Translating Functional Programs to Java

Version 3

Alessandro Coglio

August 20, 2003

This document formally defines:

1. the abstract syntax and static semantics (i.e. typing) of a higher-order, non-polymorphic functional programming language with product, sum, restriction, and quotient types and with pattern matching for sum types, called *Fun*;
2. the abstract syntax of (a subset of) Java;
3. a translation from *Fun* programs to Java programs.

In the future, this formalization should be extended with static semantics of Java and dynamic semantics (i.e. execution) of *Fun* and Java, along with a proof that translating a *Fun* program yields a Java program that is equivalent, in some sense to be made precise, to the original *Fun* program.

1 The language *Fun*

1.1 Names

The definition of *Fun* is parameterized over a set of names

$$\mathcal{N}$$

This parameterization is not just for the sake of abstraction; it is exploited to factor the concrete name translation to Java.

1.2 Types

A *Fun* program has a finite set of user-defined types

$$Ty_{\mathrm{U}} \subseteq_{\mathrm{f}} \mathcal{N}$$

It also has built-in types

$$Ty_{\mathrm{B}} = \{\mathsf{Bool}, \mathsf{Int}, \mathsf{Char}\}$$

for booleans, integers, and characters. An arrow type consists of one or more argument types and one result type (types are defined below)[1]

$$Ty_{\mathrm{A}} = \{\overline{ty} \to ty \mid \overline{ty} \in Ty^+ \;\wedge\; ty \in Ty\}$$

The types of the program are[2]

$$Ty = Ty_{\mathrm{U}} \uplus Ty_{\mathrm{B}} \uplus Ty_{\mathrm{A}}$$

We define type products, sums, restrictions, and quotients as

$$\begin{aligned}
\mathit{TyProd} &= \{p_1\ ty_1 \times \cdots \times p_n\ ty_n \mid \overline{ty} \in Ty^* \;\wedge\; \overline{p} \in \mathcal{N}^{(*)}\} \\
\mathit{TySum} &= \{c_1\ \overline{ty}_1 + \cdots + c_n\ \overline{ty}_n \mid \overline{\overline{ty}} \in (Ty^*)^+ \;\wedge\; \overline{c} \in \mathcal{N}^{(+)}\} \\
\mathit{TyRestr} &= \{ty_0|r \mid ty_0 \in Ty \;\wedge\; r \in Op_{\mathrm{U}} \;\wedge\; \tau(r) = ty_0 \to \mathsf{Bool}\} \\
\mathit{TyQuot} &= \{ty_0/q \mid ty_0 \in Ty \;\wedge\; q \in Op_{\mathrm{U}} \;\wedge\; \tau(q) = ty_0, ty_0 \to \mathsf{Bool}\}
\end{aligned}$$

(Op_{U} and τ are defined below). A product consists of zero or more factors, each factor consisting of a projector p_i and a type ty_i; projectors must be distinct. A sum consists of one or more summands, each summand consisting of a constructor c_i and zero or more argument types $\overline{ty}_i$; constructors must be distinct. A restriction consists of a type and a unary predicate over that type. A quotient consists of a type and a binary predicate over that type, which should be an equivalence relation (however, we do not enforce this in our definition of *Fun*).

Each user-defined type has a definition that is a type product, sum, restriction, or quotient

$$\Delta : Ty_{\mathrm{U}} \to \mathit{TyProd} \cup \mathit{TySum} \cup \mathit{TyRestr} \cup \mathit{TyQuot}$$

If $\Delta(ty) \in \mathit{TyProd}$ (resp. TySum, $\mathit{TyRestr}$, TyQuot), ty is called a product (resp. sum, restriction, quotient) type.

1.3 Operations

A *Fun* program has a finite set of user-defined op(eration)s

$$Op_{\mathrm{U}} \subseteq_{\mathrm{f}} \mathcal{N}$$

It also has built-in ops

$$\begin{aligned}
Op_{\mathrm{B}} = {} & \{\mathsf{true}, \mathsf{false}, \mathsf{not}, \mathsf{and}, \mathsf{or}\} \\
& \uplus \{\iota \in \mathbf{Z} \mid -2^{31} \le \iota < 2^{31}\} \\
& \uplus \{\mathsf{minus}, +, -, *, /, \mathsf{mod}\} \\
& \uplus \{<, \le, >, \ge\} \\
& \uplus \{\xi \in \mathbf{N} \mid \xi < 2^{16}\} \\
& \uplus \{\mathsf{c2i}, \mathsf{i2c}\}
\end{aligned}$$

[1] **Notation.** Given a set X: X^* is the set of all finite sequences of elements of X; $X^{(*)}$ is the set of all sequences in X^* whose elements are all distinct; X^+ is the set of all non-empty sequences in X^*; and $X^{(+)}$ is the set of all sequences in X^+ whose elements are all distinct.

[2] **Notation.** The symbol $\uplus$ denotes disjoint union.

for boolean values and connectives, two's complement 32-bit integers, basic arithmetic of integers, comparison of integers, 16-bit Unicode characters, and conversions between characters and integers. Furthermore, projectors and constructors are lifted to ops

$$
\begin{aligned}
Op_{\mathrm{P}} &= \biguplus_{\Delta(ty)=(\prod_i p_i\ ty_i)} \overline{p} \\
Op_{\mathrm{C}} &= \biguplus_{\Delta(ty)=(\sum_i c_i\ \overline{ty}_i)} \overline{c}
\end{aligned}
$$

Finally, each restriction (resp. quotient) type is accompanied by a restrictor and a relaxator (resp. a quotienter and a chooser) that map values between the restricted (resp. quotiented) type and the restriction (resp. quotient) type

$$
\begin{aligned}
Op_{\mathrm{R}} &= \{\mathsf{restr}_{ty} \mid \Delta(ty) \in \mathit{TyRestr}\} \\
&\cup \{\mathsf{relax}_{ty} \mid \Delta(ty) \in \mathit{TyRestr}\} \\
Op_{\mathrm{Q}} &= \{\mathsf{quot}_{ty} \mid \Delta(ty) \in \mathit{TyQuot}\} \\
&\cup \{\mathsf{choo}_{ty} \mid \Delta(ty) \in \mathit{TyQuot}\}
\end{aligned}
$$

The ops of the program are

$$Op = Op_{\mathrm{U}} \uplus Op_{\mathrm{B}} \uplus Op_{\mathrm{P}} \uplus Op_{\mathrm{C}} \uplus Op_{\mathrm{R}} \uplus Op_{\mathrm{Q}}$$

Each op has a type

$$\tau : Op \to Ty$$

Constants are ops that take no arguments

$$COp = \{op \in Op \mid \tau(op) \notin Ty_{\mathrm{A}}\}$$

The built-in ops have types

$$
\begin{aligned}
\tau(\mathsf{true}) = \tau(\mathsf{false}) &= \mathsf{Bool} \\
\tau(\mathsf{not}) &= \mathsf{Bool} \to \mathsf{Bool} \\
\tau(\mathsf{and}) = \tau(\mathsf{or}) &= \mathsf{Bool}, \mathsf{Bool} \to \mathsf{Bool} \\
\tau(\iota) &= \mathsf{Int} \\
\tau(\mathsf{minus}) &= \mathsf{Int} \to \mathsf{Int} \\
\tau(+) = \tau(-) = \tau(*) = \tau(/) = \tau(\mathsf{mod}) &= \mathsf{Int}, \mathsf{Int} \to \mathsf{Int} \\
\tau(<) = \tau(\leq) = \tau(>) = \tau(\geq) &= \mathsf{Int}, \mathsf{Int} \to \mathsf{Bool} \\
\tau(\xi) &= \mathsf{Char} \\
\tau(\mathsf{c2i}) &= \mathsf{Char} \to \mathsf{Int} \\
\tau(\mathsf{i2c}) &= \mathsf{Int} \to \mathsf{Char}
\end{aligned}
$$

Projectors, constructors, restrictors, relaxators, quotienters, and choosers have types[3]

$$
\begin{aligned}
\Delta(ty) = \textstyle\prod_i p_i\ ty_i &\;\Rightarrow\; \tau(p_i) = ty \to ty_i \\
\Delta(ty) = \textstyle\sum_i c_i\ \overline{ty}_i \;\wedge\; \overline{ty}_i \neq \epsilon &\;\Rightarrow\; \tau(c_i) = \overline{ty}_i \to ty \\
\Delta(ty) = \textstyle\sum_i c_i\ \overline{ty}_i \;\wedge\; \overline{ty}_i = \epsilon &\;\Rightarrow\; \tau(c_i) = ty \\
\Delta(ty) = ty_0|r &\;\Rightarrow\; \tau(\mathsf{restr}_{ty}) = ty_0 \to ty \\
\Delta(ty) = ty_0|r &\;\Rightarrow\; \tau(\mathsf{relax}_{ty}) = ty \to ty_0 \\
\Delta(ty) = ty_0/q &\;\Rightarrow\; \tau(\mathsf{quot}_{ty}) = ty_0 \to ty \\
\Delta(ty) = ty_0/q &\;\Rightarrow\; \tau(\mathsf{choo}_{ty}) = ty \to ty_0
\end{aligned}
$$

[3] **Notation.** ϵ is the empty sequence.

1.4 Terms

A variable is a name

$$V = \mathcal{N}$$

A context associates types to a finite number of variables

$$Cx = V \xrightarrow{\mathrm{f}} Ty$$

The family $\{T^{cx}_{ty}\}_{cx \in Cx, ty \in Ty}$ of sets of terms, indexed by contexts and types, is defined as[4]

$$\frac{cx(v) = ty}{v \in T^{cx}_{ty}} \quad \text{(variable)}$$

$$\frac{op \in Op}{op \in T^{cx}_{\tau(op)}} \quad \text{(op)}$$

$$\frac{\begin{array}{c} t \in T^{cx}_{\overline{ty} \to ty} \\ \forall i.\;\; t_i \in T^{cx}_{ty_i} \end{array}}{t(\overline{t}) \in T^{cx}_{ty}} \quad \text{(application)}$$

$$\frac{\begin{array}{c} \overline{v} \in V^{(+)} \\ \overline{ty} \in Ty^{+} \\ ty \in Ty \\ t \in T^{cx[\overline{v} \mapsto \overline{ty}]}_{ty} \end{array}}{\lambda \overline{v} : \overline{ty}.t : ty \in T^{cx}_{\overline{ty} \to ty}} \quad \text{(abstraction)}$$

$$\frac{\begin{array}{c} \Delta(ty) = \prod_i p_i \;\; ty_i \\ \forall i.\;\; t_i \in T^{cx}_{ty_i} \end{array}}{\{p_1 \leftarrow t_1, \ldots, p_n \leftarrow t_n\} \in T^{cx}_{ty}} \quad \text{(tuple)}$$

$$\frac{t_1, t_2 \in T^{cx}_{ty}}{(t_1 = t_2) \in T^{cx}_{\mathsf{Bool}}} \quad \text{(equality)}$$

$$\frac{t_1, t_2 \in T^{cx}_{ty}}{(t_1 \neq t_2) \in T^{cx}_{\mathsf{Bool}}} \quad \text{(inequality)}$$

[4]**Notation.** If f is a function, $f[x \mapsto y]$ is the function f' with domain $\mathcal{D}(f') = \mathcal{D}(f) \cup \{x\}$ such that $f'(x) = y$ and $f'(x') = f(x')$ for all $x' \neq x$; either $x \in \mathcal{D}(f)$ (in which case the value of the function at x is overridden to be y) or $x \notin \mathcal{D}(f)$ (in which case the function is extended to have value y at x).

$$\frac{\begin{array}{c} t_0 \in T^{cx}_{\mathsf{Bool}} \\ t_1, t_2 \in T^{cx}_{ty} \end{array}}{(\textbf{if}\ t_0\ t_1\ t_2) \in T^{cx}_{ty}} \quad \text{(conditional)}$$

$$\frac{\begin{array}{c} v \in V \\ t_0 \in T^{cx}_{ty_0} \\ t \in T^{cx[v \mapsto ty_0]}_{ty} \end{array}}{(\textbf{let}\ v \leftarrow t_0\ \textbf{in}\ t) \in T^{cx}_{ty}} \quad \text{(let binding)}$$

$$\frac{\begin{array}{c} \Delta(ty) = \sum_i c_i\ \overline{ty}_i \\ t \in T^{cx}_{ty} \\ \{i_1, \ldots, i_p\} \subseteq \{1, \ldots, n\} \\ \forall i.\ \ t_i \in T^{cx[\overline{v}_i \mapsto \overline{ty}_i]}_{ty_0} \end{array}}{(\textbf{case}\ t\ \{c_{i_1}(\overline{v}_{i_1}) \rightarrow t_{i_1}, \ldots, c_{i_p}(\overline{v}_{i_p}) \rightarrow t_{i_p}\}) \in T^{cx}_{ty_0}} \quad \text{(pattern matching)}$$

The set of all terms is $T = \bigcup_{cx \in Cx, ty \in Ty} T^{cx}_{ty}$.

It is understood that the rules above implicitly keep terms distinct whenever needed (by implicitly tagging them, as in disjoint unions): for example, a term that is a variable $v \in V = \mathcal{N}$ is considered distinct from a term that is an op $op \in Op_{\mathrm{U}} \subseteq \mathcal{N}$ even if $v = op$, i.e. even if they are the same name; as another example, a tuple of a type is considered distinct from a tuple of another type that happens to consist of the same factors in the same order. For readability, we may omit the explicit types from abstractions. Abstractions, **let**, and **case** introduce new variables into the contexts of some of their subterms; the newly introduced variables may shadow variables from the outer context. A **case** may have branches for only a subset of the sum type's summands.

The function[5] $FV : T \rightarrow \mathcal{P}_\omega(V)$ collects the free variables of a term

$$\begin{aligned}
FV(v) &= \{v\} \\
FV(op) &= \emptyset \\
FV(t(\overline{t})) &= FV(t) \cup \textstyle\bigcup_i FV(t_i) \\
FV(\lambda \overline{v}.t) &= FV(t) - \overline{v} \\
FV(\{p_i \leftarrow t_i\}_i) &= \textstyle\bigcup_i FV(t_i) \\
FV(t_1 = t_2) = FV(t_1 \neq t_2) &= FV(t_1) \cup FV(t_2) \\
FV(\textbf{if}\ t_0\ t_1\ t_2) &= FV(t_0) \cup FV(t_1) \cup FV(t_2) \\
FV(\textbf{let}\ v \leftarrow t_0\ \textbf{in}\ t) &= FV(t_0) \cup (FV(t) - \{v\}) \\
FV(\textbf{case}\ t\ \{c_i(\overline{v}_i) \rightarrow t_i\}_i) &= FV(t) \cup \textstyle\bigcup_i (FV(t_i) - \overline{v}_i)
\end{aligned}$$

We define the substitution of the free occurrences of a variable v with a

[5]**Notation.** Given a set X, $\mathcal{P}_\omega(X) = \{\widetilde{x} \mid \widetilde{x} \subseteq_{\mathrm{f}} X\}$, i.e. the set of all finite subsets of X.

variable v' in a term as

$$
\begin{aligned}
v[v'/v] &= v' \\
w \neq v \;\Rightarrow\; w[v'/v] &= w \\
op[v'/v] &= op \\
t(\bar{t})[v'/v] &= t[v'/v](\bar{t}[v'/v]) \\
v \in \overline{v} \;\Rightarrow\; \lambda\overline{v}.t[v'/v] &= \lambda\overline{v}.t \\
v \notin \overline{v} \;\Rightarrow\; \lambda\overline{v}.t[v'/v] &= \lambda\overline{v}.(t[v'/v]) \\
\{p_i \leftarrow t_i\}_i[v'/v] &= \{p_i \leftarrow t_i[v'/v]\}_i \\
(t_1 = t_2)[v'/v] &= (t_1[v'/v] = t_2[v'/v]) \\
(t_1 \neq t_2)[v'/v] &= (t_1[v'/v] \neq t_2[v'/v]) \\
(\mathbf{if}\ t_0\ t_1\ t_2)[v'/v] &= (\mathbf{if}\ t_0[v'/v]\ t_1[v'/v]\ t_2[v'/v]) \\
(\mathbf{let}\ v \leftarrow t_0\ \mathbf{in}\ t)[v'/v] &= (\mathbf{let}\ v \leftarrow t_0[v'/v]\ \mathbf{in}\ t) \\
w \neq v \;\Rightarrow\; (\mathbf{let}\ w \leftarrow t_0\ \mathbf{in}\ t)[v'/v] &= (\mathbf{let}\ w \leftarrow t_0[v'/v]\ \mathbf{in}\ t[v'/v]) \\
\left(\forall i.\ \ t'_i = \begin{cases} t_i[v'/v] & \text{if} \quad v \notin \overline{v}_i \\ t_i & \text{otherwise} \end{cases}\right) \;\Rightarrow\; & \\
(\mathbf{case}\ t\ \{c_i(\overline{v}_i) \to t_i\}_i)[v'/v] &= (\mathbf{case}\ t[v'/v]\ \{c_i(\overline{v}_i) \to t'_i\}_i)
\end{aligned}
$$

Care must be exercised, when doing substitutions, to prevent variable overloading (e.g. $(\mathbf{if}\ v\ 0\ v')[v'/v]$) and capture (e.g. $(\mathbf{let}\ v' \leftarrow t\ \mathbf{in}\ v)[v'/v]$).

1.5 Op definitions

A non-constant user-defined op has (formal) parameters that are distinct variables

$$\pi : (Op_{\mathrm{U}} - COp) \to V^{(+)}$$

A non-constant user-defined op also has a restriction term

$$\rho : (Op_{\mathrm{U}} - COp) \to T$$

such that

$$\tau(op) = \overline{ty} \to ty \;\Rightarrow\; \rho(op) \in T_{\mathsf{Bool}}^{\{\pi(op) \mapsto \overline{ty}\}}$$

i.e. the term has type Bool in the context that associates the op's argument types to the op's parameters. The term constitutes a predicate on the op's parameters that implicitly defines a restriction type of the product of the op's argument types: the op is only defined on this implicit restriction type. In other words, ρ captures the domain of ops that are not defined over all the tuples of the product of the argument types but only over some of them, as is common. The case where an op is defined over all the tuples is covered by $\rho(op) = \mathsf{true}$.

A user-defined op also has a defining term

$$\delta : Op_{\mathrm{U}} \to T$$

such that

$$
\begin{aligned}
op \in COp \;&\Rightarrow\; \delta(op) \in T_{ty}^{\vec{\emptyset}} \\
\tau(op) = \overline{ty} \to ty \;&\Rightarrow\; \delta(op) \in T_{ty}^{\{\pi(op) \mapsto \overline{ty}\}}
\end{aligned}
$$

i.e. the defining term has the op's result type in the context that associates the op's argument types to the op's parameters (if any).

1.6 Program

The program is the 7-tuple

$$\mathcal{P} = \langle Ty_{\mathrm{U}}, \Delta, Op_{\mathrm{U}}, \tau, \pi, \rho, \delta \rangle$$

2 The translation, informally

2.1 Types

The built-in types Bool, Int, and Char of *Fun* translate to the primitive types `boolean`, `int`, and `char` of Java.

A product type translates to a class with an instance field for each factor. For example, $\Delta(\mathsf{P}) = \mathsf{a}\ \mathsf{Int} \times \mathsf{b}\ \mathsf{U}$ translates to

```
class P {
    int a;
    U b;
    ...
}
```

A sum type translates to an abstract class, accompanied by a non-abstract subclass for each summand; each subclass has an instance field for each argument of the corresponding constructor. Each instance of these subclasses carries a numeric tag that identifies the subclass; the tag is stored in an instance field of the abstract superclass, which also includes static fields that give names to the possible tag values for increased readability. For example, $\Delta(\mathsf{S}) = \mathsf{c}\ (\mathsf{Bool}, \mathsf{U}) + \mathsf{d}$ translates to

```
abstract class S {
    static int TAG_c = 1;
    static int TAG_d = 2;
    int tag;
    ...
}

class S_c extends S {
    boolean arg1;
    U arg2;
    ...
}

class S_d extends S {
    ...
}
```

This also works for recursive types, e.g. $\Delta(\mathsf{List}) = \mathsf{nil} + \mathsf{cons}\ (\mathsf{Int}, \mathsf{List})$ (lists of integers) naturally translates to

```
abstract class List {
    static int TAG_nil = 1;
    static int TAG_cons = 2;
    int tag;
    ...
}

class List_nil extends List {
    ...
}

class List_cons extends List {
    int arg1;
    List arg2;
    ...
}
```

A restriction type translates to a class that encapsulates a value of the restricted type in an instance field. For example, $\Delta(\mathsf{R}) = \mathsf{U}|\mathsf{r}$ translates to

```
class R {
    U relax;
    ...
}
```

While translating type restrictions to subclasses (e.g. `class R extends U`) may seem an elegant and viable approach, there are difficulties in restricting built-in types because they do not translate to classes but to primitive types and there are difficulties in restricting sum types because their subclass structure may interfere with the one for the restriction (e.g. the restriction type of lists of length at most 3 should translate to a subclass of both `List_nil` and `List_cons`, but Java does not support multiple inheritance).

A quotient type translates to a class that encapsulates a value of the quotiented type (i.e. a member of the equivalence class that constitutes the quotient value) in an instance field. For example, $\Delta(\mathsf{Q}) = \mathsf{Int}/\mathsf{q}$ translates to

```
class Q {
    int choose;
    ...
}
```

An arrow type translates to an abstract class; instances of its subclasses realize abstractions, as explained later. For example, $(\mathsf{Int}, \mathsf{Int} \rightarrow \mathsf{Int})$, $(\mathsf{U} \rightarrow \mathsf{Bool}) \rightarrow \mathsf{Char}$, and $\mathsf{Int} \rightarrow (\mathsf{Int} \rightarrow \mathsf{Int})$ translate to

```
abstract class IntIntToInt {
    ...
}
```

```
abstract class FromUToBoolToChar {
    ...
}

abstract class IntToFromIntToInt {
    ...
}
```

In the second and third class name, `From` and `To` play the role of the opening and closing parentheses in $(\mathsf{U} \to \mathsf{Bool}) \to \mathsf{Char}$, and $\mathsf{Int} \to (\mathsf{Int} \to \mathsf{Int})$; the exact naming of arrow classes is described in Section 4.5.3.

2.2 Ops

Most built-in ops of *Fun* translate to the obvious literals and operators of Java. The conversion op $\mathsf{i2c}$ from integers to characters translates to a narrowing cast to `char`. The conversion op $\mathsf{c2i}$ from characters to integers translates to a widening cast to `int`. Usually, characters are automatically widened to integers, making the widening cast unnecessary (i.e. $\mathsf{c2i}$ could translate to nothing). However, in the presence of overloaded methods that only differ by `int` vs. `char` argument types, explicit widening casts are needed, otherwise the wrong method would be invoked. So, we uniformly use widening casts; this may be refined, in the future, to only put the necessary widening casts.

User-defined ops translate to fields if they are constants, to methods otherwise.

We consider a non-constant user-defined op first. If the op has a user-defined argument type, the op translates to an instance method of the class for that user-defined type; in the presence of multiple user-defined argument types, we choose the first (i.e. leftmost) one. The remaining arguments become the parameters of the method. For example, $\tau(\mathsf{m1}) = \mathsf{Int}, \mathsf{U}, \mathsf{V} \to \mathsf{Bool}$ and $\pi(\mathsf{m1}) = (\mathsf{i}, \mathsf{u}, \mathsf{v})$ translate to[6]

```
boolean U.m1(int i, V v)
```

If the op has no user-defined argument types but its result type is user-defined, the op translates to a static method of the class for that user-defined type, e.g. $\tau(\mathsf{m2}) = \mathsf{Int} \to \mathsf{U}$ and $\pi(\mathsf{m2}) = \mathsf{i}$ translate to

```
static U U.m2(int i)
```

If the op has no user-defined argument or result type but has an arrow argument or result type that includes a user-defined type, the op translates to a static method of the class for that user-defined type; in the presence of multiple user-defined types in arrow argument or result types, we choose the leftmost one. For example, $\tau(\mathsf{m3}) = (\mathsf{U} \to \mathsf{V}) \to \mathsf{Int}$ and $\pi(\mathsf{m3}) = \mathsf{uv}$ translate to

[6]The dotted notation is not valid Java syntax; we use it just to concisely indicate in which classes methods and fields are declared.

```
static int U.m3(UToV uv)
```

If no user-defined type occurs in the op's argument and result types, the op translates to a static method of a class used as a receptacle of all methods and fields resulting from ops that only involve built-in types (i.e primitive types in Java)

```
class Prim {
    ....
}
```

For example, $\tau(\mathsf{m4}) = \mathsf{Int}, (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$ and $\pi(\mathsf{m4}) = (\mathsf{i}, \mathsf{f})$ translate to

```
static int Prim.m4(int i, IntToInt f)
```

Now we consider a user-defined constant. If the constant's type is user-defined, the field is declared in the corresponding class, e.g. $\tau(\mathsf{n1}) = \mathsf{U}$ translates to

```
static U U.n1
```

If the constant's type is built-in, the field is declared in the special receptacle class mentioned above, e.g. $\tau(\mathsf{n2}) = \mathsf{Bool}$ translates to

```
static boolean Prim.n2
```

Projectors translate to the fields of the corresponding product class, described earlier.

Constructors translate to static fields and methods declared in the corresponding sum class. The initializers of these fields and the bodies of these methods invoke Java constructors declared in the summand classes; these Java constructors have the same arguments as the corresponding *Fun* constructors and assign the arguments to the fields and set the numeric tag. For example, we have

```
static S S.c(boolean arg1, U arg2) {
    return (new S_c(arg1,arg2));
}

static S S.d = new S_d();

S_c(boolean arg1, U arg2) {
    tag = S.TAG_c;
    this.arg1 = arg1;
    this.arg2 = arg2;
}

S_d() {
    tag = S.TAG_d;
}
```

Relaxators and choosers translate to the fields of the corresponding restriction and quotient classes, as indicated by the names of those fields. Restrictors and quotienters do not translate to any field or method; as explained below, their application translates to class instance creation expressions.

2.3 Terms

2.3.1 Variable

Fun variables normally translate to Java method parameters and local variables. An exception is for ops that translate to instance methods: the op's parameter whose (user-defined) type corresponds to the class in which the method is declared, translates to `this`. Two other exceptions are described later.

2.3.2 Constant

A built-in constant translates to the corresponding Java literal, e.g. true translates to `true` and the integer 27 translates to `27`.

A user-defined constant translates to an access of the corresponding field, e.g. n1 translates to `U.n1` and n2 translates to `Prim.n2`.

2.3.3 Op application

The application of a built-in op translates to an expression involving the corresponding Java operator or cast to `char` (for i2c) or `int` (for c2i).

The application of a non-built-in op that is not a restrictor or quotienter translates to an access to the corresponding field (if the op is a constant, projector, relaxator, or chooser) or to a call to the corresponding method (otherwise). The application of a restrictor (resp. quotienter) translates to a class instance creation expression of the corresponding restriction (resp. quotient) class.

For example, x+y translates to `x+y`, m1(i, u, v) translates to `u.m1(i,v)`, a(p) translates to `p.a`, c(true, u) translates to `S.c(true,u)`, $\mathsf{restr}_{\mathsf{R}}(\mathsf{u})$ translates to `new R(u)`, and $\mathsf{choo}_{\mathsf{Q}}(\mathsf{q})$ translates to `q.choose`.

2.3.4 Abstraction

An abstraction translates to a class instance creation expression of an anonymous subclass of the arrow class derived from the abstraction's type. Each abstract class derived from an arrow type has an abstract `apply` method whose argument and result types match the arrow type's, e.g.

```
abstract int IntIntToInt.apply(int arg1, int arg2);

abstract char FromUToBoolToChar.apply(UToBool arg1);

abstract IntToInt IntToFromIntToInt.apply(int arg1);
```

The anonymous class implements the `apply` method with code derived from the abstraction, e.g. $\lambda \mathsf{i}, \mathsf{j}.(\mathsf{i} * (\mathsf{i} + \mathsf{j}))$ translates to

```
new IntIntToInt() {
    int apply(int i, int j) {
        return i * (i + j);
    }
}
```

If the abstraction contains free variables, they can be almost simply referenced in the `apply` method of the anonymous class. “Almost” because in Java anonymous classes are only allowed to access final variables from the outer scope. So, the content of the free variables must be copied into final variables before executing the anonymous class instance creation expression. For example, $\lambda \mathsf{i}.(\mathsf{i} - \mathsf{j})$ translates to

```
new IntIntToInt() {
    int apply(int i) {
        return i - final_j;
    }
}
```

preceded by `final int final_j = j;`. As shown below, also other *Fun* constructs translate to expressions preceded by “preparatory” statements.

2.3.5 Non-constant op

In *Fun*, a non-constant op is a term by itself. If it is applied to some arguments, we translate the application to a method call, as desribed above. Otherwise, e.g. if it is assigned to a **let** variable, we need to create an expression that evaluates to the op. Since in Java methods are not values, we wrap the op in a lambda abstraction, i.e. we regard *op* as $\lambda \overline{v}.op(\overline{v})$, where $\overline{v} = \pi(op)$. For instance, `m1` translates to

```
new IntUVToBool() {
    int apply(int i, U u, V v) {
        return u.m1(i,v);
    }
}
```

as if it were $\lambda \mathsf{i}, \mathsf{u}, \mathsf{v}.\mathtt{m1}(\mathsf{i}, \mathsf{u}, \mathsf{v})$.

2.3.6 Non-op application

The application of a term that is not an op to one or more terms translates to a call of the `apply` method, e.g. if f is a variable of type $\mathsf{Int}, \mathsf{Int} \rightarrow \mathsf{Int}$ and i and j are variables of type Int, the application $\mathsf{f}(\mathsf{i}, \mathsf{j})$ translates to `f.apply(i,j)`.

It would be correct to uniformly translate an application as a call to `apply` whether the term that is applied is an op or not, so that an op would uniformly translate to an anonymous class whether it is applied or not. However, such a translation would produce code that is much less readable and efficient than our "selective" translation.

