Numer projektu:
WKP_1/1.4.1/1/2006/92/92/647/2007/U
Tytuł projektu:
Nowatorski system zintegrowanej kontroli jakości i detekcji błędów w oprogramowaniu

Definicja pojęcia funkcyjności dla języka Java

Sygnatura raportu:	KOTEK/5/2008/01
Numer zadania:	5
Tytuł zadania:	Stworzenie definicji pojęcia funkcyjności dla języka Java
Autorzy:	Jacek Chrząszcz, Łukasz Kamiński, Aleksy Schubert, Andrzej Tarlecki
Data:	2008-01-31

UNIA DLA PRZEDSIĘBIORCZYCH
PROGRAM KONKURENCYJNOŚĆ

Projekt współfinansowany przez Unię Europejską Europejski Fundusz Rozwoju Regionalnego

Spis treści

1 Wprowadzenie **3**

2 Semantyczna definicja pojęcia funkcyjności **5**
2.1 Nieformalne omówienie języka 5
2.1.1 Klasy 6
2.1.2 Instrukcje i wyrażenia 7
2.2 Składnia abstrakcyjna 7
2.3 Semantyka *Jafun* 9
2.4 Semantyczna definicja funkcyjności 17
2.5 Semantyczna definicja local-state 20

3 Efektywny model pojęcia funkcyjności **22**
3.1 Sprawdzanie poprawności klas 22
3.1.1 Sprawdzanie poprawności relacji podtypiania 23
3.2 Sprawdzanie poprawności pól 23
3.3 Sprawdzanie poprawności konstruktorów 24
3.3.1 Reguły dla określania lokalności 24
3.3.2 Reguły określające, czy wynikiem jest **this** 26
3.3.3 Reguły cok 27
3.4 Sprawdzanie poprawności metod 29
3.4.1 Reguły cok 30
3.5 Przypisywanie klas wyrażeniom 32

4 Studium możliwości realizacji prototypu kompilatora języka *Jafun* **35**
4.1 Założenia początkowe 35
4.2 Standardowe podejście budowy kompilatorów 36
4.3 Narzędzia analizy składni i semantyki języków programowania 40
4.3.1 Podejście standardowe - budowa pełnego kompilatora . 41
4.3.2 Podejście uproszczone - modyfikacja istniejącego kompilatora 42

4.4 Plan realizacji kompilatora języka *Jafun* 44
4.4.1 Realizacja standardowych aspektów języków obiektowych 44
4.4.2 Realizacja weryfikacji specyficznych modyfikatorów języka *Jafun* 45
4.5 Podsumowanie. Rekomendowany sposób realizacji kompilatora 46

Rozdział 1

Wprowadzenie

Programowanie obiektowe jest obecnie przeważającym przemysłowym paradygmatem programowania. Języki programowania obiektowego takie jak Java czy C# są bardzo często używane do tworzenia aplikacji opartych na graficznym interfejsie użytkownika, zapewniają szeroki rynek programistów oraz istnieje dla nich dobre wsparcie ze strony narzędzi wspomagających programowanie oraz bibliotek procedur, które ułatwiają projektowanie, tworzenie i utrzymywanie kodu programów. Z tego powodu przede wszystkim te języki są wybierane jako języki implementacji w przemysłowych zastosowaniach.

Obecna w tego rodzaju językach znacząca swoboda realizacji określonej funkcjonalności powoduje jednak problemy. W szczególności metody mogą w sposób niejawny, a zatem i nie zawsze czytelny, modyfikować stan programu (przez odwoływanie się i modyfikowanie obiektów udostępnionych im pośrednio, nie jako parametry, w ty do obiektu, w którym zostały zdefiniowane, nawet bez wykorzystywania **this**). Co więcej wyrzucenie wyjątku może prowadzić do powstania niespójności w stanie ważnego obiektu, przez który ten wyjątek pośrednio przechodzi. W językach tego typu możliwe jest też tworzenie podklas, które nie pasują do scenariuszy zaprojektowanych dla nadklasy. W związku z tym, aby zapanować nad obecną tutaj złożonością zjawisk, pojawiają się propozycje takie jak wzorce projektowe ([GHJV95]), systemy typów (np. [BE04]) czy metody tworzenia aplikacji na bazie modelu (np. [MM03]).

Na potrzeby niniejszego opracowania wybraliśmy pewien zestaw cech występujących w funkcyjnych językach programowania: niemutowalne algebraiczne typy danych oraz funkcje, które dla ustalonych danych zawsze dają ten sam wynik i nie mają efektów ubocznych. Korzystanie z tych elementów programowania funkcyjnego nie sprawia większych trudności programistom Javy (w odróżnieniu od np. korzystania z funkcji wyższego rzędu), a przy tym korzyści płynące z ich zastosowania są łatwo widoczne. W szczególności

napisana z użyciem tych metod aplikacja GENRAP bardzo rzadko wpada w niespójny stan, mimo iż w danym momencie rozwoju wciąż zawiera wiele usterek. Co więcej użycie tej metody pozwala na prawie pełne wyeliminowanie wyjątków NullPointerException, które są koszmarem dla większości programistów Javy.

Kluczowym elementem zaprezentowanego tutaj podejścia jest wyodrębnienie modelu prawdziwych, matematycznych funkcji, nie mających efektów ubocznych. Przedstawiamy tutaj dokładną semantyczną definicję opisującą tę własność oraz metodę statycznego weryfikowania jej zachodzenia. Inne podejście mogłoby wykorzystywać adnotacje pure obecne w języku specyfikacji Java Modeling Language (JML, [JML06]) oraz używać algorytmów zaproponowanych w pracy [SR05]. Jednak takie rozwiązanie opierałoby się w silny sposób na JML-u, a to wymagałoby istotnego przeszkolenia programistów w kierunku używania tego języka, a przy tym wsparcie ze strony narzędzi dla takiego przedsięwzięcia jest wciąż niezadowalające.

Elementy programowania funkcyjnego były wprowadzane do Javy w wielu pracach naukowych na wiele sposobów (zob. np. [Set03, Dek06]). Wiele z nich jednak koncentruje się na funkcjach wyższego rzędu, które są trudne do zrozumienia dla programistów Javy, nawet pomimo tego, iż wzorzec projektowy Visitor na dobre zadomowił się jako idiom programistyczny w świecie Javy (zob. [Bel04]). Także podejście zaproponowane przez Nauglera ([Nau03]), w którym promowane są pewne wzorce projektowe pozwalające na definiowanie funkcji (i to nawet wyższego rzędu), wymaga od programistów wyuczenia się paradygmatu funkcyjnego.

Z kolei FunctionalJ ([Dao]) realizuje pomysł Nauglera za pomocą predefiniowanego zestawu podstawowych elementów, które pozwalają na reprezentowanie funkcji jako obiektów i manipulowanie na nich. Tego rodzaju podejście nie może jednak być łatwo przyjęte, wymaga bowiem od programistów myślenia w sposób funkcyjny i traktowania funkcji, także funkcji wyższego rzędu, jak zwykłych danych, na których można swobodnie operować. Tego rodzaju podejście jest na ogół obce zwykłym programistom. Wymagałoby to więc uczenia ich nowego sposobu myślenia i budowania nowych aplikacji w sposób niezgodny z dotychczasowymi nawykami i zapewne niespójny z wieloma współpracującymi, już istniejącymi systemami.

Zamiast tego realizujemy tutaj skromniejszy, ale realistyczny, cel wyodrębnienia tych fragmentów języka, do których z powodzeniem można stosować klasyczne metody analizy zapożyczone ze świata matematycznie zdefiniowanych funkcji.

Rozdział 2

Semantyczna definicja pojęcia funkcyjności

W obecnym rozdziale przedstawimy mały język programowania *Jafun* wzorowany na języku Java, dla którego precyzyjnie określimy semantykę i zdefiniujemy pojęcie funkcyjności. Następnie podamy efektywne reguły sprawdzania, że program jest poprawnie zbudowany, które będą gwarantowały, iż metody i klasy oznaczone jako funkcyjne rzeczywiście są funkcyjne.

Rozdział ten podzielony jest na pięć części. W sekcji 2.1 zaprezentujemy najważniejsze cechy naszego języka oraz omówimy najważniejsze różnice z prawdziwą Javą. Z kolei w sekcji 2.2 przedstawiamy składnię abstrakcyjną języka *Jafun*, który odpowiada pewnemu podzbiorowi języka Java i zawiera adnotacje pozwalające na określenie, czy dana metoda jest funkcyjna, oraz pewne dodatkowe pomocnicze adnotacje pozwalające rzeczywiście stwierdzić tę własność. Następna sekcja 2.3 określa semantykę tego języka, czyli innymi słowy opisuje sposób wykonywania programów w tym języku. Kolejne sekcje 2.4 i 2.5 zawierają semantyczne definicje pojęcia funkcyjności oraz pomocniczego pojęcia local-state.

2.1 Nieformalne omówienie języka

Zaproponowany tutaj język modelujący Javę ma pewne specyficzne cechy, które omówimy obecnie w nieformalny sposób, aby dać wyobrażenie, do czego one służą.

2.1.1 Klasy

Klasy naszego języka mają zasadniczo rzecz biorąc tę samą strukturę, co klasy w Javie, a zatem domyślnie każdy obiekt dziedziczy po obiekcie klasy `Object` i możliwe jest bezpośrednie dziedziczenie tylko z jednej nadklasy. Nie przewidujemy w naszym języku istnienia interfejsów. Widoczność metod i konstruktorów odpowiada jawowej widoczności typu **public**, zaś widoczność pól – widoczności typu **protected**.

W naszym języku inna jest interpretacja sytuacji, gdy klasa nie ma zadeklarowanego żadnego konstruktora. W Javie oznacza to, iż jest zadeklarowany publiczny konstruktor bezargumentowy o pustym ciele. My rezygnujemy z tego zachowania implicite i przyjmujemy, iż brak konstruktorów oznacza, iż w istocie nie można utworzyć obiektów danej klasy (można za to ewentualnie tworzyć obiekty podklas).

Dodatkowo klasy mogą być opatrzone nieobecnymi w Javie kwalifikatorami **imm** oraz **func**. W naszym systemie będziemy uznawali, że obiekty klas oznaczonych jako **imm** mają tę własność, że w programie zawsze są widziane tak samo (jako mające ten sam stan). Z kolei obiekty klas oznaczonych jako **func** będą miały tę samą własność, co obiekty klas **imm**, ale dodatkowo metody tych obiektów będą się zachowywały jak matematyczne funkcje, tzn. nie będą miały efektów ubocznych oraz dla ustalonego zestawu argumentów zawsze będą dawały ten sam wynik.

Adnotacje **func** przypisywane są także metodom. Można je stosować, gdy nie chcemy, aby wszystkie metody danej klasy oznaczonej jako **imm** zachowywały się jak funkcje – dopuszczamy wtedy, że metody te mogą zmieniać część stanu poza granicami reprezentacji obiektu, w którym się znajdują. Dodatkowo metody (a także konstruktory) mają własną adnotację **lstate**. Adnotacja ta oznacza tyle, że metoda (lub konstruktor) modyfikuje wyłącznie stan wewnętrzny obiektu, w którym występuje, lub obiektów, które zostały podane jako jej parametry.

Niezależnie od przedstawionych wyżej ograniczeń, na modyfikację obiektów, które istniały na stercie przed wywołaniem, metody mogą dowolnie tworzyć, niszczyć i modyfikować nowe, lokalnie stworzone obiekty. Obiekty te mogą istnieć na stercie także po wyjściu z metody – na przykład gdy są dawane jako wynik.

W języku występuje także dodatkowy rodzaj adnotacji **rep** związany z polami. Adnotacje te są potrzebne do precyzyjnego określania, jakie pola obiektu mają być traktowane jako jego reprezentacja w sposób płytki (tylko sama referencja), a jakie w sposób głęboki (referencja wraz z reprezentacją obiektu przez nią wskazywanego). Ten ostatni sposób traktowania jest właśnie zaznaczany obecnością adnotacji.

2.1.2 Instrukcje i wyrażenia

W naszym języku, podobnie jak w pracy [HPSS07], uprościliśmy też postać wnętrza metod i konstruktorów. Zamiast zwykłego składania instrukcji używamy tutaj wyrażenia **let**, które jednocześnie pozwala definiować zmienne lokalne. Taka postać zmiennych lokalnych wymusza przy okazji to, iż programy są w wygodnej dla weryfikacji postaci *single-assignmet* (na każdą zmienną lokalną jest co najwyżej jedno przypisanie – przypisanie, które definiuje jej wartość).

Kolejne uproszczenia dotyczą sposobu, w jaki używane są wyrażenia. W oryginalnej Javie wywołania metod i konstruktorów, rzucanie wyjątków, czy przypisania mogły się odbywać z użyciem dowolnych wyrażeń. Nasz język jest ograniczony w ten sposób, że dopuszczamy w pozycji wyrażenia tylko zmienne. Przyjęliśmy takie rozwiązanie, gdyż upraszcza ono projekt całego systemu typów dla tego języka. Ograniczenie to nie ma wpływu na wyrażalność języka, gdyż odpowiednie wyrażenia można wprowadzić za pomocą zmiennych zadeklarowanych w odpowiednio dobranej konstrukcji **let**.

Kolejne uproszczenie polega na sposobie odwoływania się do pól obiektów. W Javie mamy dostępną nieograniczoną możliwość sięgania w głąb obiektu za pomocą wyrażeń z kropkami. Nasz język dopuszcza tylko sięgnięcie o jeden poziom w głąb. Znowu, to ograniczenie upraszcza reguły typowania, a nie ogranicza wyrażalności, gdyż można dowolny ciąg odwołań zrealizować z użyciem **let**.

Konstruktory w Javie mogą się odwoływać do konstruktorów nadklas przy pomocy słowa kluczowego **super**. Ta konstrukcja u nas nie występuje. Natomiast nie jest ona konieczna, gdyż w razie potrzeby wystarczy przepisać zawartość odpowiedniego konstruktora z nadklasy.

Warto zwrócić uwagę, iż język przez nas tutaj proponowany zawiera wyjątki w odróżnieniu od języka przedstawionego w pracy [HPSS07].

Z ważniejszych cech języka Java, które nie są objęte opisywanym tutaj modelem można wymienić: wielowątkowość, refleksję oraz interfejsy.

