Бреслав А. А.

Автоматизация разработки механизмов композиции в предметно-ориентированных языках

Оглавление

Глава 1. Расширение предметно-ориентированных языков механизмами композиции

В этой главе...

§ 1. Объектно-ориентированные модели

Объектно-ориентированные модели (для краткости, ниже мы будем писать просто “модели”) были предложены в начале 1990-х годов [**?**, **?**] и с течением времени были описаны консорциумом OMG (Object Managenemt Group, [**?**]) в виде ряда промышленных стандартов. Наибольшую известность среди “модельно-ориентированных” подходов получил унифицированный язык моделирования — UML (Unified Modelling Language, [**?**]); также очень широко используется библиотека EMF (Eclipse Modelling Framework, [**?**]), основанная на принципах основанная на принципах MOF (Meta-Object Factility, [**?**]), являющейся частью UML (Infractructure). По сути, объектно-ориентированные модели — это не привязанные к конкретному языку программирования структуры, представляющие данные (и способы их обработки) в объектно-ориентированном стиле.

Следуя работам [**?**, **?**], в последующих разделах мы будем использовать модели как универсальный способ описания предметно-ориентированных языков, однако на этом пути нас ждет небольшое затруднение: промышленные стандарты, упомянутые выше, не предоставляют математической базы для работы с моделями. В литературе описаны различные подходы к формализации этой области [**?**, **?**, **?**], в основном нацеленные на описание полной MOF. Использование этих формализаций затруднительно в первую очередь потому, что MOF — это довольно объемная система, а следовательно, и формальные теории, ее описывающие, получаются достаточно громоздкими. С другой стороны, на практике, при реализации предметно-ориентированных языков, как правило, используется не вся MOF, а лишь небольшая ее часть, "ядро", которое называется EMOF (Essential MOF, [**?**]), именно ей соответствует реализация моделей, предоставляемая популярной библиотекой EMF, упомянутой выше. В данном разделе строится формальная теория, описывающая объектно-ориентированные модели, соответствующие EMOF.

1.1. Синтаксис моделей

Чтобы строго определить понятие модели, мы вводим ряд вспомогательных определений, начиная со структурного представления и постепенно вводя ограничения.

Определение 1.1 (Модельный терм)**.** *Модельным термом* называется формула, построенная по следующим правилам[1]:

$$\begin{array}{rcl} \mathrm{MT} & ::= & \mathbf{object}\ \mathrm{MT}_{id} : \mathrm{MT}_{cl}\,\{\mathrm{P}_1, \ldots, \mathrm{P}_n\}\,, n \geq 0 \\ & | & @\mathrm{MT}_{id} \\ & | & [MT_1, \ldots, MT_n]\,, n \geq 0 \\ & | & \{MT_1, \ldots, MT_n\}\,, n \geq 0 \\ & | & \mathtt{null} \mid \Sigma^* \mid \mathbb{Z} \mid \mathbb{B}, \\ \mathrm{P} & ::= & \mathrm{MT}_p = \mathrm{MT}_v \end{array}$$

где $\mathbb{Z}$ — множество целых чисел, Σ — некоторый конечный алфавит символов, а $\mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$.

[1] Для краткости мы пишем $t_1, \ldots, t_n$ вместо использования рекурсивных продукций вида

$$tList ::= t \mid t\ ,\ tList$$

Записывая модельные термы, мы, как правило, не будем заключать строки в кавычки, а будем лишь выделять их шрифтом, чтобы отличать от *метапеременных*: так, abc — это строка из трех символов, а x — метапеременная, которая, в частности может принимать значение abc.

Для различных видов модельных термов мы будем использовать специальные названия. Приведем примеры таких термов и обозначим названия:

- $\{\mathsf{abc}, 239, \mathtt{null}\}$ — *множество*, состоящее из трех констант: строки, числа и специального значения $\mathtt{null}$;
- $[1, 2, 3]$ — *список* из трех чисел;
- $@\mathsf{x}$ — *ссылка* на идентификатор $\mathtt{x}$.
- $\mathbf{object}\ \mathsf{a} : @\mathsf{b}\{\mathsf{c} = \mathtt{false}\}$ — *объект*, где a — *идентификатор*, $@\mathsf{b}$ — *ссылка на класс*, а $\mathsf{c} = \mathtt{false}$ — *свойство* с *маркером* c и *значением* $\mathtt{false}$.

Метапеременные позволяют кратко описать множества термов, обладающих схожей структурой. Если в модельном терме используется метапеременная a, это означает, что на вместо нее может быть подставлен любой подтерм, синтаксически допустимый в данном контексте. Так, например, запись $\{a, b\}$ описывает все термы-множества, состоящие из двух элементов, а $@a$ — ссылки на все возможные идентификаторы.

Модельные термы удобны для представления структур абстрактного синтаксиса различных языков [**?**, **?**]. В качестве примера рассмотрим λ-исчисление [**?**]. Абстрактный синтаксис λ-исчисления задается тремя конструкторами:

Конструктор	Пример конкретного синтаксиса
$LT \;::=\; \mathbf{Abs}(v, LT)$	$\lambda\, x\, .\, t$
$\mid\; \mathbf{App}(LT_1, LT_2)$	$(f\ x)$
$\mid\; \mathbf{Var}(v)$	x

где v — алфавит имен переменных. Например, λ-терм $\lambda\, x\, .\, x\, x$ в абстрактном синтаксисе записывается как $\mathbf{Abs}(x, \mathbf{App}(\mathbf{Var}(x), \mathbf{Var}(x)))$, что соответствует *абстрактному синтаксическому дереву* (Abstract Syntax Tree, AST [**?**]) для данного терма. То же самое AST можно представить также в виде объекта; см. Рис. 1.1.

```
object t1 : @Abstraction {
  var = object x : @Variable {name = x} ,
  body = object t2 : @Application {
    funciton = object t3 : @Usage {var = @x} ,
    argument = object t4 : @Usage {var = @x}
  }
}
```

Рис. 1.1: Представление AST для терма $\lambda\, x\, .\, x\, x$ в виде объектов

Из Рис. 1.1 видно, что объекты модельного терма образуют дерево, соответствующее абстрактному синтаксическому представлению λ-терма, причем роль конструкторов выполняют ссылки на классы. Обратите внимание на использование ссылок в объектах t3 и t4: значения свойства var является *ссылкой* на идентификатор объекта, соответствующего определению переменной x.

Модельный терм можно рассматривать как граф: объекты являются вершинами, а отношение вложенности и ссылки — ребрами. AST является остовным деревом в этом графе — в нем участвуют только ребра вложенности объектов.

