Uniwersytet Warszawski

Wydział Matematyki, Informatyki i Mechaniki

Abstrakcyjne specyfikacje z jawnym nośnikiem modelu

(Praca doktorska)

Krzysztof Stencel

Promotor: prof. dr hab. Jan Madey

Warszawa, maj 1999

Spis treści

Rozdział 1

Wstęp

Wieloletnia historia inżynierii oprogramowania wykazała dobitnie jak wielką rolę w procesie produkcji programów odgrywa dobra dokumentacja. Należy jednak pamiętać, że ważny jest nie tylko fakt tworzenia dokumentów, ale również ich jakość. Inżynierowie innych specjalności specyfikują wyczerpująco produkt i sposób jego wytwarzania; używają w tym celu precyzyjnych pojęć zdefiniowanych przez podstawowe dziedziny wiedzy (np. matematykę, fizykę, chemię lub mechanikę). Proces produkcji oprogramowania powinien przebiegać według tych samych reguł — wszystkie decyzje projektowe powinny być starannie udokumentowane.

Jedno z podejść do systematycznego tworzenia oprogramowania, zwane *podejściem funkcyjnym*, zostało przedstawione w artykule [PM95]. W literaturze (np. [HJL96, BH97]) bywa ono nazywane *Parnas-Madey Four Variable Model*. Zgodnie z zaleceniami podejścia funkcyjnego, następujące dokumenty powinny zostać opracowane: specyfikacja wymagań, projekt systemu, przewodnik po modułach, oraz dla każdego modułu: specyfikacja interfejsu i projekt implementacji. Poniżej krótko omówiono zawartość każdego z tych dokumentów.

Specyfikacja wymagań postrzega system komputerowy jako „czarną skrzynkę" — definiuje działanie systemu z punktu widzenia użytkownika. Zawiera ona listę *wielkości obserwowanych* i *wielkości nadzorowanych* przez system, oraz relację REQ opisującą pożądane zależności między zmianami w czasie wielkości obserwowanych a zmianami wielkości nadzorowanych. Specyfikacja wymagań zawiera również relację NAT — zapis ograniczeń środowiskowych nałożonych na specyfikowany system. Te ograniczenia mogą wynikać z praw fizyki, z wymogów narzuconych przez już zainstalowane systemy, etc. Stawiane wymagania muszą być spełnialne względem więzów ustanowionych przez otoczenie.

Projekt systemu opisuje wnętrze systemu: komputery i urządzenia zewnętrzne. Zawiera on listę *rejestrów wejściowych* i *wyjściowych*, za pomocą których komputery komunikują się z urządzeniami śledzącymi wielkości obserwowane i urządzeniami wpływającymi na wielkości nadzorowane. W tym dokumencie zdefiniowane są trzy relacje: IN opisująca zależności w czasie pomiędzy wielkościami obserwowanymi a rejestrami wejściowymi, SOF opisująca zależności w czasie pomiędzy rejestrami wejściowymi a rejestrami wyjściowymi, oraz OUT opisująca zależności w czasie pomiędzy rejestrami wyjściowymi a wielkościami nadzorowanymi. Relacja IN opisuje więc działanie detektorów, które dokonują pomiarów wielkości obserwowanych i przetwarzają je na zawartości rejestrów komputerów. Zwykle nie jest to funkcja tożsamościowa, ponieważ nie ma idealnych pomiarów, a także reprezentacja danych w komputerze jest skończona. Relacja SOF definiuje zachowanie oprogramowania — jak wartości w rejestrach wejściowych są przetwarzane na wartości w rejestrach wyjściowych. Natomiast relacja OUT

przedstawia funkcjonowanie efektorów, które przekładają wartości rejestrów wyjściowych na odpowiednie zmiany wielkości nadzorowanych. Projekt systemu powinien być poprawny — złożenie relacji *IN*, *SOF* i *OUT* musi zawierać jedynie zachowania dopuszczalne przez *REQ*, oraz pełny — dla każdego wektora wielkości obserwowanych powinien istnieć co najmniej jeden wektor wielkości nadzorowanych definiowany przez złożenie *IN*, *SOF* i *OUT*.

Konstrukcja dużego systemu nie jest możliwa bez podziału pracy na mniejsze części, zwane *modułami* [Par72a]. *Przewodnik po modułach* [BP81, PCW85] jest nieformalnym dokumentem określającym podział na moduły i zakres zadań każdego modułu. Po integracji moduły powinny spełniać specyfikację wymagań.

Dla każdego modułu należy stworzyć *specyfikację interfejsu*, która definiuje ten moduł jako „czarną skrzynkę". Wymienia on sposoby komunikacji otoczenia z modułem (programy dostępu, zmienne wejściowe i wyjściowe, etc.) i opisuje wyniki takiej zewnętrznej komunikacji. Pożądane cechy specyfikacji interfejsu zostały sformułowane m.in. w artykule [Gut77] — dobra specyfikacja jest *abstrakcyjna*, co oznacza, że posługuje się jedynie pojęciami dostępnymi zewnętrznemu obserwatorowi. W szczególności specyfikacja abstrakcyjna nie określa struktury danych zastosowanej w implementacji modułu.

Projekt implementacji definiuje strukturę danych modułu i wpływ poszczególnych zdarzeń (wywołań programów dostępu, zmian wartości zmiennych wejściowych, etc.) na stan tej struktury danych. Projekt implementacji powinien być poprawny względem specyfikacji interfejsu — efekt zdarzenia na strukturze danych powinien być zgodny z tym co jest opisane w specyfikacji interfejsu. Weryfikację poprawności wspomaga zdefiniowana w projekcie implemetacji *funkcja abstrakcji*, która przyporządkowuje stanom struktury danych, stany abstrakcyjne modułu, tj. stany zdefiniowane w specyfikacji interfejsu.

Kod jest ukoronowaniem wysiłków producentów oprogramowania. Powinien być poprawny względem projektu implementacji.

Kierowany przez prof. Jana Madeya zespół pracowników i studentów Instytutu Informatyki Uniwersytetu Warszawskiego zajmuje się opracowaniem metod specyfikacji i weryfikacji oprogramowania, a w szczególności specyfikacji interfejsów i projektu implementacji. Nasz zespół współpracuje z prof. Davidem Parnasem z McMaster University, Hamilton, Ontario, Kanada, oraz prof. Michałem Iglewskim z Université du Quebéc à Hull, Hull, Quebéc, Kanada. Przegląd wyników tej współpracy można znaleźć w raportach [IMPK93, IKM$^+$95b]. Te prace przedstawiają, między innymi, przykłady zastosowania pełnej ścieżki rozwoju modułu, od specyfikacji interfejsu, poprzez projekt implementacji, do kodu programu. O projekcie implementacji można przeczytać w [IMD97]. Weryfikacją poprawności kodu względem projektu implementacji zajmuje się powstająca praca [Kub99].

Niniejsza praca jest poświęcona jednej z metod specyfikacji interfejsu — metodzie tropów (ang. *Trace Assertion Method*, w skrócie *TAM*). Podstawowym pojęciem tej metody jest *trop* — zapis historii modułu z punktu widzenia zewnętrznego obserwatora, czyli lista wszystkich zdarzeń jaki miały wpływ na specyfikowany moduł i wyników jakie wygenerował ten moduł. Metoda tropów posługuje się jedynie tropami w celu określenia oczekiwanego działania programów. Trop jest pojęciem ze świata zewnętrznego obserwatora i dlatego uważamy, że specyfikacje tropowe są abstrakcyjne. Jednocześnie, każda specyfikacja tropowa, ma naturalny model — maszynę, której stanami są tropy kanoniczne — jawnie wskazani przez specyfikację reprezentanci klas abstrakcji obserwacyjnej równoważności tropów. W ten sposób metoda tropów posiada dwie na pozór wykluczające się własności — **abstrakcyjność i jawność nośnika modelu**.

Ewolucja metody tropów jest bardzo podobna do historii rozwoju innych naukowych teo-

rii. W pierwszej fazie pojawia się nowa koncepcja — jeszcze nie do końca sformalizowana i doceniana. Druga faza to rozwój zastosowań nowej idei. W trzeciej fazie następuje dopracowanie i formalizacja nowej teorii. Zazwyczaj stosowanie takiej teorii wymaga dłuższego czasu na poznanie jej licznych szczegółów. To może łatwo zniechęcić potencjalnych użytkowników, zwłaszcza tych o mniejszym przygotowaniu matematycznym.

Jako początek pierwszej fazy rozwoju metody tropów można przyjąć ukazanie się pracy [Par72b], zaś zakończenie tej fazy wyznacza [BP78]. Apogeum drugiej fazy, to prace nad systemem wspomagającym sterowanie samolotem A-7E [Ers92] i systemem awaryjnego wyłączania reaktora w elektrowni atomowej w Darlington, Ontario, Kanada [PAM91], oraz prace na protokołami komunikacyjnymi [CP93, BIMO94]. Za koniec koniec trzeciej uważamy raport [IKM+97]. Potencjalnego użytkownika opisanej w tej pracy metody TAM'97 może zniechęcić skomplikowana składnia, zawierająca w szczególności mało intuicyjne konstrukcje. Niektórzy naukowcy [Nor95, Jan95, She97] już w trakcie trzeciej fazy rozwoju zdawali sobie sprawę z opisanych powyżej zagrożeń i proponowali nowe podejście do metody tropów.

Teza pracy

Celem niniejszej pracy jest zaproponowanie metody, która mając ściśle zdefiniowaną składnię i semantykę, zachowuje siłę wyrazu i prostotę metody tropów w jej oryginalnym ujęciu. Prezentowana w pracy nowa wersja metody tropów, zwana TAM 2000, stara się unikać komplikacji syntaktycznej TAM'97. W szczególności TAM 2000 proponuje nowe rozwiązanie problemu zmiennych wejściowych i wyjściowych, które udowodniły swą praktyczną przydatność [Ers92, CP93, BIMO94], ale ze względu na brak satysfakcjonującej definicji były świadomie pomijane przez autorów rozmaitych prac (np. [Kub94, IMPK93, Ste95, Las95, Kre98]).

Tezą tej pracy jest więc stwierdzenie, że istnieje abstrakcyjna metoda formalnej specyfikacji z rygorystycznie opisaną składnią i semantyką, która zachowując zasadnicze założenia i zalety oryginalnej metody tropów, pozwala tworzyć przejrzyste specyfikacje.

Wykażemy to poprzez przedstawienie takiej metody, oraz wybranych z literatury przykładów specyfikacji, które zapisane w zaprezentowanej metodzie, są prostsze i bardziej intuicyjne, niż specyfikacje znajdujące się w źródłowych pracach.

Struktura pracy

W rozdziale 2 dokonano porównania metody tropów z innymi metodami specyfikacji i postarano się uzasadnić celowość prowadzenia prac nad metodą tropów. Rozdział 3 zawiera kronikę metody tropów — odnotowano najważniejsze wydarzenia i ich wpływ na obecny stan prac nad metodą. Niedoskonałości najnowszego standardu TAM'97 przedstawiono w rozdziale 4; tamże zaproponowano sposoby usunięcia tych wad i przedyskutowano alternatywne rozwiązania. Te rozważania zostały zilustrowane fragmentami specyfikacji zapisanych w TAM 2000, których pełne teksty znajdują się w dodatkach. Rozdział 5 zawiera prezentację podstawowych pojęć nowej wersji metody tropów. W rozdziale 6 przedstawiono składnię TAM 2000 i opisano w języku naturalnym znaczenie każdej konstrukcji syntaktycznej. Rozdział 7 stanowi formalną definicję TAM 2000. W rozdziale 8 porównano metodę TAM 2000 z wcześniejszymi wersjami metody tropów. Starano się także uzasadnić, że specyfikacje w TAM 2000 są prostsze i bardziej intuicyjne. Rozdział 9 podsumowuje pracę. Dodatki A–C są uzupełnieniem treści rozdziału 7: A zawiera gramatykę bezkontekstową specyfikacji w TAM 2000, B — reguły statycznej poprawności specyfikacji, zaś C — reguły wyliczania wartości wyrażeń. W dodat-

kach D–I znajdują się przykłady specyfikacji w TAM 2000. Dodatki D–G są przykładami wybranymi z literatury metody tropów; ich obecność w niniejszej rozprawie służy wykazaniu jej tezy. Dodatki H i I, to specyfikacje pewnych modułów systemu Unix wzorowane na pracy magisterskiej [KM96]. Są to bardziej złożone przykłady ilustrujące praktyczne zastosowania metody TAM 2000.

Podziękowania

Pragnę gorąco podziękować mojemu promotorowi, profesorowi Janowi Madeyowi za jego nieocenioną pomoc i opiekę, ktorą otoczył mnie już w trakcie mojej nauki w szkole podstawowej. Jestem niezwykle wdzięczny profesorowi za doprowadzenie mnie do chwili, w której mogłem przedstawić swoją rozprawę doktorską.

Opracowując niniejszą rozprawę korzystałem z różnorakich rezultatów wieloletniej współpracy z profesorem Davidem Parnasem z McMaster University w Hamilton, Ontario, Kanada, profesorem Michałem Iglewskim z Université du Quebéc à Hull w Hull, Quebéc, Kanada, oraz z pracownikami naukowymi Uniwersytetu Warszawskiego: Marcinem Engelem, Marcinem Kubicą i Janiną Mincer-Daszkiewicz. Pomoc wszystkich wymienionych tu osób niezwykle przyczyniła się do powstania tej rozprawy.

Składam gorące podziękowania także mojej żonie i córce, które nie tylko wykazały się ogromną cierpliwością w stosunku do swojego męża i ojca, ale także nie szczędziły zachęty w trudnych chwilach.

Niniejsza rozprawa została zrealizowana częściowo w ramach projektu badawczego KBN nr 8 T11C 015 15.

Rozdział 2

Porównanie metod specyfikacji interfejsów modułów

W przeglądowej pracy [Wan94a] metody specyfikacji interfejsów modułów zostały podzielone na cztery klasy:

- metody modelu abstrakcyjnego (np. VDM i Z),
- metody algebraiczne (np. ACT ONE, CASL, CLEAR, Larch, OBJ),
- metody tropowe (np. TAM, TAM'97),
- metody oparte na procesach (np. Estelle, LOTOS i SDL).

Celem niniejszego rozdziału jest porównanie metod tropowych z innymi metodami specyfikacji interfejsów modułów. Najważniejszym z zastosowanych kryteriów będzie abstrakcyjność metod — czy pozostawiają maksymalną swobodę implementatorowi? czy posługują się jedynie pojęciami dostępnymi zewnętrznemu obserwatorowi?

Poprzez to zestawienie postaramy się uzasadnić, dlaczego warto prowadzić badania nad metodą tropów. Rodział ten nie jest przeglądem wszystkich istniejących metod, ale raczej podsumowaniem prac, które uważamy za istotne dla dokonywanego porównywania.

2.1 Metody modelu abstrakcyjnego

Stosując metody tej klasy zakładamy, że system ma stany, które zmieniają się w odpowiedzi na wywołania programów dostępu. Wywołanie programu, oprócz zmiany stanu, może powodować wygenerowanie wyniku. Najbardziej znanymi metodami tej klasy są VDM [Bjø81, BP83, Jon86] i Z [Spi92, Wor92]. Stany abstrakcyjne systemów są wyrażane w tych metodach za pomocą pewnych predefiniowanych bytów: w VDM są to ciągi, krotki, zbiory, przekształcenia i drzewa; w Z podstawową strukturą jest zbiór; inne, bardziej słożone struktury są konstruowane za pomocą zbiorów. Działanie programów dostępu specyfikujemy określając w jaki sposób ich wywołania zmieniają stan systemu.

Definiowanie zbioru stanów musi poprzedzać sformułowanie każdej specyfikacji. Takie podejście jest niebezpieczne, ponieważ może prowadzić do zbyt szczegółowej specyfikacji. Wybór abstrakcyjnej struktury danych może sugerować sposób implementacji. Programista musi starannie oddzielić to co jest wymaganiem stawianym systemowi (zewnętrznie obserwowalne

zachowanie), od tego co jest środkiem pozwalającym to wymaganie opisać (zbiór stanów abstrakcyjnych). W pracy [Par71] zauważono, że trudno oczekiwać, aby osoba, która ma dostęp do pewnej informacji, nie skorzystała z niej.

W metodzie Larch (por. p. 2.2), dzięki podejściu dwuwarstwowemu, nie ma tego problemu. Podobne dwie warstwy są wyróżnione w modelu funkcyjnym [PM95] (por. rozdział 1): specyfikacja interfejsu i projekt implementacji. Metoda tropów szczególnie dobrze nadaje się do pisania specyfikacji interfejsów — posługuje się jedynie tropem, zapisem historii modułu z punktu widzenia zewnętrznego obserwatora. Nie ma zatem niebezpieczeństwa nadmiernej specyfikacji. Struktura danych jest definiowana dopiero w projekcie implementacji.

Istnieją propozycje wprowadzenia hermetyzacji do tych metod, np. dodanie modułów do VDM [Bea88], czy dodanie rozdziałów do Z [SM90a]. Pozwala to osiągnąć pewien poziom „ukrywania informacji" — poprzez listy bytów importowanych, eksportowanych i prywatnych. Jednak wciąż byty eksportowane nie mogą zataić wewnętrznej struktury („abstrakcyjnej implementacji") przed swoimi użytkownikami. Propozycja dodania abstrakcyjnych typów danych do VDM [SM90b] pozwala na pożądany stopień hermetyzacji, jednak to rozszerzenie przenosi nas w świat specyfikacji algebraicznych (por. p. 2.2). Dodatkową wadą takiego rozwiązania jest jego nieadekwatność do VDM — konstruktory wartości abstrakcyjnego typu danych nie są programami dostępnymi użytkownikowi. Są jedynie ukrytymi programami pomocniczymi służącymi do budowania zbioru stanów modelu abstrakcyjnego. Szczegółowa dyskusja na temat ukrytych programów jest przeprowadzona poniżej.

2.2 Metody algebraiczne

Specyfikacja algebraiczna składa się z listy nazw i sygnatur funkcji algebry, oraz listy aksjomatów. Funkcje algebry repezentują programy dostępu definiowanego modułu. Specyfikacja algebraiczna wyznacza klasę algebr wielorodzajowych, która jest ograniczona przez wymienione w specyfikacji aksjomaty, a także przez prawidła metody. Na przykład możemy być zainteresowani jedynie algebrami początkowymi [EM85, GTWW75, GTW78], lub algebrami generowanymi przez pewien zbiór operacji (tak jak w metodzie Larch [GH83, GH86b, GH93]).

Istnieją proste koncepcyjnie typy danych, dla których nie da się napisać klasycznej specyfikacji algebraicznej o skończonym zbiorze aksjomatów. Przykładem na to może być stos wzbogacony o pewne szczególne operacje pozwalające podglądać jego zawartość poniżej czubka [Maj77]. Metody algebraiczne, takie jak np. ACT ONE [EM85], CASL [BST99], CLEAR [BG77, BG81] czy OBJ [FGJM85, GT79], radzą sobie z takimi typami danych za pomocą tzw. *ukrytych programów*, tzn. takich, które nie są elementami interfejsu.

Ukryte programy powinny być domeną implementacji i projektu. Właśnie tam, a nie w definicji interfejsu, tworzymy pomocnicze byty, które są ukryte przed użytkownikiem. Specyfikacja interfejsu z definicji powinna zajmować się jedynie rzeczami jawnymi. Kłopotu z ukrytymi programami nie ma w tych metodykach, gdzie wyraźnie odróżniono specyfikację interfejsu od projektu implementacji. Ten warunek niewątpliwie spełnia model funkcyjny [PM95] z metodą tropów w roli specyfikacji interfejsu, oraz w pewnym sensie podejście dwuwarstwowe Larch.

W metodzie tropów możemy korzystać z funkcji pomocniczych, które jednak nie są programami dostępnymi użytkownikowi. Nie mogą zatem występować w tropie i tym samym służyć do konstrukcji przestrzeni stanów (do tego są potrzebne ukryte programy w specyfikacjach algebraicznych). Abstrakcyjny stan modułu jest tropem, w całości dostępnym i zrozumiałym dla zewnętrznego obserwatora.

W dolnej warstwie specyfikacji w Larch (napisanej w LSL [GH86a]) można definiować funkcje, które nie mają swojego odpowiednika w implementacji. Rzeczywisty interfejs modułu jest zapisany w jednym z języków górnej warstwy, związanych z konkretnym językiem programowania np.: Larch/C [GH92], Larch/Modula-3 [Jon91], Larch/CLU [Win87]. Dzięki temu pojęcia pomocnicze użyte w definicji interfejsu są starannie wyróżnione wśród innych pojęć.

Innym minusem metod algebraicznych jest nadmierna specyfikacja. W klasycznej, algebraicznej definicji stosu występuje równanie:

$$pop(push(s, x)) = s$$

To równanie wyklucza wiele sensownych implementacji stosu, np. tablicową z wskaźnikiem czubka stosu, w której nie zamazujemy zdjętego elementu. Nie istnieje skończony zbiór aksjomatów, który definiowałby klasę takich "rozsądnych" implementacji stosu [ST97]. Niektóre metody algebraiczne (np. ASL [SW83]) nie mają tej wady: posiadają mechanizm *abstrakcji*, polegający na akceptowaniu także modeli obserwacyjnie równoważnych [ST87a] modelom, które spełniają specyfikację w sposób dosłowny. Niestety, tak skonstruowane specyfikacje stają się niezwykle zawikłane; do ich zrozumienia potrzebna jest żmudna i czasochłonna analiza. Metoda tropów w ramach podejścia funkcyjnego [PM95] nie ma tych wad. Dzięki rozdzieleniu specyfikacji interfejsu od projektu implementacji, funkcja abstrakcji może przyporządkowywać ten sam stan abstrakcyjny (zawartość stosu) wielu różnym stanom struktury danych zastosowanej w implementacji (zawartość tablicy).

Problem uwikłania specyfikacji poruszony w poprzednim akapicie jest dobrze rozwiązany w metodzie tropów, gdzie wprost definiujemy zbiór stanów abstrakcyjnych — tropów kanonicznych. Tę samą informację możemy jedynie *wyprowadzić* z specyfikacji algebraicznej. W Larch ten proces jest wspomagany przez klauzule **generated by**, która jednak tylko ogranicza zakres poszukiwań. Podanie zbioru stanów abstrakcyjnych *explicite* ułatwia definiowanie funkcji abstrakcji, ponieważ wprost określa jej przeciwdziedzinę.

2.3 Metody oparte na procesach

Przykładami metod opartych na procesach są LOTOS [BB87, Bri86], Estelle [BD87, Lin86] i SDL [RS82, ST87b]. W takich metodach podstawowym pojęciem jest *proces* (Estelle i SDL nazywają go odpowiednio *modułem* i *blokiem*). Procesy przesyłają pomiędzy sobą *komunikaty*, które mogą być nadawane i odbierane jedynie w ściśle określonych *punktach interakcji*. Proces jest zatem jednostką hermetyzacji o ustalonym interfejsie. Dlatego możemy porównywać metody oparte na procesach z metodami specyfikacji interfejsu.

Metody z rozważanej klasy definiują proces jako „szklaną skrzynkę" — mamy dostęp do wnętrza każdego procesu; możemy sprawdzić jak z prostszych procesów zbudowano bardziej skomplikowane. W tym sensie metody oparte na procesach są podobne do metod modelu abstrakcyjnego (por. p. 2.1).

Metoda tropów opisuje reakcje specyfikowanych obiektów na bodźce zewnętrzne; zwykle są to wyniki generowane przez wywołania programów dostępu. Moduł biorący udział w protokole komunikacyjnym na odebrane sygnały reaguje wysłaniem sygnałów. Modyfikację metody tropów pozwalającą na zapisywanie informacji o nadawanych przez moduł sygnałach zaproponowano w artykule [Hof85]. Inne podejście do tropowych specyfikacji protokołów zostało przedstawione w raporcie [Par90]. Prace [CP93, BIMO94] zawierają przykłady specyfikacji

protokołów, w których rolę kanałów komunikacyjnych spełniają opisane w raporcie [PW89] zmienne wejściowe i wyjściowe.

Truizmem jest stwierdzenie, że w protokole komunikacyjnym najważniejsza jest komunikacja pomiędzy jego uczestnikami. Można zdefiniować protokół nie ujawniając implementacji modułu biorącego w nim udział. Metoda tropów posługuje się jedynie pojęciami ze świata zewnętrzenego obserwatora. Pozwala to na wyższy poziom hermetyzacji i uniezależnienia od implementacji niż metody oparte na procesach.

Rozdział 3

Ewolucja metody tropów

W niniejszym rozdziale przedstawimy najważniejsze prace, które doprowadziły do powstania metody tropów w jej obecnym kształcie. Temu celowi poświęcone są punkty 3.1, 3.2, 3.3 i 3.4. Następnie, w punkcie 3.5, przyjrzymy się ewolucji zmiennych wejściowych i wyjściowych. Postanowiliśmy wydzielić prezentację ich historii, ponieważ uważamy je za istotny element metody tropów, a do chwili obecnej nie doczekały się one satysfakcjonującego ujęcia. W punkcie 3.6 przedstawimy i przedyskutujemy najnowsze propozycje gruntownych zmian w założeniach metody tropów.

3.1 Zanim wymyślono trop...

W pracy [Par72b] sformułowano kilka założeń fundamentalnych dla metod tropowych. Tamże przedstawiono technikę specyfikacji skonstruowaną według tych założeń. Definiowany obiekt uważamy za maszynę stanową, przy czym najczęściej liczba stanów jest skończona. Wszelka komunikacja pomiędzy obiektem i jego otoczeniem dokonuje się poprzez programy dostępu, które dzielimy dwie grupy: O-programy, których wywołania powodują zmiany stanu obiektu i V-programy, które dostarczają informacji o stanie obiektu. Nie definiujemy zbioru stanów obiektu w sposób jawny. W każdej chwili stan obiektu jest reprezentowany przez wartości jakie przekazałyby V-programy, gdyby zostały wywołane. Specyfikujemy wyniki potencjalnych wywołań V-programów w chwili zainicjowania obiektu i ich zmiany powstające w wyniku użycia O-programów.

Idee zaprezentowane w [Par72b] zostały wykorzystane przy tworzeniu metody SPECIAL [RR76]. Formalizacja tych idei doprowadziła do odkrycia konieczności stosowania ukrytych programów (por. p. 2.2) w takim podejściu. Dotyczy to nawet bardzo prostej struktury danych jaką jest stos.

3.2 Wprowadzenie pojęcia tropu

Dopiero w pracy [BP78] podano technikę specyfikacji zachowania obiektów, która obywa się bez ukrytych programów, oraz nie zawiera sugestii przyszłej implementacji w postaci namiastki struktury danych (por. p. 2.1). Pojęciem, które to umożliwiło, jest *trop* — zapis historii obiektu z punktu widzenia zewnętrznego obserwatora — lista wszystkich wywołań programów dostępu i wyników przekazanych przez te wywołania. Za pomocą tropów reprezentujemy stany obiektów. Oczywiście różne tropy mogą prowadzić do tego samego stanu.