2.2 Składnia abstrakcyjna

Gramatykę naszego języka przedstawimy w za pomocą zmodyfikowanej gramatyki BNF. Będziemy tutaj używali dodatkowych konwencji, które będą ułatwiały zapis gramatyki. Produkcja z $+$ w górnym indeksie oznacza, że dany symbol może być rozwinięty dowolną niezerową liczbę razy. Obecność $*$ w górnym indeksie oznacza możliwość rozwinięcia danego symbolu dowolną, w tym zerową, liczbę razy. Wyrażenia umieszczone w nawiasach kwadrato-

$$
\begin{array}{lcl}
\overline{C} & ::= & \mathsf{cdecl}^{+} \\
\mathsf{cdecl} & ::= & \textbf{class}\ [\mathsf{cmod}]\ C_1\ [\textbf{ext}\ C_2]\ \{\overline{F}\ \overline{K}\ \overline{M}\} \\
\mathsf{cmod} & ::= & \textbf{imm} \mid \textbf{func} \\
\mathsf{mmod} & ::= & \textbf{lstate} \mid \textbf{func} \\
\mathsf{kmod} & ::= & \textbf{lstate} \\
\mathsf{fmod} & ::= & \textbf{rep} \\
C & ::= & \ldots \textit{class name} \ldots \\
\overline{F} & ::= & [F;]^{*} \\
F & ::= & C\ [\mathsf{fmod}]\ x \\
\overline{K} & ::= & [K;]^{*} \\
K & ::= & [\mathsf{kmod}]\ k(\mathsf{args})\ [\textbf{throws}\ \overline{\mathsf{Exc}}]\ \{E\} \\
\mathsf{args} & ::= & \varepsilon \mid C\ x \mid C\ x, \mathsf{args} \\
\overline{\mathsf{Exc}} & ::= & C \mid C, \overline{\mathsf{Exc}} \\
\overline{M} & ::= & [M;]^{*} \\
M & ::= & C\ [\mathsf{mmod}]\ m(\mathsf{args})\ [\textbf{throws}\ \overline{\mathsf{Exc}}]\ \{E\} \\
E & ::= & \textbf{new}\ C.k(\overline{x}) \mid \textbf{let}\ C\ x = E_1\ \textbf{in}\ E_2 \mid \textbf{if}\ E_1\ \textbf{then}\ E_2\ \textbf{else}\ E_3 \mid \\
 & & x.m(\overline{x}) \mid \mathsf{fieldref} = E \mid \mathsf{varref} \mid \textbf{throw}\ x \mid \\
 & & \textbf{try}\ \{E_1\}\ \textbf{catch}\ (C\ x)\ \{E_2\} \mid \textbf{null} \\
\mathsf{fieldref} & ::= & x_1.x_2 \mid \textbf{this}.x \\
\mathsf{varref} & ::= & x \mid \textbf{this} \mid \mathsf{fieldref} \\
\overline{x} & ::= & \varepsilon \mid x \mid x, \overline{x} \\
x & ::= & \ldots \textit{variable name} \ldots \\
k & ::= & \ldots \textit{constructor name} \ldots \\
m & ::= & \ldots \textit{method name} \ldots
\end{array}
$$

Rysunek 2.1: Składnia abstrakcyjna języka *Jafun*

wych ([...]) są opcjonalne – mogą zostać rozwinięte raz lub nie być rozwinięte w ogóle.

Programem będziemy określali niepusty ciąg deklaracji klas, zawierający deklaracje klas:

- `Object`,
- `NullPointerException` ;

w notacji z rys. 2.1 program oznaczamy jako $\overline{C}$. Zbiór wszystkich programów oznaczamy jako Prog.

2.3 Semantyka *Jafun*

Obecnie przedstawimy semantykę języka *Jafun*. Zastosujemy tutaj semantykę operacyjną małych kroków. Jednak wykonanie kroku zaprezentowanej relacji redukcji będzie wymagało niekiedy spełnienia dodatkowych warunków, stąd reguły redukcji prezentujemy w postaci:

$$\frac{\text{warunki konieczne, aby można ją wykonać}}{\text{reguła redukcji do wykonania}}$$

W celu zdefiniowania reguł semantyki musimy najpierw określić pojęcia, z jakich ten opis będzie korzystał.

Zbiór wszystkich nazw klas będziemy oznaczali Cname. Rzutowanie na i-tą współrzędną produktu oznaczamy przez π_i. W opisie semantyki będziemy używali abstrakcyjnego zbioru lokacji Loc. Narzucamy na niego ograniczenie takie, że powinien on zawierać element **null** oraz mieć moc co najmniej $\aleph_0$.

Definicja 1 `(obiekty, sterty)`
Przez obiekt będziemy rozumieli element produktu $(\mathsf{Ident} \rightharpoonup \mathsf{Loc}) \times \mathsf{Cname}$, gdzie Loc to abstrakcyjny zbiór lokacji, zaś Ident to zbiór możliwych identyfikatorów pól klasy. Pierwszy element pary to w intencji przyporządkowanie polom obiektów, na które wskazują, zaś drugi element pary to nazwa klasy danego obiektu. Zbiór wszystkich obiektów oznaczać będziemy przez Obj, zaś jego elementy literami łacińskimi takimi jak o, p, q itp.

Wprowadzamy skrót notacyjny $o.f$ na $\pi_1(o)(f)$ oraz $\mathsf{typeof}(o)$ na $\pi_2(o)$.

Stertą będzie dowolna funkcja częściowa $h : \mathsf{Loc} \rightharpoonup \mathsf{Obj}$, która jest nieokreślona dla argumentu oraz ma skończoną dziedzinę. Zbiór wszystkich stert oznaczamy Heap. Typowo sterty oznaczamy literami łacińskimi takimi jak h, g itp.

Definicja 2 `(pierwotna inicjalizacja)`
$\mathsf{empty}_{\overline{C},C}(f) =$ **null** wtedy i tylko wtedy, gdy
$f \in \mathsf{fields}(\overline{C}, C)$ lub gdzie
fields określone jest jak w Definicji 6.

Definicja 3 `(alokacja)`
Sterty i lokacje są związane za pomocą abstrakcyjnej funkcji $\mathsf{alloc} : \mathsf{Heap} \times \mathsf{Cname} \times \mathsf{Prog} \to \mathsf{Loc} \times \mathsf{Heap}$. W intencji funkcja ta dla danej sterty, nazwy klasy i programu daje w wyniku nową stertę (druga współrzędna wyniku) z zaalokowaną świeżą lokacją (pierwsza współrzędna) wskazującą na obiekt $(\mathsf{empty}_{\overline{C},C}, C)$, czyli zainicjalizowany wartościami **null** obiekt zadanej klasy w zadanym programie.

Opis semantyki będzie wymagał dodatkowego pojęcia kontekstu:

Definicja 4 `(konteksty)`
Wyrażenia programowe, które będziemy rozważać są generowane przez regułę dla symbolu nieterminalnego E z rys. 2.1 z dodatkową produkcją po prawej stronie dopuszczającą jako wyrażenia elementy zbioru Loc.

Dodatkowo będziemy rozważać *konteksty* zbudowane zgodnie z gramatyką:

$$\begin{array}{lcl} A & ::= & C \mid \emptyset \\ \mathcal{C} & ::= & \textbf{new}\ C.k(\overline{x}_1, \mathcal{C}, \overline{x}_2) \mid \textbf{new}\ C.k(\mathcal{C}) \mid \textbf{let}\ C\ x = \mathcal{C}\ \textbf{in}\ E \mid \\ & & \textbf{if}\ \mathcal{C}\ \textbf{then}\ E_1\ \textbf{else}\ E_2 \mid x.m(\overline{x}_1, \mathcal{C}, \overline{x}_2) \mid x.m(\mathcal{C}) \mid \\ & & \textbf{try}\ \{\mathcal{C}\}\ \textbf{catch}\ (C\ x)\ \{E\} \mid [\![\ ]\!]^l_A \end{array}$$

gdzie $l \in \mathsf{Loc}$.

Definicja 5 `(wyrażenia kontekstowe)`
Wyrażenia kontekstowe to wyrażenia postaci:

$$\mathcal{C}\{[\![E]\!]^l_A / [\![\ ]\!]^l_A\} \text{ lub } \mathcal{C}\{[\![l']\!]^l_A / [\![\ ]\!]^l_A\},$$

gdzie E to wyrażenie zbudowane zgodnie z produkcją E na rys. 2.1, zaś $l, l' \in \mathsf{Loc}$.

W intencji dziura $[\![\ ]\!]^l_A$ wskazuje miejsce, w którym obecnie znajduje się ewaluacja wyrażenia. Adnotacja A wskazuje, czy ewaluacja ma charakter normalny ($A = \emptyset$), czy też wykonywana jest przy wyrzuconym, ale jeszcze nie obsłużonym wyjątku klasy C (dla $A \neq \emptyset, A = C$). Z kolei adnotacja l oznacza lokację aktualnie używaną jako **this**. Dodatkowo stosujemy też notację $\mathcal{C}[\mathcal{C}'[\![E]\!]^l_A]$ lub $\mathcal{C}[\mathcal{C}'[\![l']\!]^l_A]$ na oznaczenie odpowiednio $\mathcal{C}\{\mathcal{C}'[\![E]\!]^l_A / [\![\ ]\!]^l_A\}$ i $\mathcal{C}\{\mathcal{C}'[\![l']\!]^l_A / [\![\ ]\!]^l_A\}$. Dla uproszczenia notacji, jeśli to nie powoduje niejednoznaczności, opuszczamy indeksy górny l i dolny A.

Funkcje pomocnicze Oprócz powyższych definicji potrzebujemy jeszcze pewnych funkcji oraz pomocniczych notacji, które są używane w definicji semantyki.

Definicja 6 `(funkcje pomocnicze dotyczące klas i programów)`
Załóżmy, że deklaracja klasy C w $\overline{C}$ ma postać:

$$\textbf{class}\ [\mathsf{cmod}]\ C_1\ [\textbf{ext}\ C_2]\ \{\overline{F}\ \overline{K}\ \overline{M}\}.$$

Określamy następujące oznaczenia:

$C_1 \in \overline{C}$,	na stwierdzenie faktu, iż C_1 znajduje się w $\overline{C}$,
$\mathsf{fields}(\overline{C}, C_1) = \overline{F} \cup \mathsf{fields}(\overline{C}, \mathsf{Object})$,	gdy C_2 nie występuje,
$\mathsf{fields}(\overline{C}, C_1) = \overline{F} \cup \mathsf{fields}(\overline{C}, C_2)$,	gdy C_2 występuje,
$\mathsf{konstructors}(\overline{C}, C_1) = \overline{K}$,	
$\mathsf{methods}(\overline{C}, C_1) = \overline{M} \cup \mathsf{methods}(\overline{C}, \mathsf{Object})$,	gdy C_2 nie występuje,
$\mathsf{methods}(\overline{C}, C_1) = \overline{M} \cup \mathsf{methods}(\overline{C}, C_2)$,	gdy C_2 występuje,
$\mathsf{extends}(\overline{C}, C_1) = C_2$,	
$\mathsf{modifier}(\overline{C}, C_1) = \mathsf{cmod}$,	jeśli cmod występuje w deklaracji,
$\mathsf{modifier}(\overline{C}, C_1) = \emptyset$,	jeśli kmod nie występuje w deklaracji,
$\mathsf{isImmutable}(\overline{C}, C)$	jeśli $\mathsf{modifier}(\overline{C}, C) =$ **imm** lub $\mathsf{modifier}(\overline{C}, C) =$ **func**

Funkcja określania klasy obiektu na stercie:
$\mathsf{class}(h, l) = \pi_2(h(l))$.

Definicja 7 `(funkcje pomocnicze dotyczące pól)`
Załóżmy, że deklaracja pola:

$$C_1\ [\mathsf{fmod}]\ x$$

występuje w $\mathsf{fields}(\overline{C}, C)$. Określamy następujące oznaczenia:

$x \in \overline{F}$	na stwierdzenie faktu, iż x znajduje się w $\overline{F}$,
$\mathsf{type}(\overline{C}, C, x) = C_1$,	
$\mathsf{modifier}(\overline{C}, C, x) = \mathsf{fmod}$,	jeśli fmod występuje w deklaracji,
$\mathsf{modifier}(\overline{C}, C, x) = \emptyset$,	jeśli fmod nie występuje w deklaracji,

Definicja 8 `(funkcje pomocnicze dotyczące konstruktorów)`
Załóżmy, że deklaracja:

$$[\mathsf{kmod}]\ k(\mathsf{args})\ [\mathbf{throws}\ \overline{\mathsf{Exc}}]\ \{E\}$$

konstruktora K występuje w $\mathsf{konstructors}(\overline{C}, C)$. Określamy następujące oznaczenia:

$K \in \overline{C}$
$\mathsf{body}(\overline{C}, C, K) = E$,
$\mathsf{params}(\overline{C}, C, K) = (\mathsf{args})$,

$\mathsf{paramsNo}(\overline{C}, C, K) = n$,	jeśli $(\mathsf{args}) = (C_1 x_1, \ldots, C_n x_n)$
$\mathsf{paramType}(\overline{C}, C, K, i) = (\mathsf{args})$,	
$\mathsf{modifier}(\overline{C}, C, K) = \mathsf{kmod}$,	jeśli kmod występuje w deklaracji,
$\mathsf{modifier}(\overline{C}, C, K) = \emptyset$,	jeśli kmod nie występuje w deklaracji,
$\mathsf{name}(\overline{C}, C, K) = k$	
$\mathsf{isThrown}(\overline{C}, C, K, D)$,	o ile $D \in \overline{\mathsf{Exc}}$,
$\mathsf{throws}(\overline{C}, C, K) = \overline{\mathsf{Exc}}$.	

W związku z tym, że istnieje wzajemnie jednoznaczna odpowiedniość między nazwami konstruktorów a konstruktorami, to we wszystkich powyższych funkcjach i relacjach będziemy używali wymiennie samego konstruktora i jego nazwy. Warto zwrócić uwagę na to, że liczba parametrów n może być równa 0.

Definicja 9 `(funkcje pomocnicze dotyczące metod)`
Załóżmy, że deklaracja:

$$C_1\ [\mathsf{mmod}]\ m(\mathsf{args})\ [\mathbf{throws}\ \overline{\mathsf{Exc}}]\ \{E\}$$

metody M występuje w $\mathsf{methods}(\overline{C}, C)$. Określamy następujące oznaczenia:

$\mathsf{body}(\overline{C}, C, M) = E$,	
$\mathsf{params}(\overline{C}, C, M) = (\mathsf{args})$,	
$\mathsf{paramsNo}^C, C, M) = n$,	jeśli $(\mathsf{args}) = (C_1 x_1, \ldots, C_n x_n)$
$\mathsf{paramType}(\overline{C}, C, M, i) = (\mathsf{args})$,	
$\mathsf{returnType}(\overline{C}, C, M) = C_1$,	
$\mathsf{modifier}(\overline{C}, C, M) = \mathsf{mmod}$,	jeśli mmod występuje w deklaracji,
$\mathsf{modifier}(\overline{C}, C, M) = \emptyset$,	jeśli mmod nie występuje w deklaracji,
$\mathsf{name}(\overline{C}, C, M) = M$	
$\mathsf{isThrown}(\overline{C}, C, M, D)$,	o ile $D \in \overline{\mathsf{Exc}}$.
$\mathsf{isLocalState}(\overline{C}, C, M)$,	o ile $\mathsf{mmod} = \mathbf{lstate}$

W naszym systemie będziemy przyjmowali, inaczej niż w Javie, że nazwa metody jednoznacznie ją identyfikuje. Nie jest to zbyt dalekie od rzeczywistej Javy, gdyż tam jednoznaczna identyfikacja następuje na podstawie nazwy połączonej z sygnaturą. Aby używać naszego systemu, wystarczy więc jawnie włączyć sygnatury do nazw metod. W związku z tą uwagą, jeśli to nie powoduje niejednoznaczności, pozwalamy, aby w użyciach wyżej zdefiniowa-

nych operacji i relacji w miejscach metody stała jej nazwa. Warto zwrócić uwagę na to, że liczba parametrów n może być równa 0.

W semantyce języka *Jafun* istotną rolę gra relacja podtypu, która wskazuje, które obiekty można używać w określonym miejscu, ponieważ realizują co najmniej specyficzny dla danego miejsca zestaw komunikatów.

Relacja podtypów

$$\frac{}{\overline{C} \vdash C \leq: C} \quad \textit{(subrefl)}$$

$$\frac{}{\overline{C} \vdash C \leq: \mathsf{Object}} \quad \textit{(subobject)}$$

$$\frac{\mathsf{extends}(\overline{C}, C) = D}{\overline{C} \vdash C \leq: D} \quad \textit{(subext)}$$

$$\frac{\overline{C} \vdash C \leq: E \qquad \overline{C} \vdash E \leq: D}{\overline{C} \vdash C \leq: D} \quad \textit{(subtrans)}$$

Warto podkreślić, że w ogólności ta relacja nie musi być częściowym porządkiem, gdyż nazwa D użyta w regule (subext) nie musi być zadeklarowana w $\overline{C}$.