1.1.1. Графическая нотация для модельных термов

Наравне с нотацией, введенной выше, мы будем использовать для изображения объектов графическую нотацию, основанную на диаграммах объектов языка UML [**?**]. На Рис. 1.2 в виде такой диаграммы изображены объекты, приведенные на Рис. 1.1.

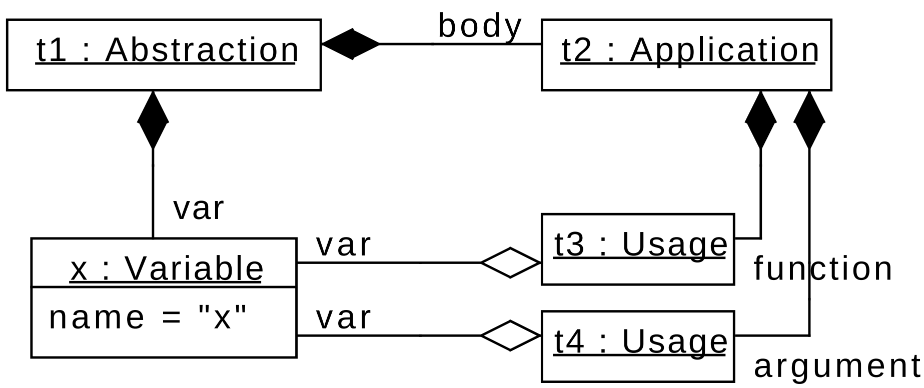

Рис. 1.2: Графическое представление для терма $\lambda x \, . \, x \, x$

Объекты обозначаются прямоугольниками. В верхней части прямоугольника располагается подчеркнутая надпись: это идентификатор объекта и *имя класса*. О классах подробно говорится ниже, а пока лишь заметим, что объекты, относящиеся к одному классу, имеют одинаковую структуру. Значения свойств изображаются либо под горизонтальной чертой в самом прямоугольнике (как для объекта `x`), либо в виде ребер графа. Ближе к концу ребра написано имя свойства, а в начале ребра изображается ромб: незакрашенный, если ребро соответствует ссылке[2], и закрашенный, если значением свойства является объект, то есть имеет место *встраивание*[3] — один объект является частью другого. Таким образом, если рассматривать только ребра с закрашенными ромбами, из графа объектов выделяется дерево, соответствующее текстовой нотации, введенной выше (см. Рис. 1.1).

[2] Что соответствует *агрегации* в терминах UML.

[3] *Композиция* — в терминах UML

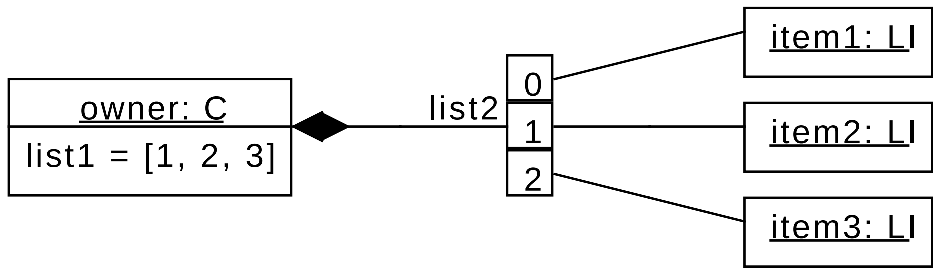

Рис. 1.3: Графическое представление упорядоченных списков

Если значением свойства является множество, оно изображается несколькими ребрами с одинаковой пометкой. В случае списка важен порядок следования элементов, поэтому мы отклоняемся от нотации UML и вводим промежуточный блок с индексами, от ячеек которого отходят ребра к элементам списка. Пример использования данной нотации приведен на Рис. 1.3: список чисел list1 записан непосредственно в теле объекта, а список (встраиваемых) объектов list2 изображен графически.

1.1.2. Конгруэнтность и правильно построенные термы

Определение 1.2. Отношение *конгруэнтности* на модельных термах, $\cong \subset \mathrm{MT} \times \mathrm{MT}$, есть минимальное отношение эквивалентности, обладающее следующими свойствами:

1. Константы конгруэнтны сами себе:
   $x \cong x$, для любого $x \in \{\, \texttt{null}\} \cup \mathbb{Z} \cup \mathbb{B} \cup \Sigma^*$;

2. Ссылки на конгруэнтные индентификаторы конгруэнтны:
   $@x \cong @y$, если $x \cong y$;

3. Списки сравниваются поэлементно:
   $[x_1, \ldots, x_n] \cong [y_1, \ldots, y_n]$, если $x_i \cong y_i$ для любого $i \in [1; n]$;

4. Множества сравниваются поэлементно, без учета порядка: $\{x_1, \ldots, x_n\} \cong \left\{x'_{\pi(1)}, \ldots, x'_{\pi(n)}\right\}$, если существует перестановка π размера n такая, что $x_i \cong x'_{\pi(i)}$ при $i \in [1; n]$.

5. Объекты сравниваются без учета порядка свойств:
$$\mathbf{object}\ id_1 : c_2 \left\{ \begin{array}{c} p_1 = v_1, \\ \ldots, \\ p_n = v_n \end{array} \right\} \cong \mathbf{object}\ id_2 : c_2 \left\{ \begin{array}{c} p'_{\pi(1)} = v'_{\pi(1)}, \\ \ldots, \\ p'_{\pi(n)} = v'_{\pi(n)} \end{array} \right\},$$
если $id_1 \cong id_2$, а также $c_1 \cong c_2$, и существует перестановка π размера n такая, что $p_i \cong p'_{\pi(i)}$ и $v_i \cong v'_{\pi(i)}$.

Приведем несколько примеров:

- $@abc \cong @abc$, поскольку идентификаторы конгруэнтны;
- $[1, 2] \ncong [2, 1]$, при сравнении списков порядок элементов важен;
- а при сравнении множеств — не важен: $\{1, 2\} \cong \{2, 1\}$;
- также не важен порядок свойств при сравнении объектов:
$$\mathbf{object}\ abc : @c \left\{ \begin{array}{ccc} a & = & b \\ c & = & d \end{array} \right\} \cong \mathbf{object}\ abc : @c \left\{ \begin{array}{ccc} c & = & d \\ a & = & b \end{array} \right\},$$
- но значения свойств важны:
$$\mathbf{object}\ abc : @c \left\{ \begin{array}{ccc} a & = & b \\ c & = & d \end{array} \right\} \ncong \mathbf{object}\ abc : @c \left\{ \begin{array}{ccc} c & = & b \\ a & = & d \end{array} \right\}.$$

