Translating Functional Programs to Java

Version 1

Alessandro Coglio

August 20, 2003

This document formally defines:

1. the abstract syntax and static semantics (i.e. typing) of a first-order, non-polymorphic functional programming language with product and sum 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

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

for booleans and integers. The types of the program are[1]

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

We define type products and sums as[2]

$$\begin{aligned} TyProd &= \{p_1\ ty_1 \times \cdots \times p_n\ ty_n \mid \overline{ty} \in Ty^* \ \wedge\ \overline{p} \in \mathcal{N}^{(*)}\} \\ TySum &= \{c_1\ \overline{ty}_1 + \cdots + c_n\ \overline{ty}_n \mid \overline{\overline{ty}} \in (Ty^*)^+ \ \wedge\ \overline{c} \in \mathcal{N}^{(+)}\} \end{aligned}$$

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.

Each user-defined type has a definition consisting in a type product or sum

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

If $\Delta(ty) \in TyProd$ (resp. $TySum$), ty is called a product (resp. sum) 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

$$\begin{aligned} Op_{\mathrm{B}} = &\ \{\mathsf{true}, \mathsf{false}, \mathsf{not}, \mathsf{and}, \mathsf{or}\} \\ &\cup \{\iota \in \mathbf{Z} \mid -2^{31} \leq \iota < 2^{31}\} \\ &\cup \{\mathsf{minus}, +, -, *, /, \mathsf{mod}\} \\ &\cup \{<, \leq, >, \geq\} \end{aligned}$$

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

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

The ops of the program are

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

Each op has zero or more argument types and a result type

$$\tau : Op \to \{\overline{ty} \to ty \mid \overline{ty} \in Ty^* \ \wedge\ ty \in Ty\}$$

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

[2]**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.

The built-in ops have types

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

Projectors and constructors have types

$$\begin{aligned}
ty = \textstyle\prod_i p_i \ ty_i \ &\Rightarrow \ \tau(p_i) = ty \to ty_i \\
ty = \textstyle\sum_i c_i \ \overline{ty}_i \ &\Rightarrow \ \tau(c_i) = \overline{ty}_i \to ty
\end{aligned}$$

1.4 Terms

A variable is a name. For clarity, we introduce the synonym

$$V = \mathcal{N}$$

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[3]

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

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

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

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

[3]**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).

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

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

$$\frac{\begin{array}{c} \Delta(ty) = \sum_i c_i\ \overline{ty}_i \\ t \in T^{cx}_{ty} \\ \forall i.\ \ t_i \in T^{cx[\overline{v}_i \mapsto \overline{ty}_i]}_{ty_0} \end{array}}{(\textbf{case}\ t\ \{c_1(\overline{v}_1) \to t_1, \ldots, c_n(\overline{v}_n) \to t_n\}) \in T^{cx}_{ty_0}} \quad \text{(pattern matching)}$$

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

Applications include projections (when $op \in Op_{\mathrm{P}}$) and constructions (when $op \in Op_{\mathrm{C}}$), as well as constant terms (when $\overline{ty} = \epsilon$, i.e. the empty sequence).

let and **case** terms introduce new variables into the contexts of some of their subterms; the newly introduced variables may shadow variables from the outer context.

The function[4] $FV : T \to \mathcal{P}_\omega(V)$ collects the free variables of a term

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

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

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

[4]**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.

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

1.5 Op definitions

A user-defined op is defined by (formal) parameters consisting in distinct variables

$$\pi : Op_{\mathrm{U}} \to V^{(*)}$$

and by a defining term

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

such that

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

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.

1.6 Program

The program is the 6-tuple

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

2 The translation, informally

2.1 Types

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

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

```
class P {
    int p;
    A q;
    ...
}
```

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. For example, $\Delta(\mathsf{S}) = \mathsf{c}\ (\mathsf{Bool}, \mathsf{A}) + \mathsf{d}$ translates to

```
abstract class S {
    ...
}

class S_c extends S {
```

```
    boolean arg1;
    A 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 {
    ...
}

class List_nil extends List {
    ...
}

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

2.2 Ops

The built-in ops of *Fun* translate to the obvious literals and operators of Java.

User-defined ops translate to fields if they are constants (i.e. they have no arguments), to methods otherwise.

If a user-defined 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, $\tau(\mathsf{m}) = \mathsf{Int}, \mathsf{A}, \mathsf{B} \rightarrow \mathsf{Bool}$ and $\pi(\mathsf{m}) = (\mathsf{i}, \mathsf{a}, \mathsf{b})$ translate to[5]

```
boolean A.m(int i, B b)
```

If the op is not a constant and all its argument types are built-in, it translates to a static method. If the op's result type is user-defined, the method is declared in the corresponding class, e.g. $\tau(\mathsf{n}) = \mathsf{Int} \rightarrow \mathsf{A}$ and $\pi(\mathsf{n}) = \mathsf{i}$ translate to

```
static A A.n(int i)
```

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

If instead the op's result type is built-in, the method is declared in a class used as a receptacle of all methods and fields resulting from ops whose argument and result types are all built-in (i.e primitive types in Java)

```
class Primitive {
    ....
}
```

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

```
static int Primitive.o(int i)
```

If the op is a constant, it translates to a static field. If the constant's type is user-defined, the field is declared in the corresponding class, e.g. $\tau(\mathsf{f}) = \mathsf{A}$ translates to

```
static A A.f
```

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

```
static boolean Primitive.g
```

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. For example, we have

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

static S S.d = new S_d();

S_c(boolean arg1, A arg2) {
    this.arg1 = arg1;
    this.arg2 = arg2;
}

S_d() { }
```

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`. Another exception is described later.

2.3.2 Application

The application of a built-in op translates to an expression involving the corresponding Java literal or operator. The application of a non-built-in op translates to an access to the corresponding field (if the op is a constant or a projector) or to a call to the corresponding method (if the op is not a constant or a projector).

For example, $\mathsf{x} + \mathsf{y}$ translates to `x+y`, $\mathsf{m}(\mathsf{i}, \mathsf{a}, \mathsf{b})$ translates to `a.m(i,b)`, `g` translates to `Primitive.g`, $\mathsf{p}(\mathsf{x})$ translates to `x.p`, and $\mathsf{c}(\mathsf{true}, \mathsf{a})$ translates to `S.c(true,a)`.

2.3.3 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 p, A q) {
    this.p = p;
    this.q = q;
}
```

and the tuple $\{\mathsf{p} \leftarrow 2, \mathsf{q} \leftarrow \mathsf{a}\}$ translates to `new P(2,a)`.

2.3.4 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 A.equals(A 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.p == eqarg.p &&
            this.q.equals(eqarg.q));
}
```

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 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 instanceof S_c)) return false;
    else {
        S_c eqargSub = (S_c) eqarg;
        return (this.arg1 == eqargSub.arg1 &&
                this.arg2.equals(eqargSub.arg2));
    }
}

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

For a constant constructor, it is sufficient (and faster) to compare the argument with the static field corresponding to the constructor, which references the unique object representing that constructor. The object is unique because a constant constructor always translates to an access of the corresponding static field, which is initialized with a reference to the object.

2.3.5 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** $\mathsf{x} \leftarrow$ (**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** $\mathsf{x} \leftarrow$ `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` in the same body.

A draconian but simple approach to avoid all these problems is to always use distinct **let** variables within the term that defines an op. So, before translating the *Fun* program to Java, we appropriately rename **let** variables to make them all distinct. This approach can be refined, in the future, to rename only those variables that would interfere; those that do not interfere (as in the above example) can be left unchanged.

2.3.6 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** $(\mathsf{x} = 6)$ $(\textbf{let}\ \mathsf{y} \leftarrow 2\ \textbf{in}\ \mathsf{y}{*}\mathsf{y})$ $(\textbf{let}\ \mathsf{z} \leftarrow 3\ \textbf{in}\ \mathsf{minus}\ \mathsf{z})$ 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.

2.3.7 Pattern matching

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 other 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.