2.3.7 Tuple

A tuple translates to a class instance creation expression of the corresponding product class, which has a constructor with one argument for each factor and which assigns the arguments to the fields. For example, we have

```
P(int a, U b) {
    this.a = a;
    this.b = b;
}
```

and the tuple $\{\mathtt{a} \leftarrow 2, \mathtt{b} \leftarrow \mathtt{u}\}$ translates to `new P(2,u)`.

2.3.8 Equality

An equality between terms with built-in type translates to a Java equality expression that uses the `==` operator.

An equality between terms with user-defined type translates to a call of the `equals` method of the corresponding class

```
boolean U.equals(U eqarg)
```

(This method does not override the `equals` method of class `Object`, because the latter has argument type `Object`.)

The `equals` method of a product class returns the conjunction of the equalities between all the components of the product, e.g.

```
boolean P.equals(P eqarg) {
    return (this.a == eqarg.a &&
            this.b.equals(eqarg.b));
}
```

For sum classes, we take advantage of Java's dynamic dispatch. We declare an abstract `equals` method in the abstract superclass. The implementing method in each subclass checks whether the argument has the same tag as the class to which the method belongs; if so, it compares all the fields. For example, we have

```
abstract boolean S.equals(S eqarg);

boolean S_c.equals(S eqarg) {
    if (eqarg.tag == S.TAG_c) {
        S_c eqargSub = (S_c) eqarg;
        return (this.arg1 == eqargSub.arg1 &&
                this.arg2.equals(eqargSub.arg2));
```

```
    } else return false;
}

boolean S_d.equals(S eqarg) {
    return (eqarg.tag == S.TAG_d);
}
```

The `equals` method of a restriction class compares its instance fields, e.g.

```
boolean R.equals(R eqarg) {
    return (this.relax.equals(eqarg.relax));
}
```

The `equals` method of a quotient class, as expected, invokes the method that is the translation of the binary relation (which should be an equivalence) that defines the quotient type, e.g.

```
boolean Q.equals(Q eqarg) {
    return (Prim.q(this.choose,eqarg.choose));
}
```

In this example, since $\Delta(\mathsf{Q}) = \mathsf{Int}/\mathsf{q}$, i.e. the quotiented type is built-in, the binary relation `q` translates to a method in class `Prim`. If the quotiented type were user-defined, the binary relation would translate to the method for that op, which is declared in the class for the quotiented type.

An equality between terms with arrow type translates to a call of the `equals` method of the corresponding class. Such a method throws an error

```
boolean IntToInt.equals(IntToInt eqarg) {
    throw (new Error("Functions cannot be " +
                     "compared for equality"));
}
```

because the equality of two functions is not computable in general. A correct *Fun* program never attempts to evaluate an equality between functions (although this is not enforced in the definition of *Fun*), so a Java program obtained by translating a correct *Fun* program never invokes the `equals` method of an arrow class. However, the method could be invoked by external, hand-written code; so, throwing the error is useful to detect bugs in external code.

2.3.9 Inequality

An inequality between terms with built-in type translates to a Java expression that uses the `!=` operator. An inequality between terms with user-defined or arrow type translates to the logical negation of the call to the `equals` method, e.g. `u1` $\neq$ `u2` translates to `!(u1.equals(u2))`.

2.3.10 Let binding

Unlike **let**, Java expressions cannot bind variables. For this reason, **let** translates to expressions preceded by assignment statements. For example, **let** $\mathsf{x} \leftarrow 3$ **in** $\mathsf{x} + \mathsf{x}$ translates to the expression `x+x` preceded by the statement `x=3;`.

So, in general, terms translate to expressions preceded by statements. The preceding statements for a term include the preceding statements for its subterms. For example, (**let** $\mathsf{x} \leftarrow 2$ **in** $3 * \mathsf{x}$) $-$ (**let** $\mathsf{y} \leftarrow 4$ **in** $8/\mathsf{y}$) translates to `3*x-8/y` preceded by `x=2;y=4;`.

Some care is needed when different **let** terms bind the same variable. For instance, if (**let** $\mathsf{x} \leftarrow 2$ **in** $3 * \mathsf{x}$) $-$ (**let** $\mathsf{x} \leftarrow 4$ **in** $8/\mathsf{x}$) (which is perfectly legal in *Fun*: the two bindings of x do not interfere with each other) translated to `3*x-8/x` preceded by `x=2;x=4;`, the resulting Java program would clearly give incorrect results. In this case, we must use two distinct Java variables (e.g. `x1` and `x2`) for the same *Fun* variable x, i.e. the term translates to `3*x1-8/x2` preceded by `x1=2;x2=4;`.

In certain cases, it is unnecessary to use different Java variables for the same *Fun* variable. For example, **let** $\mathsf{x} \leftarrow 3$ **in let** $\mathsf{x} \leftarrow \mathsf{x}{+}1$ **in** $2{*}\mathsf{x}$ can safely translate to `2*x` preceded by `x=3;x=x+1;` and **let** $\mathsf{x} \leftarrow$ (**let** $\mathsf{x} \leftarrow 1$ **in** $\mathsf{x} + 4$) **in** $2 * \mathsf{x}$ can safely translate to `2*x` preceded by `x=1;x=x+4;`. This works because the two *Fun* variables have the same type; if **let** $\mathsf{x} \leftarrow$ (**let** $\mathsf{x} \leftarrow \mathsf{true}$ **in** (**if** x 1 2)) **in** x translated to `x` preceded by `x=true;x=(x?1:2);` (translation of conditionals is explained below), the Java program would not even compile because `x` cannot have both type `int` and `boolean`.

Before translating the *Fun* program to Java, we suitably rename **let** variables to make them all distinct within each op's defining and restriction term; the renaming is such that the information about the original name is not lost. After translating the program to Java, we revisit the renamed variables and we restore their original names whenever possible (i.e. when the Java code will still compile and behave correctly after restoring the original names).

2.3.11 Conditional

If *Fun* terms simply translated to Java expressions, conditionals in *Fun* could be translated using Java's conditional operator `?:`. However, as just described, in general expressions are preceded by statements.

Thus, Java's `if-then-else` statement must be used. Since the result must be an expression, a local variable for the result is declared just before the `if-then-else`. This variable is assigned the resulting value at the end of each branch of the `if-then-else`. For example, **if** ($\mathsf{x} = 6$) (**let** $\mathsf{y} \leftarrow 2$ **in** $\mathsf{y}{*}\mathsf{y}$) (**let** $\mathsf{z} \leftarrow 3$ **in** $\mathsf{minus}\ \mathsf{z}$) translates to `ifres` preceded by

```
int ifres;
if (x == 6) {
    y = 2;
    ifres = y*y;
} else {
```

```
        z = 3;
        ifres = -z;
    }
```

If neither branch of a conditional requires preceding statements, Java's conditional operator `?:` is used. For example, **if** $(\mathsf{x} < 4)$ $(\mathsf{x} + 3)$ 1 translates to `(x<4)?(x+3):1`. This makes the code more efficient and readable. If at least one branch requires preceding statements, even if the other branch does not, `if-then-else` must be used.

If the only use of the result variable `ifres` is to be assigned to another variable, then `ifres` is omitted and the other variable is assigned the result at the end of both branches of the `if-then-else`. For instance, instead of translating **let** $\mathsf{w} \leftarrow$ **if** $(\mathsf{x} = 6)$ (**let** $\mathsf{y} \leftarrow 2$ **in** $\mathsf{y} * \mathsf{y}$) (**let** $\mathsf{z} \leftarrow 3$ **in** $\mathsf{minus}\ \mathsf{z}$) **in** $\mathsf{w}+1$ to the expression `w+1` preceded by

```
    int ifres;
    if (x == 6) {
        y = 2;
        ifres = y*y;
    } else {
        z = 3;
        ifres = -z;
    }
    w = ifres;
```

we translate it to the expression `w+1` preceded by

```
    if (x == 6) {
        y = 2;
        w = y*y;
    } else {
        z = 3;
        w = -z;
    }
```

This makes the code more readable and efficient. This applies to nested conditionals, e.g. **let** $\mathsf{z} \leftarrow$ **if** $\mathsf{w1}$ (**if** $\mathsf{w2}$ (**let** $\mathsf{x} \leftarrow 1$ **in** $3 * \mathsf{x}$) y) $\mathsf{y}/3$ **in** $\mathsf{z} + 3$ translates to `z+3` preceded by

```
    if (w1) {
        if (w2) {
            x = 1;
            z = 3*x;
        } else {
            z = y;
        }
    } else {
        z = y/3;
    }
```

A similar omission of the result variable `ifres` takes place when the only use of the variable is to be returned by a method or to be assigned to a static field (see below).

2.3.12 Pattern matching

In certain circumstances, pattern matching is realized via Java's dynamic dispatch, analogously to equality for sum classes.

This is best understood through an example. If $\tau(\mathsf{length}) = \mathsf{List} \to \mathsf{Int}$ and $\pi(\mathsf{length}) = \mathsf{list}$,

$$\delta(\mathsf{length}) = \textbf{case}\ \mathsf{list}\ \{\mathsf{nil} \to 0, \mathsf{cons}(\mathsf{head}, \mathsf{tail}) \to 1 + \mathsf{length}(\mathsf{tail})\}$$

translates to

```
abstract int List.length();

int List_nil.length() {
    return 0;
}

int List_cons.length() {
    return (1 + this.arg2.length());
}
```

Since the type of `this.arg2` is `List`, the call `this.arg2.length()` is dynamically dispatched to the appropriate implementation of the abstract method. The variables head and tail bound by the second branch of the **case** translate to field accesses `this.arg1` and `this.arg2`; this is the second exception to the general translation of *Fun* variables to Java method parameters and local variables, mentioned earlier.

In other words, each branch of the **case** becomes a subclass method that implements the abstract superclass method. If the **case** does not have a branch for every summand, the methods for the missing summands throw an error; this is exemplified later.

The realization of pattern matching via dynamic dispatch only works if the **case** is at the top level of the op's defining term and operates on the leftmost parameter with user-defined type. Otherwise, we use Java's `switch` on the numeric tag that identifies the summand. For example, if $\tau(\mathsf{fact}) = \mathsf{List} \to \mathsf{Int}$ and $\pi(\mathsf{fact}) = \mathsf{list}$,

$$\begin{aligned}\delta(\mathsf{fact}) = \ & \textbf{let}\ \mathsf{x} \leftarrow \mathsf{length}(\mathsf{list})\ \textbf{in} \\ & (\textbf{case}\ \mathsf{list}\ \{\mathsf{nil} \to 1, \mathsf{cons}(\mathsf{head}, \mathsf{tail}) \to \mathsf{x} * \mathsf{fact}(\mathsf{tail})\})\end{aligned}$$

(which baroquely computes the factorial of the length of a list) translates to

```
int List.fact() {
    int x = this.length();
```

```
        switch (this.tag) {
            case List.TAG_nil:
                return 1;
            case List.TAG_cons:
                List_cons sub = (List_cons) this;
                return (x * sub.arg2.fact());
            default:
                throw (new Error("Malformed sum value"));
        }
    }
```

The variables head and tail bound by the second branch of the **case** translate to field accesses `sub.arg1` and `sub.arg2`, where `sub` is a variable that is assigned the value of `this` as a reference to an object of the subclass; this is the third exception to the general translation of *Fun* variables to Java method parameters and local variables, mentioned earlier.

If a **case** has a branch for each of the summands, the default case of the `switch` is unreachable, unless external code erroneously constructs an object of a sum class with a numeric tag that does not correspond to any summand. Thus, the default case in the `switch` is useful to detect bugs in external code. If a **case** does not have branches for some of its summands, the default case may be reached either because the value belongs to some missing summand or because its tag does not correspond to any summand: the default case compares the tag with the maximum tag value of the sum class and throws an error like above if the tag is larger, otherwise an error whose embedded string message indicates that the sum value was unexpected (see later for an example).

Unless (the translation of) the result of a **case** is only used to be returned by a method (as in the last example) or to be assigned to a variable or static field, a local variable `caseres` for the result is declared just before the `switch` and every branch assigns the corresponding result to it, analogously to `if-then-else`.

If the target term of a **case** that translates to a `switch` is not a variable, then the possibly expensive expression that the term translates to would have to be recomputed inside the branches in order to cast the result to the `sub` variable. To avoid this, a local variable `target` is declared just before the `switch` and initialized with the expression that would be the `switch` target. For example, if $\tau(\mathsf{f}) = \mathsf{List} \rightarrow \mathsf{Int}$ and $\pi(\mathsf{f}) = \mathsf{list}$,

$$\delta(\mathsf{f}) = \textbf{case } \mathsf{g}(\mathsf{list})\ \{\mathsf{nil} \rightarrow 0, \mathsf{cons}(\mathsf{head}, \mathsf{tail}) \rightarrow \mathsf{head} + \mathsf{f}(\mathsf{tail})\}$$

translates to

```
    int List.f() {
        List target = this.g();
        switch (target.tag) {
            case List.TAG_nil:
                return 0;
            case List.TAG_cons:
```

```
                List_cons sub = (List_cons) target;
                return (sub.arg1 + sub.arg2.f());
            default:
                throw (new Error("Malformed sum value"));
        }
    }
```

In correct programs, neither summand methods that throw errors nor default cases of `switch` are ever reached. The static semantics of *Fun* does not include conditions guaranteeing that programs are correct in this sense. Thus, even a Java program that results from the translation of a *Fun* program may end up throwing those errors. Even if the *Fun* programs were correct, external code may use the generated code in ways that cause those errors to be thrown; in this case, the thrown errors may be useful to detect bugs in external code.

Instead of using `switch`, all pattern matching could be lifted to the top level within *Fun*, introducing suitable auxiliary ops that are called where the **case** terms originally are. After this transformation, pattern matching could be uniformly realized by dynamic dispatch. That is the approach followed in Version 1 of this document. However, experiments suggest that the methods derived from the auxiliary ops make the code less readable. For this reason, we realize pattern matching by dynamic dispatch only if the **case** is already at the top level; otherwise, we use `switch` without introducing auxiliary ops/methods.

2.4 Restriction terms

A restriction term translates to a Java assertion, placed at the beginning of the method that the op translates to (constants, which translate to fields, do not require an assertion because their restriction term is always true). For example, if $\tau(\mathsf{nth}) = \mathsf{List}, \mathsf{Int} \to \mathsf{Int}$ and $\pi(\mathsf{nth}) = (\mathsf{list}, \mathsf{n})$,

$$\rho(\mathsf{nth}) = (0 \leq \mathsf{n}) \text{ and } (\mathsf{n} < \mathsf{length}(\mathsf{list}))$$

translates to

```
assert (0 <= n && n < this.length());
```

So, if

$$\delta(\mathsf{nth}) = \textbf{case } \mathsf{list} \ \{\mathsf{cons}(\mathsf{head}, \mathsf{tail}) \to \textbf{if } (\mathsf{n} = 0) \ \mathsf{head} \ \mathsf{nth}(\mathsf{tail}, \mathsf{n} - 1)\}$$

the op nth translates to the methods

```
abstract int List.nth(int n);

int List_nil.nth(int n) {
    assert (0 <= n && n < this.length());
    throw (new Error("Unexpected sum value"));
}
```

```
int List_cons.nth(int n) {
    assert (0 <= n && n < this.length());
    if (n == 0) {
        return this.arg1;
    } else {
        return this.arg2.nth(n-1);
    }
}
```

If a restriction term translates to an expression with some preceding statements, the expression and the statements must be encapsulated into a method and the assertion consists in a call to this method. For example, if the restriction term on nth were baroquely expressed as

$$\rho(\mathsf{nth}) = \textbf{let}\ \mathsf{len} \leftarrow \mathsf{length}(\mathsf{list})\ \textbf{in}\ ((0 \leq \mathsf{n})\ \mathsf{and}\ (\mathsf{n} < \mathsf{len}))$$

it would translate to

```
assert this.assert_nth(n);
```

where the assertion method is

```
boolean List.assert_nth(int n) {
    int len = this.length();
    return (0 <= n && n < len);
}
```

In correct programs, assertions are always satisfied. The static semantics of *Fun* does not include conditions guaranteeing that programs are correct in this sense. Thus, even a Java program that results from the translation of a *Fun* program may end up violating some assertions. Even if the *Fun* programs were correct, external code may invoke methods with arguments that violate the assertions; in this case, assertions may be useful to detect bugs in external code.

2.5 Defining terms

The defining term of a non-constant op translates to the body of the corresponding method(s), as in the various examples given above. Given that the term translates to an expression preceded by zero or more statements, the body of the method consists of the preceding statements followed by a `return` of the expression. For example, if $\tau(\mathsf{m1}) = \mathsf{P}, \mathsf{Int} \rightarrow \mathsf{Int}$ and $\pi(\mathsf{m1}) = (\mathsf{p}, \mathsf{i})$,

$$\delta(\mathsf{m1}) = \textbf{let}\ \mathsf{z} \leftarrow \mathsf{a}(\mathsf{p}) + \mathsf{i}\ \textbf{in}\ 2 * \mathsf{z}$$

translates to

```
int P.m1(int i) {
    int z = this.a + i;
    return (2 * z);
}
```

It is assumed that $\rho(\mathsf{r}) = \mathsf{true}$, so the `assert` is omitted because it would be pointless.

When the defining term of an op is an **if**, `if-then-else` is preferred over `?:`, e.g.

```
static int Prim.m2(int i) {
    int j = i - 1;
    if (j < 0) {
        return (-j);
    } else {
        return j;
    }
}
```

is produced instead of

```
static int Prim.m2(int i) {
    int j = i - 1;
    return ((j < 0) ? (-j) : j);
}
```

The reason is that the first form tends to be more readable when the expressions are longer.

The defining term of a constant translates to an initializer of the corresponding field or to a static initializer of the class where the field is declared. Given that the term translates to an expression preceded by zero or more statements, there are two cases. If there are no preceding statements, the expression becomes the field initializer. If there are preceding statements, the class where the field is declared includes a static initializer (which is a Java block) consisting of the statements followed by an assignment of the expression to the field. For example,

$$\delta(\mathsf{n1}) = 7$$

translates to

```
static int Prim.n1 = 7;
```

while

$$\delta(\mathsf{n2}) = \textbf{let}\ \mathsf{i} \leftarrow \mathsf{n1} + 2\ \textbf{in}\ 3 * \mathsf{i}$$

translates to

```
static {
    int i = Prim.n1 + 2;
    Prim.n2 = 3 * i;
}
```

For static initializers, analogously to methods, `if-then-else` is preferred over `?:`, e.g.

```
static {
    int h = Prim.n1 * 2;
    if (h < 0) {
        Prim.n3 = -h;
    } else {
        Prim.n3 = h;
    }
}
```

is produced instead of

```
static {
    int h = Prim.n1 * 2;
    Prim.n3 = ((h < 0) ? (-h) : h);
}
```

3 The subset of Java

3.1 Names

Similarly to *Fun*, also the definition of (our formal model of this subset of) Java is parameterized over a set of names

$$\mathcal{N}$$

Despite the use of the same symbol $\mathcal{N}$ used for *Fun*, the two language definitions have disjoint scopes in the semi-formal meta-theory. This remark applies to other symbols used below. When defining the translation from *Fun* to Java, the symbols will be properly disambiguated via decorations.

3.2 Classes

A Java program declares a finite set of classes

$$C \subseteq_{\mathrm{f}} \mathcal{N}$$

Each class may extend another class, as captured by

$$ext : C \to C \uplus \{\mathsf{none}\}$$

Recall that we are formalizing the abstract syntax of Java. So, *ext* is meant to capture explicit `extends` clauses, not the implicit `extends Object` clause. In other words, $ext(c) = \mathsf{none}$ does not mean that c has no superclass; it just means that its declaration has no explicit `extends` clause (i.e. c has `Object` as direct superclass). We do not explicitly model `Object` because it is never referenced by a Java program obtained by translating a *Fun* program.

Whether a class is declared abstract is captured by the predicate

$$abs_{\mathrm{C}} \subseteq C$$

3.3 Types

We only consider three primitive types

$$PTy = \{\texttt{boolean}, \texttt{int}, \texttt{char}\}$$

Classes are the only reference types we consider (i.e. no interfaces or arrays). The types of the program are

$$Ty = PTy \uplus C$$

3.4 Fields

A Java program has a finite set of fields

$$Fld \subseteq_{\mathrm{f}} \{c.f : ty \mid c \in C \ \wedge \ f \in \mathcal{N} \ \wedge \ ty \in Ty\}$$

Formally, a field consists of the class in which it is declared, its name and its type.

Whether a field is static is captured by the predicate

$$stc_{\mathrm{F}} \subseteq Fld$$

3.5 Methods

A Java program has a finite set of methods

$$Mth \subseteq_{\mathrm{f}} \{c.m : \overline{ty} \rightarrow ty \mid c \in C \ \wedge \ m \in \mathcal{N} \ \wedge \ \overline{ty} \in Ty^* \ \wedge \ ty \in Ty\}$$

Formally, a method consists of the class in which it is declared, its name, and its argument and result types; we do not model methods that return `void` because we do not need them to translate *Fun* to Java.

Whether a method is static and/or abstract is captured by the predicates

$$stc_{\mathrm{M}} \subseteq Mth \qquad\qquad abs_{\mathrm{M}} \subseteq Mth$$

3.6 Constructors

A Java program has a finite set of constructors

$$Con \subseteq_{\mathrm{f}} \{c : \overline{ty} \mid c \in C \ \wedge \ \overline{ty} \in Ty^*\}$$

Formally, a constructor consist of the class in which it is declared and its argument types.

3.7 Variables

A variable is a name

$$V = \mathcal{N}$$

Variables capture Java's local variables and method/constructor parameters. While in Java there exist other kinds of variables, in this formalization we reserve the term only for the kinds just mentioned.

3.8 Expressions

The set E of expressions is defined as

$$\frac{v \in V}{v \in E} \quad \text{(variable)}$$

$$\frac{}{\texttt{this} \in E} \quad \text{(self-reference)}$$

$$\frac{}{\texttt{true}, \texttt{false} \in E} \quad \text{(boolean literal)}$$

$$\frac{\begin{array}{c}\iota \in \mathbf{Z} \\ -2^{31} \leq \iota < 2^{31}\end{array}}{\iota \in E} \quad \text{(integer literal)}$$

$$\frac{\begin{array}{c}\xi \in \mathbf{N} \\ \xi < 2^{16}\end{array}}{\xi \in E} \quad \text{(character literal)}$$

$$\frac{\begin{array}{c}e_1, e_2 \in E \\ \odot \in \{\&\&, ||\}\end{array}}{(e_1 \odot e_2) \in E} \quad \text{(binary logical)}$$

$$\frac{e \in E}{(!\ e) \in E} \quad \text{(unary logical)}$$

$$\frac{\begin{array}{c}e_1, e_2 \in E \\ \odot \in \{+, -, *, /, \%\}\end{array}}{(e_1 \odot e_2) \in E} \quad \text{(binary arithmetic)}$$

$$\frac{e \in E}{(-e) \in E} \quad \text{(unary arithmetic)}$$

$$\frac{\begin{array}{c} e_1, e_2 \in E \\ \odot \in \{<, <=, >, >=\} \end{array}}{(e_1 \odot e_2) \in E} \quad \text{(relational)}$$

$$\frac{e_0, e_1, e_2 \in E}{(e_0 \ ?\ e_1 : e_2) \in E} \quad \text{(conditional)}$$

$$\frac{e_1, e_2 \in E}{(e_1 == e_2) \in E} \quad \text{(equality)}$$

$$\frac{e_1, e_2 \in E}{(e_1 != e_2) \in E} \quad \text{(inequality)}$$

$$\frac{\begin{array}{c} c \in C \\ \overline{e} \in E^* \end{array}}{(\texttt{new}\ c(\overline{e})) \in E} \quad \text{(class instance creation)}$$

$$\frac{\begin{array}{c} c \in C \\ \widetilde{mth} \subseteq_{\mathrm{f}} \{\langle m, \overline{ty}, ty, \overline{v}, s\rangle \mid \\ m \in \mathcal{N}\ \wedge \\ \overline{ty} \in Ty^*\ \wedge \\ ty \in Ty\ \wedge \\ \overline{v} \in v^*\ \wedge \\ s \in S\} \end{array}}{(\texttt{new}\ c\ \widetilde{mth}) \in E} \quad \text{(anonymous class instance creation)}$$

$$\frac{\begin{array}{c} e \in E \\ f \in \mathcal{N} \end{array}}{e.f \in E} \quad \text{(instance field access)}$$

$$\frac{\begin{array}{c} c \in C \\ f \in \mathcal{N} \end{array}}{c.f \in E} \quad \text{(static field access)}$$

$$\frac{\begin{array}{c} e \in E \\ m \in \mathcal{N} \\ \overline{e} \in E^* \end{array}}{e.m(\overline{e}) \in E} \quad \text{(instance method invocation)}$$

$$\frac{c \in C \quad m \in \mathcal{N} \quad \overline{e} \in E^*}{c.m(\overline{e}) \in E} \quad \text{(static method invocation)}$$

$$\frac{ty \in Ty \quad e \in E}{((ty)\ e) \in E} \quad \text{(cast)}$$

An anonymous class instance creation expression $(\texttt{new}\ c\ \widetilde{mth})$ consists of the superclass c of the anonymous class and the methods $\widetilde{mth}$ declared in the anonymous class. We model neither arguments passed to the superclass constructor nor fields of anonymous classes because we do not need them to translate *Fun* to Java. A method $\langle m, \overline{ty}, ty, \overline{v}, s\rangle$ of an anonymous class consists of a name m, argument and result types $\overline{ty}$ and ty, parameters $\overline{v}$, and body s (the set S of statements is defined below).

The function $FV : E \to \mathcal{P}_\omega(V)$ collects the (free) variables of an expression

$$\begin{aligned}
FV(v) &= \{v\} \\
FV(\texttt{this}) = FV(\texttt{true}) = FV(\texttt{false}) &= \\
FV(\iota) = FV(\xi) = FV(c.f) &= \emptyset \\
FV(e_1 \odot e_2) = FV(e_1 == e_2) = FV(e_1 != e_2) &= FV(e_1) \cup FV(e_2) \\
FV(!\ e) = FV(-e) = FV(e.f) = FV((ty)\ e) &= FV(e) \\
FV(e_0\ ?\ e_1 : e_2) &= FV(e_0) \cup FV(e_1) \cup FV(e_2) \\
FV(\texttt{new}\ c(\overline{e})) = FV(c.m(\overline{e})) &= \textstyle\bigcup_i FV(e_i) \\
FV(\texttt{new}\ c\ \widetilde{mth}) &= \textstyle\bigcup_{\langle \ldots, \overline{v}, s\rangle \in \widetilde{mth}} (FV(s) - \overline{v}) \\
FV(e.m(\overline{e})) &= FV(e) \cup \textstyle\bigcup_i FV(e_i)
\end{aligned}$$

We define the substitution of the (free) occurrences of a variable v with a variable v' in an expression as

$$
\begin{aligned}
v[v'/v] &= v' \\
w \neq v \;\Rightarrow\; w[v'/v] &= w \\
\texttt{this}[v'/v] &= \texttt{this} \\
\texttt{true}[v'/v] &= \texttt{true} \\
\texttt{false}[v'/v] &= \texttt{false} \\
\iota[v'/v] &= \iota \\
\xi[v'/v] &= \xi \\
(e_1 \odot e_2)[v'/v] &= (e_1[v'/v] \odot e_2[v'/v]) \\
(!\ e)[v'/v] &= (!\ e[v'/v]) \\
(-e)[v'/v] &= (-e[v'/v]) \\
(e_0\ ?\ e_1 : e_2)[v'/v] &= (e_0[v'/v]\ ?\ e_1[v'/v] : e_2[v'/v]) \\
(e_1 == e_2)[v'/v] &= (e_1[v'/v] == e_2[v'/v]) \\
(e_1 != e_2)[v'/v] &= (e_1[v'/v] != e_2[v'/v]) \\
(\texttt{new}\ c(\overline{e}))[v'/v] &= (\texttt{new}\ c(\overline{e}[v'/v])) \\
(\texttt{new}\ c\ \widetilde{mth})[v'/v] &= (\texttt{new}\ c\ \widetilde{mth}[v'/v]) \\
e.f[v'/v] &= e[v'/v].f \\
c.f[v'/v] &= c.f \\
e.m(\overline{e})[v'/v] &= e[v'/v].m(\overline{e}[v'/v]) \\
c.m(\overline{e})[v'/v] &= c.m(\overline{e}[v'/v]) \\
((ty)\ e)[v'/v] &= ((ty)\ e[v'/v])
\end{aligned}
$$

with

$$
\begin{aligned}
\widetilde{mth}[v'/v] &= \{mth[v'/v] \mid mth \in \widetilde{mth}\} \\
v \in \overline{v} \;\Rightarrow\; \langle m, \overline{ty}, ty, \overline{v}, s\rangle[v'/v] &= \langle m, \overline{ty}, ty, \overline{v}, s\rangle \\
v \notin \overline{v} \;\Rightarrow\; \langle m, \overline{ty}, ty, \overline{v}, s\rangle[v'/v] &= \langle m, \overline{ty}, ty, \overline{v}, s[v'/v]\rangle
\end{aligned}
$$

3.9 Statements

The set S of statements is defined as

$$\frac{}{\texttt{mts} \in S}\quad\text{(empty)}$$

$$\frac{e \in E}{(\texttt{return}\ e) \in S}\quad\text{(return)}$$

$$\frac{ty \in Ty \quad v \in V}{(ty\ v) \in S}\quad\text{(local variable declaration)}$$

$$\frac{ty \in Ty \quad v \in V \quad e \in E}{(ty\ v = e) \in S}\quad\text{(local variable declaration with initializer)}$$

$$\frac{ty \in Ty \quad v \in V \quad e \in E}{(\texttt{fin}\ ty\ v = e) \in S} \quad \text{(final local variable declaration)}$$

$$\frac{v \in V \quad e \in E}{(v = e) \in S} \quad \text{(local variable assignment)}$$

$$\frac{e_0, e \in E \quad f \in \mathcal{N}}{(e_0.f = e) \in S} \quad \text{(instance field assignment)}$$

$$\frac{c \in C \quad f \in \mathcal{N} \quad e \in E}{(c.f = e) \in S} \quad \text{(static field assignment)}$$

$$\frac{e \in E \quad s_1, s_2 \in S}{(\texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2) \in S} \quad \text{(conditional)}$$

$$\frac{e \in E \quad \overline{e} \in E^* \quad \overline{s} \in S^* \quad s_0 \in S}{(\texttt{switch}(e)\ \{e_1 \to s_1\ \ldots\ e_n \to s_n\}\ s_0) \in S} \quad \text{(switch)}$$

$$\frac{e \in E}{(\texttt{assert}\ e) \in S} \quad \text{(assertion)}$$

$$\frac{}{\texttt{throwfuneq} \in S} \quad \text{(error throwing for function equality)}$$

$$\frac{}{\texttt{throwmalf} \in S} \quad \text{(error throwing for malformed sum value)}$$

$$\frac{}{\texttt{throwunexp} \in S} \quad \text{(error throwing for unexpected sum value)}$$

$$\frac{s_1, s_2 \in S}{s_1; s_2 \in S} \quad \text{(sequential composition)}$$

$$\frac{}{s; \mathtt{mts} = \mathtt{mts}; s = s} \quad \text{(identity)}$$

$$\frac{}{(s_1; s_2); s_3 = s_1; (s_2; s_3)} \quad \text{(associativity)}$$

The (syntactic) associativity and identity properties of statement composition allow us to omit parentheses and empty statements when statements are composed.

