathsf{names}(\mathsf{mtypes}(\sigma, \mathsf{a}, \mathsf{K})) \supseteq \mathsf{names}(\mathsf{t}) \\
\mathsf{K}\ \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{a}\ \sigma & \rightarrow \mathsf{K}'\ \mathsf{a}'\ \sigma' & \textit{if}\ \ \mathsf{K}'\ \mathsf{a}'\ \sigma' = \mathsf{behcast}(\mathsf{a}, \mathsf{t}, \sigma, \mathsf{K}) \\
\mathsf{K}\ \triangleleft \mathsf{t} \triangleright \mathsf{a}\ \sigma & \rightarrow \mathsf{K}'\ \mathsf{a}\ \sigma' & \textit{if}\ \ \mathsf{K}'\ \sigma' = \mathsf{moncast}(\mathsf{a}, \mathsf{t}, \sigma, \mathsf{K}) \\
\mathsf{K}\ \Gamma[\mathsf{e}]\ \sigma & \rightarrow \mathsf{K}'\ \Gamma[\mathsf{e}']\ \sigma' & \textit{if}\ \ \mathsf{K}\ \mathsf{e}\ \sigma \rightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'
\end{array}$$

$$\begin{array}{rl}
\Gamma & ::= \square.\mathsf{f}()\ \mid\ \square.\mathsf{f}(\mathsf{e})\ \mid\ \mathsf{a.f}(\square)\ \mid\ \square.\mathsf{m}(\mathsf{e})\ \mid\ \mathsf{a.m}(\square) \\
& \mid\ \square@\mathsf{m}(\mathsf{e})\ \mid\ \mathsf{a@m}(\square)\ \mid \\
& \mid\ <\mathsf{t}>\ \square\ \mid\ \prec \mathsf{t} \succ \square\ \mid\ \blacktriangleleft \mathsf{t} \blacktriangleright \square \\
& \mid\ \triangleleft \mathsf{t} \triangleright \square\ \mid\ \textbf{new}\ \mathsf{C}(\bar{\mathsf{a}}\,\square\,\bar{\mathsf{e}})
\end{array}$$

Types are represented as a set of methods.

$\boxed{\mathsf{M}; \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}'}$ t is a subtype of t'

$$\textsc{SRef}\ \frac{}{\mathsf{M}; \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}} \qquad \textsc{STop}\ \frac{}{\mathsf{M}; \mathsf{K} \vdash \mathsf{t} <: \{\}} \qquad \textsc{SAssump}\ \frac{\mathsf{t} <: \mathsf{t}' \in M}{\mathsf{M}; \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}'}$$

$$\textsc{SRec}\ \frac{\begin{array}{c}\mathsf{mtypes}(\mathsf{C}, \mathsf{K}) = \{\mathsf{mt}\ \overline{\mathsf{mt}_1}\,\} \qquad \mathsf{mtypes}(\mathsf{C}', \mathsf{K}) = \{\mathsf{mt}'\ \overline{\mathsf{mt}_2}\,\} \\ \mathsf{M}, \{\mathsf{mt}\ \overline{\mathsf{mt}_1}\,\} <: \{\mathsf{mt}'\ \overline{\mathsf{mt}_2}\,\}; \mathsf{K} \vdash \mathsf{mt} <: \mathsf{mt}' \\ \mathsf{M}, \{\mathsf{mt}\ \overline{\mathsf{mt}_1}\,\} <: \{\mathsf{mt}'\ \overline{\mathsf{mt}_2}\,\}; \mathsf{K} \vdash \{\overline{\mathsf{mt}_1}\} <: \{\overline{\mathsf{mt}_2}\}\end{array}}{\mathsf{M}; \mathsf{K} \vdash \mathsf{C} <: \mathsf{C}'}$$

$$\textsc{SMeth}\ \frac{\mathsf{M}; \mathsf{K} \vdash \mathsf{t}_2 <: \mathsf{t}_1 \qquad \mathsf{M}; \mathsf{K} \vdash \mathsf{t}_1' <: \mathsf{t}_2'}{\mathsf{M}; \mathsf{K} \vdash \mathsf{m}(\mathsf{t}_1)\colon \mathsf{t}_1' <: \mathsf{m}(\mathsf{t}_2)\colon \mathsf{t}_2'} \qquad \textsc{SFd1}\ \frac{}{\mathsf{M}; \mathsf{K} \vdash \mathsf{f}()\colon \mathsf{t} <: \mathsf{f}()\colon \mathsf{t}} \qquad \textsc{SFd2}\ \frac{}{\mathsf{M}; \mathsf{K} \vdash \mathsf{f}(\mathsf{t})\colon \mathsf{t} <: \mathsf{f}(\mathsf{t})\colon \mathsf{t}}$$

$\boxed{\sigma\ \mathsf{K} \vdash \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f}\colon \mathsf{t}}\ \overline{\mathsf{md}}\,\}\ \checkmark}$ Well-formed class

WFClass
$$\frac{\mathsf{names}(\overline{\mathsf{md}})\ \text{unique} \qquad \overline{\mathsf{f}}\ \text{unique} \qquad \cdot\ \cdot\ \mathsf{K} \vdash \overline{\mathsf{md}}\ \checkmark \qquad \mathsf{K} \vdash \overline{\mathsf{t}}\ \checkmark}{\mathsf{K} \vdash \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f}\colon \mathsf{t}}\ \overline{\mathsf{md}}\,\}\ \checkmark}$$

