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

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

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

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

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

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

$\boxed{\mathsf{names(md)}}$ Naming function: Method definition[1]

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

1 Generative Casts

1.1 Behavioral

Cast-Wrap

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

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 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.

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

```
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.

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

$\mathsf{wrap}(\mathsf{C}, \mathsf{C}', \mathsf{D}) =$
        $\mathsf{class\ D}\ \{$
                $\mathsf{that} : \mathsf{C}$
                $\mathsf{n}(\overline{\mathsf{x} : \mathsf{t}_2}) : \mathsf{t}'_2\ \{\blacktriangleleft \mathsf{t}'_2 \blacktriangleright \mathsf{this.that.n}(\overline{\blacktriangleleft \mathsf{t}_1 \blacktriangleright \mathsf{x}})\}$
                  $\textit{for every}\ \mathsf{n}(\overline{\mathsf{t}_1}) : \mathsf{t}'_1 \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K}) \wedge \mathsf{n}(\overline{\mathsf{t}_2}) : \mathsf{t}'_2 \in \mathsf{mtypes}(\mathsf{C}', \mathsf{K})$

                $\mathsf{n}(\overline{\mathsf{x} : \mathsf{t}_1}) : \mathsf{t}'_1\ \{\mathsf{this.that.n}(\overline{\mathsf{x}})\}$
                  $\textit{for every}\ \mathsf{n}(\overline{\mathsf{t}_1}) : \mathsf{t}'_1 \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K}) \wedge \mathsf{n}(\overline{\mathsf{t}_2}) : \mathsf{t}'_2 \not\in \mathsf{mtypes}(\mathsf{C}', \mathsf{K})$

                $\mathsf{n}(\overline{\mathsf{x} : \mathsf{t}_2}) : \mathsf{t}'_2\ \{(<\star>\ \mathsf{this})@\mathsf{m}(<\star>\ \mathsf{this})\}$
                  $\textit{for every}\ \mathsf{n}(\overline{\mathsf{t}_1}) : \mathsf{t}'_1 \not\in \mathsf{mtypes}(\mathsf{C}, \mathsf{K}) \wedge \mathsf{n}(\overline{\mathsf{t}_2}) : \mathsf{t}'_2 \in \mathsf{mtypes}(\mathsf{C}', \mathsf{K}) \wedge \mathsf{m}\ \text{fresh}$
        $\}$

$\mathsf{wrap}(\mathsf{C}, \star, \mathsf{D}) =$
        $\mathsf{class\ D}\ \{$
                $\mathsf{that} : \mathsf{C}$

                $\mathsf{n}(\overline{\mathsf{x} : \star}) : \star\ \{\blacktriangleleft \star \blacktriangleright \mathsf{this.that.n}(\overline{\blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{x}})\}$
                  $\textit{for every}\ \mathsf{n}(\overline{\mathsf{t}}) : \mathsf{t}' \in \mathsf{mtypes}(\mathsf{C},)$
        $\}$
```

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.

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

Cast-Mono (monotonic)

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

HT1

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

HT2

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

$$\boxed{\mathsf{J} \vdash \mathsf{t} \approx \mathsf{t}_1}$$

EQ1

$$\frac{\cdot \vdash \mathsf{t} <: \mathsf{t}' \qquad \cdot \vdash \mathsf{t}' <: \mathsf{t}}{\mathsf{t} \approx \mathsf{t}'}$$

RC1

$$\frac{\begin{array}{c}\mathsf{hType}(\mathsf{a},\Sigma,\sigma) = \mathsf{t}' \qquad \mathsf{meet}(\cdot,\mathsf{K},\mathsf{t}',\mathsf{t}) = \mathsf{t}''\ \mathsf{K}' \qquad \cdot \vdash \mathsf{t}' \not\approx \mathsf{t}'' \\ \mathsf{fieldTypes}(\mathsf{t}'',\mathsf{K}',\sigma,\mathsf{a}) = \mathsf{a}_1 \ldots \mathsf{a}_n\ \mathsf{t}_1 \ldots \mathsf{t}_n \qquad \Sigma_1 = \mathrm{update}(\mathsf{a} : \mathsf{t}'',\Sigma) \qquad \mathsf{K}_1 = \mathsf{K}' \\ \mathsf{recType}(\mathsf{a}_1,\mathsf{t}_1,\sigma,\mathsf{K}_1,\Sigma_1) = \Sigma_2\ \mathsf{K}_2 \quad \ldots \quad \mathsf{recType}(\mathsf{a}_n,\mathsf{t}_n,\sigma,\mathsf{K}_n,\Sigma_n) = \Sigma_{n+1}\ \mathsf{K}_{n+1}\end{array}}{\mathsf{recType}(\mathsf{a},\mathsf{t},\sigma,\mathsf{K},\Sigma) = \Sigma_{n+1},\mathsf{K}_{n+1}}$$