While in Java assignments are expressions, in this formalization we define them as statements for simplicity. We do not model blank finals (i.e. declarations of final variables without an initializer) because we do not need them to translate *Fun* to Java.

The abstract syntax of `switch` is terser than its concrete syntax, e.g. the keywords `case` and `default` (s_0 is the default statement) are omitted. An important omission is an implicit `break` at the end of each case, which must appear in the concrete syntax (unless the statement for a case terminates with a `return`). We do not explicitly capture Java's requirement that case expressions must be constant.

The abstract syntax statements `throwfuneq`, `throwmalf`, and `throwunexp` capture the concrete syntax statements `throw (new Error(...));` where the dots are placeholders for the string messages shown in Section 2 indicating an attempt to compare two functions for equality, a malformed sum value (i.e. one whose tag does not correspond to any summand), and an unexpected tag in a sum value (arising from a **case** that does not have branches for all summands). Since we only throw newly created instances of class `Error` with those three string messages, we leave the class instance creation expressions and the strings implicit in the abstract syntax.

The function $FV : S \to \mathcal{P}_\omega(V)$ collects the free variables of a statement

$$\begin{aligned}
FV(\mathtt{mts}) &= \emptyset \\
FV((\mathtt{return}\ e); s) &= \\
FV((c.f = e); s) &= \\
FV((\mathtt{assert}\ e); s) &= FV(e) \cup FV(s) \\
FV((ty\ v); s) &= FV(s) - \{v\} \\
FV((ty\ v = e); s) &= FV(e) \cup (FV(s) - \{v\}) \\
FV((\mathtt{fin}\ ty\ v = e); s) &= FV(e) \cup (FV(s) - \{v\}) \\
FV((v = e); s) &= \{v\} \cup FV(e) \cup FV(s) \\
FV((e_0.f = e); s) &= FV(e_0) \cup FV(e) \cup FV(s) \\
FV((\mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2); s) &= FV(e) \cup FV(s_1) \cup FV(s_2) \cup FV(s) \\
FV((\mathtt{switch}(e)\ \{e_i \to s_i\}_i\ s_0); s) &= FV(e) \cup \textstyle\bigcup_i FV(e_i) \cup \\
&\quad \textstyle\bigcup_i FV(s_i) \cup FV(s) \\
FV(\mathtt{throwfuneq}; s) &= \\
FV(\mathtt{throwmalf}; s) &= \\
FV(\mathtt{throwunexp}; s) &= FV(s)
\end{aligned}$$

We define the substitution of the free occurrences of a variable v with a variable v' in a statement as

$$\begin{aligned}
\mathtt{mts}[v'/v] &= \mathtt{mts} \\
((\mathtt{return}\ e); s)[v'/v] &= (\mathtt{return}\ e[v'/v]); s[v'/v] \\
((ty\ v); s)[v'/v] &= (ty\ v); s \\
w \neq v \ \Rightarrow\ ((ty\ w); s)[v'/v] &= (ty\ w); s[v'/v] \\
((ty\ v = e); s)[v'/v] &= (ty\ v = e[v'/v]); s \\
w \neq v \ \Rightarrow\ ((ty\ w = e); s)[v'/v] &= (ty\ w = e[v'/v]); s[v'/v] \\
((\mathtt{fin}\ ty\ v = e); s)[v'/v] &= (\mathtt{fin}\ ty\ v = e[v'/v]); s \\
w \neq v \ \Rightarrow\ ((\mathtt{fin}\ ty\ w = e); s)[v'/v] &= (\mathtt{fin}\ ty\ w = e[v'/v]); s[v'/v] \\
((v = e); s)[v'/v] &= (v' = e[v'/v]); s[v'/v] \\
w \neq v \ \Rightarrow\ ((w = e); s)[v'/v] &= (w = e[v'/v]); s[v'/v] \\
((e_0.f = e); s)[v'/v] &= (e_0[v'/v].f = e[v'/v]); s[v'/v] \\
((c.f = e); s)[v'/v] &= (c.f = e[v'/v]); s[v'/v] \\
s_1[v'/v] = s_1' \ \wedge\ s_2[v'/v] = s_2' \ \Rightarrow & \\
((\mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2); s)[v'/v] &= (\mathtt{if}\ (e[v'/v])\ s_1'\ \mathtt{else}\ s_2'); s[v'/v] \\
\forall i.\ \ e_i[v'/v] = e_i' \ \wedge & \\
\forall i.\ \ s_i[v'/v] = s_i' \ \wedge & \\
e[v'/v] = e' \ \Rightarrow & \\
((\mathtt{switch}(e)\ \{e_i \to s_i\}_i\ s_0); s)[v'/v] &= (\mathtt{switch}(e')\ \{e_i' \to s_i'\}_i\ s_0'); s[v'/v] \\
((\mathtt{assert}\ e); s)[v'/v] &= (\mathtt{assert}\ e[v'/v]); s[v'/v] \\
(\mathtt{throwfuneq}; s)[v'/v] &= \mathtt{throwfuneq}; s[v'/v] \\
(\mathtt{throwmalf}; s)[v'/v] &= \mathtt{throwmalf}; s[v'/v] \\
(\mathtt{throwunexp}; s)[v'/v] &= \mathtt{throwunexp}; s[v'/v]
\end{aligned}$$

3.10 Parameters

Each method and each constructor has (formal) parameters that are variables

$$param : Mth \cup Con \to V^*$$

3.11 Bodies

Each non-abstract method and each constructor has a body that is a statement

$$body : Mth \cup Con \xrightarrow{\mathrm{p}} S$$

3.12 Static (field) initializers

Some static fields have initializers that are expressions

$$sfinit : Fld \xrightarrow{\mathrm{p}} E$$

Each class has a finite set of static initializers that are statements

$$sinit : C \to \mathcal{P}_\omega(S)$$

3.13 Program

The program is the 13-tuple

$$\mathcal{P} = \langle C, ext, abs_{\mathrm{C}}, Fld, stc_{\mathrm{F}}, Mth, stc_{\mathrm{M}}, abs_{\mathrm{M}}, Con, param, body, sfinit, sinit\rangle$$

4 The translation, formally

The translation from *Fun* to Java consists of four phases. First, **let** variables are made distinct within each op's defining and restriction term. This phase takes place within *Fun*; its purpose is to make the program amenable to the next phase, namely the language translation to Java. In the third phase, variables in the Java program are restored to their original names whenever possible; this phase takes place within Java. The last phase also takes place within Java: it inlines all the assertion method that consist of single expressions, at the same time eliminating those that are simply `true`.

4.1 Variable renaming

Consider an arbitrary *Fun* program

$$\mathcal{P} = \langle Ty_{\mathrm{U}}, \Delta, Op_{\mathrm{U}}, \tau, \pi, \rho, \delta\rangle$$

The result of variable renaming is the *Fun* program

$$\mathcal{P}' = \langle Ty_{\mathrm{U}}, \Delta, Op_{\mathrm{U}}, \tau, \pi, \rho', \delta'\rangle$$

defined as follows.

4.1.1 Names

If $\mathcal{P}$ uses names from $\mathcal{N}$, $\mathcal{P}'$ uses names from

$$\mathcal{N}' = \mathcal{N} \uplus \{v_k \mid v \in V \ \wedge \ k \in \mathbf{N}_+\}$$

i.e. besides the names in $\mathcal{N}$, $\mathcal{P}'$ uses names obtained by tagging variables of $\mathcal{P}$ with positive naturals.

4.1.2 Term transformation

The idea is very simple: we traverse each term carrying around the set of **let** variables encountered so far. When we find a **let** variable already in the set, we rename it by tagging its name with a numeric index and we also add the new name to the set.

For cosmetic reasons, we define this transformation via a 4-ary relation

$$\leadsto \subseteq T \times \mathcal{P}_\omega(\mathcal{N}') \times T' \times \mathcal{P}_\omega(\mathcal{N}')$$

that is functional; the relational form just makes the rules below more readable by having the transformation look like rewriting. The meaning of $(t \ \widetilde{v} \leadsto t' \ \widetilde{v}')$ is that the result of transforming the term t when the currently used variables are $\widetilde{v}$, is the term t' and that the variables used after that are $\widetilde{v}'$. We use T and T' because while the first term belongs to $\mathcal{P}$ (which uses the names in $\mathcal{N}$), the second term belongs to $\mathcal{P}'$ (which uses the names in $\mathcal{N}'$).

The relation is defined as

$$\frac{}{v \ \widetilde{v} \leadsto v \ \widetilde{v}}$$

$$\frac{}{op \ \widetilde{v} \leadsto op \ \widetilde{v}}$$

$$\frac{\begin{array}{c} t \ \widetilde{v} \leadsto t' \ \widetilde{v}_0 \\ \forall i. \ \ t_i \ \widetilde{v}_{i-1} \leadsto t'_i \ \widetilde{v}_i \end{array}}{t(\overline{t}) \ \widetilde{v} \leadsto t'(\overline{t'}) \ \widetilde{v}_n}$$

$$\frac{t \ \overline{v} \leadsto t' \ \widetilde{v}'}{\lambda\overline{v}.t \ \widetilde{v} \leadsto \lambda\overline{v}.t' \ \widetilde{v}}$$

$$\frac{\forall i. \ \ t_i \ \widetilde{v}_{i-1} \leadsto t'_i \ \widetilde{v}_i}{\{p_i \leftarrow t_i\}_i \ \widetilde{v}_0 \leadsto \{p_i \leftarrow t'_i\}_i \ \widetilde{v}_n}$$

$$\frac{\begin{array}{c} t_1 \ \widetilde{v}_0 \leadsto t'_1 \ \widetilde{v}_1 \\ t_2 \ \widetilde{v}_1 \leadsto t'_2 \ \widetilde{v}_2 \end{array}}{(t_1 = t_2) \ \widetilde{v}_0 \leadsto (t'_1 = t'_2) \ \widetilde{v}_2}$$

$$\frac{\begin{array}{c} t_1 \ \ \widetilde{v}_0 \ \leadsto \ t'_1 \ \ \widetilde{v}_1 \\ t_2 \ \ \widetilde{v}_1 \ \leadsto \ t'_2 \ \ \widetilde{v}_2 \end{array}}{(t_1 \neq t_2) \ \ \widetilde{v}_0 \ \leadsto \ (t'_1 \neq t'_2) \ \ \widetilde{v}_2}$$

$$\frac{\begin{array}{c} t_0 \ \ \widetilde{v} \ \leadsto \ t'_0 \ \ \widetilde{v}_0 \\ t_1 \ \ \widetilde{v}_0 \ \leadsto \ t'_1 \ \ \widetilde{v}_1 \\ t_2 \ \ \widetilde{v}_1 \ \leadsto \ t'_2 \ \ \widetilde{v}_2 \end{array}}{(\mathbf{if} \ t_0 \ t_1 \ t_2) \ \ \widetilde{v} \ \leadsto \ (\mathbf{if} \ t'_0 \ t'_1 \ t'_2) \ \ \widetilde{v}_2}$$

$$\frac{\begin{array}{c} t_0 \ \ \widetilde{v} \ \leadsto \ t'_0 \ \ \widetilde{v}_0 \\ v \notin \widetilde{v}_0 \\ t \ \ \widetilde{v}_0 \cup \{v\} \ \leadsto \ t' \ \ \widetilde{v}' \end{array}}{(\mathbf{let} \ v \leftarrow t_0 \ \mathbf{in} \ t) \ \ \widetilde{v} \ \leadsto \ (\mathbf{let} \ v \leftarrow t'_0 \ \mathbf{in} \ t') \ \ \widetilde{v}'}$$

$$\frac{\begin{array}{c} t_0 \ \ \widetilde{v} \ \leadsto \ t'_0 \ \ \widetilde{v}_0 \\ v \in \widetilde{v}_0 \\ k = \min\{k \in \mathbf{N}_+ \mid v_k \notin \widetilde{v}_0\} \\ t[v_k/v] \ \ \widetilde{v}_0 \cup \{v_k\} \ \leadsto \ t' \ \ \widetilde{v}' \end{array}}{(\mathbf{let} \ v \leftarrow t_0 \ \mathbf{in} \ t) \ \ \widetilde{v} \ \leadsto \ (\mathbf{let} \ v_k \leftarrow t'_0 \ \mathbf{in} \ t') \ \ \widetilde{v}'}$$

$$\frac{\begin{array}{c} t \ \ \widetilde{v} \ \leadsto \ t' \ \ \widetilde{v}_{i_0} \\ \forall j. \ \ t_{i_j} \ \ \widetilde{v}_{i_{j-1}} \ \leadsto \ t'_{i_j} \ \ \widetilde{v}_{i_j} \end{array}}{(\mathbf{case} \ t \ \{c_i(\overline{v}_i) \rightarrow t_i\}_i) \ \ \widetilde{v} \ \leadsto \ (\mathbf{case} \ t' \ \{c_i(\overline{v}_i) \rightarrow t'_i\}_i) \ \ \widetilde{v}_{i_p}}$$

It is easy to see that if $(t \ \ \widetilde{v} \ \leadsto \ t' \ \ \widetilde{v}')$ then $FV(t) = FV(t')$.

Most rules are straightforward: subterms are recursively transformed and used variables are threaded through.

The interesting rules are those for **let**. When **let** is encountered, the subterm t_0 is first transformed. Then, there are two cases. If the bound variable has not been used yet, the variable is added to the set of used variables and the subterm t of the **let** is transformed. If instead the variable has been used, a minimal index is added to it to make it distinct from the variables used so far. The term resulting from substituting the variable in the subterm t is then transformed. The variable substitution does not cause variable overloading or capture because the new variable is in $\mathcal{N}' - \mathcal{N}$ and thus it does not occur in the term where the variable is substituted, whose variables are all in $\mathcal{N}$.

When transforming **case**, it is unnecessary to add the names of the variables bound in the branches to the set of used variables, because variables bound by **case** always translate to field accesses in Java. Thus, variables bound by **case** do not interfere with **let** variables.

When transforming abstractions, the used variables are reset to be the abstraction's parameters $\overline{v}$, without considering $\widetilde{v}$, because abstractions translate to anonymous classes, whose variables shadow variables from the outer classes where the anonymous classes occur. So, outer variables do not interfere with inner variables.

This transformation leaves **let** variables unchanged if they are already distinct.

4.1.3 Transformed program

The only program components that change are the ops' restriction and defining terms

$$(\rho(op) \ \ \pi(op) \ \leadsto \ t \ \ \widetilde{v}) \ \Rightarrow \ \rho'(op) = t$$
$$(\delta(op) \ \ \pi(op) \ \leadsto \ t \ \ \widetilde{v}) \ \Rightarrow \ \delta'(op) = t$$

The set of used variables is initialized with the parameters, because Java local variables and method parameters share the same name space.

4.2 Language translation

Consider a *Fun* program

$$\mathcal{P} = \langle Ty_{\mathrm{U}}, \Delta, Op_{\mathrm{U}}, \tau, \pi, \rho, \delta\rangle$$

such that **let** variables are distinct within each op's defining and restriction term, as resulting from the previous translation phase.

The result of language translation is the Java program

$$\mathcal{P}' = \langle C, ext, abs_{\mathrm{C}}, Fld, stc_{\mathrm{F}}, Mth, stc_{\mathrm{M}}, abs_{\mathrm{M}}, Con, param, body, sfinit, sinit\rangle$$

defined as follows.

4.2.1 Names

If $\mathcal{P}$ uses names from $\mathcal{N}$, $\mathcal{P}'$ uses names from

$$\begin{aligned}
\mathcal{N}' = \mathcal{N} \\
& \uplus \mathcal{N}_{\mathrm{A}} \\
& \uplus \{\mathsf{sumd}^{ty}_{c_i} \mid \Delta(ty) = \textstyle\sum_i c_i \ \overline{ty}_i\} \\
& \uplus \{\mathsf{arg}_j \mid j \in \mathbf{N}_+\} \\
& \uplus \{\mathsf{ires}_k \mid k \in \mathbf{N}_+\} \\
& \uplus \{\mathsf{cres}_l \mid l \in \mathbf{N}_+\} \\
& \uplus \{\mathsf{sub}^{c_i}_l \mid \exists ty. \ \ \Delta(ty) = \textstyle\sum_i c_i \ \overline{ty}_i \ \wedge \ l \in \mathbf{N}_+\} \\
& \uplus \{\mathsf{tgt}_l \mid l \in \mathbf{N}_+\} \\
& \uplus \{\mathsf{tagc}_{c_i} \mid \exists ty. \ \ \Delta(ty) = \textstyle\sum_i c_i \ \overline{ty}_i\} \\
& \uplus \{\mathsf{fin}_v \mid v \in V\} \\
& \uplus \{\mathsf{asrt}_{op} \mid op \in Op_{\mathrm{U}}\} \\
& \uplus \{\mathsf{prim}, \mathsf{eq}, \mathsf{apply}, \mathsf{eqarg}, \mathsf{eqargsub}, \mathsf{tag}, \mathsf{relax}, \mathsf{choose}\}
\end{aligned}$$

where

$$\mathcal{N}_{\mathrm{A}} = \{\mathsf{arrow}^{\overline{ty}}_{ty} \mid \overline{ty} \in Ty^+ \ \wedge \ ty \in Ty\}$$

4.2.2 Types

$\mathcal{P}'$ has an arrow class for every arrow type that occurs in $\mathcal{P}$.

The function $ac : Ty \to \mathcal{P}_\omega(\mathcal{N}_\mathrm{A})$ collects the arrow classes for the arrow types that occur in a type

$$\begin{aligned} ty \in Ty_\mathrm{B} \uplus Ty_\mathrm{U} \ \Rightarrow \ ac(ty) &= \emptyset \\ ac(\overline{ty} \to ty) &= \textstyle\bigcup_i ac(ty_i) \cup ac(ty) \cup \{\mathsf{arrow}^{\overline{ty}}_{ty}\} \end{aligned}$$

The arrow classes for the arrow types that occur in the type definitions of $\mathcal{P}$ are

$$\begin{aligned} AC_\mathrm{TD} = {} & \{ac(ty_i) \mid \exists ty.\ \ \Delta(ty) = \textstyle\prod_i p_i \ ty_i\} \\ \cup {} & \{ac(ty_{j,i}) \mid \exists ty.\ \ \Delta(ty) = \textstyle\sum_i c_i \ \overline{ty}_i\} \\ \cup {} & \{ac(ty_0) \mid \exists ty.\ \ \Delta(ty) = ty_0|r\} \\ \cup {} & \{ac(ty_0) \mid \exists ty.\ \ \Delta(ty) = ty_0/q\} \end{aligned}$$

The arrow classes for the arrow types that occur in the argument and result types of the ops of $\mathcal{P}$ are

$$\begin{aligned} AC_\mathrm{OT} = {} & \{ac(ty_i) \mid \exists op.\ \ \tau(op) = \overline{ty} \to ty\} \\ \cup {} & \{ac(ty) \mid \exists op.\ \ \tau(op) = \overline{ty} \to ty\} \end{aligned}$$

The arrow type of the op itself is not collected because in general it is unnecessary to have the corresponding arrow class, since non-constant ops translate to methods.

The function $ac : T \to \mathcal{P}_\omega(\mathcal{N}_\mathrm{A})$ collects the arrow classes for the arrow types that occur in a term

$$\begin{aligned} ac(v) &= \emptyset \\ op \in COp \ \Rightarrow \ ac(op) &= \emptyset \\ \tau(op) = \overline{ty} \to ty \ \Rightarrow \ ac(op) &= \{\mathsf{arrow}^{\overline{ty}}_{ty}\} \\ ac(op(\overline{t})) = ac(\{p_i \leftarrow t_i\}_i) &= \textstyle\bigcup_i ac(t_i) \\ t \notin Op \ \Rightarrow \ ac(t(\overline{t})) &= ac(t) \cup \textstyle\bigcup_i ac(t_i) \\ ac(\lambda \overline{v}\!:\!\overline{ty}.t\!:\!ty) &= ac(t) \cup \{\mathsf{arrow}^{\overline{ty}}_{ty}\} \\ ac(t_1 = t_2) = ac(t_1 \neq t_2) &= ac(t_1) \cup ac(t_2) \\ ac(\mathbf{if}\ t_0\ t_1\ t_2) &= ac(t_0) \cup ac(t_1) \cup ac(t_2) \\ ac(\mathbf{let}\ v \leftarrow t_0\ \mathbf{in}\ t) &= ac(t_0) \cup ac(t) \\ ac(\mathbf{case}\ t\ \{c_i(\overline{v}_i) \to t_i\}_i) &= ac(t) \cup \textstyle\bigcup_i ac(t_i) \end{aligned}$$

Note that an arrow class is collected from a non-constant op op only if it is not applied to some argument.

The arrow classes for the arrow types that occur in op's restriction terms and defining terms are

$$AC_\mathrm{OR} = \{ac(\rho(op)) \mid op \in Op_\mathrm{U} - COp\}$$

$$AC_\mathrm{OD} = \{ac(\delta(op)) \mid op \in Op_\mathrm{U}\}$$

The arrow classes for the arrow types that occur in $\mathcal{P}$ are

$$AC = AC_{\mathrm{TD}} \uplus AC_{\mathrm{OT}} \uplus AC_{\mathrm{OR}} \uplus AC_{\mathrm{OD}}$$

Besides an arrow class for every arrow type, $\mathcal{P}'$ has a class for each user-defined type, a class for each summand of each sum type, and a class to collect all the fields and methods with primitive types

$$\begin{aligned} C = {} & AC \\ & \uplus\ \mathit{Ty}_{\mathrm{U}} \\ & \uplus\ \{\mathsf{sumd}^{ty}_{c_i} \mid \Delta(ty) = \textstyle\sum_i c_i\ \overline{ty}_i\} \\ & \uplus\ \{\mathsf{prim}\} \end{aligned}$$

Only the summand classes have explicit direct superclasses (the sum classes)

$$\begin{aligned} c \in AC \uplus \mathit{Ty}_{\mathrm{U}} \uplus \{\mathsf{prim}\} \ \Rightarrow\ \mathit{ext}(c) &= \mathsf{none} \\ \mathit{ext}(\mathsf{sumd}^{ty}_{c_i}) &= ty \end{aligned}$$

Only the sum and the arrow classes are abstract

$$\mathit{abs}_{\mathrm{C}}(c) \ \Leftrightarrow\ \Delta(c) \in \sum_i c_i\ \overline{ty}_i \ \vee\ c \in AC$$

The type translation from $\mathcal{P}$ to $\mathcal{P}'$ is captured by the function $tt : \mathit{Ty} \to \mathit{Ty}'$, defined as

$$\begin{aligned} tt(\mathsf{Bool}) &= \mathtt{boolean} \\ tt(\mathsf{Int}) &= \mathtt{int} \\ tt(\mathsf{Char}) &= \mathtt{char} \\ ty \in \mathit{Ty}_{\mathrm{U}} \ \Rightarrow\ tt(ty) &= ty \\ tt(\overline{ty} \to ty) &= \mathsf{arrow}^{ty}_{\overline{ty}} \end{aligned}$$

4.2.3 Fields

There is a field for each projector, declared in the product class

$$\mathit{Fld}_{\mathrm{P}} = \{ty.p_i : tt(ty_i) \mid \Delta(ty) = \prod_i p_i\ ty_i\}$$

There is a field for each constructor argument, declared in the summand class

$$\mathit{Fld}_{\mathrm{CA}} = \{\mathsf{sumd}^{ty}_{c_i}.\mathsf{arg}_j : tt(ty_{j,i}) \mid \Delta(ty) = \sum_i c_i\ \overline{ty}_i\}$$

There is a tag field for each sum type, declared in the sum class

$$\mathit{Fld}_{\mathrm{T}} = \{ty.\mathsf{tag} : \mathtt{int} \mid \Delta(ty) \in \mathit{TySum}\}$$

There is a tag field for each constructor, declared in the sum class

$$\mathit{Fld}_{\mathrm{TC}} = \{ty.\mathsf{tagc}_{c_i} : \mathtt{int} \mid \Delta(ty) = \sum_i c_i\ \overline{ty}_i\}$$

There is a relaxator field for each restriction type, declared in the restriction class

$$\mathit{Fld}_{\mathrm{R}} = \{\mathit{ty}.\mathsf{relax}\colon \mathit{tt}(\mathit{ty}_0) \mid \Delta(\mathit{ty}) = \mathit{ty}_0|r\}$$

There is a chooser field for each quotient type, declared in the quotient class

$$\mathit{Fld}_{\mathrm{C}} = \{\mathit{ty}.\mathsf{choose}\colon \mathit{tt}(\mathit{ty}_0) \mid \Delta(\mathit{ty}) = \mathit{ty}_0/q\}$$

There is a field for each constant constructor, declared in the sum class

$$\mathit{Fld}_{\mathrm{CC}} = \{\mathit{ty}.c_i\colon \mathit{ty} \mid \Delta(\mathit{ty}) = \sum_i c_i\ \overline{\mathit{ty}}_i\ \wedge\ \overline{\mathit{ty}}_i = \epsilon\}$$

There is a field for each user-defined constant with built-in type, declared in the class that collects all the primitive fields and methods

$$\mathit{Fld}_{\mathrm{CB}} = \{\mathsf{prim}.\mathit{op}\colon \mathit{tt}(\mathit{ty}) \mid \mathit{op} \in \mathit{Op}_{\mathrm{U}}\ \wedge\ \tau(\mathit{op}) = \mathit{ty} \in \mathit{Ty}_{\mathrm{B}}\}$$

There is a field for each constant with user-defined type (the constant must be user-defined, because all built-in constants have built-in types), declared in the class for that user-defined type

$$\mathit{Fld}_{\mathrm{CU}} = \{\mathit{ty}.\mathit{op}\colon \mathit{ty} \mid \mathit{op} \in \mathit{Op}_{\mathrm{U}}\ \wedge\ \tau(\mathit{op}) = \mathit{ty} \in \mathit{Ty}_{\mathrm{U}}\}$$

Those are all the fields of $\mathcal{P}'$

$$\mathit{Fld} = \mathit{Fld}_{\mathrm{P}} \uplus \mathit{Fld}_{\mathrm{CA}} \uplus \mathit{Fld}_{\mathrm{T}} \uplus \mathit{Fld}_{\mathrm{TC}} \uplus \mathit{Fld}_{\mathrm{R}} \uplus \mathit{Fld}_{\mathrm{C}} \uplus \mathit{Fld}_{\mathrm{CC}} \uplus \mathit{Fld}_{\mathrm{CB}} \uplus \mathit{Fld}_{\mathrm{CU}}$$

The only static fields are those for constructor tags, constant constructors, and constants

$$\mathit{stc}_{\mathrm{F}}(\mathit{fld})\ \Leftrightarrow\ \mathit{fld} \in \mathit{Fld}_{\mathrm{TC}} \uplus \mathit{Fld}_{\mathrm{CC}} \uplus \mathit{Fld}_{\mathrm{CB}} \uplus \mathit{Fld}_{\mathrm{CU}}$$

4.2.4 Methods

There is an equality method declared in each product, sum, restriction, and quotient

$$\begin{aligned}
\mathit{Mth}_{\mathrm{EP}} &= \{\mathit{ty}.\mathsf{eq}\colon \mathit{ty} \to \mathtt{boolean} \mid \Delta(\mathit{ty}) \in \mathit{TyProd}\}\\
\mathit{Mth}_{\mathrm{ES}} &= \{\mathit{ty}.\mathsf{eq}\colon \mathit{ty} \to \mathtt{boolean} \mid \Delta(\mathit{ty}) \in \mathit{TySum}\}\\
\mathit{Mth}_{\mathrm{ER}} &= \{\mathit{ty}.\mathsf{eq}\colon \mathit{ty} \to \mathtt{boolean} \mid \Delta(\mathit{ty}) \in \mathit{TyRestr}\}\\
\mathit{Mth}_{\mathrm{EQ}} &= \{\mathit{ty}.\mathsf{eq}\colon \mathit{ty} \to \mathtt{boolean} \mid \Delta(\mathit{ty}) \in \mathit{TyQuot}\}
\end{aligned}$$

There is also an equality method declared in each summand class (which implements the abstract equality method declared in the sum class)

$$\mathit{Mth}_{\mathrm{ESS}} = \{\mathsf{sumd}^{\mathit{ty}}_{c_i}.\mathsf{eq}\colon \mathit{ty} \to \mathtt{boolean} \mid \Delta(\mathit{ty}) = \sum_i c_i\ \overline{\mathit{ty}}_i\}$$

There is also an equality method declared in each arrow class

$$\mathit{Mth}_{\mathrm{EA}} = \{\mathsf{arrow}^{\overline{\mathit{ty}}}_{\mathit{ty}}.\mathsf{eq}\colon \mathsf{arrow}^{\overline{\mathit{ty}}}_{\mathit{ty}} \to \mathtt{boolean} \mid \mathsf{arrow}^{\overline{\mathit{ty}}}_{\mathit{ty}} \in \mathit{AC}\}$$