$\boxed{\sigma\ \mathsf{K} \vdash \mathsf{md}\ \checkmark}$ Well-formed method

WFMeth
$$\frac{\Gamma, \mathsf{x}\colon \mathsf{t}'\ \sigma \vdash \mathsf{e} : \mathsf{t} \qquad \mathsf{K} \vdash \mathsf{t}'\ \checkmark \qquad \mathsf{K} \vdash \mathsf{t}\ \checkmark}{\Gamma\ \sigma\ \mathsf{K} \vdash \mathsf{m}(\mathsf{x}\colon \mathsf{t}')\colon \mathsf{t}\ \{\,\mathsf{e}\,\}\ \checkmark}$$

WFFieldS
$$\frac{\Gamma, \mathsf{x}\colon \mathsf{t}'\ \sigma \vdash \mathsf{e} : \mathsf{t} \qquad \mathsf{K} \vdash \mathsf{t}'\ \checkmark \qquad \mathsf{K} \vdash \mathsf{t}\ \checkmark}{\Gamma\ \sigma\ \mathsf{K} \vdash \mathsf{f}(\mathsf{x}\colon \mathsf{t}')\colon \mathsf{t}\ \{\,\mathsf{e}\,\}\ \checkmark}$$

WFFieldR
$$\frac{\Gamma\ \sigma \vdash \mathsf{e} : \mathsf{t} \qquad \mathsf{K} \vdash \mathsf{t}\ \checkmark}{\Gamma\ \sigma\ \mathsf{K} \vdash \mathsf{f}()\colon \mathsf{t}\ \{\,\mathsf{e}\,\}\ \checkmark}$$

$\boxed{\mathsf{K} \vdash \mathsf{t}\ \checkmark}$ Well-formed types

WFStar
$$\frac{}{\mathsf{K} \vdash \star\ \checkmark}$$

WFClass
$$\frac{\mathsf{C} \in \mathrm{dom}(\mathsf{K})}{\mathsf{K} \vdash \mathsf{C}\ \checkmark}$$

Type checking is standard.

Field accessor rules W3 and W4 require a typed receiver, since $\star$ does not have any methods a receiver typed at $\star$ will never typecheck.

Shallow casts, W9, do not change the type of the expression. We are casting to the name of t not to t. In practice that means that all expression types in Transient will drift towards $\star$.

$\boxed{\Gamma\ \sigma \vdash \mathsf{e} : \mathsf{t}}$ e has type t in environment Γ against heap σ

W1
$$\frac{\Gamma(\mathsf{x}) = \mathsf{t}}{\Gamma\ \sigma \vdash \mathsf{x} : \mathsf{t}}$$

W2
$$\frac{\Gamma\ \sigma \vdash \mathsf{e} : \mathsf{t}' \qquad \cdot \vdash \mathsf{t}' <: \mathsf{t}}{\Gamma\ \sigma \vdash \mathsf{e} : \mathsf{t}}$$

W3
$$\frac{\Gamma\ \sigma \vdash \mathsf{e} : \mathsf{t} \qquad \mathsf{f}()\colon \mathsf{t}' \in \mathsf{t}}{\Gamma\ \sigma \vdash \mathsf{e.f}() : \mathsf{t}'}$$

W4
$$\frac{\Gamma\ \sigma \vdash \mathsf{e} : \mathsf{t} \qquad \mathsf{f}(\mathsf{t}')\colon \mathsf{t}' \in \mathsf{t} \qquad \Gamma\ \sigma \vdash \mathsf{e}' : \mathsf{t}'}{\Gamma\ \sigma \vdash \mathsf{e.f}(\mathsf{e}') : \mathsf{t}'}$$

W5
$$\frac{\Gamma\ \sigma \vdash \mathsf{e} : \mathsf{t} \qquad \mathsf{m}(\mathsf{t}')\colon \mathsf{t}'' \in \mathsf{t} \qquad \Gamma\ \sigma \vdash \mathsf{e}' : \mathsf{t}'}{\Gamma\ \sigma \vdash \mathsf{e.m}(\mathsf{e}') : \mathsf{t}''}$$

W6
$$\frac{\Gamma\ \sigma \vdash \mathsf{e} : \star \qquad \Gamma\ \sigma \vdash \mathsf{e}' : \star}{\Gamma\ \sigma \vdash \mathsf{e@m}(\mathsf{e}') : \star}$$

W7
$$\frac{\overline{\Gamma\ \sigma \vdash \mathsf{e} : \mathsf{t}} \qquad \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f}\colon \mathsf{t}}\ \overline{\mathsf{md}}\,\}}{\Gamma\ \sigma \vdash \textbf{new}\ \mathsf{C}(\overline{\mathsf{e}}) : \mathsf{C}}$$

W8
$$\frac{\Gamma\ \sigma \vdash \mathsf{e} : \mathsf{t}'}{\Gamma\ \sigma \vdash <\mathsf{t}>\ \mathsf{e} : \mathsf{t}}$$

W9
$$\frac{\Gamma\ \sigma \vdash \mathsf{e} : \mathsf{t}'}{\Gamma\ \sigma \vdash \prec \mathsf{t} \succ \mathsf{e} : \star}$$

W10
$$\frac{\Gamma\ \sigma \vdash \mathsf{e} : \mathsf{t}'}{\Gamma\ \sigma \vdash \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e} : \mathsf{t}}$$

W11
$$\frac{\Gamma\ \sigma \vdash \mathsf{e} : \mathsf{t}'}{\Gamma\ \sigma \vdash \triangleleft \mathsf{t} \triangleright \mathsf{e} : \mathsf{t}}$$

W12
$$\frac{\sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}_1}\}}{\Gamma\ \sigma \vdash \mathsf{a} : \mathsf{C}}$$