Введем функцию $\mathcal{S}(t)$, вычисляющую множество всех подтермов терма t:
$$\mathcal{S}(t) \stackrel{\text{def}}{=} \{t\} \cup \begin{cases} \mathcal{S}(id) \cup \mathcal{S}(c) \cup \bigcup\limits_i \mathcal{S}(p_i) \cup \bigcup\limits_i \mathcal{S}(v_i), & \\ \qquad\qquad t = \mathbf{object}\ id : c\,\{p_1 = v_1, \ldots, p_n = v_n\} & \\ \mathcal{S}(r), & t = @r \\ \bigcup\limits_i \mathcal{S}(x_i), & t = [x_1, \ldots, x_n] \text{ или } t = \{x_1, \ldots, x_n\} \\ \emptyset, & \text{в остальных случаях} \end{cases} \tag{1.1}$$

Пример: $\mathcal{S}\left(\mathbf{object}\ \mathsf{a} : @\mathsf{b}\left\{1 = [\mathtt{null}, \mathtt{false}]\right\}\right) = \{$

$$\mathsf{a},\quad @\mathsf{b},\quad \mathsf{b},\quad 1,$$
$$[null, false],\quad \mathtt{null},\quad \mathtt{false}$$
$$\mathbf{object}\ \mathsf{a} : @\mathsf{b}\left\{1 = [null, false]\right\}\}.$$

Как правило, нам будут нужны множества подтермов какого-то определенного вида, например, множество подтермов терма t, являющихся ссылками, мы будем обозначать $\mathcal{S}_{ref}\left(t\right)$. Пример:

$$\mathcal{S}_{ref}\left(\mathbf{object}\ \mathsf{a} : @\mathsf{b}\left\{1 = [\mathtt{null}, \mathtt{false}]\right\}\right) = \{@\mathsf{b}\}.$$

Аналогично $\mathcal{S}_{set}\left(t\right)$ — для подтермов-множеств и $\mathcal{S}_{obj}\left(t\right)$ — для подтермов-объектов.

Определение 1.3. Модельный терм t называется *правильно построенным*, если выполняются следующие условия:

1. Среди подтермов t не существует двух объектов с конгруэнтными идентификаторами;
2. В $\mathcal{S}_{obj}\left(t\right)$ не существует объекта, содержащего два свойства с конгруэнтными именами;
3. В $\mathcal{S}_{set}\left(t\right)$ не существует множества, содержащего два конгруэнтных элемента.

Так, например, терм $\left[\mathbf{object}\ a : @c\left\{\right\}, \mathbf{object}\ a : @c\left\{x = y\right\}\right]$ не является правильно построенным (пункт 1). Терм $\mathbf{object}\ a : @c\left\{a = b, a = c\right\}$ тоже не является правильно построенным, но по другой причине см. пункт 2. Согласно пункту 3, терм $\{1, 2, 1\}$ также не является правильно построенным.

1.1.3. Ссылочные контексты и пред-модели

Определение 1.4 (Ссылочный контекст)**.** Будем называть *ссылочным контекстом* частичную функцию $\Psi : \mathrm{MT} \to \mathrm{MT}$, обладающую следующими свойствами:

1. Если $\Psi(id)$ определено, то $\Psi(id) = \{\mathbf{object}\ id' : \ldots \{\ldots\}\}$, где $id \cong id'$;

2. $\Psi(id) \cong \Psi(id')$ тогда и только тогда, когда $id \cong id'$.

Другими словами, ссылочный контекст “находит” объект по его идентификатору. Для удобства мы будем писать $\Psi(id) = \perp$ в случае, если $\Psi(id)$ не определено.

Обозначим множество всех подтермов, *упомянутых* в контексте Ψ, то есть являющихся подтермами элементов области определения Ψ и/или области значений Ψ, через $\mathcal{S}(\Psi)$. Аналогично вводятся $\mathcal{S}_{ref}(\Psi)$ и $\mathcal{S}_{obj}(\Psi)$.

Определение 1.5. Будем называть ссылочный контекст Ψ *замкнутым*, если

$$o = \mathbf{object}\ id : \ldots \{\ldots\} \in \mathcal{S}_{obj}(\Psi) \Rightarrow \Psi(id) = o$$

и

$$r = @id \in \mathcal{S}_{ref}(\Psi) \Rightarrow \Psi(id) \neq \perp$$

Все объекты, упомянутые в замкнутом контексте, могут быть получены по своим идентификаторам, и каждая ссылка ссылается на существующий объект. Из этого, в частности, следует, что в таком контексте каждому объекту присвоен уникальный идентификатор. Так, например, следующий контекст является замкнутым (здесь функция рассматривается как множество пар вида “аргумент $\mapsto$ результат”):

$$\Psi(id) = \{a \mapsto \mathbf{object}\ a : @a \{\}\}$$

Приведем также пример незамкнутого контекста:

$$\Psi(id) = \{a \mapsto \mathbf{object}\ a : @b \{c = \mathbf{object}\ b : @a \{\}\}\}$$

в последнем примере нарушаются оба требования определения 1.5:

- объект, упомянутый в контексте, не принадлежит области его значений;
- ссылка, упомянутая в контексте, не принадлежит его области определения.

Определение 1.6. Пусть Ψ_1 и Ψ_2 – ссылочные контексты с непересекающимися областями определения ($\forall x \in \mathbf{dom}(\Psi_1), y \in \mathbf{dom}(\Psi_2) : x \not\cong y$). Тогда их *объединение* определяется следующей формулой:

$$(\Psi_1 \cup \Psi_2)\,(id) \stackrel{\text{def}}{=} \begin{cases} \Psi_1(id), & \Psi_1(id) \neq \bot \\ \Psi_2(id), & \Psi_2(id) \neq \bot \\ \bot, & \text{в противном случае.} \end{cases}$$

По модельному терму t можно построить ссылочный контекст, упоминающий все объекты, входящие в t:

$$\Psi_t \stackrel{\text{def}}{=} \{id \mapsto o \mid o = \mathbf{object}\; id : \ldots \{\ldots\} \in \mathcal{S}_{obj}\,(t)\}$$

Определение 1.7 (Пред-модель)**.** Будем называть правильно построенный модельный терм t *пред-моделью в контексте* Ψ_0, если контекст $\Psi_0 \cup \Psi_t$ является замкнутым.

Пред-модель обладает важным свойством: любой объект в ней однозначно определяется своим идентификатором, и любая ссылка указывает на известный объект.

1.2. Классы и типы

На практике для удобства навигации по моделям и контроля над их корректностью вводятся дополнительные ограничения (в виде системы типов), регулирующие структуру моделей. В этом разделе мы вводим понятие *класса* для объектов, которое является основой для таких ограничений.

Определение 1.8 (Класс)**.** *Классом* называется кортеж $\langle abs, id, S, P\rangle$, где

1. abs — признак абстрактного класса, принимает значения `true` или `false`; Если $abs = \mathtt{false}$, класс называется *конкретным*, иначе — *абстрактным*,

2. id — *идентификатор* класса (имя),

3. S — множество классов, называемых *предками* данного класса (*суперклассов*),

4. P — множество пар вида $p : \tau$, где p — *идентификатор* свойства, а τ — *тип* свойства (множество типов определяется ниже).