Postać relacji przejścia Relacja przejścia naszej semantyki będzie miała postać:

$$\overline{C}, h, \mathcal{C}_1[\![E_1]\!]^{l_1}_{A_1} :: \cdots :: \mathcal{C}_n[\![E_n]\!]^{l_n}_{A_n} \to h', \mathcal{C}'_1[\![E_1]\!]^{l'_1}_{A'_1} :: \cdots :: \mathcal{C}'_m[\![E_m]\!]^{l'_1}_{A'_1}$$

Aktualny stan wykonania programu reprezentowany jest tutaj przez parę h, $\mathcal{C}_1[\![E_1]\!]^{l_1}_{A_1} :: \cdots :: \mathcal{C}_n[\![E_n]\!]^{l_n}_{A_n}$. Element $h \in \mathsf{Heap}$ reprezentuje zawartość sterty w momencie wykonania, zaś ciąg : $C_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!]$ określa stos wywołań metod. Zauważmy, że po wykonaniu kroku stos wywołań może się zmienić, jednak zmiana ta będzie co najwyżej zwiększeniem lub zmniejszeniem stosu ramek o jeden.

Ewaluacja programu startuje w konfiguracji $\overline{C}, h, [\![E]\!]^{\mathbf{null}}_{\emptyset}$, gdzie $\overline{C} \in \mathsf{Prog}$, h dowolną stertą, która na ustalonej lokacji `npe` ma pewien obiekt klasy `NullPointerException`, zaś E wyrażeniem programu, dla którego wyliczamy semantykę (w wypadku zwykłych programów w Javie byłoby to ciało metody `main` jakiejś ustalonej klasy ze zbioru $\overline{C}$).

Zwykle semantykę określa się za pomocą domknięcia zwrotno-przechodniego relacji jednokrokowej. W tym wypadku domknięcie to ma specyficzną postać dlatego zdefiniujemy teraz relację $\rightarrow^*$, a także pomocniczą relację $\rightarrow^{(*)}$, która będzie wykorzystywana w późniejszych definicjach semantycznych.

Definicja 10 (`relacje` $\rightarrow^*$ `oraz` $\rightarrow^{(*)}$)
Relację $\rightarrow^*$ definiujemy jako najmniejszą relację spełniającą warunki:

- $\overline{C}, h, \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!] \rightarrow^* h, \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!]$;
- jeśli $\overline{C}, h, \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!] \rightarrow h', \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_m[\![E_m]\!]$
  oraz $\overline{C}, h', \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_m[\![E_m]\!] \rightarrow^* h'', \mathcal{C}'_1[\![E'_1]\!] :: \cdots :: \mathcal{C}'_k[\![E'_k]\!]$,
  to $\overline{C}, h, \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!] \rightarrow^* h'', \mathcal{C}'_1[\![E'_1]\!] :: \cdots :: \mathcal{C}'_k[\![E'_k]\!]$.

Relację $\rightarrow^{(*)_n}$ definiujemy jako najmniejszą relację spełniającą warunki:

- $\overline{C}, h, \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!] \rightarrow^{(*)_n} h, \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!]$;
- jeśli $\overline{C}, h, \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!] :: \cdots :: \mathcal{C}_k[\![E_k]\!] \rightarrow$
  $h', \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!] :: \cdots :: \mathcal{C}'_{k'}[\![E'_{k'}]\!]$, gdzie $k' \geq n$
  oraz $\overline{C}, h', \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!] :: \cdots :: \mathcal{C}'_{k'}[\![E'_{k'}]\!] \rightarrow^{(*)_n}$
  $h'', \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!] :: \cdots :: \mathcal{C}''_{k''}[\![E''_{k''}]\!]$,
  to $\overline{C}, h, \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!] :: \cdots :: \mathcal{C}_k[\![E_k]\!] \rightarrow^{(*)_n}$
  $h'', \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!] :: \cdots :: \mathcal{C}''_{k''}[\![E''_{k''}]\!]$.

Dodatkowo piszemy

$$\overline{C}, h, \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!] \rightarrow^{(*)} h'', \mathcal{C}'_1[\![E'_1]\!] :: \cdots :: \mathcal{C}'_m[\![E'_m]\!],$$

jeśli zachodzi

$$\overline{C}, h, \mathcal{C}_1[\![E_1]\!] :: \cdots :: \mathcal{C}_n[\![E_n]\!] \rightarrow^{(*)_n} h'', \mathcal{C}'_1[\![E'_1]\!] :: \cdots :: \mathcal{C}'_m[\![E'_m]\!].$$

Relacja przejścia

Obecnie możemy przedstawić właściwe reguły opisujące semantyczną relację przejścia $\rightarrow$. Najpierw przedstawimy reguły opisujące normalne wykonanie programu (bez propagacji wyjątków), a później przejdziemy do opisu propagacji wyjątków.

Reguły określające semantykę wyrażenia new:

$$\frac{\begin{array}{c} n \geq 0 \qquad \mathsf{alloc}(h) = (l_0, h'') \qquad E_0 = \mathsf{body}(\overline{C}, C, k) \\ l_1, \ldots, l_n \in \mathsf{Loc} \\ \mathsf{params}(\overline{C}, C, k) = (y_1, \ldots, y_n) \quad E = E_0\{l_1/y_1, \ldots, l_n/y_n\} \\ h'(l) = \begin{cases} h''(l) & \text{dla } l \neq l_0 \\ \mathsf{empty}_{\overline{C},C} & \text{dla } l = l_0 \end{cases} \end{array}}{\begin{array}{l} \overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![\textbf{new } C.k(l_1, \ldots, l_n)]\!]^{l'}_{\emptyset} \rightarrow \\ \qquad h', \overline{\mathcal{C}} :: \mathcal{C}[\![\textbf{new } C.k(l_1, \ldots, l_n)]\!]^{l'}_{\emptyset} :: [\![E]\!]^{l_0}_{\emptyset} \end{array}} \quad \textit{(newk)}$$

$$\frac{n \geq 0 \qquad l \in \mathsf{Loc}}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![\textbf{new } C.k(l_1, \ldots, l_n)]\!]^{l'}_{\emptyset} :: [\![l]\!]^{l''}_{\emptyset} \rightarrow h, \overline{\mathcal{C}} :: \mathcal{C}[\![l]\!]^{l'}_{\emptyset}} \quad \textit{(newret)}$$

Reguły określające semantykę wyrażenia let:

$$\frac{l \in \mathsf{Loc}}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\textbf{let } C\ x = [\![l]\!]^{l'}_{\emptyset} \textbf{ in } E] \rightarrow h, \overline{\mathcal{C}} :: \mathcal{C}[\![E\{l/x\}]\!]^{l'}_{\emptyset}} \quad \textit{(let)}$$

Reguły określające semantykę wyrażenia if:

$$\frac{}{\begin{array}{l} \overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3]\!]^{l'}_{\emptyset} \rightarrow \\ \qquad h, \overline{\mathcal{C}} :: \mathcal{C}[\textbf{if } [\![E_1]\!]^{l'}_{\emptyset} \textbf{ then } E_2 \textbf{ else } E_3] \end{array}} \quad \textit{(ifin)}$$

$$\frac{l \neq \textbf{null}}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\textbf{if } [\![l]\!]^{l'}_{\emptyset} \textbf{ then } E_2 \textbf{ else } E_3] \rightarrow h, \overline{\mathcal{C}} :: \mathcal{C}[\![E_2]\!]^{l'}_{\emptyset}} \quad \textit{(ifnonnull)}$$

$$\frac{}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\textbf{if } [\![\textbf{null}]\!]^{l'}_{\emptyset} \textbf{ then } E_2 \textbf{ else } E_3] \rightarrow h, \overline{\mathcal{C}} :: \mathcal{C}[\![E_3]\!]^{l'}_{\emptyset}} \quad \textit{(ifnull)}$$

Reguły określające semantykę wywołania metody:

$$\frac{}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![\textbf{null}.m(\bar{l})]\!]^{l'}_{\emptyset} \rightarrow h, \overline{\mathcal{C}} :: \mathcal{C}[\![\mathsf{npe}]\!]^{l'}_{\mathsf{npe}}} \quad \textit{(methodnpe)}$$

$$\frac{\begin{array}{c} \mathsf{class}(h, l) = D \quad \mathsf{body}(\overline{C}, D, m) = E_0 \quad E = E_0[l/\textbf{this}] \\ n \geq 0 \qquad l, l_1, \ldots, l_n \in \mathsf{Loc} \qquad l \neq \textbf{null} \end{array}}{\begin{array}{l} \overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![l.m(l_1, \ldots, l_n)]\!]^{l'}_{\emptyset} \rightarrow \\ \qquad h, \overline{\mathcal{C}} :: \mathcal{C}[\![l.m(l_1, \ldots, l_n)]\!]^{l'}_{\emptyset} :: [\![E]\!]^{l}_{\emptyset} \end{array}} \quad \textit{(method)}$$

$$\frac{l_{n+1} \in \mathsf{Loc} \qquad n \geq 0}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![l.m(l_1, \ldots, l_n)]\!]^{l'}_{\emptyset} :: [\![l_{n+1}]\!]^{l''}_{\emptyset} \rightarrow h, \overline{\mathcal{C}} :: \mathcal{C}[\![l_{n+1}]\!]^{l'}_{\emptyset}} \quad \textit{(methodret)}$$

Reguły określające semantykę przypisania:

$$\frac{l, l_1 \in \mathsf{Loc} \qquad l_2 = h(l_1)(x) \qquad h'(l') = \begin{cases} h(l') & \text{dla } l' \neq l_2 \\ l & \text{dla } l' = l_2 \end{cases}}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[l_1.x = [\![l]\!]^{l'}_{\emptyset}] \to h', \overline{\mathcal{C}} :: \mathcal{C}[\![l]\!]^{l'}_{\emptyset}} \quad \textit{(assignev)}$$

Reguły określające semantykę odwołania do zmiennej:

$$\frac{}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![\mathbf{null}.x]\!]^{l'}_{\emptyset} \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![\mathsf{npe}]\!]^{l'}_{\mathsf{npe}}} \quad \textit{(varrefnull)}$$

$$\frac{l \in \mathsf{Loc} \quad l \neq \mathbf{null} \quad l'' = h(l)(x)}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![l.x]\!]^{l'}_{\emptyset} \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![l'']\!]^{l'}_{\emptyset}} \quad \textit{(varrefnonull)}$$

Reguły określające semantykę rzucania wyjątku:

$$\frac{l \in \mathsf{Loc} \qquad \mathsf{class}(h, l) = D}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![\mathbf{throw}\ l]\!]^{l'}_{\emptyset} \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![l]\!]^{l'}_{D}} \quad \textit{(throw)}$$

Reguły określające semantykę łapania wyjątku:

$$\frac{}{\begin{array}{l}\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![\mathbf{try}\ \{E_1\}\ \mathbf{catch}\ (C\ x)\ \{E_2\}]\!]^{l'}_{\emptyset} \to \\ \qquad\qquad h, \overline{\mathcal{C}} :: \mathcal{C}[\mathbf{try}\ \{[\![E_1]\!]^{l'}_{\emptyset}\}\ \mathbf{catch}\ (C\ x)\ \{E_2\}]\end{array}} \quad \textit{(catchin)}$$

$$\frac{l \in \mathsf{Loc}}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\mathbf{try}\ \{[\![l]\!]^{l'}_{\emptyset}\}\ \mathbf{catch}\ (C\ x)\ \{E_2\}] \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![l]\!]^{l'}_{\emptyset}} \quad \textit{(catchnormal)}$$

$$\frac{l \in \mathsf{Loc} \qquad E_2' = E_2\{l/x\} \qquad C' \leq: C}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\mathbf{try}\ \{[\![l]\!]^{l'}_{C'}\}\ \mathbf{catch}\ (C\ x)\ \{E_2\}] \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![E_2']\!]^{l'}_{\emptyset}} \quad \textit{(catchexok)}$$

Reguły propagacji wyjątków:

$$\frac{l \in \mathsf{Loc} \qquad C \neq \emptyset}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![\mathbf{new}\ C.k(l_1, \ldots, l_n)]\!]^{l'}_{\emptyset} :: [\![l]\!]^{l''}_{C} \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![l]\!]^{l'}_{C}} \quad \textit{(newex)}$$

$$\frac{l \in \mathsf{Loc} \qquad C \neq \emptyset}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\mathbf{let}\ C\ x = [\![l]\!]^{l'}_{C}\ \mathbf{in}\ E] \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![l]\!]^{l'}_{C}} \quad \textit{(letex)}$$

$$\frac{l \in \mathsf{Loc} \qquad C \neq \emptyset}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\textbf{if } [\![l]\!]^{l'}_{C} \textbf{ then } E_2 \textbf{ else } E_3] \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![l]\!]^{l'}_{C}} \quad \textit{(ifex)}$$

$$\frac{l_{n+1} \in \mathsf{Loc} \qquad C \neq \emptyset}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![l.m(l_1, \ldots, l_n)]\!]^{l'}_{\emptyset} :: [\![l_{n+1}]\!]^{l''}_{C} \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![l_{n+1}]\!]^{l'}_{C}} \quad \textit{(methodex)}$$

$$\frac{l \in \mathsf{Loc} \qquad C \neq \emptyset}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[l_1.x_2 = [\![l]\!]^{l'}_{C}] \to h', \overline{\mathcal{C}} :: \mathcal{C}[\![l]\!]^{l'}_{C}} \quad \textit{(assignex)}$$

$$\frac{l \in \mathsf{Loc} \qquad C' \neq \emptyset \qquad C' \not\leq: C}{\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\textbf{try } \{[\![l]\!]^{l'}_{C'}\} \textbf{ catch } (C\ x)\ \{E_2\}] \to h, \overline{\mathcal{C}} :: [\![l]\!]^{l'}_{C'}} \quad \textit{(catchexnok)}$$

2.4 Semantyczna definicja funkcyjności

W celu określenia kluczowego dla pojęcia funkcyjności terminu funkcji potrzebne jest najpierw opisanie, czym są wartości, na jakich operują funkcje. W wypadku języków obiektowych nie jest to oczywiste, gdyż chcielibyśmy z jednej strony móc myśleć o wartościach reprezentowanych przez obiekty, a z drugiej strony takie obiekty mają zmienny stan oraz złożoną budowę. W związku z tym wprowadzimy pojęcie reprezentacji określające, jaka część sterty odpowiada za reprezentacje konkretnego obiektu rozumianego jako wartość.

Definicja 11 `(reprezentacja)`
Reprezentacja obiektu $h(l)$, ozn. $\mathsf{rep}(l, h)$ to zbiór lokacji zdefiniowany następująco:

- $l \in \mathsf{rep}(l, h)$,
- jeśli $l' \in \mathsf{rep}(l, h)$ i $C = \mathsf{typeof}(h(l'))$ oraz $f \in \mathsf{fields}(\overline{C}, C)$, to $h(l')(f) \in \mathsf{rep}(l, h)$.

Intuicyjnie, do reprezentacji obiektu należy jego własne pola oraz pola z domknięcia przechodniego branego po polach oznaczonych jako **rep**.

Definicja 12 `(obserwowalny ciąg kontekstów)`
Mówimy, że ciąg kontekstów $\overline{\mathcal{C}}$ jest obserwowalny ze względu na lokację l w stercie h dla programu $\overline{C}$, gdy ma postać:

- $\overline{\mathcal{C}}_0 :: \mathcal{C}[\![\textbf{new } C.k(\bar{l})]\!]^{l'}_{\emptyset} :: [\![l]\!]^{l''}_{\emptyset}$,
- $\overline{\mathcal{C}}_0 :: \mathcal{C}[\![l.x]\!]^{l'}_{\emptyset}$ i $l \neq l'$,
- $\overline{\mathcal{C}}_0 :: \mathcal{C}[\![[l.x = l']\!]^{l''}_{\emptyset}]$, gdzie $l' \in \mathsf{Loc}$,
- $\overline{\mathcal{C}}_0 :: \mathcal{C}[\![l.m(\bar{l})]\!]^{l'}_{\emptyset}$,
- $\overline{\mathcal{C}}_0 :: \mathcal{C}[\![l.m(\bar{l})]\!]^{l'}_{\emptyset} :: [\![l']\!]^{l'}_{A}$, gdzie A jest dowolną etykietą, a $l' \in \mathsf{Loc}$.