$\boxed{\mathsf{mdef}(\mathsf{m}, \mathsf{C}, \mathsf{K})}$ Auxiliary function: Method definition

$$\mathsf{mdef}(\mathsf{m}, \mathsf{C}, \mathsf{K}) = \mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}\ \{\,\mathsf{e}\,\}\ \ \textit{s.t.}\ \ \Big\{\mathsf{K}(\mathsf{C}) = \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f}\colon \mathsf{t}}\ \ldots \mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}\ \{\,\mathsf{e}\,\} \ldots\}$$

$\boxed{\mathsf{mtype}(\mathsf{m},\mathsf{C},\mathsf{K})}$ Auxiliary function: Method definition

$$\mathsf{mtype}(\mathsf{m},\mathsf{C},\mathsf{K}) = \mathsf{cnvtMD}(\mathsf{mdef}(\mathsf{m},\mathsf{C},\mathsf{K}))$$

$\boxed{\mathsf{mtypes}(\mathsf{C},\mathsf{K})}$ Auxiliary function: Type definition

$$\mathsf{mtypes}(\mathsf{C},\mathsf{K}) = \mathsf{MT}\ \ s.t.\ \begin{cases}\mathsf{K}(\mathsf{C}) = \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f}:\mathsf{t}}\ \overline{\mathsf{md}}\,\}\\ \mathsf{MT} = \{\mathsf{cnvtMD}(\overline{\mathsf{md}}) \oplus \forall\ \mathsf{f}':\mathsf{t}' \in \overline{\mathsf{f}:\mathsf{t}} \mid \mathsf{f}' \notin \overline{\mathsf{names}(\mathsf{md})}\ .\ \mathsf{cnvtFD}(\mathsf{f}':\mathsf{t}')\}\end{cases}$$

$\boxed{\mathsf{mtypes}(\mathsf{a},\sigma,\mathsf{K})}$ Auxiliary function: Type definition

$$\mathsf{mtypes}(\mathsf{a},\sigma,\mathsf{K}) = \mathsf{MT}\ \ s.t.\ \begin{cases}\sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}}\}\\ \mathsf{K}(\mathsf{C}) = \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f}:\mathsf{t}}\ \overline{\mathsf{md}}\,\}\\ \mathsf{MT} = \{\mathsf{cnvtMD}(\overline{\mathsf{md}}) \oplus \forall\ \mathsf{f}':\mathsf{t}' \in \overline{\mathsf{f}:\mathsf{t}} \mid \mathsf{f}' \notin \overline{\mathsf{names}(\mathsf{md})}\ .\ \mathsf{cnvtFD}(\mathsf{f}':\mathsf{t}')\}\end{cases}$$

$\boxed{\mathsf{field}(\mathsf{C},\mathsf{K})}$ Auxiliary function: Field definition

$$\mathsf{field}(\mathsf{C},\mathsf{K}) = \overline{\mathsf{f}:\mathsf{t}}\ \ s.t.\ \begin{cases}\mathsf{K}(\mathsf{C}) = \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f}:\mathsf{t}}\ \overline{\mathsf{md}}\,\}\end{cases}$$

$\boxed{\mathsf{method}(\mathsf{C},\mathsf{K})}$ Auxiliary function: Method definition

$$\mathsf{method}(\mathsf{C},\mathsf{K}) = \overline{\mathsf{md}}\ \ s.t.\ \begin{cases}\mathsf{K}(\mathsf{C}) = \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f}:\mathsf{t}}\ \overline{\mathsf{md}}\,\}\end{cases}$$

$\boxed{\mathsf{method}(\mathsf{a},\sigma,\mathsf{K})}$ Auxiliary function: Method definition

$$\mathsf{method}(\mathsf{a},\sigma,\mathsf{K}) = \overline{\mathsf{md}}\ \ s.t.\ \begin{cases}\sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}}\}\\ \mathsf{K}(\mathsf{C}) = \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f}:\mathsf{t}}\ \overline{\mathsf{md}}\,\}\end{cases}$$

$\boxed{\mathsf{read}(\sigma,\mathsf{a},\mathsf{f})}$ Auxiliary function: Field read

$$\mathsf{read}(\sigma,\mathsf{a},\mathsf{f}_i) = \mathsf{a}_i\ \ s.t.\ \begin{cases}\sigma(\mathsf{a}) = \mathsf{C}\{\ldots\mathsf{a}_i\ldots\}\\ \mathsf{K}(\mathsf{C}) = \textbf{class}\ \mathsf{C}\,\{\,\ldots\mathsf{f}_i:\mathsf{t}_i\ldots\ \overline{\mathsf{md}}\,\}\end{cases}$$

$\boxed{\mathsf{write}(\sigma,\mathsf{a},\mathsf{f},\mathsf{a}')}$ Auxiliary function: Field write

$$\mathsf{write}(\sigma, \mathsf{a}, \mathsf{f}_i, \mathsf{a}') = \sigma' \quad s.t. \begin{cases} \sigma(\mathsf{a}) = \mathsf{C}\{\ldots \mathsf{a}_i \ldots\} \\ \mathsf{K}(\mathsf{C}) = \textbf{class}\ \mathsf{C}\,\{\ldots \mathsf{f}_i : \mathsf{t}_i \ldots\ \overline{\mathsf{md}}\,\} \\ \sigma' = \sigma[\mathsf{a} \mapsto \mathsf{C}\{\ldots \mathsf{a}' \ldots\}] \end{cases}$$

$\boxed{\mathsf{cnvtMD}(\mathsf{md})}$ Conversion function: Method definition to method type