Множество P не содержит различных пар, в которых совпадают идентификаторы свойств.

Для удобства, мы будем обозначать конкретный класс $\langle \mathtt{false}, id, S, P\rangle$ через $\underline{\textbf{class}}\, id : S\,\{P\}$, а абстрактный класс $\langle \mathtt{true}, id, S, P\rangle$ — через $\textbf{\textit{class}}\, id : S\,\{P\}$. Когда значения признака абстрактного класса не важно или однозначно определяется из контекста, мы будем использовать обозначение $\textbf{class}\, id : S\,\{P\}$.

Определение 1.9. Множество типов $\mathfrak{T}$ описывается следующим образом:

$$
\begin{aligned}
\mathfrak{T} \;::=\;& \mathsf{Char} \mid \mathsf{String} \mid \mathsf{Int} \mid \mathsf{Bool} \\
\mid\;& \mathfrak{T}^{?} \mid \{\mathfrak{T}^{+}\} \mid \{\mathfrak{T}^{*}\} \mid [\mathfrak{T}^{+}] \mid [\mathfrak{T}^{*}] \\
\mid\;& \textbf{val}\,(c) \mid \textbf{ref}\,(c),\ \text{где } c \text{ — идентификатор некоторого класса} \\
\mid\;& \top
\end{aligned}
$$

Поясним интуитивное значение данного определения:

- Char, String, Int, Bool обозначают примитивные типы: символы, строки, целые числа и булевские значения соответственно.

- Тип $\tau^?$ допускает значение `null` или значение типа τ, например, $\mathsf{Char}^?$ соответствует множеству значений $\Sigma \cup \{\mathtt{null}\}$.
- $\{\tau^*\}$ обозначает тип множеств, содержащих ноль или более элементов типа τ. Например, $\{\mathsf{String}^*\}$ — это множества строк, а $\{\{\mathsf{Int}^*\}^*\}$ — множества множеств целых чисел.
- Аналогично, $[\tau^*]$ обозначает списки.
- Если вместо “*” используется “+”, то множество или список должны содержать не менее одного элемента. Например, $\{\tau^+\}$ — непустые множества, составленные из элементов τ.
- $\mathbf{val}\,(c)$, где c — некоторый класс (точнее, его идентификатор), соответствует *встраиваемому* объекту класса c. Напомним, *встраиванием* называется ситуация, при которой один объект входит в другой как подтерм, например: $\mathbf{object}\ a : @c\,\{a = \mathbf{object}\ b : @c\,\{\}\}$.
- $\mathbf{ref}\,(c)$ обозначает тип ссылок на объекты класса c, то есть значений вида $@a$, где a — идентификатор объекта, принадлежащего к классу c (см. ниже).
- $\top$ (“Top”) — супертип всех остальных типов (см. ниже).

1.2.1. Графическая нотация для классов

Наравне с текстовой, мы будем использовать для классов графическую нотацию, основанную на диаграммах классов языка UML (см. Рис. 1.4). Класс обозначается прямоугольником, разделенным на две секции: верхняя секция содержит имя класса (*курсивом* в случае абстрактного класса), а нижняя — его свойства.

Для обозначения отношения “подкласс-суперкласс” или *наследования* на диаграммах используются ребра с треугольником на кон-

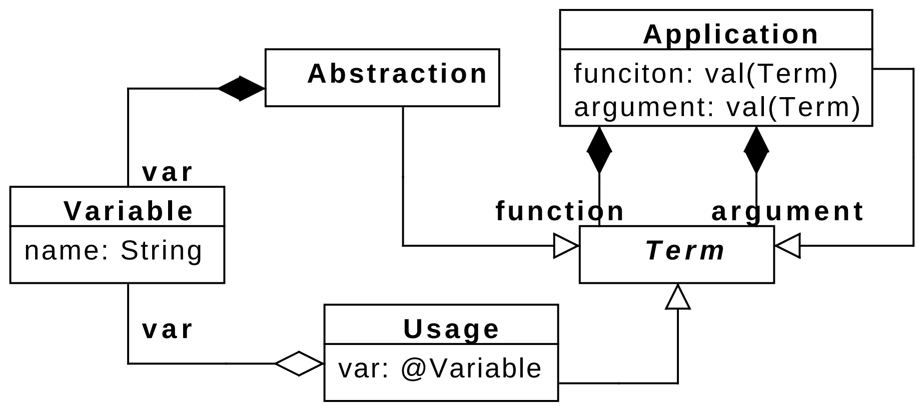

Рис. 1.4: Графическая нотация для классов λ-исчисления

це, обозначающим суперкласс. Заметим, что циклов по наследованию и встраиванию на диаграмме классов быть не должно.

Как и в случае с объектами, некоторые свойства могут обозначаться ребрами (вместо или одновременно с записью в теле класса). Начало ребра обозначается закрашенным или незакрашенным ромбом — для встраивания ($\mathbf{val}\,(\tau)$) и ссылок ($\mathbf{ref}\,(\tau)$) соответственно, а имя свойства пишется ближе к концу ребра, причем для обозначения множеств и списков используется нотация, проиллюстрированная на Рис. 1.5. Для обозначения свойства $\mathsf{bset} : \{\mathbf{val}\,(\mathsf{B})^*\}$ на диаграм-

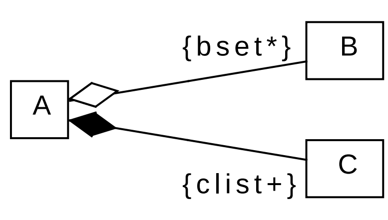

Рис. 1.5: Нотация для множеств и списков на диаграмме классов

ме изображается ребро от класса A, в котором объявлено свойство, к классу B, ближе к концу ребра пишется имя свойства — bset заключенное в фигурные скобки вместе со знаком “$*$”, обозначающие что тип свойства — множество объектов класса B. На Рис. 1.5 начало ребра, изображающего свойство bset отмечено закрашенным ромбом, следовательно элементы множества встраиваются в объекты класса A, что завершает описание типа свойства — $\left\{\mathbf{val}\left(\mathsf{B}\right)^*\right\}$. Аналогично, для обозначения списка $\mathsf{clist} : \left[\mathbf{ref}\left(\mathsf{C}\right)^+\right]$ начало ребра отмечено незакрашенным ромбом, а в конце ребра мы пишем `[clist+]`.