Takie tropy nazywamy równoważnymi. Podzbiór tropów, zwany *tropami legalnymi*, opisuje poprawne użycie obiektu; pozostałe (np. POP na pustym stosie) reprezentują sekwencje wywołań, których użytkownik powinien unikać. Specyfikacja w ujęciu [BP78] zawiera listę nazw i sygnatur programów dostępu, oraz listę formuł określających relację równoważności tropów, legalność tropów i wartości przekazywane przez V-programy w zależności od historii obiektu opisanej tropem.

W artykule [McL84] uściślono technikę z [BP78]. Wprowadzono *język tropowy* i podano w sposób formalny jego składnię i semantykę. Zaprezentowano również system dedukcyjny dla tego języka oraz udowodniono jego pełność i poprawność.

3.3 Wprowadzenie pojęcia tropu kanonicznego

Kolejny kamień milowy stanowi praca [Hof85], w której przedstawiono metodykę tworzenia specyfikacji tropowych w ramach podejścia i składni opisanej w [BP78]. Zaobserwowano, że łatwiej jest posługiwać się pojedynczymi tropami niż klasami równoważności tropów. Metodyka z [Hof85] zaleca wybór co najmniej jednego reprezentanta z każdej klasy abstrakcji. Wybrane tropy nazywamy *postaciami normalnymi*. Działanie modułu definiujemy w dwóch krokach. Specyfikujemy wyniki generowane przez programy dostępu dla tropów będących postaciami normalnymi. Dla każdego tropu wskazujemy jedną z równoważnych mu postaci normalnych. Wykorzystujemy tu indukcję — definiujemy równoważną postać normalną tylko dla tropów będących postaciami normalnymi wydłużonymi o jedno wywołanie programu dostępu.

W raporcie [PW89] opisano nowa wersja metody tropów, której składnia wymusza stosowanie metodyki z [Hof85]. Specyfikacja tropowa została podzielona na cztery sekcje, w których definiujemy kolejno: nazwy i argumenty programów dostępu, zbiór *tropów kanonicznych* (które spełniają rolę postaci normalnych z [Hof85]), *funkcję rozszerzenia* (przyporządkowanie jednoelementowym rozszerzeniom tropów kanonicznych równoważnych im tropów kanonicznych), *relację wyjścia* (określenie jakie wyniki są generowane przez programy dostępu dla tropów kanoniczych).

Metoda tropów posługuje się jedynie tropami, które są pojęciem ze świata zewnętrznego obserwatora. Wprowadzenie tropu kanonicznego spowodowało, że każda specyfikacja tropowa ma naturalny i jednoznaczny model — maszynę, której stanami są właśnie tropy kanoniczne. Metoda tropów posiada zatem dwie na pozór wykluczające się własności — abstrakcyjność i jawność nośnika modelu.

3.4 Uściślanie podstaw metody

Raport [IMPK93] uściśla metodę przedstawioną w [PW89]. W szczególności zdefiniowano uniwersalne sposoby przekazywania argumentów programów dostępu. W [IMPK93] opisano także projekt implementacji i przedstawiono w jaki sposób jest on powiązany ze specyfikacją tropową. Tamże znajdują się przykłady — specyfikacje tropowe, projekty implementacji i kody źródłowe pewnych niebanalnych typów danych.

Intuicje związane z metodą tropów zostały ponownie sformułowane w [IMS94]. Wynikiem przeglądu podstaw była obserwacja, że specyfikacja tropowa definiuje dwie, niekoniecznie identyczne, relacje równoważności. Dwa tropy są *obserwacyjnie równoważne*, gdy przyszłe zachowanie obiektu będzie takie same, niezależnie od tego, który z tych dwóch tropów opisuje dotychczasową historię tego obiektu. Idea obserwacyjnej równoważności tropów jest bardzo

podobna do obserwacyjnej równoważności algebr [ST87a], jednak pojęcia te istotnie się różnią — pierwsze dotyczy pojedynczych stanów modułu, zaś drugie całych implementacji modułów. Dwa tropy są *specyfikacyjnie równoważne*, gdy redukują się do tego samego tropu kanonicznego. Praca [IMS94] uważa za poprawną specyfikację, w której te dwie relacje na tropach nie są sobie równe. Wówczas pewna klasa obserwacyjnej równoważności ma co najmniej dwóch reprezentantów wśród tropów kanonicznych. To znaczy, że istnieją różne stany abstrakcyjne, których zewnętrzny obserwator nie może rozróżnić.

3.5 Zmienne wejściowe i zmienne wyjściowe

Za pomocą zmiennych wejściowych i zmiennych wyjściowych moduł odczytuje i przekazuje wielkości fizyczne, np. ciśnienie, temperaturę, wilgotność. Intencją wprowadzenia takiego mechanizmu komunikacji było umożliwienie specyfikowania systemów wbudowanych. Jednak raport [PW89] jedynie wzmiankuje zmienne wejściowe, zaś zmiennym wyjściowym wyznacza specjalną rolę — wyniki generowane przez programy dostępu są wyznaczane na podstawie wartości zmiennych wyjściowych i argumentów wywołania. Zmienne wyjściowe służą zatem do przechowywania wartości, które zostaną przekazane przez ewentualne wywołania programów. Przypomina to technikę z [Par72b] (por. p. 3.1), choć dodawanie ukrytych programów nie jest tu konieczne, ponieważ stan modułu jest wyznaczony nie tylko przez wartości zmiennych wyjściowych, ale i przez trop kanoniczny. Ta szczególna rola zmiennych wyjściowych powoduje zatracenie ich sprzętowego charakteru. Natomiast do zalet takiego rozwiązania należy zaliczyć ułatwienie specyfikowania pewnych modułów niedeterministycznych [IMDS95].

Naturalne zastosowania zmiennych wejściowych i wyjściowych zostały przedstawione w pracach [Ers92, vSPM93, CP93, BIMO94, IKM$^+$97]. Wartości zmiennych wyjściowych są definiowane jako funkcja stanu modułu. Zmiennych wejściowych moduł nie wykorzystuje bezpośrednio. Zmiany ich wartości powodują *zdarzenia wejściowe*. W specyfikacji definiowane są *klasy zdarzeń* — zbiory zdarzeń o podobnych właściwościach. W [Ers92, vSPM93, CP93] wyróżniono zdarzenia warunkowe i bezwarunkowe. Każda klasa *zdarzeń warunkowych* jest związana z formułą logiczną, w której zmiennymi wolnymi są nazwy zmiennych wejściowych. Zdarzenie należące do tej klasy zachodzi, gdy ta formuła staje się prawdziwa, lub fałszywa. Klasy *zdarzeń bezwarunkowych* są związane z pojedynczą zmienną; zdarzenia będące elementami takiej klasy mają miejsce zawsze ilekroć zmieni się wartość tej zmiennej. Praca [BIMO94] dopuszcza jedynie zdarzenia bezwarunkowe.

Zmienne wejściowe i wyjściowe występują również SCR [HKL$^+$98] — metodzie modelu abstrakcyjnego opartej na modelu funkcyjnym [PM95]. Każde zdarzenie wejściowe w SCR jest powiązane z pojedynczą zmienną. Dodatkowo przyjmuje się założenie, że w każdej chwili może zajść co najwyżej jedno zdarzenie. Raport TAM'97 [IKM$^+$97] definiuje zmienne wejściowe i wyjściowe tak jak w SCR — zdarzenia warunkowe muszą zależeć od dokładnie jednej zmiennej; dopuszczono też zdarzenia bezwarunkowe.

Zmienne wyjściowe stanowią aktywny interfejs wyjściowy modułu, ponieważ mogą spowodować zdarzenia w innych modułach. Wyniki wywołań programów dostępu są wyjściami pasywnymi — na ich podstawie moduł wywołujący decyduje jakie akcje zostaną podjęte. Inny pomysł na aktywne wyjście zaprezentowano w [Hof85] — na wywołania programów dostępu program reaguje *wywołaniami wychodzącymi*. Moduł nie wie, co powodują wychodzące wywołania; podobnie jak nie wie, dokąd są podłączone jego zmienne wyjściowe. Specyfikacja interfejsu nie jest miejscem dla takich informacji; połączenia międzymodułowe są zagadnie-

niem implementacyjnym i powinny być definiowane w projekcie implementacji. Podobny pomysł został zaproponowany w [Nor95], lecz w tym ujęciu dotyczy wykorzystywania jednych modułów przez inne — został zatem przystosowany do projektowania implementacji, a nie do specyfikowania interfejsu.

3.6 Stan obecny

Raporty [IKM+97, Kub97] zawierają pierwszą od czasu publikacji [McL84], pełną definicję metody tropów. W [IKM+97] zapisano formalnie składnię specyfikacji tropowych i przedstawiono ścisły opis semantyki w języku naturalnym. W [Kub97] ta semantyka została zapisana w sposób formalny. W tych dwóch pracach uwzględniono wyniki analizy metody tropów zawartej m. in. w pracy magisterskiej [Kub94]. Rezultatami rygorystycznego podejścia do definicji metody tropów były liczne, dokonane w niej zmiany. Niektóre z nich spowodowały skomplikowanie wielu pojęć, które wcześniej łatwo poddawały się intuicji. Metoda z [IKM+97] nosi nazwę TAM'97, ponieważ znacznie odbiega (m.in. stopniem złożoności) od poprzednich wersji TAM.

Równolegle, w McMaster University, Hamilton, Ontario (Kanada) prowadzono nad metodą tropów prace, których celem było udoskonalenie metody poprzez gruntowną modyfikację jej podstaw.

W raporcie [Jan95] zaproponowano wprowadzenie relacji w miejsce funkcji rozszerzenia. Oznacza, to, że w każdej chwili stan abstrakcyjny modułu jest reprezentowany przez zbiór *tropów konstruktorów*, które są odpowiednikami tropów kanonicznych z klasycznych specyfikacji tropowych. Zaletą wprowadzenia relacji rozszerzenia jest ułatwienie specyfikowania niektórych modułów niedeterministycznych i wygodna formalizacja pojęcia tropu nielegalnego (taki trop redukuje się do zbioru pustego). Janicki zaproponował również systematyczne podejście do specyfikowania efektów błędnych wywołań programów dostępu. Najpierw definiujemy reakcje modułu na poprawne wywołania; dopiero potem do specyfikacji dodawane są informacje o wynikach pozostałych wywołań. Te ostatnie powinny być wyraziście odróżnione od tych pierwszych za pomocą środków syntaktycznych.

Raport [Nor95] porządkuje teorię TAM przedstawiając dowody podstawowych twierdzeń o własnościach specyfikacji tropowych. W tej pracy rozważano również konsekwencje wprowadzenia relacji rozszerzenia, formułując i dowodząc podobne twierdzenia jak dla metody w ujęciu oryginalnym. Tamże zaroponowano również pewne zmiany w składni metody. Pierwszą z nich jest zastosowanie tabelki funkcji wektorowej (ang. *vector equality table* [Par92]) do jednoczesnego definiowania funkcji rozszerzenia i relacji wyjścia dla każdego programu dostępu. Druga propozycja, to wprowadzenie makrodefinicji. W [Nor95] znajdują się też rozważania semantyczne — wprowadzono automaty dwu- i trójkolorowe jako model specyfikacji tropowej. Czarne stany reprezentują stany modułu, natomiast stany zielone i ewentualne czerwone są przejściowe — moduł przebywa w nich w czasie, odpowiednio, generowania wyników i generowania wychodzących wywołań (por. p. 3.5).

Rewolucyjne zmiany metody tropów zostały zaproponowane w pracy [She97] — stany abstrakcyjne mogą być dowolne, np. ciągi, zbiory, krotki. Wynikiem zaakceptowania tej daleko idącej propozycji jest całkowita rezygnacja z pojęcia tropu. Raport [Doo99] przedstawia metodę ITAM, która realizuje propozycje zawarte w [She97]. Podobnie jak w notacji Z (por. p. 2.1) pojęciem pierwotnym w ITAM jest zbiór. Za pomocą zbiorów definiujemy stan abstrakcyjny obiektu, zwany *reprezentacją kanoniczną*.

Dyskusja

Wprowadzenie relacji rozszerzenia rozważane w [Jan95, Nor95] oprócz zalet, ma również istotne wady. Po pierwsze, czasem nie można napisać specyfikacji bez wprowadzania ukrytych programów (por. p. 2.2), które są niezbędne przy definiowaniu zbioru tropów konstruktorów. Po drugie, specyfikacja posługuje się zbiorami tropów, a nie pojedynczymi tropami, i tym samym metoda zatraca walory, które uzyskała dzięki pomysłowi opisanemu w p. 3.3.

Makrodefinicje w ujęciu [Nor95] uważamy za trudne do zaakceptowania, ponieważ nie mając parametrów, mogą zawierać zmienne wolne, które zostaną związane na zewnątrz wystąpienia makrodefinicji. Naszym zdaniem nie ma również potrzeby wprowadzania wielokolorowych automatów; do zdefiniowania semantyki specyfikacji tropowych w zupełności wystarczą maszyny Mealego [Mea55].

Rezygnacja z pojęcia tropu proponowana w pracach [She97, Doo99] powoduje, że przestajemy mieć do czynienia z metodą tropów. W istocie metoda ITAM opisana w [Doo99] jest metodą modelu abstrakcyjnego (por. p. 2.1), a nie metodą tropową. Uważamy, że nie jest to właściwy kierunek — łatwiej jest skorzystać z jednej z bardzo popularnych metod tej klasy, niż rozwijać własną. W szczególności, model pojęciowy ITAM oparty na zbiorach powoduje, że ta metoda jest bardzo podobna do notacji Z, a nie ma przy tym ani elegancji, ani dojrzałości Z.

Przedstawione powyżej argumenty uzasadniają, dlaczego w TAM 2000 nie zostaną uwzględnione propozycje z prac [Jan95, Nor95, Doo99].

Rozdział 4

Propozycje zmian w metodzie TAM'97

W niniejszym rozdziale zaproponujemy modyfikacje metody TAM'97, których podstawowym celem jest zmniejszenie złożoności syntaktycznej. Proponowane zmiany zmierzają także do udoskonalenia pewnych aspektów metody, np.: aktywny interfejs modułu (związany ze zmiennymy wyjściowymi), parametryzacja specyfikacji. Wszystkie propozycje zostaną zilustrowane fragmentami specyfikacji dwóch modułów.

Pierwsza z nich definiuje stos (por. dodatek D). Do standardowych programów dostępu stosu (PUSH, POP i TOP) dodano programy JOIN i SPLIT. Argumentami JOIN i SPLIT są dwa stosy. Wywołanie JOIN powoduje skopiowanie zawartości całego drugiego stosu na czubek pierwszego stosu. Wywołanie SPLIT powoduje przeniesienie „górnej" połówki pierwszego stosu na wierzch drugiego stosu. Jeśli na pierwszym stosie znajduje się nieparzysta liczba elementów, to przenosimy na drugi stos o jeden element więcej, niż pozostanie ich na pierwszym stosie. Stos z operacją JOIN wykorzystano jako przykład w raporcie TAM'97 [IKM$^+$97], oraz w artykule [EKMD98]. W TAM 2000 nie można uzależnić działanie programu od występowania aliasów wśród argumentów jego wywołania (por. p. 4.1), zatem JOIN w TAM 2000 jest programem, który modyfikuje dokładnie jeden obiekt. Wzbogaciliśmy przykład stosu o program SPLIT, aby wykazać, że TAM 2000 pozwala specyfikować programy, które modyfikują więcej niż jeden obiekt.

Druga z przykładowych specyfikacji (por. dodatek F) definiuje zarządcę nazw i pochodzi z raportu [IMPK93]. Obiekty tego modułu administrują pewną pulą wartości, które mogą być wykorzystywane jako nazwy. Każdy obiekt ma dwa programy dostępu: PICK rezerwuje pewną nazwę z zarządzanej puli i przekazuje ją wywołującemu; DISPOSE powoduje zwolnienie nazwy podanej jako argument i dołączenie jej do puli dostępnych nazw. Program PICK jest niedeterministyczny — przekazana nazwa może być dowolną z puli. Przykład zarządcy nazw ilustruje zatem, jak w TAM 2000 można definiować moduły niedeterministyczne.

4.1 Nazwy obiektów

Pierwotnie, metoda tropów wspomagała jedynie specyfikowanie *modułów jednoobiektowych*. Nie było wówczas mowy o nazwie obiektu, ponieważ do identyfikacji jedynego obiektu nie była ona potrzebna. W raporcie [PW89] wprowadzono *moduły wieloobiektowe*. Argumenty wywołania programów dostępu takiego modułu muszą wskazywać obiekty, na które to wywo-

łanie wpływa. W tym celu nadano obiektom nazwy, które były przekazywane jako argumenty programów. Wyróżniono kilka sposobów przekazywania argumentów. Poprzez argument wejściowy jest przekazywana wartość obiektu. Argument wyjściowy udostępnia nazwę obiektu, któremu zostanie przypisana wynikowa wartość. Argument wejściowo-wyjściowy jest kombinacją dwóch poprzednich — udostępnia wartość i nazwę obiektu.

Gdy program dostępu ma dwa argumenty wyjściowe tego samego typu, może nastąpić wywołanie, w którym ten sam obiekt zostanie przekazany poprzez dwa różne argumenty wyjściowe. W takim przypadku działanie programu może być inne niż gdyby przekazano dwa różne obiekty. Metoda tropów w ujęciu [IMPK93, IKM+97] pozwala specyfikować programy, których działanie zależy od występowania aliasów wśród parametrów aktualnych. Ceną za taką możliwość jest konieczność wprowadzenia do metody mało intuicyjnych pojęć. Alternatywnym rozwiązaniem jest, postulowane w niniejszej pracy, ograniczenie liczby argumentów wyjściowych programu dostępu do jednego, które pozwala uniknąć wspomnianych problemów. Poniżej przedstawiamy szczegółowo przyczyny powstawania tych kłopotów, oraz rozważamy argumenty za i przeciw wymienionym rozwiązaniom.

Trop jest zapisem wszystkich wywołań dotyczących obserwowanego obiektu. W tropie T, wartość obiektu w chwili n-tego wywołania programu jest reprezentowana przez trop tego obiektu sprzed wywołania, czyli $n-1$ elementowy prefiks T. Ta obserwacja pozwala zastosować skrót notacyjny, polegający na zastąpieniu wartości obiektu obserwowanego w chwili wywołania przez specjalny symbol; w [PW89] użyto „$", zaś w [IMPK93] „$*$". Na przykład, trop[1] obiektu o nazwie ω

$$\boxed{\mathrm{PUSH}((\omega,\_),1)}.\mathrm{PUSH}((\omega,\boxed{\mathrm{PUSH}((\omega,\_),1)}),2)$$

zapiszemy jako

$$\mathrm{PUSH}((\omega,*),1).\mathrm{PUSH}((\omega,*),2)$$

Występowanie w tropie nazwy obserwowanego obiektu powoduje, że nie jest możliwe porównywanie tropów różnych obiektów. Również ten problem możemy rozwiązać za pomocą symbolu wprowadzonego poprzednio. Spośród wszystkich obiektów modułu wskazujemy jeden i spośród jego tropów wybieramy tropy kanoniczne. Zakładamy, że wszystkie obiekty w module są jednakowe i dlatego w specyfikacji wystarczy określić jak tropy tych obiektów redukują się do tropów kanonicznych wskazanego obiektu. Nie musimy w specyfikacji wymieniać wskazanego obiektu — gwiazdka oznaczać będzie nazwę (w przypadku argumentu wyjściowego) lub parę nazwa i stan tego obiektu (argument wejściowo-wyjściowy). Wówczas przykładowy trop zapiszemy następująco:

$$\mathrm{PUSH}(*,1).\mathrm{PUSH}(*,2)$$

W pracy magisterskiej [Kub94] zauważono, że trop powinien być niezależny nie tylko od nazwy obserwowanego obiektu, ale również od nazw innych obiektów, które pojawiły się w wywołaniach wymienionych w tym tropie. Jedyną istotną informacją jest tożsamość obiektów biorących udział w wywołaniu. Dlatego wprowadzono gwiazdki z indeksami na oznaczenie nazw obiektów innych niż obserwowany. Na przykład w wywołaniu $\mathrm{SPLIT}(*,*_1)$ pierwszym argumentem jest obiekt obserwowany, zaś drugim inny obiekt, różny od obserwowanego. Jako obydwa argumenty wywołania $\mathrm{SPLIT}(*,*)$ podano obiekt obserwowany. W raporcie metody

[1] Za pomocą ramek wyróżniono powtarzający się prefiks. Ramki nie są elementem tropu.

TAM'97 [IKM$^+$97] zastosowano się do propozycji zawartych w [Kub94] — w zapisie tropu występują gwiazdki z indeksami.

Takie rozwiązanie, choć matematycznie bez zarzutu, wprowadza do składni skomplikowane i nieintuicyjne pojęcią (zwykłe i indeksowane gwiazdki), które mogą zniechęcić do poznawania i używania metody. Dlatego proponujemy aby specyfikacje napisane w TAM 2000 nie zajmowały się nazwami obiektów. Cel ten osiągniemy dzięki ograniczeniu polegającemu na tym, że każdy program dostępu może mieć wpływ co najwyżej na jeden obiekt. Wówczas nie musimy zajmować się kwestią tożsamości przekazanych przez argumenty obiektów; nie ma też potrzeby wprowadzany indeksowanych gwiazdek na oznaczenie nazw innych (poza obserwowanym) obiektów biorących udział w wywołaniu. Jednak rezygnacja z uwzględniania nazw obiektów w specyfikacji ma też swoje wady. Nie możemy zdefiniować programów, których działanie zależy od istnienia aliasów wśród argumentów. Przykładami takich programów są CATENATE [IMPK93] i JOIN [IKM$^+$97].

Programy dostępu zwykle dzielimy na *konstruktory*, *modyfikatory* i *obserwatory* [GH93]. Konstruktory służą do budowania przestrzeni stanów. Modyfikatory zmieniają stan obiektu, ale nie wprowadzają go w stan niemożliwy do osiągnięcia za pomocą konstruktorów. Obserwatory dostarczają na zewnątrz obiektu informacje o stanie obiektu, nie zmieniając go. W TAM 2000 wyróżnimy tylko obserwatory i modyfikatory (w tym pojęciu mieszczą się również wcześniej rozważane konstruktory). Uważamy, że wydzielenie konstruktorów jest niepożądanym w specyfikacji interfejsu sugerowaniem przyszłej implementacji. Wśród modyfikatorów są programy generujące niedeterministyczne wyniki, ponieważ te wyniki mogą wpłynąć na przyszłe zachowanie obiektu. Na przykład, w module zarządcy nazwami, wynik przekazany przez PICK nie może być ponownie przekazany do czasu wywołania DISPOSE z tym wynikiem jako argumentem.

Obiekt, którego stan ma ulec zmianie, jest niejawnym argumentem wywołania modyfikatora. Ten pomysł, wzięty z obiektowych języków programowania, wraz z założeniem, że każdy program modyfikuje co najwyżej jeden obiekt, pozwala w pełni abstrahować od nazwy obiektu obserwowanego. Wystarczy założenie, że wszystkie wywołania obecne w tropie miały ten sam obiekt jako swój niejawny argument. Obserwatory traktujemy jak zwykłe funkcje, które przekazują jedną wartość. Takie programy nie mają niejawnych argumentów. Zaproponowane modyfikacje prowadzą do jeszcze jednego uproszczenia — niepotrzebne stają się deskryptory argumentów. Wyjście generowane przez wywołanie nie jest związane z żadnym argumentem. Fakt zmiany lub nie stanu obiektu specyfikowanego modułu jest zadany przez rodzaj programu (obserwator albo modyfikator).

Proponując ograniczenie liczby obiektów, które może zmienić jedno wywołanie programu dostępu, musimy zastanowić się, czy nie zmniejszamy w ten sposób siły wyrazu metody tropów. Po pierwsze warto zauważyć, że interfejs obiektu powinien być jak najbardziej przejrzysty — skuteczną receptą na osiągnięcie tego celu jest specyfikowanie programów wykonujących pojedyncze czynności i zwracających pojedyncze wyniki. Metoda tropów może być jednak stosowana także przez osoby, które nie uważają tej rady za słuszną.

Program deterministyczny zmieniający stan kilku obiektów można zastąpić oddzielnymi programami, których zadaniem będzie zmodyfikowanie stanu pojedynczych obiektów. Można tu stosować rozmaite konwencje, które pozwolą kilka takich atomowych programów zaimplementować za pomocą jednego. Na przykład, program SPLIT możemy zapisać jako dwa modyfikatory SPLIT_1 i SPLIT_2 opisujące zmiany odpowiednich argumentów oryginalnego programu. W artykule [IMD97] zbadano cykl rozwoju implementacji modułu i stwierdzono, że przekształcenie interfejsu opisanego w specyfikacji na interfejs programistyczny może być

niebanalnym zadaniem. Konwencje definiujące to przekształcenie nie są ogólnie przyjętymi normami i dlatego zwykle są tworzone na nowo dla każdego projektu. W [IMD97] pojawia się nowy dokument zwany *przewodnikiem projektu* (ang. *Project Guide*) zawierający m.in. wspomniane konwencje. Nie ma zatem przeszkód aby dodać to tego dokumentu reguły opisujące sposób łączenia programów zdefiniowanych w specyfikacji interfejsu w jedną procedurę w kodzie modułu.

To samo można uczynić dla programów, które przekazują niedeterministyczne wyniki. W pracy [IKM95a] udowodniono, że w metodzie tropów nie da się wyspecyfikować pewnych typów danych o niedeterministycznym działaniu. Dotyczy to programów niedeterministycznych modyfikujących w sposób skorelowany wiele obiektów definiowanego modułu. Nie tracimy zatem siły wyrazu metody, ograniczając działanie programów do modyfikacji jednego obiektu. W przypadku wyników generowanych przez program dostępu, rozdzielenie go na kilka części pozwala wyrazić więcej niż było to możliwe w TAM'97, gdzie wyniki przekazywane przez poszczególne argumenty wyjściowe musiały być od siebie niezależne. Przykładem może być tu program z dwoma wyjściami — dowolnymi liczbami, których różnica musi być równa jeden. Po rozdzieleniu na dwa programy, oryginalne wywołanie jest reprezentowane w tropie jako dwa wywołania. Drugi wynik jest generowany na podstawie historii poprzednich wywołań, można zatem uzależnić go od wyniku pierwszego wywołania. Dzięki temu osiągamy efekt niewyrażalny w TAM'97.