$\mathsf{cnvtMD}(\mathsf{m}(\mathsf{x} : \mathsf{t}) : \mathsf{t}\ \{\,\mathsf{e}\,\}) = \mathsf{m}(\mathsf{t}) : \mathsf{t}$
$\mathsf{cnvtMD}(\mathsf{f}(\mathsf{x} : \mathsf{t}) : \mathsf{t}\ \{\,\mathsf{e}\,\}) = \mathsf{f}(\mathsf{t}) : \mathsf{t}$
$\mathsf{cnvtMD}(\mathsf{f}() : \mathsf{t}\ \{\,\mathsf{e}\,\}) = \mathsf{f}() : \mathsf{t}$

$\boxed{\mathsf{cnvtFD}(\mathsf{f} : \mathsf{t})}$ Conversion function: Field definition to field types

$\mathsf{cnvtFD}(\mathsf{f} : \mathsf{t}) = \mathsf{f}(\mathsf{t}) : \mathsf{t},\ \mathsf{f}() : \mathsf{t}$

$\boxed{\mathsf{names}(\mathsf{mt})}$ Naming function: Method types

$\mathsf{names}(\mathsf{m}(\mathsf{x} : \mathsf{t}) : \mathsf{t}\ \{\,\mathsf{e}\,\}) = \mathsf{m}$
$\mathsf{names}(\mathsf{f}(\mathsf{x} : \mathsf{t}) : \mathsf{t}\ \{\,\mathsf{e}\,\}) = \mathsf{f}$
$\mathsf{names}(\mathsf{f}() : \mathsf{t}\ \{\,\mathsf{e}\,\}) = \mathsf{f}$

$\boxed{\mathsf{names}(\mathsf{md})}$ Naming function: Method definition[7]

$\mathsf{names}(\mathsf{m}(\mathsf{t}) : \mathsf{t}) = \mathsf{m}$
$\mathsf{names}(\mathsf{f}(\mathsf{t}) : \mathsf{t}) = \mathsf{f}$
$\mathsf{names}(\mathsf{f}() : \mathsf{t}) = \mathsf{f}$

1 Generative Casts

1.1 Behavioral

CAST-WRAP

$$\frac{\mathsf{D}\ \textit{fresh} \qquad \mathsf{a}'\ \textit{fresh} \qquad \mathsf{C}\{\overline{\mathsf{a}_1}\} = \sigma(\mathsf{a}) \qquad \sigma' = \sigma[\mathsf{a}' \mapsto \mathsf{D}\{\mathsf{a}\}] \qquad \mathsf{k} = \mathsf{wrap}(\mathsf{C}, \mathsf{t}, \mathsf{D})}{\mathsf{K},\ \mathsf{D} \mapsto \mathsf{k}\quad \mathsf{a}'\ \sigma' = \mathsf{behcast}(\mathsf{a}, \mathsf{t}, \sigma, \mathsf{K})}$$

What does the wrap function do? In english.

At its core, the wrap function protects types. If a function on an object has a declared argument type, then the wrap function will produce a wrapper for that function that ensures that any argument passed in matches the declared type. Likewise, if we assert that an untyped function has a return type, then the generated wrapper guarantees that the returned value of the untyped function is of the asserted type by inserting a cast to the right type.

In our context, wrappers are just generated classes, produced when the runtime system encounters a behavioral cast. These casts need to ensure two key properties, which we

[7] All of these simply functions should go into the appendix.

will refer to as *soundness* and *completeness*. In the context of casts, soundness refers to correct enforcement of types, whereby protected methods will not observably violate their type guarantees, while completeness ensures that a cast will not lose methods.

This last requirement is somewhat unconventional, and its need is illustrated in a simple example.

```
class C {
  m(x:int):int { ... }
}
class D {
  m(x:*):* { ... }
  f(x:*):* { ... }
}
(<D>(<C>new D()))@f(2)
```

In this example, if we were to implement wrappers by wrapping over only the declared methods, despite D having a method f, it is "lost" when D is cast to C. As a result, when we cast it back from C to D, the wrapper added to ensure that C's invariants held does not have a method f, and our call will produce a method not understood exception. To avoid this issue, our wrappers will have to retain all untyped methods when casting untyped to typed, and typed methods if a typed cast does not mention them, so that they may be recovered in a later cast.

$$\mathsf{n} ::= \mathsf{m} \mid \mathsf{f}$$

$$\begin{array}{l}
\mathsf{wrap}(\mathsf{C},\mathsf{C}', \mathsf{D}) = \\
\quad \mathsf{class\ D}\ \{ \\
\quad\quad \mathsf{that} : \mathsf{C} \\
\quad\quad \mathsf{n}(\overline{\mathsf{x} : \mathsf{t}_2}) : \mathsf{t}'_2\ \{\ \blacktriangleleft \mathsf{t}'_2 \blacktriangleright \mathsf{this.that.n}(\overline{\blacktriangleleft \mathsf{t}_1 \blacktriangleright \mathsf{x}})\ \} \\
\quad\quad\quad \textit{for every}\ \mathsf{n}(\overline{\mathsf{t}_1}) : \mathsf{t}'_1 \in \mathsf{mtypes}(\mathsf{C},\mathsf{K}) \wedge \mathsf{n}(\overline{\mathsf{t}_2}) : \mathsf{t}'_2 \in \mathsf{mtypes}(\mathsf{C}',\mathsf{K}) \\
\\
\quad\quad \mathsf{n}(\overline{\mathsf{x} : \mathsf{t}_1}) : \mathsf{t}'_1\ \{\ \mathsf{this.that.n}(\overline{\mathsf{x}})\ \} \\
\quad\quad\quad \textit{for every}\ \mathsf{n}(\overline{\mathsf{t}_1}) : \mathsf{t}'_1 \in \mathsf{mtypes}(\mathsf{C},\mathsf{K}) \wedge \mathsf{n}(\overline{\mathsf{t}_2}) : \mathsf{t}'_2 \notin \mathsf{mtypes}(\mathsf{C}',\mathsf{K}) \\
\\
\quad\quad \mathsf{n}(\overline{\mathsf{x} : \mathsf{t}_2}) : \mathsf{t}'_2\ \{\ \mathbf{new}\ \mathsf{Error}()\mathsf{@error}()\ \} \\
\quad\quad\quad \textit{for every}\ \mathsf{n}(\overline{\mathsf{t}_1}) : \mathsf{t}'_1 \notin \mathsf{mtypes}(\mathsf{C},\mathsf{K}) \wedge \mathsf{n}(\overline{\mathsf{t}_2}) : \mathsf{t}'_2 \in \mathsf{mtypes}(\mathsf{C}',\mathsf{K}) \\
\quad \} \\
\\
\mathsf{wrap}(\mathsf{C},\star, \mathsf{D}) = \\
\quad \mathsf{class\ D}\ \{ \\
\quad\quad \mathsf{that} : \mathsf{C} \\
\\
\quad\quad \mathsf{n}(\overline{\mathsf{x} : \star}) : \star\ \{\ \blacktriangleleft \star \blacktriangleright \mathsf{this.that.n}(\overline{\blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{x}})\ \} \\
\quad\quad\quad \textit{for every}\ \mathsf{n}(\overline{\mathsf{t}}) : \mathsf{t}' \in \mathsf{mtypes}(\mathsf{C},) \\
\quad \}
\end{array}$$