RC2

$$\frac{\mathsf{hType}(\mathsf{a},\Sigma,\sigma) = \mathsf{t}' \qquad \mathsf{meet}(\cdot,\mathsf{K},\mathsf{t}',\mathsf{t}) = \mathsf{t}''\ \mathsf{K}' \qquad \cdot \vdash \mathsf{t}' \approx \mathsf{t}''}{\mathsf{recType}(\mathsf{a},\mathsf{t},\sigma,\mathsf{K},\Sigma) = \Sigma\ \mathsf{K}}$$

SP1
$$\frac{\mathsf{spec}(\Sigma, \sigma, \mathsf{K}) = \sigma'}{\mathsf{spec}(\mathsf{a} : \mathsf{D}\ \Sigma, \sigma[\mathsf{a} \mapsto \mathsf{C}\{\overline{\mathsf{a}'}\}], \mathsf{K}) = \sigma'[\mathsf{a} \mapsto \mathsf{D}\{\overline{\mathsf{a}'}\}]}$$

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

The spec functin.

$$\boxed{\mathsf{meet}(\mathsf{P}, \mathsf{K}, \mathsf{t}, \mathsf{t}') = \mathsf{t}''\ \mathsf{K}}$$

$$\mathsf{P} ::= \overline{(\mathsf{t}, \mathsf{t}') \mapsto \mathsf{t}''}$$

M1
$$\frac{}{\mathsf{meet}(\mathsf{P}, \mathsf{K}, \mathsf{t}, \star) = \mathsf{t}\ \mathsf{K}}$$

M2
$$\frac{}{\mathsf{meet}(\mathsf{P}, \mathsf{K}, \star, \mathsf{t}) = \mathsf{t}\ \mathsf{K}}$$

M3
$$\frac{}{\mathsf{meet}(\mathsf{P}, \mathsf{K}, \mathsf{t}, \mathsf{t}) = \mathsf{t}\ \mathsf{K}}$$

M4
$$\frac{\begin{array}{c}\mathsf{E}\ \text{fresh} \qquad (\mathsf{C}, \mathsf{D}) \notin \text{dom}(\mathsf{P}) \\ \mathsf{P}' = \mathsf{P},\ (\mathsf{C}, \mathsf{D}) \mapsto \mathsf{E} \qquad \text{mtypes}(\mathsf{C}, \mathsf{K}) = \{\overline{\mathsf{mt}}\} \qquad \text{mtypes}(\mathsf{D}, \mathsf{K}) = \{\overline{\mathsf{mt}'}\} \\ \mathsf{meet}(\mathsf{P}', \mathsf{K}, \{\overline{\mathsf{mt}}\}, \{\overline{\mathsf{mt}'}\}) = \mathsf{t}''\ \mathsf{K}' \qquad \mathsf{K}'' = \mathsf{K}'\ \mathsf{classGen}(\mathsf{C}, \mathsf{t}'', \mathsf{E}, \mathsf{K}')\end{array}}{\mathsf{meet}(\mathsf{P}, \mathsf{K}, \mathsf{C}, \mathsf{D}) = \mathsf{E}\ \mathsf{K}''}$$

M5
$$\frac{\mathsf{P}(\mathsf{C}, \mathsf{D}) = \mathsf{E}}{\mathsf{meet}(\mathsf{P}, \mathsf{K}, \mathsf{C}, \mathsf{D}) = \mathsf{E}\ \mathsf{K}}$$

The meet functions takes four arguments, an environment P, a class table K, the original type t, the cast type t', and outputs a type t'' and a class table K'. The environment P is a set of mappings from a pair of types $(\mathsf{t}, \mathsf{t}')$ to a type t''.

$$\boxed{\mathsf{meet}(\mathsf{P}, \mathsf{K}, \{\overline{\mathsf{mt}}\}, \{\overline{\mathsf{mt}}\}) = \{\overline{\mathsf{mt}}\}\ \mathsf{K}}$$

M6
$$\frac{}{\mathsf{meet}(\mathsf{P}, \mathsf{K}, \{\overline{\mathsf{mt}}\}, \{\}) = \{\overline{\mathsf{mt}}\}\ \mathsf{K}}$$