There is a method for each non-constant constructor, declared in the sum class

$$Mth_{\mathrm{C}} = \{ty.c_i : tt(\overline{ty}_i) \to ty \mid \Delta(ty) = \sum_i c_i\ \overline{ty}_i\ \wedge\ \overline{ty}_i \neq \epsilon\}$$

There is an application method for each arrow class

$$Mth_{\mathrm{A}} = \{\mathsf{arrow}^{ty}_{\overline{ty}}.\mathsf{apply} : tt(\overline{ty}) \to tt(ty) \mid \mathsf{arrow}^{ty}_{\overline{ty}} \in AC\}$$

There are methods for each op with at least a user-defined argument type and whose defining term is a **case** that operates on the leftmost parameter with user-defined type, which must be a sum type. A method is declared in the sum class and a method is declared in each subclass[7]

$$\begin{aligned}
Mth_{\mathrm{PM}} = \{&ty_h.op : tt(del(\overline{ty}, h)) \to tt(ty) \mid \\
&op \in Op_{\mathrm{U}}\ \wedge\ \tau(op) = \overline{ty} \to ty\ \wedge \\
&h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\}\ \wedge\ \delta(op) = (\mathbf{case}\ \pi(op)_h\ \ldots)\} \\
Mth_{\mathrm{PMS}} = \{&\mathsf{sumd}^{ty_h}_{c_i}.op : tt(del(\overline{ty}, h)) \to tt(ty) \mid \\
&op \in Op_{\mathrm{U}}\ \wedge\ \tau(op) = \overline{ty} \to ty\ \wedge \\
&h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\}\ \wedge\ \delta(op) = (\mathbf{case}\ \pi(op)_h\ \ldots)\}
\end{aligned}$$

There is a method for each op with at least a user-defined argument type and whose defining term is not a **case** that operates on the leftmost parameter with user-defined type; the method is declared in the class for the leftmost user-defined argument type

$$\begin{aligned}
Mth_{\mathrm{NPM}} = \{&ty_h.op : tt(del(\overline{ty}, h)) \to tt(ty) \mid \\
&op \in Op_{\mathrm{U}}\ \wedge\ \tau(op) = \overline{ty} \to ty\ \wedge \\
&h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\}\ \wedge\ \delta(op) \neq (\mathbf{case}\ \pi(op)_h\ \ldots)\}
\end{aligned}$$

There is a method for each non-constant user-defined op with no user-defined argument types and with user-defined result type, declared in the class for that user-defined type

$$\begin{aligned}
Mth_{\mathrm{UR}} = \{&ty.op : tt(\overline{ty}) \to ty \mid \\
&op \in Op_{\mathrm{U}}\ \wedge\ \tau(op) = \overline{ty} \to ty\ \wedge\ \overline{ty} \cap Ty_{\mathrm{U}} = \emptyset\ \wedge\ ty \in Ty_{\mathrm{U}}\}
\end{aligned}$$

The function $ut : Ty \to Ty^*_{\mathrm{U}}$ collects all the user-defined types that occur in a type, in left-to-right order

$$\begin{aligned}
ty \in Ty_{\mathrm{B}} \ &\Rightarrow\ ut(ty) = \epsilon \\
ty \in Ty_{\mathrm{U}} \ &\Rightarrow\ ut(ty) = ty \\
ut(\overline{ty} \to ty) &= (ut(ty_1), \ldots, ut(ty_n), ut(ty))
\end{aligned}$$

There is a method for each non-constant user-defined op with no user-defined argument or result types but with at least a user-defined type occurring in (some

[7] **Notation.** If $\overline{x}$ is a sequence, $del(\overline{x}, i)$ is the sequence obtained by deleting the i-th element from $\overline{x}$.

arrow type in) its argument or result types; the method is declared in the class for the leftmost user-defined type

$$\begin{aligned} \mathit{Mth}_{\mathrm{UA}} = \{ & \mathit{ty}'.\mathit{op} : \mathit{tt}(\overline{\mathit{ty}}) \to \mathit{tt}(\mathit{ty}) \mid \\ & \mathit{op} \in \mathit{Op}_{\mathrm{U}} \ \wedge\ \tau(\mathit{op}) = \overline{\mathit{ty}} \to \mathit{ty} \ \wedge \\ & (\overline{\mathit{ty}}, \mathit{ty}) \cap \mathit{Ty}_{\mathrm{U}} = \emptyset \ \wedge\ \mathit{ut}(\overline{\mathit{ty}} \to \mathit{ty}) = (\mathit{ty}', \ldots)\} \end{aligned}$$

There is a method for each non-constant user-defined op with all built-in types, declared in the class that collects all the primitive fields and methods

$$\begin{aligned} \mathit{Mth}_{\mathrm{B}} = \{ & \mathsf{prim}.\mathit{op} : \mathit{tt}(\overline{\mathit{ty}}) \to \mathit{tt}(\mathit{ty}) \mid \\ & \mathit{op} \in \mathit{Op}_{\mathrm{U}} \ \wedge\ \tau(\mathit{op}) = \overline{\mathit{ty}} \to \mathit{ty} \ \wedge \\ & (\overline{\mathit{ty}}, \mathit{ty}) \cap \mathit{Ty}_{\mathrm{U}} = \emptyset \ \wedge\ \mathit{ut}(\overline{\mathit{ty}} \to \mathit{ty}) = \epsilon\} \end{aligned}$$

There is an assertion method for each method in $\mathit{Mth}_{\mathrm{PM}} \uplus \mathit{Mth}_{\mathrm{NPM}}$

$$\begin{aligned} \mathit{Mth}_{\mathrm{AsU}} = \{ & \mathit{ty}_h.\mathsf{asrt}_{\mathit{op}} : \mathit{tt}(\mathit{del}(\overline{\mathit{ty}}, h)) \to \mathtt{boolean} \mid \\ & \mathit{op} \in \mathit{Op}_{\mathrm{U}} \ \wedge\ \tau(\mathit{op}) = \overline{\mathit{ty}} \to \mathit{ty} \ \wedge\ h = \min\{h \mid \mathit{ty}_h \in \mathit{Ty}_{\mathrm{U}}\}\} \end{aligned}$$

There is an assertion method for each method in $\mathit{Mth}_{\mathrm{UR}}$

$$\begin{aligned} \mathit{Mth}_{\mathrm{AsUR}} = \{ & \mathit{ty}.\mathsf{asrt}_{\mathit{op}} : \mathit{tt}(\overline{\mathit{ty}}) \to \mathtt{boolean} \mid \\ & \mathit{op} \in \mathit{Op}_{\mathrm{U}} \ \wedge\ \tau(\mathit{op}) = \overline{\mathit{ty}} \to \mathit{ty} \ \wedge\ \overline{\mathit{ty}} \cap \mathit{Ty}_{\mathrm{U}} = \emptyset \ \wedge\ \mathit{ty} \in \mathit{Ty}_{\mathrm{U}}\} \end{aligned}$$

There is an assertion method for each method in $\mathit{Mth}_{\mathrm{UA}}$

$$\begin{aligned} \mathit{Mth}_{\mathrm{AsUA}} = \{ & \mathit{ty}'.\mathsf{asrt}_{\mathit{op}} : \mathit{tt}(\overline{\mathit{ty}}) \to \mathtt{boolean} \mid \\ & \mathit{op} \in \mathit{Op}_{\mathrm{U}} \ \wedge\ \tau(\mathit{op}) = \overline{\mathit{ty}} \to \mathit{ty} \ \wedge \\ & (\overline{\mathit{ty}}, \mathit{ty}) \cap \mathit{Ty}_{\mathrm{U}} = \emptyset \ \wedge\ \mathit{ut}(\overline{\mathit{ty}} \to \mathit{ty}) = (\mathit{ty}', \ldots)\} \end{aligned}$$

There is an assertion method for each method in $\mathit{Mth}_{\mathrm{B}}$

$$\begin{aligned} \mathit{Mth}_{\mathrm{AsB}} = \{ & \mathsf{prim}.\mathsf{asrt}_{\mathit{op}} : \mathit{tt}(\overline{\mathit{ty}}) \to \mathtt{boolean} \mid \\ & \mathit{op} \in \mathit{Op}_{\mathrm{U}} \ \wedge\ \tau(\mathit{op}) = \overline{\mathit{ty}} \to \mathit{ty} \ \wedge \\ & (\overline{\mathit{ty}}, \mathit{ty}) \cap \mathit{Ty}_{\mathrm{U}} = \emptyset \ \wedge\ \mathit{ut}(\overline{\mathit{ty}} \to \mathit{ty}) = \epsilon\} \end{aligned}$$

Those are all the methods of $\mathcal{P}'$

$$\begin{aligned} \mathit{Mth} = {} & \mathit{Mth}_{\mathrm{EP}} \uplus \mathit{Mth}_{\mathrm{ES}} \uplus \mathit{Mth}_{\mathrm{ER}} \uplus \mathit{Mth}_{\mathrm{EQ}} \uplus \mathit{Mth}_{\mathrm{ESS}} \uplus \mathit{Mth}_{\mathrm{EA}} \\ & \uplus \mathit{Mth}_{\mathrm{C}} \uplus \mathit{Mth}_{\mathrm{A}} \\ & \uplus \mathit{Mth}_{\mathrm{PM}} \uplus \mathit{Mth}_{\mathrm{PMS}} \uplus \mathit{Mth}_{\mathrm{NPM}} \uplus \mathit{Mth}_{\mathrm{UR}} \uplus \mathit{Mth}_{\mathrm{UA}} \uplus \mathit{Mth}_{\mathrm{B}} \\ & \uplus \mathit{Mth}_{\mathrm{AsU}} \uplus \mathit{Mth}_{\mathrm{AsUR}} \uplus \mathit{Mth}_{\mathrm{AsUA}} \uplus \mathit{Mth}_{\mathrm{AsB}} \end{aligned}$$

The only static methods are those for constructors, those for ops without user-defined argument types, and the assertion methods for them

$$\begin{aligned} \mathit{stc}_{\mathrm{M}}(\mathit{mth}) \ \Leftrightarrow\ & \mathit{mth} \in \mathit{Mth}_{\mathrm{C}} \uplus \mathit{Mth}_{\mathrm{UR}} \uplus \mathit{Mth}_{\mathrm{UA}} \uplus \mathit{Mth}_{\mathrm{B}} \uplus \\ & \mathit{Mth}_{\mathrm{AsUR}} \uplus \mathit{Mth}_{\mathrm{AsUA}} \uplus \mathit{Mth}_{\mathrm{AsB}} \end{aligned}$$

The only abstract methods are those for equality of sum types, the application method, and those, declared in sum classes, for ops with at least a user-defined argument type and whose defining term is a **case** that operates on the leftmost argument with user-defined type

$$\mathit{abs}_{\mathrm{M}}(\mathit{mth}) \ \Leftrightarrow\ \mathit{mth} \in \mathit{Mth}_{\mathrm{ES}} \uplus \mathit{Mth}_{\mathrm{A}} \uplus \mathit{Mth}_{\mathrm{PM}}$$

4.2.5 Constructors

There is a constructor in every product, summand, restriction, and quotient class

$$\begin{aligned} Con_{\mathrm{P}} &= \{ty : \overline{ty} \mid \Delta(ty) = \textstyle\prod_i p_i \ ty_i\} \\ Con_{\mathrm{S}} &= \{\mathsf{sumd}_{c_i}^{ty} : \overline{ty}_i \mid \Delta(ty) = \textstyle\sum_i c_i \ \overline{ty_i}\} \\ Con_{\mathrm{R}} &= \{ty : ty_0 \mid \Delta(ty) = ty_0 | r\} \\ Con_{\mathrm{Q}} &= \{ty : ty_0 \mid \Delta(ty) = ty_0 / q\} \\ Con &= Con_{\mathrm{P}} \uplus Con_{\mathrm{S}} \uplus Con_{\mathrm{R}} \uplus Con_{\mathrm{Q}} \end{aligned}$$

4.2.6 Parameters

The methods have parameters

$$\frac{mth \in Mth_{\mathrm{EP}} \uplus Mth_{\mathrm{ES}} \uplus Mth_{\mathrm{ESS}} \uplus Mth_{\mathrm{ER}} \uplus Mth_{\mathrm{EQ}} \uplus Mth_{\mathrm{EA}}}{param(mth) = \mathsf{eqarg}}$$

$$\frac{mth \in Mth_{\mathrm{C}}}{param(mth) = \overline{\mathsf{arg}}}$$

$$\frac{mth \in Mth_{\mathrm{A}}}{param(mth) = \overline{\mathsf{arg}}}$$

$$\frac{\begin{array}{c} mth = c.op : tt(del(\overline{ty}, h)) \to tt(ty) \in Mth_{\mathrm{PM}} \uplus Mth_{\mathrm{PMS}} \uplus Mth_{\mathrm{NPM}} \\ \tau(op) = \overline{ty} \to ty \\ h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\} \end{array}}{param(mth) = del(\pi(op), h)}$$

$$\frac{mth = c.op : \overline{ty} \to ty \in Mth_{\mathrm{UR}} \uplus Mth_{\mathrm{UA}} \uplus Mth_{\mathrm{B}}}{param(mth) = \pi(op)}$$

$$\frac{mth = c.\mathsf{asrt}_{op} : tt(del(\overline{ty}, h)) \to \texttt{boolean} \in Mth_{\mathrm{AsU}}}{param(mth) = del(\pi(op), h)}$$

$$\frac{mth = c.\mathsf{asrt}_{op} : \overline{ty} \to \texttt{boolean} \in Mth_{\mathrm{AsUR}} \uplus Mth_{\mathrm{AsUA}} \uplus Mth_{\mathrm{AsB}}}{param(mth) = \pi(op)}$$

For methods derived from user-defined ops, the parameters are derived from those of the ops. For equality methods, we use a parameter with name eqarg. For methods derived from constructors, we use parameters arg_j.

The constructors have parameters

$$\begin{aligned} con = ty : \overline{ty} \in Con_{\mathrm{P}} \ \wedge \ ty = \textstyle\prod_i p_i \ ty_i \ &\Rightarrow \ param(con) = \overline{p} \\ con = \mathsf{sumd}_{c_i}^{ty} : \overline{ty}_i \in Con_{\mathrm{S}} \ &\Rightarrow \ param(con) = \overline{\mathsf{arg}} \\ con = ty : ty_0 \in Con_{\mathrm{R}} \ &\Rightarrow \ param(con) = \mathsf{relax} \\ con = ty : ty_0 \in Con_{\mathrm{Q}} \ &\Rightarrow \ param(con) = \mathsf{choose} \end{aligned}$$

For product constructors, we use the projectors as parameters. For summand constructors, we use parameters arg_j. For restriction and quotient constructors, we use parameters relax and choose.

4.2.7 Translation of terms to expressions and statements

In general, each *Fun* term translates to a Java expression preceded by a Java statement. The statement assigns values to local variables that are used in the expression.

A *Fun* variable does not always translate to a Java variable. When an op translates to an instance method, the leftmost parameter with user-defined type translates to `this`. When an op translates to methods of the summand classes, the variables bound in each branch of the **case** translate to field accesses of the corresponding summand classes. When a **case** translates to a `switch`, the variables bound in each branch also translate to field accesses. To capture the translation of a finite number of *Fun* variables to `this` or to field accesses, we use translation contexts

$$TC = V \xrightarrow{\mathrm{f}} \{\mathtt{this}\} \uplus \{\mathtt{this}.f \mid f \in \mathcal{N}'\} \uplus \{\mathsf{sub}_l^{c_i}.f \mid \mathsf{sub}_l^{c_i}, f \in \mathcal{N}'\}$$

Since **if** may translate to `if`, we need to generate a fresh local variable ires_k to store the result computed by the two branches. We do that by threading a positive natural k while we traverse and translate the terms. We also thread a positive natural l to generate fresh local variables cres_l to store the results of `switch`, fresh local variables $\mathsf{sub}_l^{c_i}$ to store the result of casting (references to) sum class instances to summand classes inside the branches of `switch`, and fresh local variables tgt_l to store the targets of `switch`.

Before proceeding, it is useful to define a function $eq : Ty \times E \times E \to E$ that produces an equality expression for two given expressions

$$\begin{aligned} ty \in Ty_{\mathrm{B}} &\Rightarrow eq_{ty}(e_1, e_2) = (e_1 == e_2) \\ ty \in Ty_{\mathrm{U}} \uplus Ty_{\mathrm{A}} &\Rightarrow eq_{ty}(e_1, e_2) = e_1.\mathsf{eq}(e_2) \end{aligned}$$

If the first argument is a built-in type, equality is realized via the `==` operator; if it is a user-defined type, by calling the equality method. This function merely serves to factor these two cases from some of the definitions below. We define an analogous function $ineq : Ty \times E \times E \to E$

$$\begin{aligned} ty \in Ty_{\mathrm{B}} &\Rightarrow ineq_{ty}(e_1, e_2) = (e_1 != e_2) \\ ty \in Ty_{\mathrm{U}} \uplus Ty_{\mathrm{A}} &\Rightarrow ineq_{ty}(e_1, e_2) = !\ e_1.\mathsf{eq}(e_2) \end{aligned}$$

To abbreviate the translation rules below, we define a function *bot* that translates the binary built-in ops of *Fun* to the corresponding Java binary operators

$$
\begin{aligned}
bot(\mathsf{and}) &= \texttt{\&\&} \\
bot(\mathsf{or}) &= \texttt{||} \\
bot(+) &= + \\
bot(-) &= - \\
bot(*) &= * \\
bot(/) &= / \\
bot(\mathsf{mod}) &= \% \\
bot(<) &= < \\
bot(\leq) &= <= \\
bot(>) &= > \\
bot(\geq) &= >=
\end{aligned}
$$

The translation of terms to expressions preceded by statements is captured by a 9-ary functional relation

$$\rightsquigarrow \subseteq TC \times Cx \times T \times \mathbf{N}_+ \times \mathbf{N}_+ \times S \times E \times \mathbf{N}_+ \times \mathbf{N}_+$$

The meaning of $(t \;\; k \;\; l \;\rightsquigarrow^{tc,cx}\; s \;\; e \;\; k' \;\; l')$ is that, in the translation context tc, the result of translating the term t with context cx when the currently available indices for ires and $\mathsf{cres}/\mathsf{sub}/\mathsf{tgt}$ variables are k and l, is the expression e preceded by the statement s and that the next available indices are k' and l'. For readability, tc and cx may be left implicit.

As explained in Section 2.3.11, if the only use of an ires variable is to be assigned to another variable, then the ires variable is omitted and the other variable is assigned inside the branches of the `if`. An analogous omission takes place when the only use of the ires variable is to be returned by a method or to be assigned to a static field. The same is done with cres variables. This could be formalized as a further transformation taking place within Java after the language translation, which eliminates unneeded ires and cres variables. However, it is actually easier to incorporate the omission in the language translation, by means of variants of the $\rightsquigarrow$ relation that embed information about the use of the expression resulting from translating a term and that produce statements (without expressions) that use the expressions according to the embedded information.

For return by a method, there is an 8-ary functional relation

$$\rightsquigarrow_{\mathrm{Ret}} \subseteq TC \times Cx \times T \times \mathbf{N}_+ \times \mathbf{N}_+ \times S \times \mathbf{N}_+ \times \mathbf{N}_+$$

The meaning of $(t \;\; k \;\; l \;\rightsquigarrow^{tc,cx}_{\mathrm{Ret}}\; s \;\; k' \;\; l')$ is that, in the translation context tc, the statement that returns the expression resulting from translating the term t with context cx when the currently available indices for ires and $\mathsf{cres}/\mathsf{sub}/\mathsf{tgt}$ variables are k and l, is s and that the next available indices are k' and l'. For readability, tc and cx may be left implicit.

For assignment to a newly declared variable, there is a 10-ary functional relation

$$\rightsquigarrow_{\mathrm{AsgNV}} \subseteq Ty' \times V' \times TC \times Cx \times T \times \mathbf{N}_+ \times \mathbf{N}_+ \times S \times \mathbf{N}_+ \times \mathbf{N}_+$$

The meaning of $(t \quad k \quad l \quad \rightsquigarrow^{tc,cx}_{\mathrm{AsgNV}(ty,v)} \quad s \quad k' \quad l')$ is that, in the translation context tc, the statement that assigns to the newly declared variable v of type ty the expression resulting from translating the term t with context cx when the currently available indices for ires and $\mathsf{cres}/\mathsf{sub}/\mathsf{tgt}$ variables are k and l, is s and that the next available indices are k' and l'. For readability, tc and cx may be left implicit.

For assignment to a (non-newly declared) variable, there is a 9-ary functional relation

$$\rightsquigarrow_{\mathrm{AsgV}} \subseteq V' \times TC \times Cx \times T \times \mathbf{N}_+ \times \mathbf{N}_+ \times S \times \mathbf{N}_+ \times \mathbf{N}_+$$

The meaning of $(t \quad k \quad l \quad \rightsquigarrow^{tc,cx}_{\mathrm{AsgV}(v)} \quad s \quad k' \quad l')$ is that, in the translation context tc, the statement that assigns to the variable v the expression resulting from translating the term t with context cx when the currently available indices for ires and $\mathsf{cres}/\mathsf{sub}/\mathsf{tgt}$ variables are k and l, is s and that the next available indices are k' and l'. For readability, tc and cx may be left implicit.

For assignment to a static field, there is a 10-ary functional relation

$$\rightsquigarrow_{\mathrm{AsgF}} \subseteq C \times \mathcal{N}' \times TC \times Cx \times T \times \mathbf{N}_+ \times \mathbf{N}_+ \times S \times \mathbf{N}_+ \times \mathbf{N}_+$$

The meaning of $(t \quad k \quad l \quad \rightsquigarrow^{tc,cx}_{\mathrm{AsgF}(c,f)} \quad s \quad k' \quad l')$ is that, in the translation context tc, the statement that assigns to the static field with name f of class c the expression resulting from translating the term t with context cx when the currently available indices for ires and $\mathsf{cres}/\mathsf{sub}/\mathsf{tgt}$ variables are k and l, is s and that the next available indices are k' and l'. For readability, tc and cx may be left implicit.

We assume that the variables in V' are endowed with a linear order, so that there exists a function

$$order : \mathcal{P}_\omega(V') \to (V')^{(*)}$$

that orders the elements of a finite set of variables in a sequence (without repetitions, because each element of the set is only picked once).

The five relations are (mutually recursively) defined as

$$\frac{v \in \mathcal{D}(tc)}{v \quad k \quad l \rightsquigarrow \texttt{mts} \quad tc(v) \quad k \quad l}$$

$$\frac{v \notin \mathcal{D}(tc)}{v \quad k \quad l \rightsquigarrow \texttt{mts} \quad v \quad k \quad l}$$

$$\frac{}{\mathsf{true} \quad k \quad l \rightsquigarrow \texttt{mts} \quad \texttt{true} \quad k \quad l}$$

$$\frac{}{\mathsf{false} \quad k \quad l \rightsquigarrow \texttt{mts} \quad \texttt{false} \quad k \quad l}$$

$$\frac{}{\iota \;\; k \;\; l \;\leadsto\; \texttt{mts} \;\; \iota \;\; k \;\; l}$$

$$\frac{}{\xi \;\; k \;\; l \;\leadsto\; \texttt{mts} \;\; \xi \;\; k \;\; l}$$

$$\frac{t \;\; k \;\; l \;\leadsto\; s \;\; e \;\; k' \;\; l'}{(\mathsf{not}\; t) \;\; k \;\; l \;\leadsto\; s \;\; (!\, e) \;\; k' \;\; l'}$$

$$\frac{t \;\; k \;\; l \;\leadsto\; s \;\; e \;\; k' \;\; l'}{(\mathsf{minus}\; t) \;\; k \;\; l \;\leadsto\; s \;\; (-e) \;\; k' \;\; l'}$$

$$\frac{\begin{array}{c} t_1 \;\; k_0 \;\; l_0 \;\leadsto\; s_1 \;\; e_1 \;\; k_1 \;\; l_1 \\ t_2 \;\; k_1 \;\; l_1 \;\leadsto\; s_2 \;\; e_2 \;\; k_2 \;\; l_2 \\ bot(\circ) = \odot \end{array}}{(t_1 \circ t_2) \;\; k_0 \;\; l_0 \;\leadsto\; s_1; s_2 \;\; (e_1 \odot e_2) \;\; k_2 \;\; l_2}$$

$$\frac{t \;\; k \;\; l \;\leadsto\; s \;\; e \;\; k' \;\; l'}{\mathsf{c2i}(t) \;\; k \;\; l \;\leadsto\; s \;\; ((\texttt{int})\; e) \;\; k' \;\; l'}$$

$$\frac{t \;\; k \;\; l \;\leadsto\; s \;\; e \;\; k' \;\; l'}{\mathsf{i2c}(t) \;\; k \;\; l \;\leadsto\; s \;\; ((\texttt{char})\; e) \;\; k' \;\; l'}$$

$$\frac{mth = \langle \mathsf{apply}, \texttt{boolean}, \texttt{boolean}, \mathsf{arg}_1, (\texttt{return}\; (!\; \mathsf{arg}_1)) \rangle}{\mathsf{not} \;\; k \;\; l \;\leadsto\; \texttt{mts} \;\; (\texttt{new}\; \mathsf{arrow}^{\mathsf{Bool}}_{\mathsf{Bool}}\; \{mth\}) \;\; k \;\; l}$$

$$\frac{mth = \langle \mathsf{apply}, \texttt{int}, \texttt{int}, \mathsf{arg}_1, (\texttt{return}\; (-\mathsf{arg}_1)) \rangle}{\mathsf{minus} \;\; k \;\; l \;\leadsto\; \texttt{mts} \;\; (\texttt{new}\; \mathsf{arrow}^{\mathsf{Int}}_{\mathsf{Int}}\; \{mth\}) \;\; k \;\; l}$$

$$\frac{\begin{array}{c} \circ \in \{\mathsf{and}, \mathsf{or}\} \\ bot(\circ) = \odot \\ s = (\texttt{return}\; (\mathsf{arg}_1 \odot \mathsf{arg}_2)) \\ mth = \langle \mathsf{apply}, (\texttt{boolean}, \texttt{boolean}), \texttt{boolean}, (\mathsf{arg}_1, \mathsf{arg}_2), s \rangle \end{array}}{\circ \;\; k \;\; l \;\leadsto\; \texttt{mts} \;\; (\texttt{new}\; \mathsf{arrow}^{\mathsf{Bool},\mathsf{Bool}}_{\mathsf{Bool}}\; \{mth\}) \;\; k \;\; l}$$

$$\frac{\begin{array}{c} \circ \in \{+, -, *, /, \mathsf{mod}\} \\ bot(\circ) = \odot \\ s = (\texttt{return}\; (\mathsf{arg}_1 \odot \mathsf{arg}_2)) \\ mth = \langle \mathsf{apply}, (\texttt{int}, \texttt{int}), \texttt{int}, (\mathsf{arg}_1, \mathsf{arg}_2), s \rangle \end{array}}{\circ \;\; k \;\; l \;\leadsto\; \texttt{mts} \;\; (\texttt{new}\; \mathsf{arrow}^{\mathsf{Int},\mathsf{Int}}_{\mathsf{Int}}\; \{mth\}) \;\; k \;\; l}$$

$$\frac{\begin{array}{c}\circ \in \{<, \leq, >, \geq\} \\ bot(\circ) = \odot \\ s = (\texttt{return}\ (\mathsf{arg}_1 \odot \mathsf{arg}_2)) \\ mth = \langle \mathsf{apply}, (\texttt{int}, \texttt{int}), \texttt{boolean}, (\mathsf{arg}_1, \mathsf{arg}_2), s\rangle\end{array}}{\circ\ \ k\ \ l\ \rightsquigarrow\ \texttt{mts}\ \ (\texttt{new}\ \mathsf{arrow}^{\mathsf{Int},\mathsf{Int}}_{\mathsf{Bool}}\ \{mth\})\ \ k\ \ l}$$

$$\frac{mth = \langle \mathsf{apply}, \texttt{char}, \texttt{int}, \mathsf{arg}_1, (\texttt{return}\ (\texttt{int})\ \mathsf{arg}_1)\rangle}{\mathsf{c2i}\ \ k\ \ l\ \rightsquigarrow\ \texttt{mts}\ \ (\texttt{new}\ \mathsf{arrow}^{\mathsf{Char}}_{\mathsf{Int}}\ \{mth\})\ \ k\ \ l}$$

$$\frac{mth = \langle \mathsf{apply}, \texttt{int}, \texttt{char}, \mathsf{arg}_1, (\texttt{return}\ (\texttt{char})\ \mathsf{arg}_1)\rangle}{\mathsf{i2c}\ \ k\ \ l\ \rightsquigarrow\ \texttt{mts}\ \ (\texttt{new}\ \mathsf{arrow}^{\mathsf{Int}}_{\mathsf{Char}}\ \{mth\})\ \ k\ \ l}$$

$$\frac{\begin{array}{c}op \in Op_{\mathrm{U}} \\ \tau(op) = ty \in Ty_{\mathrm{B}}\end{array}}{op\ \ k\ \ l\ \rightsquigarrow\ \texttt{mts}\ \ \mathsf{prim}.op\ \ k\ \ l}$$

$$\frac{\begin{array}{c}op \in Op_{\mathrm{U}} \\ \tau(op) = ty \in Ty_{\mathrm{U}}\end{array}}{op\ \ k\ \ l\ \rightsquigarrow\ \texttt{mts}\ \ ty.op\ \ k\ \ l}$$

$$\frac{\begin{array}{c}op \in Op_{\mathrm{U}} \\ \tau(op) = \overline{ty} \to ty \\ h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\} \\ \forall i.\ \ t_i\ \ k_{i-1}\ \ l_{i-1}\ \rightsquigarrow\ s_i\ \ e_i\ \ k_i\ \ l_i\end{array}}{op(\overline{t})\ \ k_0\ \ l_0\ \rightsquigarrow\ s_1; \ldots; s_n\ \ e_h.op(del(\overline{e}, h))\ \ k_n\ \ l_n}$$