Powyższa definicja jest wzorowana na definicji stanów widzialnych (ang. visible states) z [JML06, Section 8.2].

Teraz jesteśmy już gotowi do określenia, które obiekty będziemy uważali za dopuszczalne wartości. Wzorujemy się tutaj na pojęciu niemutowalności wprowadzonym w [HPSS07].

Definicja 13 `(obiekty niemutowalne)`
Obiekt $h(l)$ jest niemutowalny w h dla programu $\overline{C}$, gdy dla każdej pary ciągów kontekstów $\overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2$ obserwowalnych ze względu na l takich, że

$$\overline{C}, h, \overline{\mathcal{C}}_1 \rightarrow^* h', \overline{\mathcal{C}}_2$$

spełnione są warunki:

- $\mathsf{rep}(l, h) = \mathsf{rep}(l, h')$ oraz
- dla każdego $l' \in \mathsf{rep}(l, h')$ zachodzi $h(l') = h'(l')$.

Intuicyjnie, obiekt jest niemutowalny, gdy nie można zmienić zawartości żadnego z podobiektów, które stanowią jego reprezentację. Możemy teraz rozszerzyć tę definicję na klasy:

Definicja 14 `(klasy niemutowalne)`
Klasa C jest niemutowalna dla programu $\mathcal{C}$, jeśli dla każdej lokacji l i sterty h spełniających $\mathsf{class}(h, l) = C$ zachodzi, że obiekt $h(l)$ jest niemutowalny w h dla programu $\mathcal{C}$.

Mając pojęcie obiektów niemutowalnych, możemy dla nich sensownie określić równość. Równość ta będzie użyta do określenia naszego pojęcia funkcyjności.

Definicja 15 `(równość obiektów niemutowalnych)`
Niech $h_1, h_2 \in \mathsf{Heap}$ oraz $l_1 \in \mathsf{Dom}(h_1), l_2 \in \mathsf{Dom}(h_2)$. Relację równości obiektów niemutowalnych w programie $\overline{C}$, ozn. $l_1[h_1] =_{\overline{C}} l_2[h_2]$, określamy jako największą relację równoważności zawartą w $\mathsf{Heap} \times \mathsf{Loc}$ spełniającą warunki:

- $\mathsf{typeof}(l_1(h_1)) = \mathsf{typeof}(l_2(h_2)) = C$ i C jest klasą niemutowalną,
- dla każdej deklaracji C_1 [fmod] $x \in \mathsf{fields}(\overline{C}, C)$
  - jeśli fmod w deklaracji nie występuje, to $h_1(l_1)(x) = h_2(l_2)(x)$,
  - jeśli $\mathsf{fmod} = \mathbf{rep}$, to $h_1(l_1)(x)[h_1] =_{\overline{C}} h_2(l_2)(x)[h_2]$.

Jeśli $\overline{C}$ jest jasne z kontekstu lub nieistotne, piszemy $l_1[h_1] = l_2[h_2]$.

Powyższa definicja jest tak skonstruowana, że pozwala na porównywanie także niemutowalnych struktur, które są zacyklone.

Możemy teraz przystąpić do zdefiniowania pierwszego istotnego warunku koniecznego dla naszego pojęcia funkcyjności — ekstensjonalności metod.

Definicja 16 `(ekstensjonalność metod)`
Niech metoda m w klasie $C \in \overline{C}$ będzie zadeklarowana:

$$C_{n+1}[\mathsf{mmod}]\ m(\mathsf{args})\ [\mathbf{throws}\ \overline{\mathsf{Exc}}]\ \{E\},$$

gdzie $\mathsf{args} = C_1\ x_1, \ldots, C_n\ x_n$ oraz $\overline{\mathsf{Exc}} = D_1, \ldots, D_k$. Niech dodatkowo typy $C, C, C_1, \ldots, C_{n+1}, D_1, \ldots, D_k$ będą niemutowalne. Metoda m jest *ekstensjonalna*, gdy dla każdych $h_1, h_1', h_2, h_2' \in \mathsf{Heap}$ oraz $l, l_1, \ldots, l_{n+1}, l', l_1', \ldots, l_{n+1}'$ takich, że

$$\begin{gathered} C = \mathsf{typeof}(h_1(l)), \quad C_i = \mathsf{typeof}(h_1(l_i)) \text{ dla } i = 1, \ldots, n, \\ \mathsf{typeof}(h_1'(l_{n+1})) \in \{C_{n+1}, D_1, \ldots, D_k\}, \\ C = \mathsf{typeof}(h_2(l')), \quad C_i = \mathsf{typeof}(h_2(l_i')) \text{ dla } i = 1, \ldots, n, \\ \mathsf{typeof}(h_2'(l_{n+1}')) \in \{C_{n+1}, D_1, \ldots, D_k\} \end{gathered}$$

jeśli zachodzą warunki:

- $l[h_1] = l'[h_2]$ i $l_i[h_1] = l_i'[h_2]$ dla $i = 1, \ldots n$,
- $\overline{C}, h_1, \overline{\mathcal{C}} :: \mathcal{C}[\![l.m(l_1, \ldots, l_n)]\!]_\emptyset \to^{(*)} h_1', \overline{\mathcal{C}} :: \mathcal{C}[\![l.m(l_1, \ldots, l_n)]\!]_\emptyset :: [\![l_{n+1}]\!]_A$,
- $\overline{C}, h_2, \overline{\mathcal{C}} :: \mathcal{C}[\![l'.m(l_1', \ldots, l_n')]\!]_\emptyset \to^{(*)} h_2', \overline{\mathcal{C}} :: \mathcal{C}[\![l'.m(l_1', \ldots, l_n')]\!]_\emptyset :: [\![l_{n+1}']\!]_A$,

to

$$l_{n+1}[h_1'] = l_{n+1}'[h_2'].$$

Do zdefiniowania drugiego pojęcia, czystości metod, dodatkowo potrzebujemy określenia dopasowania stert z dokładnością do wskazanego zbioru lokacji (na którym mogą się one różnić):

Definicja 17 `(dopasowanie stert poza zbiorem)`
Niech A będzie zbiorem lokacji. Określamy relację $\sqsubseteq_A$ jak następuje:

$h_1 \sqsubseteq_A h_2$ wtedy i tylko wtedy gdy dla każdego l takiego, że $l \notin A$ oraz $l \in \mathsf{Dom}(h_1)$, zachodzi $h_1(l) = h_2(l)$.

Definicja 18 `(metoda czysta)`
Mówimy, że metoda m w klasie $C \in \overline{C}$ jest *czysta*, gdy dla każdej sterty $h \in \mathsf{Heap}$, ciągu kontekstów $^{:}C :: \mathcal{C}[\![]\!]_\emptyset$ oraz dowolnych lokacji $l, l_1, \ldots, l_n$ takich, że $\mathsf{typeof}(()h(l)) = C$ oraz typy $l_1, \ldots, l_n$ odpowiednio zgadzają się z parametrami formalnymi metody m zachodzi: jeśli

$$\overline{C}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![l.m(l_1, \ldots, l_n)]\!]_\emptyset \rightarrow^{(*)} h', \overline{\mathcal{C}} :: \mathcal{C}[\![l.m(l_1, \ldots, l_n)]\!]_\emptyset :: [\![l']\!]_D,$$

to $h \sqsubseteq_\emptyset h'$.

Definicja 19 `(metoda funkcyjna)`
Mówimy, że metoda m w klasie $C \in \overline{C}$ jest *funkcyjna*, gdy jest ekstensjonalna i czysta.

2.5 Semantyczna definicja local-state

W naszym efektywnym modelu do stwierdzenia zachodzenia funkcyjności będziemy używali pomocniczego pojęcia *local-state*. Obecnie podamy semantyczną definicję tego ostatniego.

Zaczniemy od definicji pomocniczych, które pozwolą nam na wyrażenie tego pojęcia.

Definicja 20 `(stan związany z wywołaniem metody)`
Niech metoda m w klasie $C \in \overline{C}$ będzie zadeklarowana:

$$C_{n+1}[\mathsf{mmod}]\ m(\mathsf{args})\ [\mathbf{throws}\ \overline{\mathsf{Exc}}]\ \{E\},$$

gdzie $\mathsf{args} = C_1\ x_1, \ldots, C_n\ x_n$. Dla dowolnej sterty $h \in \mathsf{Heap}$, obiektu $h(l)$ takiego, że $\mathsf{typeof}(h(l)) = C$, oraz lokacji $l_i \in \mathsf{Loc}$ takich, że $\mathsf{typeof}(h(l_i)) = C_i$ dla $i = 1, \ldots, n$, określamy *stan związany z wywołaniem metody* jako:

$$\mathsf{Rlvnt}(\overline{C}, C, m, h, l, l_1, \ldots, l_n) = \mathsf{rep}(l, h) \cup \bigcup_{1=1}^{n} \mathsf{rep}(l_i, h).$$

Teraz możemy zdefiniować pojęcie *local-state*:

Definicja 21 `(metody local-state)`
Niech metoda m w klasie $C \in \overline{C}$ będzie zadeklarowana:

$$C_{n+1}[\mathsf{mmod}]\ m(\mathsf{args})\ [\mathbf{throws}\ \overline{\mathsf{Exc}}]\ \{E\},$$

gdzie $\mathsf{args} = C_1\ x_1, \ldots, C_n\ x_n$. Powiemy, że metoda ta jest *local-state*, gdy dla dowolnych stert $h_1, h_2 \in \mathsf{Heap}$, lokacji l, takiej że $\mathsf{typeof}(h(l)) = C$, lokacji

$l_i \in \mathsf{Loc}$ takich, że $\mathsf{typeof}(h(l_i)) = C_i$ dla $i = 1, \ldots, n$, lokacji l' oraz ciągu kontekstów $\overline{\mathcal{C}} :: \mathcal{C}[\![\ ]\!]_\emptyset$, jeśli

$$\overline{C}, h_1, \overline{\mathcal{C}} :: \mathcal{C}[\![l.m(l_1, \ldots, l_n)]\!]_\emptyset \rightarrow^{(*)} h_2, \overline{\mathcal{C}} :: \mathcal{C}[\![l.m(l_1, \ldots, l_n)]\!]_\emptyset :: [\![l']\!],$$

to $h_1 \sqsubseteq_A h_2$, gdzie $A = \mathsf{Rlvnt}(\overline{C}, C, m, h_1, l, l_1, \ldots, l_n)$.

Rozdział 3

Efektywny model pojęcia funkcyjności

Przedstawiona w poprzednim rozdziale definicja pojęcia funkcyjności nie jest efektywna — jej stwierdzanie dla programów jest w ogólnym wypadku nierozstrzygalne. W związku z tym w obecnym rozdziale przedstawimy system sprawdzania tych własności, który w intencji ma je gwarantować. Oczywiście jednocześnie system ten dla niektórych metod nie będzie w stanie stwierdzić ich funkcyjności, mimo iż w rzeczywistości własność ta będzie zachodzić.

Rozdział ten podzielony jest na pięć części. W sekcji 3.1 zaprezentujemy reguły sprawdzania poprawności klas. Następnie w sekcjach 3.2, 3.3, 3.4 opiszemy odpowiednio reguły sprawdzania poprawności pól, konstruktorów i metod, by wreszcie w sekcji 3.5 przedstawić reguły przypisywania klas do poszczególnych wyrażeń naszego języka.

3.1 Sprawdzanie poprawności klas

Przedstawione tutaj reguły zapoczątkowują proces sprawdzania poprawności typowania programu. W istocie sprowadzają one poprawność programu do poprawności jego klas, a poprawność klas jest sprowadzana do poprawności ich składników.

$$\frac{\forall C \in \overline{C}.\ \overline{C} \vdash C : \mathsf{ok}}{\vdash \overline{C} : \mathsf{ok}} \quad \textit{(prog)}$$

$$\frac{\begin{array}{c}\forall F \in \mathsf{fields}(\overline{C}, C).\ \overline{C}; C : \mathsf{modifier}(\overline{C}, C) \vdash F : \mathsf{ok} \\ \forall K \in \mathsf{konstructors}(\overline{C}, C).\ \overline{C}; C : \mathsf{modifier}(\overline{C}, C) \vdash K : \mathsf{ok} \\ \forall M \in \mathsf{methods}(\overline{C}, C).\ \overline{C}; C : \mathsf{modifier}(\overline{C}, C) \vdash M : \mathsf{ok}\end{array}}{\overline{C} \vdash C : \mathsf{ok}} \quad \textit{(class)}$$

3.1.1 Sprawdzanie poprawności relacji podtypiania

$$\frac{\forall C \in \overline{C}.\overline{C} \vdash C : \mathsf{subok}}{\overline{C} \vdash \mathsf{subok}} \quad \textit{(subok)}$$

$$\frac{\begin{array}{c} C \in \overline{C} \quad \mathsf{extends}(\overline{C}, C) = D \quad \overline{C} \vdash D : \mathsf{subok} \\ \mathsf{modifier}(\overline{C}, C) = \mathbf{imm} \Rightarrow \mathsf{isImmutable}(\overline{C}, D) \end{array}}{\overline{C} \vdash C : \mathsf{subok}} \quad \textit{(extssubokimm)}$$

$$\frac{\begin{array}{c} C \in \overline{C} \quad \mathsf{extends}(\overline{C}, C) = D \quad \overline{C} \vdash D : \mathsf{subok} \\ \mathsf{modifier}(\overline{C}, C) = \mathbf{func} \Rightarrow \mathsf{modifier}(\overline{C}, D) = \mathbf{func} \end{array}}{\overline{C} \vdash C : \mathsf{subok}} \quad \textit{(extssubokfunc)}$$

$$\frac{\begin{array}{c} C \in \overline{C} \quad \mathsf{extends}(\overline{C}, C) = D \quad \overline{C} \vdash D : \mathsf{subok} \\ \mathsf{modifier}(\overline{C}, C) = \emptyset \Rightarrow \mathsf{modifier}(\overline{C}, D) = \emptyset \end{array}}{\overline{C} \vdash C : \mathsf{subok}} \quad \textit{(extssubokfunc)}$$

$$\frac{\begin{array}{c} C \in \overline{C} \\ \mathsf{extends}(\overline{C}, C) \text{ nieokreślone} \end{array}}{\overline{C} \vdash C : \mathsf{subok}} \quad \textit{(extendssubok)}$$

3.2 Sprawdzanie poprawności pól

$$\frac{F \equiv D\ [\mathsf{fmod}]\ x \qquad D \in \mathsf{Dom}(\overline{C})}{\overline{C}; C : \emptyset \vdash F : \mathsf{ok}} \quad \textit{(normal field)}$$

$$\frac{F \equiv D\ [\mathsf{fmod}]\ x \qquad D \in \mathsf{Dom}(\overline{C})}{\overline{C}; C : \mathbf{imm} \vdash F : \mathsf{ok}} \quad \textit{(immutable field)}$$

$$\frac{\begin{array}{c} F \equiv D\ [\mathsf{fmod}]\ x \qquad D \in \mathsf{Dom}(\overline{C}) \\ \mathsf{isImmutable}(\overline{C}, \mathsf{type}(\overline{C}, C, x)) \quad \mathsf{modifier}(\overline{C}, C, x) = \mathbf{rep} \end{array}}{\overline{C}; C : \mathbf{func} \vdash F : \mathsf{ok}} \quad \textit{(func rep field)}$$

$$\frac{\begin{array}{c} F \equiv D\ [\mathsf{fmod}]\ x \qquad D \in \mathsf{Dom}(\overline{C}) \\ \mathsf{modifier}(\overline{C}, C, x) \neq \mathbf{rep} \end{array}}{\overline{C}; C : \mathbf{imm} \vdash F : \mathsf{ok}} \quad \textit{(func nrep field)}$$

3.3 Sprawdzanie poprawności konstruktorów

Sprawdzanie poprawności konstruktorów odbywa się za pomocą reguły:

$$\frac{\begin{array}{c} Ex = \mathsf{throws}(\overline{C}, C, K) \\ Loc = \mathsf{par2loc}(\mathsf{params}(\overline{C}, C, K), \mathsf{kmod}) \\ \mathsf{modifier}(\overline{C}, C, K) = \mathsf{kmod} \\ \forall D \in Ex. D \in \overline{C} \qquad Loc \vdash_{\mathsf{this}} \mathsf{body}(\overline{C}, C, K) : \mathsf{this} \\ \overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash \mathsf{body}(\overline{C}, C, K) : \mathsf{cok} \end{array}}{\overline{C}; C : \mathsf{cmod} \vdash K : \mathsf{ok}} \quad \textit{(kons cok ok)}$$

W regule tej użyto operacji par2loc. Operacja ta przekształca ciąg deklaracji postaci Cx w zbiór przypisań postaci $x : \langle C, \mathsf{local}, \mathsf{nthis}\rangle$, jeśli parametr $\mathsf{kmod} = \mathbf{lstate}$ lub $x : \langle C, \mathsf{nlocal}, \mathsf{nthis}\rangle$, jeśli parametr $\mathsf{kmod} \neq \mathbf{lstate}$.