This translation 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. On the other hand, if $\tau(\mathsf{fact}) = \mathsf{List} \rightarrow \mathsf{Int}$ and $\pi(\mathsf{fact}) = \mathsf{list}$,

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

(which baroquely computes the factorial of the length of a list) is realized by means of an auxiliary method that is dynamically dispatched and that is called by the method for the op

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

abstract int List.factAux(int x);

int List_nil.factAux(int x) {
    return 1;
}

int List_cons.factAux(int x) {
    return (x * this.arg2.fact());
}
```

The free variables in the **case** become (extra) parameters to the auxiliary method, e.g. the variable x above becomes the parameter `x`.

Thus, before translating the *Fun* program to Java, we lift all pattern matching to the top level, introducing suitable auxiliary ops that are called where the **case** terms originally are. For the last example, we introduce an auxiliary op $\mathsf{factAux}$ with $\tau(\mathsf{factAux}) = \mathsf{List}, \mathsf{Int} \rightarrow \mathsf{Int}$, $\pi(\mathsf{factAux}) = (\mathsf{list}, \mathsf{x})$, and

$$\delta(\mathsf{factAux}) = \mathbf{case}\ \mathsf{list}\ \{\mathsf{nil} \rightarrow 1, \mathsf{cons}(\mathsf{head},\mathsf{tail}) \rightarrow \mathsf{x} * \mathsf{fact}(\mathsf{tail})\}$$

In addition, we change to

$$\delta(\mathsf{fact}) = \mathbf{let}\ \mathsf{x} \leftarrow \mathsf{length}(\mathsf{list})\ \mathbf{in}\ \mathsf{factAux}(\mathsf{list}, \mathsf{x})$$

After all pattern matching has been lifted to the top level in this manner, the newly introduced ops translate to dynamically dispatched methods. In the last example, we obtain the Java code shown earlier.

2.4 Defining terms

The defining term of a non-constant op translates to the body of the corresponding method(s), as in the pattern matching translation examples given earlier. 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 $\tau(\mathsf{r}) = \mathsf{P}, \mathsf{Int} \rightarrow \mathsf{Int}$ and $\pi(\mathsf{r}) = (\mathsf{x}, \mathsf{i})$,

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

translates to

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

In two cases, we use a slightly different strategy that produces equivalent but more natural and idiomatic code. If the expression is a conditional one, we turn it into an `if-then-else` whose branches return the subexpressions, e.g. instead of

```
static int Primitive.r1(int i) {
    int j = i - 1;
    return ((j < 0) ? (-j) : j);
}
```

we produce

```
static int Primitive.r1(int i) {
    int j = i - 1;
    if (j < 0) {
        return (-j);
    } else {
        return j;
    }
}
```

(which tends to be more readable when the expressions are longer). If the expression is an `ifres` variable set in a preceding `if-then-else`, we eliminate the `ifres` variables incorporating the `return` into the branches, e.g. instead of

```
static int Primitive.r2(int i) {
    int j = 2 * i;
    int ifres;
```

```
    if (j >= 8) {
      int k = i + j;
      ifres = k * k;
    } else {
      ifres = 0;
    }
    return ifres;
}
```

we produce

```
static int Primitive.r2(int i) {
    int j = 2 * i;
    if (j >= 8) {
        int k = i + j;
        return (k * k);
    } else {
        return 0;
    }
}
```

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,

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

translates to

```
static int Primitive.s = 7;
```

while

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

translates to

```
static {
    int i = Primitive.s + 2;
    Primitive.t = 3 * i;
}
```

For static initializers that assign conditional expressions or `ifres`, we use the modified strategy that we use for methods, producing equivalent but more natural and idiomatic code. So, instead of

```
static {
    int h = Primitive.s * 2;
    Primitive.t1 = ((h < 0) ? (-h) : h);
}
```

we produce

```
static {
    int h = Primitive.s * 2;
    if (h < 0) {
        Primitive.t1 = -h;
    } else {
        Primitive.t1 = h;
    }
}
```

and instead of

```
static {
    int ifres;
    if (Primitive.s != 5) {
        int h = Primitive.s * 2;
        ifres = h / 3;
    } else {
        ifres = 1;
    }
    Primitive.t2 = ifres;
}
```

we produce