$$\frac{\begin{array}{c}op \in Op_{\mathrm{U}} \\ \tau(op) = \overline{ty} \to ty \\ \overline{ty} \cap Ty_{\mathrm{U}} = \emptyset \\ ty \in Ty_{\mathrm{U}} \\ \forall i.\ \ t_i\ \ k_{i-1}\ \ l_{i-1}\ \rightsquigarrow\ s_i\ \ e_i\ \ k_i\ \ l_i\end{array}}{op(\overline{t})\ \ k_0\ \ l_0\ \rightsquigarrow\ s_1; \ldots; s_n\ \ ty.op(\overline{e})\ \ k_n\ \ l_n}$$

$$\frac{\begin{array}{c}op \in Op_{\mathrm{U}} \\ \tau(op) = \overline{ty} \to ty \\ (\overline{ty}, ty) \cap Ty_{\mathrm{U}} = \emptyset \\ ut(\overline{ty} \to ty) = (ty', \ldots) \\ \forall i.\ \ t_i\ \ k_{i-1}\ \ l_{i-1}\ \rightsquigarrow\ s_i\ \ e_i\ \ k_i\ \ l_i\end{array}}{op(\overline{t})\ \ k_0\ \ l_0\ \rightsquigarrow\ s_1; \ldots; s_n\ \ ty'.op(\overline{e})\ \ k_n\ \ l_n}$$

$$\frac{\begin{array}{c} op \in Op_{\mathrm{U}} \\ \tau(op) = \overline{ty} \to ty \\ (\overline{ty}, ty) \cap Ty_{\mathrm{U}} = \emptyset \\ ut(\overline{ty} \to ty) = \epsilon \\ \forall i.\ \ t_i \ \ k_{i-1} \ \ l_{i-1} \ \leadsto \ s_i \ \ e_i \ \ k_i \ \ l_i \end{array}}{op(\overline{t}) \ \ k_0 \ \ l_0 \ \leadsto \ s_1; \ldots; s_n \ \ \mathsf{prim}.op(\overline{e}) \ \ k_n \ \ l_n}$$

$$\frac{\begin{array}{c} op \in Op_{\mathrm{U}} \\ \tau(op) = \overline{ty} \to ty \\ h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\} \\ mth = \langle \mathsf{apply}, tt(\overline{ty}), tt(ty), \overline{\mathsf{arg}}, (\mathtt{return}\ \mathsf{arg}_h.op(del(\overline{\mathsf{arg}}, h))) \rangle \end{array}}{op \ \ k \ \ l \ \leadsto \ \mathtt{mts} \ \ (\mathtt{new}\ \mathsf{arrow}_{ty}^{\overline{ty}}\ \{mth\}) \ \ k \ \ l}$$

$$\frac{\begin{array}{c} op \in Op_{\mathrm{U}} \\ \tau(op) = \overline{ty} \to ty \\ \overline{ty} \cap Ty_{\mathrm{U}} = \emptyset \\ ty \in Ty_{\mathrm{U}} \\ mth = \langle \mathsf{apply}, tt(\overline{ty}), ty, \overline{\mathsf{arg}}, (\mathtt{return}\ ty.op(\overline{\mathsf{arg}})) \rangle \end{array}}{op \ \ k \ \ l \ \leadsto \ \mathtt{mts} \ \ (\mathtt{new}\ \mathsf{arrow}_{ty}^{\overline{ty}}\ \{mth\}) \ \ k \ \ l}$$

$$\frac{\begin{array}{c} op \in Op_{\mathrm{U}} \\ \tau(op) = \overline{ty} \to ty \\ (\overline{ty}, ty) \cap Ty_{\mathrm{U}} = \emptyset \\ ut(\overline{ty} \to ty) = (ty', \ldots) \\ mth = \langle \mathsf{apply}, tt(\overline{ty}), tt(ty), \overline{\mathsf{arg}}, (\mathtt{return}\ ty'.op(\overline{\mathsf{arg}})) \rangle \end{array}}{op \ \ k \ \ l \ \leadsto \ \mathtt{mts} \ \ (\mathtt{new}\ \mathsf{arrow}_{ty}^{\overline{ty}}\ \{mth\}) \ \ k \ \ l}$$

$$\frac{\begin{array}{c} op \in Op_{\mathrm{U}} \\ \tau(op) = \overline{ty} \to ty \\ (\overline{ty}, ty) \cap Ty_{\mathrm{U}} = \emptyset \\ ut(\overline{ty} \to ty) = \epsilon \\ mth = \langle \mathsf{apply}, tt(\overline{ty}), tt(ty), \overline{\mathsf{arg}}, (\mathtt{return}\ \mathsf{prim}.op(\overline{\mathsf{arg}})) \rangle \end{array}}{op \ \ k \ \ l \ \leadsto \ \mathtt{mts} \ \ (\mathtt{new}\ \mathsf{arrow}_{ty}^{\overline{ty}}\ \{mth\}) \ \ k \ \ l}$$

$$\frac{\begin{array}{c} \Delta(ty) = \prod_i p_i \ \ ty_i \\ \forall i.\ \ t_i \ \ k_{i-1} \ \ l_{i-1} \ \leadsto \ s_i \ \ e_i \ \ k_i \ \ l_i \end{array}}{\{p_i \leftarrow t_i\}_i \ \ k_0 \ \ l_0 \ \leadsto \ s_1; \ldots; s_n \ \ (\mathtt{new}\ ty(\overline{e})) \ \ k_n \ \ l_n}$$

$$\frac{\begin{array}{c} \Delta(ty) = \prod_i p_i \ \ ty_i \\ t \ \ k \ \ l \ \leadsto \ s \ \ e \ \ k' \ \ l' \end{array}}{p_i(t) \ \ k \ \ l \ \leadsto \ s \ \ e.p_i \ \ k' \ \ l'}$$

$$\frac{\Delta(ty) = \prod_i p_i \;\; ty_i \qquad mth = \langle \mathsf{apply}, ty, tt(ty_i), \mathsf{arg}_1, (\texttt{return}\ \mathsf{arg}_1.p_i)\rangle}{p_i \;\; k \;\; l \rightsquigarrow \texttt{mts} \;\; (\texttt{new}\ \mathsf{arrow}^{ty}_{ty_i}\ \{mth\}) \;\; k \;\; l}$$

$$\frac{\Delta(ty) = \sum_i c_i \;\; \overline{ty}_i \qquad \overline{ty}_i = \epsilon}{c_i \;\; k \;\; l \rightsquigarrow \texttt{mts} \;\; ty.c_i \;\; k \;\; l}$$

$$\frac{\Delta(ty) = \sum_i c_i \;\; \overline{ty}_i \qquad \overline{ty}_i \neq \epsilon \qquad \forall j.\;\; t_j \;\; k_{j-1} \;\; l_{j-1} \rightsquigarrow s_j \;\; e_j \;\; k_j \;\; l_j}{c_i(\overline{t}) \;\; k_0 \;\; l_0 \rightsquigarrow s_1; \ldots ; s_m \;\; ty.c_i(\overline{e}) \;\; k_m \;\; l_m}$$

$$\frac{\Delta(ty) = \sum_i c_i \;\; \overline{ty}_i \qquad \overline{ty}_i \neq \epsilon \qquad mth = \langle \mathsf{apply}, tt(\overline{ty}), ty, \overline{\mathsf{arg}}, (\texttt{return}\ ty.c_i(\overline{\mathsf{arg}}))\rangle}{c_i \;\; k \;\; l \rightsquigarrow \texttt{mts} \;\; (\texttt{new}\ \mathsf{arrow}^{\overline{ty}_i}_{ty}\ \{mth\}) \;\; k \;\; l}$$

$$\frac{t \;\; k \;\; l \rightsquigarrow s \;\; e \;\; k' \;\; l'}{\mathsf{restr}_{ty}(t) \;\; k \;\; l \rightsquigarrow s \;\; (\texttt{new}\ ty(e)) \;\; k' \;\; l'}$$

$$\frac{t \;\; k \;\; l \rightsquigarrow s \;\; e \;\; k' \;\; l'}{\mathsf{relax}_{ty}(t) \;\; k \;\; l \rightsquigarrow s \;\; (e.\mathsf{relax}) \;\; k' \;\; l'}$$

$$\frac{t \;\; k \;\; l \rightsquigarrow s \;\; e \;\; k' \;\; l'}{\mathsf{quot}_{ty}(t) \;\; k \;\; l \rightsquigarrow s \;\; (\texttt{new}\ ty(e)) \;\; k' \;\; l'}$$

$$\frac{t \;\; k \;\; l \rightsquigarrow s \;\; e \;\; k' \;\; l'}{\mathsf{choo}_{ty}(t) \;\; k \;\; l \rightsquigarrow s \;\; (e.\mathsf{choose}) \;\; k' \;\; l'}$$

$$\frac{\Delta(ty) = ty_0|r \qquad mth = \langle \mathsf{apply}, tt(ty_0), ty, \mathsf{arg}_1, (\texttt{return}\ (\texttt{new}\ ty(\mathsf{arg}_1)))\rangle}{\mathsf{restr}_{ty} \;\; k \;\; l \rightsquigarrow \texttt{mts} \;\; (\texttt{new}\ \mathsf{arrow}^{ty_0}_{ty}\ \{mth\}) \;\; k \;\; l}$$

$$\frac{\Delta(ty) = ty_0|r \qquad mth = \langle \mathsf{apply}, ty, tt(ty_0), \mathsf{arg}_1, (\texttt{return}\ (\mathsf{arg}_1.\mathsf{relax}))\rangle}{\mathsf{relax}_{ty} \;\; k \;\; l \rightsquigarrow \texttt{mts} \;\; (\texttt{new}\ \mathsf{arrow}^{ty}_{ty_0}\ \{mth\}) \;\; k \;\; l}$$

$$\frac{\begin{array}{c}\Delta(ty) = ty_0/q \\ mth = \langle \mathsf{apply}, tt(ty_0), ty, \mathsf{arg}_1, (\mathtt{return}\ (\mathtt{new}\ ty(\mathsf{arg}_1)))\rangle\end{array}}{\mathsf{quot}_{ty}\ \ k\ \ l\ \rightsquigarrow\ \mathtt{mts}\ \ (\mathtt{new}\ \mathsf{arrow}_{ty}^{ty_0}\ \{mth\})\ \ k\ \ l}$$

$$\frac{\begin{array}{c}\Delta(ty) = ty_0/q \\ mth = \langle \mathsf{apply}, ty, tt(ty_0), \mathsf{arg}_1, (\mathtt{return}\ \mathsf{arg}_1.\mathsf{choose})\rangle\end{array}}{\mathsf{choo}_{ty}\ \ k\ \ l\ \rightsquigarrow\ \mathtt{mts}\ \ (\mathtt{new}\ \mathsf{arrow}_{ty_0}^{ty}\ \{mth\})\ \ k\ \ l}$$

$$\frac{\begin{array}{c}t \notin Op \\ t\ \ k\ \ l\ \rightsquigarrow\ s\ \ e\ \ k_0\ \ l_0 \\ \forall i.\ \ t_i\ \ k_{i-1}\ \ l_{i-1}\ \rightsquigarrow\ s_i\ \ e_i\ \ k_i\ \ l_i\end{array}}{t(\overline{t})\ \ k\ \ l\ \rightsquigarrow\ s; s_1; \ldots; s_n\ \ e.\mathsf{apply}(\overline{e})\ \ k_n\ \ l_n}$$

$$\frac{\begin{array}{c}(w_1, \ldots, w_p) = order(FV(t) - \overline{v}) \\ \forall i.\ \ s_i = \begin{cases}(\mathtt{fin}\ cx(w_i)\ \mathsf{fin}_{w_i} = tc(w_i)) & \text{if}\quad w_i \in \mathcal{D}(tc) \\ (\mathtt{fin}\ cx(w_i)\ \mathsf{fin}_{w_i} = w_i) & \text{otherwise}\end{cases} \\ t\ \ 1\ \ 1\ \rightsquigarrow_{\underline{\mathrm{Ret}}}^{\{\overline{w} \mapsto \mathsf{fin}_{\overline{w}}\}, \{\overline{v} \mapsto \overline{ty}\}}\ \ s\ \ k'\ \ l' \\ e = (\mathtt{new}\ \mathsf{arrow}_{ty}^{\overline{ty}}\ \{\langle \mathsf{apply}, tt(\overline{ty}), tt(ty), \overline{v}, s\rangle\})\end{array}}{\lambda \overline{v} : \overline{ty}. t : ty\ \ k\ \ l\ \rightsquigarrow\ s_1; \ldots; s_p\ \ e\ \ k\ \ l}$$

$$\frac{\begin{array}{c}t_1, t_2 \in T_{ty}^{cx} \\ t_1\ \ k_0\ \ l_0\ \rightsquigarrow\ s_1\ \ e_1\ \ k_1\ \ l_1 \\ t_2\ \ k_1\ \ l_1\ \rightsquigarrow\ s_2\ \ e_2\ \ k_2\ \ l_2\end{array}}{(t_1 = t_2)\ \ k_0\ \ l_0\ \rightsquigarrow\ s_1; s_2\ \ eq_{ty}(e_1, e_2)\ \ k_2\ \ l_2}$$

$$\frac{\begin{array}{c}t_1, t_2 \in T_{ty}^{cx} \\ t_1\ \ k_0\ \ l_0\ \rightsquigarrow\ s_1\ \ e_1\ \ k_1\ \ l_1 \\ t_2\ \ k_1\ \ l_1\ \rightsquigarrow\ s_2\ \ e_2\ \ k_2\ \ l_2\end{array}}{(t_1 \neq t_2)\ \ k_0\ \ l_0\ \rightsquigarrow\ s_1; s_2\ \ ineq_{ty}(e_1, e_2)\ \ k_2\ \ l_2}$$

$$\frac{\begin{array}{c}t_0\ \ k\ \ l\ \rightsquigarrow\ s_0\ \ e_0\ \ k'\ \ l' \\ t_1\ \ k'\ \ l'\ \rightsquigarrow\ \mathtt{mts}\ \ e_1\ \ k'\ \ l' \\ t_2\ \ k'\ \ l'\ \rightsquigarrow\ \mathtt{mts}\ \ e_2\ \ k'\ \ l'\end{array}}{(\mathbf{if}\ t_0\ t_1\ t_2)\ \ k\ \ l\ \rightsquigarrow\ s_0\ \ (e_0\ ?\ e_1 : e_2)\ \ k'\ \ l'}$$

$$
\frac{\begin{array}{c}
t_1, t_2 \in T^{cx}_{ty} \\
t_0 \quad (k+1) \quad l \;\leadsto\; s_0 \quad e_0 \quad k_0 \quad l_0 \\
t_1 \quad k_0 \quad l_0 \;\leadsto_{\mathrm{AsgV}(\mathsf{ires}_k)}\; s_1 \quad k_1 \quad l_1 \\
t_2 \quad k_1 \quad l_1 \;\leadsto_{\mathrm{AsgV}(\mathsf{ires}_k)}\; s_2 \quad k_2 \quad l_2 \\
s = (tt(ty)\ \mathsf{ires}_k); s_0; (\texttt{if}\ (e_0)\ s_1\ \texttt{else}\ s_2) \\
t_1 \quad k \quad l \;\leadsto\; s'_1 \quad e'_1 \quad k'_1 \quad l'_1 \\
t_2 \quad k \quad l \;\leadsto\; s'_2 \quad e'_2 \quad k'_2 \quad l'_2 \\
s'_1 \neq \texttt{mts} \ \vee\ s'_2 \neq \texttt{mts}
\end{array}}{(\textbf{if}\ t_0\ t_1\ t_2) \quad k \quad l \;\leadsto\; s \quad \mathsf{ires}_k \quad k_2 \quad l_2}
$$

$$
\frac{\begin{array}{c}
t_0 \in T^{cx}_{ty_0} \\
t_0 \quad k \quad l \;\leadsto_{\mathrm{AsgNV}(tt(ty_0), v)}\; s_0 \quad k_0 \quad l_0 \\
t \quad k_0 \quad l_0 \;\leadsto^{tc, cx[v \mapsto ty_0]}\; s \quad e \quad k' \quad l'
\end{array}}{(\textbf{let}\ v \leftarrow t_0\ \textbf{in}\ t) \quad k \quad l \;\leadsto\; s_0; s \quad e \quad k' \quad l'}
$$

$$
\frac{\begin{array}{c}
t \in T^{cx}_{ty} \\
\Delta(ty) = \sum_i c_i\ \overline{ty}_i \\
t_i \in T^{cx[\overline{v}_i \mapsto \overline{ty}_i]}_{ty_0} \\
(\forall i.\ \ \overline{v}_i \cap FV(t_i) = \emptyset) \ \vee\ t \in V \\
t \quad k \quad (l+1) \;\leadsto\; s \quad e \quad k_{i_0} \quad l_{i_0} \\
\forall j.\ \ t_{i_j} \quad k_{i_{j-1}} \quad l_{i_{j-1}} \;\leadsto^{tc[\overline{v}_{i_j} \mapsto \mathsf{sub}_l^{c_{i_j}}.\overline{\mathsf{arg}}],\, cx[\overline{v}_{i_j} \mapsto \overline{ty}_{i_j}]}_{\mathrm{AsgV}(\mathsf{cres}_l)}\; s_{i_j} \quad k_{i_j} \quad l_{i_j} \\
\forall i.\ \ s'_i = \begin{cases} (\mathsf{sumd}^{ty}_{c_i}\ \mathsf{sub}^{c_i}_l = ((\mathsf{sumd}^{ty}_{c_i})\ e)) & \text{if} \quad \overline{v}_i \cap FV(t_i) \neq \emptyset \\ \texttt{mts} & \text{otherwise} \end{cases} \\
s_0 = \begin{cases} (\texttt{if}\ (e.\mathsf{tag} > ty.\mathsf{tagc}_{c_n})\ \texttt{throwmalf}\ \texttt{else}\ \texttt{throwunexp}) & \\ \qquad \text{if} \quad \{i_1, \ldots, i_p\} \neq \{1, \ldots, n\} & \\ \texttt{throwmalf} \qquad \text{otherwise} & \end{cases} \\
s' = (tt(ty_0)\ \mathsf{cres}_l); s; (\texttt{switch}(e.\mathsf{tag})\ \{ty.\mathsf{tagc}_{c_i} \to (s'_i; s_i)\}_i\ s_0)
\end{array}}{(\textbf{case}\ t\ \{c_i(\overline{v}_i) \to t_i\}_i) \quad k \quad l \;\leadsto\; s' \quad \mathsf{cres}_l \quad k_{i_p} \quad l_{i_p}}
$$

$$
\frac{\begin{array}{c}
t \in T^{cx}_{ty} \\
\Delta(ty) = \sum_i c_i\ \overline{ty}_i \\
t_i \in T^{cx[\overline{v}_i \mapsto \overline{ty}_i]}_{ty_0} \\
(\exists i.\ \ \overline{v}_i \cap FV(t_i) \neq \emptyset) \ \wedge\ t \notin V \\
t \quad k \quad (l+1) \;\leadsto_{\mathrm{AsgNV}(ty, \mathsf{tgt}_l)}\; s \quad k_{i_0} \quad l_{i_0} \\
\forall j.\ \ t_{i_j} \quad k_{i_{j-1}} \quad l_{i_{j-1}} \;\leadsto^{tc[\overline{v}_{i_j} \mapsto \mathsf{sub}_l^{c_{i_j}}.\overline{\mathsf{arg}}],\, cx[\overline{v}_{i_j} \mapsto \overline{ty}_{i_j}]}_{\mathrm{AsgV}(\mathsf{cres}_l)}\; s_{i_j} \quad k_{i_j} \quad l_{i_j} \\
\forall i.\ \ s'_i = \begin{cases} (\mathsf{sumd}^{ty}_{c_i}\ \mathsf{sub}^{c_i}_l = ((\mathsf{sumd}^{ty}_{c_i})\ \mathsf{tgt}_l)) & \text{if} \quad \overline{v}_i \cap FV(t_i) \neq \emptyset \\ \texttt{mts} & \text{otherwise} \end{cases} \\
s_0 = \begin{cases} (\texttt{if}\ (\mathsf{tgt}_l.\mathsf{tag} > ty.\mathsf{tagc}_{c_n})\ \texttt{throwmalf}\ \texttt{else}\ \texttt{throwunexp}) & \\ \qquad \text{if} \quad \{i_1, \ldots, i_p\} \neq \{1, \ldots, n\} & \\ \texttt{throwmalf} \qquad \text{otherwise} & \end{cases} \\
s' = (tt(ty_0)\ \mathsf{cres}_l); s; (\texttt{switch}(\mathsf{tgt}_l.\mathsf{tag})\ \{ty.\mathsf{tagc}_{c_i} \to (s'_i; s_i)\}_i\ s_0)
\end{array}}{(\textbf{case}\ t\ \{c_i(\overline{v}_i) \to t_i\}_i) \quad k \quad l \;\leadsto\; s' \quad \mathsf{cres}_l \quad k_{i_p} \quad l_{i_p}}
$$

$$
\frac{\begin{array}{c}
t_0 \;\; k \;\; l \;\leadsto\; s_0 \;\; e_0 \;\; k_0 \;\; l_0 \\
t_1 \;\; k_0 \;\; l_0 \;\leadsto_{\mathrm{Ret}}\; s_1 \;\; k_1 \;\; l_1 \\
t_2 \;\; k_1 \;\; l_1 \;\leadsto_{\mathrm{Ret}}\; s_2 \;\; k_2 \;\; l_2
\end{array}}{(\textbf{if}\ t_0\ t_1\ t_2) \;\; k \;\; l \;\leadsto_{\mathrm{Ret}}\; s_0; (\texttt{if}\ (e_0)\ s_1\ \texttt{else}\ s_2) \;\; k_2 \;\; l_2}
$$

$$
\frac{\begin{array}{c}
t \in T^{cx}_{ty} \\
\Delta(ty) = \sum_i c_i\ \overline{ty}_i \\
t_i \in T^{cx[\overline{v}_i \mapsto \overline{ty}_i]}_{ty_0} \\
(\forall i.\ \ \overline{v}_i \cap FV(t_i) = \emptyset) \ \vee\ t \in V \\
t \;\; k \;\; (l+1) \;\leadsto\; s \;\; e \;\; k_{i_0} \;\; l_{i_0} \\
\forall j.\ \ t_{i_j} \;\; k_{i_{j-1}} \;\; l_{i_{j-1}} \;\leadsto^{tc[\overline{v}_{i_j} \mapsto \mathsf{sub}_l^{c_{i_j}}.\overline{\mathsf{arg}}],\, cx[\overline{v}_{i_j} \mapsto \overline{ty}_{i_j}]}_{\mathrm{Ret}} \;\; s_{i_j} \;\; k_{i_j} \;\; l_{i_j} \\
\forall i.\ \ s'_i = \begin{cases} (\mathsf{sumd}^{ty}_{c_i}\ \mathsf{sub}^{c_i}_l = ((\mathsf{sumd}^{ty}_{c_i})\ e)) & \text{if} \quad \overline{v}_i \cap FV(t_i) \neq \emptyset \\ \texttt{mts} & \text{otherwise} \end{cases} \\
s_0 = \begin{cases} (\texttt{if}\ (e.\mathsf{tag} > ty.\mathsf{tagc}_{c_n})\ \texttt{throwmalf}\ \texttt{else}\ \texttt{throwunexp}) \\ \qquad \text{if} \quad \{i_1, \ldots, i_p\} \neq \{1, \ldots, n\} \\ \texttt{throwmalf} \qquad \text{otherwise} \end{cases} \\
s' = (tt(ty_0)\ \mathsf{cres}_l); s; (\texttt{switch}(e.\mathsf{tag})\ \{ty.\mathsf{tagc}_{c_i} \to (s'_i; s_i)\}_i\ s_0)
\end{array}}{(\textbf{case}\ t\ \{c_i(\overline{v}_i) \to t_i\}_i) \;\; k \;\; l \;\leadsto_{\mathrm{Ret}}\; s' \;\; k_{i_p} \;\; l_{i_p}}
$$

$$
\frac{\begin{array}{c}
t \in T^{cx}_{ty} \\
\Delta(ty) = \sum_i c_i\ \overline{ty}_i \\
t_i \in T^{cx[\overline{v}_i \mapsto \overline{ty}_i]}_{ty_0} \\
(\exists i.\ \ \overline{v}_i \cap FV(t_i) \neq \emptyset) \ \wedge\ t \notin V \\
t \;\; k \;\; (l+1) \;\leadsto_{\mathrm{AsgNV}(ty, \mathsf{tgt}_l)}\; s \;\; k_{i_0} \;\; l_{i_0} \\
\forall j.\ \ t_{i_j} \;\; k_{i_{j-1}} \;\; l_{i_{j-1}} \;\leadsto^{tc[\overline{v}_{i_j} \mapsto \mathsf{sub}_l^{c_{i_j}}.\overline{\mathsf{arg}}],\, cx[\overline{v}_{i_j} \mapsto \overline{ty}_{i_j}]}_{\mathrm{Ret}} \;\; s_{i_j} \;\; k_{i_j} \;\; l_{i_j} \\
\forall i.\ \ s'_i = \begin{cases} (\mathsf{sumd}^{ty}_{c_i}\ \mathsf{sub}^{c_i}_l = ((\mathsf{sumd}^{ty}_{c_i})\ \mathsf{tgt}_l)) & \text{if} \quad \overline{v}_i \cap FV(t_i) \neq \emptyset \\ \texttt{mts} & \text{otherwise} \end{cases} \\
s_0 = \begin{cases} (\texttt{if}\ (\mathsf{tgt}_l.\mathsf{tag} > ty.\mathsf{tagc}_{c_n})\ \texttt{throwmalf}\ \texttt{else}\ \texttt{throwunexp}) \\ \qquad \text{if} \quad \{i_1, \ldots, i_p\} \neq \{1, \ldots, n\} \\ \texttt{throwmalf} \qquad \text{otherwise} \end{cases} \\
s' = (tt(ty_0)\ \mathsf{cres}_l); s; (\texttt{switch}(\mathsf{tgt}_l.\mathsf{tag})\ \{ty.\mathsf{tagc}_{c_i} \to (s'_i; s_i)\}_i\ s_0)
\end{array}}{(\textbf{case}\ t\ \{c_i(\overline{v}_i) \to t_i\}_i) \;\; k \;\; l \;\leadsto_{\mathrm{Ret}}\; s' \;\; k_{i_p} \;\; l_{i_p}}
$$

$$
\frac{\begin{array}{c}
t \neq (\textbf{if}\ \ldots) \ \wedge\ t \neq (\textbf{case}\ \ldots) \\
t \;\; k \;\; l \;\leadsto\; s \;\; e \;\; k' \;\; l'
\end{array}}{t \;\; k \;\; l \;\leadsto_{\mathrm{Ret}}\; s; (\texttt{return}\ e) \;\; k' \;\; l'}
$$

$$
\frac{\begin{array}{c}
t_0 \;\; k \;\; l \;\leadsto\; s_0 \;\; e_0 \;\; k_0 \;\; l_0 \\
t_1 \;\; k_0 \;\; l_0 \;\leadsto_{\mathrm{AsgV}(v)}\; s_1 \;\; k_1 \;\; l_1 \\
t_2 \;\; k_1 \;\; l_1 \;\leadsto_{\mathrm{AsgV}(v)}\; s_2 \;\; k_2 \;\; l_2
\end{array}}{(\textbf{if}\ t_0\ t_1\ t_2) \;\; k \;\; l \;\leadsto_{\mathrm{AsgNV}(ty, v)}\; (ty\ v); s_0; (\texttt{if}\ (e_0)\ s_1\ \texttt{else}\ s_2) \;\; k_2 \;\; l_2}
$$

$$\frac{\begin{gathered} t \in T^{cx}_{ty} \\ \Delta(ty) = \sum_i c_i\ \overline{ty}_i \\ t_i \in T^{cx[\overline{v}_i \mapsto \overline{ty}_i]}_{ty_0} \\ (\forall i.\ \ \overline{v}_i \cap FV(t_i) = \emptyset)\ \vee\ t \in V \\ t\ \ k\ \ (l+1)\ \rightsquigarrow\ s\ \ e\ \ k_{i_0}\ \ l_{i_0} \\ \forall j.\ \ t_{i_j}\ \ k_{i_{j-1}}\ \ l_{i_{j-1}}\ \rightsquigarrow^{tc[\overline{v}_{i_j} \mapsto \mathsf{sub}_l^{c_{i_j}}.\overline{\mathsf{arg}}],\, cx[\overline{v}_{i_j} \mapsto \overline{ty}_{i_j}]}_{\mathrm{AsgNV}(ty_0,v)}\ s_{i_j}\ \ k_{i_j}\ \ l_{i_j} \\ \forall i.\ \ s'_i = \begin{cases} (\mathsf{sumd}^{ty}_{c_i}\ \mathsf{sub}^{c_i}_l = ((\mathsf{sumd}^{ty}_{c_i})\ e)) & \text{if}\ \ \overline{v}_i \cap FV(t_i) \neq \emptyset \\ \mathtt{mts} & \text{otherwise}\end{cases} \\ s_0 = \begin{cases} (\mathtt{if}\ (e.\mathsf{tag} > ty.\mathsf{tagc}_{c_n})\ \mathtt{throwmalf\ else\ throwunexp}) \\ \qquad \text{if}\ \ \{i_1,\ldots,i_p\} \neq \{1,\ldots,n\} \\ \mathtt{throwmalf} \qquad \text{otherwise} \end{cases} \\ s' = (tt(ty_0)\ \mathsf{cres}_l); s; (\mathtt{switch}(e.\mathsf{tag})\ \{ty.\mathsf{tagc}_{c_i} \to (s'_i; s_i)\}_i\ s_0) \end{gathered}}{(\textbf{case}\ t\ \{c_i(\overline{v}_i) \to t_i\}_i)\ \ k\ \ l\ \rightsquigarrow_{\mathrm{AsgNV}(ty_0,v)}\ s'\ \ k_{i_p}\ \ l_{i_p}}$$