Intuicyjnie cok sprawdza, że wyrażenie jest prawidłowo zbudowane ze względu na oczekiwane własności $\mathbf{func}, \mathbf{lstate}$ dla konstruktorów oraz ma poprawny typ.

Intuicyjnie $\vdash_{\mathsf{this}}$ sprawdza, czy wyrażenie daje w wyniku **this**. Technicznie, użyte w osądzie $Loc \vdash_{\mathsf{this}} \mathsf{body}(\overline{C}, C, K) : \mathsf{this}$ otoczenie Loc przypisuje zmiennym trójki, a osąd będzie wymagał w tym miejscu par. W związku z tym przyjmujemy tutaj implicite konwersję trójek na pary z odrzuceniem drugiej współrzędnej.

3.3.1 Reguły dla określania lokalności

Przedstawiamy tutaj definicją osądów postaci $Loc \vdash_{\mathsf{local}} E : \tau$, gdzie τ należą do zbioru $\mathcal{L} = \{\mathsf{local}, \mathsf{nlocal}\}$, zaś Loc jest zbiorem przyporządkowań postaci $x : \langle C, \sigma\rangle$ takich, że każde x występuje co najwyżej raz w Loc, C jest nazwą klasy, zaś $\sigma \in \mathcal{L}$. Osądy te określają, czy wyrażenie E daje w otoczeniu Loc obiekt, który został utworzony lokalnie lub nielokalnie. Przy wyprowadzaniu tych własności używana będzie operacja $\wedge : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$. Operacja ta działa zgodnie z poniższą tabelką:

$\wedge$	local	nlocal
local	local	nlocal
nlocal	nlocal	nlocal

Operację tę w naturalny sposób rozszerzamy do operacji wieloargumentowej $\bigwedge$ (local możemy zinterpretować jako prawdę, a nlocal jako fałsz, wtedy $\wedge$ będzie w naturalny sposób odpowiadać koniunkcji).

$$\frac{\begin{array}{c}\forall y_i \in \overline{x}.Loc \vdash_{\mathsf{local}} y_i : \tau_i \\ \mathsf{isLocalState}(\overline{C}, C, k)\end{array}}{Loc \vdash_{\mathsf{local}} \mathbf{new}\ D.k(\overline{x}) : \bigwedge_{y_i \in y} \tau_i} \quad \textit{(localnew)}$$

$$\frac{\neg\mathsf{isLocalState}(\overline{C}, C, k)}{Loc \vdash_{\mathsf{local}} \mathbf{new}\ D.k(\overline{x}) : \mathsf{nlocal}} \quad \textit{(nlocalnew)}$$

$$\frac{Loc \vdash_{\mathsf{local}} E_1 : \tau_1 \qquad Loc, x : \langle C, \tau_1 \rangle \vdash_{\mathsf{local}} E_2 : \tau_2}{Loc \vdash_{\mathsf{local}} \mathbf{let}\ C\ x = E_1\ \mathbf{in}\ E_2 : \tau_2} \quad \textit{(localllet)}$$

$$\frac{Loc \vdash_{\mathsf{local}} E_2 : \tau_1 \qquad Loc \vdash_{\mathsf{local}} E_3 : \tau_2}{Loc \vdash_{\mathsf{local}} \mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 : \tau_1 \wedge \tau_2} \quad \textit{(localif)}$$

$$\frac{\begin{array}{c}Loc \vdash_{\mathsf{local}} x : \tau \qquad \forall y_i \in \overline{x}.Loc \vdash_{\mathsf{local}} y_i : \tau_i \\ \mathsf{isLocalState}(\overline{C}, C, m)\end{array}}{Loc \vdash_{\mathsf{local}} x.m(\overline{x}) : \tau \wedge \bigwedge_{y_i \in y} \tau_i} \quad \textit{(localmeth)}$$

$$\frac{\neg\mathsf{isLocalState}(\overline{C}, C, m)}{Loc \vdash_{\mathsf{local}} x.m(\overline{x}) : \mathsf{nlocal}} \quad \textit{(nlocalmeth)}$$

$$\frac{}{Loc \vdash_{\mathsf{local}} \mathbf{this} : \mathsf{nlocal}} \quad \textit{(localthis)}$$

$$\frac{}{Loc, x : \langle C, \tau \rangle \vdash_{\mathsf{local}} x : \tau} \quad \textit{(localvar)}$$

$$\frac{}{Loc, x : \langle C, \mathsf{nlocal} \rangle \vdash_{\mathsf{local}} x.y : \mathsf{nlocal}} \quad \textit{(nlocalobj)}$$

$$\frac{\mathsf{modifier}(\overline{C}, C, y) = \mathbf{rep}}{Loc, x : \langle C, \mathsf{local} \rangle \vdash_{\mathsf{local}} x.y : \mathsf{local}} \quad \textit{(localrepfield)}$$

$$\frac{\mathsf{modifier}(\overline{C}, C, y) \neq \mathbf{rep}}{Loc, x : \langle C, \mathsf{local} \rangle \vdash_{\mathsf{local}} x.y : \mathsf{nlocal}} \quad \textit{(localfield)}$$

W powyższej regule możemy uznać, że odwołanie do pola jest nielokalne, gdyż gdyby było lokalne, to i tak mamy w środowisku (na którejś zmiennej zadeklarowanej w wyrażeniu typu **let**) dostępną referencję do tej lokalnej wartości.

$$\frac{Loc \vdash_{\mathsf{local}} y : \tau}{Loc \vdash_{\mathsf{local}} x.f = y : \tau} \quad \textit{(localassign)}$$

$$\frac{}{Loc \vdash_{\mathsf{local}} \mathbf{throw}\ x : \mathsf{local}} \quad \textit{(localthrw)}$$

$$\frac{Loc \vdash_{\mathsf{local}} E_1 : \tau_1 \qquad Loc, x : \langle C, \mathsf{nlocal} \rangle \vdash_{\mathsf{local}} E_2 : \tau_2}{Loc \vdash_{\mathsf{local}} \mathbf{try}\ \{E_1\}\ \mathbf{catch}\ (C\ x)\ \{E_2\} : \tau_1 \wedge \tau_2} \quad \textit{(localtry)}$$

$$\frac{}{Loc \vdash_{\mathsf{local}} \mathbf{null} : \mathsf{local}} \quad \textit{(localnull)}$$

3.3.2 Reguły określające, czy wynikiem jest this

Przedstawiamy tutaj definicją osądów postaci $Loc \vdash_{\mathsf{this}} E : \tau$, gdzie τ należą do zbioru $\mathcal{T} = \{\mathsf{this}, \mathsf{nthis}\}$, zaś Loc jest zbiorem przyporządkowań postaci $x : \langle C, \sigma \rangle$ takich, że każde x występuje co najwyżej raz w Loc, C jest nazwą klasy, zaś $\sigma \in \mathcal{T}$. Osądy te określają, czy wyrażenie E daje w otoczeniu Loc obiekt, który jest równy aktualnemu obiektowi oznaczonemu jako **this**.

Przy wyprowadzaniu tych własności używana będzie operacja $\wedge : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$. Operacja ta działa zgodnie z poniższą tabelką:

$\wedge$	this	nthis
this	this	nthis
nthis	nthis	nthis

Operację tę w naturalny sposób rozszerzamy do operacji wieloargumentowej $\bigwedge$ na podobnej zasadzie, jak w wypadku lokalności.

$$\frac{}{Loc \vdash_{\mathsf{this}} \mathbf{new}\ D.k(\overline{x}) : \mathsf{nthis}} \quad \textit{(thisnew)}$$

$$\frac{Loc \vdash_{\mathsf{this}} E_1 : \tau_1 \qquad Loc, x : \langle C, \tau_1 \rangle \vdash_{\mathsf{this}} E_2 : \tau_2}{Loc \vdash_{\mathsf{this}} \mathbf{let}\ C\ x = E_1\ \mathbf{in}\ E_2 : \tau_2} \quad \textit{(thislet)}$$

$$\frac{Loc \vdash_{\mathsf{this}} E_2 : \tau_1 \qquad Loc \vdash_{\mathsf{this}} E_3 : \tau_2}{Loc \vdash_{\mathsf{this}} \mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 : \tau_1 \wedge \tau_2} \quad \textit{(thisif)}$$

$$\frac{}{Loc \vdash_{\mathsf{this}} x.m(\overline{x}) : \mathsf{nthis}} \quad \textit{(thismeth)}$$

$$\frac{}{Loc \vdash_{\mathsf{this}} \mathbf{this} : \mathsf{this}} \quad \textit{(thisthis)}$$

$$\frac{}{Loc, x : \langle C, \tau \rangle \vdash_{\mathsf{this}} x : \tau} \quad \textit{(thisvar)}$$

$$\frac{}{Loc, x : \langle C, \tau \rangle \vdash_{\mathsf{this}} x.y : \mathsf{nthis}} \quad \textit{(thisobj)}$$

$$\frac{Loc \vdash_{\mathsf{this}} y : \tau}{Loc \vdash_{\mathsf{this}} x.f = y : \tau} \quad \textit{(thisassign)}$$

$$\frac{}{Loc, x : \langle C, \tau \rangle \vdash_{\mathsf{this}} \mathbf{throw}\ x : \tau} \quad \textit{(thisthrw)}$$

$$\frac{Loc \vdash_{\mathsf{this}} E_1 : \tau_1 \qquad Loc, x : \langle C, \mathsf{nthis} \rangle \vdash_{\mathsf{this}} E_2 : \tau_2}{Loc \vdash_{\mathsf{this}} \mathbf{try}\ \{E_1\}\ \mathbf{catch}\ (C\ x)\ \{E_2\} : \tau_1 \wedge \tau_2} \quad \textit{(thistry)}$$

$$\frac{}{Loc \vdash_{\mathsf{this}} \mathbf{null} : \mathsf{nthis}} \quad \textit{(thisnull)}$$

3.3.3 Reguły cok

Reguły tutaj prezentowane będą wyprowadzały informacje o tym, czy ciało konstruktora jest poprawnie uformowane ze względu na założenia o jego roli (np. w zależności od tego, czy klasa jest funkcyjna). Wyprowadzać będą one osądy postaci:

$$\overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash E : \mathsf{cok}$$

gdzie $\overline{C}$ jest programem, w którym sprawdzamy poprawność wyrażenia E, C jest klasą, w której E jest interpretowane, Ex jest ciągiem nazw klas, w intencji oznaczających wyjątki, jakie mogą zostać wyrzucone z wyrażenia E i wreszcie Loc jest skończoną funkcją częściową z nazw zmiennych do trójek należących do zbioru $\overline{C} \times \mathcal{L} \times \mathcal{T}$ – opisuje ona własności zmiennych lokalnych występujących w E.

$$\frac{\overline{C};\ C : \emptyset, k : \emptyset;\ Ex;\ Loc \vdash \mathbf{new}\ D.k(\overline{x}) : D}{\overline{C};\ C : \emptyset;\ Ex;\ Loc \vdash \mathbf{new}\ D.k(\overline{x}) : \mathsf{cok}} \quad \textit{(emptynew cok)}$$

$$\frac{\overline{C};\ C : \emptyset, k : \emptyset;\ Ex;\ Loc \vdash \mathbf{new}\ D.k(\overline{x}) : D}{\overline{C};\ C : \mathbf{imm};\ Ex;\ Loc \vdash \mathbf{new}\ D.k(\overline{x}) : \mathsf{cok}} \quad \textit{(immnew cok)}$$

$$\frac{\begin{array}{c}\forall y \in \overline{x}.Loc \vdash_{\mathsf{local}} y : \mathsf{local} \\ \overline{C};\ C : \emptyset, k : \emptyset;\ Ex;\ Loc \vdash \mathbf{new}\ D.k(\overline{x}) : D\end{array}}{\overline{C};\ C : \mathbf{func};\ Ex;\ Loc \vdash \mathbf{new}\ D.k(\overline{x}) : \mathsf{cok}} \quad \textit{(emptynew cok)}$$

$$\frac{\begin{array}{c}\overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash E_1 : \mathsf{cok} \\ \overline{C} \vdash \mathsf{subok} \qquad \overline{C} \vdash C_1' \leq: C_1 \\ Loc \vdash_{\mathsf{local}} E_1 : \tau_1 \qquad Loc \vdash_{\mathsf{this}} E_1 : \tau_2 \\ \overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc, x : \langle C_1, \tau_1, \tau_2\rangle \vdash E_2 : \mathsf{cok} \\ \overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc, x : \langle C_1, \tau_1, \tau_2\rangle \vdash \\ \mathbf{let}\ C_1\ x = E_1\ \mathbf{in}\ E_2 : D\end{array}}{\overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash \mathbf{let}\ C_1\ x = E_1\ \mathbf{in}\ E_2 : \mathsf{cok}} \quad \textit{(let cok)}$$

$$\frac{\begin{array}{c}C_1 \in \overline{C} \\ \overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash \mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 : C_1 \\ \overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash E_1 : \mathsf{cok} \\ \overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash E_2 : \mathsf{cok} \\ \overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash E_3 : \mathsf{cok}\end{array}}{\overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash \mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 : \mathsf{cok}} \quad \textit{(if cok)}$$

$$\frac{\begin{array}{c}C_1 \in \overline{C} \qquad x : \langle D, \tau, \mathsf{nthis}\rangle \in Loc \\ x_i : \langle D_i, \tau_i, \mathsf{nthis}\rangle \in Loc \text{ dla } i = 1, \ldots, k \\ \overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash x.m(x_1, \ldots, x_k) : C_1\end{array}}{\overline{C};\ C : \mathbf{func};\ Ex;\ Loc \vdash x.m(x_1, \ldots, x_k) : \mathsf{cok}} \quad \textit{(methfun cok)}$$

$$\frac{\begin{array}{c}C_1 \in \overline{C} \qquad \mathsf{cmod} \neq \mathbf{func} \\ \overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash x.m(x_1, \ldots, x_k) : C_1\end{array}}{\overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash x.m(x_1, \ldots, x_k) : \mathsf{cok}} \quad \textit{(methnfun cok)}$$

$$\frac{\begin{array}{c} C_2\ [\mathsf{fmod}]\ f \in \mathsf{fields}(\overline{C}, C_1) \\ \overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc, x : \langle C_1, \tau_1, \tau_2 \rangle \vdash x.f = y : C_2 \\ \overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc, x : \langle C_1, \tau_1, \tau_2 \rangle \vdash y : \mathsf{cok} \end{array}}{\overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc, x : \langle C_1, \tau_1, \tau_2 \rangle \vdash x.f = y : \mathsf{cok}} \quad \textit{(assign cok)}$$

$$\frac{\overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc, x : \langle C_1, \tau_1, \tau_2 \rangle \vdash x : C_1}{\overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc, x : \langle C_1, \tau_1, \tau_2 \rangle \vdash x : \mathsf{cok}} \quad \textit{(var cok)}$$

$$\frac{\overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash x.y : C_1}{\overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash x.y : \mathsf{cok}} \quad \textit{(field cok)}$$

$$\frac{}{\overline{C};\ C : \mathsf{cmod};\ Ex, C_1;\ Loc, x : \langle C_1, \tau_1, \tau_2 \rangle \vdash \mathbf{throw}\ x : \mathsf{cok}} \quad \textit{(thrw cok)}$$

$$\frac{\begin{array}{c} \overline{C};\ C : \mathsf{cmod};\ Ex, C_1;\ Loc \vdash \mathbf{try}\ \{E_1\}\ \mathbf{catch}\ (C_1\ x)\ \{E_2\} : C_2 \\ \overline{C};\ C : \mathsf{cmod};\ Ex, C_1;\ Loc \vdash E_1 : \mathsf{cok} \\ \overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash E_2 : \mathsf{cok} \end{array}}{\overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash \mathbf{try}\ \{E_1\}\ \mathbf{catch}\ (C_1\ x)\ \{E_2\} : \mathsf{cok}} \quad \textit{(try cok)}$$

$$\frac{}{\overline{C};\ C : \mathsf{cmod};\ Ex;\ Loc \vdash \mathbf{null} : \mathsf{cok}} \quad \textit{(null cok)}$$