```
static {
    if (Primitive.s != 5) {
        int h = Primitive.s * 2;
        Primitive.t2 = h / 3;
    } else {
        Primitive.t2 = 1;
    }
}
```

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 by 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

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

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, $\mathit{ext}(c) = \mathsf{none}$ 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).

Whether a class is declared abstract is captured by the predicate

$$\mathit{abs}_{\mathrm{C}} \subseteq C$$

3.3 Types

We only consider two primitive types

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

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

$$\mathit{Ty} = \mathit{PTy} \uplus C$$

3.4 Fields

A Java program has a finite set of fields

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

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

$$\mathit{stc}_{\mathrm{F}} \subseteq \mathit{Fld}$$

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`.

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

Variables are defined as in *Fun*

$$V = \mathcal{N}$$

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

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

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

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

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

$$\frac{e_1, e_2 \in E \quad \odot \in \{\texttt{\&\&}, \texttt{||}\}}{(e_1 \odot e_2) \in E} \quad \text{(binary logical)}$$

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

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

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

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

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

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

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

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

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

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

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

$$\frac{c \in C \quad e \in E}{((c)\ e) \in E} \quad \text{(cast)}$$

$$\frac{e \in E \quad c \in C}{(e\ \texttt{instanceof}\ c) \in E} \quad \text{(class comparison)}$$

3.9 Statements

The set S of statements is defined as

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

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

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

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

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

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

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

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

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

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

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

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.

3.10 Parameters

Each method and each constructor has (formal) parameters consisting in variables

$$\mathit{param} : \mathit{Mth} \cup \mathit{Con} \to V^*$$

3.11 Bodies

Each non-abstract method and each constructor has a body consisting in a statement

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

3.12 Static (field) initializers

Some static fields have initializers consisting in expressions

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

Each class has a finite set of static initializers consisting in statements

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

3.13 Program

The program is the 13-tuple

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

4 The translation, formally

The translation from *Fun* to Java consists of three phases. First, all pattern matching is lifted to the top level. Next, **let** variables are made distinct within each op's defining term. These two phases take place within *Fun*; their purpose is to make the program amenable to the last phase, namely the language translation to Java.

4.1 Pattern matching lifting

Consider an arbitrary *Fun* program

$$\mathcal{P} = \langle \mathit{Ty}_{\mathrm{U}}, \Delta, \mathit{Op}_{\mathrm{U}}, \tau, \pi, \delta\rangle$$

The result of pattern matching lifting is the *Fun* program

$$\mathcal{P}' = \langle \mathit{Ty}'_{\mathrm{U}}, \Delta', \mathit{Op}'_{\mathrm{U}}, \tau', \pi', \delta'\rangle$$

defined as follows.

4.1.1 Names

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

$$\mathcal{N}' = \mathcal{N} \uplus \{\mathsf{aux}^{op}_k \mid op \in \mathcal{N} \ \wedge \ k \in \mathbf{N}_+\}$$

i.e. besides the names in $\mathcal{N}$, $\mathcal{P}'$ uses names obtained by tagging aux by names in $\mathcal{N}$ (which, as it turns out, will be ops) and positive naturals.

This phase of the translation, like the other two, is parameterized over the set $\mathcal{N}$ of names used by the source program $\mathcal{P}$. In fact, the set $\mathcal{N}'$ of names used by the target program is defined in terms of $\mathcal{N}$.

4.1.2 Term transformation and auxiliary op introduction

We replace every **case** that is not at the top level or that does not operate on the leftmost parameter with user-defined type, with an application of a newly introduced, auxiliary op. The defining term of this auxiliary op is, roughly speaking, the **case** that gets replaced with the application.

We capture this process by means of a function that maps a term to the transformed term (which does not have any pattern matching) plus the set of the needed auxiliary ops, accompanied by their types and definitions. The function is recursively defined over terms.

How do we name the auxiliary ops? We utilize the new names of the form aux_k^{op}. The op part of this name is the op of $\mathcal{P}$ whose defining term is being transformed. The k part is a numeric index that is incremented as we traverse a term; it is threaded through the transformation function.

It is useful to introduce a notion of 5-tuples that completely describe the auxiliary ops

$$\begin{aligned} \mathit{AuxOp} = \{\langle op, \overline{ty}, ty, \overline{v}, t\rangle \mid \\ op \in \mathcal{N}' \ \wedge \ \overline{ty} \in \mathit{Ty}^* \ \wedge \ ty \in \mathit{Ty} \ \wedge \ \overline{v} \in (V')^* \ \wedge \ t \in T'\} \end{aligned}$$

Each tuple $\langle op, \overline{ty}, ty, \overline{v}, t\rangle$ captures an op op with types $\overline{ty} \to ty$, parameters $\overline{v}$, and defining term t. We use V' and T' because those variables and terms use names in $\mathcal{N}'$, and thus belong to $\mathcal{P}'$; we just use Ty because, as defined below, it coincides with Ty'.

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

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

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

For purely cosmetic reasons, we define the lifting function as a 7-ary relation

$$\rightsquigarrow \subseteq \mathit{Op}_{\mathrm{U}} \times \mathit{Cx} \times T \times \mathbf{N}_+ \times T' \times \mathbf{N}_+ \times \mathcal{P}_\omega(\mathit{AuxOp})$$

The meaning of ($t \ \ k \ \ \overset{op,cx}{\rightsquigarrow} \ \ t' \ \ k' \ \ \widetilde{ao}$) is that the result of transforming the subterm t of $\delta(op)$ with context cx, when the currently available index is k, is the term t', that the next available index is k', and that the transformation gives rise to the auxiliary ops described by $\widetilde{ao}$. For readability, op and cx may be left implicit.

This relation is functional; the relational form just makes the rules below more readable by having the transformation look like rewriting

$$\frac{}{v \ \ k \ \rightsquigarrow \ v \ \ k \ \ \emptyset}$$

$$\frac{\forall i. \ \ t_i \ \ k_{i-1} \ \rightsquigarrow \ t'_i \ \ k_i \ \ \widetilde{ao}_i}{op'(\overline{t}) \ \ k_0 \ \rightsquigarrow \ op'(\overline{t'}) \ \ k_n \ \ (\bigcup_i \widetilde{ao}_i)}$$

$$\frac{\forall i.\;\; t_i \;\; k_{i-1} \;\rightsquigarrow\; t'_i \;\; k_i \;\; \widetilde{ao}_i}{\{p_i \leftarrow t_i\}_i \;\; k_0 \;\rightsquigarrow\; \{p_i \leftarrow t'_i\}_i \;\; k_n \;\; (\bigcup_i \widetilde{ao}_i)}$$