$$\frac{\begin{gathered} t \in T^{cx}_{ty} \\ \Delta(ty) = \sum_i c_i\ \overline{ty}_i \\ t_i \in T^{cx[\overline{v}_i \mapsto \overline{ty}_i]}_{ty_0} \\ (\exists i.\ \ \overline{v}_i \cap FV(t_i) \neq \emptyset)\ \wedge\ t \notin V \\ t\ \ k\ \ (l+1)\ \rightsquigarrow_{\mathrm{AsgNV}(ty,\mathsf{tgt}_l)}\ s\ \ k_{i_0}\ \ l_{i_0} \\ \forall j.\ \ t_{i_j}\ \ k_{i_{j-1}}\ \ l_{i_{j-1}}\ \rightsquigarrow^{tc[\overline{v}_{i_j} \mapsto \mathsf{sub}_l^{c_{i_j}}.\overline{\mathsf{arg}}],\, cx[\overline{v}_{i_j} \mapsto \overline{ty}_{i_j}]}_{\mathrm{AsgNV}(ty_0,v)}\ s_{i_j}\ \ k_{i_j}\ \ l_{i_j} \\ \forall i.\ \ s'_i = \begin{cases} (\mathsf{sumd}^{ty}_{c_i}\ \mathsf{sub}^{c_i}_l = ((\mathsf{sumd}^{ty}_{c_i})\ \mathsf{tgt}_l)) & \text{if}\ \ \overline{v}_i \cap FV(t_i) \neq \emptyset \\ \mathtt{mts} & \text{otherwise}\end{cases} \\ s_0 = \begin{cases} (\mathtt{if}\ (\mathsf{tgt}_l.\mathsf{tag} > ty.\mathsf{tagc}_{c_n})\ \mathtt{throwmalf\ else\ throwunexp}) \\ \qquad \text{if}\ \ \{i_1,\ldots,i_p\} \neq \{1,\ldots,n\} \\ \mathtt{throwmalf} \qquad \text{otherwise} \end{cases} \\ s' = (tt(ty_0)\ \mathsf{cres}_l); s; (\mathtt{switch}(\mathsf{tgt}_l.\mathsf{tag})\ \{ty.\mathsf{tagc}_{c_i} \to (s'_i; s_i)\}_i\ s_0) \end{gathered}}{(\textbf{case}\ t\ \{c_i(\overline{v}_i) \to t_i\}_i)\ \ k\ \ l\ \rightsquigarrow_{\mathrm{AsgNV}(ty_0,v)}\ s'\ \ k_{i_p}\ \ l_{i_p}}$$

$$\frac{\begin{gathered} t \neq (\textbf{if}\ \ldots)\ \wedge\ t \neq (\textbf{case}\ \ldots) \\ t\ \ k\ \ l\ \rightsquigarrow\ s\ \ e\ \ k'\ \ l' \end{gathered}}{t\ \ k\ \ l\ \rightsquigarrow_{\mathrm{AsgNV}(ty,v)}\ s; (ty\ v = e)\ \ k'\ \ l'}$$

$$\frac{\begin{gathered} t_0\ \ k\ \ l\ \rightsquigarrow\ s_0\ \ e_0\ \ k_0\ \ l_0 \\ t_1\ \ k_0\ \ l_0\ \rightsquigarrow_{\mathrm{AsgV}(v)}\ s_1\ \ k_1\ \ l_1 \\ t_2\ \ k_1\ \ l_1\ \rightsquigarrow_{\mathrm{AsgV}(v)}\ s_2\ \ k_2\ \ l_2 \end{gathered}}{(\textbf{if}\ t_0\ t_1\ t_2)\ \ k\ \ l\ \rightsquigarrow_{\mathrm{AsgV}(v)}\ s_0; (\mathtt{if}\ (e_0)\ s_1\ \mathtt{else}\ s_2)\ \ k_2\ \ l_2}$$

$$\frac{\begin{array}{c} t \in T^{cx}_{ty} \\ \Delta(ty) = \sum_i c_i\ \overline{ty}_i \\ t_i \in T^{cx[\overline{v}_i \mapsto \overline{ty}_i]}_{ty_0} \\ (\forall i.\ \ \overline{v}_i \cap FV(t_i) = \emptyset)\ \vee\ t \in V \\ t\ \ k\ \ (l+1)\ \rightsquigarrow\ s\ \ e\ \ k_{i_0}\ \ l_{i_0} \\ \forall j.\ \ t_{i_j}\ \ k_{i_{j-1}}\ \ l_{i_{j-1}}\ \rightsquigarrow^{tc[\overline{v}_{i_j} \mapsto \mathsf{sub}^{c_{i_j}}_l.\overline{\mathsf{arg}}],cx[\overline{v}_{i_j} \mapsto \overline{ty}_{i_j}]}_{\mathrm{AsgV}(v)}\ \ s_{i_j}\ \ k_{i_j}\ \ l_{i_j} \\ \forall i.\ \ s'_i = \begin{cases} (\mathsf{sumd}^{ty}_{c_i}\ \mathsf{sub}^{c_i}_l = ((\mathsf{sumd}^{ty}_{c_i})\ e)) & \text{if}\quad \overline{v}_i \cap FV(t_i) \neq \emptyset \\ \mathtt{mts} & \text{otherwise} \end{cases} \\ s_0 = \begin{cases} (\mathtt{if}\ (e.\mathsf{tag} > ty.\mathsf{tagc}_{c_n})\ \mathtt{throwmalf}\ \mathtt{else}\ \mathtt{throwunexp}) \\ \qquad \text{if}\quad \{i_1, \ldots, i_p\} \neq \{1, \ldots, n\} \\ \mathtt{throwmalf} \qquad \text{otherwise} \end{cases} \\ s' = (tt(ty_0)\ \mathsf{cres}_l); s; (\mathtt{switch}(e.\mathsf{tag})\ \{ty.\mathsf{tagc}_{c_i} \to (s'_i; s_i)\}_i\ s_0) \end{array}}{(\mathbf{case}\ t\ \{c_i(\overline{v}_i) \to t_i\}_i)\ \ k\ \ l\ \rightsquigarrow_{\mathrm{AsgV}(v)}\ s'\ \ k_{i_p}\ \ l_{i_p}}$$

$$\frac{\begin{array}{c} t \in T^{cx}_{ty} \\ \Delta(ty) = \sum_i c_i\ \overline{ty}_i \\ t_i \in T^{cx[\overline{v}_i \mapsto \overline{ty}_i]}_{ty_0} \\ (\exists i.\ \ \overline{v}_i \cap FV(t_i) \neq \emptyset)\ \wedge\ t \notin V \\ t\ \ k\ \ (l+1)\ \rightsquigarrow_{\mathrm{AsgNV}(ty, \mathsf{tgt}_l)}\ s\ \ k_{i_0}\ \ l_{i_0} \\ \forall j.\ \ t_{i_j}\ \ k_{i_{j-1}}\ \ l_{i_{j-1}}\ \rightsquigarrow^{tc[\overline{v}_{i_j} \mapsto \mathsf{sub}^{c_{i_j}}_l.\overline{\mathsf{arg}}],cx[\overline{v}_{i_j} \mapsto \overline{ty}_{i_j}]}_{\mathrm{AsgV}(v)}\ \ s_{i_j}\ \ k_{i_j}\ \ l_{i_j} \\ \forall i.\ \ s'_i = \begin{cases} (\mathsf{sumd}^{ty}_{c_i}\ \mathsf{sub}^{c_i}_l = ((\mathsf{sumd}^{ty}_{c_i})\ \mathsf{tgt}_l)) & \text{if}\quad \overline{v}_i \cap FV(t_i) \neq \emptyset \\ \mathtt{mts} & \text{otherwise} \end{cases} \\ s_0 = \begin{cases} (\mathtt{if}\ (\mathsf{tgt}_l.\mathsf{tag} > ty.\mathsf{tagc}_{c_n})\ \mathtt{throwmalf}\ \mathtt{else}\ \mathtt{throwunexp}) \\ \qquad \text{if}\quad \{i_1, \ldots, i_p\} \neq \{1, \ldots, n\} \\ \mathtt{throwmalf} \qquad \text{otherwise} \end{cases} \\ s' = (tt(ty_0)\ \mathsf{cres}_l); s; (\mathtt{switch}(\mathsf{tgt}_l.\mathsf{tag})\ \{ty.\mathsf{tagc}_{c_i} \to (s'_i; s_i)\}_i\ s_0) \end{array}}{(\mathbf{case}\ t\ \{c_i(\overline{v}_i) \to t_i\}_i)\ \ k\ \ l\ \rightsquigarrow_{\mathrm{AsgV}(v)}\ s'\ \ k_{i_p}\ \ l_{i_p}}$$

$$\frac{\begin{array}{c} t \neq (\mathbf{if}\ \ldots)\ \wedge\ t \neq (\mathbf{case}\ \ldots) \\ t\ \ k\ \ l\ \rightsquigarrow\ s\ \ e\ \ k'\ \ l' \end{array}}{t\ \ k\ \ l\ \rightsquigarrow_{\mathrm{AsgV}(v)}\ s; (v = e)\ \ k'\ \ l'}$$

$$\frac{\begin{array}{c} t_0\ \ k\ \ l\ \rightsquigarrow\ s_0\ \ e_0\ \ k_0\ \ l_0 \\ t_1\ \ k_0\ \ l_0\ \rightsquigarrow_{\mathrm{AsgF}(c,f)}\ s_1\ \ k_1\ \ l_1 \\ t_2\ \ k_1\ \ l_1\ \rightsquigarrow_{\mathrm{AsgF}(c,f)}\ s_2\ \ k_2\ \ l_2 \end{array}}{(\mathbf{if}\ t_0\ t_1\ t_2)\ \ k\ \ l\ \rightsquigarrow_{\mathrm{AsgF}(c,f)}\ s_0; (\mathtt{if}\ (e_0)\ s_1\ \mathtt{else}\ s_2)\ \ k_2\ \ l_2}$$

$$\frac{\begin{array}{c} t \in T^{cx}_{ty} \\ \Delta(ty) = \sum_i c_i\ \overline{ty}_i \\ t_i \in T^{cx[\overline{v}_i \mapsto \overline{ty}_i]}_{ty_0} \\ (\forall i.\ \ \overline{v}_i \cap FV(t_i) = \emptyset)\ \vee\ t \in V \\ t\ \ k\ \ (l+1)\ \rightsquigarrow\ s\ \ e\ \ k_{i_0}\ \ l_{i_0} \\ \forall j.\ \ t_{i_j}\ \ k_{i_{j-1}}\ \ l_{i_{j-1}}\ \rightsquigarrow^{tc[\overline{v}_{i_j} \mapsto \mathsf{sub}^{c_{i_j}}_l.\overline{\mathsf{arg}}], cx[\overline{v}_{i_j} \mapsto \overline{ty}_{i_j}]}_{\mathrm{AsgF}(c,f)}\ s_{i_j}\ \ k_{i_j}\ \ l_{i_j} \\ \forall i.\ \ s'_i = \begin{cases} (\mathsf{sumd}^{ty}_{c_i}\ \mathsf{sub}^{c_i}_l = ((\mathsf{sumd}^{ty}_{c_i})\ e)) & \text{if} \quad \overline{v}_i \cap FV(t_i) \neq \emptyset \\ \mathtt{mts} & \text{otherwise} \end{cases} \\ s_0 = \begin{cases} (\mathtt{if}\ (e.\mathsf{tag} > ty.\mathsf{tagc}_{c_n})\ \mathtt{throwmalf\ else\ throwunexp}) \\ \qquad \text{if} \quad \{i_1, \ldots, i_p\} \neq \{1, \ldots, n\} \\ \mathtt{throwmalf} \qquad \text{otherwise} \end{cases} \\ s' = (tt(ty_0)\ \mathsf{cres}_l); s; (\mathtt{switch}(e.\mathsf{tag})\ \{ty.\mathsf{tagc}_{c_i} \to (s'_i; s_i)\}_i\ s_0) \end{array}}{(\mathbf{case}\ t\ \{c_i(\overline{v}_i) \to t_i\}_i)\ \ k\ \ l\ \rightsquigarrow_{\mathrm{AsgF}(c,f)}\ s'\ \ k_{i_p}\ \ l_{i_p}}$$

$$\frac{\begin{array}{c} t \in T^{cx}_{ty} \\ \Delta(ty) = \sum_i c_i\ \overline{ty}_i \\ t_i \in T^{cx[\overline{v}_i \mapsto \overline{ty}_i]}_{ty_0} \\ (\exists i.\ \ \overline{v}_i \cap FV(t_i) \neq \emptyset)\ \wedge\ t \notin V \\ t\ \ k\ \ (l+1)\ \rightsquigarrow_{\mathrm{AsgNV}(ty, \mathsf{tgt}_l)}\ s\ \ k_{i_0}\ \ l_{i_0} \\ \forall j.\ \ t_{i_j}\ \ k_{i_{j-1}}\ \ l_{i_{j-1}}\ \rightsquigarrow^{tc[\overline{v}_{i_j} \mapsto \mathsf{sub}^{c_{i_j}}_l.\overline{\mathsf{arg}}], cx[\overline{v}_{i_j} \mapsto \overline{ty}_{i_j}]}_{\mathrm{AsgF}(c,f)}\ s_{i_j}\ \ k_{i_j}\ \ l_{i_j} \\ \forall i.\ \ s'_i = \begin{cases} (\mathsf{sumd}^{ty}_{c_i}\ \mathsf{sub}^{c_i}_l = ((\mathsf{sumd}^{ty}_{c_i})\ \mathsf{tgt}_l)) & \text{if} \quad \overline{v}_i \cap FV(t_i) \neq \emptyset \\ \mathtt{mts} & \text{otherwise} \end{cases} \\ s_0 = \begin{cases} (\mathtt{if}\ (\mathsf{tgt}_l.\mathsf{tag} > ty.\mathsf{tagc}_{c_n})\ \mathtt{throwmalf\ else\ throwunexp}) \\ \qquad \text{if} \quad \{i_1, \ldots, i_p\} \neq \{1, \ldots, n\} \\ \mathtt{throwmalf} \qquad \text{otherwise} \end{cases} \\ s' = (tt(ty_0)\ \mathsf{cres}_l); s; (\mathtt{switch}(\mathsf{tgt}_l.\mathsf{tag})\ \{ty.\mathsf{tagc}_{c_i} \to (s'_i; s_i)\}_i\ s_0) \end{array}}{(\mathbf{case}\ t\ \{c_i(\overline{v}_i) \to t_i\}_i)\ \ k\ \ l\ \rightsquigarrow_{\mathrm{AsgF}(c,f)}\ s'\ \ k_{i_p}\ \ l_{i_p}}$$

$$\frac{\begin{array}{c} t \neq (\mathbf{if}\ \ldots)\ \wedge\ t \neq (\mathbf{case}\ \ldots) \\ t\ \ k\ \ l\ \rightsquigarrow\ s\ \ e\ \ k'\ \ l' \end{array}}{t\ \ k\ \ l\ \rightsquigarrow_{\mathrm{AsgF}(c,f)}\ s; (c.f = e)\ \ k'\ \ l'}$$

The rules for $\rightsquigarrow$ over variables, ops, applications, tuples, equalities, and inequalities are straightforward.

An abstraction translates to an anonymous class instance creation expression. The body, types, and parameters of the apply method are derived from the abstraction. The expression is preceded by statements that copy the contents of the free variables used by the abstraction into final variables, because anonymous classes are prohibited from referencing non-final variables from the outer scope.

For $\rightsquigarrow$ over **if**, there are two rules. If neither branch requires a preceding statement, a conditional expression is produced. Otherwise, an ires variable is declared and assigned values inside the branches of the `if`, using $\rightsquigarrow_{\mathrm{AsgV}}$. The last three premises of the rule just say that at least one branch requires a preceding statement; their presence is necessary because otherwise the rule and the previous one would be both applicable; note that s_1', s_2', etc. are not used in the actual translation.

The rule for $\rightsquigarrow$ over **let** uses $\rightsquigarrow_{\mathrm{AsgNV}}$ to assign the appropriate value to the bound variable, then the regular $\rightsquigarrow$ for the other subterm t.

There are two rules for $\rightsquigarrow$ over **case**. The first rule applies to the case that no branch of the **case** uses the variables bound therein (which is relatively infrequent) and/or the target t of the **case** is a variable: in this case, we do not introduce a tgt variable for the target of the `switch` because the variable would never be used and/or the expression is inexpensive to compute, being a variable or field access. The second rule applies to the complementary case, i.e. at least a branch uses the variables bound therein and the target t of the **case** is not a variable: in this case, we introduce a tgt variable which is used at least in some branches, preventing a potentially expensive expression from being computed twice; the tgt variables is declared and assigned using $\rightsquigarrow_{\mathrm{AsgNV}}$. In both rules, a cres variable is declared first. Each branch of the **case** is translated as follows. The variables bound in the branch translate to field accesses to a newly declared sub variable, initialized by casting the target of the `switch` to the appropriate subclass. If a particular branch does not use the variables bound therein, then no sub variable is introduced. To assign a value to the cres variable, $\rightsquigarrow_{\mathrm{AsgV}}$ is used. For the default case of the `switch`, there are two possibilities. If the `switch` has a case for each summand, an error is thrown signaling a malformed sum value (i.e. whose tag corresponds to no summand). Otherwise, the tag is compared with the highest-numbered tag: if greater, an error is thrown signaling a malformed sum value; otherwise, an error is thrown signaling an unexpected sum value.

The rules for $\rightsquigarrow_{\mathrm{Ret}}$ translate **if** and **case** by recursively using $\rightsquigarrow_{\mathrm{Ret}}$ over the branches; for the other kinds of terms, $\rightsquigarrow$ is used and the resulting expression is returned via `return`. It would be possible to return a conditional expression when neither branch of an `if` requires a preceding statement; however, the use of an explicit `if` seems more natural and idiomatic even for this case.

The rules for $\rightsquigarrow_{\mathrm{AsgNV}}$, $\rightsquigarrow_{\mathrm{AsgV}}$, and $\rightsquigarrow_{\mathrm{AsgF}}$ are analogous to those for $\rightsquigarrow_{\mathrm{Ret}}$.

4.2.8 Bodies

The body of a product equality method returns the conjunction of the equalities of the instance fields (i.e. product components)

$$\frac{\begin{array}{c}\Delta(ty) = \prod_i p_i \;\; ty_i \\ mth = ty.\mathsf{eq} : ty \to \texttt{boolean} \in Mth_{\mathrm{EP}}\end{array}}{body(mth) = (\texttt{return}\ (\ldots \ \&\& \ eq_{ty_i}(\texttt{this}.p_i, \mathsf{eqarg}.p_i) \ \&\& \ \ldots))}$$

The body of a summand equality method associated to a non-constant constructor first checks if the argument's tag coincides with the summand tag and, if that is the case, returns the conjunction of the equalities of the instance fields

$$\frac{\begin{array}{c}\Delta(ty) = \sum_i c_i\ \overline{ty}_i \\ \overline{ty}_i \neq \epsilon \\ mth = \mathsf{sumd}^{ty}_{c_i}.\mathsf{eq} : ty \rightarrow \mathtt{boolean} \in Mth_{\mathrm{ESS}} \\ e = (\ldots\ \&\&\ eq_{ty_{j,i}}(\mathtt{this}.\mathsf{arg}_j, \mathsf{eqargsub}.\mathsf{arg}_j)\ \&\&\ \ldots) \\ s = (\mathsf{sumd}^{ty}_{c_i}\ \mathsf{eqargsub} = ((\mathsf{sumd}^{ty}_{c_i})\ \mathsf{eqarg})); (\mathtt{return}\ e)\end{array}}{body(mth) = (\mathtt{if}\ (\mathsf{eqarg.tag} == ty.\mathsf{tagc}_{c_i})\ s\ \mathtt{else}\ (\mathtt{return\ false}))}$$

The body of a summand equality method associated to a constant constructor just checks if the argument's tag coincides with the summand tag

$$\frac{\begin{array}{c}\Delta(ty) = \sum_i c_i\ \overline{ty}_i \\ \overline{ty}_i = \epsilon \\ mth = \mathsf{sumd}^{ty}_{c_i}.\mathsf{eq} : ty \rightarrow \mathtt{boolean} \in Mth_{\mathrm{ESS}}\end{array}}{body(mth) = (\mathtt{return}\ (\mathsf{eqarg.tag} == ty.\mathsf{tagc}_{c_i}))}$$

The body of a restriction equality method returns the result of comparing for equality the encapsulated values of the restricted type

$$\frac{\begin{array}{c}\Delta(ty) = ty_0|r \\ mth = ty.\mathsf{eq} : ty \rightarrow \mathtt{boolean} \in Mth_{\mathrm{ER}}\end{array}}{body(mth) = (\mathtt{return}\ eq_{ty_0}(\mathtt{this}.\mathsf{relax}, \mathsf{eqarg.relax}))}$$

The body of a quotient equality method invokes the equivalence relation over the encapsulated values of the quotiented type

$$\frac{\begin{array}{c}\Delta(ty) = ty_0/q \\ mth = ty.\mathsf{eq} : ty \rightarrow \mathtt{boolean} \in Mth_{\mathrm{EQ}} \\ e = \begin{cases}\mathtt{this}.\mathsf{choose}.q(\mathsf{eqarg.choose}) & \text{if} \quad ty_0 \in Ty_{\mathrm{U}} \\ ty'.q(\mathtt{this}.\mathsf{choose}, \mathsf{eqarg.choose}) & \text{if} \quad ty_0 \in Ty_{\mathrm{A}} \wedge \\ & \qquad ut(ty_0) = (ty', \ldots) \\ \mathsf{prim}.q(\mathtt{this}.\mathsf{choose}, \mathsf{eqarg.choose}) & \text{otherwise}\end{cases}\end{array}}{body(mth) = (\mathtt{return}\ e)}$$

If the quotiented type is user-defined, the method derived from the equivalence relation is not static and is declared in the class for that user-defined type; otherwise, if the quotiented type is an arrow type with at least a user-defined type, the method derived from the equivalence relation is static and declared in the class for the leftmost user-defined type occurring in ty_0; otherwise, i.e. if no user-defined type occurs in ty_0, the method derived from the equivalence relation is static and declared in prim.

The body of an arrow equality method throws an error signaling an attempt to compare two functions for equality

$$\frac{mth = c.\mathsf{eq} : c \rightarrow \mathtt{boolean} \in Mth_{\mathrm{EA}}}{body(mth) = \mathtt{throwfuneq}}$$

The body of a method derived from a non-constant constructor returns a newly created object of the corresponding subclass

$$\frac{mth = ty.c_i : \overline{ty} \to ty \in Mth_{\mathrm{C}}}{body(mth) = (\texttt{return}\ (\texttt{new}\ \mathsf{sumd}^{ty}_{c_i}(\overline{\mathsf{arg}})))}$$

The body of a method derived from an op with at least a user-defined argument type and whose defining term is a **case** operating on the leftmost parameter with user-defined type, is derived from the translation of the corresponding branch subterm if the **case** has a branch for the summand

$$\frac{\begin{array}{c}\Delta(ty) = \sum_i c_i\ \overline{ty}_i \\ mth = \mathsf{sumd}^{ty}_{c_i}.op : tt(del(\overline{ty}, h)) \to tt(ty) \in Mth_{\mathrm{PMS}} \\ \tau(op) = \overline{ty} \to ty \\ h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\} \\ \pi(op) = \overline{v} \\ \delta(op) = \textbf{case}\ v_h\ \{c_i(\overline{v}_i) \to t_i\}_i \\ i \in \{i_1, \ldots, i_p\} \\ tc = \{v_h \mapsto \texttt{this}\}[\overline{v}_i \mapsto \texttt{this}.\overline{\mathsf{arg}}] \\ cx = \{\overline{v} \mapsto \overline{ty}\}[\overline{v}_i \mapsto \overline{ty}_i] \\ t_i\ \ 1\ \ 1\ \rightsquigarrow^{tc,cx}_{\mathrm{Ret}}\ \ s\ \ k\ \ l\end{array}}{body(mth) = (\texttt{assert this}.\mathsf{asrt}_{op}(del(\overline{v}, h))); s}$$

The translation context maps the leftmost parameter with user-defined type to `this` (because these are instance methods) and the variables bound in the i-th branch to field accesses of the i-th summand class; note that the mapping of a variable bound in the i-th branch may shadow the mapping of the parameter v_h. The method starts by asserting the result of the associated assertion method. If the **case** has no branch for the summand, the body throws an error signaling an unexpected tag in a sum value, after asserting the result of the associated assertion method

$$\frac{\begin{array}{c}\Delta(ty) = \sum_i c_i\ \overline{ty}_i \\ mth = \mathsf{sumd}^{ty}_{c_i}.op : tt(del(\overline{ty}, h)) \to tt(ty) \in Mth_{\mathrm{PMS}} \\ \tau(op) = \overline{ty} \to ty \\ h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\} \\ \pi(op) = \overline{v} \\ i \notin \{i_1, \ldots, i_p\}\end{array}}{body(mth) = (\texttt{assert this}.\mathsf{asrt}_{op}(del(\overline{v}, h))); \texttt{throwunexp}}$$

The body of a method derived from an op with at least a user-defined argument type and whose defining term is not a **case** operating on the leftmost parameter with user-defined type, is derived from the translation of the op's defining term

$$\frac{\begin{array}{c} mth = ty_h.op : tt(del(\overline{ty}, h)) \rightarrow tt(ty) \in Mth_{\mathrm{NPM}} \\ \tau(op) = \overline{ty} \rightarrow ty \\ h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\} \\ \pi(op) = \overline{v} \\ \delta(op) \;\; 1 \;\; 1 \;\; \leadsto^{\{v_h \mapsto \mathtt{this}\}, \{\overline{v} \mapsto \overline{ty}\}}_{\mathrm{Ret}} \;\; s \;\; k \;\; l \end{array}}{body(mth) = (\mathtt{assert\ this}.\mathsf{asrt}_{op}(del(\overline{v}, h))); s}$$

The body of a method without user-defined argument types is derived from the translation of the op's defining term

$$\frac{\begin{array}{c} mth = c.op : tt(\overline{ty}) \rightarrow tt(ty) \in Mth_{\mathrm{UR}} \uplus Mth_{\mathrm{UA}} \uplus Mth_{\mathrm{B}} \\ \tau(op) = \overline{ty} \rightarrow ty \\ \delta(op) \;\; 1 \;\; 1 \;\; \leadsto^{\vec{\emptyset}, \{\pi(op) \mapsto \overline{ty}\}}_{\mathrm{Ret}} \;\; s \;\; k \;\; l \end{array}}{body(mth) = (\mathtt{assert}\ c.\mathsf{asrt}_{op}(\pi(op))); s}$$

Since these methods are static, we use the empty translation context. The associated assertion methods are also static.

The body of the assertion method for an op with at least a user-defined argument type is derived from the translation of the op's restriction term

$$\frac{\begin{array}{c} mth = ty_h.\mathsf{asrt}_{op} : tt(del(\overline{ty}, h)) \rightarrow \mathtt{boolean} \in Mth_{\mathrm{AsU}} \\ \tau(op) = \overline{ty} \rightarrow ty \\ h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\} \\ \pi(op) = \overline{v} \\ \rho(op) \;\; 1 \;\; 1 \;\; \leadsto^{\{v_h \mapsto \mathtt{this}\}, \{\overline{v} \mapsto \overline{ty}\}}_{\mathrm{Ret}} \;\; s \;\; k \;\; l \end{array}}{body(mth) = s}$$

The body of the assertion method for an op without user-defined argument types is derived from the translation of the op's restriction term

$$\frac{\begin{array}{c} mth = c.\mathsf{asrt}_{op} : tt(\overline{ty}) \rightarrow \mathtt{boolean} \in Mth_{\mathrm{AsUR}} \uplus Mth_{\mathrm{AsUA}} \uplus Mth_{\mathrm{AsB}} \\ \tau(op) = \overline{ty} \rightarrow ty \\ \rho(op) \;\; 1 \;\; 1 \;\; \leadsto^{\vec{\emptyset}, \{\pi(op) \mapsto \overline{ty}\}}_{\mathrm{Ret}} \;\; s \;\; k \;\; l \end{array}}{body(mth) = s}$$

The body of a product constructor assigns its parameters to the instance fields

$$\frac{\begin{array}{c} \Delta(ty) = \prod_i p_i \;\; ty_i \\ con = ty : \overline{ty} \in Con_{\mathrm{P}} \end{array}}{body(con) = (\ldots; (\mathtt{this}.p_i = p_i); \ldots)}$$

The body of a summand constructor assigns its parameters to the instance fields and initializes the tag

$$\frac{\begin{array}{c} \Delta(ty) = \sum_i c_i \;\; \overline{ty}_i \\ con = \mathsf{sumd}^{ty}_{c_i} : \overline{ty}_i \in Con_{\mathrm{S}} \end{array}}{body(con) = (\mathtt{this}.\mathsf{tag} = ty.\mathsf{tagc}_{c_i}); (\ldots; (\mathtt{this}.\mathsf{arg}_j = \mathsf{arg}_j); \ldots)}$$

The body of a restriction constructor assigns the argument to the field

$$\frac{\begin{array}{c}\Delta(ty) = ty_0|r \\ con = ty\,{:}\,tt(ty_0) \in Con_{\mathrm{R}} \\ e = \begin{cases} \mathsf{relax}.r() & \text{if} \quad ty_0 \in Ty_{\mathrm{U}} \\ ty'.r(\mathsf{relax}) & \text{if} \quad ty \in Ty_{\mathrm{A}} \ \wedge \ ut(ty_0) = (ty', \ldots) \\ \mathsf{prim}.r(\mathsf{relax}) & \text{otherwise} \end{cases}\end{array}}{body(con) = (\mathtt{assert}\ e); (\mathtt{this}.\mathsf{relax} = \mathsf{relax})}$$

The body starts with an assertion that the supplied argument satisfies the restriction predicate. If the restricted type is user-defined, the method derived from the restriction relation is not static and is declared in the class for that user-defined type; otherwise, if the restricted type is an arrow type with at least a user-defined type, the method derived from the restriction predicate is static and declared in the class for the leftmost user-defined type occurring in ty_0; otherwise, i.e. if no user-defined type occurs in ty_0, the method derived from the restriction predicate is static and declared in prim.