M7
$$\frac{\begin{array}{c}\mathsf{meet}(\mathsf{P}, \mathsf{K}, \mathsf{t}_3, \mathsf{t}_1) = \mathsf{t}_5\ \mathsf{K}' \\ \mathsf{meet}(\mathsf{P}, \mathsf{K}', \mathsf{t}_2, \mathsf{t}_4) = \mathsf{t}_6\ \mathsf{K}'' \qquad \mathsf{meet}(\mathsf{P}, \mathsf{K}'', \{\overline{\mathsf{mt}_1}\}, \{\overline{\mathsf{mt}_2}\}) = \{\overline{\mathsf{mt}_3}\}\ \mathsf{K}'''\end{array}}{\mathsf{meet}(\mathsf{P}, \mathsf{K}, \{\mathsf{m}(\mathsf{t}_1)\colon \mathsf{t}_2\ \overline{\mathsf{mt}_1}\}, \{\mathsf{m}(\mathsf{t}_3)\colon \mathsf{t}_4\ \overline{\mathsf{mt}_2}\}) = \{\mathsf{m}(\mathsf{t}_5)\colon \mathsf{t}_6\ \overline{\mathsf{mt}_3}\}\ \mathsf{K}'''}$$

M8
$$\frac{\begin{array}{c}\mathsf{meet}(\mathsf{P}, \mathsf{K}, \mathsf{t}_3, \mathsf{t}_1) = \mathsf{t}_5\ \mathsf{K}' \\ \mathsf{meet}(\mathsf{P}, \mathsf{K}', \mathsf{t}_2, \mathsf{t}_4) = \mathsf{t}_6\ \mathsf{K}'' \qquad \mathsf{meet}(\mathsf{P}, \mathsf{K}'', \{\overline{\mathsf{mt}_1}\}, \{\overline{\mathsf{mt}_2}\}) = \{\overline{\mathsf{mt}_3}\}\ \mathsf{K}'''\end{array}}{\mathsf{meet}(\mathsf{P}, \mathsf{K}, \{\mathsf{f}(\mathsf{t}_1)\colon \mathsf{t}_2\ \overline{\mathsf{mt}_1}\}, \{\mathsf{f}(\mathsf{t}_3)\colon \mathsf{t}_4\ \overline{\mathsf{mt}_2}\}) = \{\mathsf{f}(\mathsf{t}_5)\colon \mathsf{t}_6\ \overline{\mathsf{mt}_3}\}\ \mathsf{K}'''}$$

M9
$$\frac{\mathsf{meet}(\mathsf{P}, \mathsf{K}, \mathsf{t}_2, \mathsf{t}_4) = \mathsf{t}_6\ \mathsf{K}' \qquad \mathsf{meet}(\mathsf{P}, \mathsf{K}', \{\overline{\mathsf{mt}_1}\}, \{\overline{\mathsf{mt}_2}\}) = \{\overline{\mathsf{mt}_3}\}\ \mathsf{K}''}{\mathsf{meet}(\mathsf{P}, \mathsf{K}, \{\mathsf{f}()\colon \mathsf{t}_2\ \overline{\mathsf{mt}_1}\}, \{\mathsf{f}()\colon \mathsf{t}_4\ \overline{\mathsf{mt}_2}\}) = \{\mathsf{f}()\colon \mathsf{t}_6\ \overline{\mathsf{mt}_3}\}\ \mathsf{K}''}$$

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

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

```
$\mathsf{classGen}(\mathsf{C}, \mathsf{C}', \mathsf{D}, \mathsf{K}) =$
        class D {
            $\mathsf{f} : \mathsf{t}_1'$
              *for every* $\mathsf{f} : \mathsf{t}_1 \in \mathsf{field}(\mathsf{C}, \mathsf{K}) \wedge \mathsf{f} : \mathsf{t}_1' \in \mathsf{ftype}(\mathsf{f}, \mathsf{C}', \mathsf{K})$

            $\mathsf{n}(\overline{\mathsf{x} : \star}) : \star\ \{\triangleleft\ \star \triangleright \mathsf{this.n}(\overline{\triangleleft\, \mathsf{t}_1 \triangleright \mathsf{x}})\}$
              *for every* $\mathsf{n}(\overline{\mathsf{t}_1}) : \mathsf{t}_1' \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K}) \wedge \mathsf{t}_1 \neq \star$

            $\mathsf{n}(\overline{\mathsf{x} : \mathsf{t}_1}) : \mathsf{t}_1'\ \{\triangleleft\, \mathsf{t}_1' \triangleright \mathsf{e}\}$
              *for every* $\mathsf{n}(\overline{\mathsf{t}_1}) : \mathsf{t}_1' \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K}) \wedge \mathsf{n}(\overline{\mathsf{x} : \mathsf{t}_2}) : \mathsf{t}_2'\ \{\mathsf{e}\} \in \mathsf{methods}(\mathsf{C}, \mathsf{K}) \wedge \mathsf{t}_1 \neq \star$

            $\mathsf{n}(\overline{\mathsf{x} : \star}) : \star\ \{\mathsf{e}\}$
              *for every* $\mathsf{n}(\overline{\star}) : \star \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K}) \wedge \mathsf{n}(\overline{\mathsf{x} : \star}) : \star\ \{\mathsf{e}\} \in \mathsf{methods}(\mathsf{C}, \mathsf{K})$
        }
```