4.2 Hermetyzacja

W specyfikacji tropowej wymieniamy listę *typów obcych*. Są to typy, których semantyka ma wpływ na typ definiowany w rozważanej specyfikacji. Niektóre pojęcia wprowadzone przez specyfikację typu obcego mogą pojawiać się w tekście bieżącej specyfikacji. Uważamy, że w TAM'97 lista pojęć dostępnych w ten sposób jest zbyt szeroka. Są na niej obecne funkcje pomocnicze, programy dostępu, predykat *canonical*, funkcja rozszerzenia i relacja wyjścia. Proponujemy tę listę ograniczyć wyłącznie do programów dostępu. To rozwiązanie jest zgodne z założeniem o pełnej hermetyzacji modułu.

TAM'97 pozwala na korzystanie z dwóch zbiorów tropów typu obcego: wszystkie tropy i tropy kanoniczne. Wprowadzono dodatkowe symbole na oznaczenie tych zbiorów, odpowiednio: <<typ>> i <typ>. Zapisane w specyfikacji wywołanie programu dostępu obcego modułu może być elementem tropu lub wyrażeniem strzałkowym, którego wartość jest wynikiem wygenerowanym przez to wywołanie. Strzałka „$\searrow$" wskazuje argument wyjściowy, który ma być wyliczony jako wartość takiego wyrażenia. Na przykład:

$$\mathrm{PUSH}(*, 3)$$

jest elementem tropu, zaś

$$\mathrm{PUSH}((*_1, \_) \searrow, 3)$$

oblicza wartość przekazaną przez pierwszy argument.

W TAM 2000 specyfikacja korzysta z typów obcych nie zakładając, że zostały one zdefiniowane w metodzie tropów. Jedyne wymagania to istnienie zbioru stanów abstrakcyjnych (tropy kanoniczne dla specyfikacji tropowej), oraz programów dostępu jako środka komunikacji z obiektami. Dzięki takiemu rozwiązaniu możemy odwoływać się do modułów zdefiniowanych za pomocą innych metod specyfikacji.

Szczelna hermetyzacja, wraz z ograniczeniem do jednego liczby wyników generowanych przez pojedynczy program dostępu (por. p. 4.1), pozwala na znaczne uproszczenie metody tropów. Dzięki tym zmianom wyeliminujemy niestandardowe symbole: strzałki i nawiasowe oznaczenia zbiorów tropów (<...> i <<...>>). Liczba pojęć zostanie zmniejszona, ponieważ dla typu obcego będziemy rozważać jedynie jego zbiór wartości.

W TAM 2000 odczytanie wartości przekazanej przez programy dostępu typu obcego jest bardzo proste i intuicyjne. Wywołanie obserwatora traktujemy jak użycie klasycznej funkcji algebraicznej, np.

$$\mathrm{TOP}(T)$$

to wartość przekazana przez wywołanie programu TOP na stosie w stanie T. Trudniej jest z modyfikatorami ze względu na ich niejawne argumenty. Użyjemy średnika do oddzielenia argumentu niejawnego od jawnych, np.

$$\mathrm{PUSH}(T; 3)$$

oznacza stan stosu po wykonaniu na nim operacji PUSH(3), w chwili gdy był w stanie T. Uważamy, że takie wyrażenia są bardziej intuicyjne niż ich odpowiedniki z TAM'97:

$$\mathrm{TOP}(T) \searrow \quad \text{i} \quad \mathrm{PUSH}((*_1, T) \searrow, 3)$$

Konsekwencją wprowadzenia proponowanej szczelnej hermetyzacji jest wymaganie, aby obie relacje równoważności tropów były sobie równe (por. p. 3.3). Innymi słowy, każdy trop kanoniczny musi być obserwacyjnie odróżnialny od pozostałych tropów kanonicznych. Bez tego wymagania nie można sensownie korzystać z równości na wartościach typu obcego, ponieważ ta sama wartość abstrakcyjna mogłaby mieć wielu reprezentantów wśród tropów kanonicznych. Dzięki nałożeniu na specyfikacje takiego ograniczenia otrzymujemy metodę bardziej abstrakcyjną — zbiór możliwych implementacji jest największy z możliwych. Specyfikacja, w której równoważność specyfikacyjna nie jest równa obserwacyjnej, ma mniejszy zbiór implementacji, ponieważ stany konkretnej struktury danych muszą być odwzorowane przez funkcję abstrakcji *na* zbiór tropów kanonicznych. Stąd wynika, że obserwacyjnie równoważne, różne tropy kanoniczne muszą mieć różną reprezentację w implementacji. To wyklucza implementacje, w których te dwa równoważne tropy kanoniczne mają tę samą reprezentację.

4.3 Dopasowywanie wzorca

Dopasowywanie wzorca jest bardzo naturalną techniką wykorzystywaną przy tworzeniu specyfikacji tropowych. Zostało to zaobserwowane m.in. w pracach [WP92, Wan94b], które były poświęcone tworzeniu prototypów specyfikacji tropowych. W TAM'97 występuje wyrażenie *where*, za pomocą którego można dokonać rozkładu tropu na podtropy według zadanego szablonu, np. w specyfikacji stosu (wg TAM'97):

$$\mathrm{POP}((n, T) \searrow) = T_1 \ \textit{where}\ T_1 : \text{<stack>}; x : \text{<int>}[T = T_1.\mathrm{PUSH}(x)]$$

To samo można znacznie prościej zapisać za pomocą dopasowywania wzorca:

$$\mathrm{POP}((n, T.\mathrm{PUSH}(x)) \searrow) = T$$

Zapis jest znacznie krótszy i bardziej intuicyjny. Zmienne występujące w wzorcu są wprowadzane przez sam wzorzec i nie potrzebują osobnej deklaracji. Dlatego w TAM 2000 zastępujemy wyrażenie *where* przez dopasowywanie wzorca w argumentach funkcji. Przy czym zakładamy, że dopasowanie dokonuje się w kolejności deklaracji poszczególnych wzorców, tak jak to jest w wielu językach funkcyjnych (np. w SML [MCP93]). Przyjrzyjmy się definicji predykatu *canonical* w specyfikacji stosu przedstawionej w dodatku D:

$$canonical([\mathrm{PUSH}(x_i)]_{i=1}^{s}) \Leftrightarrow \textbf{true}$$

$$canonical(other) \Leftrightarrow \textbf{false}$$

Powyższa definicja jest poprawna, pomimo tego, że zmienna *other* w drugim równaniu może być dopasowana do każdego tropu. Drugie równanie będzie brane pod uwagę tylko wtedy, gdy nie uda się dopasowanie do wzorca w pierwszym równaniu.

Wyeliminujemy z metody tropów (TAM'97) także konstrukcję rozszerzającą na sąsiednie wyrażenie, zasięg zmiennych związanych kwantyfikatorem $\exists!$. Mogła ona występować w tabelkach, np.:

$\mathrm{POP}((n,T)\searrow) =$

Condition	Value
$T = \_$	T
$\exists! T_1 : \text{<stack>}; x : \text{<int>}[T = T_1.\mathrm{PUSH}(x)]$	T_1

W drugim wierszu tabeli zasięg zmiennej T_1 wykracza poza wyrażenie pod kwantyfikatorem i obejmuje także wyrażenie z sąsiedniej kolumny z tego samego wiersza. Takie wyrażenia istotnie odbiegają od standardowej semantyki kwantyfikatorów i przez to mogą utrudnić posługiwanie się metodą tropów. Można zamiast nich używać dopasowywania wzorca. Z tych powodów tabelki, w których zasięg pewnych zmiennych jest szerszy niż zasięg wprowadzającego je kwantyfikatora, nie będą występowały w TAM 2000.

4.4 Tabelki

W raporcie [Par92] sklasyfikowano rozmaite formy notacji tabelarycznej. Jedynym typem tabeli występującym w składni TAM'97 jest jednowymiarowa tabela funkcyjna (wg systematyki z [Par92]), która ma pewne dodatkowe własności (por. p. 4.3). Występowanie tych specjalnych cech utrudniało rozszerzanie metody tropów o nowe rodzaje tabel. Ich wyeliminowanie sprawia, że wprowadzanie do metody dowolnych form notacji tabelarycznej jest bardzo naturalne. Jedyny wymaganiem jest dobra definicja ich semantyki[2].

W tym kontekście bardzo ważną pracą jest [ORS95], w której pokazano sposób korzystania z tabel w ramach systemu PVS [OSR93]. PVS jest narzędziem do tworzenia i weryfikacji dowodów, który z powodzeniem wykorzystano do dowodzenia własności specyfikacji tropowych [Kre98, EKMD98]. Praca [Eng99] szczegółowo bada podobieństwa i różnice pomiędzy semantyką języka wbudowanego w PVS, a semantyką metody tropów.

4.5 Sygnały

Ewolucję pojęcia zmiennych wejściowych i wyjściowych przedstawiliśmy w punkcie 3.5. W opisanych tam wersjach metody tropów zdarzenia wejściowe były wyłącznym sposobem od-

[2]Składnia TAM 2000 zdefiniowana w rozdziale 6 zawiera jedynie jednowymiarowe tabele funkcyjne, jednak jej konstrukcja pozwala na bardzo łatwe rozszerzenie o inne typy tabel.

czytywania wartości zmiennych wejściowych. Obiekt nie mógł zatem korzystać z nich bezpośrednio, poprzez odczyt ich obecnych wartości. Takie podejście powoduje, że sposób przekazywania danych wejściowych przez te zmienne jest taki sam jak dla argumentów wejściowych programów dostępu. Jedyną różnicę stanowi możliwość definiowania, które zmiany wartości zmiennych wejściowych generują zdarzenie, a które nie. W szczególności w raporcie [IKM+97] przy definiowaniu zdarzenia wyjściowego można brać pod uwagę wartość tylko jednej zmiennej wejściowej, zaś w pracy [BIMO94] nie ma nawet tej możliwości — każda zmiana wywołuje zdarzenie. Warto również przyjrzeć się formułom, które definiują zdarzenia. Nigdy nie brano pod uwagę uzależnienia ich od czasu, co ogranicza twórcę specyfikacji i jest niezgodne z duchem podejścia funkcyjnego [PM95], od którego rozpoczęliśmy nasze rozważania (rozdział 1). W tym podejściu dziedzinami definiowanych relacji są funkcje zależne od czasu. Każdy obiekt wyspecyfikowany w metodzie tropów reaguje jedynie na bodźce zewnętrzne, nie ma więc możliwości zdefiniowania zegara, który generuje sygnał co określony kwant czasu.

Brak możliwości satysfakcjonującego traktowania czasu i duże podobieństwo do programów dostępu to podstawowe przyczyny, dla których w TAM 2000 rezygnujemy ze zmiennych wejściowych. Gdy potrzebujemy wyspecyfikować moduł, którego działanie zależy od upływającego czasu, lepiej jest skorzystać z formalizmów do tego przystosowanych. Może to być np. rachunek trwania [EKM+93] powstały w ramach podejścia funkcyjnego [PM95].

W oryginalnej metodzie tropów zmienne wyjściowe stanowiły mechanizm dualny dla zmiennych wejściowych. W TAM 2000 obiekt zamiast nadawania wartości zmiennym wyjściowym, będzie generował *sygnały*. Każdy sygnał ma nazwę i pewną liczbę argumentów. Wystąpienie sygnału zawiera wartości dla wszystkich tych argumentów. W specyfikacji definiujemy funkcję *signals*, która dla każdego wywołania programu dostępu określa listę generowanych sygnałów.

Ten pomysł pojawił się już w pracach [Hof85, Nor95]. Do zalet takiego rozwiązania należy zaliczyć jego komplementarność w stosunku do programów dostępu — sygnały generowane przez dany obiekt mogą zostać sprzężone z modyfikatorami innego obiektu. Eliminując zmienne wejściowe i wyjściowe upraszczamy metodę poprzez istotnie zmniejszenie liczby pojęć. Wychodzące sygnały pozwalają na jeszcze jedno uproszczenie opisane w punkcie 4.6.

W pracy [BIMO94] opisano istotne wady zmiennych wyjściowych polegające na tym, że dane przez nie są przekazywane bez przerwy. Po pierwsze, czasem zachodzi potrzeba nadania zmiennej wyjściowej wartości, o której mamy pewność, że nie wywoła zdarzenia w sprzężonym obiekcie. To prowadzi do nieakceptowalnej ingerencji w interfejs innego modułu. Gdy posługujemy się sygnałami, nie ma takiego problemu — po prostu nie jest wysyłany sygnał. Po drugie, występuje kłopot, gdy chcemy w sprzężonym obiekcie wywołać zdarzenie poprzez przekazanie przez zmienną wyjściową tej samej wartości co poprzednio. W takiej sytuacji musimy stworzyć dodatkową zmienną wyjściową typu logicznego, która zmieniając swoją wartość na przeciwną poinformuje sprzężony obiekt o zajściu zdarzenia. Jak widać możemy sobie poradzić, ale znów kosztem ingerencji w interfejs innego modułu. Gdy posługujemy się sygnałami, po prostu wysyłany jest kolejny sygnał niosący tę samą wartość.

4.6 Legalność wywołań

Jeśli użytkownik korzysta z modułu w sposób właściwy, jedynie *legalne* wywołania programów dostępu powinny mieć miejsce. W specyfikacji tropowej rozdzielamy je od wywołań nielegalnych. Począwszy od raportu [PW89] służą do tego *żetony statusu* — otoczone znakami „%” napisy informujące o błędach wywołania. Wyróżniony żeton %legal% jest przypisany do wy-

wołań legalnych.

Znaczenie żetonów statusu w implementacji modułu nigdy nie zostało zdefiniowane. W pracach o projekcie implementacji to pojęcie w ogóle nie występuje. W raporcie [IMPK93] do każdego programu w przykładowej specyfikacji dodano argument przekazujący tę samą informację, którą niosły żetony. Dzięki temu zabiegowi, w implementacji pojawiła się kontrola błędów, lecz nie miała nic wspólnego z żetonami.

Biorąc pod uwagę trudności powodowane przez żetony, TAM 2000 wprowadza w ich miejsce predykat *legal*, który wskazuje czy wywołanie było legalne czy nie. Wprowadzony do TAM 2000 mechanizm sygnałów (por. p. 4.5) pozwala w bardzo elegancki sposób przekazywać informację o przyczynie błędu. Funkcja *signals* definiująca sygnały wysyłane w odpowiedzi na wywołania jest określona także dla wywołań nielegalnych. Sygnały mają argumenty, co stanowi ich ważną zaletę w stosunku do żetonów — mogą przekazać dodatkowe informacje poza swoją nazwą.

4.7 Niejawna wzajemna rekurencja

W TAM'97 wszystkie definicje mogą być wzajemnie rekurencyjne, np.: predykat *canonical* definiuje dziedziny funkcji pomocniczych, podczas gdy tych funkcji użyto w definicji tego predykatu; w zapisie funkcji rozszerzenia i relacji wyjścia można odwoływać się do tych dwóch pojęć poprzez wyrażenia strzałkowe; oczywiście, również same funkcje pomocnicze mogą być wzajemnie rekurencyjne. Takie zależności nie są nigdzie deklarowane.

Proponujemy wykluczyć z TAM 2000 wzajemną rekurencję, ponieważ zmniejsza ona przejrzystość specyfikacji. W sekcji CANONICAL TRACES kolejność definicji będzie następująca: najpierw funkcje, które posłużą do definicji predykatu *canonical*, potem tenże predykat, a następnie pozostałe funkcje pomocnicze. Predykat *canonical* jest niezwykle ważnym elementem specyfikacji, zatem jego definicja powinna być wyraźnie wyróżniona za pomocą środków syntaktycznych. Poszczególne definicje mogą korzystać jedynie z definicji, które pojawiły się wcześniej.

W TAM 2000 definicje *legal*, *feasible*, *transition* i *signals* nie mogą być rekurencyjne i nie mogą nawzajem się do siebie odwoływać. Jedynym wyjątkiem jest tu funkcja *signals*, która może w swej definicji korzystać z predykatu *legal*. Wynika to z faktu, że *signals* musi być określona także dla nielegalnych wywołań.

Takie ograniczenia nie zmniejszają siły wyrazu metody — funkcje rekurencyjne można definiować jako funkcje pomocnicze i potem się do nich odwoływać w *legal*, *feasible*, *transition* i *signals*. Korzyść z wprowadzenia tej zmiany polega na wymuszeniu pisania bardziej przejrzystych i uporządkowanych, a przez to łatwiejszych do zrozumienia specyfikacji.

4.8 Specyfikacje sparametryzowane

Możliwość parametryzacji jest bardzo przydatnym udogodnieniem, ponieważ pozwala wielokrotnie wykorzystywać ten sam tekst, np. specyfikacji lub procedury. To udogodnienie występuje również w metodzie tropów — specyfikacje mogą być parametryzowane typem lub wartościami ustalonego typu.

W TAM'97 specyfikacja podaje jedynie nazwę parametru formalnego dla każdego parametru typowego. Jednocześnie w treści specyfikacji można odwoływać się do programów dostępu i funkcji pomocniczych zdefiniowanych przez te typy — milcząco zakładamy, że typ będący

parametrem aktualnym będzie takie programy udostępniał. Uważamy, że określenie jedynie nazwy parametru typowego nie jest wystarczające, ponieważ osoba pragnąca wykorzystać specyfikację sparametryzowaną musi zanalizować ją całą w poszukiwaniu odwołań do programów dostępu typu-parametru, aby stwierdzić czy dany typ może stać się odpowiednim argumentem formalnym tej specyfikacji.

W TAM 2000 wyeliminujemy tę niedogodność — dla każdego parametru typowego specyfikacji należy określić jakie programy dostępu musi mieć typ będący odpowiednim parametrem aktualnym. Na przykład, zarządca nazwami (por. dodatek F) ma parametr, który wskazuje typ zarządzanych nazw. Ten typ musi definiować program „$\leq$". Fakt ten jest podany w chwili wprowadzania argumentu — nie trzeba już analizować całej specyfikacji, aby stwierdzić czy dany typ może stać się jej argumentem formalnym.

W punkcie 4.2 wspomniano, że TAM 2000 umożliwia łączenie metod specyfikacji. Na przykład, typ wyspecyfikowany w pewnej metodzie algebraicznej może być parametrem aktualnym specyfikacji tropowej, jeśli odpowiedni parametr formalny wprowadza jedynie obserwatory — w specyfikacjach algebraicznych wszystkie programy są funkcjami i odpowiadają obserwatorom w TAM 2000. Oczywiście możliwe są także interpretacje specyfikacji algebraicznych, w których niektóre funkcje są traktowane jako modyfikatory.

Rozdział 5

Wprowadzenie do metody TAM 2000

W niniejszym rozdziale przedstawiono podstawe pojęcia metody TAM 2000. Rozdział jest przeznaczony głównie dla osób, które nigdy nie zetknęły się z metodą tropów. Osoby znające poprzednie wersje tej metody również powinny zapoznać się z treścią tego rozdziału, ponieważ zmiany zaproponowane w rozdziale 4 powodują pewne modyfikacje podstaw metody. Przedstawione poniżej wprowadzenie do TAM 2000 jest oparte na pierwszej części raportu [IMS94] i drugim rozdziale pracy magisterskiej [Ste95].

5.1 Obiekt

Metoda tropów zajmuje się specyfikowaniem obiektów. Dowolny byt nazywamy *obiektem* wtedy i tylko wtedy, gdy spełnia on następujące warunki:

- przyjmuje stany, może być poddany działaniu zdarzeń i w odpowiedzi na zdarzenia może wytwarzać wyniki[1],
- w każdej chwili jego stan jest jednoznacznie określony,
- może zmienić stan tylko w wyniku zdarzenia,
- wyniki mogą być wytwarzane tylko w odpowiedzi na zdarzenia,
- wynik związany z pewnym zdarzeniem jest wytwarzany natychmiast po zajściu tego zdarzenia,
- jego nowy stan oraz wytworzony wynik zależą jedynie od poprzedniego stanu i zdarzenia, w wyniku którego nastąpiła zmiana stanu.

Dodatkowo zakładamy, że w każdej chwili może zajść co najwyżej jedno zdarzenie.

5.2 Podstawowe pojęcia

Metoda tropów opisuje obiekty z punktu widzenia *zewnętrznego obserwatora*, który może zauważyć jedynie zdarzenia oddziaływujące na obiekty i wyniki wytwarzane w odpowiedzi

[1] Stany, zdarzenia i wyniki traktujemy jako pojęcia pierwotne.

na te zdarzenia. Przedstawimy teraz, jak za pomocą metody tropów można wyspecyfikować pewien ustalony obiekt.

Niech $\mathcal{E}$ oznacza zbiór wszystkich zdarzeń dla opisywanego obiektu, zaś $\mathcal{R}$ — zbiór wszystkich wyników, które mogą zostać wytworzone przez ten obiekt. *Tropem* nazywamy każde słowo nad alfabetem $\mathcal{E} \times \mathcal{R}$. Tropy zapisujemy w następującej postaci:

$$E_1{\uparrow}O_1.E_2{\uparrow}O_2.\ldots.E_n{\uparrow}O_n$$

gdzie $E_i \in \mathcal{E}$ i $O_i \in \mathcal{R}$ dla $i = 1, 2, \ldots, n$, a kropka to łącznik składowych.

Przyjmujemy przy tym, że kropka jest również operatorem konkatenacji tropów. Słowo puste nazywamy *tropem pustym* i oznaczamy przez "_". Jest to element neutralny operacji ".".

Za pomocą tropów pragniemy opisać zewnętrznie obserwowalne zachowanie obiektu, ale nie każdy trop może temu służyć. Aby rozróżnić tropy zgodnie z tym kryterium, definiujemy tzw. sensowność tropów. Jeśli dla każdego $i \in \{1, 2, \ldots, n\}$ obiekt może wytworzyć wynik O_i w odpowiedzi na zdarzenie E_i po sekwencji zdarzeń i wyników $E_1{\uparrow}O_1.E_2{\uparrow}O_2.\ldots.E_{i-1}{\uparrow}O_{i-1}$, to $E_1{\uparrow}O_1.E_2{\uparrow}O_2.\ldots.E_n{\uparrow}O_n$ nazywamy *tropem sensownym*.

Jeśli T i $T.S$ są tropami sensownymi, to S nazywamy *sensownym rozszerzeniem* T, a zbiór wszystkich sensownych rozszerzeń T — *zachowaniem obiektu po* T.

Korzystając z powyżej opisanych pojęć, wśród tropów sensownych definiujemy pewną relację oznaczaną przez "≡" i nazywaną *obserwacyjną równoważnością tropów*. Dwa tropy sensowne T_1, T_2 są *obserwacyjnie równoważne* jedynie, jeśli zachowanie obiektu po T_1 jest równe zachowaniu obiektu po T_2. Innymi słowy $T_1 \equiv T_2$ jedynie, jeśli T_1 i T_2 mają te same sensowne rozszerzenia.

Zauważmy, że jeśli znamy relację "≡", to opis zachowania obiektu po pewnym tropie sensownym T określa również zachowanie tego obiektu po wszystkich tropach obserwacyjnie równoważnych T. Fakt ten ma bardzo duże znaczenie dla metody tropów.

5.3 Specyfikacja tropowa modułu

Moduł implementuje pewną liczbę niezależnych od siebie, identycznych obiektów. Identyczne obiekty mają te same zbiory tropów sensownych. Nie tracąc ogólności możemy zatem przyjąć, że w opisywanym module jest tylko jeden obiekt, nazywany *obiektem wzorcowym*.

Opiszemy teraz, jak ogólne pojęcia związane z obiektami (por. p. 5.1) pojawiają się w metodzie tropów. Komunikacja pomiędzy obiektem a światem zewnętrznym odbywa się jedynie za pomocą zbioru programów, które mogą być używane przez inne obiekty, w celu pobrania pewnych informacji z obiektu i/lub zmiany stanu tego obiektu. Takie programy nazywamy *programami dostępu*.

Zbiór zdarzeń, których działaniu może być poddany obiekt, jest podzbiorem wszystkich *wywołań programów dostępu*. Elementy tego podzbioru nazywamy *wywołaniami legalnymi*. Jeśli użytkownik korzysta z obiektu w sposób właściwy, wywołania spoza tego podzbioru nie powinny mieć miejsca (np. nie należy wywoływać programu POP na pustym stosie).

Wyniki wytwarzane w odpowiedzi na zdarzenia, to przekazywane wartości i wysyłane sygnały. *Sygnał* składa się z nazwy i listy wartości. W odpowiedzi na pojedyncze wywołanie może zostać przekazana co najwyżej jedna wartość i dowolna liczba sygnałów.

Zgodnie z p. 5.2, w specyfikacjach tropowych opisujemy obiekty z punktu widzenia obserwatora zewnętrznego — powinniśmy więc określić wszystkie dopuszczalne wyniki obserwacji,

czyli tropy sensowne. Zwykle nie jest to jednak łatwe zadanie: już dla bardzo prostych przykładów formuły charakteryzujące takie tropy są skomplikowane. Autorzy metody tropów stworzyli inny, prostszy sposób opisu obiektów, który jest oparty na ostatnim spostrzeżeniu z p. 5.2. Okazuje się również, że pełna definicja relacji obserwacyjnej równoważności nie jest konieczna. Wystarczy opisać tylko pewien niewielki jej podzbiór.

Proces specyfikacji obiektu składa się z następujących etapów:

1. Wybieramy pewien podzbiór tropów, nazywając je *kanonicznymi*. Oznaczmy ten pozbiór przez $\mathcal{C}$. Będziemy opisywać zachowanie obiektu tylko po tropach kanonicznych.

2. Definiujemy zbiór $legal \subseteq \mathcal{C} \times \mathcal{E}$. $\langle T, E\rangle \in legal$, oznacza, że gry tropem obiektu jest T, wywołanie E jest legalne.

3. Definiujemy *relację wyjścia*, *out*; jest to podzbiór produktu $legal \times \mathcal{R}$. Określamy w ten sposób jakie wyniki mogą być wytworzone przez obiekt w odpowiedzi na zdarzenie E, gdy dotychczas zaobserwowano ciąg zdarzeń i wyników opisany tropem T. Zarazem jest to definicja zbioru sensownych, jednozdarzeniowych rozszerzeń tropów kanonicznych.

4. Aby opisać wspomniany powyżej podzbiór relacji obserwacyjnej równoważności definiujemy *funkcję rozszerzenia*, *transition*, która przekształca sensowne jednozdarzeniowe rozszerzenia tropów kanonicznych na tropy kanoniczne. Jeśli z definicji *transition* wynika, że dla pewnego tropu kanonicznego T, zdarzenia E i wyniku O takich, że $\langle T, E, O\rangle \in out$, zachodzi $transition(T, E, O) = S$, to ze specyfikacji wynika, że $T.E{\uparrow}O$ jest obserwacyjnie równoważny tropowi kanonicznemu S.