The body of a quotient constructor assigns the argument to the field

$$\frac{\begin{array}{c}\Delta(ty) = ty_0/q \\ con = ty\,{:}\,tt(ty_0) \in Con_{\mathrm{Q}}\end{array}}{body(con) = (\mathtt{this}.\mathsf{choose} = \mathsf{choose})}$$

4.2.9 Static (field) initializers

The static field that holds the numeric tag for a summand is initialized with the appropriate numeric tag (for this to work it is required that no sum type has more than $(2^{31} - 1)$ summands, which is a realistic assumption)

$$\frac{fld = ty.\mathsf{tagc}_{c_i}\,{:}\,\mathtt{int} \in Fld_{\mathrm{TC}}}{sfinit(fld) = i}$$

The static field derived from a constant constructor is initialized with a new object for the summand

$$\frac{fld = ty.c_i\,{:}\,ty \in Fld_{\mathrm{CC}}}{sfinit(fld) = (\mathtt{new}\ \mathsf{sumd}^{ty}_{c_i}(\,))}$$

The field derived from a user-defined constant whose defining term translates to an expression without a preceding statement, is initialized with that expression

$$\frac{\begin{array}{c}fld = c.op\,{:}\,ty \in Fld_{\mathrm{CB}} \uplus Fld_{\mathrm{CU}} \\ \delta(op) \ \ 1 \ \ 1 \ \rightsquigarrow^{\vec{\emptyset},\vec{\emptyset}} \ \mathtt{mts} \ \ e \ \ k \ \ l\end{array}}{sfinit(fld) = e}$$

The field derived from a user-defined constant whose defining term translates to an expression preceded by a non-empty statement, is initialized in a static initializer

$$\frac{\begin{array}{c} \mathit{fld} = c.op : ty \in \mathit{Fld}_{\mathrm{CB}} \uplus \mathit{Fld}_{\mathrm{CU}} \\ \delta(op)\ \ 1\ \ 1 \leadsto^{\vec{\emptyset},\vec{\emptyset}}_{\mathrm{AsgF}(c,op)}\ \ s\ \ k\ \ l \\ \delta(op)\ \ 1\ \ 1 \leadsto^{\vec{\emptyset},\vec{\emptyset}}\ \ s'\ \ e'\ \ k'\ \ l' \\ s' \neq \mathtt{mts} \end{array}}{s \in \mathit{sinit}(c)}$$

The third and fourth conditions of this rule say that the expression derived from $\delta(op)$ requires a non-empty preceding statement; note that s' and e' are not used in the actual translation.

4.3 Variable restoration

Consider a Java program

$$\mathcal{P} = \langle C, \mathit{ext}, \mathit{abs}_{\mathrm{C}}, \mathit{Fld}, \mathit{stc}_{\mathrm{F}}, \mathit{Mth}, \mathit{stc}_{\mathrm{M}}, \mathit{abs}_{\mathrm{M}}, \mathit{Con}, \mathit{param}, \mathit{body}, \mathit{sfinit}, \mathit{sinit}\rangle$$

that uses names in

$$\mathcal{N} = \mathcal{N}_0 \uplus \{v_k \mid v \in \mathcal{N}_0 \ \wedge\ k \in \mathbf{N}_+\}$$

as resulting from the first translation phase; the names introduced in the second translation phase, e.g. ires_k, are considered to belong to $\mathcal{N}_0$.

The result of variable restoration is the Java program

$$\mathcal{P}' = \langle C, \mathit{ext}, \mathit{abs}_{\mathrm{C}}, \mathit{Fld}, \mathit{stc}_{\mathrm{F}}, \mathit{Mth}, \mathit{stc}_{\mathrm{M}}, \mathit{abs}_{\mathrm{M}}, \mathit{Con}, \mathit{param}, \mathit{body}', \mathit{sfinit}, \mathit{sinit}'\rangle$$

defined as follows.

4.3.1 Names

$\mathcal{P}'$ uses the same names $\mathcal{N}$ used by $\mathcal{P}$.

4.3.2 Statement transformation

The idea is that we traverse each statement and whenever we find a declaration of a variable of the form v_k, we check whether the variable v has the same type as v_k and v is never used after the declaration. If that is not the case, we leave the variable unchanged. If that is the case, we safely rename v_k to v throughout the rest of the statement. If the declaration has an initializer, the declaration is turned into an assignment; otherwise, the declaration is removed altogether.

To keep track of the types of the variables encountered while traversing the statement, we use a context that is defined in complete analogy with *Fun* and that is threaded through

$$Cx = V \xrightarrow{\mathrm{f}} Ty$$

Since statements can be nested, in order to check whether a variable is never used after a declaration, it is not enough to look at the free variables in the substatements where the declaration occurs. For instance, if we find a declaration of v_k in a statement s_1 that is nested in $((\texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2); s)$, it is safe to rename v_k to v if v is used neither in the rest of s_1 nor in s. The set of variables used in outer statements is passed as input when transforming an inner statement.

The transformation is captured by a functional 5-ary relation

$$\leadsto \subseteq Cx \times \mathcal{P}_\omega(V) \times S \times Cx \times S$$

The meaning of $(cx \;\; \widetilde{v} \;\; s \;\leadsto\; cx' \;\; s')$ is that the result of transforming the statement s when the context of the variables declared so far is cx and when the variables used in outer statements are $\widetilde{v}$, is s' and that the updated context is cx'. The relation is defined as

$$\frac{}{cx \;\; \widetilde{v} \;\; \texttt{mts} \;\leadsto\; cx \;\; \texttt{mts}}$$

$$\frac{cx \;\; \widetilde{v} \;\; s \;\leadsto\; cx' \;\; s'}{cx \;\; \widetilde{v} \;\; (\texttt{return}\ e); s \;\leadsto\; cx' \;\; (\texttt{return}\ e); s'}$$

$$\frac{\begin{array}{c} v \in \mathcal{N}_0 \\ cx \;\; \widetilde{v} \;\; s \;\leadsto\; cx' \;\; s' \end{array}}{cx \;\; \widetilde{v} \;\; (ty\ v); s \;\leadsto\; cx' \;\; (ty\ v); s'}$$

$$\frac{\begin{array}{c} cx(v) = ty \\ v \notin \widetilde{v} \cup FV(s) \\ cx \;\; \widetilde{v} \;\; s[v/v_k] \;\leadsto\; cx' \;\; s' \end{array}}{cx \;\; \widetilde{v} \;\; (ty\ v_k); s \;\leadsto\; cx' \;\; s'}$$

$$\frac{\begin{array}{c} cx(v) \neq ty \;\vee\; v \in \widetilde{v} \cup FV(s) \\ cx[v_k \mapsto ty] \;\; \widetilde{v} \;\; s \;\leadsto\; cx' \;\; s' \end{array}}{cx \;\; \widetilde{v} \;\; (ty\ v_k); s \;\leadsto\; cx' \;\; (ty\ v_k); s'}$$

$$\frac{\begin{array}{c} v \in \mathcal{N}_0 \\ cx \;\; \widetilde{v} \;\; s \;\leadsto\; cx' \;\; s' \end{array}}{cx \;\; \widetilde{v} \;\; (ty\ v = e); s \;\leadsto\; cx' \;\; (ty\ v = e); s'}$$

$$\frac{\begin{array}{c} cx(v) = ty \\ v \notin \widetilde{v} \cup FV(s) \\ cx \;\; \widetilde{v} \;\; s[v/v_k] \;\leadsto\; cx' \;\; s' \end{array}}{cx \;\; \widetilde{v} \;\; (ty\ v_k = e); s \;\leadsto\; cx' \;\; (v = e); s'}$$

$$\frac{\begin{array}{c} cx(v) \neq ty \ \vee \ v \in \widetilde{v} \cup FV(s) \\ cx[v_k \mapsto ty] \ \ \widetilde{v} \ \ s \ \rightsquigarrow \ cx' \ \ s' \end{array}}{cx \ \ \widetilde{v} \ \ (ty \ v_k = e); s \ \rightsquigarrow \ cx' \ \ (ty \ v_k = e); s'}$$

$$\frac{cx \ \ \widetilde{v} \ \ s \ \rightsquigarrow \ s' \ \ cx'}{cx \ \ \widetilde{v} \ \ (\texttt{fin} \ ty \ v = e); s \ \rightsquigarrow \ cx' \ \ (\texttt{fin} \ ty \ v = e); s'}$$

$$\frac{cx \ \ \widetilde{v} \ \ s \ \rightsquigarrow \ cx' \ \ s'}{cx \ \ \widetilde{v} \ \ (v = e); s \ \rightsquigarrow \ cx' \ \ (v = e); s'}$$

$$\frac{cx \ \ \widetilde{v} \ \ s \ \rightsquigarrow \ cx' \ \ s'}{cx \ \ \widetilde{v} \ \ (e_0.f = e); s \ \rightsquigarrow \ cx' \ \ (e_0.f = e); s'}$$

$$\frac{cx \ \ \widetilde{v} \ \ s \ \rightsquigarrow \ cx' \ \ s'}{cx \ \ \widetilde{v} \ \ (c.f = e); s \ \rightsquigarrow \ cx' \ \ (c.f = e); s'}$$

$$\frac{\begin{array}{c} cx \ \ (\widetilde{v} \cup FV(s)) \ \ s_1 \ \rightsquigarrow \ cx_1 \ \ s_1' \\ cx \ \ (\widetilde{v} \cup FV(s)) \ \ s_2 \ \rightsquigarrow \ cx_2 \ \ s_2' \\ cx \ \ \widetilde{v} \ \ s \ \rightsquigarrow \ cx' \ \ s' \end{array}}{cx \ \ \widetilde{v} \ \ (\texttt{if} \ (e) \ s_1 \ \texttt{else} \ s_2); s \ \rightsquigarrow \ cx' \ \ (\texttt{if} \ (e) \ s_1' \ \texttt{else} \ s_2'); s'}$$

$$\frac{\begin{array}{c} \forall i. \ \ cx_{i-1} \ \ (\widetilde{v} \cup FV(s) \cup \bigcup_{i' > i} FV(s_{i'}) \cup FV(s_0)) \ \ s_i \ \rightsquigarrow \ cx_i \ \ s_i' \\ cx_n \ \ (\widetilde{v} \cup FV(s)) \ \ s_0 \ \rightsquigarrow \ cx \ \ s_0' \\ cx_0 \ \ \widetilde{v} \ \ s \ \rightsquigarrow \ cx' \ \ s' \end{array}}{cx_0 \ \ \widetilde{v} \ \ (\texttt{switch}(e) \ \{e_i \to s_i\}_i \ s_0); s \ \rightsquigarrow \ cx' \ \ (\texttt{switch}(e) \ \{e_i \to s_i'\}_i \ s_0'); s'}$$

$$\frac{cx \ \ \widetilde{v} \ \ s \ \rightsquigarrow \ cx' \ \ s'}{cx \ \ \widetilde{v} \ \ (\texttt{assert} \ e); s \ \rightsquigarrow \ cx' \ \ (\texttt{assert} \ e); s'}$$

$$\frac{cx \ \ \widetilde{v} \ \ s \ \rightsquigarrow \ cx' \ \ s'}{cx \ \ \widetilde{v} \ \ \texttt{throwfuneq}; s \ \rightsquigarrow \ cx' \ \ \texttt{throwfuneq}; s'}$$

$$\frac{cx \ \ \widetilde{v} \ \ s \ \rightsquigarrow \ cx' \ \ s'}{cx \ \ \widetilde{v} \ \ \texttt{throwmalf}; s \ \rightsquigarrow \ cx' \ \ \texttt{throwmalf}; s'}$$

$$\frac{cx \ \ \widetilde{v} \ \ s \ \rightsquigarrow \ cx' \ \ s'}{cx \ \ \widetilde{v} \ \ \texttt{throwunexp}; s \ \rightsquigarrow \ cx' \ \ \texttt{throwunexp}; s'}$$

Final variables are left unchanged because otherwise they would not be final any more. Final variables are used for free variables in *Fun* abstractions; they must be final to be used inside anonymous classes.

The contexts cx_1 and cx_2 obtained by transforming the branches s_1 and s_2 of an `if` are discarded because variables declared inside the branches are not visible outside the `if`. The context cx is passed to both branches because the branches are parallel, i.e. variables declared in a branch are not visible in the other branch. The set of used variables is augmented with the free variables of s prior to visiting the branches.

The context cx obtained by transforming a `switch` is discarded because variables declared inside the `switch` block are not visible outside. Differently from `if`, the context is threaded through the branches (i.e. cases) of the `switch` because the branches all live in the same block. The set of used variables is augmented with not only the free variables of s but also the free variables of the branches after i prior to visiting s_i.

4.3.3 Transformed program

The only program components that change are method bodies and static initializers

$$\frac{\begin{array}{c} mth = c.m\!:\!\overline{ty} \rightarrow ty \in \mathcal{D}(body) \\ param(mth) = \overline{v} \\ \{\overline{v} \mapsto \overline{ty}\} \;\; \emptyset \;\; body(mth) \;\rightsquigarrow\; cx \;\; s \end{array}}{body'(mth) = s}$$

$$\frac{\begin{array}{c} s \in sinit(c) \\ \vec{\emptyset} \;\; \emptyset \;\; s \;\rightsquigarrow\; cx \;\; s' \end{array}}{s' \in sinit'(c)}$$

In both cases, the set of used variables is initially empty because the body or the initializer is at the top level.

4.4 Assertion inlining

Consider a Java program

$$\mathcal{P} = \langle C, ext, abs_{\mathrm{C}}, Fld, stc_{\mathrm{F}}, Mth, stc_{\mathrm{M}}, abs_{\mathrm{M}}, Con, param, body, sfinit, sinit\rangle$$

that uses names in

$$\mathcal{N} = \mathcal{N}_0 \uplus \{\mathsf{asrt}_{op} \mid op \in \mathcal{N}_0\}$$

as resulting from the second translation phase; the other names introduced in the second translation phase, e.g. ires_k, are considered to belong to $\mathcal{N}_0$.

The result of assertion inlining is the Java program

$$\mathcal{P}' = \langle C, ext, abs_{\mathrm{C}}, Fld, stc_{\mathrm{F}}, Mth', stc'_{\mathrm{M}}, abs_{\mathrm{M}}, Con, param', body', sfinit, sinit\rangle$$

defined as follows.

4.4.1 Names

$\mathcal{P}'$ uses the same names $\mathcal{N}$ used by $\mathcal{P}$.

4.4.2 Transformed program

The only program components that change are the methods and some associated predicates and functions.

Assertion methods consisting of single expressions are removed

$$\begin{aligned} \mathit{Mth}' = \mathit{Mth} - \{ \mathit{mth} = c.\mathsf{asrt}_{op} : \overline{ty} \rightarrow \texttt{boolean} \in \mathit{Mth} \mid \\ \exists e \in E.\ \ \mathit{body}(\mathit{mth}) = (\texttt{return}\ e) \} \end{aligned}$$

The predicate for static methods is accordingly shrinked

$$\mathit{stc}'_{\mathrm{M}} = \mathit{stc}_{\mathrm{M}} \cap \mathit{Mth}'$$

The parameters of the surviving methods are unchanged

$$\mathit{mth} \in \mathit{Mth}' \ \Rightarrow\ \mathit{param}'(\mathit{mth}) = \mathit{param}(\mathit{mth})$$

If a method starts by asserting the result of one of the removed assertion methods, the expression returned by the removed method replaces the method call, provided it is not `true`

$$\frac{\begin{array}{c} \mathit{mth} \in \mathit{Mth}' \\ \mathit{body}(\mathit{mth}) = (\texttt{assert}\ x.\mathsf{asrt}_{op}(\overline{e})); s \\ \mathit{mth}' = c.\mathsf{asrt}_{op} : \overline{ty} \rightarrow ty \in \mathit{Mth} - \mathit{Mth}' \\ \mathit{body}(\mathit{mth}') = (\texttt{return}\ e) \\ e \neq \texttt{true} \end{array}}{\mathit{body}'(\mathit{mth}) = (\texttt{assert}\ e); s}$$

If the expression is `true`, the assertion is removed altogether

$$\frac{\begin{array}{c} \mathit{mth} \in \mathit{Mth}' \\ \mathit{body}(\mathit{mth}) = (\texttt{assert}\ x.\mathsf{asrt}_{op}(\overline{e})); s \\ \mathit{mth}' = c.\mathsf{asrt}_{op} : \overline{ty} \rightarrow ty \in \mathit{Mth} - \mathit{Mth}' \\ \mathit{body}(\mathit{mth}') = (\texttt{return}\ \texttt{true}) \end{array}}{\mathit{body}'(\mathit{mth}) = s}$$

For all other methods, the body is unchanged

$$\frac{\begin{array}{c} \mathit{mth} \in \mathit{Mth}' \\ \nexists\, x, op, \overline{e}, s.\ \ \mathit{body}(\mathit{mth}) \neq (\texttt{assert}\ x.\mathsf{asrt}_{op}(\overline{e})); s \end{array}}{\mathit{body}'(\mathit{mth}) = \mathit{body}(\mathit{mth})}$$

$$\frac{\begin{array}{c} \mathit{mth} \in \mathit{Mth}' \\ \mathit{body}(\mathit{mth}) = (\texttt{assert}\ e_0.\mathsf{asrt}_{op}(\overline{e})); s \\ c.\mathsf{asrt}_{op} : \overline{ty} \rightarrow ty \in \mathit{Mth}' \end{array}}{\mathit{body}'(\mathit{mth}) = \mathit{body}(\mathit{mth})}$$

4.5 Concrete name translation

The overall translation from a *Fun* program $\mathcal{P}$ to a Java program $\mathcal{P}'$, consisting of the four phases specified above, is parameterized over the names $\mathcal{N}$ used by $\mathcal{P}$. $\mathcal{P}'$ uses names

$$
\begin{aligned}
\mathcal{N}' = \mathcal{N} & \\
& \uplus \mathcal{N}_\mathrm{A} \\
& \uplus \{v_k \mid v \in V \ \wedge \ k \in \mathbf{N}_+\} \\
& \uplus \{\mathsf{sumd}^{ty}_{c_i} \mid \Delta(ty) = \textstyle\sum_i c_i \ \overline{ty}_i\} \\
& \uplus \{\mathsf{arg}_j \mid j \in \mathbf{N}_+\} \\
& \uplus \{\mathsf{ires}_k \mid k \in \mathbf{N}_+\} \\
& \uplus \{\mathsf{cres}_l \mid l \in \mathbf{N}_+\} \\
& \uplus \{\mathsf{sub}^{c_i}_l \mid \exists ty. \ \ \Delta(ty) = \textstyle\sum_i c_i \ \overline{ty}_i \ \wedge \ l \in \mathbf{N}_+\} \\
& \uplus \{\mathsf{tgt}_l \mid l \in \mathbf{N}_+\} \\
& \uplus \{\mathsf{tagc}_{c_i} \mid \exists ty. \ \ \Delta(ty) = \textstyle\sum_i c_i \ \overline{ty}_i\} \\
& \uplus \{\mathsf{fin}_v \mid v \in V\} \\
& \uplus \{\mathsf{fin}_{v_k} \mid v \in V \ \wedge \ k \in \mathbf{N}_+\} \\
& \uplus \{\mathsf{asrt}_{op} \mid op \in Op_\mathrm{U}\} \\
& \uplus \{\mathsf{prim}, \mathsf{eq}, \mathsf{apply}, \mathsf{eqarg}, \mathsf{eqargsub}, \mathsf{tag}, \mathsf{relax}, \mathsf{choose}\}
\end{aligned}
$$

where

$$\mathcal{N}_\mathrm{A} = \{\mathsf{arrow}^{\overline{ty}}_{ty} \mid \overline{ty} \in Ty^+ \ \wedge \ ty \in Ty\}$$

In order to produce valid Java code, the names in $\mathcal{N}'$ used by $\mathcal{P}'$ must be translated to Java identifiers that are distinct within the various name spaces (i.e. packages, classes, and method/constructor bodies) of $\mathcal{P}'$.

A Java identifier is a non-empty sequence of Unicode characters that starts with a letter or underscore or dollar, continues with letters, digits, underscores, and dollars, and is not a keyword or literal

$$
\begin{aligned}
\mathcal{J} = \{(ch, \overline{ch}) \mid \ & ch \in \mathcal{C} \ \wedge \ \overline{ch} \in \mathcal{C}^* \ \wedge \\
& (alpha(ch) \ \vee \ ch \in \{\_, \$\}) \ \wedge \\
& (\forall i. \ \ alphanum(ch_i) \ \vee \ ch_i \in \{\_, \$\})\} \ - \ \mathcal{J}_\mathrm{KL}
\end{aligned}
$$

where

$$\mathcal{C}$$

is the set of Unicode characters,

$$\mathcal{J}_\mathrm{KL}$$

is the set of (Unicode character sequences forming) Java keywords and (boolean and null) literals, and the predicates

$$alpha \subseteq \mathcal{C} \qquad\qquad alphanum \subseteq \mathcal{C}$$

capture whether a Unicode character is alphabetic (i.e. letter) and/or alphanumeric (i.e. letter or digit).[8]

[8] In Java, _ and $ are considered letters. In this formalization, we do not consider them letters but our definition of Java identifiers coincides with the official one.

We now define a possible concrete name translation, under mundane assumptions about $\mathcal{N}$ and $\mathcal{P}$. Other translations are possible. The examples in Section 2 do not exactly follow this name translation for simplicity.

4.5.1 Assumptions on source program names

Fun uses identifiers consisting of non-empty sequences of Unicode characters starting with a letter, continuing with letters, digits, underscores, and question marks, and perhaps distinct from certain reserved Unicode character sequences (e.g. keywords in the concrete syntax of *Fun*, which is not specified in this document)

$$\begin{array}{rl}\mathcal{I} = \{(ch, \overline{ch}) \mid & ch \in \mathcal{C} \ \wedge \ \overline{ch} \in \mathcal{C}^* \ \wedge \ alpha(ch) \ \wedge \\ & (\forall i.\ \ alphanum(ch_i) \ \vee \ ch_i \in \{\_, ?\})\} \ - \ \mathcal{I}_{\mathrm{R}}\end{array}$$

where the exact contents of $\mathcal{I}_{\mathrm{R}}$ are unimportant because we only translate the identifiers in $\mathcal{I}$. Since ASCII characters are Unicode characters, this assumption covers the more restrictive possibility that *Fun* identifiers only consist of ASCII characters.

User-defined types are identifiers

$$Ty_{\mathrm{U}} \subseteq \mathcal{I}$$

Projectors and constructors are also identifiers

$$\Delta(ty) = \prod_i p_i \ ty_i \ \Rightarrow \ \overline{p} \subseteq \mathcal{I} \qquad\qquad \Delta(ty) = \sum_i c_i \ \overline{ty}_i \ \Rightarrow \ \overline{c} \subseteq \mathcal{I}$$

A user-defined op consists of an identifier accompanied by the op's type

$$Op_{\mathrm{U}} \subseteq \{oid^{ty} \mid oid \in \mathcal{I} \ \wedge \ ty \in Ty\}$$

$$oid^{ty} \in Op_{\mathrm{U}} \ \Rightarrow \ \tau(oid^{ty}) = ty$$

which implies that ops can be overloaded, i.e. two ops can have the same identifier but different types. In lifting projectors and constructors to ops, we tag them with their types as well

$$\begin{array}{l}Op_{\mathrm{P}} = \{p_i^{ty \to ty_i} \mid \Delta(ty) = \prod_i p_i \ \ ty_i\} \\ Op_{\mathrm{C}} = \{c_i^{\overline{ty}_i \to ty} \mid \Delta(ty) = \sum_i c_i \ \ \overline{ty}_i\}\end{array}$$

Even if two product types have the same projector identifier, the projector ops are distinct because the product types are different; an analogous fact applies to constructors. User-defined ops, as well as projectors and constructors (lifted as ops) are required to be distinct

$$Op_{\mathrm{P}} \cap Op_{\mathrm{C}} = \emptyset \qquad\qquad Op_{\mathrm{P}} \cap Op_{\mathrm{U}} = \emptyset \qquad\qquad Op_{\mathrm{C}} \cap Op_{\mathrm{U}} = \emptyset$$

Variables are also identifiers

$$V \subseteq \mathcal{I}$$

4.5.2 Preliminaries

There is considerable overlap between $\mathcal{I}$ and $\mathcal{J}$. Normally, a *Fun* identifier translates to itself, as a Java identifier. But unfortunately, *Fun* identifiers may include `?`, which is disallowed in Java identifiers. In addition, a *Fun* identifier may happen to be a Java keyword or literal in $\mathcal{J}_{\mathrm{KL}}$.

Identifier translation is captured by the function $it : \mathcal{I} \to \mathcal{J}$ defined as

$$\begin{aligned} id \in \mathcal{J} \;&\Rightarrow\; it(id) = id \\ \texttt{?} \in id \;&\Rightarrow\; it(id) = id[\texttt{\$Q}/\texttt{?}] \\ id \in \mathcal{J}_{\mathrm{KL}} \;&\Rightarrow\; it(id) = (id, \texttt{\$}) \end{aligned}$$

i.e. *Fun* identifiers that are also Java identifiers translate to themselves, `?` is replaced[9] by `$Q` (`Q` for "question mark"), and Java keywords and literals are suffixed by `$`. For example, $it(\texttt{fact}) = \texttt{fact}$, $it(\texttt{empty?}) = \texttt{empty\$Q}$, and $it(\texttt{null}) = \texttt{null\$}$. The function it is injective because `$` is disallowed in *Fun* identifiers and is always followed by `Q` when it replaces `?`: given $it(id)$, we can always determine id.

We will need to translate natural numbers to their decimal ASCII representation via the injective function $nt : \mathbf{N} \to \mathcal{C}^*$ defined as

$$\begin{aligned} nt(0) &= \mathsf{0} \\ &\vdots \\ nt(9) &= \mathsf{9} \\ n \geq 10 \;\Rightarrow\; nt(n) &= (nt(n \;\mathbf{div}\; 10), nt(n \;\mathbf{mod}\; 10)) \end{aligned}$$

where **div** and **mod** are integer division and remainder.

We will also need to generate ASCII representations of *Fun* (sequences of) types via the injective function $trp : Ty^* \to \mathcal{C}^*$ defined as

$$\begin{aligned} trp(\mathsf{Bool}) &= \texttt{\$B} \\ trp(\mathsf{Int}) &= \texttt{\$I} \\ trp(\mathsf{Char}) &= \texttt{\$C} \\ ty \in Ty_{\mathrm{U}} \;\Rightarrow\; trp(ty) &= (\texttt{\$U}, it(ty)) \\ trp(\overline{ty} \to ty) &= (\texttt{\$F}, trp(\overline{ty}), \texttt{\$T}, trp(ty)) \\ trp(\overline{ty}) &= (trp(ty_1), \ldots, trp(ty_n)) \end{aligned}$$

i.e. Bool, Int, and Char are represented as `$B`, `$I`, and `$C` (for "boolean", "integer", and "character"), user-defined types are represented by prepending `$U` to their translation (for "user-defined"), and arrow types are represented by `$F` followed by the argument types' representation followed by `$T` followed by the result type's representation; `$F` and `$T` (for "from" and "to") play the role of parentheses, e.g. $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$ is represented as `$F$F$I$T$I$T$I` while $\mathsf{Int} \to (\mathsf{Int} \to \mathsf{Int})$ is represented as `$F$I$T$F$I$T$I`.

[9] We use the same substitution notation used for terms, expressions, and statements.

4.5.3 Class names

The classes comprising $\mathcal{P}'$ are meant to live in their own package (unnamed or otherwise). Thus, we must ensure that their (simple) names are distinct.

Class name translation is captured by the injective function $ct : C \to \mathcal{J}$ defined as

$$\begin{aligned}
ct(\mathsf{prim}) &= \texttt{Prim} \\
ty \in Ty_{\mathrm{U}} \;\Rightarrow\; ct(ty) &= \begin{cases} it(ty) & \text{if} \quad ty \neq \texttt{Prim} \\ \texttt{Prim\$} & \text{otherwise} \end{cases} \\
ct(\mathsf{sumd}^{ty}_{c_i}) &= (it(ty), \texttt{\$\$}, it(c_i)) \\
ct(\mathsf{arrow}^{\overline{ty}}_{ty}) &= (tyt(\overline{ty}), \texttt{\$To\$}, tyt(ty))
\end{aligned}$$

where the injective function $tyt : Ty^* \to \mathcal{C}^*$ is defined as

$$\begin{aligned}
tyt(\mathsf{Bool}) &= \texttt{boolean} \\
tyt(\mathsf{Int}) &= \texttt{int} \\
tyt(\mathsf{Char}) &= \texttt{char} \\
ty \in Ty_{\mathrm{U}} \;\Rightarrow\; tyt(ty) &= \begin{cases} it(ty) & \text{if} \quad ty \neq \texttt{Prim} \\ \texttt{Prim\$} & \text{otherwise} \end{cases} \\
tyt(\overline{ty} \to ty) &= (\texttt{\$From\$}, tyt(\overline{ty}), \texttt{\$To\$}, tyt(ty)) \\
tyt(\overline{ty}) &= (tyt(ty_1), \texttt{\$\$}, \ldots, \texttt{\$\$}, tyt(ty_n))
\end{aligned}$$

The name prim of the class that collects all fields and methods derived from ops with all built-in types always translates to `Prim`.

The names Ty_{U} of the product and sum classes translate to $it(Ty_{\mathrm{U}})$ if they are distinct from `Prim`, otherwise a `$` is appended. All these names are distinct because of the injectivity of it and because `Prim` is not a Java keyword; they are also obviously distinct from `Prim`.

The names $\mathsf{sumd}^{ty}_{c_i}$ of the summand classes translate to the concatenation of $it(ty)$ and $it(c_i)$ separated by `$$`, e.g. $\mathsf{sumd}^{\texttt{List}}_{\texttt{nil}}$ and $\mathsf{sumd}^{\texttt{List}}_{\texttt{cons}}$ translate to `List$$nil` and `List$$cons`. The sum types are distinct and the constructors of any sum type are distinct. Every occurrence of `$` in $it(Ty_{\mathrm{U}})$ is immediately followed by `Q` or is at the end of the identifier. Thus, the summand class names are distinct from each other and from the other class names.