1.2.2. Типы как множества значений

Для типа τ, множество упомянутых в данном типе классов определяется следующим образом:

$$\overline{C}\left(\tau\right) \overset{\text{def}}{=} \begin{cases} \{c\}, & \tau \in \left\{\mathbf{val}\left(c\right), \mathbf{ref}\left(c\right)\right\} \\ \overline{C}\left(\sigma\right), & \tau \in \left\{\left[\sigma^+\right], \left[\sigma^*\right], \left\{\sigma^+\right\}, \left\{\sigma^*\right\}, \sigma^?\right\} \\ \emptyset, & \text{в остальных случаях} \end{cases}$$

Пусть дано множество классов

$$C = \left\{\mathbf{class}\ id_i : S_i \left\{p^i_j : \tau^i_j \mid j \in [1; m_i]\right\} \mid i \in [1; n]\right\},$$

обозначим $\overline{S}\left(C\right) \overset{\text{def}}{=} C \cup \overline{S}\left(\bigcup_1^n S_i\right)$ множество всех классов из C, их предков, предков их предков и т.д.

Аналогично определяется множество всех свойств класса $c = \mathbf{class}\ id : S\left\{p_i : \tau_i\right\}$. Кроме свойств p_i, объявленных в самом c, рассматриваются также свойства, *унаследованные* из суперклассов: $\overline{P}\left(c\right) \overset{\text{def}}{=} \bigcup\limits_{i=1}^{n} \{p^i_j : \tau^i_j \mid j \in [1; m_i]\} \cup \overline{P}\left(\overline{S}\left(c\right)\right)$. Определим также множество всех классов, встраиваемых в c:

$$\overline{C}_{comp}\left(c\right) \overset{\text{def}}{=} \{c' \mid \quad \exists (p : \tau) \in \overline{P}\left(c\right), \text{ где } \tau \in \left\{\mathbf{val}\left(c'\right), \left\{\mathbf{val}\left(c'\right)^+\right\}, \left[\mathbf{val}\left(c'\right)^+\right]\right\}\right\}.$$

Определение 1.10. Конечный набор классов C называется *корректным*, если выполняются следующие свойства:

1. Все предки классов из C содержатся в C: $\overline{S}\left(C\right) = C$.

2. Все типы свойств классов из C упоминают только классы из C: пусть $c \in C$ и $\overline{P}\left(c\right) = \left\{p_i : \tau_i\right\}$, тогда $\bigcup_i \overline{C}\left(\tau_i\right) \subseteq C$.

3. Никакой класс из C не является прямым или непрямым предком самого себя: если $c = \mathbf{class}\ id : S\,\{\ldots\} \in C$, то $c \notin \overline{S}\,(S)$.

4. Никакой класс из C прямо или косвенно не встраивается сам в себя: $c \notin \overline{C}_{comp}\,(c)$.

На типах, порожденных корректным набором классов C, вводится отношение “подтип” ($\preceq$) — это наименьшее транзитивное рефлексивное отношение, обладающее следующими свойствами[4]:

$$\frac{\mathbf{class}\ c : S\,\{\ldots\} \in C \qquad s \in S}{\mathbf{val}\,(c) \preceq \mathbf{val}\,(s) \wedge \mathbf{ref}\,(c) \preceq \mathbf{ref}\,(s)}\ \mathit{subclass}$$

Другими словами, подклассы порождают подтипы. Кроме того, тип $\tau^?$ шире типа τ, поскольку свойства этого типа могут принимать на одно значение — `null` — больше[5]:

$$\frac{}{\tau \preceq \tau^?}\ \mathit{nullable}$$

Аналогично для списков и множеств:

$$\frac{\tau \preceq \sigma \qquad \iota, \kappa \in \{+, *\} \qquad \iota \leq \kappa}{\{\tau^\iota\} \preceq \{\sigma^\kappa\}}\ \mathit{set} \qquad \frac{\tau \preceq \sigma \qquad \iota, \kappa \in \{+, *\} \qquad \iota \leq \kappa}{[\tau^\iota] \preceq [\sigma^\kappa]}\ \mathit{list}$$

Здесь отношение порядка на знаках $+$ и $*$ определяется правилом

$$+ < *.$$

Классы и типы используются для описания множеств допустимых модельных термов. Пусть зафиксирован замкнутый ссылочный контекст Ψ. Ниже мы определим функцию $[\![\bullet]\!]_\Psi^C : \mathfrak{T} \to 2^{\mathbf{MT}}$, которая для корректного набора классов C сопоставляет каждому типу множество его значений. Для этого мы сначала введем вспомогательную функцию $\lceil \bullet \rceil_\Psi : \mathfrak{T} \to 2^{\mathbf{MT}}$, которая возвращает все модельные

[4] Здесь и далее, следуя нотации изображения правил вывода в классической логике, мы записываем *посылки* над горизонтальной чертой, а *заключение* — под чертой; справа курсивом указывается имя или номер, на который мы будем ссылаться в тексте.

[5] Правила, не имеющие посылок над горизонтальной чертой, являются *аксиомами*, то есть выполняются для любых значений метапеременных.

термы, упомянутые в контексте Ψ, принадлежащие к данному типу и ни к какому другому, то есть не согласует отношение “подтип” с отношением вложенности множеств: $\tau \preceq \rho \not\Rightarrow \lceil\tau\rceil_\Psi \subseteq \lceil\rho\rceil_\Psi$. Эта функция задается следующим образом. Типу $\mathbf{val}\,(c)$ ставится в соответствие множество $\lceil\mathbf{val}\,(c)\rceil_\Psi$ всех объектов из $\mathcal{S}_{obj}\,(\Psi)$, в которых ссылкой на класс является терм $@c$, и имеющих все свойства из множества $\overline{P}\,(c)$ и только их, причем значение каждого свойства имеет тип, указанный для данного свойства в классе:

$$\frac{\mathbf{class}\ c : S\,\{\ldots\} \qquad \overline{P}\,(c) = \{p_1 : \tau_1, \ldots, p_n : \tau_n\}}{\lceil\mathbf{val}\,(c)\rceil_\Psi = \left\{\mathbf{object}\ id : @c\,\{p_i = v_i\} \in \mathcal{S}_{obj}\,(\Psi) \mid v_i \in [\![\tau_i]\!]^C_\Psi\right\}}\ objects$$