Zapis obserwacji obiektu, który dotychczas nie był poddany działaniu żadnych zdarzeń, to trop pusty; jeśli nie jest on kanoniczny, to na podstawie relacji wyjścia i funkcji rozszerzenia nie można określić reakcji tego obiektu na pierwsze zdarzenie. W takiej sytuacji należy w specyfikacji określić trop kanoniczny, który ma być obserwacyjnie równoważny tropowi pustemu.

Funkcja rozszerzenia i definicja tropu kanonicznego równoważnego tropowi pustemu pozwalają redukować tropy do obserwacyjnie równoważnych im tropów kanonicznych (szerzej będzie o tym mowa w p. 5.5). Na podstawie relacji *out* przewidujemy wyniki wytwarzane przez obiekt w odpowiedzi na zdarzenia, gdy zapis obserwacji jest tropem kanonicznym. Umiemy zatem określić jakie wyniki może wytworzyć dany obiekt w każdej chwili obserwacji.

Specyfikacja tropowa modułu składa się z nazwy modułu i opisu obiektu wzorcowego, który zawiera:

- listę programów dostępu wraz z opisem ich argumentów i przekazywanych wyników;
- definicję predykatu charakterystycznego zbioru tropów kanonicznych oraz definicję tropu kanonicznego równoważnego tropowi pustemu, jeśli ten nie jest kanoniczny,
- definicję zbioru wywołań legalnych,
- definicję relacji wyjścia,
- definicję funkcji rozszerzenia.

5.4 Klasyfikacja programów dostępu i determinizm wyników

Niektóre z wytwarzanych wyników mogą być jednoznacznie wyznaczone przez dotychczasową historię (trop) obiektu; takie wyniki nazwiemy *deterministycznymi*. Zauważmy, że w zapisie tropu występującym w specyfikacji nie ma potrzeby umieszczania takich wyników, ponieważ można je wywnioskować z definicji relacji *out*, która w tym przypadku jest funkcją. Zakładamy, że sygnały są zawsze wysyłane w sposób deterministyczny. Natomiast wartości przekazywane przez programy dostępu mogą być zarówno deterministyczne, jak i niedeterministyczne.

Programy dostępu dzielimy na *modyfikatory*, które mogą zmienić stan obiektu, i *obserwatory*, które nie zmieniają stanu obiektu. Jedynym zadaniem obserwatorów jest przekazanie na zewnątrz obiektu informacji o jego stanie. Wyniki wytwarzane przez obserwatory nie mogą być niedeterministyczne, ponieważ taki wynik mógłby mieć wpływ na przyszłe zachowanie obiektu, czyli mógłby prowadzić do modyfikacji stanu obiektu.

Rozważmy moduł implementujący zarządcę nazw, o którym była mowa na początku rozdziału 4. Przypomnijmy, że obiekty tego modułu administrują pewną pulą wartości, które mogą być wykorzystywane jako nazwy. Każdy obiekt ma dwa programy dostępu: PICK rezerwuje pewną nazwę z zarządzanej puli i przekazuje ją wywołującemu; DISPOSE powoduje zwolnienie nazwy podanej jako argument i dołączenie jej do puli dostępnych nazw. Wynik wytwarzany przez PICK jest niedeterministyczny — przekazana nazwa może być dowolną z puli. Zauważmy, że wywołanie PICK *musi* prowadzić do modyfikacji stanu obiektu, ponieważ, ta sama nazwa nie może zostać ponownie przekazana do czasu jej zwolniena poprzez wywołanie programu DISPOSE. Z powyżej wymienionych powodów traktujemy programy przekazujące niedeterministyczny wynik jako modyfikatory.

Relacja *out* w przypadku deterministycznego wyniku jest funkcją. Wykorzystamy to, aby uprościć zapis specyfikacji. Relację *out* definiujemy poprzez określenie trzech funkcji:

- $signals$, która określa listę sygnałów wysyłanych w odpowiedzi na wywołanie programu;
- $result$, która określa wartość przekazywaną przez wywołanie obserwatora;
- $feasible$, która określa zbiór poprawnych wartości jakie może wytworzyć wywołanie modyfikatora. Faktyczny wynik przekazany przez wystąpenie takiego wywołania musi być elementem tego zbioru.

Upraszczamy również definicję funkcji $transition$ — definiujemy ją tylko dla modyfikatorów, ponieważ obserwator nie może zmienić stanu obiektu.

5.5 Funkcja redukująca

Z definicji zbioru tropów kanonicznych, zbioru $legal$, funkcji rozszerzenia ($transition$) i relacji wyjścia (out), można wywnioskować, które tropy są sensowne. Aby uzasadnić to stwierdzenie, zdefiniujemy predykat charakterystyczny zbioru tropów sensownych ($feasibleTrace$) i *funkcję redukującą* (r), która przekształca tropy sensowne na obserwacyjnie równoważne im tropy kanoniczne. W poniższych wzorach $\Theta = \_$, gdy trop pusty jest kanoniczny, zaś w przeciwnym przypadku Θ oznacza określony w specyfikacji trop kanoniczny równoważny tropowi pustemu.

$$feasibleTrace(\_) \wedge feasibleTrace(\Theta)$$

$$r(\_) = \Theta$$

$$feasibleTrace(T.E{\uparrow}O) \Leftrightarrow feasible(T) \wedge \langle T, E\rangle \in legal \wedge \langle r(T), E, O\rangle \in out$$

$$r(T.E{\uparrow}O) = transition(r(T), E, O)$$

Powyższe równania definiują $feasibleTrace$ i r poprzez równoczesną indukcję ($feasibleTrace$ definiuje dziedzinę r). Przypomnijmy, że zbiór tropów sensownych jest podstawą definicji obserwacyjnej równoważności tropów (p. 5.2).

Zbiór tropów kanonicznych nie może być dowolny — dwa różne tropy kanonicznie nie mogą być obserwacyjnie równoważne:

$$\forall\ T_1, T_2 \in \mathcal{C}\ \ (T_1 \not\equiv T_2)$$

Ta własność zbioru $\mathcal{C}$ implikuje, że każdy trop kanoniczny musi być punktem stałym funkcji redukującej:

$$\forall\, T \in \mathcal{C}\ (r(T) = T)$$

Udowodnimy prawdziwość tej implikacji. Załóżmy, że trop kanoniczny T nie jest punktem stałym r. Wartością funkcji redukującej jest trop obserwacyjnie równoważny argumentowi, czyli T jest równoważny $r(T)$. Przeciwdziedziną r jest zbiór tropów kanonicznych. Znaleźliśmy zatem dwa obserwacyjnie równoważne tropy kanoniczne (T i $r(T)$). Otrzymana sprzeczność dowodzi, że każdy trop kanoniczny musi być punktem stałym funkcji redukującej.

Poniżej przedstawiono przykład, w którym naiwne podejście do definicji zbioru tropów kanonicznych prowadzi do niepoprawnej specyfikacji.

Przykład Obserwowanym obiektem jest stos liczb naturalnych z następującymi programami dostępu:

modyfikator	PUSH(a)	kładzie a na wierzchołku stosu
modyfikator	POP	usuwa wierzchołek stosu lub nic nie zmienia gdy stos jest pusty
obserwator	TOP	przekazuje wartość wierzchołka stosu lub -1 gdy stos jest pusty

Najbardziej naturalnymi tropami kanonicznymi są ciągi zbudowane z dowolnych wywołań programu PUSH. Takie sekwencje precyzyjnie opisują zawartość stosu i żadne dwa tropy kanoniczne nie są sobie równoważne.

Zmodyfikujmy nieznacznie nasz przykład — teraz program TOP powinien zwrócić resztę z dzielenia wierzchołka stosu przez 256. Specyfikacja takiego stosu może różnić się od poprzedniej tylko opisem relacji wyjścia. Wówczas tropy PUSH(5) i PUSH(261) są kanoniczne i zarazem obserwacyjnie równoważne — jedyny program produkujący wyniki (TOP) zwróci w obu przypadkach 5. Taka specyfikacja jest niepoprawna. Należy ją poprawić, wybierając inaczej tropy kanoniczne — powinny być to ciągi wywołań PUSH(x), gdzie $0 \leq x \leq 255$:

$$transition(T, \mathrm{PUSH}(x)) = T.\mathrm{PUSH}(x\ \mathbf{mod}\ 256)$$

Rozdział 6

Prezentacja metody TAM 2000

W niniejszym rozdziale przedstawimy metodę TAM 2000. Zaprezentujemy składnię specyfikacji tropowych i wymienimy kontekstowe warunki poprawności. Znaczenie każdej konstrukcji syntaktycznej zostanie wyjaśnione w języku naturalnym i zilustrowane fragmentami przykładowych specyfikacji, o których była mowa na początku rozdziału 4. Struktura prezentacji jest wzorowana na raporcie TAM'97 [IKM+97]. Rozdział 7 zawiera formalną definicję metody TAM 2000.

Składnia prezentacyjna Specyfikacje w TAM 2000 można zapisać za pomocą znaków standardu ASCII. To pozwala przesyłać specyfikacje za pomocą poczty elektronicznej bez żadnych kłopotów związanych z kodowaniem. Ułatwia to też konstruowanie narzędzi wspomagających stosowanie metody — przy ich tworzeniu można skorzystać np. ze standardowych analizatorów składniowych. Jednak często pragniemy przedstawić specyfikację tropową w bardziej estetyczny sposób niż tylko za pomocą znaków ASCII. Z tego powodu dla niektórych konstrukcji definiujemy również tzw. składnię prezentacyjną. Pozwala ona korzystać z udogodnień udostępnianych przez programy wspomagające przygotowywanie dokumentów. Dobór kroju czcionki i wyrównanie tekstu są przykładami takich udogodnień. Pozwalają one na opracowanie estetycznego dokumentu, nie mając przy tym wpływu na semantykę specyfikacji. Innym udogodnieniem może być formatowanie tabel, które są bardzo ważnym elementem metody tropów.

Postać składni prezentacyjnej zostanie zilustrowana fragmentami specyfikacji zapisanych zgodnie z tą składnią.

Metasymbole Użyjemy następujących metasymboli:

$x \mid y$ oznacza wybór pomiędzy x i y;

$[\![x]\!]^*(y)$ oznacza listę (być może pustą) złożoną z wystąpień x pooddzielanych za pomocą y;

$[\![x]\!]^+(y)$ oznacza niepustą listę złożoną z wystąpień x pooddzielanych za pomocą y;

W dwóch ostatnich metasymbolach można pominąć część nawiasową, wówczas taki symbol oznacza listę bez separatorów. Metasymbol „::=" oddziela definiowany symbol od jego definicji, zaś „ε" oznacza słowo puste. Symbole nieterminalne są wyróżnione za pomocą *zwykłej czcionki pochylonej*. Symbole terminalne są wydrukowane `czcionką maszynową`.

6.1 Identyfikatory i liczby

Identyfikator (*IDENT*) jest rozpoczynającym się od litery ciągiem liter, cyfr i podkreśleń. Identyfikatory służą do nazywania bytów używanych i definiowanych w specyfikacji: programów dostępu, funkcji pomocniczych, zmiennych i typów.

Liczba całkowita (*INTEGER*) jest ciągiem cyfr, który może być poprzedzony minusem.

6.2 Specyfikacja

specification ::=
characteristics
syntax
canonical
semanticsMutators
semanticsObservers

Specyfikacja tropowa składa się z pięciu części. W sekcji *characteristics* określamy nazwę specyfikowanego typu, wymieniamy inne typy używane w tej specyfikacji, oraz parametry. Sekcja *syntax* jest listą programów dostępu obiektów specyfikowanego typu, wymienia również sygnały jakie te obiekty mogą wysyłać. Sekcja *canonical* definiuje pomocnicze pojęcia, w tym również predykat *canonical* określający zbiór tropów kanonicznych. W dwóch ostatnich sekcjach specyfikujemy działanie programów dostępu, odpowiednio modyfikatorów i obserwatorów.

Miejsca definicji wykorzystanych symboli nieterminalnych

Symbol	Punkt	Strona
characteristics	6.3	29
syntax	6.4	31
canonical	6.5	33
semanticsMutators	6.6	36
semanticsObservers	6.7	38

Przykład

W dodatkach D–I znajdują się pełne teksty przykładowych specyfikacji.

6.3 Sekcja CHARACTERISTICS

characteristics ::=
`(0) CHARACTERISTICS`
`* type specified:` *IDENT*
⟦*typeParameter*⟧*
valueParameters
foreignTypes

typeParameter ::=
`* type parameter:` *IDENT*
$[\![$*mutator*$]\!]^*$
$[\![$*observer*$]\!]^*$

valueParameters ::=
ε | `* value parameters:` $[\![$*parameter*$]\!]^+$(;)

parameter ::=
$[\![$*IDENT*$]\!]^+$(,) `:` *IDENT*

foreignTypes ::=
ε | `* foreign types:` $[\![$*IDENT*$]\!]^+$(,)

Sekcja CHARACTERISTICS wskazuje *typ specyfikowany*. Po nim wymienione są *typy-parametry* specyfikacji. Odpowiadające im typy-parametry aktualne muszą udostępniać wymienione programy — modyfikatory i obserwatory (listy $[\![$*mutator*$]\!]^*$() i $[\![$*observer*$]\!]^*$() w *typeParameter*). Następnie są wyliczone *parametry-wartości* i ich typy (*IDENT* po dwukropku w *parameter* określa typ); odpowiadające im parametry aktualne powinny być wartościami zadanego typu. Na końcu sekcji znajduje się lista *typów obcych*.

W specyfikacji nie wolno korzystać z innych typów niż typ specyfikowany, typy-parametry i typy obce. Spośród wszystkich programów dostępu zdefiniowanych przez typ-parametr aktualny, w specyfikacji są widoczne jedynie te, które zostały wymienione przez odpowiedni *typeParameter*. Natomiast widoczne są wszystkie programy dostępu zdefiniowane przez specyfikacje typów obcych. Od tego miejsca będziemy używali terminu „typ obcy" na oznaczenie zarówno typów-parametrów jak i typów zdefiniowanych uprzednio jako obce.

W TAM 2000 używamy nazwy bez żadnych dekoracji zarówno na oznaczenie typu, jak i jego zbioru wartości (por. p. 4.2). Tym zbiorem jest zawsze zbiór tropów kanonicznych tego typu (lub zbiór stanów abstrakcyjnych, gdy specyfikacja typu została napisana za pomocą innej metody niż metoda tropów). Możemy także odwoływać się do zbioru wszystkich tropów, jednak dotyczy to tylko typu specyfikowanego. Ten zbiór jest oznaczony specjalnym symbolem „*traces*". Z tego powodu identyfikator "*traces*" nie może być nazwą żadnego typu ani typu-parametru.

W każdej specyfikacji są dostępne dwa predefiniowane typy obce: int i bool, oznaczające odpowiednio liczby całkowite i wartości logiczne.

Miejsca definicji wykorzystanych symboli nieterminalnych

Symbol	Punkt	Strona
mutator	6.4	31
observer	6.4	31

Składnia prezentacyjna

W składni prezentacyjnej używamy tłustej kropki "•" zamiast gwiazdki "*" na początku tytułów poszczególnych podpunktów sekcji CHARACTERISTICS.

Przykład

Oto sekcja CHARACTERISTICS ze specyfikacji stosu (por. dodatek D):

```
(0) CHARACTERISTICS
  * type specified: stack
  * type parameter: elem
```

Typem specyfikowanym jest `stack`; nie wprowadzono żadnych typów obcych (brak frazy `* foreign types`). Jedynym typem-parametrem jest `elem` (z dalszej części tej specyfikacji wynika, że jest to typ obiektów odkładanych na stosie). Nie zadeklarowano żadnych parametrów-wartości (brak frazy `* value parameters`).

Ta sama sekcja zapisana zgodnie z wymienioną powyżej regułą składni prezentacyjnej ma postać:

(0) CHARACTERISTICS

- type specified: stack
- type parameter: *elem*

6.4 Sekcja SYNTAX

syntax ::=
`(1) SYNTAX`
⟦*mutator*⟧*
⟦*observer*⟧*
⟦*signal*⟧*

mutator ::=
`mutator` *IDENT argTypes nonDetResult*

observer ::=
`observer` *IDENT argTypes* `result` *IDENT*

signal ::=
`signal` *IDENT argTypes*

argTypes ::=
ε | `args` ⟦*IDENT*⟧+

nonDetResult ::=
ε | `non-det result` *IDENT*

Sekcja SYNTAX składa się z trzech części, w których są deklarowane kolejno modyfikatory, obserwatory i sygnały.

Modyfikatory Pierwsza część sekcji SYNTAX jest poświęcona modyfikatorom, czyli programom, które zmieniają stan obiektu. Każdy z modyfikatorów ma jeden niejawny argument, wskazujący obiekt, którego stan podlega zmianie. Pierwszy *IDENT* w definicji modyfikatora (*mutator*) określa jego nazwę. Nazwy modyfikatorów muszą być parami różne. Po nazwie następuje lista typów jawnych argumentów modyfikatora (*argTypes*). Definicja modyfikatora może być zakończona wskazaniem typu niedeterministycznie przekazywanego wyniku (*nonDetResult*). Musi to być typ obcy. Program przekazujący niedeterministyczny wynik jest zawsze modyfikatorem, ponieważ taki wynik może mieć wpływ na przyszłe działanie obiektu.

Obserwatory Druga część sekcji SYNTAX jest poświęcona obserwatorom, czyli deterministycznym programom, które dostarczają na zewnątrz obiektu informacje o jego stanie. Pierwszy *IDENT* w definicji obserwatora (*observer*) określa jego nazwę. Nazwy obserwatorów muszą być parami różne. Wszystkie argumenty obserwatora są jawne; ich typy są wymienione na liście *argTypes*. Ostatni *IDENT* w *observer* wskazuje typ przekazywanego wyniku. Może to być typ obcy lub specyfikowany. Nazwy obserwatorów muszą być parami różne.

Sygnały Trzecia część sekcji SYNTAX jest poświęcona sygnałom, które mogą być wysyłane przez obiekty specyfikowanego typu w odpowiedzi na wywołania programów dostępu. Deklaracja sygnału (*signal*) zawiera jego nazwę (*IDENT*) i listę typów argumentów (*argTypes*). Nazwy sygnałów muszą być parami różne. Zbiory nazw obserwatorów, modyfikatorów i sygnałów muszą być parami rozłączne.

Składnia prezentacyjna

Poszczególne elementy sekcji SYNTAX mogą zostać zapisane za pomocą notacji tabelarycznej. W składni prezentacyjnej ta sekcja składa się z trzech tabel, odpowiednio dla modyfikatorów, obserwatorów i sygnałów. Tabele te są poprzedzone tytułami, odpowiednio: MUTATORS, OBSERVERS i SIGNALS. Każdy z definiowanych bytów (modyfikator, obserwator lub sygnał) staje się wierszem właściwej tabeli. Każda z tabel ma następujące kolumny:

- jedna kolumna dla nazwy bytu (nagłówki: `Program Name` lub `Signal Name`),
- kolumny na typy argumentów (nagłówki: `Arg#1`, `Arg#2`, ...; liczba tych kolumn w danej tabeli jest równa maksymalnej liczbie argumentów bytów danego rodzaju),
- jedna kolumna na typ przekazywanego wyniku (tylko tabela dla obserwatorów; nagłówek: `Result Type`),
- jedna kolumna na typ niedeterministycznie przekazywanego wyniku (tylko tabela dla modyfikatorów i jedynie wtedy, gdy co najmniej jeden z modyfikatorów przekazuje niedeterministyczny wynik; nagłówek: `Non-det. Result`).

W tabelach umieszczamy jedynie nazwy programów i nazwy typów (*IDENT*), nie pojawiają się w nich symbole terminalne użyte w gramatyce definiującej sekcję SYNTAX (`mutator`, `observer`, `signal`, `args`, `result`, `non-det result`).

Przykład

Oto sekcja SYNTAX ze specyfikacji stosu (por. dodatek D):

```
(1) SYNTAX
  mutator JOIN    args stack
  mutator PUSH    args elem
  mutator POP
  mutator SPLIT_1 args stack
  mutator SPLIT_2 args stack
  oberver TOP     args stack result elem
  signal  EMPTY
```

W tej sekcji zadeklarowano pięć modyfikatorów, jeden obserwator i jeden sygnał. Trzy z tych modyfikatorów (`JOIN`, `SPLIT_1`, `SPLIT_2`) przyjmują jeden argument typu `stack`. `POP` jest programem bezargumentowym, zaś typem jedynego argumentu `PUSH` jest `elem`. Typem argumentów obserwatora `TOP` jest `stack`, zaś typem wyniku `elem`. Sygnał `EMPTY` jest bezargumentowy.

Ta sama sekcja zapisana zgodnie z wymienionymi powyżej regułami składni prezentacyjnej ma postać:

(1) SYNTAX

MUTATORS

Program Name	Arg #1
JOIN	stack
PUSH	*elem*
POP	
SPLIT_1	stack
SPLIT_2	stack

OBSERVERS

Program Name	Arg #1	Result Type
TOP	stack	*elem*

SIGNALS

Signal Name
EMPTY

6.5 Sekcja CANONICAL TRACES

canonical ::=
`(2) CANONICAL TRACES`
⟦*auxDefinition*⟧$^+$

auxDefinition ::=
    signature
  | *domainRestriction*
  | *valueDefinition*

signature ::=
    IDENT `:` ⟦*IDENT*⟧* `->` *IDENT*

domainRestriction ::=
    `legal (` *IDENT* `(` ⟦*traceExpr*⟧*(,) `) )` `<=>` *logExpr*

valueDefinition ::=
    IDENT `(` ⟦*traceExpr*⟧*(,) `) =` *traceExpr*
  | *IDENT* `(` ⟦*traceExpr*⟧*(,) `) <=>` *logExpr*

Definiowane w sekcji CANONICAL TRACES funkcje pomocnicze spełniają w specyfikacji istotną rolę. Po pierwsze, pozwalają one zapisać w jednym miejscu wielokrotnie używane wyrażenia, co skraca i upraszcza specyfikację. Po drugie, w definicjach funkcji pomocniczych może występować rekurencja, co nie jest dopuszczalne w innych częściach specyfikacji (por. p. 4.7). Po trzecie, jedną z tych funkcji jest niezwykle ważny w specyfikacji predykat *canonical*, który definiuje zbiór tropów kanonicznych.

Definicja pojedynczej funkcji pomocniczej składa się z trzech części: sygnatury (dokładnie jedno *signature*), ograniczenia dziedziny (dowolna liczba *domainRestriction*) i definicji wartości (co najmniej jedno *valueDefinition*). Pierwsze wystąpienie *IDENT* w każdym z *signature*, *domainRestriction*, *valueDefinition* wskazuje nazwę funkcji. Nazwy funkcji muszą być parami różne i nie mogą pokrywać się z nazwami programów dostępu ani sygnałów. Części definicji ustalonej funkcji nie powinny być rozdzielone fragmentami definicji innych funkcji.

Sygnatura Wystąpienie *signature* wprowadza nazwę funkcji (pierwszy *IDENT*), listę typów argumentów (⟦*IDENT*⟧*) i typ przekazywanej wartości (ostatni *IDENT*). Każdy z tych typów musi być typem obcym lub zbiorem *traces*; typ specyfikowany może występować dopiero w tekście znajdującym się za definicją predykatu *canonical*. Wynika to z faktu, że *canonical* definiuje zbiór tropów kanonicznych, które są wartościami typu specyfikowanego.

Dopasowywanie wzorca W definicji ograniczenia dziedziny, podobnie jak w definicjach wszystkich funkcji w specyfikacji, możemy korzystać z dopasowywania wzorca (por. p. 4.3). W tym miejscu opiszemy szczegółowo semantykę dopasowywania wzorca; jako przykład wykorzystamy *domainRestriction*. W punktach przedstawiających inne części specyfikacji będziemy odwoływać się do niniejszego opisu.

Definicja funkcji może składać się z wielu równań. Parametrami formalnymi każdego równania (np. *domainRestriction*) są wyrażenia tropowe (⟦*traceExpr*⟧+(,)), a nie zmienne jak w klasycznym ujęciu. Listę tych wyrażeń tropowych nazwiemy *wzorcem*. We wzorcu mogą występować zmienne. Dla zadanych wartości parametrów aktualnych szukamy dopasowania, czyli wartościowania, przy którym wzorzec staje się równy liście parametrów aktualnych. Następnie to wartościowanie stosujemy do wyrażenia definiującego wartość rozważanej funkcji (w *domainRestriction* jest to *logExpr*). Otrzymany wynik jest wartością funkcji dla tych parametrów aktualnych. Dla każdego elementu dziedziny i ustalonego wzorca może istnieć co najwyżej jedno takie wartościowanie.

Dla każdego elementu dziedziny musi istnieć w definicji funkcji co najmniej jedno równanie, w którym wzorzec da się dopasować do tego elementu dziedziny. Gdy takich równań jest więcej niż jedno, wartość funkcji dla takiego elementu dziedziny definiuje to z nich, które występuje najwcześniej w specyfikacji.

Ograniczenie dziedziny Dziedzinę funkcji częściowej definiujemy za pomocą *domainRestriction*. Jeśli dla pewnej funkcji pomocniczej nie występuje ani jedno *domainRestriction*, jej dziedziną jest iloczyn kartezjański zbiorów wskazanych przez sygnaturę. Jeśli ograniczenie dziedziny występuje, musi być definiowane dla wszystkich elementów tego iloczynu. Do dziedziny funkcji należą jedynie te jego elementy, dla których *logExpr* z odpowiedniego *domainRestriction* jest prawdziwe. Typy wyrażeń tropowych we wzorcu muszą odpowiadać typom zadeklarowanym w sygnaturze. W *domainRestriction* można korzystać jedynie z wcześniej zdefiniowanych funkcji pomocniczych.

Definicja wartości Co najmniej jedno *valueDefinition* musi występować dla każdej funkcji pomocniczej. Zestaw wszystkich *valueDefinition* dotyczących pewnej funkcji musi określać jej wartość dla wszystkicj elementów jej dziedziny. We wzorcu wszystkie wyrażenia tropowe muszą odpowiadać typom zadeklarowanym w sygnaturze. Jeśli typem wyniku jest bool, korzystamy z drugiej postaci *valueDefinition* (`<=>` *logExpr*). W przeciwnym przypadku, *valueDefinition* występuje w pierwszej postaci (`=` *traceExpr*); wówczas typ ostatniego wyrażenia musi odpowiadać typowi wyniku zadeklarowanemu w sygnaturze. W *valueDefinition* można korzystać jedynie z wcześniej zadeklarowanych funkcji pomocniczych. Jedyny wyjątek stanowi wyrażenie określające wynik (ostatnie *logExpr* lub *traceExpr*), w którym można odwołać się do definiowanej funkcji. To pozwala pisać definicje rekurencyjne.