$$\frac{\begin{array}{c} t_1 \;\; k_0 \;\rightsquigarrow\; t'_1 \;\; k_1 \;\; \widetilde{ao}_1 \\ t_2 \;\; k_1 \;\rightsquigarrow\; t'_2 \;\; k_2 \;\; \widetilde{ao}_2 \end{array}}{(t_1 = t_2) \;\; k_0 \;\rightsquigarrow\; (t'_1 = t'_2) \;\; k_2 \;\; (\widetilde{ao}_1 \cup \widetilde{ao}_2)}$$

$$\frac{\begin{array}{c} t_0 \;\; k \;\rightsquigarrow\; t'_0 \;\; k_0 \;\; \widetilde{ao}_0 \\ t_1 \;\; k_0 \;\rightsquigarrow\; t'_1 \;\; k_1 \;\; \widetilde{ao}_1 \\ t_2 \;\; k_1 \;\rightsquigarrow\; t'_2 \;\; k_2 \;\; \widetilde{ao}_2 \end{array}}{(\mathbf{if}\; t_0\; t_1\; t_2) \;\; k \;\rightsquigarrow\; (\mathbf{if}\; t'_0\; t'_1\; t'_2) \;\; k_2 \;\; (\widetilde{ao}_0 \cup \widetilde{ao}_1 \cup \widetilde{ao}_2)}$$

$$\frac{\begin{array}{c} t_0 \in T^{cx}_{ty_0} \\ t_0 \;\; k \;\rightsquigarrow\; t'_0 \;\; k_0 \;\; \widetilde{ao}_0 \\ t \;\; k_0 \;\overset{op,cx[v\mapsto ty_0]}{\rightsquigarrow}\; t' \;\; k' \;\; \widetilde{ao} \end{array}}{(\mathbf{let}\; v \leftarrow t_0 \;\mathbf{in}\; t) \;\; k \;\rightsquigarrow\; (\mathbf{let}\; v \leftarrow t'_0 \;\mathbf{in}\; t') \;\; k' \;\; (\widetilde{ao}_0 \cup \widetilde{ao})}$$

$$\frac{\begin{array}{c} t \in T^{cx}_{ty} \\ \Delta(ty) = \sum_i c_i \; \overline{ty}_i \\ t \;\; (k+1) \;\rightsquigarrow\; t' \;\; k_0 \;\; \widetilde{ao} \\ t_i \in T^{cx[\overline{v}_i \mapsto \overline{ty}_i]}_{ty_0} \\ \forall i.\;\; t_i \;\; k_{i-1} \;\overset{op,cx[\overline{v}_i \mapsto \overline{ty}_i]}{\rightsquigarrow}\; t'_i \;\; k_i \;\; \widetilde{ao}_i \\ \overline{v} = \mathit{order}(\bigcup_i (FV(t_i) - \overline{v}_i)) \\ \widetilde{ao}' = \{\langle \mathsf{aux}^{op}_k, (ty, cx(\overline{v})), ty_0, (\mathsf{aux}^{op}_k, \overline{v}), \mathbf{case}\; \mathsf{aux}^{op}_k \;\{c_i(\overline{v}_i) \to t'_i\}_i \rangle\} \end{array}}{(\mathbf{case}\; t\; \{c_i(\overline{v}_i) \to t_i\}_i) \;\; k \;\rightsquigarrow\; \mathsf{aux}^{op}_k(t', \overline{v}) \;\; k_n \;\; \left(\widetilde{ao} \cup (\bigcup_i \widetilde{ao}_i) \cup \widetilde{ao}'\right)}$$

It is easy to see that if ($t \;\; k \;\rightsquigarrow\; t' \;\; k' \;\; \widetilde{ao}$) then $FV(t) = FV(t')$.

Most rules are straightforward: subterms are recursively transformed, indices are threaded through, and auxiliary ops are collected.

The interesting rule is the last one. First, the currently available index k is reserved for the auxiliary op about to be created. Next, all subterms are transformed, threading $k+1$ through (so that additional ops created for subterms use different indices). We (re-)use aux^{op}_k as the first parameter of the auxiliary op aux^{op}_k, since ops and variables live in separate name spaces. The other parameters of aux^{op}_k are derived from the free variables of the branches, excluding the variables bound therein. Since op parameters must be ordered, we order those free variables before making them additional parameters of aux^{op}_k. The argument types of aux^{op}_k are the sum type followed by the types of the free variables just ordered (determined by the context of the **case**). The result type of aux^{op}_k is the type of the branch terms. The defining term of aux^{op}_k does pattern matching on the first parameter; each branch returns the transformed branch term. The original **case** is turned into an application of aux_k^{op}: the first argument is the transformed pattern matching target; the other arguments are the free variables of the branches.

If the defining term of an op of $\mathcal{P}$ already does pattern matching at the top level on its leftmost parameter with user-defined type, there is no need to introduce an extra level of indirection; all we need to do is transform the branch subterms. Otherwise, we transform the defining term altogether. These two cases are captured by the function

$$\mathit{lift} : \mathit{Op}_{\mathrm{U}} \to T \times \mathcal{P}_\omega(\mathit{AuxOp})$$

defined as

$$\frac{\begin{array}{c}\tau(op) = \overline{ty} \to ty \\ \pi(op) = \overline{v} \\ h = \min\{h \mid ty_h \in \mathit{Ty}_{\mathrm{U}}\} \\ \delta(op) = (\mathbf{case}\ v_h\ \{c_i(\overline{v}_i) \to t_i\}_i) \\ \Delta(ty_h) = \sum_i c_i\ \overline{ty}_i \\ k_0 = 1 \\ \forall i.\ \ t_i\ \ k_{i-1} \overset{op,\{\overline{v}\mapsto\overline{ty}\}[\overline{v}_i\mapsto\overline{ty}_i]}{\rightsquigarrow} t'_i\ \ k_i\ \ \widetilde{ao}_i\end{array}}{\mathit{lift}(op) = \langle \mathbf{case}\ v_h\ \{c_i(\overline{v}_i) \to t'_i\}_i, \bigcup_i \widetilde{ao}_i\rangle}$$

$$\frac{\begin{array}{c}\tau(op) = \overline{ty} \to ty \\ \pi(op) = \overline{v} \\ h = \min\{h \mid ty_h \in \mathit{Ty}_{\mathrm{U}}\} \ \Rightarrow\ \delta(op) \neq (\mathbf{case}\ v_h\ \{c_i(\overline{v}_i) \to t_i\}_i) \\ \delta(op)\ \ 1 \overset{op,\{\overline{v}\mapsto\overline{ty}\}}{\rightsquigarrow}\ t\ \ k\ \ \widetilde{ao}\end{array}}{\mathit{lift}(op) = \langle t, \widetilde{ao}\rangle}$$

4.1.3 Transformed program

The types and their definitions are unchanged

$$\mathit{Ty}'_{\mathrm{U}} = \mathit{Ty}_{\mathrm{U}} \qquad\qquad \Delta'(ty) = \Delta(ty)$$

We add the auxiliary ops as user-defined ops

$$\mathit{Op}'_{\mathrm{U}} = \mathit{Op}_{\mathrm{U}} \uplus \{\mathsf{aux}_k^{op} \mid \mathit{lift}(op) = \langle \ldots, \widetilde{ao}\rangle\ \wedge\ \langle \mathsf{aux}_k^{op}, \ldots\rangle \in \widetilde{ao}\}$$

The types and definitions of the old user-defined ops are unchanged

$$op \in \mathit{Op}_{\mathrm{U}}\ \Rightarrow\ \tau'(op) = \tau(op)\ \wedge\ \pi'(op) = \pi(op)\ \wedge\ \delta'(op) = \delta(op)$$

while the types and definitions of the new auxiliary ops are

$$\begin{array}{c}\mathit{lift}(op) = \langle \ldots, \widetilde{ao}\rangle\ \wedge\ \langle \mathsf{aux}_k^{op}, \overline{ty}, ty, \overline{v}, t\rangle \in \widetilde{ao}\ \Rightarrow \\ \tau'(\mathsf{aux}_k^{op}) = \overline{ty} \to ty\ \wedge\ \pi'(\mathsf{aux}_k^{op}) = \overline{v}\ \wedge\ \delta'(\mathsf{aux}_k^{op}) = t\end{array}$$

So, the resulting program $\mathcal{P}'$ is such that all pattern matching is at the top level and operates on the leftmost parameters with user-defined types.

4.2 Variable renaming

Consider an arbitrary *Fun* program

$$\mathcal{P} = \langle \mathit{Ty}_{\mathrm{U}}, \Delta, \mathit{Op}_{\mathrm{U}}, \tau, \pi, \delta \rangle$$

The result of variable renaming is the *Fun* program

$$\mathcal{P}' = \langle \mathit{Ty}'_{\mathrm{U}}, \Delta', \mathit{Op}'_{\mathrm{U}}, \tau', \pi', \delta' \rangle$$

defined as follows.

While in our overall translation this transformation is applied to the program resulting from pattern matching lifting, the transformation does not require pattern matching to be at the top level. In other words, it is applicable to any *Fun* program.

4.2.1 Names

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

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

i.e. besides the names in $\mathcal{N}$, $\mathcal{P}'$ uses names obtained by tagging names in $\mathcal{N}$ (which, as it turns out, will be variables) with positive naturals.

4.2.2 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 similar to the previous transformation, we define this transformation via a functional 4-ary relation

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

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

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

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

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

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

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

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

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

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

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, 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 subterms of the **let** are recursively 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 variable is not changed in t_0 because such a term is outside the scope of the bound variable. This term and the term resulting from substituting the variable in the other subterm are then recursively 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 all pattern matching is lifted to the top level before translating to Java. Thus, variables bound by **case** do not interfere with **let** variables.