wrapClassis more complex than TODO, because different wrappers are needed, depending on if the target type is a concrete type C, or is the dynamic type $\star$. If the target type is C, then it will generate the wrapper methods for the type C, then insert "passthrough" methods. methods of the same type as the original and that just call the original internally, that provide the completness property mentioned above. Likewise, if the target type is $\star$, then wrapClasswill simply generate a method that casts its argument to the right type and its return type to $\star$ for every one of the typed methods in the source, ensuring soundness and completness.

1.2 Monotone

Monotonic aims to have *every typed reference still be valid*, including the ones that previously existed, as well as the reference created by the cast. If a reference has a type with a non-star value, then the monotonic semantics will ensure that that type is not violated. To accomplish this, the monotonic cast needs to make sure of two properties:

- The values that *currently* exist cannot violate any of the types that point to them. Casting needs to recursively ensure that all values referred to by the current object are of the claimed type.
- Functions can not be called with or return values that violate any of the types that they are referred to with. The behaviour of the class needs to check that its types are not violated by lesser-typed call sites.

The second property is strongly reminisicent of the behavioural semantics, though with the interesting caveat that we now need to make sure that *all* method invocations follow the typed calling conventions, rather than just the ones that inherit this particular type assertion.

Cast-Mono (monotonic)

$$\frac{\mathsf{rectype}(\cdot, \cdot, \sigma, \mathsf{a}, \mathsf{t}) = \Sigma, \Omega \qquad \mathsf{spec}(\Sigma, \Omega, \mathsf{K}, \sigma) = \sigma', \mathsf{K}'}{\mathsf{K}'\ \sigma' = \mathsf{moncast}(\mathsf{a}, \mathsf{t}, \sigma, \mathsf{K})}$$

$$\Sigma ::= \overline{\mathsf{a} : \mathsf{t}}$$

K adds ^s

HT1

$$\frac{\mathsf{a} \notin \mathrm{dom}(\Sigma)}{\mathsf{htype}(\mathsf{a}, \Sigma, \sigma[\mathsf{a} \mapsto \mathsf{C}\{\overline{\mathsf{a}'}\}]) = \mathsf{C}}$$

HT2

$$\frac{}{\mathsf{htype}(\mathsf{a}, \Sigma\ \mathsf{a} : \mathsf{t}\ \Sigma, \sigma) = \mathsf{t}}$$

RC1

Ω

$$\frac{\mathsf{t}' \neq \mathsf{t}'' \qquad \begin{array}{c}\mathsf{htype}(\mathsf{a}, \Sigma, \sigma) = \mathsf{t}' \\ \mathsf{fieldtypes}(\mathsf{t}'', \mathsf{K}, \sigma, \mathsf{a}) = \overline{\mathsf{a}'}, \overline{\mathsf{t}_f}\end{array} \qquad \begin{array}{c}\emptyset, \Omega, \mathsf{K} \vdash \mathsf{t}' \sqcap \mathsf{t} \equiv \mathsf{t}'' \dashv \Omega' \\ \hline \mathsf{rectype}(\Sigma \sqcap \mathsf{a}' : \mathsf{t}, \sigma, \mathsf{a}', \mathsf{t}_f) = \Sigma'\end{array}}{\mathsf{rectype}(\Sigma, \Omega, \sigma, \mathsf{a}, \mathsf{t}) = \left(\prod \overline{\Sigma'}\right), \Omega'}$$ K, Ω, Ω''

RC2

$$\frac{\mathsf{htype}(\mathsf{a}, \Sigma, \sigma) = \mathsf{t}' \qquad \emptyset, \Omega, \mathsf{K} \vdash \mathsf{t}' \sqcap \mathsf{t} \equiv \mathsf{t}'' \dashv \Omega \qquad \mathsf{t}' = \mathsf{t}''}{\mathsf{rectype}(\Sigma, \Omega, \sigma, \mathsf{a}, \mathsf{t}) = \Sigma, \Omega}$$

RC3

C{}

D Free/Fresh K' = K D ↦ classgen(C, t)