3.4 Sprawdzanie poprawności metod

Sprawdzanie poprawności metod wykonywane jest za pomocą reguły:

$$\frac{\begin{array}{c} Ex = \mathsf{throws}(\overline{C}, C, M) \\ Loc = \mathsf{par2loc}(\mathsf{params}(\overline{C}, C, M)) \\ \forall D \in Ex.D \in \overline{C} \\ m = \mathsf{name}(\overline{C}, C, M) \qquad \mathsf{mmod} = \mathsf{modifier}(\overline{C}, C, M) \\ \overline{C}; C : \mathsf{cmod}, m : \mathsf{mmod}; Ex; Loc \vdash \mathsf{body}(\overline{C}, C, M) : \mathsf{mok} \end{array}}{\overline{C}; C : \mathsf{cmod} \vdash M : \mathsf{ok}} \quad \textit{(meth mok ok)}$$

Intuicyjnie mok sprawdza, że wyrażenie jest prawidłowo zbudowane ze względu na oczekiwane własności **func**, **lstate** dla metod oraz ma poprawny typ.

3.4.1 Reguły cok

Reguły wyprowadzające mok będą wyprowadzały informacje o tym, czy ciało metody jest poprawnie uformowane ze względu na założenia o jej roli (np. w zależności od tego, czy klasa lub metoda jest funkcyjna). Wyprowadzać będą one osądy postaci:

$$\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E : \mathsf{mok}$$

gdzie $\overline{C}$ jest programem, w którym sprawdzamy poprawność wyrażenia E, C jest klasą, w której E jest interpretowane, m jest metodą, w której E jest interpretowane, Ex jest ciągiem nazw klas, w intencji oznaczających wyjątki, jakie mogą zostać wyrzucone z wyrażenia E, i wreszcie Loc jest skończoną funkcją częściową z nazw zmiennych do trójek należących do zbioru $\overline{C} \times \mathcal{L} \times \mathcal{T}$ – opisuje ona własności zmiennych lokalnych występujących w E.

Reguły dla mok w metodach funkcyjnych

$$\frac{\begin{array}{c}\mathsf{modifier}(\overline{C}, D, k) = \mathbf{lstate} \\ \forall y \in \overline{x}.Loc \vdash_{\mathsf{local}} y : \mathsf{local} \\ \overline{C};\ C : \mathsf{cmod}, k : \emptyset;\ Ex;\ Loc \vdash \mathbf{new}\ D.k(\overline{x}) : D\end{array}}{\overline{C}; C : \mathsf{cmod}, m : \mathbf{func}; Ex; Loc \vdash \mathbf{new}\ D.k(\overline{x}) : \mathsf{mok}} \quad \textit{(funcnew mok)}$$

$$\frac{\begin{array}{c}C_1, C_2 \in \overline{C} \qquad x : \langle C_1, \tau_1, \tau_2 \rangle \in Loc \\ \mathsf{modifier}(\overline{C}, C_1, m') = \mathbf{func} \\ \overline{C};\ C : \mathsf{cmod}, m : \mathbf{func};\ Ex;\ Loc \vdash x.m'(\overline{x}) : C_2\end{array}}{\overline{C};\ C : \mathsf{cmod}, m : \mathbf{func};\ Ex;\ Loc \vdash x.m'(\overline{x}) : \mathsf{mok}} \quad \textit{(fmethfunc mok)}$$

$$\frac{\begin{array}{c}C_1, C_2 \in \overline{C} \qquad x : \langle C_1, \tau_1, \tau_2 \rangle \in Loc \\ \mathsf{modifier}(\overline{C}, C_1, m') = \mathbf{lstate} \quad \forall y \in \overline{x}.Loc \vdash_{\mathsf{local}} y : \mathsf{local} \\ \overline{C};\ C : \mathsf{cmod}, m : \mathbf{func};\ Ex;\ Loc \vdash x.m'(\overline{x}) : C_2\end{array}}{\overline{C};\ C : \mathsf{cmod}, m : \mathbf{func};\ Ex;\ Loc \vdash x.m'(\overline{x}) : \mathsf{mok}} \quad \textit{(fmethls mok)}$$

Warto zauważyć, że nie mamy reguły typowania przypisania dla metod funkcyjnych. W istocie oznacza to, iż metody funkcyjne nie mogą bezpośrednio wykonywać żadnych przypisań.

Reguły dla mok w metodach local-state

$$\frac{\begin{array}{c}\mathsf{modifier}(\overline{C}, D, k) = \mathbf{lstate} \\ \forall y \in \overline{x}.Loc \vdash_{\mathsf{local}} y : \mathsf{local} \\ \overline{C};\ C : \mathsf{cmod}, k : \emptyset;\ Ex;\ Loc \vdash \mathbf{new}\ D.k(\overline{x}) : D\end{array}}{\overline{C};\ C : \mathsf{cmod}, m : \mathbf{lstate};\ Ex;\ Loc \vdash \mathbf{new}\ D.k(\overline{x}) : \mathsf{mok}} \quad \textit{(lsnew mok)}$$

$$\frac{\begin{array}{c}C_1, C_2 \in \overline{C} \qquad x : \langle C_1, \tau_1, \tau_2\rangle \in Loc \\ \mathsf{modifier}(\overline{C}, C_1, m') = \mathbf{func} \\ \overline{C};\ C : \mathsf{cmod}, m : \mathbf{lstate};\ Ex;\ Loc \vdash x.m'(\overline{x}) : C_2\end{array}}{\overline{C};\ C : \mathsf{cmod}, m : \mathbf{lstate};\ Ex;\ Loc \vdash x.m'(\overline{x}) : \mathsf{mok}} \quad \textit{(lsmeth mok)}$$

$$\frac{\begin{array}{c}C_1, C_2 \in \overline{C} \qquad x : \langle C_1, \tau_1, \tau_2\rangle \in Loc \\ \forall y \in \overline{x}.y : \langle C_y, \mathsf{local}, \tau_2\rangle \in Loc \\ \mathsf{modifier}(\overline{C}, C_1, m') = \mathbf{lstate} \\ \overline{C};\ C : \mathsf{cmod}, m : \mathbf{lstate};\ Ex;\ Loc \vdash x.m'(\overline{x}) : C_2\end{array}}{\overline{C};\ C : \mathsf{cmod}, m : \mathbf{lstate};\ Ex;\ Loc \vdash x.m'(\overline{x}) : \mathsf{mok}} \quad \textit{(lsmeth mok)}$$

$$\frac{\begin{array}{c}C_1, C_1' \in \overline{C} \qquad \overline{C} \vdash C_1' \leq: C_1 \qquad \overline{C} \vdash \mathsf{subok} \\ \mathsf{modifier}(\overline{C}, C_1, f) = \mathbf{rep} \qquad Loc \vdash_{\mathsf{this}} x : \mathbf{this} \\ \overline{C};\ C : \mathsf{cmod}, m : \mathbf{lstate};\ Ex;\ Loc, x : \langle C_1, \tau_1, \tau_2\rangle \vdash \\ y : C_1'\end{array}}{\begin{array}{r}\overline{C}; C : \mathsf{cmod}, m : \mathbf{lstate}; Ex; Loc, x : \langle C_1, \tau_1, \tau_2\rangle \vdash \\ x.f = y : \mathsf{mok}\end{array}} \quad \textit{(lsassign mok)}$$

Reguły dla mok w metodach zwykłych

$$\frac{\begin{array}{c}\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E_1 : C_1' \\ \overline{C} \vdash C_1' \leq: C_1 \qquad \overline{C} \vdash \mathsf{subok} \qquad Loc \vdash_{\mathsf{local}} E_1 : \tau_1 \\ Loc \vdash_{\mathsf{this}} E_1 : \tau_2 \qquad Loc' = Loc, x : \langle C_1, \tau_1, \tau_2\rangle \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E_1 : \mathsf{mok} \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc' \vdash E_2 : C_2 \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc' \vdash E_2 : \mathsf{mok}\end{array}}{\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash \mathbf{let}\ C_1\ x = E_1\ \mathbf{in}\ E_2 : \mathsf{mok}} \quad \textit{(let mok)}$$

$$\frac{\begin{array}{c} C_1, C_2 \in \overline{C} \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E_1 : C_1 \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E_2 : C_2 \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E_3 : C_2 \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E_1 : \mathsf{mok} \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E_2 : \mathsf{mok} \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E_3 : \mathsf{mok} \end{array}}{\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash \mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 : \mathsf{mok}} \quad \textit{(ifmok)}$$

$$\frac{\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc, x : \langle C_1, \tau_1, \tau_2 \rangle \vdash x : C_1}{\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc, x : \langle C_1, \tau_1, \tau_2 \rangle \vdash x : \mathsf{mok}} \quad \textit{(var mok)}$$

$$\frac{\begin{array}{c} C_1 \in \overline{C} \qquad C_2\ [\mathsf{fmod}]\ y \in \mathsf{fields}(\overline{C}, C_1) \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc, x : \langle C_1, \tau_1, \tau_2 \rangle \vdash x.y : C_2 \end{array}}{\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc, x : \langle C_1, \tau_1, \tau_2 \rangle \vdash x.y : \mathsf{mok}} \quad \textit{(field mok)}$$

$$\frac{\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc, x : \langle C_1, \tau_1, \tau_2 \rangle \vdash x : C_1}{\begin{array}{l} \overline{C}; C : \mathsf{cmod}, m : \mathsf{mmod}; Ex, C_1; Loc, x : \langle C_1, \tau_1, \tau_2 \rangle \vdash \\ \qquad\qquad \mathbf{throw}\ x : \mathsf{mok} \end{array}} \quad \textit{(thrw mok)}$$

$$\frac{\begin{array}{c} \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex, C_1;\ Loc \vdash E_1 : C_2 \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E_2 : C_2 \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex, C_1;\ Loc \vdash E_1 : \mathsf{mok} \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E_2 : \mathsf{mok} \end{array}}{\begin{array}{l} \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash \\ \qquad\qquad \mathbf{try}\{E_1\}\ \mathbf{catch}\ (C_1\ x)\ \{E_2\} : \mathsf{mok} \end{array}} \quad \textit{(try mok)}$$

$$\frac{}{\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash \mathbf{null} : \mathsf{mok}} \quad \textit{(null mok)}$$

3.5 Przypisywanie klas wyrażeniom

Reguły przypisujące klasy wyrażeniom będą wyprowadzały informacje o tym, jakiej klasy jest wyrażenie. Wyprowadzać będą one osądy postaci:

$$\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E : D$$

gdzie $\overline{C}$ jest programem, w którym sprawdzamy poprawność wyrażenia E, C jest klasą, w której E jest interpretowane, m jest metodą, w której E jest interpretowane, Ex jest ciągiem nazw klas, w intencji oznaczających wyjątki, jakie mogą zostać wyrzucone z wyrażenia E, Loc jest skończoną funkcją częściową z nazw zmiennych do trójek należących do zbioru $\overline{C} \times \mathcal{L} \times \mathcal{T}$ – opisuje ona własności zmiennych lokalnych występujących w E, oraz wreszcie D jest klasą wyrażenia D.

Użyty wyżej identyfikator m może oznaczać tak metodę, jak i konstruktor. Nie wprowadzamy dla poniższych reguł rozróżnienia między tymi kategoriami, gdyż reguły typowania dla obu z nich są takie same.

$$\frac{\begin{array}{c}\mathsf{throws}(\overline{C}, C', k) \subseteq Ex \\ \mathsf{paramsNo}(\overline{C}, D, k) = n \qquad C' \in \overline{C} \\ \overline{C}, C : \mathsf{cmod}, m : \mathsf{mmod}; Ex; Loc \vdash x_i : C_i \ \text{ for } i = 1, \ldots, n \\ \overline{C} \vdash \mathsf{subok} \quad C_i \leq: \mathsf{paramType}(\overline{C}, C', k, i) \ \text{ for } i = 1, \ldots, n\end{array}}{\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash \mathbf{new}\ C'.k(x_1, \ldots, x_n) : C'} \quad \textit{(typ new)}$$

$$\frac{\begin{array}{c}\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc, x : \langle C'', \mathsf{nlocal}, \mathsf{nthis}\rangle \vdash \\ E_2 : C' \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E_1 : C''\end{array}}{\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash \mathbf{let}\ C''\ x = E_1\ \mathbf{in}\ E_2 : C'} \quad \textit{(typ let)}$$

$$\frac{\begin{array}{c}\overline{C} \vdash \mathsf{subok} \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E_1 : C_1' \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E_2 : C_2' \qquad C_2' \leq: C' \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash E_3 : C_3' \qquad C_3' \leq: C'\end{array}}{\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash \mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 : C'} \quad \textit{(typ if)}$$

$$\frac{\begin{array}{c}\mathsf{throws}(\overline{C}, C', m') \subseteq Ex \qquad \mathsf{returnType}(\overline{C}, C, m') = C' \\ \mathsf{paramsNo}(\overline{C}, D, m') = n \qquad C' \in \overline{C} \\ \overline{C}, C : \mathsf{cmod}, m : \mathsf{mmod}; Ex; Loc \vdash x_i : C_i \ \text{ for } i = 1, \ldots, n \\ \overline{C} \vdash \mathsf{subok} \qquad C_i \leq: \mathsf{paramType}(\overline{C}, D, m', i) \ \text{ for } i = 1, \ldots, n\end{array}}{\begin{array}{l}\overline{C}; C : \mathsf{cmod}, m : \mathsf{mmod}; Ex; Loc, x : \langle D, \tau_1, \tau_2\rangle \vdash \\ \qquad x.m'(x_1, \ldots, x_n) : C'\end{array}} \quad \textit{(typ meth)}$$

$$\frac{\begin{array}{c}\overline{C} \vdash \mathsf{subok} \\ C \leq: D \qquad C'f \in \mathsf{fields}(\overline{C}, D) \qquad C'' \leq: C' \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc, x : \langle D, \tau_1, \tau_2\rangle \vdash E : C''\end{array}}{\begin{array}{l}\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc, x : \langle D, \tau_1, \tau_2\rangle \vdash \\ \qquad x.f = E : C'\end{array}} \quad \textit{(typ assign)}$$

$$\frac{\overline{C} \vdash \mathsf{subok} \qquad C \leq: D \qquad C' f \in \mathsf{fields}(\overline{C}, D)}{\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc, x : \langle D, \tau_1, \tau_2 \rangle \vdash x.f : C'} \quad \textit{(typ varref)}$$

$$\frac{\overline{C} \vdash \mathsf{subok} \qquad D' \leq: D \qquad D, D', C' \in \overline{C}}{\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex, D;\ Loc, x : \langle D', \tau_1, \tau_2 \rangle \vdash \mathbf{throw}\ x : C'} \quad \textit{(typ thrw)}$$

$$\frac{\begin{array}{c}\overline{C} \vdash \mathsf{subok} \\ C', C_1, C_1', C_2' \in \overline{C} \qquad C_1' \leq: C' \qquad C_2' \leq: C' \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex, C_1;\ Loc \vdash E_1 : C_1' \\ \overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc, x : \langle C_1, \tau_1, \tau_2 \rangle \vdash E_2 : C_2'\end{array}}{\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash \mathbf{try}\ \{E_1\}\ \mathbf{catch}\ (C_1\ x)\ \{E_2\} : C'} \quad \textit{(typ try)}$$

$$\frac{C' \in \overline{C}}{\overline{C};\ C : \mathsf{cmod}, m : \mathsf{mmod};\ Ex;\ Loc \vdash \mathbf{null} : C'} \quad \textit{(typ null)}$$