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

4.2.3 Transformed program

The types and their definitions are unchanged

$$Ty' = Ty \qquad\qquad \Delta'(ty) = \Delta(ty)$$

The ops and their types and parameters are also unchanged

$$Op'_{\mathrm{U}} = Op_{\mathrm{U}} \qquad\qquad \tau'(op) = \tau(op) \qquad\qquad \pi'(op) = \pi(op)$$

What changes are the ops' defining terms

$$(\delta(op) \;\; \overline{v} \; \rightsquigarrow \; t \;\; \widetilde{v}) \;\Rightarrow\; \delta'(op) = t$$

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

4.3 Language translation

Consider a *Fun* program

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

such that (1) all pattern matching is at the top level and operates on the leftmost parameters with user-defined types and (2) **let** variables are distinct within each op's defining term, as resulting from the previous two translation phases.

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.

4.3.1 Names

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

$$\begin{aligned} \mathcal{N}' = \mathcal{N} & \\ & \uplus \{\mathsf{sumd}^{ty}_{c_i} \mid ty, c_i \in \mathcal{N}\} \\ & \uplus \{\mathsf{arg}_j \mid j \in \mathbf{N}_+\} \\ & \uplus \{\mathsf{ifres}_k \mid k \in \mathbf{N}_+\} \\ & \uplus \{\mathsf{prim}, \mathsf{eq}, \mathsf{eqarg}, \mathsf{eqargsub}\} \end{aligned}$$

The use of the additional names is explicated below.

4.3.2 Types

$\mathcal{P}'$ 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

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

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

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

Only the sum classes are abstract

$$\mathit{abs}_{\mathrm{C}}(c) \ \Leftrightarrow\ \Delta(c) \in \mathit{TySum}$$

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

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

4.3.3 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 field for each constant constructor, declared in the sum class

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

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

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

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

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

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

$$Fld = Fld_{\mathrm{P}} \uplus Fld_{\mathrm{CA}} \uplus Fld_{\mathrm{CC}} \uplus Fld_{\mathrm{CB}} \uplus Fld_{\mathrm{CU}}$$

The only static fields are those for constant constructors and constants

$$stc_{\mathrm{F}}(fld) \;\Leftrightarrow\; fld \in Fld_{\mathrm{CC}} \uplus Fld_{\mathrm{CB}} \uplus Fld_{\mathrm{CU}}$$

4.3.4 Methods

There is an equality method declared in each product or sum class

$$\begin{aligned} Mth_{\mathrm{EP}} &= \{ty.\mathsf{eq} : ty \rightarrow \texttt{boolean} \mid \Delta(ty) \in TyProd\} \\ Mth_{\mathrm{ES}} &= \{ty.\mathsf{eq} : ty \rightarrow \texttt{boolean} \mid \Delta(ty) \in TySum\} \end{aligned}$$

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

$$Mth_{\mathrm{ESS}} = \{\mathsf{sumd}^{ty}_{c_i}.\mathsf{eq} : ty \rightarrow \texttt{boolean} \mid \Delta(ty) = \sum_i c_i\ \overline{ty}_i\}$$

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

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

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

$$\begin{aligned} Mth_{\mathrm{B}} = \{&\mathsf{prim}.op : tt(\overline{ty}) \rightarrow tt(ty) \mid \\ &op \in Op_{\mathrm{U}} \ \wedge\ \tau(op) = \overline{ty} \rightarrow ty \ \wedge\ \overline{ty} \in Ty^{+}_{\mathrm{B}} \ \wedge\ ty \in Ty_{\mathrm{B}}\} \end{aligned}$$

There is a method for each non-constant op with all built-in argument types but user-defined result type, declared in the class for that user-defined type

$$\begin{aligned} Mth_{\mathrm{BA}} = \{&ty.op : tt(\overline{ty}) \rightarrow ty \mid \\ &op \in Op_{\mathrm{U}} \ \wedge\ \tau(op) = \overline{ty} \rightarrow ty \ \wedge\ \overline{ty} \in Ty^{+}_{\mathrm{B}} \ \wedge\ ty \in Ty_{\mathrm{U}}\} \end{aligned}$$

There are methods for each op with at least a user-defined argument type and whose defining term is a **case**. Recall that, by virtue of the first translation phase, the pattern matching target is the leftmost parameter with user-defined type, which is a sum type. A method is declared in the sum class and a method is declared in each subclass[6]

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

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

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

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

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

$$\begin{aligned} Mth = {} & Mth_{\mathrm{EP}} \uplus Mth_{\mathrm{ES}} \uplus Mth_{\mathrm{ESS}} \\ & \uplus Mth_{\mathrm{C}} \uplus Mth_{\mathrm{B}} \uplus Mth_{\mathrm{BA}} \\ & \uplus Mth_{\mathrm{PM}} \uplus Mth_{\mathrm{PMS}} \uplus Mth_{\mathrm{NPM}} \end{aligned}$$

The only static methods are those for constructors and those for ops with all built-in argument types

$$stc_{\mathrm{M}}(mth) \ \Leftrightarrow \ mth \in Mth_{\mathrm{C}} \uplus Mth_{\mathrm{B}} \uplus Mth_{\mathrm{BA}}$$

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

$$abs_{\mathrm{M}}(mth) \ \Leftrightarrow \ mth \in Mth_{\mathrm{ES}} \uplus Mth_{\mathrm{PM}}$$

4.3.5 Constructors

There is a constructor in every product class and one in every summand class

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

4.3.6 Parameters

The methods have parameters

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

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

$$\frac{\begin{array}{c} mth = c.op : \overline{ty} \rightarrow ty \in Mth_{\mathrm{B}} \uplus Mth_{\mathrm{BA}} \\ \pi(op) = \overline{v} \end{array}}{param(mth) = \overline{v}}$$

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

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

$$\begin{array}{r} con = ty : \overline{ty} \in Con_{\mathrm{P}} \ \wedge \ ty = \prod_i p_i \ ty_i \ \Rightarrow \ param(con) = \overline{p} \\ con = \mathsf{sumd}^{ty}_{c_i} : \overline{ty}_i \in Con_{\mathrm{S}} \ \Rightarrow \ param(con) = \overline{\mathsf{arg}} \end{array}$$

For product constructors, we use the projectors as parameters. For summand constructors, we use parameters arg_j.

4.3.7 Translation of terms to expressions and statements

Each *Fun* term translates to a Java expression preceded by a Java statement. The statement assigns values to the 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 whose defining term is a **case** translates to methods of the summand classes, the variables bound in each branch translate to field accesses in the corresponding class. To capture the translation of a finite number of *Fun* variables to `this` or to field accesses, we use translation contexts

$$TC = V \xrightarrow{\mathrm{f}} \{\texttt{this}\} \uplus \{\texttt{this}.f \mid f \in \mathcal{N}'\}$$

Since a (**if** ...) term may translate to (`if` ...), we need to generate a fresh local variable to store the result computed by the two branches. We do that by threading a positive natural while we traverse and translate the terms.

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

$$\begin{array}{l} ty \in Ty_{\mathrm{B}} \ \Rightarrow \ eq_{ty}(e_1, e_2) = (e_1 == e_2) \\ ty \in Ty_{\mathrm{U}} \ \Rightarrow \ eq_{ty}(e_1, e_2) = e_1.\mathsf{eq}(e_2) \end{array}$$

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.

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

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

The translation of terms to expressions preceded by statements is captured by a 7-ary functional relation

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

The meaning of $(t \;\; k \overset{tc,cx}{\rightsquigarrow} \;\; s \;\; e \;\; k')$ is that, in the translation context tc, the result of translating the term t with context cx when the currently available index for conditional result variables is k, is the expression e preceded by the statement s and that the next available index is k'. For readability, tc and cx may be left implicit.