The names $\mathsf{arrow}^{\overline{ty}}_{ty}$ of the arrow classes recursively translate to the translations (via tyt) of the *Fun* argument types $\overline{ty}$ separated by `$$`, followed by `$To$`, followed by the translation (via tyt) of the *Fun* result type ty. The built-in types translate to the Java keywords that denote the corresponding Java types. The user-defined types translate like the corresponding classes. Nested arrow types translate like arrow classes prepended by `$From$`. For example, $\mathsf{arrow}^{\mathsf{Int},\mathsf{Bool}\to\mathsf{U}}_{\mathsf{Char}}$ translates to `$From$int$$boolean$To$U$To$char` and $\mathsf{arrow}^{\mathsf{Int}}_{\mathsf{Bool},\mathsf{U}\to\mathsf{Char}}$ translates to `int$To$$From$boolean$$U$To$char`. The strings `$From$` and `$To$` play the role of parentheses. The arrow class names are distinct from each other and from the other class names, which have no `$To$` embedded in them.

4.5.4 Field names

The fields declared in a class must have distinct names.

Field name translation is captured by the function $ft : Fld \to \mathcal{J}$ defined as

$$\frac{fld = ty.p_i : ty_i \in Fld_{\mathrm{P}}}{ft(fld) = it(p_i)}$$

$$\frac{fld = \mathsf{sumd}^{ty}_{c_i}.\mathsf{arg}_j : ty_{j,i} \in Fld_{\mathrm{CA}}}{ft(fld) = (\mathtt{arg}, nt(j))}$$

$$\frac{fld = ty.\mathsf{tag} : \mathtt{int} \in Fld_{\mathrm{T}}}{ft(fld) = \mathtt{tag\$}}$$

$$\frac{fld = ty.\mathsf{tagc}_{c_i} : \mathtt{int} \in Fld_{\mathrm{TC}}}{ft(fld) = (\mathtt{TAG\$\$}, it(c_i))}$$

$$\frac{fld = ty.\mathsf{relax} : ty_0 \in Fld_{\mathrm{R}}}{ft(fld) = \mathtt{relax}}$$

$$\frac{fld = ty.\mathsf{choose} : ty_0 \in Fld_{\mathrm{C}}}{ft(fld) = \mathtt{choose}}$$

$$\frac{fld = ty.c_i : ty \in Fld_{\mathrm{CC}}}{ft(fld) = it(c_i)}$$

$$\frac{fld = \mathsf{prim}.oid^{\mathsf{Bool}} : \mathtt{boolean} \in Fld_{\mathrm{CB}}}{ft(fld) = \begin{cases} it(oid) & \text{if} \quad \{oid^{\mathsf{Int}}, oid^{\mathsf{Char}}\} \cap Op_{\mathrm{U}} \neq \emptyset \\ (it(oid), \mathtt{\$B}) & \text{otherwise} \end{cases}}$$

$$\frac{fld = \mathsf{prim}.oid^{\mathsf{Int}} : \mathtt{int} \in Fld_{\mathrm{CB}}}{ft(fld) = \begin{cases} it(oid) & \text{if} \quad \{oid^{\mathsf{Bool}}, oid^{\mathsf{Char}}\} \cap Op_{\mathrm{U}} \neq \emptyset \\ (it(oid), \mathtt{\$I}) & \text{otherwise} \end{cases}}$$

$$\frac{fld = \mathsf{prim}.oid^{\mathsf{Char}} : \mathtt{char} \in Fld_{\mathrm{CB}}}{ft(fld) = \begin{cases} it(oid) & \text{if} \quad \{oid^{\mathsf{Int}}, oid^{\mathsf{Bool}}\} \cap Op_{\mathrm{U}} \neq \emptyset \\ (it(oid), \mathtt{\$C}) & \text{otherwise} \end{cases}}$$

$$\frac{\begin{array}{c} \mathit{fld} = \mathit{ty}.\mathit{oid}^{\mathit{ty}} : \mathit{ty} \in \mathit{Fld}_{\mathrm{CU}} \\ \Delta(\mathit{ty}) \in \mathit{TySum} \end{array}}{\mathit{ft}(\mathit{fld}) = \mathit{it}(\mathit{oid})}$$

$$\frac{\begin{array}{c} \mathit{fld} = \mathit{ty}.\mathit{oid}^{\mathit{ty}} : \mathit{ty} \in \mathit{Fld}_{\mathrm{CU}} \\ \Delta(\mathit{ty}) = \prod_i p_i\ \mathit{ty}_i \end{array}}{\mathit{ft}(\mathit{fld}) = \begin{cases} \mathit{it}(\mathit{oid}) & \text{if} \quad \mathit{oid} \notin \overline{p} \\ (\mathit{it}(\mathit{oid}), \mathtt{\$}) & \text{otherwise} \end{cases}}$$

$$\frac{\begin{array}{c} \mathit{fld} = \mathit{ty}.\mathit{oid}^{\mathit{ty}} : \mathit{ty} \in \mathit{Fld}_{\mathrm{CU}} \\ \Delta(\mathit{ty}) \in \mathit{TyRestr} \end{array}}{\mathit{ft}(\mathit{fld}) = \begin{cases} \mathit{it}(\mathit{oid}) & \text{if} \quad \mathit{oid} \neq \mathtt{relax} \\ \mathtt{relax\$} & \text{otherwise} \end{cases}}$$

$$\frac{\begin{array}{c} \mathit{fld} = \mathit{ty}.\mathit{oid}^{\mathit{ty}} : \mathit{ty} \in \mathit{Fld}_{\mathrm{CU}} \\ \Delta(\mathit{ty}) \in \mathit{TyQuot} \end{array}}{\mathit{ft}(\mathit{fld}) = \begin{cases} \mathit{it}(\mathit{oid}) & \text{if} \quad \mathit{oid} \neq \mathtt{choose} \\ \mathtt{choose\$} & \text{otherwise} \end{cases}}$$

A summand class only declares instance fields $\mathtt{arg}_j$, one for each argument of the corresponding constructor. These fields translate to `arg1`, `arg2`, etc., which are distinct.

A sum class ty declares a static field $\mathtt{tagc}_{c_i}$ for each constructor, an instance field `tag` for the numeric tag, a static field c_i for each constant constructor, and a static field oid^{ty} for each user-defined constant with that sum type. The constructors of a sum type are distinct. Moreover, the assumptions on $\mathcal{P}$ prevent two user-defined constants of type ty from having the same identifier and also prevent any user-defined constant of type ty from having the same identifier as a constant constructor of ty. Thus, we translate c_i and oid^{ty} to $it(c_i)$ and $it(oid)$. We translate `tag` to `tag$`, which is distinct from all the fields derived from constant constructors and from user-defined constants because `tag` is not a Java keyword. We translate $\mathtt{tagc}_{c_i}$ by prepending `TAG$$` to the translation of c_i, which yields identifiers that are distinct from each other (because constructors are distinct) and from the other fields because the other fields do not contain `$$`.

A product class ty declares an instance field p_i for each projector and a static field oid^{ty} for each user-defined constant with that product type. The projectors of a product type are distinct. While the assumptions on $\mathcal{P}$ prevent two user-defined constants of type ty from having the same identifier, nothing prevents one such constant to have the same identifier as a projector. We always translate a projector p_i to $it(p_i)$. A user-defined constant oid^{ty} translates to $it(oid)$ if oid is distinct from every projector of the product type. Otherwise, we append `$` to $it(oid)$. Either way, we end up with distinct field names because no projector translates to a Java keyword.

A restriction class ty declares an instance field relax holding a value of the restricted type and a static field oid^{ty} for each user-defined constant with that restriction type. We always translate relax to `relax`. While the assumptions on $\mathcal{P}$ prevent two user-defined constants of type ty from having the same identifier, nothing prevents one such constant to have the identifier `relax`. A user-defined constant oid^{ty} translates to $it(oid)$ if oid is not `relax`, otherwise we append `$`. Either way, we end up with distinct field names because `relax` is not a Java keyword.

A quotient class ty declares an instance field choose holding a value of the quotiented type and a static field oid^{ty} for each user-defined constant with that quotient type. We always translate choose to `choose`. While the assumptions on $\mathcal{P}$ prevent two user-defined constants of type ty from having the same identifier, nothing prevents one such constant to have the identifier `choose`. A user-defined constant oid^{ty} translates to $it(oid)$ if oid is not `choose`, otherwise we append `$`. Either way, we end up with distinct field names because `choose` is not a Java keyword.

The class prim declares a static field oid^{ty} for each user-defined constant with built-in type. Nothing prevents the existence of two or three overloaded constants with the same identifier but different types among Bool, Int, and Char. If oid^{ty} is not overloaded, it translates to $it(oid)$. If it is overloaded, we append `$B` or `$I` or `$C` to it.

4.5.5 Method names

The methods declared in a class must have distinct names or argument types.

Normally, a method name oid^{ty} translates to $it(oid)$, as defined below. However, two methods with the same oid may end up with the same argument types. For example, there could be ops $\mathtt{m}^{ty,\mathsf{Int}\rightarrow ty} \neq \mathtt{m}^{\mathsf{Int},ty\rightarrow ty} \neq \mathtt{m}^{\mathsf{Int}\rightarrow ty}$ with $ty \in Ty_{\mathrm{U}}$, whose corresponding methods are $ty.\mathtt{m}^{ty,\mathsf{Int}\rightarrow ty} : \mathtt{int} \rightarrow ty$, $ty.\mathtt{m}^{\mathsf{Int},ty\rightarrow ty} : \mathtt{int} \rightarrow ty$, and $ty.\mathtt{m}^{\mathsf{Int}\rightarrow ty} : \mathtt{int} \rightarrow ty$ (the first two are instance methods, the third one is a static method). In these conflicting situations, the translated identifiers must be suitably disambiguated.

We capture conflicts via a predicate $\mathit{confl} \subseteq Mth$ on methods defined as

$$\frac{\begin{array}{c} mth = c.oid^{\overline{ty}\rightarrow ty} : tt(del(\overline{ty},h)) \rightarrow tt(ty) \in Mth_{\mathrm{PM}} \uplus Mth_{\mathrm{PMS}} \uplus Mth_{\mathrm{NPM}} \\ h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\} \\ mth' = c.oid^{\overline{ty}'\rightarrow ty'} : tt(del(\overline{ty}',h')) \rightarrow tt(ty') \in Mth_{\mathrm{PM}} \uplus Mth_{\mathrm{PMS}} \uplus Mth_{\mathrm{NPM}} \\ h' = \min\{h' \mid ty'_{h'} \in Ty_{\mathrm{U}}\} \\ mth \neq mth' \\ tt(del(\overline{ty},h)) = tt(del(\overline{ty}',h')) \end{array}}{\mathit{confl}(mth)}$$

$$\frac{\begin{array}{c} mth = c.oid^{\overline{ty}\to ty} : tt(del(\overline{ty},h)) \to tt(ty) \in Mth_{\mathrm{PM}} \uplus Mth_{\mathrm{NPM}} \\ h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\} \\ mth' = c.oid^{\overline{ty}'\to ty'} : tt(\overline{ty}') \to tt(ty') \in Mth_{\mathrm{UR}} \uplus Mth_{\mathrm{UA}} \uplus Mth_{\mathrm{C}} \\ tt(del(\overline{ty},h)) = tt(\overline{ty}') \end{array}}{\mathit{confl}(mth)}$$

$$\frac{mth = c.\mathtt{equals}^{ty,ty\to\mathsf{Bool}} : ty \to \mathtt{boolean} \in Mth_{\mathrm{PM}} \uplus Mth_{\mathrm{PMS}} \uplus Mth_{\mathrm{NPM}}}{\mathit{confl}(mth)}$$

$$\frac{\begin{array}{c} mth = c.oid^{\overline{ty}\to ty} : tt(\overline{ty}) \to tt(ty) \in Mth_{\mathrm{UA}} \\ mth' = c.oid^{\overline{ty}\to ty'} : tt(\overline{ty}) \to tt(ty') \in Mth_{\mathrm{UA}} \\ mth \neq mth' \end{array}}{\mathit{confl}(mth)}$$

$$\frac{\begin{array}{c} mth = c.oid^{\overline{ty}\to ty} : tt(\overline{ty}) \to tt(ty) \in Mth_{\mathrm{UA}} \\ mth' = c.oid^{\overline{ty}\to ty'} : tt(\overline{ty}) \to ty' \in Mth_{\mathrm{UR}} \end{array}}{\mathit{confl}(mth)}$$

$$\frac{\begin{array}{c} mth = \mathsf{prim}.oid^{\overline{ty}\to ty} : tt(\overline{ty}) \to tt(ty) \in Mth_{\mathrm{B}} \\ mth' = \mathsf{prim}.oid^{\overline{ty}\to ty'} : tt(\overline{ty}) \to tt(ty') \in Mth_{\mathrm{B}} \\ mth \neq mth' \end{array}}{\mathit{confl}(mth)}$$

Method name translation is captured by the function $mt : Mth \to \mathcal{J}$ defined as

$$\frac{mth \in Mth_{\mathrm{EP}} \uplus Mth_{\mathrm{ES}} \uplus Mth_{\mathrm{ER}} \uplus Mth_{\mathrm{EQ}} \uplus Mth_{\mathrm{ESS}} \uplus Mth_{\mathrm{EA}}}{mt(mth) = \mathtt{equals}}$$

$$\frac{mth = ty.c_i : \overline{ty}_i \to ty \in Mth_{\mathrm{C}}}{mt(mth) = \begin{cases} \mathtt{equals\$} & \text{if} \quad c_i = \mathtt{equals} \ \wedge \ \overline{ty}_i = ty \\ it(c_i) & \text{otherwise} \end{cases}}$$

$$\frac{mth \in Mth_{\mathrm{A}}}{mt(mth) = \mathtt{apply}}$$

$$\frac{\begin{array}{c} mth = c.oid^{\overline{ty}\to ty} : tt(del(\overline{ty},h)) \to tt(ty) \in Mth_{\mathrm{PM}} \uplus Mth_{\mathrm{PMS}} \uplus Mth_{\mathrm{NPM}} \\ h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\} \end{array}}{mt(mth) = \begin{cases} it(oid) & \text{if} \quad \neg\mathit{confl}(mth) \\ (it(oid), \mathtt{\$}, nt(h), trp(ty)) & \text{otherwise} \end{cases}}$$

$$\frac{mth = ty.oid^{\overline{ty}\to ty} : tt(\overline{ty}) \to ty \in Mth_{\mathrm{UR}}}{mt(mth) = it(oid)}$$

$$\frac{mth = c.oid^{\overline{ty}\to ty} : tt(\overline{ty}) \to tt(ty) \in Mth_{\mathrm{UA}}}{mt(mth) = \begin{cases} it(oid) & \text{if} \quad \neg confl(mth) \\ (it(oid), trp(ty)) & \text{otherwise} \end{cases}}$$

$$\frac{mth = \mathsf{prim}.oid^{\overline{ty}\to ty} : tt(\overline{ty}) \to tt(ty) \in Mth_{\mathrm{B}}}{mt(mth) = \begin{cases} (it(oid), trp(ty)) & \text{if} \quad \neg confl(mth) \\ it(oid) & \text{otherwise} \end{cases}}$$

$$\frac{\begin{array}{c} mth = ty_h.\mathsf{asrt}_{oid^{\overline{ty}\to ty}} : tt(del(\overline{ty}, h)) \to \texttt{boolean} \in Mth_{\mathrm{AsU}} \\ h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\} \end{array}}{mt(mth) = (\texttt{assert\$}, mt(ty_h.oid^{\overline{ty}\to ty} : tt(del(\overline{ty}, h)) \to tt(ty)))}$$

$$\frac{mth = ty.\mathsf{asrt}_{oid^{\overline{ty}\to ty}} : tt(\overline{ty}) \to \texttt{boolean} \in Mth_{\mathrm{AsUR}}}{mt(mth) = (\texttt{assert\$}, mt(ty.oid^{\overline{ty}\to ty} : tt(\overline{ty}) \to ty))}$$

$$\frac{mth = c.\mathsf{asrt}_{oid^{\overline{ty}\to ty}} : tt(\overline{ty}) \to \texttt{boolean} \in Mth_{\mathrm{AsUA}}}{mt(mth) = (\texttt{assert\$}, mt(c.oid^{\overline{ty}\to ty} : tt(\overline{ty}) \to tt(ty)))}$$

$$\frac{mth = \mathsf{prim.asrt}_{oid^{\overline{ty}\to ty}} : tt(\overline{ty}) \to \texttt{boolean} \in Mth_{\mathrm{AsB}}}{mt(mth) = (\texttt{assert\$}, mt(\mathsf{prim}.oid^{\overline{ty}\to ty} : tt(\overline{ty}) \to tt(ty)))}$$

A product class ty declares an equality method `eq` with argument type ty, which translates to `equals`.

A product class ty also declares a static method (in Mth_{UR}) $oid^{\overline{ty}\to ty}$ with argument types $tt(\overline{ty})$ for each op with $\overline{ty} \cap Ty_{\mathrm{U}} = \emptyset$, which translates to $it(oid)$. The assumptions on $\mathcal{P}$ guarantee that if two of these methods have the same oid then they have distinct argument types, because the corresponding ops have the same result type ty and thus must differ in their argument types. Moreover, the class argument type ty of `eq` differs from the argument types of these methods, which are not user-defined.

A product class ty' also declares a static method (in Mth_{UA}) $oid^{\overline{ty}\to ty}$ with argument types $tt(\overline{ty})$ for each op with $(\overline{ty}, ty) \cap Ty_{\mathrm{U}} = \emptyset$ and with ty' being the leftmost user-defined type occurring in (arrow types in) $(\overline{ty}, ty)$. In the absence of conflicts, this method translates to $it(oid)$. In the presence of conflicts, the assumptions on $\mathcal{P}$ guarantee that this method's result type differs from the conflicting methods'; thus, we append (the ASCII representation of) the result type to $it(oid)$. The class argument type ty' of `eq` differs from the argument types of this method, which are not user-defined.

A product class ty_h also declares an instance method (in Mth_{NPM}) $oid^{\overline{ty} \to ty}$ with argument types $tt(del(\overline{ty}, h))$ for each op with $h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\}$. In the absence of conflicts, this method translates to $it(oid)$. In the presence of conflicts, the assumptions on $\mathcal{P}$ guarantee that $oid^{\overline{ty} \to ty}$ differs from the conflicting ops in the position h of the leftmost user-defined argument type and/or in the result type ty; thus, we append (the ASCII representation of) the position h preceded by `$` and of the result type ty. For instance, the method $\mathtt{mm}^{\mathsf{Int}, ty_h \to \mathsf{Bool}}$ translates to `mm$2$B` if it conflicts with some other method; otherwise, just to `mm`.

A sum class ty declares an equality method `eq` with argument type ty, which translates to `equals`.

A sum class ty also declares a static method c_i with argument types $tt(\overline{ty}_i)$ for each non-constant constructor of ty. If c_i is `equals` and $\overline{ty}_i = ty$, we append `$` to it to make it distinct from the equality method. Otherwise, the method translates to $it(c_i)$.

A sum class ty also declares a static method (in Mth_{UR}) $oid^{\overline{ty} \to ty}$ with argument types $tt(\overline{ty})$ for each op with $\overline{ty} \cap Ty_{\mathrm{U}} = \emptyset$, which translates to $it(oid)$. Similarly to product classes above, these methods have names or argument types distinct from each other and from the equality method. Moreover, the assumptions on $\mathcal{P}$ guarantee that these methods have names or argument types distinct from the methods for the constructors c_i.

A sum class ty' also declares a static method (in Mth_{UA}) $oid^{\overline{ty} \to ty}$ with argument types $tt(\overline{ty})$ for each op with $(\overline{ty}, ty) \cap Ty_{\mathrm{U}} = \emptyset$ and with ty' being the leftmost user-defined type occurring in (arrow types in) $(\overline{ty}, ty)$. Similarly to product classes, in the absence of conflicts we translate the method to $it(oid)$; in the presence of conflicts, we append the result type.

A sum class ty_h also declares an instance method (in Mth_{PM} or Mth_{NPM}) $oid^{\overline{ty} \to ty}$ with argument types $tt(del(\overline{ty}, h))$ for each op with $h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\}$. Similarly to product classes, in the absence of conflicts we translate $oid^{\overline{ty} \to ty}$ to $it(oid)$; in the presence of conflicts, we append to $it(oid)$ the position h and the result type ty.

A summand class $\mathsf{sumd}_{c_i}^{ty}$ declares an equality method `eq` with argument type ty, which translates to `equals`.

A summand class $\mathsf{sumd}_{c_i}^{ty_h}$ also declares a method (in Mth_{PMS}) $oid^{\overline{ty} \to ty}$ with argument types $tt(del(\overline{ty}, h))$ for each op with $h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\}$. Similarly to product and sum classes, in the absence of conflicts the method translates to $it(oid)$; in the presence of conflicts, we append to $it(oid)$ the position h and the result type ty.

A restriction or quotient class ty declares an equality method `eq` with argument type ty, which translates to `equals`.

A restriction or quotient class ty also declares a static method (in Mth_{UR})

$oid^{\overline{ty} \to ty}$ with argument types $tt(\overline{ty})$ for each op with $\overline{ty} \cap Ty_{\mathrm{U}} = \emptyset$, which translates to $it(oid)$. Similarly to product and sum classes above, these static methods have names or argument types distinct from each other and from the equality method.

A restriction or quotient class ty' also declares a static method (in Mth_{UA}) $oid^{\overline{ty} \to ty}$ with argument types $tt(\overline{ty})$ for each op with $(\overline{ty}, ty) \cap Ty_{\mathrm{U}} = \emptyset$ and with ty' being the leftmost user-defined type occurring in (arrow types in) $(\overline{ty}, ty)$. Similarly to product and sum classes, in the absence of conflicts we translate the method to $it(oid)$; in the presence of conflicts, we append the result type.

A restriction or quotient class ty_h also declares an instance method (in Mth_{NPM}) $oid^{\overline{ty} \to ty}$ with argument types $tt(del(\overline{ty}, h))$ for each op with $h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\}$. Similarly to product, sum, and summand classes, in the absence of conflicts we translate $oid^{\overline{ty} \to ty}$ to $it(oid)$; in the presence of conflicts, we append to $it(oid)$ the position h and the result type ty.

An arrow class $\mathsf{arrow}_{ty}^{\overline{ty}}$ declares an application method apply with argument types $tt(\overline{ty})$, which translates to `apply`.

The class prim declares a method $oid^{\overline{ty} \to ty}$ with argument types $tt(\overline{ty})$ for each op with $(\overline{ty}, ty) \cap Ty_{\mathrm{U}} = \emptyset$ and no user-defined type occurring in (arrow types in) $(\overline{ty}, ty)$. The assumptions on $\mathcal{P}$ ensure that if two such ops have the same *oid* and the same argument types $\overline{ty}$, they must differ in their result type ty. Thus, we translate $oid^{\overline{ty} \to ty}$ to $it(oid)$ if there are no conflicts; otherwise, we append a representation of the result type.

All assertion methods are translated by prepending `assert$` to the translation of the corresponding methods. Since no non-assertion method name has `assert$` as a proper prefix ("proper" means that there is at least another character following `assert$`; note that an op called `assert` translates, in the absence of conflicts, to `assert$` because `assert` is a Java keyword) and since the translation of non-assertion methods yield methods with all distinct names and argument types within each class, all the assertion methods declared in a class have names and argument types distinct from each other and from non-assertion methods.

4.5.6 Variables

The variables used within a method or constructor (i.e. method/constructor parameters and, if the method is not abstract, local variables) must be distinct.

Variable translation is captured by the function $vt : V \xrightarrow{\mathrm{p}} \mathcal{J}$ defined as

$$
\begin{aligned}
vt(\mathsf{eqarg}) &= \mathtt{eqarg} \\
vt(\mathsf{eqargsub}) &= \mathtt{eqargSub} \\
vt(\mathsf{relax}) &= \mathtt{relax} \\
vt(\mathsf{choose}) &= \mathtt{choose} \\
v \in \mathcal{I} \;\Rightarrow\; vt(v) &= it(v) \\
vt(v_k) &= (it(v), \mathtt{\$}, nt(k)) \\
vt(\mathsf{arg}_j) &= (\mathtt{arg}, nt(j)) \\
vt(\mathsf{ires}_k) &= (\mathtt{ifres\$\$}, nt(k)) \\
vt(\mathsf{cres}_l) &= (\mathtt{caseres\$\$}, nt(l)) \\
vt(\mathsf{sub}_l^{c_i}) &= (\mathtt{sub\$\$}, it(c_i), \mathtt{\$\$}, nt(l)) \\
vt(\mathsf{tgt}_l) &= (\mathtt{target\$\$}, nt(l)) \\
\Delta(ty) = \textstyle\prod_i p_i \;\; ty_i \;\Rightarrow\; vt(p_i) &= it(p_i) \\
vt(\mathsf{fin}_v) &= (\mathtt{final\$\$}, it(v)) \\
vt(\mathsf{fin}_{v_k}) &= (\mathtt{final\$\$}, it(v)\mathtt{\$}, nt(k))
\end{aligned}
$$

The equality methods have parameter eqarg and those in summand classes with at least one field also declare a local variable eqargsub. By translating eqarg and eqargsub to `eqarg` and `eqargSub`, we have distinct variables within these methods.

A method derived from a non-constant constructor has parameters arg_j and declares no local variables. Thus, it is sufficient to translate the parameters to `arg1`, `arg2`, etc.

All the other methods, derived from user-defined ops, have parameters and local variables derived from the variables of the ops' defining terms, plus local variables introduced to store results of **if** and **case** as well as `switch` targets and results of summand casts. These variables are either simple *Fun* identifiers or have one of the forms v_k, ires_k, cres_l, $\mathsf{sub}_l^{c_i}$, tgt_l, fin_v, and fin_{v_k}. All we have to do is translate them to distinct Java identifiers. We translate v to $it(v)$, v_k by appending `$` and k to $it(v)$, $\mathsf{ires}_k/\mathsf{cres}_l/\mathsf{tgt}_l$ to `ifres$$1/caseres$$1/target$$1`, `ifres$$2/caseres$$2/target$$2`, etc., $\mathsf{sub}_l^{c_i}$ by appending $it(c_i)$ and l, separated by `$$`, to `sub`, and $\mathsf{fin}_v/\mathsf{fin}_{v_k}$ by prepending `final$$` to the translation of v/v_k.

The constructors of product classes have projectors p_i as parameters and declare no local variables; we translate their parameters to their corresponding Java identifiers via it. The constructors of restriction classes have one parameter relax and declare no local variable; we translate the parameter to `relax`. The constructors of quotient classes have one parameter choose and declare no local variable; we translate the parameter to `choose`. The constructors of summand classes have parameters arg_j and declare no local variables; we translate their parameters to `arg1`, `arg2`, etc.

4.5.7 The dollar character

The concrete name translation defined above makes extensive use of `$` (which is disallowed in *Fun* identifiers) to encode `?` (which is disallowed in Java identifiers) and to ensure name distinction within the various name spaces of the Java program. The resulting translation is relatively local, in the sense that most names translate to identifiers independently from other names, e.g. *Fun* variables v_k always translate to $(it(v), \texttt{\$}, nt(k))$.

In the absence of a character like `$`, disallowed in *Fun* identifiers but allowed in Java identifiers, a more complex and less local translation would be necessary. For instance, while $\texttt{x}_2$ could normally be translated to `x2` (instead of `x$2`), this would work only if the op's defining term where $\texttt{x}_2$ occurs does not happen to use a variable `x2` already. So, in general the translation of $\texttt{x}_2$ would have to depend on the other variables occurring in the term being translated.

5 Properties

We conjecture that the translation from *Fun* to Java defined in this document is correct, in the sense that the resulting Java program is accepted by any compliant Java compiler and that its execution on any compliant Java Virtual Machine is "equivalent" to (i.e. "simulates") the execution of the source *Fun* program.

In particular, the Java program will throw no exceptions during its execution, except when division by zero is attempted, when attempting to compare functions for equality, and when a non-existent **case** branch is reached; these circumstances would cause some kind of error in *Fun* as well.

6 Differences with Version 2

- The definition of *Fun* has been changed as follows:
  - a built-in type for characters has been added, along with built-in constants for characters and ops to convert between characters and integers;
  - higher-order features have been added, consisting of arrow types, abstractions, and applications of terms with arrow types;
  - the notion of constant has been made explicit;
  - inequalities have been added;
  - the requirement that **let** and **case** cannot appear in restriction terms has been removed.
- Our formal model of Java has been changed as follows:
  - characters have been added;
  - inequality has been added;
  - anonymous classes (in slightly restricted form, e.g. no fields) have been added;

  - casts to primitive types have been added;
  - final local variables with initializers (i.e. non-blank) have been added;
  - the unique error throwing statement has been differentiated into three kinds of errors.

- In each translation phase, the names used by the target program are now defined in terms of the actual source program, not just the names it uses. For instance, in the language translation phase, the c_i appearing in $\mathsf{sumd}^{ty}_{c_i}$ are explicitly required to be constructors of the source *Fun* program, as opposed to just names used by the program. This explicit dependency of names from the source program does not change the essence of the translation but makes it clearer where the new names come from.

- The language translation has been extended to cover characters as well as higher-order features.

- The type, variable, class, and (field) name arguments ty, v, c, and f of $\leadsto_{\mathrm{AsgNV}}$, $\leadsto_{\mathrm{AsgV}}$, and $\leadsto_{\mathrm{AsgF}}$ are always explicit for clarity (i.e. no longer left implicit).

- The language translation phase uniformly encapsulates the computation of assertions into methods; the newly added fourth translation phase inlines all assertion methods that consist of single expressions.

- Assertions derived from restriction terms are also checked by methods derived from non-existent branches of top-level **case** terms, just before throwing the exception.

- Different errors are thrown when a sum value has a tag that does not correspond to any summand and when it has a value that corresponds to a summand for which there is no branch in the **case**.