Definicja zbioru tropów kanonicznych Wśród funkcji pomocniczych musi być zdefiniowany predykat *canonical*. Jest to funkcja o sygnaturze:

canonical : *traces* `->` bool

Predykat *canonical* nie może mieć ograniczeń dziedziny (*domainRestriction*). Tropy, dla których wartością *canonical* jest prawda, nazywamy *kanonicznymi*.

Wbudowane funkcje pomocnicze W każdej specyfikacji tropowej można korzystać z dwóch funkcji pomocniczych: *length* i *count*. Jedynym argumentem funkcji *length* jest trop; wartością tej funkcji jest liczba całkowita równa liczbie wywołań programów dostępu występujących w tym tropie. Funkcja *count* ma dwa argumenty: trop i nazwę programu dostępu; wartością *count* jest liczba całkowita równa liczbie wywołań tego programu dostępu występujących w tym tropie.

Miejsca definicji wykorzystanych symboli nieterminalnych

Symbol	Punkt	Strona
logExpr	6.13	48
traceExpr	6.11	44

Przykład

W sekcji CANONICAL TRACES specyfikacji stosu (por. dodatek D) użyto składni prezentacyjnej — wykorzystano zestaw symboli specjalnych (por. p. 6.16) oraz konwencje dotyczące zapisu iteracji (por. p. 6.11). Poniżej zamieszczono zapis tej samej sekcji jedynie za pomocą znaków ASCII:

```
(2) CANONICAL TRACES
  canonical : traces -> bool
  canonical([PUSH(x[i])](i=1...s)) <=> true
  canonical(other) <=> false
```

Zdefiniowano tu jedynie funkcję `canonical`, której dziedziną jest zbiór wszystkich tropów (`traces`), natomiast przeciwdziedziną — zbiór wartości logicznych (`bool`). Dziedzina ta nie jest ograniczona żadnym predykatem (brak frazy `legal`). Definicja tej funkcji składa się z dwóch równań. Dla tropów, które są postaci `[PUSH(x[i])](i=1...s)`, wartość `canonical` jest określona przez pierwsze równanie i jest to `true`. Wartość `canonical` dla pozostałych tropów jest określona przez drugie równanie i wynosi `false`. Zauważmy, że każdy trop da się dopasować do wzorca w drugim równaniu — wystarczy przyjąć dla zmiennej `other` wartość równą temu tropowi. Jednakże zgodnie z semantyką specyfikacji w TAM 2000, dla tropów, które pasują do wzorców w obu równaniach, ważniejsze jest pierwsze z nich.

6.6 Sekcja SEMANTICS OF MUTATORS

semanticsMutators ::=
`(3) SEMANTICS OF MUTATORS`
emptyEquivalence
$[\![$*mutatorDefinition*$]\!]^+$

emptyEquivalence ::=
ε | `_ =>` *traceExpr*

mutatorDefinition ::=
mutatorLegality
| *mutatorFeasibility*
| *mutatorSignals*
| *mutatorTransition*

mutatorLegality ::=
`legal (` *traceExpr* `;` *application* `) <=>` *logExpr*

mutatorFeasibility ::=
`feasible (` *traceExpr* `;` *application* `) <=>` *logExpr*

mutatorSignals ::=
`signals (` *traceExpr* `;` *application* `) =` *signalExpr*

mutatorTransition ::=
`transition (` *traceExpr* `;` *application* `) =` *traceExpr*

W sekcji SEMANTICS OF MUTATORS specyfikujemy działanie modyfikatorów. Dokonujemy tego poprzez określenie czterech funkcji. Pierwsza z nich, *legal*, określa, które wywołania są legalne. (por. p. 4.6). Druga, *feasible*, definiuje zbiór, z którego jest wybierany niedeterministyczny wynik. Trzecia, *signals*, wymienia sygnały jakie zostaną wysłane przez obiekt w odpowiedzi na wywołanie. Czwarta, *transition* opisuje zmianę stanu, która zachodzi w wyniku wywołania. Każda z tych funkcji ma dwa argumenty: trop kanoniczny reprezentujący stan obiektu i wywołanie programu dostępu. Typem pierwszego *traceExpr* w równaniach definiujących te funkcje musi być typ specyfikowany. Wystąpienia *application* w *mutatorLegality*, *mutatorFeasibility*, *mutatorSignals* i *mutatorTransition* muszą być wywołaniami modyfikatorów. W definicjach możemy korzystać z dopasowywania wzorca (por. p. 6.5). Kolejność poszczególnych równań może być dowolna. Sugerujemy zapisywanie równań w grupach dla poszczególnych modyfikatorów. We wszystkich specyfikacjach w dodatkach D–G zastosowano się do tej sugestii. W TAM 2000 przyjęto konwencje, które określają wartości domyślne funkcji *feasible* i *signals* dla pewnych szczególnych argumentów. Dla takich argumentów definicje tych funkcji pomija się. Konwencje te zostały wymienione poniżej.

Legalność wywołań Funkcja *legal* musi być określona dla wszystkich możliwych par — trop kanoniczny i wywołanie. Wywołanie będące argumentem nie zawiera niedeterministycznie przekazywanego wyniku (zawsze jest to pierwsza postać *application* — por. p. 6.8). W stanie zadanym pewnym tropem kanonicznym, ustalone wywołanie jest legalne, jedynie jeśli wartością funkcji *legal* dla tego tropu i tego wywołania jest prawda.

Poprawność niedeterministycznie przekazanego wyniku Funkcja *feasible* musi być określona dla wszystkich legalnych wywołań. Jeśli program nie przekazuje niedeterministycznego wyniku, *feasible* jest prawdą dla każdego legalnego wywołania. Dla takich programów nie umieszczamy w specyfikacji definicji funkcji *feasible*. Wystąpienie *application* w *mutatorFeasibility* musi mieć drugą postać zdefiniowaną dla tego symbolu (z "^*traceExpr*" — por. p. 6.8). W stanie zadanym pewnym tropem kanonicznym, niedeterministyczny wynik wywołania jest poprawny, gdy wartością *feasible* dla tego tropu i tego wywołania jest prawda.

Sygnały Funkcja *signals* wylicza sygnały wysyłane przez obiekt w odpowiedzi na wywołanie programu dostępu. Jest ona określona dla wszystkich wywołań (również nielegalnych). Wywołanie będące argumentem nie zawiera niedeterministycznie przekazywanego wyniku (zawsze jest to pierwsza postać *application* — por. p. 6.8). W *signalExpr* możemy korzystać z funkcji *legal*. Często, w odpowiedzi na wywołanie, obiekt nie wysyła żadnych sygnałów. Z tego powodu przyjęto, że jeśli dla pewnej pary, trop kanoniczny i wywołanie, nie istnieje równanie, w którym wzorzec pasowałby do tej pary, to w odpowiedzi na to wywołanie w stanie opisanym tym tropem kanonicznym nie są wysyłane żadne sygnały.

Zmiana stanu Funkcja *transition* musi być określona dla wszystkich legalnych wywołań. Przeciwdziedziną tej funkcji jest zbiór tropów kanonicznych. Jeśli program nie przekazuje niedeterministycznego wyniku, *application* w *mutatorFeasibility* musi być w pierwszej postaci (bez "^*traceExpr*" — por. p. 6.8). Jeśli program przekazuje niedeterministyczny wynik, *application* w *mutatorTransition* musi być w drugiej postaci (z "^*traceExpr*" — por. p. 6.8). Typem ostatniego *traceExpr* w *mutatorTransition* musi być typ specyfikowany. W stanie zadanym

pewnym tropem kanonicznym, w wyniku ustalonego wywołania, obiekt przechodzi do stanu wskazanego przez wartość funkcji *transition* dla tego tropu i tego wywołania.

Miejsca definicji wykorzystanych symboli nieterminalnych

Symbol	Punkt	Strona
application	6.8	40
logExpr	6.13	48
signalExpr	6.10	43
traceExpr	6.11	44

Przykład

W sekcji SEMANTICS OF MUTATORS specyfikacji stosu (por. dodatek D) użyto składni prezentacyjnej — wykorzystano zestaw symboli specjalnych (por. p. 6.16). Poniżej zamieszczono zapis fragmentu tej samej sekcji jedynie za pomocą znaków ASCII:

```
(3) SEMANTICS OF MUTATORS
  legal(T; POP()) <=> T != _
  signals(_, POP()) = [ EMPTY() ]
  transition(T.PUSH(x); POP()) = T
```

Powyższy fragment określa semantykę programu `POP`. Pierwsze z tych równań definiuje legalność wywołania — wywołanie `POP` jest legalne, jedyne jeśli stos, którego dotyczy to wywołanie jest niepusty, tzn. stan tego stosu nie jest reprezentowany tropem pustym (`T != _`).

Drugie równanie definiuje zbiór sygnałów wysyłanych w odpowiedzi na wywołanie `POP`. Występujący w nim wzorzec ("`_`") pasuje do tropu pustego, zatem wywołanie `POP` na pustym stosie powoduje wysłanie sygnału `EMPTY()`. Nie ma równania definiującego funkcję `signals` dla pozostałych tropów i wywołania `POP`. Zatem zgodnie z semantyką specyfikacji w TAM 2000 w odpowiedzi na `POP` nie są wysyłane żadne sygnały, gdy stan stosu jest reprezentowany tropem innym niż pusty.

Trzecie równanie definiuje stan, do którego przechodzi obiekt w wyniku legalnego wywołania programu `POP`. Takie wywołanie musi odbywać się na niepustym stosie, tzn. takim, którego stan jest reprezentowany przez trop złożony z co najmniej jednego wywołania modyfikatora `PUSH`[1]. Zatem dla każdego legalnego wywołania `POP`, trop reprezentujący stan stosu da się dopasować do wzorca `T.PUSH(x)`. Wartość zmiennej `T` wyznaczona przy dopasowywaniu jest tropem kanonicznym reprezentującym stan stosu po zakończeniu wywołania `POP`.

6.7 Sekcja SEMANTICS OF OBSERVERS

semanticsObservers ::=

ε

| `(4) SEMANTICS OF OBSERVERS` *observerDefinition*$^+$

[1] Przypomnijmy, w specyfikacji stosu trop kanoniczny składa się jedynie z wywołań modyfikatora `PUSH` (por. przykład w p. 6.5).

observerDefinition ::=
    observerLegality
  | *observerSignals*
  | *observerResult*

observerLegality ::=
    `legal (` *application* `) <=>` *logExpr*

observerSignals ::=
    `signals (` *application* `) =` *signalExpr*

observerResult ::=
    `result (` *application* `) =` *traceExpr*
  | `result (` *application* `) <=>` *logExpr*

W sekcji SEMANTICS OF OBSERVERS specyfikujemy działanie obserwatorów. Dokonujemy tego poprzez określenie trzech funkcji. Pierwsza z nich, *legal*, określa, które wywołania są legalne (por. p. 4.6). Druga, *signals*, wymienia sygnały jakie zostaną wysłane w odpowiedzi na wywołanie. Trzecia, *result*, opisuje wynik przekazany przez wywołanie. Jedynym argumentem każdej z tych funkcji jest wywołanie programu dostępu. Wystąpienia *application* w *observerLegality*, *observerSignals* i *observerResult* muszą być wywołaniami obserwatorów. W definicjach możemy korzystać z dopasowywania wzorca (por. p. 6.5). Kolejność poszczególnych równań może być dowolna. Sugerujemy zapisywanie równań w grupach dla poszczególnych obserwatorów. We wszystkich specyfikacjach w dodatkach D–G zastosowano się do tej sugestii. Jeśli typ specyfikowany nie wprowadza obserwatorów, sekcję SEMANTICS OF OBSERVERS pomija się.

Legalność wywołań Funkcja *legal* musi być określona dla wszystkich wywołań. Wywołanie jest legalne, jedynie jeśli wartością funkcji *legal* dla tego tego wywołania jest prawda.

Sygnały Funkcja *signals* wylicza sygnały wysyłane w odpowiedzi na wywołanie programu dostępu. Jest ona określona dla wszystkich wywołań (również nielegalnych). W *signalExpr* możemy korzystać z funkcji *legal*. Często, w odpowiedzi na wywołanie, nie wysyła się żadnych sygnałów. Z tego powodu przyjęto, że jeśli dla pewnego wywołania nie istnieje równanie, w którym wzorzec pasowałby do niego, to w odpowiedzi na to wywołanie nie wysyłane są żadne sygnały.

Wynik przekazywany przez wywołanie Funkcja *result* musi być określona dla wszystkich legalnych wywołań. Jeśli typem wyniku programu wymienionego przez *application* w *observerResult* jest bool, korzystamy z drugiej postaci *observerResult* (`<=>` *logExpr*). W przeciwnym przypadku, *observerResult* występuje w pierwszej postaci (`=` *traceExpr*); wówczas typ ostatniego wyrażenia musi odpowiadać typowi wyniku tego programu. Gdy jest to typ specyfikowany, wartością ostatniego *traceExpr* w *observerResult* musi być trop kanoniczny. Wynik przekazany przez wywołanie jest wartością funkcji *result* dla tego wywołania.

Miejsca definicji wykorzystanych symboli nieterminalnych

Symbol	Punkt	Strona
application	6.8	40
logExpr	6.13	48
signalExpr	6.10	43
traceExpr	6.11	44

Przykład

W sekcji SEMANTICS OF OBSERVERS specyfikacji stosu (por. dodatek D) użyto składni prezentacyjnej — wykorzystano zestaw symboli specjalnych (por. p. 6.16). Poniżej zamieszczono zapis tej samej sekcji jedynie za pomocą znaków ASCII:

```
(4) SEMANTICS OF OBSERVERS
  legal(TOP(T)) <=> T != _
  signals(TOP(_)) = [ EMPTY() ]
  result(TOP(T.PUSH(x))) = x
```

Określono tu semantykę programu `TOP`. Pierwsze z tych równań definiuje legalność wywołania — wywołanie `TOP(T)` jest legalne, jedyne jeśli stos, będący argumentem tego wywołania jest niepusty, tzn. stan tego stosu nie jest reprezentowany tropem pustym (`T != _`).

Drugie równanie definiuje zbiór sygnałów wysyłanych w odpowiedzi na wywołanie `TOP(_)`. Występujący w nim wzorzec ("`_`") pasuje do tropu pustego, zatem wywołanie `TOP`, którego argumentem jest stos pusty, powoduje wysłanie sygnału `EMPTY()`. Nie ma równania definiującego funkcję `signals` dla wywołań `TOP(T)`, gdy `T` nie jest równe tropowi pustemu. Zatem zgodnie z semantyką specyfikacji w TAM 2000, w odpowiedzi na takie `TOP(T)` nie są wysyłane żadne sygnały.

Trzecie równanie definiuje wynik przekazywany przez legalne wywołania programu `TOP`. Argumentem takiego wywołania musi być niepusty stos, tzn. taki, którego stan jest reprezentowany przez trop złożony z co najmniej jednego wywołania modyfikatora `PUSH`[2]. Zatem dla każdego legalnego wywołania `TOP`, jego argument da się dopasować do wzorca `T.PUSH(x)`. Wartość zmiennej `x` wyznaczona przy dopasowywaniu jest wynikiem, który zostanie przekazany przez to wywołanie.

6.8 Wywołania specyfikowanych programów dostępu i funkcji pomocniczych

application ::=
  IDENT `(` ⟦*traceExpr*⟧*(,) `)`
| *IDENT* `(` ⟦*traceExpr*⟧*(,) `)` `^` *traceExpr*

Każde wystąpienie *application* spełnia jedną z czterech ról: parametr formalny funkcji definiowanych w sekcjach SEMANTICS OF MUTATORS i SEMANTICS OF OBSERVERS, wywołanie modyfikatora będące elementem wyrażenia tropowego, zastosowanie funkcji pomocniczej, oraz sygnał w wyrażeniu sygnałowym. Poniżej omówiono każdą z tych ról.

[2]Przypomnijmy, w specyfikacji stosu trop kanoniczny składa się jedynie z wywołań modyfikatora `PUSH` (por. przykład w p. 6.5).

Parametr formalny w sekcjach SEMANTICS Pierszy *IDENT* w takim *application* musi być nazwą programu dostępu zadeklarowanego w bieżącej specyfikacji. Typy i liczba argumentów ($[\![traceExpr]\!]^*(,)$) muszą odpowiadać deklaracji tego programu. Jedynie dla modyfikatorów, które przekazują niedeterministyczny wynik, *application* może być zakończone oznaczającą ten wynik frazą "`^`*traceExpr*". Typ tego *traceExpr* musi odpowiadać typowi niedeterministycznego wyniku tego modyfikatora.

Wywołanie modyfikatora jako element wyrażenia tropowego Pierwszy *IDENT* musi być nazwą modyfikatora zadeklarowanego w bieżącej specyfikacji. Typy i liczba argumentów ($[\![traceExpr]\!]^*(,)$) muszą odpowiadać deklaracji tego modyfikatora. Takie *application* jest zakończone frazą "`^`*traceExpr*" oznaczającą niedeterministycznie przekazany wynik, jedynie jeśli deklaracja tego modyfikatora przewiduje taki wynik. Typ *traceExpr* za symbolem "`^`" musi odpowiadać typowi niedeterministycznego wyniku tego modyfikatora.

Typem takiego *application* jako wyrażenia tropowego jest typ specyfikowany. Jego wartością jest wywołanie modyfikatora powstające z tego *application* poprzez zastąpienie wszystkich *traceExpr* ich wartościami.

Zastosowanie funkcji pomocniczej Pierwszy *IDENT* musi być nazwą funkcji pomocniczej zadeklarowanej w bieżącej specyfikacji. Typy i liczba argumentów ($[\![traceExpr]\!]^*(,)$) muszą odpowiadać deklaracji tej funkcji. Takie *application* nie może być zakończone frazą "`^`*traceExpr*". Krotka złożona z wartości wyrażeń z listy $[\![traceExpr]\!]^*(,)$ musi być elementem dziedziny tej funkcji. Typem takiego *application* jako wyrażenia tropowego jest typ wyniku tej funkcji pomocniczej. Jego wartością jest wynik zastosowania tej funkcji do wartości wszystkich *traceExpr* będących argumentami tego *application*.

Sygnał w wyrażeniu sygnałowym Pierwszy *IDENT* musi być nazwą sygnału zadeklarowanego w bieżącej specyfikacji. Typy i liczba argumentów ($[\![traceExpr]\!]^*(,)$) muszą odpowiadać deklaracji tego sygnału. Takie *application* nie może być zakończone frazą "`^`*traceExpr*". Wartością takiego *application* jako wyrażenia sygnałowego jest sygnał, powstający z tego *application* poprzez zastąpienie wszystkich *traceExpr* ich wartościami.

Miejsca definicji wykorzystanych symboli nieterminalnych

Symbol	Punkt	Strona
traceExpr	6.11	44

Przykład

Oto jedno z równań z sekcji SEMANTICS OF MUTATORS specyfikacji zarządcy nazw (por. przykład w p. 6.6):

```
transition(T; PICK()^a) = lessElems(T, a).PICK()^a.greaterElems(T, a)
```

Pierwsze wystąpienie `PICK()^a` w tym równaniu jest jego parametrem formalnym, wzorcem do którego dopasowywane są parametry aktualne. Natomiast drugie wystąpienie `PICK()^a` jest elementem wyrażenia tropowego. Identyfikatory `lessElems` i `greaterElems` są nazwami funkcji pomocniczych zadeklarowanych w specyfikacji zarządcy nazw — `lessElems(T, a)` i `greaterElems(T, a)` są więc wywołaniami funkcji pomocniczych.

Obliczenie funkcji `transition` dla tropu T_0 i wywołania PICK`^`a_0 polega na dopasowaniu T_0 do wzorca `T` i PICK`^`a_0 do wzorca `PICK()^a`. Z tego dopasowania otrzymamy wartościowanie, które zmiennej `T` przypisuje wartość T_0, zaś zmiennej `a` wartość a_0. Następnie obliczamy wartości funkcji `lessElems` i `greaterElems` dla pary argumentów $\langle T_0, a_0 \rangle$. Poszukiwaną wartością funkcji `transition` jest sklejenie następujących tropów (w takiej kolejności): wartości funkcji `lessElems` dla $\langle T_0, a_0 \rangle$, tropu PICK`^`a_0 i wartości funkcji `greaterElems` dla $\langle T_0, a_0 \rangle$.

Następujące równanie pochodzi z sekcji SEMANTICS OF OBSERVES specyfikacji stosu (por. przykład w p. 6.7):

```
signals(TOP(_)) = [ EMPTY() ]
```

Wyrażenie `EMPTY()` jest tu sygnałem, ponieważ zgodnie z sekcją SYNTAX specyfikacji stosu (por. przykład w 6.4) `EMPTY` jest nazwą sygnału. Fraza `EMPTY()` jest częścią wyrażenia sygnałowego `[ EMPTY() ]`.

6.9 Wywołania obcych programów dostępu

foreignApplication ::=
    IDENT `::` *IDENT* `(` ⟦*traceExpr*⟧*(,) `)`
  | *IDENT* `::` *IDENT* `(` *traceExpr* `;` ⟦*traceExpr*⟧*(,) `)`
  | *IDENT* `::` *IDENT* `(` *traceExpr* `;` ⟦*traceExpr*⟧*(,) `)` `^` *traceExpr*

Poprzez *foreignApplication* możemy odwoływać się do programów dostępu typów obcych. *IDENT* znajdujący się przed symbolem "`::`" wskazuje typ obcy, z którego pochodzi program. Postać *foreignApplication* zależy od rodzaju programu, do którego się odwołuje.

Obserwator Jeśli program wskazany przez *IDENT* występujący za symbolem "`::`" jest obserwatorem, *foreignApplication* występuje w pierwszej z wymienionych w regule postaci. Typy i liczba argumentów (⟦*traceExpr*⟧*(,)) muszą odpowiadać deklaracji tego obserwatora. Typem takiego *foreignApplication* jest typ wyniku tego obserwatora. Wartością jest wynik przekazywany przez ten obserwator dla argumentów równych wartościom wszystkich *traceExpr* wchodzących w skład tego *foreignApplication*.

Modyfikator Jeśli program wskazany przez *IDENT* występujący za symbolem "`::`" jest modyfikatorem, który nie przekazuje niedeterministycznego wyniku, *foreignApplication* występuje w drugiej z wymienionych w regule postaci. Pierwszy argument, oddzielony od pozostałych średnikiem, reprezentuje niejawny argument modyfikatora — stan obiektu, na który to wywołanie ma wpływ. Typem tego argumentu musi być typ obcy, z którego pochodzi ten modyfikator. Typy i liczba pozostałych (jawnych) argumentów (⟦*traceExpr*⟧*(,) za średnikiem) muszą odpowiadać deklaracji tego modyfikatora. Typem takiego *foreignApplication* jest typ obcy, z którego pochodzi ten modyfikator. Wartością jest stan, w którym znajdzie się zmieniany obiekt, po wywołaniu tego modyfikatora z argumentami równymi wartościom wszystkich *traceExpr* wchodzących w skład tego *foreignApplication*.

Modyfikator z niedeterministycznym wynikiem Jeśli program wskazany przez *IDENT* występujący za symbolem "`::`" jest modyfikatorem, który przekazuje niedeterministyczny

wynik, *foreignApplication* występuje w trzeciej z wymienionych w regule postaci. Typ *traceExpr* znajdującego się za symbolem "`^`" musi odpowiadać typowi wyniku tego modyfikatora. Wartością takiego *foreignApplication* jako wyrażenia tropowego jest stan, w którym znajdzie się zmieniany obiekt, po wywołaniu tego modyfikatora przy niedeterministycznym wyniku i argumentach równych wartościom odpowiednich *traceExpr* wchodzących w skład tego *foreignApplication*. Pozostałe własności są identyczne jak dla modyfikatora, który nie przekazuje wyniku.

Miejsca definicji wykorzystanych symboli nieterminalnych

Symbol	Punkt	Strona
traceExpr	6.11	44

Przykład

Przypuśćmy, że w pewnej specyfikacji typem obcym jest stack (stos, którego specyfikacja znajduje się w dodatku D). W tej specyfikacji wartość wyrażenia

```
stack :: TOP(T)
```

zależy od definicji obserwatora `TOP` w specyfikacji stosu, jest więc równa wartości wierzchołka stosu, którego stan przechowuje zmienna `T`. Z kolei wartością wyrażenia

```
stack :: PUSH(T; 5)
```

jest stan, w którym znajdzie się stos po wywołaniu `PUSH(5)`, gdy jego stan sprzed tego wywołania był równy wartości zmiennej `T`.

Załóżmy, że w tejże specyfikacji również nameManager (zarządca nazw, którego specyfikacja znajduje się w dodatku F) jest typem obcym. W tej specyfikacji wartością wyrażenia

```
nameManager :: PICK(T;)^5
```

jest stan, w którym znajdzie się zarządca nazw po wywołaniu `PICK()`, które przekazało wartość `5`, gdy jego stan sprzed tego wywołania był równy wartości zmiennej `T`.

6.10 Wyrażenia sygnałowe

signalExpr ::=
`[` ⟦*application*⟧*(,) `]`
| *signalTable*

Wartością wyrażenia sygnałowego jest lista sygnałów. Lista ta może być pusta. Symbole "`[`" i "`]`" są ogranicznikami tej listy.

Miejsca definicji wykorzystanych symboli nieterminalnych

Symbol	Punkt	Strona
application	6.8	40
signalTable	6.15	50

Przykład

Następujące równanie pochodzi z sekcji SEMANTICS OF OBSERVES specyfikacji stosu (por. przykład w p. 6.7):

```
signals(TOP(_)) = [ EMPTY() ]
```

Fraza `[ EMPTY() ]` jest wyrażeniem sygnałowym. Lista sygnałów, które zostaną wysłane w odpowiedzi na wywołanie `TOP(_)` ma jeden element — sygnał `EMPTY()`.

6.11 Wyrażenia tropowe

traceExpr ::=
- *IDENT*
- *IDENT* `[` $[\![$*traceExpr*$]\!]^{+}(,)$ `]`
- *application*
- *foreignApplication*
- *INTEGER*
- `_`
- *IDENT* `::` `_`
- *traceExpr* `.` *traceExpr*
- `(` *traceExpr* `)`
- `[` *traceExpr* `]` `(` *IDENT* `=` *traceExpr* `...` *traceExpr* `)`
- *traceExpr* `+` *traceExpr*
- *traceExpr* `-` *traceExpr*
- *traceExpr* `*` *traceExpr*
- *traceExpr* `/` *traceExpr*
- `-` *traceExpr*
- *traceTable*

Każde wyrażenie tropowe (*traceExpr*) ma przypisany typ. Może to być typ specyfikowany lub jeden z typów obcych. Typ wyrażenia ogranicza miejsca, w których to wyrażenie może być użyte. Poniżej opiszemy reguły określające typ każdego wyrażenia tropowego i sposób wyliczania jego wartości. Kolejność prezentacji jest zgodna z kolejnością wariantów w regule dla *traceExpr*.