The relation is defined as

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

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

$$\frac{}{\textsf{true} \;\; k \rightsquigarrow \texttt{mts} \;\; \texttt{true} \;\; k}$$

$$\frac{}{\textsf{false} \;\; k \rightsquigarrow \texttt{mts} \;\; \texttt{false} \;\; k}$$

$$\frac{}{\iota \;\; k \rightsquigarrow \texttt{mts} \;\; \iota \;\; k}$$

$$\frac{t \;\; k \rightsquigarrow s \;\; e \;\; k'}{(\textsf{not}\; t) \;\; k \rightsquigarrow s \;\; (!\, e) \;\; k'}$$

$$\frac{t \;\; k \rightsquigarrow s \;\; e \;\; k'}{(\textsf{minus}\; t) \;\; k \rightsquigarrow s \;\; (-e) \;\; k'}$$

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

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

$$\frac{\begin{array}{c} op \in Op_{\mathrm{U}} \\ \tau(op) = ty \in Ty_{\mathrm{U}} \end{array}}{op \;\; k \;\leadsto\; \mathtt{mts} \;\; ty.op \;\; k}$$

$$\frac{\begin{array}{c} op \in Op_{\mathrm{U}} \\ \tau(op) = \overline{ty} \to ty \\ \overline{ty} \in Ty_{\mathrm{B}}^{+} \;\wedge\; ty \in Ty_{\mathrm{B}} \\ \forall i.\;\; t_i \;\; k_{i-1} \;\leadsto\; s_i \;\; e_i \;\; k_i \end{array}}{op(\overline{t}) \;\; k_0 \;\leadsto\; s_1; \ldots; s_n \;\; \mathsf{prim}.op(\overline{e}) \;\; k_n}$$

$$\frac{\begin{array}{c} op \in Op_{\mathrm{U}} \\ \tau(op) = \overline{ty} \to ty \\ \overline{ty} \in Ty_{\mathrm{B}}^{+} \;\wedge\; ty \in Ty_{\mathrm{U}} \\ \forall i.\;\; t_i \;\; k_{i-1} \;\leadsto\; s_i \;\; e_i \;\; k_i \end{array}}{op(\overline{t}) \;\; k_0 \;\leadsto\; s_1; \ldots; s_n \;\; ty.op(\overline{e}) \;\; k_n}$$

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

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

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

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

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

$$\frac{t_1, t_2 \in T^{cx}_{ty} \qquad t_1\ \ k_0\ \rightsquigarrow\ s_1\ \ e_1\ \ k_1 \qquad t_2\ \ k_1\ \rightsquigarrow\ s_2\ \ e_2\ \ k_2}{(t_1 = t_2)\ \ k_0\ \rightsquigarrow\ s_1; s_2\ \ eq_{ty}(e_1, e_2)\ \ k_2}$$

$$\frac{t_0\ \ k\ \rightsquigarrow\ s_0\ \ e_0\ \ k' \qquad t_1\ \ k'\ \rightsquigarrow\ \texttt{mts}\ \ e_1\ \ k' \qquad t_2\ \ k'\ \rightsquigarrow\ \texttt{mts}\ \ e_2\ \ k'}{(\textbf{if}\ t_0\ t_1\ t_2)\ \ k\ \rightsquigarrow\ s_0\ \ (e_0\ ?\ e_1 : e_2)\ \ k'}$$

$$\frac{\begin{array}{c} t_1, t_2 \in T^{cx}_{ty} \\ t_0\ \ k+1\ \rightsquigarrow\ s_0\ \ e_0\ \ k_0 \\ t_1\ \ k_0\ \rightsquigarrow\ s_1\ \ e_1\ \ k_1 \\ t_2\ \ k_1\ \rightsquigarrow\ s_2\ \ e_2\ \ k_2 \\ s_1 \neq \texttt{mts}\ \vee\ s_2 \neq \texttt{mts} \\ v = \mathsf{ifres}_k \\ s = s_0; (tt(ty)\ v); (\texttt{if}\ (e_0)\ (s_1; (v = e_1))\ \texttt{else}\ (s_2; (v = e_2))) \end{array}}{(\textbf{if}\ t_0\ t_1\ t_2)\ \ k\ \rightsquigarrow\ s\ \ v\ \ k_2}$$

$$\frac{t_0 \in T^{cx}_{ty_0} \qquad t_0\ \ k\ \rightsquigarrow\ s_0\ \ e_0\ \ k_0 \qquad t\ \ k_0\ \overset{tc, cx[v \mapsto ty_0]}{\rightsquigarrow}\ s\ \ e\ \ k'}{(\textbf{let}\ v \leftarrow t_0\ \textbf{in}\ t)\ \ k\ \rightsquigarrow\ (s_0; (tt(ty_0)\ v = e_0); s)\ \ e\ \ k'}$$

The relation is not defined on **case**: as defined shortly, the branch subterms (which do not themselves contain pattern matching, as a result of the first translation phase) are translated to separate methods.

4.3.8 Bodies

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

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

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

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

The body of a summand equality method associated to a constant constructor just checks if the argument is the value of the static field that realizes the constant constructor

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

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

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

We capture the embellishment of bodies that would otherwise return conditional expressions or conditional result variables by a function $emb : S \rightarrow S$ defined as

$$\frac{s = (s_0; (\texttt{return}\ (e_0\ ?\ e_1 : e_2)))}{emb(s) = (s_0; (\texttt{if}\ (e_0)\ (\texttt{return}\ e_1)\ \texttt{else}\ (\texttt{return}\ e_2)))}$$

$$\frac{\begin{array}{c}v = \mathsf{ifres}_1 \\ s = (s_0; (ty\ v); (\texttt{if}\ (e_0)\ (s_1; (v = e_1))\ \texttt{else}\ (s_2; (v = e_2))); (\texttt{return}\ v))\end{array}}{emb(s) = decCV(s_0; (\texttt{if}\ (e_0)\ (s_1; (\texttt{return}\ e_1))\ \texttt{else}\ (s_2; (\texttt{return}\ e_2))))}$$

$$\frac{\begin{pmatrix} s \neq (s_0; (\texttt{return}\ (e_0\ ?\ e_1 : e_2))) \\ v = \mathsf{ifres}_1\ \Rightarrow \\ s \neq (s_0; (ty\ v); (\texttt{if}\ (e_0)\ (s_1; (v = e_1))\ \texttt{else}\ (s_2; (v = e_2))); (\texttt{return}\ v)) \end{pmatrix}}{emb(s) = s}$$

where the function

$$decCV : S \xrightarrow{\text{p}} S$$

operates on statements that do not contain the local variable ifres_1 by replacing every occurrence of the conditional result variable ifres_k with ifres_{k-1} (i.e. it decrements the numeric indices); we omit its definition since it is straightforward. Decrementing the indices is unnecessary, but it makes the code look more natural by starting conditional result variable indices from 1 instead of 2 (if there are at least two conditional result variables).

The body of a method derived from a non-constant op with all built-in argument types is derived from the translation of the op's defining term

$$\frac{\begin{array}{c} mth = c.op : tt(\overline{ty}) \to tt(ty) \in Mth_{\text{B}} \uplus Mth_{\text{BA}} \\ \tau(op) = \overline{ty} \to ty \\ \delta(op)\ \ 1\ \overset{\vec{\emptyset},\{\pi(op)\mapsto\overline{ty}\}}{\rightsquigarrow}\ \ s\ \ e\ \ k \end{array}}{body(mth) = emb(s; (\texttt{return}\ e))}$$

Since these methods are static, we use the empty translation context.

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**, is derived from the translation of the op's defining term

$$\frac{\begin{array}{c} mth = c.op : tt(del(\overline{ty}, h)) \to tt(ty) \in Mth_{\text{NPM}} \\ \tau(op) = \overline{ty} \to ty \\ h = \min\{h \mid ty_h \in Ty_{\text{U}}\} \\ \delta(op)\ \ 1\ \overset{\{v_h\mapsto\texttt{this}\},\{\pi(op)\mapsto\overline{ty}\}}{\rightsquigarrow}\ \ s\ \ e\ \ k \end{array}}{body(mth) = emb(s; (\texttt{return}\ e))}$$

Since these are instance methods, the translation context maps the leftmost parameter with user-defined type to `this`.

The body of a method derived from an op with at least a user-defined argument type and whose defining term is a **case**, is derived from the translation of the corresponding branch subterm

$$\frac{\begin{array}{c}\Delta(ty) = \sum_i c_i\ \overline{ty}_i \\ mth = \mathsf{sumd}^{ty}_{c_i}.op : tt(del(\overline{ty}, h)) \to tt(ty) \in Mth_{\mathrm{PMS}} \\ \tau(op) = \overline{ty} \to ty \\ h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\} \\ \delta(op) = \mathbf{case}\ v_h\ \{c_i(\overline{v}_i) \to t_i\}_i \\ tc = \{v_h \mapsto \mathtt{this}\}[\overline{v_i \mapsto \mathtt{this.arg}}] \\ cx = \{\overline{\pi(op) \mapsto ty}\}[\overline{v_i \mapsto ty_i}] \\ t_i\ \ 1 \overset{tc,cx}{\rightsquigarrow}\ \ s\ \ e\ \ k\end{array}}{body(mth) = emb(s; (\mathtt{return}\ e))}$$

The translation context maps the leftmost parameter with user-defined type to `this` 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 v bound in the i-th branch may shadow the mapping of a parameter v.

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

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

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

4.3.9 Static (field) initializers

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

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

The embellishment of static initializers that would assign conditional expressions or conditional result variables to the associated static field, is captured by a function $emb : S \to S$ defined as

$$\frac{s = (s_0; (c.f = (e_0\ ?\ e_1 : e_2)))}{emb(s) = (s_0; (\mathtt{if}\ (e_0)\ (c.f = e_1)\ \mathtt{else}\ (c.f = e_2)))}$$