SP1
$$\frac{\mathsf{classgen}(\mathsf{C},\mathsf{t}) = \mathsf{D} \qquad \mathsf{spec}(\Sigma,\sigma,\mathsf{K}) = \sigma',\mathsf{K}'}{\mathsf{spec}(\mathsf{a}\ \Sigma, \sigma[\mathsf{a}\mapsto \mathsf{C}\{\overline{\mathsf{a}'}\}],\mathsf{K}) = \sigma'[\mathsf{a}\mapsto \mathsf{D}\{\overline{\mathsf{a}'}\}],\mathsf{K}';\mathsf{D}}$$

:t

↦ classgen output

SP2
$$\mathsf{spec}(\cdot,\cdot,\mathsf{K}) = \cdot,\cdot$$

spec(·, σ, K) = σ, K

$\boxed{\mathsf{t}_1 \sqcap \mathsf{t}_2 \equiv \mathsf{t}}$ The most specific type common to t_1 and t_2 is t

M1
$$\frac{}{\Phi,\Omega,\mathsf{K}\vdash \mathsf{t}\sqcap\star\equiv\mathsf{t}\dashv\cancel{\Phi}\ \Omega}$$

M2
$$\frac{}{\Phi,\Omega,\mathsf{K}\vdash \star\sqcap\mathsf{t}\equiv\mathsf{t}\dashv\cancel{\Phi}\ \Omega}$$

M3
$$\frac{}{\Phi,\Omega,\mathsf{K}\vdash \mathsf{t}\sqcap\mathsf{t}\equiv\mathsf{t}\dashv\cancel{\Phi}\ \Omega}$$

M4
$$\frac{\overline{\Phi,\Omega,\mathsf{K}\vdash \mathsf{t}_3\sqcap\mathsf{t}_1\equiv\mathsf{t}_5\dashv\Omega\ \Omega'} \qquad \Phi,\Omega,\mathsf{K}\vdash \mathsf{t}_2\sqcap\mathsf{t}_4\equiv\mathsf{t}_6\dashv\Omega\ \Omega'' \qquad \Phi,\Omega,\mathsf{K}\vdash \{\overline{\mathsf{mt}_1}\}\sqcap\{\overline{\mathsf{mt}_2}\}\equiv\{\overline{\mathsf{mt}_3}\}\dashv\Omega\ \Omega'''}{\Phi,\Omega,\mathsf{K}\vdash \{\mathsf{n}(\overline{\mathsf{t}_1}):\mathsf{t}_2\ \overline{\mathsf{mt}_1}\}\sqcap\{\mathsf{n}(\overline{\mathsf{t}_3}):\mathsf{t}_4\ \overline{\mathsf{mt}_2}\}\equiv\{\mathsf{m}(\overline{\mathsf{t}_5}):\mathsf{t}_6\ \overline{\mathsf{mt}_3}\}\dashv\Omega\ \overline{\Omega'}\ \Omega''\ \Omega'''}$$

Σ ⊓ Σ

M5
$$\frac{\mathsf{C}\sqcap\mathsf{D}\notin\mathrm{dom}(\Phi) \qquad \Phi\ \mathsf{C}\sqcap\mathsf{D}=\mathsf{E},\Omega,\mathsf{K}\vdash \mathrm{typeof}(\mathsf{C})\sqcap\mathrm{typeof}(\mathsf{D})\equiv\mathsf{t}\dashv\Omega'}{\Phi,\Omega,\mathsf{K}\vdash\mathsf{C}\sqcap\mathsf{D}\equiv\mathsf{E}\dashv\Omega'\ \mathsf{E}\mapsto\mathsf{t}}$$

{ mt3} ⊓ {3}

M6
$$\frac{\Phi(\mathsf{C}\sqcap\mathsf{D})=\mathsf{E}}{\Phi,\Omega,\mathsf{K}\vdash\mathsf{C}\sqcap\mathsf{D}\equiv\mathsf{E}\dashv\Omega}$$

Φ = C ⊓ D = E

Refine-Method
$$\frac{\overline{\mathsf{t}_1\sqcap\mathsf{t}_1'=\mathsf{t}_1''} \qquad \mathsf{t}_2\sqcap\mathsf{t}_2'=\mathsf{t}_2''}{\mathsf{spec}(\mathsf{m}(\overline{\mathsf{x}:\mathsf{t}_1}):\mathsf{t}_2=\mathsf{e},\mathsf{m}(\overline{\mathsf{t}_1'}):\mathsf{t}_2') = (\mathsf{m}(\overline{\mathsf{x}:\mathsf{t}_1''}):\mathsf{t}_2'' = <\mathsf{t}_2''>\ \mathsf{e}),\ (\mathsf{m_u}(\overline{\mathsf{x}:\star}):\star = <\star>\ \mathsf{this.m}(\overline{<\mathsf{t}_1''>\ \mathsf{x}}))}$$

Refine-Class
$$\frac{\overline{\mathsf{t}\sqcap\mathsf{t}'=\mathsf{t}''}}{\mathsf{spec}(\mathbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f}:\mathsf{t}}\ \overline{\mathsf{md}}\,\},\mathsf{D},\{\overline{\mathsf{f}():\mathsf{t}'},\overline{\mathsf{mt}}\}) = \mathbf{class}\ \mathsf{D}\,\{\,\overline{\mathsf{f}:\mathsf{t}''}\ \overline{\mathsf{spec}(\mathsf{f}:\mathsf{t},\mathsf{t}')},\overline{\mathsf{spec}(\mathsf{md},\mathsf{mt})}\,\}}$$