Odczytanie wartości zmiennej prostej *IDENT* musi być identyfikatorem zmiennej prostej. Wartością i typem takiego wyrażenia tropowego są odpowiednio wartość i typ tej zmiennej.

Odczytanie wartości zmiennej indeksowanej *IDENT* musi być identyfikatorem zmiennej indeksowanej. Występujące w nawiasach wyrażenia tropowe muszą być typu int. Ich liczba musi odpowiadać liczbie wymiarów tej zmiennej indeksowanej. Ich wartości muszą mieścić się w odpowiednich zakresach tej zmiennej indeksowanej. Typem takiego wyrażenia tropowego jest typ pojedynczych wartości przechowywanych przez tę zmienną indeksowaną. Jego wartością jest wartość tej zmiennej indeksowanej w pozycji o numerze wskazanym przez znajdujące się w nawiasach $[\![$*traceExpr*$]\!]^{+}(,)$.

Wyrażenia *application* **i** *foreignApplication* Ich znaczenie zostało opisane odpowiednio w punktach 6.8 i 6.9.

Liczba całkowita Jest to wyrażenie typu int; jego wartością jest liczba całkowita reprezentowana przez wystąpienie *INTEGER*.

Trop pusty typu specyfikowanego Wartością wyrażenia "`_`" jest pusty ciąg wywołań programów dostępu, czyli *trop pusty*. Typem takiego *traceExpr* jest typ specyfikowany.

Stan początkowy obiektu typu obcego W wyrażeniu "*IDENT* `::` `_`" jedyny identyfikator musi być nazwą typu obcego. Wartością takiego *traceExpr* jest stan początkowy każdego obiektu tego typu obcego. Typem tego wyrażenia jest wskazany przez *IDENT* typ obcy.

Łączenie tropów Kropka "`.`" jest operatorem łączenia tropów. Oba jej argumenty muszą być typu specyfikowanego. Wartością "*traceExpr* `.` *traceExpr*" jest trop powstały ze sklejenia tropów będących wartościami obydwóch *traceExpr*. W tropie wynikowym na początku umieszczamy wywołania z tropu będącego wartością pierwszego *traceExpr*, po nich dopiero następują wywołania z tropu będącego wartością drugiego *traceExpr*. Trop pusty jest elementem neutralnym operacji łączenia tropów.

Nawiasy Ujęcie wyrażenia tropowego w nawiasy ma klasyczne znaczenie. Typ i wartość otoczonego nawiasami *traceExpr* są jednocześnie typem i wartością "`(`*traceExpr*`)`".

Iteracja Za pomocą iteracji możemy wygodnie zapisywać tropy, w których występuje wielokrotnie powtarzające się podwyrażenie. W iteracji:

`[` *traceExpr* `]` `(` *IDENT* `=` *traceExpr* `...` *traceExpr* `)`

pierwsze *traceExpr* musi być typu specyfikowanego, dwa następne muszą być typu int; *IDENT* wprowadza *zmienną sterującą* iteracji. Oznaczmy przez *tex* pierwsze wystąpienie *traceExpr*, przez s zmienną sterującą, zaś przez p i k wartości odpowiednio drugiego i trzeciego *traceExpr*. Jeśli $p > k$, wartością iteracji jest trop pusty. W przeciwnym przypadku, jej wartością jest wartość wyrażenia:

$$tex[s \leftarrow p].tex[s \leftarrow p+1]. \ \ldots \ .tex[s \leftarrow k]$$

prz czym $tex[s \leftarrow r]$ oznacza wyrażenie *tex*, w którym wszystkie wystąpienia zmiennej s zastąpiono liczbą r. Typem iteracji jest zawsze typ specyfikowany.

Operatory arytmetyczne Dwuargumentowe operatory "`+`", "`-`", "`*`", "`/`" i jednoargumentowy "`-`" są odwołaniami do obserwatorów odpowiednich typów obcych. Operatory te są przeciążone, tzn. przyporządkowanie wystąpieniu takiego symbolu programu dostępu, do którego się ten symbol odwołuje, zależy od typów argumentów aktualnych tego wystąpienia. Typem takiego *traceExpr* jest typ wyniku odpowiedniego obserwatora. Wartością jest wynik przekazywany przez ten obserwator dla tych argumentów.

W szczególności typ int wprowadza te pięć obserwatorów; ich działanie jest standardowe.

Wyrażenie *traceTable* Jego znaczenie zostało opisane w punkcie 6.15.

Miejsca definicji wykorzystanych symboli nieterminalnych

Symbol	Punkt	Strona
application	6.8	40
foreignApplication	6.9	42
traceTable	6.15	50

Składnia prezentacyjna

W składni prezentacyjnej iteracji pomija się nawiasy "`( )`" i symbol "`...`". Deklarację zmiennej sterującej rozbija się na dwie cześci "*IDENT* `=` *traceExpr*" i *traceExpr*. Pierwszą z nich umieszcza się w dolnym indeksie, zaś drugą w górnym indeksie zamykającego "`]`":

$$[traceExpr]^{traceExpr}_{IDENT=traceExpr}$$

W składni prezentacyjnej odczytania wartości zmiennej indeksowanej pomija się nawiasy "`[ ]`". Listę wyrażeń indeksujących ($[\![traceExpr]\!]^+(,)$) umieszcza się w dolnym indeksie nazwy zmiennej:

$$IDENT_{[\![traceExpr]\!]^+(,)}$$

Przykład

Następujące równanie występuje w sekcji CANONICAL TRACES specyfikacji stosu (por. przykład w p. 6.5):

```
canonical([PUSH(x[i])](i=1...s)) <=> true
```

Wyrażenie `[PUSH(x[i])](i=1...s)` jest iteracją; jedno z jego podwyrażeń `x[i]` jest odwołaniem do wartości zmiennej indeksowanej. Dla `s = 3`, `x[1] = 3`, `x[2] = 1` i `x[3] = 4`, wartością tej iteracji jest:

```
PUSH(3).PUSH(1).PUSH(4)
```

W dodatku D powyższe równanie zostało zapisane zgodnie z opisanymi powyżej regułami składni prezentacyjnej i z użyciem symboli specjalnych wymienionych w punkcie 6.16:

$$canonical([\mathrm{PUSH}(x_i)]_{i=1}^{s}) \Leftrightarrow \mathbf{true}$$

Iterację `[PUSH(x[i])](i=1...s)` zapisano jako $[\mathrm{PUSH}(x_i)]_{i=1}^{s}$. Odwołanie do zmiennej indeksowanej `x[i]` występuje w postaci x_i.

6.12 Deklaracje zmiennych

varDeclarationList ::=
$[\![varDeclaration]\!]^+(;)$

varDeclaration ::=
$[\![variableDeclaration]\!]^+(,)$: *IDENT*

variableDeclaration ::=
    IDENT
  | *IDENT* `[` $[\![$*traceExpr*$]\!]^{+}(,)$ `...` $[\![$*traceExpr*$]\!]^{+}(,)$ `]`

Wystąpienie *varDeclarationList* oznacza ciąg deklaracji zmiennych. Wartości wszystkich zmiennych wprowadzanych przez pojedyncze *varDeclaration* pochodzą z tego samego zbioru wskazanego przez *IDENT*. Może to być nazwa typu specyfikowanego, typu obcego lub *traces*. Zmienne mogą być *proste* (pierwsza możliwość w regule dla *variableDeclaration*), lub *indeksowane* (druga możliwość). Zmienna indeksowana przechowuje wielowymiarową tablicę wartości. Obie listy ($[\![$*traceExpr*$]\!]^{+}(,)$) definiujące jej rozmiar muszą być jednakowej długości. Wszystkie wyrażenia tropowe, znajdujące się na tych listach muszą być typu int.

Miejsca definicji wykorzystanych symboli nieterminalnych

Symbol	Punkt	Strona
traceExpr	6.11	44

Składnia prezentacyjna

W składni prezentacyjnej w drugiej postaci *variableDeclaration* pomija się nawiasy otaczające deklarację zakresu ("$[\![$*traceExpr*$]\!]^{+}(,)$ `...` $[\![$*traceExpr*$]\!]^{+}(,)$"), samą zaś deklarację umieszcza się w dolnym indeksie po nazwie zmiennej.

Przykład

W specyfikacji zarządcy nazw (por. dodatek F) występuje wyrażenie logiczne:

```
forall R, S : traces [ T != R.PICK^a.S ]
```

Jego fragment `R, S : traces` jest deklaracją zmiennych prostych, które mogą przyjmować wartości będące dowolnymi tropami typu specyfikowanego.

Następująca definicja tropu kanonicznego jest równoważna tej, która występuje w specyfikacji stosu w dodatku D i była rozważana w punkcie 6.5:

```
canonical(T) <=> exists s : int; x[1...s] : elem [ T = [PUSH(x[i])](i=1...s) ]
```

Fraza `x[1...s] : elem` jest deklaracją zmiennej indeksowanej o `s` pozycjach numerowanych liczbami od `1` do `s`. Typem wartości przechowywanej w pojedynczej pozycji jest `elem`.

Powyższe równanie może zostać zapisane zgodnie z opisanymi powyżej regułami składni prezentacyjnej i z użyciem symboli specjalnych wymienionych w punkcie 6.16:

$$canonical(T) \Leftrightarrow \exists s : \mathrm{int}; x_{1\ldots s} : elem[T = [\mathrm{PUSH}(x_i)]_{i=1}^{s}]$$

Deklarację zmiennej indeksowanej `x[1...s] : elem` zapisano jako $x_{1\ldots s} : elem$.

6.13 Wyrażenia logiczne

logExpr ::=
    IDENT
  | *IDENT* `[` $[\![$*traceExpr*$]\!]^{+}(,)$ `]`
  | *application*
  | *foreignApplication*
  | `true`
  | `false`
  | *traceExpr* `=` *traceExpr*
  | *traceExpr* `!=` *traceExpr*
  | *traceExpr* `<` *traceExpr*
  | *traceExpr* `>` *traceExpr*
  | *traceExpr* `>=` *traceExpr*
  | *traceExpr* `<=` *traceExpr*
  | `(` *logExpr* `)`
  | *logExpr* `/\` *logExpr*
  | *logExpr* `\/` *logExpr*
  | *logExpr* `=>` *logExpr*
  | *logExpr* `<=>` *logExpr*
  | `~` *logExpr*
  | `forall` *varDeclarationList constraint* `[` *logExpr* `]`
  | `exists` *varDeclarationList constraint* `[` *logExpr* `]`
  | `exists !` *varDeclarationList constraint* `[` *logExpr* `]`
  | *logTable*

constraint ::=
    ε | `(` *logExpr* `)`

Każde wyliczane *logExpr* musi być albo prawdziwe albo fałszywe. Dotyczy to również tych wyrażeń, w których korzystamy z funkcji częściowych. Możemy unikać wyliczenia takiej funkcji dla argumentów spoza jej dziedziny za pomocą tabelek i więzów (*constraint*) umieszczanych pod kwantyfikatorami. W tabelce ewaluowane jest tylko jedno wyrażenie — to, w którym warunek jest prawdziwy. Więz (*constraint*) pozwala ograniczyć wartościowania zmiennych wprowadzanych przez kwantyfikator. Właściwe wyrażenie pod kwantyfikatorem jest wyliczane jedynie dla tych wartościowań, dla których jest spełniony *constraint*.

Odwołania do zmiennych, *application* **i** *foreignApplication* Semantyka tych wyrażeń logicznych jest oparta na znaczeniu podobnych wyrażeń tropowych. Gdy wyrażenia te występują jako *logExpr*, ich typem musi być bool. Wartości tych wyrażeń mają naturalną interpretację jako wartości logiczne.

Stałe `true` i `false` Wyrażenia `true` ma wartość logiczną prawda, zaś `false` — fałsz.

Równość i różność wyrażeń tropowych Znaczenie symboli "`=`" jest standardowe — "*traceExpr* `=` *traceExpr*" jest prawdziwe, jedynie jeśli wartości obydwu *traceExpr* są sobie równe. Wyrażenie "*traceExpr* `!=` *traceExpr*" jest prawdziwe, jedynie jeśli wartości obydwu *traceExpr* nie są sobie równe.

Operatory porządkowe Symbole "`<`", "`>`", "`>=`", "`<=`" stanowią odwołania do obserwatorów odpowiednich typów obcych. Obserwatory te muszą przekazywać wynik typu bool. Operatory te są przeciążone, tzn. przyporządkowanie wystąpieniu takiego symbolu programu dostępu, do którego się ten symbol odwołuje, zależy od typów argumentów aktualnych tego wystąpienia. Wartością takiego *logExpr* jest wynik przekazywany przez ten obserwator dla tych argumentów, zinterpretowany jako wartość logiczne.

Nawiasy Ujęcie wyrażenia logicznego w nawiasy ma klasyczne znaczenie. Wartość otoczonego nawiasami *logExpr* jest jednocześnie wartością "`(`*logExpr*`)`".

Spójniki logiczne Symbole "`/\`", "`\/`", "`=>`", "`<=>`" i "`~`" oznaczają standardowe operacje logiczne; są to odpowiednio: koniunkcja, alternatywa, implikacja, równoważność i negacja.

Kwantyfikatory Rozważmy wyrażenie logicznego postaci:

quantifier varDeclarationList constraint `[` *logExpr* `]`

w którym *quantifier* jest jednym z symboli: "`forall`", "`exists`" lub "`exists!`". Jego prawdziwość zależy od prawdziwości występującego w nim *logExpr* dla pewnego zbioru wartościowań zmiennych wprowadzonych przez *varDeclarationList*. W skład tego zbioru wchodzą te wartościowania, które spełniają warunek zapisany przez *constraint*. Jeśli *constraint* nie jest obecny, przyjmujemy, że ma postać "`(true)`". Wartość logiczna takiego wyrażenia zależy od rodzaju kwantyfikatora. Jeśli *quantifier* jest symbolem:

`forall` wyrażenie kwantyfikowane jest prawdziwe, jedynie jeśli dla każdego wartościowania z tego zbioru *logExpr* występujące w tym wyrażeniu jest prawdziwe;

`exists` wyrażenie kwantyfikowane jest prawdziwe, jedynie jeśli dla co najmniej jednego wartościowania z tego zbioru *logExpr* występujące w tym wyrażeniu jest prawdziwe; to *logExpr* musi być dobrze określone dla wszystkich wartościowań z tego zbioru;

`exists!` wyrażenie kwantyfikowane jest prawdziwe, jedynie jeśli dla dokładnie jednego wartościowania z tego zbioru *logExpr* występujące w tym wyrażeniu jest prawdziwe; to *logExpr* musi być dobrze określone dla wszystkich wartościowań z tego zbioru.

Wyrażenie *logTable* Jego znaczenie zostało opisane w punkcie 6.15.

Miejsca definicji wykorzystanych symboli nieterminalnych

Symbol	Punkt	Strona
application	6.8	40
foreignApplication	6.9	42
logTable	6.15	50
traceExpr	6.11	44
varDeclarationList	6.12	46

Przykład

W przykładzie w punkcie 6.12 rozważano następującą równoważną definicję tropu kanonicznego stosu:

```
canonical(T) <=> exists s : int; x[1...s] : elem [ T = [PUSH(x[i])](i=1...s) ]
```

Zawiera ona przykład użycia kwantyfikatora egzystencjalnego. Wyrażenie po prawej stronie symbolu `<=>` jest prawdziwe dla tropu `T` jedynie, jeśli istnieje wartościowanie dla zmiennej prostej `s` i zmiennej indeksowanej `x`, przy którym wyrażenie `T = [PUSH(x[i])](i=1...s)` jest prawdziwe.

Dzielenie liczb jest operacją częściową — wynik dzielenia przez zero nie istnieje. Warunek ograniczający rozważane pod kwantyfikatorem wartościowania (*constraint*) pozwala bezpiecznie posługiwać się funkcjami częściowymi. Następująca formuła wyraża dobrze znaną zasadę:

```
forall x, y, z : int (x != 0) [ y = z => y / x = z / x ]
```

Wartość formuły `y = z => y / x = z / x` jest wyznaczana jedynie wtedy, gdy prawdziwy jest warunek `x != 0`. Dzięki temu unikamy dzielenia przez zero.

6.14 Priorytet i łączność operatorów

Wszystkie operatory dwuargumentowe są lewostronnie łączne. Priorytety prezentuje poniższa tabela. Każdy klatka zawiera operatory o tym samym priorytecie. Niżej położone operatory mają niższy priorytet.

^
jednoargumentowy -
+ -
* /
.
= < > >= <= !=
~
/\
\/
=> <=>

6.15 Jednowymiarowe normalne tabele funkcyjne

traceTable ::=
`table` ⟦*traceRow*⟧$^+$ `end`

traceRow ::=
`#` *logExpr* `#` *traceExpr* `#`

logTable ::=
`table` ⟦*logRow*⟧$^+$ `end`

logRow ::=
`#` *logExpr* `#` *logExpr* `#`

signalTable ::=
`table` ⟦*signalRow*⟧$^+$ `end`

signalRow ::=
`#` *logExpr* `#` *signalExpr* `#`

Wszystkie trzy rodzaje tabel (*logTable*, *signalTable* i *traceTable*) mają taką samą semantykę. Dla każdego wartościowania zmiennych wolnych tabeli, w kontekście, w którym występuje tabela, musi być spełniony następujący warunek: w dokładnie jednym wierszu tej tabeli (*logRow*, *signalRow* lub *traceRow*) *logExpr* po pierwszym "`#`" jest prawdziwe. Wartością tej tabeli jest wartość wyrażenia (*logExpr*, *signalExpr* lub *traceExpr*) znajdującego się po drugim symbolu "`#`" w tym samym wierszu.

Na *traceTable* są nałożone dodatkowe więzy kontekstowe dotyczące typów — we wszystkich wierszach (*traceRow*) typ *traceExpr* występującego po drugim "`#`" musi być taki sam. Typ ten jest typem *traceTable* jako wyrażenia tropowego.

Miejsca definicji wykorzystanych symboli nieterminalnych

Symbol	Punkt	Strona
logExpr	6.13	48
signalExpr	6.10	43
traceExpr	6.11	44

Składnia prezentacyjna

W składni prezentacyjnej *logTable*, *signalTable* i *traceTable* mogą zostać zapisane w formie klasycznej tabeli, która ma dwie kolumny; nagłówkiem pierwszej z nich jest słowo *Condition*, zaś drugiej — *Value*. Każde z wystąpień jednego z symboli *logRow*, *signalRow* lub *traceRow* staje się wierszem tabeli. W każdym wierszu, w pierwszej kolumnie umieszczamy *logExpr* występujące po pierwszym "`#`", w drugiej kolumnie znajdzie się wyrażenie zza drugiego "`#`" (zależnie od rodzaju tabeli będzie to *logExpr*, *signalExpr* lub *traceExpr*). W tabelach w składni prezentacyjnej nie pojawiają się symbole terminalne użyte w gramatyce (`table`, `#`, `end`).

Przykład

W sekcji SEMANTICS OF MUTATORS specyfikacji zarządcy nazw (por. dodatek F) znajduje się równanie zapisane zgodnie z powyżej wymienionymi regułami składni prezentacyjnej:

$$signals(T; \mathrm{PICK}()) =$$

Condition	Value
$legal(T; \mathrm{PICK}())$	[ ]
$\neg legal(T; \mathrm{PICK}())$	[NO_MORE_NAMES()]

Posługując się jedynie znakami ASCII możemy je zapisać następująco:

```
signals(T; PICK()) = table
                       #  legal(T; PICK()) #         [ ]         #
                       # ~legal(T; PICK()) # [ NO_MORE_NAMES() ] #
                     end
```

Wartością tabeli występującej w tym równaniu jest pusta lista sygnałów `[ ]`, gdy zachodzi warunek `legal(T; PICK())`. Natomiast jeśli prawdziwe jest `~legal(T; PICK())`, to wartością tej tabeli jest lista `[ NO_MORE_NAMES() ]`. Tabela ta jest poprawna, ponieważ dla każdego wartościowania zmiennej `T`, dokładnie jeden z warunków występujących w pierwszej kolumnie jest prawdziwy.

6.16 Składnia prezentacyjna niektórych symboli

Jeśli dostępny jest zestaw odpowiednich symboli, można korzystać z następujących odpowiedników symboli wprowadzonych w niniejszym rozdziale:

Symbol	Odpowiednik
`->`	$\rightarrow$
`!=`	$\neq$
`>=`	$\geq$
`<=`	$\leq$
`/\`	$\wedge$
`\/`	$\vee$
`=>`	$\Rightarrow$
`<=>`	$\Leftrightarrow$
`~`	$\neg$
`forall`	$\forall$
`exists`	$\exists$
`^`	$\uparrow$

Rozdział 7

Formalny opis metody TAM 2000

Formalny opis metody TAM 2000 składa się z trzech części: składni, semantyki statycznej i semantyki dynamicznej.

Składnia specyfikacji tropowych jest zdefiniowana za pomocą gramatyki bezkontekstowej. Została ona omówiona w punkcie 7.1. Pełny tekst tej gramatyki znajduje się w dodatku A.

Sposób definicji obu semantyk oraz zastosowane konwencje notacyjne są przedstawione w punkcie 7.2.

Semantyka statyczna określa warunki kontekstowej poprawności specyfikacji i reguły wyprowadzania typów dla poszczególnych konstrukcji syntaktycznych. Jej głównym składnikiem jest zestaw reguł statycznej poprawności specyfikacji. Punkt 7.3 wprowadza podstawowe pojęcia semantyki statycznej, a także wyjaśnia znaczenie przykładowej reguły. Pełny zestaw reguł statycznej poprawności specyfikacji został umieszczony w dodatku B.

Semantyka dynamiczna określa zachowanie obiektów definiowanego w specyfikacji modułu. Jej podstawowym elementem jest zestaw reguł wyliczania wartości wyrażeń. Punkt 7.4 definiuje semantykę dynamiczną w opraciu o fakty, które można wywnioskować korzystając z tych reguł. Punkt ten zawiera także wyjaśnienie znaczenia przykładowej reguły. Komplet reguł wyliczania wartości wyrażeń stanowi treść dodatku C.

Uzasadnienie spójności semantyki opisanej w punktach 7.2–7.4 znajduje się w punkcie 7.5.

7.1 Składnia specyfikacji w TAM 2000

Składnia specyfikacji tropowych przedstawiona w rozdziale 6 jest jednoznaczna[1], jednak nie ma własności LALR(1) [WG89]. Po wprowadzeniu kilku modyfikacji i dodatkowych symboli nieterminalnych, możemy otrzymać gramatykę spełniającą warunek LALR(1)[1], która generuje ten sam język. Taka gramatyka znajduje się w dodatku A w formie plików wejściowych dla programów **yacc** i **lex** [Gaw93].

Przyczyną, dla której składnia zaprezentowana w rozdziale 6 nie należy do klasy LALR(1), są konflikty *reduce-reduce* wynikające ze wspólnych wariantów w regułach dla symboli *traceExpr* i *logExpr*. Tymi wariantami są: zmienna, zmienna indeksowana, *application* i *foreignApplication*. Aby wyeliminować te konflikty byliśmy zmuszeni wprowadzić dwa dodatkowe symbole (*commonExpr* i *commonTable*) i nadać redukcjom do tych symboli większy priorytet niż redukcjom do *logExpr*, *traceExpr*, *logTable* i *traceTable*. Po tych modyfikacjach otrzymaliśmy gramatykę, która znajduje się w dodatku A. Konflikty zostały wyeliminowane,

[1] Z dokładnością do zdefiniowania priorytetów i łączności operatorów dwuargumentowych.

ponieważ w sytuacjach gdy nie wiadomo do jakiego symbolu redukować (*logExpr* czy *traceExpr*; *logTable* czy *traceTable*), następuje wysokopriorytetowa redukcja do *commonExpr* lub *commonTable*.

7.2 Wprowadzenie do definicji semantyki

Definicja semantyki statycznej i dynamicznej metody TAM 2000 zostanie przedstawiona w formie tzw. *semantyki naturalnej* [Kah88], zwanej też *strukturalną semantyką operacyjną* [Plo81]. Definicja ta jest zbiorem reguł wnioskowania, które określają znaczenie podstawowych elementów specyfikacji, oraz pokazują jak semantyka wyrażenia złożonego zależy od semantyki jego składowych. Reguły mają postać:

$$\frac{\text{przesłanka}_1 \quad \ldots \quad \text{przesłanka}_n}{\text{wniosek}_1 \quad \ldots \quad \text{wniosek}_m}$$

Fakty (przesłanki i wnioski) są zapisywane w postaci:

$$\Phi \vdash e \Longrightarrow \alpha$$

Taki zapis oznacza, że w środowisku Φ wyrażenie e ma własność α. W zapisie faktu po lewej stronie symbolu "$\vdash$" można pominąć środowisko. Napis "$\vdash e \Longrightarrow \alpha$" oznacza, że e ma własność α w pustym środowisku. Intuicyjnie, w takim środowisku nie jest zdefiniowany żaden symbol.

Zapis "$\Phi[x := t]$" oznacza środowisko, w którym wartość wyrażenia x została zastąpiona przez t.

Zostaną podane reguły dla każdego prawidła składni specyfikacji. W zapisie elementów składni użyjemy tych samych konwencji, jak w znajdującej się w rozdziale 6 definicji składni. Różne wystąpienia tego samego symbolu nieterminalnego w jednej regule będą rozróżniane za pomocą dolnych indeksów.

Napis **Idents** oznacza zbiór wszystkich poprawnych identyfikatorów; $\mathcal{N}$ — zbiór liczb naturalnych; $\mathcal{Z}$ — zbiór liczb całkowitych. Operatory na zbiorach $\cap, \cup, -, \times$ mają tradycyjne znaczenie. Wyrażenie X^* oznacza zbiór wszystkich skończonych list zbudowanych z elementów zbioru X; $[X \to Y]$ oznacza zbiór wszystkich funkcji ze zbioru X w zbiór Y.

Jeśli dla każdego $i = 1, 2, \ldots, n$, e_i należy do X_i, to krotka $\langle e_1, \ldots, e_n \rangle$ należy do $X_1 \times \ldots \times X_n$. Jeśli dla każdego $i = 1, 2, \ldots, n$, e_i należy do X, to lista $[e_1, \ldots, e_n]$ należy do X^*. Operatorem sklejania list jest "$\bullet$"; lista pusta ("[ ]") jest jego elementem neutralnym.

7.3 Semantyka statyczna TAM 2000

7.3.1 Środowisko statyczne

Fakty semantyki statycznej są wyprowadzane w *środowisku statycznym*, które jest uporządkowaną czwórką:

$$\Phi = (\textbf{specified}, \textbf{programs}, \textbf{functions}, \textbf{variables})$$

Do składników Φ możemy odwoływać się za pomocą kwalifikatorów, np. Φ.**specified**.