$$\frac{\begin{array}{c} v = \mathsf{ifres}_1 \\ s = (s_0; (ty\ v); (\texttt{if}\ (e_0)\ (s_1; (v = e_1))\ \texttt{else}\ (s_2; (v = e_2))); (c.f = v)) \end{array}}{emb(s) = decCV(s_0; (\texttt{if}\ (e_0)\ (s_1; (c.f = e_1))\ \texttt{else}\ (s_2; (c.f = e_2))))}$$

$$\frac{\begin{array}{c} s \neq (s_0; (c.f = (e_0\ ?\ e_1 : e_2))) \\ \left(\begin{array}{c} v = \mathsf{ifres}_1 \ \Rightarrow \\ s \neq (s_0; (ty\ v); (\texttt{if}\ (e_0)\ (s_1; (v = e_1))\ \texttt{else}\ (s_2; (v = e_2))); (c.f = v)) \end{array} \right) \end{array}}{emb(s) = s}$$

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

$$\frac{\begin{array}{c} \mathit{fld} = c.op : ty \in \mathit{Fld}_{\mathrm{CB}} \uplus \mathit{Fld}_{\mathrm{CU}} \\ \delta(op)\ \ 1 \overset{\vec{\emptyset},\vec{\emptyset}}{\rightsquigarrow}\ \ s\ \ e\ \ k \\ s \neq \texttt{mts} \end{array}}{emb(s; (c.op = e)) \in sinit(c)}$$

4.4 Concrete name translation

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

$$\begin{aligned} \mathcal{N}' = \mathcal{N} & \\ & \uplus \{\mathsf{aux}_k^{op} \mid op \in \mathcal{N} \ \wedge \ k \in \mathbf{N}_+\} \\ & \uplus \{v_k \mid v \in \mathcal{N} \ \wedge \ k \in \mathbf{N}_+\} \\ & \uplus \{\mathsf{sumd}_{c_i}^{ty} \mid ty, c_i \in \mathcal{N}\} \\ & \uplus \{\mathsf{arg}_j \mid j \in \mathbf{N}_+\} \\ & \uplus \{\mathsf{ifres}_k \mid k \in \mathbf{N}_+\} \\ & \uplus \{\mathsf{prim}, \mathsf{eq}, \mathsf{eqarg}, \mathsf{eqargsub}\} \end{aligned}$$

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

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

capture whether a Unicode character is alphabetic (i.e. letter) and/or alphanumeric (i.e. letter or digit).[7]

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.

4.4.1 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)

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

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.

Two identifiers are reserved for the built-in types

$$\mathsf{Bool}, \mathsf{Int} \in \mathcal{I}$$

whose exact character composition is unimportant because built-in types translate to Java types `boolean` and `int`. User-defined types are identifiers distinct from Bool and Int

$$\mathit{Ty}_{\mathrm{U}} \subseteq \mathcal{I} \qquad \mathit{Ty}_{\mathrm{U}} \cap \mathit{Ty}_{\mathrm{B}} = \emptyset$$

which means that the disjoint union $\mathit{Ty} = \mathit{Ty}_{\mathrm{U}} \uplus \mathit{Ty}_{\mathrm{B}}$ can be replaced by the union $\mathit{Ty} = \mathit{Ty}_{\mathrm{U}} \cup \mathit{Ty}_{\mathrm{B}} \subseteq \mathcal{I}$.

Projectors and constructors are also identifiers

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

A user-defined op consists of an identifier accompanied by the op's argument and result types

$$\mathit{Op}_{\mathrm{U}} \subseteq \{\mathit{oid}^{\overline{\mathit{ty}} \to \mathit{ty}} \mid \mathit{oid} \in \mathcal{I} \ \wedge \ \overline{\mathit{ty}} \in \mathit{Ty}^* \ \wedge \ \mathit{ty} \in \mathit{Ty}\}$$

[7] 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.

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

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

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

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

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

Since the built-in ops translate to Java literals and operators, their exact nature is unimportant, the only requirement being that they are distinct from the other ops

$$Op_{\mathrm{B}} \cap (Op_{\mathrm{U}} \cup Op_{\mathrm{P}} \cup Op_{\mathrm{C}}) = \emptyset$$

Thus, the disjoint union $Op = Op_{\mathrm{U}} \uplus Op_{\mathrm{B}} \uplus Op_{\mathrm{P}} \uplus Op_{\mathrm{C}}$ can be replaced by the union $Op = Op_{\mathrm{U}} \cup Op_{\mathrm{B}} \cup Op_{\mathrm{P}} \cup Op_{\mathrm{C}} \subseteq \mathcal{I} \cup Op_{\mathrm{B}}$.

Variables are also identifiers

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

So, under the above assumptions we have

$$\mathcal{N} = \mathcal{I} \cup \{oid^{\overline{ty}\to ty} \mid oid \in \mathcal{I} \;\wedge\; \overline{ty} \in \mathcal{I}^* \;\wedge\; ty \in \mathcal{I}\}$$

Types and variables are simple identifiers while ops (maybe except built-in ones) are structures $oid^{\overline{ty}\to ty}$ of identifiers.

4.4.2 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[8] by `$Q` (`Q` for "question mark"), and Java keywords and literals are

[8] We use the same substitution notation used for terms.

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

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

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

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

$$\begin{aligned} trp(\mathsf{Bool}) &= \texttt{\$B} \\ trp(\mathsf{Int}) &= \texttt{\$I} \\ ty \in Ty_{\mathrm{U}} \ \Rightarrow \ trp(ty) &= (\texttt{\$U}, it(ty)) \end{aligned}$$

i.e. Bool and Int are represented as `$B` and `$I` (`B` and `I` for "boolean" and "integer"), while user-defined types are represented by prepending `$U` to their translation (`U` for "user-defined"). We lift this function to type sequences

$$trp(\overline{ty}) = (trp(ty_1), \ldots, trp(ty_n))$$

For example, $trp(\mathsf{Int}, \texttt{List}, \mathsf{Bool}) = \texttt{\$I\$UList\$B}$.

4.4.3 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 function $ct : C \to \mathcal{J}$ defined as

$$\begin{aligned} ty \in Ty_{\mathrm{U}} \ \Rightarrow \ ct(ty) &= it(ty) \\ ct(\mathsf{sumd}^{ty}_{c_i}) &= (it(ty), \texttt{\$\$}, it(c_i)) \\ \texttt{Primitive} \notin Ty_{\mathrm{U}} \ \Rightarrow \ ct(\mathsf{prim}) &= \texttt{Primitive} \\ \texttt{Primitive} \in Ty_{\mathrm{U}} \ \Rightarrow \ ct(\mathsf{prim}) &= \texttt{Primitive\$} \end{aligned}$$

The names Ty_{U} of the product and sum classes translate to $it(Ty_{\mathrm{U}})$, which are distinct because of the injectivity of it.

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 product and sum class names.

Finally, we normally translate prim to `Primitive`. While this is certainly distinct from the summand class names (which all contain `$`), a user-defined type may be just `Primitive`. In that case, we append `$` to `Primitive`, making it distinct from the other class names because `Primitive` is not a Java keyword or literal.

4.4.4 Field names

The fields declared in a class must have distinct names.

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

$$\frac{\mathit{fld} = \mathit{ty}.p_i : \mathit{ty}_i \in \mathit{Fld}_{\mathrm{P}}}{\mathit{ft}(\mathit{fld}) = \mathit{it}(p_i)}$$

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

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

$$\frac{\mathit{fld} = \mathsf{prim}.\mathit{oid}^{\mathsf{Bool}} : \mathtt{boolean} \in \mathit{Fld}_{\mathrm{CB}}}{\mathit{ft}(\mathit{fld}) = \begin{cases} \mathit{it}(\mathit{oid}) & \text{if} \quad \mathit{oid}^{\mathsf{Int}} \notin \mathit{Op}_{\mathrm{U}} \\ (\mathit{it}(\mathit{oid}), \mathtt{\$B}) & \text{otherwise} \end{cases}}$$

$$\frac{\mathit{fld} = \mathsf{prim}.\mathit{oid}^{\mathsf{Int}} : \mathtt{int} \in \mathit{Fld}_{\mathrm{CB}}}{\mathit{ft}(\mathit{fld}) = \begin{cases} \mathit{it}(\mathit{oid}) & \text{if} \quad \mathit{oid}^{\mathsf{Bool}} \notin \mathit{Op}_{\mathrm{U}} \\ (\mathit{it}(\mathit{oid}), \mathtt{\$I}) & \text{otherwise} \end{cases}}$$

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

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

A summand class only declares instance fields arg_j, one for each argument of the corresponding constructor. These names translate to `arg1`, `arg2`, etc., which are distinct.

A sum class ty declares a static field c_i for each constant constructor and a static field $\mathit{oid}^{\mathit{ty}}$ 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 $it(oid)$.

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 `$U` to $it(oid)$. Either way, we end up with distinct field names because `$U` cannot appear in any projector translation $it(p_i)$.

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

4.4.5 Method names

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