Rozdział 4

Studium możliwości realizacji prototypu kompilatora języka *Jafun*

W niniejszym rozdziale przedstawimy wyniki badań możliwości realizacji kompilatora języka *Jafun*. Analiza będzie polegała na kompleksowej analizie problemu stworzenia kompilatora. Na początku przedstawimy założenia dotyczące zakresu funkcjonalności kompilatora. Następnie przedstawimy standardowy sposób realizacji kompilatorów języków programowania. Kolejnym elementem będzie przeanalizowanie funkcjonalności istniejących narzędzi do budowy kompilatorów w kontekście języka *Jafun* z uwzględnieniem sposobów ich użycia. W głównym rozdziale zaprezentujemy szczegółowy plan realizacji kompilatora w oparciu o składnię i semantykę języka *Jafun* przedstawioną w poprzednich częściach niniejszego dokumentu. W końcu podsumujemy wyniki badań i przedstawimy rekomendację do kolejnych etapów prac.

4.1 Założenia początkowe

Założona funkcjonalność kompilatora będzie obejmować zarówno weryfikację składni języka jak i weryfikację semantyki. Wynikiem działania kompilatora będzie bytecode dla pewnej maszyny uruchomieniowej [LY99]. Jednak w ramach niniejszej analizy nie będziemy rozważać funkcjonalności tej maszyny wirtualnej. Zakładamy, że funkcjonalność maszyny oraz implementacja będą przedmiotem prac w kolejnych etapach.

Podstawą budowy analizatora składniowego będzie gramatyka bezkontekstowa, która została przedstawiona we wcześniejszych rozdziałach. Przy budowie analizatora składniowego konieczne będzie zapewnienie odpowied-

niej regularności gramatyki w celu bezproblemowego zbudowania parsera. Nie były prowadzone testy regularności przedstawionej gramatyki, jednak na podstawie bardzo dużego doświadczenia z różnorakimi gramatykami, nie będzie żadnych problemów z taką modyfikacją gramatyki, aby otrzymać równoważną gramatykę, z punktu widzenia semantyki, ale klasy LALR.

4.2 Standardowe podejście budowy kompilatorów

Kompilator zostanie zbudowany w standardowy sposób z podziałem na dwie fazy. Pierwsza faza to faza analizy, w której program źródłowy jest przetwarzany w celu stworzenia wewnętrznej reprezentacji programu. W tej fazie przeprowadzana jest także pełna weryfikacji poprawności programu źródłowego. Druga faza to faza syntezy polegająca na przetworzeniu wewnętrznej reprezentacji programu do postaci wynikowej będącej programem w postaci zrozumiałej przez maszynę uruchomieniową.

Faza analizy programu źródłowego dzieli się na dwa zasadnicze etapy. Pierwszy etap to analiza składniowa programu źródłowego. Druga faza to analiza semantyki programu. W fazie pierwszej następuje wczytanie programu z plików źródłowych i zostają one rozłożone na podstawie definicji gramatyki języka. W tej fazie badana jest zgodność programu z definicją gramatyki w sensie zapisu nie zaś w sensie znaczenia zapisu. Proces analizy składniowej programu jest wspierany przez wiele dostępnych narzędzi. Kluczowym elementem analizy jest wydajność analizy programu. Teoria gramatyk bezkontekstowy definiuje klasy języków, dla których znane są wydajne analizatory składni. Dlatego istotne jest, aby przed stworzeniem analizatora przekształcić gramatykę w taki sposób, aby istniał dla niej wydajny analizator. W chwili bieżącej wydaje się, że klasa gramatyk LALR daje rozsądny kompromis pomiędzy wydajnością analizatora a elastycznością konstrukcji gramatyki.

Poniżej przedstawiamy propozycję składni języka *Jafun*, która jest przyjmowana bez błędów przez tokenizer Flex oraz analizator składniowy Bison. Przykład ten pokazuje, że bez problemu zaproponowana w poprzednich rozdziałach składnia może zostać przekształcona na potrzeby efektywnego parsowania języka *Jafun*.

Plik definicji tokenów dla programu Flex.

```
%{
#include "kk.tab.h"
```

```
%}

%%

[0-9]+  return NUMBER;

class   return CLASS;
immutable  return IMMUTABLE;
functional return FUNCTIONAL;
local_state return LOCALSTATE;
rep     return REP;
ext     return EXT;
throws  return THROWS;
new     return NEW;
let     return LET;
in      return IN;
if      return IF;
then    return THEN;
else    return ELSE;
throw   return THROW;
try     return TRY;
catch   return CATCH;
null    return TNULL;
var     return VAR;
this    return THIS;
constructor return CONSTRUCTOR;
;       return SEMICOL;
,       return COLON;
\.      return DOT;
\(      return LPPAR;
\)      return RPPAR;
=       return EQ;
\{                      return LPAR;
\}                      return RPAR;

[a-zA-Z][a-zA-Z0-9]* return IDENT;

%%
```

Plik opisu gramatyki języka *Jafun* dla analizatora składni Bison.

```
%{
#include <stdio.h>
```

```
#include <string.h>

void yyerror(const char *str)
{
fprintf(stderr,"error: %s\n",str);
}

int yywrap()
{
return 1;
}

main()
{
yyparse();
}

%}

%token NUMBER CLASS IMMUTABLE FUNCTIONAL LOCALSTATE REP EXT
       THROWS NEW LET IN IF THEN ELSE THROW TRY CATCH TNULL
       IDENT SEMICOL LPAR RPAR COLON LPPAR RPPAR CONSTRUCTOR
       VAR THIS DOT EQ


%%

c_:     cclasses
cclasses: /* empty */ | cclasses cclass

cmodifier: /* empty */ | IMMUTABLE | FUNCTIONAL

ext: /* empty */ | EXT IDENT

fmodifier: /* empty */ | REP

field: VAR IDENT fmodifier IDENT SEMICOL

fields: /* empty */ | fields field

paramdef: IDENT IDENT

paramdeflist: paramdef | paramdeflist COLON paramdef
```

```
paramdefs: LPPAR RPPAR | LPPAR paramdeflist RPPAR

kmodifier: /* empty */ | LOCALSTATE

paramlist: expr | paramlist COLON expr

params: LPPAR RPPAR | LPPAR paramlist RPPAR

fieldref: THIS DOT IDENT
        | IDENT DOT IDENT
        | THIS DOT fieldref
        | IDENT DOT fieldref

varref: THIS | IDENT | fieldref

expr: NEW IDENT DOT IDENT params
    | IF expr THEN expr ELSE expr
    | IDENT DOT IDENT params
    | TNULL
    | fieldref EQ expr
    | LET IDENT IDENT EQ expr IN expr
    | THROW expr
    | TRY LPAR expr RPAR CATCH LPPAR IDENT IDENT RPPAR
      LPAR expr RPAR
    | NUMBER
    | varref

identlist: IDENT | identlist COLON IDENT

throwdefs: /* empty */ | THROWS identlist

constr: kmodifier CONSTRUCTOR DOT IDENT paramdefs throwdefs
        LPAR expr RPAR

constrs: /* empty */ | constrs constr

mmodifier: /* empty */ | LOCALSTATE | FUNCTIONAL

methods: IDENT mmodifier IDENT paramdefs throwdefs LPAR expr RPAR

cclass: CLASS cmodifier IDENT ext
        LPAR fields constrs methods RPAR SEMICOL
```

W czasie analizy składni programu tworzone jest drzewo składni programu. Drzewo składni inaczej AST jest podstawą dalszej analizy programu w fazie analizy semantycznej programu. Faza ta polega w dużej mierze na iterowaniu po strukturze AST i dodawaniu informacji w węzłach AST. Informacje zgromadzone w czasie tej fazy muszą być wystarczające, aby po pierwsze zweryfikować spełnianie w programie reguł semantycznych wypisanych w poprzednim rozdziale oraz po drugie muszą wystarczyć do wygenerowania kodu do wykonania programu.

Zajmiemy się teraz dwoma aspektami budowy analizatora semantyki. Po pierwsze można spojrzeć na język *Jafun* jako na pewien podzbiór standardowych języków obiektowych takich jak C++ czy Java rozszerzony o nowe konstrukcje. Dlatego konstrukcję fazy analizy semantycznej języka będziemy dzielić na elementy standardowe dla języka Java oraz na rozszerzenia języka Java. Po drugie fakt istnienia modyfikatorów z jednej strony wymusza konieczność weryfikacji spełniania przez program reguł semantycznych, ale z drugiej strony dodane modyfikatory nie wpływają na fazę generowania kodu. Dlatego drzewo składni zostanie w fazie analizy składni wzbogacone o elementy konieczne do weryfikacji reguł semantycznych, ale z drugiej strony faza generowania kodu wynikowego jest niezależna od wprowadzonych rozszerzeń języka.

Na koniec dochodzimy do konkluzji, że faza generowania kodu dla języka *Jafun* jest analogiczna do fazy generacji kodu w języku Java. Na ten temat istnieje wiele publikacji, które w bardzo precyzyjny sposób podają metody generacji kodu dla języka Java. Konkretne odniesienia do wzorcowych sposobów realizacji głównych aspektów języka Java zostaną przedstawione przy okazji opisu planu realizacji języka *Jafun*. Z drugiej strony, dostępne są kody źródłowe gotowych i w pełni działających interpreterów i kompilatorów języka Java. Więcej na temat narzędzi, które potencjalnie można wykorzystać w tej fazie w kolejnym rozdziale.

4.3 Narzędzia analizy składni i semantyki języków programowania

W tym rozdziale przedstawiona zostanie analiza istniejących narzędzi do budowy kompilatorów w kontekście przydatności do budowy kompilatora języka *Jafun*. Analiza zostanie podzielona na dwie zasadnicze części. W pierwszej części rozważony zostanie scenariusz polegający na stworzeniu od podstaw całego kompilatora języka *Jafun*, czyli zostaną przedstawione narzędzia,

które umożliwiają budowę pełnych kompilatorów języków programowania. W drugiej części rozdziału zostaną przedstawione istniejące pełne narzędzia do kompilacji pewnych języków programowania, które dają nadzieję na łatwe przystosowanie do potrzeb języka *Jafun*.

4.3.1 Podejście standardowe - budowa pełnego kompilatora

Analizę standardowych narzędzi rozpoczyna para analizatora tokenów Lex oraz analizatora składniowego Yacc. Ogólna idea tej pary narzędzi polega na zbudowaniu automatu, który akceptowałby wpisaną gramatykę oraz pozwalałby programiście na wykonywanie dowolnego kodu w momencie wykrycia każdej z konstrukcji zadeklarowanych w pliku opisu gramatyki. Wynikiem działania pary Lex i Yacc jest kod źródłowy programu parsującego zadaną gramatykę. Historycznie nazwy Lex i Yacc wywodzą się od pierwszej implementacji tych narzędzi, która wspierała język C, jako język do którego generowany jest kod analizatora.

Od czasów powstania oryginalnego Lexa i Yacca powstało wiele klonów tych programów dla wielu języków (chodzi tu o języki, w których powstaje kompilator a nie języki które są kompilowane).

Lista istniejących obecnie klonów dla różnych języków programowania.

Flex i Bison są to napisane w ramach projektu GNU klonu dla języków C i C++. Charakteryzuje je duża zgodność z oryginalnymi programami oraz duża dojrzałość projektów, które są rozwijane od prawie 8 lat. Do zalet tej pary programów należą, przyjmowanie specyfikacji gramatyki w postaci bardzo zbliżonej do BNF, rozpowszechniona wiedza programistów na temat tych dwóch programów oraz liczne publikacje na temat rozwiązań zastosowanych w tych programach ([LMB92, ASU86, AG98]). Bison standardowo akceptuje gramatyki LALR(1). Przykładowe pliku z propozycją gramatyki z poprzedniego rozdziału są w formacie akceptowanych przez Flexa i Bisona. Oznacza to, że gramatyka języka *Jafun* może zostać w bardzo łatwy sposób dostosowana tak aby była LALR(1). Dodatkowo Bison wspiera także języka Java, ale w kontekście innych klonów dedykowanych dla tego języka nie jest to polecany sposób użycia. Dokładne informacje na temat projektu znajdują się pod adresem `http://www.gnu.org/software/bison/` [DS06].

JFlex i JavaCup są klonami dla języka Java. Ogólna idea działania jest dokładnie taka sama, jak w przypadku pierwowzorów. Istnieją różnice

na poziomie składni gramatyki dla analizatora składni, jednak składnia dla Cupa jest również oparta na notacji BNF więc przekształcenie zapisu gramatyki do postaci użytecznej dla Cupa nie będzie problemem. Więcej na temat narzędzi na stronach domowych projektu Flex ([Kle08]) oraz JavaCUP ([Hud06]).

GPPG oraz GPLEX są klonami przeznaczonymi dla języka C#. Analizator radzi sobie, podobnie do pierwowzoru, z gramatykami LALR(1). Oprócz oczywistych różnic wynikających z innego języka docelowego są także różnice w zapisie tokenów oraz gramatyki. Szczegółowe informacje na temat sposobów użycia narzędzi znajdują się na stronach domowych projektu ([GK07]).

Inne języki Oto krótka lista przykładów rozwiązań dla innych języków programowania. Lista jest niepełna. Wynika z niej tylko tyle, że z bardzo dużym prawdopodobieństwem dla dowolnego wybranego języka istnieją odpowiednie klony Lexa i Yacca przeznaczone dla tego języka. TP Lex/Yacc, dla języka Pascal (Turbo Pascal, Delphi, FPC) ([Gra]), dla języka Python ([Bea]), dla języka Erlang ([yec]), dla języka List ([Chr]), dla języka SML ([TA00]) oraz dla języka Haskell ([MG01]).

W powyższym zestawieniu opisano jedynie główne narzędzia. Budowa kompilatorów jest generalnie tematem bardzo mocno eksploatowanym od lat więc istnieje bardzo wiele różnorodnych narzędzi wspierających proces tworzenia kompilatorów.