Insert-Recv (eager cast)
$$\frac{\Gamma\vdash\mathsf{e}_1\hookrightarrow\mathsf{e}_3\Uparrow\mathsf{t}_1 \qquad \mathsf{m}(\overline{x:\mathsf{t}_2}):\mathsf{t}_3\in\mathsf{t}_1 \qquad \overline{\Gamma\vdash\mathsf{e}_2\Downarrow\mathsf{t}_2\hookrightarrow\mathsf{e}_4}}{\mathsf{invoke}(\Gamma,\mathsf{e}_1.\mathsf{m}(\overline{\mathsf{e}_2})) = \mathsf{e}_3.\mathsf{m}(\overline{\mathsf{e}_4}),\mathsf{t}_3}$$

Insert-Dyn
$$\frac{\Gamma\vdash\mathsf{e}_1\hookrightarrow\mathsf{e}_3\Uparrow\star \qquad \overline{\Gamma\vdash\mathsf{e}_2\Downarrow\star\hookrightarrow\mathsf{e}_4}}{\mathsf{invoke}(\Gamma,\mathsf{e}_1.\mathsf{m}(\overline{\mathsf{e}_2})) = <(\{\mathsf{m_u}(\overline{\star}):\star\}>\ \mathsf{e}_3).\mathsf{m_u}(\overline{\mathsf{e}_4}),\star}$$

Insert-Check (lazy cast)
$$\frac{\Gamma\vdash\mathsf{e}_1\hookrightarrow\mathsf{e}_3\Uparrow\mathsf{t}_1 \qquad \mathsf{m}(\overline{x:\mathsf{t}_2}):\mathsf{t}_3\in\mathsf{t}_1 \qquad \overline{\Gamma\vdash\mathsf{e}_2\Downarrow\mathsf{t}_2\hookrightarrow\mathsf{e}_4}}{\mathsf{invoke}(\Gamma,\mathsf{e}_1.\mathsf{m}(\overline{\mathsf{e}_2})) = <\mathsf{t}_1>\ ((<\{\mathsf{m_u}(\overline{\star}):\star\}>\ \mathsf{e}_3).\mathsf{m_u}(\overline{<\star>\ \mathsf{e}_4})),\mathsf{t}_1}$$

FT1
$$\frac{\sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}}\} \qquad \mathsf{K}(\mathsf{C}) = \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f} : \mathsf{t}}\ \overline{\mathsf{md}}\,\} \qquad |\,\overline{\mathsf{a}}\,| = |\,\overline{\mathsf{f} : \mathsf{t}}\,|}{\mathsf{fieldtypes}(\mathsf{C}, \mathsf{K}, \sigma, \mathsf{a}) = \overline{\mathsf{a}}, \mathsf{typeof}(\overline{\mathsf{f} : \mathsf{t}})}$$

f_i, f_2

FT2
$$\frac{\sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}}\} \qquad \mathsf{K}(\mathsf{C}) = \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f} : \mathsf{t}}\ \overline{\mathsf{md}}\,\} \qquad |\,\overline{\mathsf{a}}\,| = |\,\overline{\mathsf{f} : \mathsf{t}}\,|}{\mathsf{fieldtypes}(\star, \mathsf{K}, \sigma, \mathsf{a}) = \overline{\mathsf{a}}, \mathsf{typeof}(\overline{\mathsf{f} : \mathsf{t}})}$$

fnames({mt}) ⊆ names(f:t)

FT3
$$\frac{\begin{array}{c}\sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}}\} \qquad \mathsf{K}(\mathsf{C}) = \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f} : \mathsf{t}}\ \overline{\mathsf{md}}\,\} \\ \forall\, \mathsf{f} \in \mathsf{names}(\overline{\mathsf{f} : \mathsf{t}})\ .\ \mathsf{f} \in \mathsf{names}(\{\overline{\mathsf{mt}}\}) \qquad \overline{\mathsf{t}_f} = \forall\, \mathsf{f} \in \mathsf{names}(\overline{\mathsf{f} : \mathsf{t}})\ .\ \mathsf{types}(\mathsf{f}, \{\overline{\mathsf{mt}}\})\end{array}}{\mathsf{fieldtypes}(\{\overline{\mathsf{mt}}\}, \mathsf{K}, \sigma, \mathsf{a}) = \overline{\mathsf{a}}, \overline{\mathsf{t}_f}}$$

change
gradual guarantee

TY
$$\frac{}{\mathsf{types}(\mathsf{f},\ \ldots\ \mathsf{f}(\mathsf{t}) : \mathsf{t}\ \ldots\ \mathsf{f}() : \mathsf{t}\ \ldots) = \mathsf{t}}$$

classgen(C,t) =

class D {

$\mathsf{n}(\overline{\mathsf{x} : \mathsf{t}_2}) : \mathsf{t}_2' \ \{ \blacktriangleleft \mathsf{t}_2' \blacktriangleright \mathsf{this.that.n}(\overline{\blacktriangleleft \mathsf{t}_1 \blacktriangleright \mathsf{x}}) \}$

for every $\mathsf{n}(\overline{\mathsf{t}_1}) : \mathsf{t}_1' \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K}) \wedge \mathsf{n}(\overline{\mathsf{t}_2}) : \mathsf{t}_2' \in \mathsf{mtypes}(\mathsf{C}', \mathsf{K})$

$\mathsf{n}(\overline{\mathsf{x} : \mathsf{t}_1}) : \mathsf{t}_1' \ \{ \mathsf{this.that.n}(\overline{\mathsf{x}}) \}$

for every $\mathsf{n}(\overline{\mathsf{t}_1}) : \mathsf{t}_1' \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K}) \wedge \mathsf{n}(\overline{\mathsf{t}_2}) : \mathsf{t}_2' \notin \mathsf{mtypes}(\mathsf{C}', \mathsf{K})$

$\mathsf{n}(\overline{\mathsf{x} : \mathsf{t}_2}) : \mathsf{t}_2' \ \{ \textbf{new}\ \mathsf{Error}()@\mathsf{error}() \}$

for every $\mathsf{n}(\overline{\mathsf{t}_1}) : \mathsf{t}_1' \notin \mathsf{mtypes}(\mathsf{C}, \mathsf{K}) \wedge \mathsf{n}(\overline{\mathsf{t}_2}) : \mathsf{t}_2' \in \mathsf{mtypes}(\mathsf{C}', \mathsf{K})$

}

do fully typed case

TO
$$\frac{}{\mathsf{typeof}(\mathsf{f} : \mathsf{t}) = \mathsf{t}}$$

2 Translations

Our core language supports the core functionality of gradual typing, but none of the surface syntax or quality of life features that full gradual tyyping systems do. To provide a representative surface syntax, we define several translations that convert surface syntax programs in an approximation of the source language into core language ones.