Normally, a method name $oid^{\overline{ty}\to ty}$ translates to $it(oid)$ and a method name $(\mathsf{aux}_k^{oid^{\overline{ty}'\to ty'}})^{\overline{ty}\to ty}$ translates to $(it(oid), \texttt{\$A}, nt(k))$, as defined below. However, two methods with the same oid (and possibly k) may end up with the same argument types. For example, there could be ops $\mathtt{m}^{ty,\mathsf{Int}\to ty} \neq \mathtt{m}^{\mathsf{Int},ty\to ty} \neq \mathtt{m}^{\mathsf{Int}\to ty}$ with $ty \in Ty_{\mathrm{U}}$, whose corresponding methods are $ty.\mathtt{m}^{ty,\mathsf{Int}\to ty} : \mathtt{int} \to ty$, $ty.\mathtt{m}^{\mathsf{Int},ty\to ty} : \mathtt{int} \to ty$, and $ty.\mathtt{m}^{\mathsf{Int}\to ty} : \mathtt{int} \to ty$. In these conflicting situations, the translated identifiers must be suitably disambiguated.

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

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

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

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

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

$$\frac{mth = c.\mathsf{eq} : ty \rightarrow \mathtt{boolean} \in \mathit{Mth}_{\mathrm{EP}} \uplus \mathit{Mth}_{\mathrm{ES}} \uplus \mathit{Mth}_{\mathrm{ESS}}}{mt(mth) = \mathtt{equals}}$$

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

$$\frac{mth = \mathsf{prim}.oid^{\overline{ty}\rightarrow\mathsf{Bool}} : tt(\overline{ty}) \rightarrow \mathtt{boolean} \in \mathit{Mth}_{\mathrm{B}}}{mt(mth) = \begin{cases} it(oid) & \text{if} \quad oid^{\overline{ty}\rightarrow\mathsf{Int}} \notin \mathit{Op}_{\mathrm{U}} \\ (it(oid), \mathtt{\$B}) & \text{otherwise} \end{cases}}$$

$$\frac{mth = \mathsf{prim}.oid^{\overline{ty}\rightarrow\mathsf{Int}} : tt(\overline{ty}) \rightarrow \mathtt{int} \in \mathit{Mth}_{\mathrm{B}}}{mt(mth) = \begin{cases} it(oid) & \text{if} \quad oid^{\overline{ty}\rightarrow\mathsf{Bool}} \notin \mathit{Op}_{\mathrm{U}} \\ (it(oid), \mathtt{\$I}) & \text{otherwise} \end{cases}}$$

$$\frac{mth = ty.oid^{\overline{ty}\rightarrow ty} : tt(\overline{ty}) \rightarrow ty \in \mathit{Mth}_{\mathrm{BA}}}{mt(mth) = it(oid)}$$

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

$$\frac{\begin{array}{c} mth = c.(\mathsf{aux}_k^{oid^{\overline{ty}'\rightarrow ty'}})^{\overline{ty}\rightarrow ty} : tt(del(\overline{ty}, h)) \rightarrow tt(ty) \in \mathit{Mth}_{\mathrm{PM}} \uplus \mathit{Mth}_{\mathrm{PMS}} \\ h = \min\{h \mid ty_h \in \mathit{Ty}_{\mathrm{U}}\} \end{array}}{mt(mth) = \begin{cases} (it(oid), \mathtt{\$A}, nt(k)) & \text{if} \quad \neg\mathit{confl}(mth) \\ (it(oid), trp(\overline{ty}'), trp(ty'), \mathtt{\$A}, nt(k)) & \text{otherwise} \end{cases}}$$

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

A product class ty also declares a static method $oid^{\overline{ty}\rightarrow ty}$ with argument types $tt(\overline{ty})$ for each op with $\overline{ty} \in \mathit{Ty}_{\mathrm{B}}^{+}$. We translate $oid^{\overline{ty}\rightarrow ty}$ to $it(oid)$. The assumptions on $\mathcal{P}$ guarantee that if two of these static methods have the same

oid then they have distinct (primitive) 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` obviously differs from the primitive argument types of these static methods.

A product class ty_h declares an instance method $oid^{\overline{ty} \to ty}$ 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, $oid^{\overline{ty} \to ty}$ translates to $it(oid)$. In the presence of conflicts, the assumptions on $\mathcal{P}$ imply that $oid^{\overline{ty} \to ty}$ must differ 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 $\mathtt{mm}^{\mathsf{Int}, ty \to \mathsf{Bool}}$ translates to `mm$2$B` if the method conflicts with some other method; otherwise, just to `mm`.

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

A sum class ty also declares a static method $oid^{\overline{ty} \to ty}$ with argument types $tt(\overline{ty})$ for each op with $\overline{ty} \in Ty_{\mathrm{B}}^{+}$. We translate $oid^{\overline{ty} \to ty}$ to $it(oid)$. Similarly to product classes above, these static methods have names or argument types distinct from each other and from the equality method.

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. The assumptions on $\mathcal{P}$ ensure that these methods have names or argument types distinct from the other static methods mentioned just above. We translate c_i to $it(c_i)$; if $c_i = \mathtt{equals}$ and $\overline{ty}_i = ty$, we append `$` to make it distinct from the equality method.

A sum class ty_h also declares a method $oid^{\overline{ty} \to ty}$ 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 sum class ty_h declares a method $(\mathsf{aux}_k^{oid^{\overline{ty}' \to ty'}})^{\overline{ty} \to ty}$ for each auxiliary op introduced during the first translation phase, where $h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\}$. In the absence of conflicts, we translate it by appending `$A` and $nt(k)$ to $it(oid)$ (`A` for "auxiliary"). In the presence of conflicts, it may be insufficient just to append the position h and the result type ty as above, because the assumptions on $\mathcal{P}$ apply to $oid^{\overline{ty}' \to ty'}$, whose types in general have no connection with the argument types $\overline{ty}$ of the method. Since $oid^{\overline{ty}' \to ty'}$ is guaranteed to be unique, we embed the ASCII representation of $\overline{ty}'$ and ty' into the translated identifier. For example, we translate $(\mathsf{aux}_3^{op})^{ty, \mathsf{Int} \to \mathsf{Int}}$, with $op = \mathtt{mmm}^{\mathsf{Int} \to \mathsf{Bool}}$, to `mmm$A3` if there are no conflicts, to `mmm$I$B$A3` otherwise.

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

A summand class $\mathsf{sumd}_{c_i}^{ty_h}$ also declares a method $oid^{\overline{ty} \to ty}$ 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 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 $\mathsf{sumd}_{c_i}^{ty_h}$ declares a method $(\mathsf{aux}_k^{oid^{\overline{ty}'\to ty'}})^{\overline{ty}\to ty}$ with argument types $tt(del(\overline{ty},h))$ for each auxiliary op with $h = \min\{h \mid ty_h \in Ty_{\mathrm{U}}\}$. In the absence of conflicts we translate it by appending `$A` and k to $it(oid)$. In the presence of conflicts, we also embed $\overline{ty}'$ and ty'.

The class prim declares a method $oid^{\overline{ty}\to ty}$ with argument types $tt(\overline{ty})$ for each op with $\overline{ty} \in Ty_{\mathrm{B}}^{+}$ and $ty \in Ty_{\mathrm{B}}$. 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 is no other op $oid^{\overline{ty}\to ty'}$ with $ty' \in Ty_{\mathrm{B}}$ and $ty' \neq ty$. Otherwise, we append `$B` or `$I` to $it(oid)$, incorporating a representation of the result type.

4.4.6 Variables

The variables used within a method or constructor (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

$$\begin{aligned}
vt(\mathsf{eqarg}) &= \mathtt{eqarg} \\
vt(\mathsf{eqargsub}) &= \mathtt{eqargSub} \\
v \in \mathcal{I} \ \Rightarrow\ vt(v) &= it(v) \\
vt(v_k) &= (it(v), \mathtt{\$}, nt(k)) \\
vt(\mathsf{arg}_j) &= (\mathtt{arg}, nt(j)) \\
vt(\mathsf{ifres}_k) &= (\mathtt{\$ifres}, nt(k)) \\
\Delta(ty) = \textstyle\prod_i p_i\ ty_i \ \Rightarrow\ vt(p_i) &= it(p_i)
\end{aligned}$$

The equality methods have parameter eqarg and those in summand classes with at least one field also declare a local variable $\mathsf{eqargsub}$. By translating eqarg and $\mathsf{eqargsub}$ 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-then-else`. These variables are either simple *Fun* identifiers or have one of the forms v_k and ifres_k; variables aux_k^{op} are always translated to `this`. 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 $it(v)$, and ifres_k to `$ifres1`, `$ifres2`, etc.

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 summand classes have parameters arg_j and declare no local variables; we translate their parameters to `arg1`, `arg2`, etc.

4.4.7 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* constructors c_i always translate to $it(c_i)$ and 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 op's defining term.

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 arithmetic exceptions when division by zero is attempted, which would cause some kind of error in *Fun* as well.