Pierwszy element środowiska wskazuje typ specyfikowany:

$$\Phi.\textbf{specified} \in \textbf{Idents}$$

Drugi element środowiska zawiera informację o znanych programach dostępu i sygnałach poszczególnych typów. Jest to funkcja ze zbioru nazw typów w zbiór funkcji przypisujących każdej nazwie jego sygnaturę, rozszerzony o specjalny symbol **error**. Jeśli dla pewnego identyfikatora wartość funkcji równa się **error**, ten identyfikator nie został wprowadzony. Formalnie:

$$\begin{aligned}\Phi.\mathbf{programs} \in [\ \mathbf{Idents} \to \{\mathbf{error}\} \cup [\ \mathbf{Idents} \ &\to \ \{\mathbf{error}\} \\ &\cup \ (\{\mathbf{m},\mathbf{s}\} \times \mathbf{Idents}^*) \\ &\cup \ (\{\mathbf{m},\mathbf{o}\} \times \mathbf{Idents}^* \times \mathbf{Idents})\ ]\ ]\end{aligned}$$

Przy czym **m**, **o**, **s** oznaczają odpowiednio modyfikator, obserwator i sygnał.

Trzeci składnik środowiska przechowuje informacje o znanych funkcjach pomocniczych. Jest to funkcja, która przyporządkowuje identyfikatorom sygnatury odpowiadających im funkcji pomocniczych, lub **error**, w przypadku gdy identyfikator nie jest nazwą żadnej funkcji pomocniczej. Formalnie:

$$\Phi.\mathbf{functions} \in [\ \mathbf{Idents} \to \{\mathbf{error}\} \cup (\mathbf{Idents}^* \times \mathbf{Idents})\ ]$$

Czwarty składnik środowiska przechowuje informacje o zmiennych. Jest to funkcja, która przyporządkowuje identyfikatorom pary: typ i liczba wyrażeń indeksujących. Dla zmiennych prostych drugi składnik jest równy zero, dla indeksowanych wynosi co najmniej jeden. Dla nieznanych zmiennych, wartością środowiska jest **error**. Formalnie:

$$\Phi.\mathbf{variables} \in [\ \mathbf{Idents} \to \{\mathbf{error}\} \cup (\mathbf{Idents} \times \mathcal{N})\ ]$$

Wprowadzimy relację częściowego porządku środowisk statycznych, oznaczaną przez "$\preceq$". Intuicyjnie środowisko Φ jest nie większe niż Ψ, jedynie jeśli Φ i Ψ mają takie same pierwsze trzy składniki (**specified**, **programs**, **functions**), zaś $\Psi.\mathbf{variables}$, wprowadza więcej zmiennych niż $\Phi.\mathbf{variables}$; zmienne zdefiniowane w obu środowiskach mają te same typy. Formalnie:

$$\begin{aligned}\Phi \preceq \Psi \ &\stackrel{\text{df}}{\Leftrightarrow}\ \Phi.\mathbf{specified} = \Psi.\mathbf{specified} \\ &\wedge\ \Phi.\mathbf{programs} = \Psi.\mathbf{programs} \\ &\wedge\ \Phi.\mathbf{functions} = \Psi.\mathbf{functions} \\ &\wedge\ \forall x(\Phi.\mathbf{variables}(x) = \mathbf{error} \vee \Phi.\mathbf{variables}(x) = \Psi.\mathbf{variables}(x))\end{aligned}$$

Pustym środowiskiem statycznym nazwiemy takie, które nie wprowadza żadnych zmiennych, ani funkcji pomocniczych, zaś jedyne programy dostępu są zdefiniowane przez typ bool ("TRUE", "FALSE") i int ("SUCC", "PRED", "+", "−", "∗", "/", "unary −", "<", ">", ">=", "<="). Formalnie, Φ jest pustym środowiskiem jedynie, jeśli spełnia następujące warunki:

$$\begin{aligned}\forall x\ (\ &(\Phi.\mathbf{variables}(x) = \mathbf{error} \wedge \Phi.\mathbf{functions}(x) = \mathbf{error}) \\ \wedge\ &(x \neq \text{"bool"} \wedge x \neq \text{"int"} \Leftrightarrow \Phi.\mathbf{programs}(x) = \mathbf{error}) \\ \wedge\ &(x = \text{"bool"} \ \Rightarrow\ (\forall y \in \{\text{"TRUE"},\text{"FALSE"}\}\ (\Phi.\mathbf{programs}(x)(y) = \langle \mathbf{m}, [\ ]\rangle) \\ &\qquad\qquad \wedge\ \forall y \notin \{\text{"TRUE"},\text{"FALSE"}\}\ (\Phi.\mathbf{programs}(x)(y) = \mathbf{error}))) \\ \wedge\ &(x = \text{"int"} \ \Rightarrow\ (\forall y \in \{\text{"SUCC"},\text{"PRED"}\}\ \ (\Phi.\mathbf{programs}(x)(y) = \langle \mathbf{m}, [\ ]\rangle) \\ &\qquad\qquad \wedge\ \forall y \in \{\text{"+"},\text{"−"},\text{"∗"},\text{"/"}\} \\ &\qquad\qquad\qquad (\Phi.\mathbf{programs}(x)(y) \ = \ \langle \mathbf{o}, [\text{"int"},\text{"int"}],\text{"int"}\rangle) \\ &\qquad\qquad \wedge\ \Phi.\mathbf{programs}(x)(\text{"unary −"}) \ = \ \langle \mathbf{o}, [\text{"int"}],\text{"int"}\rangle \\ &\qquad\qquad \wedge\ \forall y \in \{\text{"<"},\text{">"},\text{">="},\text{"<="}\} \\ &\qquad\qquad\qquad (\Phi.\mathbf{programs}(x)(y) \ = \ \langle \mathbf{o}, [\text{"int"},\text{"int"}],\text{"bool"}\rangle) \\ &\qquad\qquad \wedge\ \forall y \notin \{\text{"SUCC"},\text{"PRED"},\text{"+"},\text{"−"},\text{"∗"},\text{"/"},\text{"unary −"},\text{"<"} \\ &\qquad\qquad\qquad \text{">"},\text{">="},\text{"<="}\} \quad (\Phi.\mathbf{programs}(x)(y) = \mathbf{error}))))\end{aligned}$$

7.3.2 Reguły statycznej poprawności specyfikacji

W dodatku B znajduje się komplet reguł statycznej poprawności specyfikacji. Dla każdej produkcji gramatyki istnieje co najmniej jedna reguła wyznaczająca semantykę wyrażeń wyprowadzonych zgodnie z tą produkcją. Struktura i kolejność punktów w dodatku B odpowiada prezentacji metody z rozdziału 6. W tym punkcie przestawimy przykład wyjaśniający konstrukcję i znaczenie tych reguł. Oto reguła wyprowadzania typu dla operacji sklejania tropów:

$$\boxed{\mathit{traceExpr} ::= \mathit{traceExpr}_{left} \ . \ \mathit{traceExpr}_{right}}$$

$$\frac{\Phi \vdash \mathit{traceExpr}_{left} \Longrightarrow \Phi.\mathbf{specified} \quad \Phi \vdash \mathit{traceExpr}_{right} \Longrightarrow \Phi.\mathbf{specified}}{\Phi \vdash \mathit{traceExpr} \Longrightarrow \Phi.\mathbf{specified}}$$

Wyrażenie tropowe (*traceExpr*) skonstruowane zgodnie z umieszczoną w ramce produkcją składa się z dwóch podwyrażeń oznaczonych tutaj przez $\mathit{traceExpr}_{left}$ i $\mathit{traceExpr}_{right}$. Reguła ta stanowi, że jeśli dla obu podwyrażeń możemy w środowisku Φ wyprowadzić typ Φ.**specified** (tzn. typ specyfikowy), to wyrażenie "$\mathit{traceExpr}_{left}$. $\mathit{traceExpr}_{right}$" (we wniosku reguły oznaczone przez *traceExpr*) jest poprawne i jego typem jest również Φ.**specified**.

7.3.3 Statyczna poprawność specyfikacji

Specyfikacja *specification* jest *statycznie poprawna* jedynie jeśli, za pomocą reguł przedstawionych w dodatku B można wyprowadzić fakt:

$$\vdash \mathit{specification} \Longrightarrow \Phi$$

Semantyką statyczną specyfikacji nazywamy występujące w powyższej definicji środowisko Φ.

7.4 Semantyka dynamiczna TAM 2000

7.4.1 Zbiór wszystkich tropów

Niniejszy punkt definiuje najważniejszą dziedzinę semantyczną TAM 2000 — zbiór wszystkich tropów. Przyjmijmy, że specyfikacja, której semantykę będziemy definiować jest statycznie poprawna i Φ oznacza jej statyczną semantykę (por. p. 7.3.2). Zakładamy, że znane są denotacje specyfikacji wszystkich typów obcych występujących w rozważanej specyfikacji. Oznaczmy przez $\mathcal{S}_{type}$ denotację specyfikacji typu *type*. Postać denotacji specyfikacji jest opisana w punkcie 7.4.2. W niniejszym punkcie będziemy korzystać jedynie z jednego składnika denotacji specyfikacji typów obcych — **states**, który oznacza zbiór stanów obiektów danego typu obcego. Zdefiniujemy dwie dziedziny semantyczne:

Events jest zbiorem wszystkich zdarzeń powodowanych przez wywołania modyfikatorów; (element **Events** określa nazwę modyfikatora, jego argumenty oraz niedeterministyczny wynik, jeśli dany modyfikator przekazuje taki wynik);

Traces jest zbiorem wszystkich tropów, czyli ciągów zdarzeń;

Formalnie, **Events** i **Traces** są najmniejszymi zbiorami, które łącznie spełniają poniższe warunki:

1. jeśli $\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(id) = \langle \mathbf{m}, [t_1, \ldots, t_n] \rangle$,
   oraz dla każdego $j = 1, \ldots, n$,
   $V_j \in \mathcal{S}_{t_j}.\mathbf{states}$ gdy $t_j \neq \Phi.\mathbf{specified}$, zaś
   $V_j \in \mathbf{Traces}$ gdy $t_j = \Phi.\mathbf{specified}$,
   to $id(V_1, \ldots, V_n) \in \mathbf{Events}$

2. jeśli $\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(id) = \langle \mathbf{m}, [t_1, \ldots, t_n], t_{n+1} \rangle$,
   oraz dla każdego $j = 1, \ldots, n+1$,
   $V_j \in \mathcal{S}_{t_j}.\mathbf{states}$ gdy $t_j \neq \Phi.\mathbf{specified}$, zaś
   $V_j \in \mathbf{Traces}$ gdy $t_j = \Phi.\mathbf{specified}$,
   to $id(V_1, \ldots, V_n)^\frown V_{n+1} \in \mathbf{Events}$

3. $\mathbf{Traces} = \mathbf{Events}^*$

7.4.2 Denotacja specyfikacji

Semantyka dynamiczna dla każdej specyfikacji wyznacza jej denotację, która jest uporządkowaną siódemką:

$$\mathcal{S} = (\mathbf{states}, \mathbf{initial}, \mathbf{legal}, \mathbf{signals}, \mathbf{feasible}, \mathbf{transition}, \mathbf{result})$$

Do składników $\mathcal{S}$ możemy odwoływać się za pomocą kwalifikatorów, np. $\mathcal{S}.\mathbf{states}$. Poniżej, w kolejnych punktach, omówimy wszystkie składniki denotacji specyfikacji.

7.4.2.1 states

Pierwszy element denotacji to zbiór tropów kanonicznych:

$$\mathcal{S}.\mathbf{states} \subseteq \mathbf{Traces}$$

Zbiór tropów kanonicznych zostanie użyty w definicji następujących dziedzin semantycznych (występuje on w poniższych definicjach jako $\mathcal{S}_{t_j}.\mathbf{states}$ gdy $t_j = \Phi.\mathbf{specified}$):

MutatorInvocs jest zbiorem wszystkich wywołań modyfikatorów, w których argumenty typu specyfikowanego są tropami kanonicznymi; elementy **MutatorInvocs** nie zawierają ewentualnego niedeterministycznego wyniku modyfikatora;

ObserverInvocs jest zbiorem wszystkich wywołań obserwatorów, w których wszystkie argumenty typu specyfikowanego są tropami kanonicznymi;

Signals jest zbiorem wszystkich sygnałów, w których wszystkie argumenty typu specyfikowanego są tropami kanonicznymi;

Formalnie, **MutatorInvocs**, **ObserverInvocs** i **Signals** są najmniejszymi zbiorami, które łącznie spełniają następujące warunki:

1. jeśli $\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(id) = \langle \mathbf{m}, [t_1, \ldots, t_n] \rangle$,
   oraz dla każdego $j = 1, \ldots, n$, $V_j \in \mathcal{S}_{t_j}.\mathbf{states}$
   to $id(V_1, \ldots, V_n) \in \mathbf{MutatorInvocs}$

2. jeśli $\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(id) = \langle \mathbf{m}, [t_1, \ldots, t_n], t_{n+1} \rangle$,
   oraz dla każdego $j = 1, \ldots, n+1$, $V_j \in \mathcal{S}_{t_j}.\mathbf{states}$
   to $id(V_1, \ldots, V_n) \in \mathbf{MutatorInvocs}$

3. jeśli $\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(id) = \langle \mathbf{o}, [t_1, \ldots, t_n], t_{n+1} \rangle$,
   oraz dla każdego $j = 1, \ldots, n$, $V_j \in \mathcal{S}_{t_j}.\mathbf{states}$
   to $id(V_1, \ldots, V_n) \in \mathbf{ObserverInvocs}$

4. jeśli $\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(id) = \langle \mathbf{s}, [t_1, \ldots, t_n] \rangle$,
   oraz dla każdego $j = 1, \ldots, n$, $V_j \in \mathcal{S}_{t_j}.\mathbf{states}$
   to $id(V_1, \ldots, V_n) \in \mathbf{Signals}$

7.4.2.2 initial

Drugi element denotacji to trop kanoniczny reprezentujący stan początkowy obiektu typu specyfikowanego.

$$\mathcal{S}.\mathbf{initial} \in \mathcal{S}.\mathbf{states}$$

Zwykle będzie to trop pusty. Jeśli nie jest to trop pusty, trop pusty nie może być tropem kanonicznym.

7.4.2.3 legal

Trzeci element denotacji to zbiór legalnych wywołań programów dostępu. Jest to podzbiór zbioru wszystkich wywołań[2].

$$\mathcal{S}.\mathbf{legal} \subseteq (\mathcal{S}.\mathbf{states} \times \mathbf{MutatorInvocs}) \cup \mathbf{ObserverInvocs}$$

Zbiór legalnych wywołań programów dostępu zostanie użyty w definicji następującej dziedziny semantycznej:

LegalEvents jest zbiorem wszystkich par, trop kanoniczny i zdarzenie spowodowane przez legalne wywołanie modyfikatora, gdy stan obiektu jest reprezentowany przez ten trop kanoniczny;

Formalnie, **LegalEvents** jest najmniejszym zbiorem, który spełnia następujące warunki:

1. jeśli $\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(id) = \langle \mathbf{m}, [t_1, \ldots, t_n] \rangle$,
   oraz dla każdego $j = 1, \ldots, n$, $V_j \in \mathcal{S}_{t_j}.\mathbf{states}$
   a także $\langle T, id(V_1, \ldots, V_n) \rangle \in \mathcal{S}.\mathbf{legal}$
   to $\langle T, id(V_1, \ldots, V_n) \rangle \in \mathbf{LegalEvents}$

2. jeśli $\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(id) = \langle \mathbf{m}, [t_1, \ldots, t_n], t_{n+1} \rangle$,
   oraz dla każdego $j = 1, \ldots, n+1$, $V_j \in \mathcal{S}_{t_j}.\mathbf{states}$
   a także $\langle T, id(V_1, \ldots, V_n) \in \mathcal{S}.\mathbf{legal} \rangle$
   to $\langle T, id(V_1, \ldots, V_n) \hat{} V_{n+1} \rangle \in \mathbf{LegalEvents}$

[2]Można rozważać funkcję charakterystyczną tego podzbioru; wybraliśmy zbiór ze względu na wygodę opisu.

7.4.2.4 signals

Czwarty element denotacji to funkcja przyporządkowująca wszystkim wywołaniom programów dostępu listę sygnałów wysyłanych w odpowiedzi na te wywołania. Funkcja ta jest również określona dla wywołań nieleganych.

$$\mathcal{S}.\mathbf{signals} \in [\ ((\mathcal{S}.\mathbf{states} \times \mathbf{MutatorInvocs}) \cup \mathbf{ObserverInvocs}) \rightarrow \mathbf{Signals}^*\ ]$$

7.4.2.5 feasible

Piąty element denotacji to zbiór zdarzeń sensownych. Jest to podzbiór zbioru wszystkich par, trop kanoniczny i zdarzenie spowodowane przez legalne wywołanie modyfikatora. Informacja niesiona przez **feasible** jest istotna jedynie dla modyfikatorów przekazujących niedeterministyczny wynik. Zdarzenie jest sensowne wtedy, gdy wymieniony w nim niedeterministyczny wynik jest poprawny. Wywołania modyfikatorów, które nie przekazują niedeterministycznego wyniku, zawsze odpowiadają zdarzeniom sensownym.

$$\mathcal{S}.\mathbf{feasible} \subseteq \mathbf{LegalEvents}$$

7.4.2.6 transition

Szósty element denotacji to funkcja rozszerzenia. Przekształca ona pary, trop kanoniczny i sensowne zdarzenie, na tropy kanoniczne.

$$\mathcal{S}.\mathbf{transition} \in [\ \mathcal{S}.\mathbf{feasible} \rightarrow \mathcal{S}.\mathbf{states}\ ]$$

7.4.2.7 result

Siódmy element denotacji to funkcja wyniku. Przekształca ona wywołania obserwatorów na wyniki generowane przez te wywołania. Jej przeciwdziedziną jest suma zbiorów stanów wszystkich typów znanych w specyfikacji (specyfikowanego i obcych).

$$\mathcal{S}.\mathbf{result} \in [\ \mathbf{ObserverInvocs} \rightarrow \bigcup_{\Phi.\mathbf{programs}(id) \neq \mathbf{error}} \mathcal{S}_{id}.\mathbf{states}\ ]$$

7.4.3 Specyfikacje sparametryzowane

Specyfikacja sparametryzowana ma semantykę funktorową. Znaczeniem takiej specyfikacji jest funkcja, która przekształca aktualne parametry-wartości i denotacje specyfikacji typów będących aktualnymi parametrami-typami w semantykę niesparametryzowanej specyfikacji.

7.4.4 Więzy narzucone przez funkcję redukującą

Dziedziną funkcji redukującej (por. p. 5.5) jest zbiór tropów sensownych. Znając denotację specyfikacji można te dwa pojęcia zdefiniować. Oznaczmy przez *reduce* funkcję redukującą, przez *feasibleTraces* — zbiór tropów sensownych, zaś przez $\mathcal{S}$ — denotację rozważanej specyfikacji. Funkcja:

$$\mathit{reduce} \in [\ \mathit{feasibleTraces} \rightarrow \mathcal{S}.\mathbf{states}\ ]$$

ma następującą rekurencyjną definicję:

$$\begin{array}{lcl} \mathit{reduce}([\,]) & = & \mathcal{S}.\mathbf{initial} \\ \mathit{reduce}(T \bullet [E]) & = & \mathcal{S}.\mathbf{transition}(\langle \mathit{reduce}(T), E\rangle) \end{array}$$

Zaś *feasibleTraces* jest najmniejszym podzbiorem **Traces**, o następujących własnościach:

$$[\,] \in \mathit{feasibleTraces}$$

$$T \in \mathit{feasibleTraces} \quad \wedge \quad \langle \mathit{reduce}(T), E\rangle \in \mathcal{S}.\mathbf{feasible} \quad \Rightarrow \quad T \bullet [E] \in \mathit{feasibleTraces}$$

Dla każdej specyfikacji następujące warunki muszą być spełnione:

$$\begin{array}{lcl} & \mathcal{S}.\mathbf{states} & \subseteq \quad \mathit{feasibleTraces} \\ \forall \quad T \in \mathcal{S}.\mathbf{states} & & (\mathit{reduce}(T) = T) \end{array}$$

7.4.5 Więzy narzucone przez obserwacyjną równoważność

Dwa różne tropy kanoniczne nie mogą być obserwacyjnie równoważne (por. p. 5.5). Oznacza to, że dla każdej pary, różnych tropów kanonicznych, T_1 i T_2, musi być spełniony co najmniej jeden z poniższych warunków:

1. istnieje trop T taki, że $T_1 \bullet T \in \mathit{feasibleTraces}$ nie jest równoważne $T_2 \bullet T \in \mathit{feasibleTraces}$.
2. istnieje trop T i zdarzenie E takie, że:
   (a) $T_1 \bullet T \in \mathit{feasibleTraces}$ i $T_2 \bullet T \in \mathit{feasibleTraces}$, oraz
   (b) $\mathcal{S}.\mathbf{signals}(\langle \mathit{reduce}(T_1 \bullet T), E\rangle) \neq \mathcal{S}.\mathbf{signals}(\langle \mathit{reduce}(T_2 \bullet T), E\rangle)$
3. istnieje trop T i dwa wywołania pewnego obserwatora — O_1 i O_2 takie, że:
   (a) $T_1 \bullet T \in \mathit{feasibleTraces}$ i $T_2 \bullet T \in \mathit{feasibleTraces}$, oraz
   (b) O_2 może powstać z O_1 poprzez zastąpienie w O_1 niektórych argumentów o wartości $\mathit{reduce}(T_1 \bullet T)$ przez $\mathit{reduce}(T_2 \bullet T)$ i niektórych argumentów o wartości $\mathit{reduce}(T_2 \bullet T)$ przez $\mathit{reduce}(T_1 \bullet T)$, oraz
   (c) co najmniej jedno zdanie z wymienionych poniżej jest prawdziwe:
      i. $O_1 \in \mathcal{S}.\mathbf{legal}$ nie jest równoważne $O_2 \in \mathcal{S}.\mathbf{legal}$, lub
      ii. $\mathcal{S}.\mathbf{signals}(O_1) \neq \mathcal{S}.\mathbf{signals}(O_2)$, lub
      iii. $\mathcal{S}.\mathbf{result}(O_1) \neq \mathcal{S}.\mathbf{result}(O_2)$.

7.4.6 Wartościowanie zmiennych

Wartościowanie zmiennych występujących w specyfikacji tropowej musi uwzględniać zmienne indeksowane i dlatego jego konstrukcja nieznacznie odbiega od klasycznego pojęcia wartościowania zmiennych. Wartościowanie jest funkcją, której dziedzina składa się z par, identyfikator i ciąg wartości indeksów. Przeciwdziedziną tej funkcji jest suma zbioru wszystkich tropów specyfikowanych obiektów i zbiorów stanów wszystkich typów obcych. Do przeciwdziedziny należy jeszcze **error**, który reprezentuje wartość nieokreślonych zmiennych. Jeśli $\mathcal{W}$ jest wartościowaniem, to

$$\mathcal{W} \in [\ (\mathbf{Idents} \times \mathcal{Z}^*) \rightarrow (\{\mathbf{error}\} \cup \mathbf{Traces} \cup \bigcup_{\substack{\Phi.\mathbf{programs}(id) \neq \mathbf{error} \\ \wedge\ id \neq \Phi.\mathbf{specified}}} \mathcal{S}_{id}.\mathbf{states})\ ]$$

Zapis "$\mathcal{W}[\langle var, l\rangle := x]$" oznacza wartościowanie $\mathcal{W}$, w którym wartość zmiennej *var* dla listy indeksów l zastąpiono przez x.

Wprowadzimy relację częściowego porządku wartościowań, oznaczaną przez "$\preceq$". Intuicyjnie wartościowanie $\mathcal{W}$ jest nie większe niż $\mathcal{Q}$, jedynie jeśli $\mathcal{Q}$ określa więcej zmiennych niż $\mathcal{W}$. Zmienne określone w obu wartościowaniach mają te same wartości. Formalnie:

$$\mathcal{W} \preceq \mathcal{Q} \quad \stackrel{\text{df}}{\Leftrightarrow} \quad \forall x\, l\ (\mathcal{W}(x,l) = \mathbf{error} \vee \mathcal{W}(x,l) = \mathcal{Q}(x,l))$$

7.4.7 Reguły wyliczania wartości wyrażeń

W dodatku C znajduje się komplet reguł wyliczania wartości wyrażeń, które mogą pojawić się w specyfikacji tropowej. Dla każdej produkcji generującej wyrażenia (tropowe — *traceExpr*, logiczne — *logExpr*, oraz sygnałowe — *signalExpr*) istnieje co najmniej jedna reguła wyznaczająca semantykę wyrażeń wyprowadzonych zgodnie z tą produkcją. Struktura i kolejność punktów w dodatku C odpowiada odpowiednim fragmentom rozdziału 6, które dotyczą wyrażeń w specyfikacji.

Środowiskami, w których będziemy wyliczać wartości wyrażeń, są wartościowania zmiennych. Fakty wyprowadzane za pomocą tych reguł mają więc postać:

$$\mathcal{W} \vdash e \Longrightarrow v$$

Taki zapis oznacza, że przy wartościowaniu $\mathcal{W}$ wyrażenie e ma wartość v. W pewnych regułach v będzie symbolem **OK**; dotyczy to tych reguł, które wyznaczają jedynie dynamiczną poprawność wyrażenia. Analizując reguły z dodatku C należy pamiętać, że definiujemy semantykę wyrażeń, które są częścią specyfikacji poprawnych statycznie.

W tym punkcie przestawimy przykład wyjaśniający konstrukcję i znaczenie tych reguł. Oto reguła wyliczania wartości sklejania tropów:

$$\boxed{\mathit{traceExpr} ::= \mathit{traceExpr}_{left}\ .\ \mathit{traceExpr}_{right}}$$

$$\frac{\mathcal{W} \vdash \mathit{traceExpr}_{left} \Longrightarrow v_{left} \quad \mathcal{W} \vdash \mathit{traceExpr}_{right} \Longrightarrow v_{right}}{\mathcal{W} \vdash \mathit{traceExpr} \Longrightarrow v_{left} \bullet v_{right}}$$

Wyrażenie tropowe (*traceExpr*) skonstruowane zgodnie z umieszczoną w ramce produkcją składa się z dwóch podwyrażeń oznaczonych tutaj przez $\mathit{traceExpr}_{left}$ i $\mathit{traceExpr}_{right}$. Reguła ta stanowi, że jeśli przy wartościowaniu $\mathcal{W}$, wyrażenia $\mathit{traceExpr}_{left}$ i $\mathit{traceExpr}_{right}$ mają odpowiednio wartości v_{left} i v_{right}, to wyrażenie "$\mathit{traceExpr}_{left}\ .\ \mathit{traceExpr}_{right}$" (we wniosku reguły oznaczone przez *traceExpr*) przy wartościowaniu $\mathcal{W}$ ma wartość $v_{left} \bullet v_{right}$. Przypomnijmy, (por. p. 7.2), że "$\bullet$" jest operacją sklejania list.