Другие типы характеризуются следующим образом:

$$\frac{\mathbf{class}\ c : S\,\{\ldots\}}{\lceil\mathbf{ref}\,(c)\rceil_\Psi = \{@id \in \mathcal{S}_{ref}\,(\Psi) \mid \Psi(id) \in \lceil\mathbf{val}\,(c)\rceil_\Psi\}}\ refs$$

$$\frac{}{\lceil\{\tau^+\}\rceil_\Psi = \{\{x_1, \ldots, x_n\} \mid x_i \in \lceil\tau\rceil_\Psi\}}\ set$$

$$\frac{}{\lceil[\tau^+]\rceil_\Psi = \{[x_1, \ldots, x_n] \mid x_i \in \lceil\tau\rceil_\Psi\}}\ list$$

$$\frac{}{\lceil\{\tau^*\}\rceil_\Psi = \lceil\{\tau^+\}\rceil_\Psi \cup \{\{\,\}\}}\ eset \qquad \frac{}{\lceil[\tau^*]\rceil_\Psi = \lceil[\tau^+]\rceil_\Psi \cup \{[\,]\}}\ elist$$

$$\frac{}{\lceil\tau^?\rceil_\Psi = \lceil\tau\rceil_\Psi \cup \{\mathtt{null}\}}\ null$$

$$\frac{}{\lceil\mathsf{Int}\rceil = \mathbb{Z}}\ int \qquad \frac{}{\lceil\mathsf{Bool}\rceil = \mathbb{B}}\ bool$$

$$\frac{}{\lceil\mathsf{Char}\rceil = \Sigma}\ char \qquad \frac{}{\lceil\mathsf{String}\rceil = \Sigma^*}\ string$$

Теперь нам осталось лишь “склеить” значения функции $\lceil \bullet \rceil_\Psi$ согласно отношению $\preceq$, чтобы получить $[\![\bullet]\!]^C_\Psi$:

$$\begin{array}{rcl} [\![\tau]\!]^C_\Psi & = & \lceil \tau \rceil_\Psi \cup \bigcup_{\sigma \preceq \tau} \lceil \sigma \rceil_\Psi \\ [\![\top]\!]^C_\Psi & = & \bigcup\limits_{\tau} \lceil \tau \rceil_\Psi \end{array}$$

Определение 1.11. Будем называть объекты, входящие в множество $[\![\mathbf{val}\,(c)]\!]^C_\Psi$, *экземплярами* класса c.

Мы будем изображать объекты и классы на одной диаграмме, проводя пунктирные ребра от экземпляров к классам[6]. Чтобы не загромождать рисунки, мы будем проводить лишь некоторые такие ребра. Так, например, на Рис. 1.6 к объектам с Рис. 1.2 добавлены некоторые классы с Рис. 1.4. Обратите внимание на то, что один и тот же

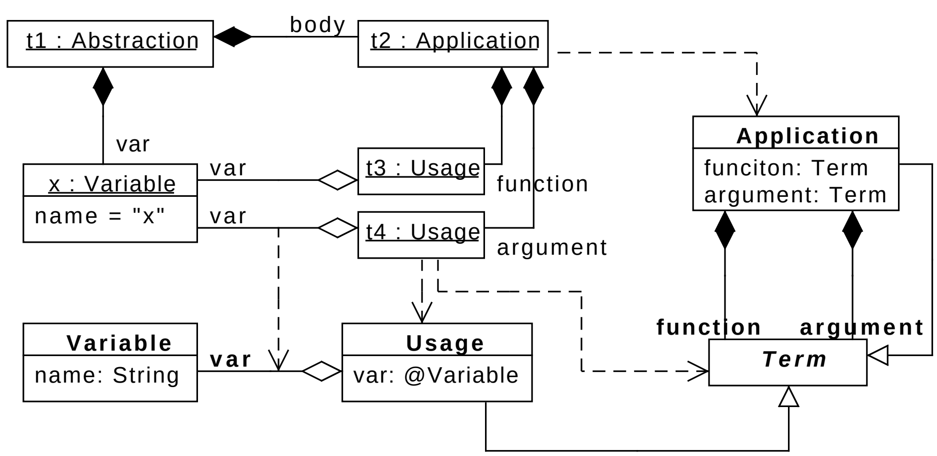

Рис. 1.6: Классы и их экземпляры

объект может являться экземпляром сразу нескольких классов: t4 является экземпляром классов Usage и Term одновременно. Заметим, что все экземпляры одного имеют общий набор свойств, описанный в классе: все экземпляры класса Term разделяют пустой набор свойств,

[6] В нотации UML на таких ребрах пишется стереотип «`instance of`», который мы опускаем, чтобы сэкономить место на рисунках.

а все экземпляры Usage — набор из одного свойства var. Таким образом, классы *регламентируют структуру своих экземпляров*, гарантируя наличие тех или иных свойств у объектов.

1.3. Модели и мета-модели

Выше мы потребовали, чтобы экземпляры класса $C = \mathbf{class}\ c : S\{P\}$, имели вид $\mathbf{object}\ id : @c\{\ldots\}$, то есть содержали ссылку на идентификатор класса c. Если класс C сам не является объектом, то контекст, в котором упоминаются его экземпляры, не может быть замкнутым. Для того, чтобы устранить это неудобство, мы представим классы в виде объектов и сделаем, таким образом, синтаксис модельных термов замкнутым.

На Рис. 1.7 приведена диаграмма, охватывающая основные аспекты представления классов в виде объектов: приведены представления класса Class, экземпляры которого обозначают классы, класса Type — для типов и $\mathsf{Property}$ — для свойств. Пунктирные ребра на этой диаграмме проведены не между классами и объектами, а между объектами, представляющими классы, и объектами, являющимися экземплярами этих классов. Также проведено пунктирное ребро между ребром, изображающим одно из значений свойства $@\mathsf{properties}$ объекта Class, и объектом описывающим это свойство. Обратите внимание на то, что идентификаторы свойств на этой диаграмме имеют вид $@x$, то есть являются ссылками на объекты, описывающие сами свойства. Такое кодирование наиболее удобно, поскольку позволяет, например, узнать тип свойства, просто получив описывающий его объект по идентификатору.

У некоторых объектов на диаграмме Рис. 1.7 не указаны идентификаторы. Это сделано для краткости: эти идентификаторы нигде не используются, соответственно, единственное, что требуется — это чтобы они не совпадали ни с какими другими идентификаторами объ-