2.1 Strongscript

$\mathsf{t} \ ::= \ \star \ | \ \mathsf{C} \ | \ !\mathsf{C}$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x \hookrightarrow x \Uparrow t}\ \text{A1} \qquad \frac{\Gamma \vdash e_1 \hookrightarrow e_3 \Uparrow !C \quad m(t_1) : t_2 \in !C \quad \Gamma \vdash e_2 \Downarrow t_1 \hookrightarrow e_4}{\Gamma \vdash e_1.m(e_2) \hookrightarrow e_3.m(e_4) \Uparrow t_2}\ \text{A2}$$

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_3 \Uparrow !C \quad f(t_1) : t_1 \in !C \quad \Gamma \vdash e_2 \Downarrow t_1 \hookrightarrow e_4}{\Gamma \vdash e_1.f(e_2) \hookrightarrow e_3.f(e_4) \Uparrow t_1}\ \text{A3}$$

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_2 \Uparrow !C \quad f() : t_1 \in !C}{\Gamma \vdash e_1.f() \hookrightarrow e_2.f() \Uparrow t_1}\ \text{A4}$$

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_3 \Uparrow C \quad m(t_1) : !C \in C \quad \Gamma \vdash e_2 \Downarrow t_1 \hookrightarrow e_4}{\Gamma \vdash e_1.m(e_2) \hookrightarrow < C > \ e_3@m(e_4) \Uparrow !C}\ \text{A5}$$

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_3 \Uparrow C \quad m(t_1) : C \in C \quad \Gamma \vdash e_2 \Downarrow t_1 \hookrightarrow e_4}{\Gamma \vdash e_1.m(e_2) \hookrightarrow e_3@m(e_4) \Uparrow C}\ \text{A6}$$

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_3 \Uparrow C \quad f(t_1) : t_1 \in C \quad \Gamma \vdash e_2 \Downarrow t_1 \hookrightarrow e_4}{\Gamma \vdash e_1.f(e_2) \hookrightarrow e_3@f(e_4) \Uparrow t_1}\ \text{A6} \qquad \frac{\Gamma \vdash e_1 \hookrightarrow e_2 \Uparrow C \quad f() : t_1 \in C}{\Gamma \vdash e_1.f() \hookrightarrow e_2@f() \Uparrow t_1}\ \text{A7}$$

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_3 \Uparrow \star \quad \Gamma \vdash e_2 \Downarrow \star \hookrightarrow e_4}{\Gamma \vdash e_1.m(e_2) \hookrightarrow e_3@m(e_4) \Uparrow \star}\ \text{A8} \qquad \frac{\Gamma \vdash e_1 \hookrightarrow e_3 \Uparrow \star \quad \Gamma \vdash e_2 \Downarrow \star \hookrightarrow e_4}{\Gamma \vdash e_1.f(e_2) \hookrightarrow e_3@f(e_4) \Uparrow \star}\ \text{A9}$$

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_2 \Uparrow \star}{\Gamma \vdash e_1.f() \hookrightarrow e_2@f() \Uparrow \star}\ \text{A10} \qquad \frac{\overline{\Gamma \vdash e_1 \Downarrow t \hookrightarrow e_2} \quad \mathbf{class}\ C \{ \overline{f : t}\ \overline{md} \}}{\Gamma \vdash \mathbf{new}\ C(\overline{e_1}) \hookrightarrow \mathbf{new}\ C(\overline{e_2}) \Uparrow C}\ \text{A11}$$

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_2 \Uparrow !C_2 \quad C_2 <: C_1}{\Gamma \vdash e_1 \Downarrow !C_1 \hookrightarrow e_2}\ \text{AASC1} \qquad \frac{\Gamma \vdash e_1 \hookrightarrow e_2 \Uparrow !C_2 \quad C_2 <: C_1}{\Gamma \vdash e_1 \Downarrow C_1 \hookrightarrow e_2}\ \text{AASC2}$$

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_2 \Uparrow C_2 \quad C_2 <: C_1}{\Gamma \vdash e_1 \Downarrow C_1 \hookrightarrow e_2}\ \text{AASC3} \qquad \frac{\Gamma \vdash e_1 \hookrightarrow e_2 \Uparrow C}{\Gamma \vdash e_1 \Downarrow !C \hookrightarrow < C > \ e_2}\ \text{AASC4} \qquad \frac{\Gamma \vdash e_1 \hookrightarrow e_2 \Uparrow \star}{\Gamma \vdash e_1 \Downarrow !C \hookrightarrow < C > \ e_2}\ \text{AASC5}$$

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_2 \Uparrow \star}{\Gamma \vdash e_1 \Downarrow C \hookrightarrow e_2}\ \text{AASC6} \qquad \frac{\Gamma \vdash e_1 \hookrightarrow e_2 \Uparrow t \quad t \neq \star}{\Gamma \vdash e_1 \Downarrow \star \hookrightarrow e_2}\ \text{AASC7}$$