4.3.2 Podejście uproszczone - modyfikacja istniejącego kompilatora

Jeśli celem ma być możliwie proste stworzenie kompilatora języka *Jafun*, to rozsądnym rozwiązaniem może być próba modyfikacji pewnego istniejącego kompilatora języka Java. Przy ocenie wielkości języka Java i języka *Jafun* nietrudno stwierdzić, że wielkość języka *Jafun* w stosunku do Javy jest bardzo mała. Może być to wadą oraz zaletą. Oczywista zaleta polega na tym, że przypuszczalna modyfikacja kompilatora języka Java, która sprawdza nowe reguły semantyczne wprowadzone w *Jafun* będzie bardzo mała. To daje nadzieję, że przez mała modyfikację istniejącego kompilatora można otrzymać z jednej strony w pełni działający kompilator języka Java (lub bardzo zbliżonego do Javy) a z drugiej strony języka, który posiada nowe konstrukcje wprowadzone przez *Jafun*. Jednak te same cechy mogą być też wadami, ponieważ rozległość oryginalnej Javy może doprowadzić do dużej kompilacji

przy weryfikacji reguł semantycznych w *Jafun*. Ryzyko dużej czasochłonności może być nieuzasadnione jeśli rzeczywistym celem nie ma być stworzenie prawdziwego kompilatora, ale jedynie proof-of-concept.

Naturalnymi kandydatami wśród dostępnych kompilatorów Javy są:

Jikes jest otwartym kompilatorem, który był przez długi czas rozwijany przez firmę IBM i został udostępniony na otwartej licencji. Jikes jest całkowicie napisany w C/C++. Kod źródłowy kompilatora jest do pobrania ze strony domowej projektu ([jik]).

Kaffe jest kompilatorem Javy wywodzącym się z KJC. Jest całkowicie napisanym w Javie i udostępnionym na licencji GPL. Opis kompilatora, dokumentacja oraz kod źródłowy projektu znajduje się na stronie domowej ([kaf]).

Na koniec rozważań na temat gotowych rozwiązań przeanalizowane będzie zagadnienie wykorzystania adnotacji do realizacji kompilatora *Jafun*. Jednym ze standardowych podejść do modyfikacji istniejących języków programowania, które istnieje praktycznie od początku istnienia języków jest wykorzystanie mechanizmu komentarzy do wzbogacania funkcjonalności, które oferuje język. Niewątpliwą zaletą tego rozwiązania jest możliwość wykorzystania gotowego w pełni działającego kompilatora języka do kompilacji rozszerzonego języka, ponieważ zawartość komentarzy jest dla kompilatora zupełnie przezroczysta.

Dla języka Java istnieje wiele implementacji tejże idei. Począwszy od dobrych praktyk polegających na dodawaniu komentarzy w kodzie, chociażby to to, aby objaśniać działania kodu i oczywiście nie posiadających żadnych narzędzi do weryfikacji, poprzez różnorodne tagi, np. javadoc, mogące służyć do generacji dokumentacji, a skończywszy na pełnych językach adnotacji, które służą np. do weryfikacji pewnych aspektów kodu, jak JML ([JML06]).

Wykorzystanie adnotacji jest tym bardziej wskazane dla języka *Jafun*, ponieważ jak wynika z analizy z poprzednich rozdziałów, rozszerzenia języka *Jafun* nie wpływają na oryginalne działanie kodu. Czyli przez wprowadzenie konstrukcji *Jafun* pewien kod źródłowy może stać się niepoprawny, ale jeśli będzie poprawny, to jego sposób działania wewnątrz *Jafun* nie ulegnie zmianie. To sugeruje możliwość rozwiązania polegającego na dodaniu pewnego zbioru adnotacji do języka Java opisującego konstrukcje języka *Jafun*, które to adnotacje zostaną sprawdzone w momencie weryfikacji poprawności, ale sam proces kompilacji kodu wynikowego zostanie niezmieniony.

Potencjalnymi narzędziami do wykorzystania są narzędzia operujące na języku JML. Przypuszczalnie najprostsze wprowadzenie weryfikacji reguł języka *Jafun* polegałoby na wykorzystaniu narzędzia Esc/Java ([CK05]) do

statycznego sprawdzania reguł JML-owych. Analiza tego podejścia przeprowadzona w poprzednich etapach badań pozwalała na stwierdzenie, że tego typu podejście może być bardzo efektywne w przypadku tworzenia pierwszej wersji działającego weryfikatora.

Innym narzędziem możliwym do wykorzystania jest mechanizm adnotacji wbudowany w standard Javy od wersji 1.5 (5.0).

4.4 Plan realizacji kompilatora języka *Jafun*

W niniejszej analizie przedstawiony zostanie zalecany sposób postępowania przy budowie kompilatora języka *Jafun*.

Na wstępnie należy zauważyć, że pewne elementy występujące w języku *Jafun* są analogiczne do innych obiektowych języków programowania. Elementami tymi są:

- typy danych
- klasy
- dostęp do pól
- wywołania metod
- podział na pola, konstruktory oraz metody klas
- dziedziczenie klas
- automatyczne zarządzanie pamięcią (ang. garbage collector)
- wyjątki

Zatem realizacja tych elementów powinna być zrealizowana w sposób analogiczny dla innych języków obiektowych.

4.4.1 Realizacja standardowych aspektów języków obiektowych

Podstawowymi elementami języków obiektowych są klasy. Ich realizacja w języku *Jafun* jest analogiczna jak w Javie. Co więcej, w porównaniu z innymi językami programowania takimi jak np. C++ język *Jafun* jest znacznie prostszy w aspekcie klas, ponieważ nie posiada wielodziedziczenia implementacji. Takie ograniczenie znacznie upraszcza realizację kompilatora. Obecnie

temat realizacji klas wewnątrz języków obiektowych został już dokładnie zbadany i istnieje wiele publikacji na ten temat. Konkretną publikacją, w której omówione są szczegółowo mechanizmy realizacji klas jest [Lip96].

Drugim aspektem, który powinien być zrealizowany przy konstrukcji języka *Jafun* jest zarządzanie pamięcią. Założeniem podstawowym jest realizacja automatycznego odśmiecania pamięci. W przeciwieństwie do implementacji obiektów, realizacja zarządzania pamięcią jest tematem, który jest ciągle tematem badań. Opisy różnych implementacji dla języka Java można znaleźć np. w następujących publikacjach: [BFG02, BR01, Nil96, NS98].

Na koniec omówienia zastosowanych technik zwróćmy uwagę, że przy tworzeniu pierwszej implementacji kompilatora pewne współcześnie analizowane aspekty konstrukcji kompilatorów są zupełnie nieistotne. Obecnie tematami, które są bardzo rozwijane, są zagadnienia JIT, czyli kompilacji fragmentów bytecodu do kodu maszynowego już w czasie wykonania programu, lub zagadnienia zaawansowanego odśmiecania pamięci. To że nowe maszyny wirtualne Javy potrafią adaptować zarządzanie pamięcią do charakteru programu, który jest wykonywany, nie znaczy, iż takie działanie jest niezbędne w pierwszej wersji kompilatora języka *Jafun*. W pierwszej wersji z całą pewnością wystarczy pewna standardowa, działająca implementacja wszystkich mechanizmów.

4.4.2 Realizacja weryfikacji specyficznych modyfikatorów języka *Jafun*

W tym rozdziale rozważany będzie sposób realizacji weryfikacji specyficznych modyfikatorów dla języka *Jafun*. Przedstawiona będzie realizacji weryfikacji modyfikatorów:

- **lstate** (dla konstruktora)
- **lstate** (dla metody)
- **func** (dla metody)
- **func** (dla klasy)
- **imm**

Podstawowym narzędziem, które umożliwi weryfikację jest drzewo składni programu. Zatem weryfikacja powinna być przeprowadzona po (lub w trakcie) zbudowaniu drzewa składni programu.

Sprawdzenie konstruktorów będzie bazować na regułach z rozdziału 3. Analiza reguł tego rozdziału pokazuje, że weryfikacja spełnialności modyfikatorów może być zrealizowana przez lokalną analizę kodu i deklaracji konstruktora. Weryfikacja podstawowej reguły dla konstruktora polega na prostych sprawdzeniach poprawności deklaracji klasy oraz na weryfikacji ciała konstruktora. Weryfikacja ciała konstruktora składa się z dwóch etapów. Pierwszym etapem jest sprawdzenia poprawności obsługi odwołania **this**, zaś drugim etapem jest sprawdzenie lokalności odwołań. W części pierwszej sprawdzenie polega na przejściu przez całe ciało konstruktora i sprawdzenie czy **this** nie jest przekazywany do nieakceptowanych metod w sposób jawny lub niejawny.

Weryfikacja drugiego etapu polega na strukturalnej weryfikacji przez kolejne konstrukcje programu. Widać, że weryfikacja tej fazy może być zrealizowana za pomocą standardowej analizy zstępującej po strukturze AST. W ten sposób weryfikacja modyfikatorów odbywa się de facto od najniższych węzłów drzewa aż do korzenia.

Po tej analizie można domniemywać, że analogiczny sposób weryfikacji można wykorzystać także przy modyfikatorach dla metod oraz dla klas.

Jest tak w istocie ze względu na komplementarność modyfikatorów języka *Jafun*. Przykładowo reguły inwokacji metod przy weryfikacji modyfikatora **func** dla metod korzysta z zadeklarowania własności **lstate** dla wywoływanej metody. Taka własność umożliwia weryfikację wszystkich modyfikatorów przy jednym przejściu drzewa składni programu.

4.5 Podsumowanie. Rekomendowany sposób realizacji kompilatora

Podsumowując przeprowadzone badania stwierdzamy, że można podzielić zadanie budowy kompilatora na dwa odrębne cele. Cel pierwszy to stworzenie proof-of-concept, cel drugi to napisanie pełnego kompilatora.

W przypadku decyzji o realizacji celu pierwszego, czyli przy chęci szybkiego stworzenia pewnej działającej realizacji najważniejszych aspektów języka *Jafun*, zależałoby wykorzystać możliwie wiele istniejących narzędzi do kompilacji języka Java. Dlatego w tym przypadku najlepszym rozwiązaniem wydaje się wykorzystanie języka JML do opisu nowych właściwości języka *Jafun*, następnie wykorzystania narzędzi Esc/Java do weryfikacji nowych reguł oraz standardowego kompilatora języka Java do kompilacji programów. To rozwiązanie wydaje się najlepsze, ponieważ jest w pełni zgodne z ideą dodania do języka Java małych elementów do statycznej weryfikacji bez ko-

nieczności zmiany procesu kompilacji. Wcześniejsze analizy tego problemu sugerują, że takie rozwiązanie jest możliwe.

W przypadku realizacji pełnego kompilatora języka *Jafun* na podstawie semantyki przedstawionej w pierwszej części tego dokumentu wydaje się, że najlepszym rozwiązaniem jest stworzenie pełnego kompilatora za pomocą klonów Lexa i Yacca. Takie rozwiązanie jest najlepsze ze względu na ściśle określoną postać języka *Jafun*, która abstrahuje od wielu w istocie najbardziej czasochłonnych aspektów budowy kompilatora. Przedstawiona postać języka *Jafun* jest w istocie minimalnym językiem programowania, który pozwala z jednej strony zdefiniować istotne nowe idee języka *Jafun* a z drugiej strony dowodzi, że da się język *Jafun* rozbudować do pełnego języka zbliżonego do Javy. Z tego powodu w kompilatorze języka *Jafun* nie będzie konieczne, ani implementowanie skomplikowanych wyrażeń, ani jakakolwiek optymalizacja. Z tego punktu widzenia najlepszym rozwiązanie jest implementacja pełnego kompilatora w oparciu o JFlexa i JavaCupa.

Bibliografia

[AG98] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Press Syndicate of the University of Cambridge, 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 1986.

[BE04] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA'04*, pages 35–49, October 26–28, 2004.

[Bea] David Beazley. PLY (Python Lex-Yacc). Dostępny z `http://www.dabeaz.com/ply/`.

[Bel04] Abhijit Belapurkar. Functional programming in the Java language. *IBM DevelopersWork*, 2004.

[BFG02] David F. Bacon, Stephen J. Fink, and David Grove. Space- and time-efficient implementation of the java object model. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 111–132, London, UK, 2002. Springer-Verlag.

[BR01] William S. Beebee and Martin C. Rinard. An implementation of scoped memory for real-time java. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 289–305, London, UK, 2001. Springer-Verlag.

[Chr] Juliusz Chroboczek. *The CL-Yacc Manual*. Dostępne z `http://www.pps.jussieu.fr/~jch/software/cl-yacc/`.

[CK05] D. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using esc/java2 and a

report on a case study involving the use of esc/java2 to verify portions of an internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, volume 3362 of *LNCS*, pages 108–128. Springer, 2005. Narzędzie dostępne z `http://kind.ucd.ie/products/opensource/ESCJava2/`.

[Dao] Frederic Daoud. FunctionalJ – A library for functional programming in Java. Available from `http://functionalj.sourceforge.net/`.

[Dek06] Anthony H. Dekker. Lazy functional programming in Java. *SIGPLAN Not.*, 41(3):30–39, 2006.

[DS06] Charles Donnelly and Richard Stallman. *Bison*. Free Software Foundation, Inc., May 2006. Dostępne z `http://www.gnu.org/software/bison/`.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GK07] John Gough and Wayne Kelly. *The GPPG Parser Generator*. Queensland University of Technology, November 2007.

[Gra] Albert Graef. Turbo pascal lex/yacc. Dostępne z `http://www.musikwissenschaft.uni-mainz.de/~ag/tply/tply.html`.

[HPSS07] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. In R. De Nicola, editor, *European Symposium on Programming*, volume 4421 of *LNCS*, pages 347–362. Springer-Verlag, 2007.

[Hud06] Scott E. Hudson. *CUP User's Manual*. Graphics Visualization and Usability Center, Georgia Institute of Technology, March 2006. Dostępne z `http://www2.cs.tum.edu/projects/cup/`.

[jik] Kompilator Javy Jikes. Dostępny z `http://jikes.sourceforge.net/`.

[JML06] *JML Reference Manual (Draft)*. `http://www.cs.iastate.edu/~leavens/JML/jmlrefman`, Date retrieved: August 1, 2006.

[kaf] Wirtualna maszyna Javy Kaffe. Dostępna z `http://www.kaffe.org/`.

[Kle08] Gerwin Klein. *JFlex User's Manual*, June 2008. Dostępne z `http://www.jflex.de/`.

[Lip96] Stanley B. Lippman. *Inside the C++ object model*. Addison Wesley Longman Publishing Co., Inc., 1996.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc (2nd ed.)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1992.

[LY99] Tim Lindholm and Frank Yellin. *The Java (TM) Virtual Machine Specification (Second Edition)*. Prentice Hall, 1999.

[MG01] Simon Marlow and Andy Gill. *Happy User Guide*, 2001. Dostępne z `http://www.haskell.org/happy/`.

[MM03] Jishnu Mukerji and Joaquin Miller. Overview and guide to OMG's architecture. Technical report, Object Management Group, 2003. available from http://www.omg.org/docs/omg/03-06-01.pdf.

[Nau03] David R. Naugler. Functional programming in Java. *J. Comput. Small Coll.*, 18(6):112–118, 2003.

[Nil96] Kelvin Nilsen. Issues in the design and implementation of real-time java. *Java Developer's Journal*, (1:44), June 1996.

[NS98] Gor Nishanov and Sibylle Schupp. Design and implementation of the fgc garbage collector. Technical report, Rensselaer Polytechnic Institute, NY, 1998. Technical Report 98-7.

[Set03] Anton Setzer. Java as a functional programming language. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs: International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002. Selected Papers.*, pages 279 – 298. LNCS 2646, 2003.

[SR05] Alexandru Salcianu and Martin Rinard. Purity and side effect analysis for java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation*, Paris, France, January 2005.

[TA00] David R. Tarditi and Andrew W. Appel. *ML-Yacc User's Manual*, 2000. `http://www.smlnj.org/doc/ML-Yacc/index.html`.

[yec] Parsing with yecc. Część instrukcji FAQ do języka Erlang. `http://www.erlang.org/faq/parsing.html`.