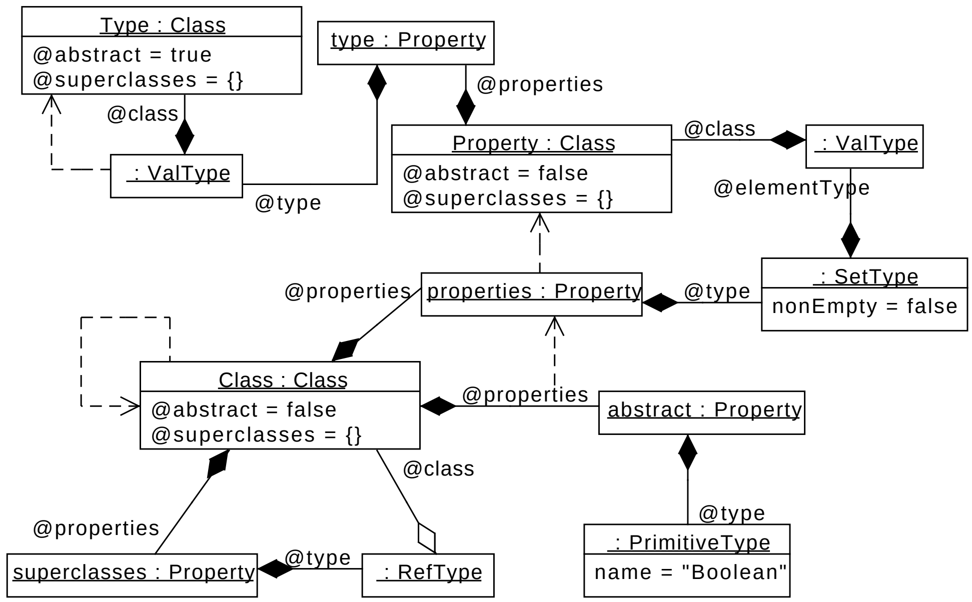

Рис. 1.7: Классы как объекты

ектов. Поскольку такие несовпадающие идентификаторы всегда можно выбрать, мы можем позволить себе не изображать их на рисунке.

Формальное правило для преобразования объектного представления в классы (см. Приложение 1), описанные выше, записывается довольно длинно и довольно тривиально, поэтому мы не приводим его здесь целиком. Основная идея состоит в том, что объекты вида $\mathbf{object}\ c : @\mathsf{Class}\,\{p\}$ преобразуются в классы вида $\mathbf{class}\ c : S\,\{P\}$, где S, P и признак абстрактности вычисляются из значений соответствующих свойств из p.

Представление типов приведено на Рис. 1.8 и Рис. 1.9. Оно также непосредственно трансформируется в типы, определенные выше.

Определение 1.12 (Типовый контекст)**.** Пусть m является предмоделью в некотором ссылочном контексте Ψ_0, и $\Psi = \Psi_0 \cup \Psi_m$. Если по множеству всех экземпляров класса Class, упомянутых в Ψ, стро-

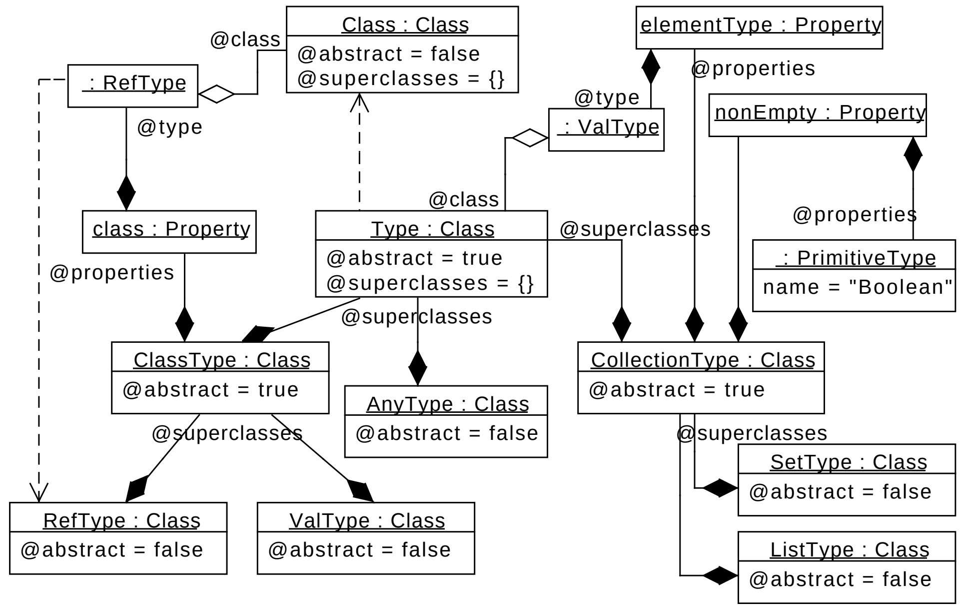

Рис. 1.8: Классы для типов (I)

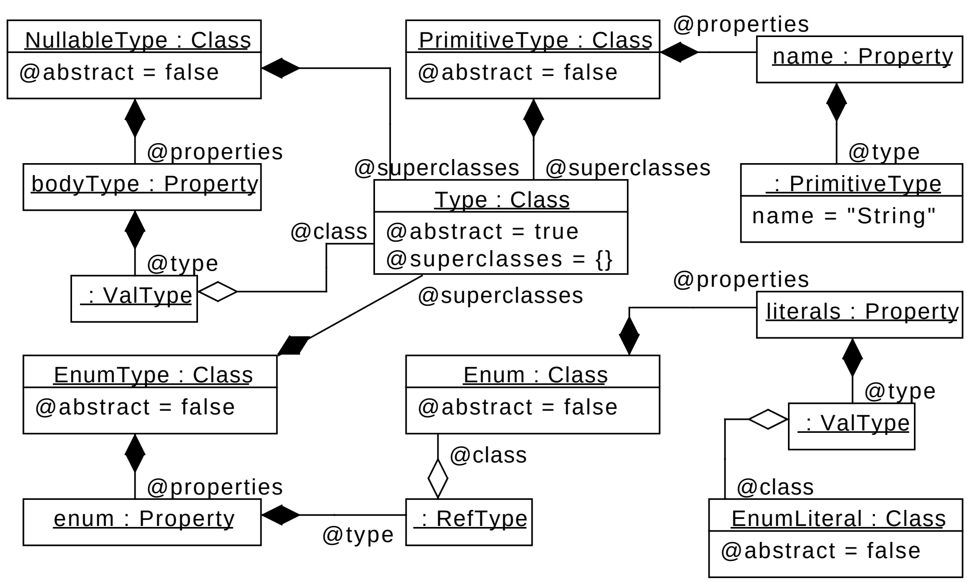

Рис. 1.9: Классы для типов (II)

ится корректный набор классов C, то функция

$$\mathfrak{T}_m(t) = \begin{cases} \tau, & \text{где } t \in [\![\tau]\!]^C_\Psi, \\ \perp, & \text{в противном случае,} \end{cases}$$

называется *типовым контекстом*, порожденным m. Термы t для которых $\mathfrak{T}_m(t) \neq \perp$, называются *типизируемыми* в контексте $\mathfrak{T}_m$.

Основная идея данного определения в том, что “правильные” термы являются типизируемыми. В дальнейшем мы будем работать только с такими термами. Введем понятие *модели*, соответствующее замкнутому типизируемому терму:

Определение 1.13 (Модель)**.** Пред-модель m, всякий подтерм в которой типизируется в контексте $\mathfrak{T}_m$, называется *моделью*.

Предложение 1.1. *Модельный терм, изображенный на Рис. 1.7, Рис. 1.8 и Рис. 1.9 (в совокупности) является моделью в пустом ссылочном контексте.*

Доказательство. Данное утверждение проверяется непосредственно. □

Данная модель имеет фундаментальное значение, поскольку с ее помощью можно описывать классы, позволяющие типизировать термы в других моделях, поэтому для нее используется специальный термин и обозначение:

Определение 1.14. Модель, изображенная на Рис. 1.7, Рис. 1.8 и Рис. 1.9 называется *метаметамоделью* и обозначается $\mathfrak{M}$.

Таким образом, предложение 1.1 можно записать как

$$\mathfrak{T}_\mathfrak{M}(\mathfrak{M}) \neq \perp.$$

!!!Enums

Вообще-то это надо доказать

Приложение 1. Преобразование объектного представления метамоделей в классы

Ниже приводится программа на языке HASKELL [?], которая преобразует модельный терм (такой, например, как на Рис. 1.7) набор классов (согласно определению 1.8). Вначале определяются типы данных для модельных термов (согласно определению 1.1):

```
import Data.List

data ModelTerm
  = Object {
      id :: ModelTerm,
      classRef :: ModelTerm,
      properties :: [Property]
    }
  | Ref { base :: ModelTerm }
  | List [ModelTerm]
  | Set [ModelTerm]
  | Null
  | Int Int
  | Bool Bool
  | Char Char
  | String String
  deriving (Eq)

data Property = Property {
    name :: ModelTerm,
    value :: ModelTerm
  }
  deriving (Eq)
```

Далее определяется функция $\mathcal{S}(t)$ (см. формулу 1.1).

```
subterms :: ModelTerm -> [ModelTerm]
subterms o = o : case o of
  Object id cr p -> (subterms id) ++ (subterms cr) ++ (concat (map pSubterms p))
  Ref    b       -> subterms b
  List   l       -> concat (map subterms l)
  Set    l       -> concat (map subterms l)
  _              -> []

pSubterms :: Property -> [ModelTerm]
pSubterms (Property n v) = (subterms n) ++ (subterms v)
```

Тип данных и функция для построения ссылочных контекстов (см. определение 1.4):

```
type RContext = [(ModelTerm, ModelTerm)]

rcontext :: ModelTerm -> RContext
rcontext t = map toPair (filter isObject (subterms t))
```

```
  where
    toPair o@(Object id _ _) = (id, o)

    isObject (Object _ _ _) = True
    isObject _              = False
```

Типы данных для классов и типов (см. определения 1.8 и 1.9):

```
data Class = Class {
    abstract :: Bool,
    class_id :: String,
    superclasses :: [Class],
    propertyDescriptors :: [PropertyDescriptor]
  }

data PropertyDescriptor = PropertyDescriptor String Type

data Type
  = CharT
  | StringT
  | IntT
  | BoolT
  | NullableT Type
  | SetT Type Bool
  | ListT Type Bool
  | ValT Class
  | RefT Class
  | TopT
```

Функция, преобразующая модельный терм в набор классов (см. раздел 1.3):

```
extractClasses :: ModelTerm -> [Class]
extractClasses t@(Set objects) = map (extractClass (rcontext t)) objects

extractClass :: RContext -> ModelTerm -> Class
extractClass context (Object (String id) (Ref (String "Class")) props)
   = Class
       (extractAbstract props)
       id
       (concat (map (superclasses context) props))
       (concat (map (toDescriptors context) props))

extractAbstract :: [Property] -> Bool
extractAbstract l = head (concat (map isAbstract l))

isAbstract :: Property -> [Bool]
isAbstract (Property (Ref (String "Class.abstract")) (Bool v)) = [v]
isAbstract _                                                   = []

superclasses :: RContext -> Property -> [Class]
superclasses context
    (Property
            (Ref (String "Class.superclasses"))
            (Set refs)) = map (toClass context) refs
superclasses _ _ = []
```

```
toClass :: RContext -> ModelTerm -> Class
toClass context (Ref id) =
    case lookup id context of
        Just a -> extractClass context a
toClass _ r = Class False ("E: " ++ (show r)) [] []

toDescriptors :: RContext -> Property -> [PropertyDescriptor]
toDescriptors context
    (Property
        (Ref (String "Class.propertyDescriptors"))
        (Set s)) = map (toDescriptor context) s
toDescriptors _ _ = []

toDescriptor :: RContext -> ModelTerm -> PropertyDescriptor
toDescriptor context
    (Object
        (String id)
        (Ref (String "PropertyDescriptor"))
        [Property
            (Ref (String "PropertyDescriptor.type"))
            propType]) = PropertyDescriptor id (toType context propType)

toType :: RContext -> ModelTerm -> Type
toType context
    (Object
        _
        (Ref (String "PrimitiveType"))
        [(Property
            (Ref (String "PrimitiveType.type"))
            (String "Boolean"))]) = BoolT
toType context
    (Object
        _
        (Ref (String "PrimitiveType"))
        [(Property
            (Ref (String "PrimitiveType.type"))
            (String "String"))]) = StringT
toType context
    (Object
        _
        (Ref (String "PrimitiveType"))
        [(Property
            (Ref (String "PrimitiveType.type"))
            (String "Integer"))]) = IntT
toType context
    (Object
        _
        (Ref (String "PrimitiveType"))
        [(Property
            (Ref (String "PrimitiveType.type"))
            (String "Char"))]) = CharT

toType context
    (Object
        _
        (Ref (String "NullableType"))
```

```
        [(Property
            (Ref (String "NullableType.type"))
            body)]) = NullableT (toType context body)
toType context
    (Object
        _
        (Ref (String "ReferenceType"))
        [(Property
            (Ref (String "ClassType.class"))
            (classRef))]) = RefT (toClass context classRef)
toType context
    (Object
        _
        (Ref (String "ObjectType"))
        [(Property
            (Ref (String "ClassType.class"))
            (classRef))]) = ValT (toClass context classRef)

toType context
    (Object
        _
        (Ref (String "SetType"))
        [
            (Property
                (Ref (String "CollectionType.nonEmpty"))
                (Bool nonEmpty)),
            (Property
                (Ref (String "CollectionType.elementType"))
                elementType)
        ]) = SetT (toType context elementType) nonEmpty
toType context
    (Object
        _
        (Ref (String "ListType"))
        [
            (Property
                (Ref (String "CollectionType.nonEmpty"))
                (Bool nonEmpty)),
            (Property
                (Ref (String "CollectionType.elementType"))
                elementType)
        ]) = ListT (toType context elementType) nonEmpty

toType context (Object _ (Ref (String "AnyType")) []) = TopT
toType context _ = TopT
```