7.4.8 Definicje funkcji

W specyfikacji tropowej definiujemy pewien zestaw funkcji. Definicja pojedynczej funkcji może składać się z wielu *równań*. Oto reguły składni opisujące postać tych równań:

domainRestriction	::=	`legal (` *IDENT* `(` $traceExpr_1$, ..., $traceExpr_n$ `) )` `<=>` *logExpr*
valueDefinition	::=	*IDENT* `(` $traceExpr_1$, ..., $traceExpr_n$ `)` `=` $traceExpr_{result}$
valueDefinition	::=	*IDENT* `(` $traceExpr_1$, ..., $traceExpr_n$ `)` `<=>` *logExpr*
mutatorLegality	::=	`legal (` *traceExpr* `;` *application* `)` `<=>` *logExpr*
observerLegality	::=	`legal (` *application* `)` `<=>` *logExpr*
mutatorSignals	::=	`signals (` *traceExpr* `;` *application* `)` `=` *signalExpr*
observerSignals	::=	`signals (` *application* `)` `=` *signalExpr*
mutatorFeasibility	::=	`feasible (` *traceExpr* `;` *application* `)` `<=>` *logExpr*
mutatorTransition	::=	`transition (` $traceExpr_{arg}$ `;` *application* `)` `=` $traceExpr_{value}$
observerResult	::=	`result (` *application* `)` `=` *traceExpr*

Na pierwszym miejscu w każdej z tych definicji znajduje się nazwa funkcji: `legal(`*IDENT*, *IDENT*, `legal`, `signals`, `feasible`, `transition` lub `result`. W nawiasach (w przypadku *domainRestriction* w wewnętrznych) znajdują się *parametry formalne* tej definicji. Po symbolu "`=`" lub "`<=>`" występuje *wyrażenie definiujące wartość funkcji*. W definicjach tych korzystamy z dopasowywania wzorca, które zostało nieformalnie opisane w punkcie 6.5. Tu zdefiniujemy ten mechanizm formalnie.

Przyjmijmy, że $\mathcal{P}$ jest wartościowaniem, które przyporządkowuje formalnym parametrom wartościom ich aktualne odpowiedniki. Załóżmy, że $\langle v_1, \ldots, v_n \rangle$ jest krotką należącą do dziedziny funkcji f. Wyrażenia $e_1, \ldots, e_n$ są parametrami formalnymi pochodzącymi z pierwszego w specyfikacji równania definiującego f, takiego, że istnieje wartościowanie $\mathcal{F}$, o następujących własnościach:

1. $\mathcal{P} \preceq \mathcal{F}$, oraz
2. dla każdego $i = 1, \ldots, n$ można wyprowadzić $\mathcal{F} \vdash e_i \Longrightarrow v_i$.

Oznaczmy przez $\mathcal{E}$ najmniejsze wartościowanie, spośród wszystkich $\mathcal{F}$. Przyjmijmy, że e_{result} jest wyrażeniem definiującym wartość funkcji, które występuje w tym samym równaniu co $e_1, \ldots, e_n$. Wówczas wartością f dla argumentów $\langle v_1, \ldots, v_n \rangle$ jest r takie, że

$$\mathcal{E} \vdash e_{result} \Longrightarrow r$$

Sytuacja, w której nie istnieje któryś z elementów występujących w powyższej definicji (równanie, którego parametry formalne dadzą się uzgodnić z $\langle v_1, \ldots, v_n \rangle$, lub najmniejsze wartościowanie $\mathcal{E}$, lub wartość r) oznacza błąd w specyfikacji. Jedyny wyjątek od tej reguły zostanie opisany w punkcie 7.4.8.4.

W kolejnych punktach określimy związek między funkcjami definiowanymi w specyfikacji, a denotacją tej specyfikacji (por. p. 7.4.2). Tę denotację będziemy oznaczać przez $\mathcal{S}$.

7.4.8.0 Funkcje pomocnicze

Sygnatura funkcji pomocniczej ma postać:

$$signature ::= IDENT_{name} \texttt{ : } IDENT_1 \ \ldots \ IDENT_n \texttt{ -> } IDENT_{n+1}$$

Dla każdego $i = 1, \ldots, n+1$ oznaczmy przez X_i zbiór wartości definiowanych przez $IDENT_i$:

$$X_i = \begin{cases} \textbf{Traces} & \text{gdy } IDENT_i = \text{"traces"} \\ \mathcal{S}_{IDENT_i}\textbf{.states} & \text{gdy } IDENT_i \neq \text{"traces"} \end{cases}$$

Wyrażenie $\mathcal{S}_{\Phi.\mathbf{specified}}.\mathbf{states}$ oznacza zbiór tropów kanonicznych. Z tego powodu typ specyfikowany może pojawiać się jedynie w sygnaturach funkcji pomocniczych, których definicja występuje za definicją predykatu "*canonical*" (por. p. 6.5 i B.4).

Zbiór $X_1 \times \ldots \times X_n$ jest dziedziną funkcji `legal(`$IDENT_{name}$. Jeśli w specyfikacji nie występują równania definiujące `legal(`$IDENT_{name}$, to przyjmujemy, że jest ona stale równa **true**. Do dziedziny funkcji $IDENT_{name}$ należą te i tylko te krotki $\langle v_1, \ldots, v_n \rangle$ dla których wartością `legal(`$IDENT_{name}$ jest **true**.

Przeciwdziedziną funkcji $IDENT_{name}$ jest zbiór X_{n+1}.

7.4.8.1 states

$\mathcal{S}$.**states** jest zbiorem tropów kanonicznych, tzn. tych tropów, dla których wartością funkcji "*canonical*" jest **true**. Ta funkcja musi być zdefiniowana w specyfikacji (por. p. 6.5 i B.4). Formalnie:

$$\mathcal{S}.\mathbf{states} = \{x \in \mathbf{Traces} \mid \mathcal{P}[\langle T, [\,] \rangle := x] \vdash \mathit{canonical}(T) \Longrightarrow \mathbf{true}\}$$

7.4.8.2 initial

Jeśli w rozważanej specyfikacji symbol nieterminalny *emptyEquivalence* ma rozwinięcie "ε", to stan początkowy obiektów specyfikowanego typu jest reprezentowany przez trop pusty:

$$\mathcal{S}.\mathbf{initial} = [\,]$$

Jeśli w rozważanej specyfikacji *emptyEquivalence* ma rozwinięcie "`_ =>` *traceExpr*", oraz

$$\mathcal{P} \vdash \mathit{traceExpr} \Longrightarrow i$$

to stan początkowy obiektów specyfikowanego typu jest reprezentowany przez i:

$$\mathcal{S}.\mathbf{initial} = i$$

Sytuacja, gdy takie i nie istnieje oznacza błąd w specyfikacji.

7.4.8.3 legal

Formalnie (por. p. 7.4.2.3):

$$\texttt{legal} \in [\ (\mathcal{S}.\mathbf{states} \times \mathbf{MutatorInvocs}) \cup \mathbf{ObserverInvocs} \to \{\mathbf{true}, \mathbf{false}\}\ ]$$

Funkcja `legal` definiuje zbiór $\mathcal{S}$.**legal**:

$$\mathcal{S}.\mathbf{legal} = \{x \in (\mathcal{S}.\mathbf{states} \times \mathbf{MutatorInvocs}) \cup \mathbf{ObserverInvocs} \mid \texttt{legal}(x) = \mathbf{true}\}$$

7.4.8.4 signals

Formalnie (por. p. 7.4.2.4):

$$\texttt{signals} \in [\ (\mathcal{S}.\mathbf{states} \times \mathbf{MutatorInvocs}) \cup \mathbf{ObserverInvocs} \to \mathbf{Signals}\ ]$$

Jednak jeśli dla pewnego elementu dziedziny wartością funkcji `signals` jest pusta lista sygnałów, specyfikacja nie musi zawierać równania, które to określa. To jest właśnie wyjątek, o którym wspomniano w punkcie 7.4.8. Polega on na tym, że wartością funkcji `signals` jest

lista pusta, dla każdego elementu jej dziedziny, dla którego nie istnieje równanie, którego parametry formalne dadzą się uzgodnić z tym elementem. Funkcja `signals` definiuje funkcję $\mathcal{S}$.**signals**:

$$\mathcal{S}.\textbf{signals} = \texttt{signals}$$

7.4.8.5 feasible

Funkcja `feasible` określa sensowność zdarzeń spowodowanych legalnymi wywołaniami modyfikatorów przekazujących niedeterministyczny wynik. Formalnie (por. p. 7.4.2.5):

$$\texttt{feasible} \in [\ (\textbf{LegalEvents} - \mathcal{S}.\textbf{legal}) \to \{\textbf{true}, \textbf{false}\}\ ]$$

Funkcja `feasible` definiuje zbiór $\mathcal{S}$.**feasible**[3]:

$$\begin{aligned}\mathcal{S}.\textbf{feasible} \quad = \quad & (\textbf{LegalEvents} \cap \mathcal{S}.\textbf{legal}) \\ & \cup \{x \in (\textbf{LegalEvents} - \mathcal{S}.\textbf{legal}) \mid \texttt{feasible}(x) = \textbf{true}\}\end{aligned}$$

7.4.8.6 transition

Formalnie (por. p. 7.4.2.6):

$$\texttt{transition} \in [\ \mathcal{S}.\textbf{feasible} \to \mathcal{S}.\textbf{states}\ ]$$

Funkcja `transition` definiuje funkcję $\mathcal{S}$.**transition**:

$$\mathcal{S}.\textbf{transition} = \texttt{transition}$$

7.4.8.7 result

Formalnie (por. p. 7.4.2.7):

$$\texttt{result} \in [\ \textbf{ObserverInvocs} \to \bigcup_{\Phi.\textbf{programs}(id) \neq \textbf{error}} \mathcal{S}_{id}.\textbf{states}\ ]$$

Funkcja `result` definiuje funkcję $\mathcal{S}$.**result**:

$$\mathcal{S}.\textbf{result} = \texttt{result}$$

7.4.9 Maszyna stanowa

Denotację specyfikacji tropowej można zinterpretować jako maszynę stanową podobną do maszyny Mealego [Mea55, HU94]. Maszyna Mealego jest uporządkowaną szóstką:

$$\mathcal{M} = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$$

- Σ jest skończonym zbiorem zwanym *alfabetem wejściowym*;
- Δ jest skończonym zbiorem zwanym *alfabetem wyjściowym*;
- Q jest skończonym zbiorem *stanów*;

[3] Oprócz zdarzeń, którymi zajmuje się funkcja `feasible`, sensowne są również wszystkie zdarzenia spowodowane wywołaniami modyfikatorów nie przekazujących niedeterministycznego wyniku.

- $q_0 \in Q$ jest *stanem początkowym*;
- $\delta \subseteq Q \times \Sigma \times Q$ jest *relacją przejścia*;
- $\lambda \subseteq \delta \times \Delta$ jest *relacją wyjścia*.

Maszyna, która jest interpretacją denotacji specyfikacji tropowej, różni się od maszyny Mealego jedynie tym, że zbiory Σ, Δ, Q mogą być nieskończone. Zakładamy, że $\mathbf{sink} \notin \mathcal{S}.\mathbf{states}$, oraz $\mathbf{nil} \notin \bigcup_{\Phi.\mathbf{programs}(id) \neq \mathbf{error}} \mathcal{S}_{id}.\mathbf{states}$. Oto elementy tej maszyny:

$$\begin{aligned}
\Sigma &= \mathbf{MutatorInvocs} \cup \mathbf{ObserverInvocs} \\
\Delta &= \mathbf{Signals}^* \times (\{\mathbf{nil}\} \cup \bigcup_{\Phi.\mathbf{programs}(id) \neq \mathbf{error}} \mathcal{S}_{id}.\mathbf{states}) \\
Q &= \mathcal{S}.\mathbf{states} \cup \{\mathbf{sink}\} \\
q_0 &= \mathcal{S}.\mathbf{initial}
\end{aligned}$$

$\langle b, i, e\rangle \in \delta$ jedynie, jeśli $b \neq \mathbf{sink}$, oraz prawdziwy jest co najmniej jeden z poniższych warunków:

1. $i \in \mathbf{ObserverInvocs}$, $i \in \mathcal{S}.\mathbf{legal}$ oraz $b = e$, lub
2. $i \in \mathbf{ObserverInvocs}$, $i \notin \mathcal{S}.\mathbf{legal}$ oraz $e = \mathbf{sink}$, lub
3. $i \in \mathbf{MutatorInvocs}$, $\langle b, i\rangle \in \mathcal{S}.\mathbf{feasible}$ oraz $\mathcal{S}.\mathbf{transition}(\langle b, i\rangle) = e$, lub
4. $i \in \mathbf{MutatorInvocs}$, $\langle b, i\hat{}r\rangle \in \mathcal{S}.\mathbf{feasible}$ oraz $\mathcal{S}.\mathbf{transition}(\langle b, i\hat{}r\rangle) = e$ dla pewnego r, lub
5. $i \in \mathbf{MutatorInvocs}$, $\langle b, i\rangle \notin \mathcal{S}.\mathbf{legal}$, oraz $e = \mathbf{sink}$.

$\langle\langle b, i, e\rangle, o\rangle \in \lambda$ jedynie, jeśli $b \neq \mathbf{sink}$, $o = \langle \mathcal{S}.\mathbf{signals}(i), r\rangle$, oraz prawdziwy jest co najmniej jeden z poniższych warunków:

1. $i \in \mathbf{ObserverInvocs}$, $i \in \mathcal{S}.\mathbf{legal}$ oraz $r = \mathcal{S}.\mathbf{result}(i)$, lub
2. $i \in \mathbf{ObserverInvocs}$, $i \notin \mathcal{S}.\mathbf{legal}$ oraz $r = \mathbf{nil}$, lub
3. $i \in \mathbf{MutatorInvocs}$, $\langle b, i\rangle \in \mathcal{S}.\mathbf{feasible}$ oraz $r = \mathbf{nil}$, lub
4. $i \in \mathbf{MutatorInvocs}$, $\langle b, i\hat{}r\rangle \in \mathcal{S}.\mathbf{feasible}$, $\mathcal{S}.\mathbf{transition}(\langle b, i\hat{}r\rangle) = e$, lub
5. $i \in \mathbf{MutatorInvocs}$, $\langle b, i\rangle \notin \mathcal{S}.\mathbf{legal}$, oraz $r = \mathbf{nil}$.

7.5 Uzasadnienie spójności definicji semantyki

Po zdefiniowaniu semantyki musimy zastanowić się czy ta semantyka jest spójna — czy jednoznacznie definiuje ona znaczenie każdej konstrukcji syntaktycznej? Postaramy się uzasadnić, że tak w istocie jest. Dla każdego poprawnego wyrażenia (z wyjątkiem *application*) istnieją dokładnie dwa fakty jakie możemy dla niego wyprowadzić — jeden mówiący o własnościach statycznych, drugi zaś o dynamicznych. Zatem statyczne i dynamiczne znaczenie każdego poprawnego wyrażenia (poza *application*) jest jednoznacznie określone.

Wspomniany wyjątek (*application*) również nie wprowadza niejednoznaczności semantyki. W punkcie B.7 na stronie 101 występują dwie reguły, które mają po trzy wnioski. Zatem dla pewnych *application* można wyprowadzić aż trzy różne fakty. Jednakże każdy z tych faktów określa inne własności *application*.

- Własność **m complete** jest przypisana do *application* jedynie, jeśli jest ono pełnym opisem zdarzenia powodowanego wywołaniem modyfikatora, tzn. że w przypadku, gdy ten modyfikator przekazuje niedeterministyczny wynik, wynik ten jest określony w rozważanym *application*. Ta własność jest wykorzystywana jedynie we wnioskowaniu o poprawności frazy *mutatorTransition*, co więcej ta fraza "konsumuje" jedynie fakty, które stwierdzają taką własność. W tym przypadku nie może zatem wystąpić niejednoznaczność.

- Własność **m without result** jest przypisana do *application* jedynie, jeśli jest ono opisem wywołania modyfikatora, tzn. że w przypadku, gdy ten modyfikator przekazuje niedeterministyczny wynik, wynik ten nie jest określony w rozważanym *application*. Ta własność jest wykorzystywana jedynie we wnioskowaniu o poprawności fraz *mutatorLegality* i *mutatorSignals*, co więcej te frazy "konsumują" jedynie fakty, które stwierdzają taką własność. W tym przypadku nie może zatem wystąpić niejednoznaczność.

- Własność **m with result** jest przypisana do *application* jedynie, jeśli jest ono opisem zdarzenia powodowanego wywołaniem modyfikatora, który przekazuje niedeterministyczny wynik, oraz ten wynik jest określony w rozważanym *application*. Ta własność jest wykorzystywana jedynie we wnioskowaniu o poprawności frazy *mutatorFeasibility*, co więcej ta fraza "konsumuje" jedynie fakty, które stwierdzają taką własność. W tym przypadku nie może zatem wystąpić niejednoznaczność.

- Własność Φ.**specified** mają te i tylko te *application*, które mają własność **m complete**. Φ.**specified** określa typ *application*. Ta własność jest wykorzystywana jedynie we wnioskowaniu o typie i poprawnośći wyrażeń tropowych (*traceExpr*), co więcej to wnioskowanie "konsumuje" jedynie fakty, które stwierdzają taką własność. W tym przypadku nie może zatem wystąpić niejednoznaczność.

Po wykluczeniu potencjalnych niejednoznaczności w regułach dla *application* możemy stwierdzić, że semantyka TAM 2000 przedstawiona w niniejszym rozdziale jednoznacznie określa denotacje specyfikacji.

Rozdział 8

Analiza metody TAM 2000

W niniejszym rozdziale postaramy się uzasadnić, że TAM 2000 jest metodą, która może posłużyć do wykazania tezy niniejszej rozprawy. Teza ta została postawiona w rozdziale 1 i stanowi, że istnieje metoda specyfikacji, która:

1. ma ściśle zdefiniowaną składnię i semantykę,
2. zachowuje podstawowe założenia i siłę wyrazu oryginalnej metody tropów, oraz
3. jest mniej skomplikowana niż TAM'97; w szczególności dotyczy to złożoności syntaktycznej.

TAM 2000 spełnia pierwszy warunek — formalną definicję metody można znaleźć w dodatkach A, B, C. Spełnienie drugiego z tych warunków zostanie rozważone w punkcie 8.1, zaś trzeciego — w punktach 8.2–8.6. Wspomnianą w tezie rozprawy propozycję rozwiązania problemu zmiennych wejściowych i wyjściowych zanalizujemy w punkcie 8.7. W punkcie 8.8 spisano refleksje zebrane w trakcie opracowywania bardziej złożonych specyfikacji, które definiują pewne moduły systemu Unix.

W niniejszym rozdziale będziemy korzystać z przykładów przedstawionych na początku rozdziału 4, oraz z dwóch innych specyfikacji, które opiszemy poniżej.

Pierwsza z nich definiuje typ index — obiekty tego typu są tablicami liczb całkowitych o zmiennej, ale ograniczonej długości (por. dodatek E). Przykład ten pochodzi z raportu [IMPK93]. Program INIT inicjuje obiekt i musi być wywołany zanim zostaną użyte inne programy. Działanie DESTROY jest odwrotne niż działanie INIT. PUT_INT i GET_INT pozwalają umieścić i odczytać liczbę przechowywaną w tablicy. DELETE usuwa element. Wywołanie CATENATE skleja zawartości dwóch tablic, zaś COPY kopiuje tablice. NB_ELEMS przekazuje bieżącą długość tablicy. IS_EQUAL stwierdza, czy dwie tablice mają identyczną zawartość.

Druga z nich definiuje typ receiver — odbiorcę pakietów działającego zgodnie z protokołem przesuwających się okien (por. dodatek G). Przykład ten pochodzi z pracy [BIMO94]. Programy N−1_IND_SAP[1] i N−1_ACCEPTED[1] są wywoływane przez niższą warstwę protokołu. N−1_IND_SAP transmituje ramkę. N−1_IND_SAP informuje o gotowości do przyjęcia potwierdzenia. Sygnał N−1_RES_SAP[1] potwierdzeniem otrzymania ramki wysy-

[1]Nazwy tych programów i sygnałów nie są identyfikatorami wg definicji z punktu 6.1, ponieważ zawierają znak minus. W niniejszej rozprawie posługujemy się ich oryginalną pisownią, aby zachować zgodność z [BIMO94].

łanym do warstwy niższej. Sygnał N_IND_SAP przekazuje otrzymany komunikat do warstwy wyższej protokołu.

8.1 Wierność podstawom i siła wyrazu

Podstawowe założenia metody TAM 2000 zostały opisane w rozdziale 5. Nie odbiegają one od założeń metody tropów w oryginalnym ujęciu — wszystkie pojęcia zostały zdefiniowane tak samo (obiekt, trop, sensowność tropów, równoważność obserwacyjna, trop kanoniczny, funkcja rozszerzenia, relacja wyjścia). Zmiany dotyczą mniej istotnych cech metody — zmienne wejściowe i zmienne wyjściowe zostały zastąpione przez sygnały (por. p. 4.5, 8.7).

Siła wyrazu TAM 2000 jest inna, ale ani większa, ani mniejsza niż siła wyrazu TAM'97. Decydują o tym dwie sprawy. Po pierwsze, w TAM 2000 nie można uzależnić działania programu dostępu od występowania aliasów wsród argumentów wyjściowych wywołania tego programu. W TAM'97 jest to wyrażalne i w znacznym stopniu odpowiada za nadmierną komplikację tej metody. Ponieważ potrzeba definiowania takich programów występuje dość rzadko, w TAM 2000 zrezygnowaliśmy z takiego udogodnienia. Po drugie, w TAM 2000 można definiować programy dostępu, które przekazują wiele niedeterministycznych wzajemnie zależnych wyników (por. p. 4.1). Takie programy nie mogą być zdefiniowane w TAM'97.

8.2 Zmniejszenie złożoności gramatyki

O uproszczeniu metody może świadczyć zmniejszennie złożoności jej gramatyki. Oczywiście nie można stosować tego kryterium bezkrytycznie — zwykle da się zapisać gramatykę z użyciem mniejszej liczby reguł i symboli, niestety odbywa się to kosztem jej czytelności. Porównując TAM'97 i TAM 2000 możemy z tego kryterium skorzystać, ponieważ gramatyka TAM 2000 została opracowana w takim samym stylu jak gramatyka TAM'97. W poniższej tabeli znajdują się informacje o złożoności tych gramatyk:

Metoda	Liczba symboli nieterm.	Liczba reguł gramatyki
TAM'97	84	172
TAM 2000	59	132

Z tego zestawienia wynika, że udało się wyeliminować niemal 30% symboli nieterminalnych i 23% reguł. To oznacza istotne uproszczenie zważywszy, że nie zostały zmienione fundamenty metody (por. p. 8.1).

Uwaga Gramatyka LALR(1) dla TAM 2000 przedstawiona w dodatku A zawiera 68 symboli nieterminalnych i 150 reguł. Tę gramatykę należy jednak porównywać z gramatyka LALR(1) opracowana na użytek edytora strukturalnego dla metody TAM'97 [IKM94, DIKR94], która ma 87 symboli nieterminalnych i aż 324 reguły.

8.3 Zmniejszenie liczby pojęć

TAM 2000 jest metodą prostszą niż TAM'97 również ze względu na mniejszą liczbę wprowadzanych pojęć. W TAM 2000 nie ma zmiennych wejściowych i zdarzeń z nimi związanych,

zmiennych wyjściowych, oraz żetonów statusu. Jedynym nowowprowadzonym pojęciem jest sygnał.

Zmienne wejściowe stały się niepotrzebne, ponieważ wyeliminowano zdarzenia z nimi związane. W roli tych zdarzeń z powodzeniem mogą występować wywołania programów dostępu.

Zmienne wyjściowe i żetony zastąpiono sygnałami (por. p. 8.7).

8.4 Nazwy obiektów

Specyfikacja napisana w TAM 2000 jest całkowicie niezależna od nazw obiektów (por. p. 4.1). To powoduje, że nie jesteśmy w stanie definiować programów dostępu, których działanie zależy od występowania aliasów wśród argumentów wywołań. Taka możliwość istnieje w TAM'97 i jest najbardziej istotną przyczyną wysokiej złożoności tej metody. Powoduje ona skomplikowaną definicję tropu, a w szczególności to, że tropy musimy rozważać z dokładnością do pewnej relacji równoważności. Dwa tropy są w tej relacji, jedynie jeśli jeden można osiągnąć z drugiego poprzez różnowartościowe przemianowanie gwiazdek oznaczających nazwy obiektów. Obserwacyjna równoważność tropów jest definiowana dla klas abstrakcji relacji wskazanej w poprzednim zdaniu.

Występujące w TAM'97 symbole "$*$", "$\searrow$", "$*_1$", "$*_2$", itd., przyczyniały się do nadmiernej złożoności syntaktycznej tej metody. Dodatkowym utrudnieniem było to, że w takim znaczeniu te symbole nie występują nigdzie indziej w informatyce i matematyce. Ich nieobecność w TAM 2000 znacznie upraszcza tę metodę, nie zmniejszając istotnie jej siły wyrazu.

Uproszczenie metody tropowej można mierzyć także objętościa definicji tropu. Jest ona ponad czterokrotnie obszerniejsza w TAM'97 (por. p. 4.6 raportu [IKM$^+$97]) niż w TAM 2000 (por. p. 7.4.1).

8.5 Hermetyzacja

W TAM'97 wszystkie wartości są traktowane jako tropy. Powoduje to kłopoty, ponieważ w przypadku typu obcego jesteśmy raczej zainteresowani zbiorem jego stanów, a nie zbiorem jego tropów. W TAM'97 potrzebujemy więc symboli na oznaczenie tych dwóch zbiorów. Używamy w tym celu oznaczeń, odpowiednio, <typ> i <<typ>>. W TAM 2000 rozważamy tylko pierwszy z tych zbiorów i dlatego możemy używać nazwy typu bez żadnych dekoracji na oznaczenie tego zbioru. To zmniejsza liczbę niestandardowych konwencji notacyjnych i upraszcza metodę.

Kolejną konsekwencją korzystania jedynie ze zbioru tropów kanonicznych typów obcych jest eleminacja niejednoznaczności związanych z semantyką wywołań programów dostępu. W TAM'97 mogły być one interpretowane na dwa sposoby: jako budulec tropu albo jako odwołania do funkcji semantycznych związanych z danym programem dostępu. Symbol strzałki służył do rozróżnienia, który ze sposobów jest w danym miejscu obowiązujący. Załóżmy, że w rozważanej specyfikacji typ index jest typem obcym. Wówczas, wg TAM'97 wyrażenie:

$$\text{PUT\_INT}(*, 1, 3)$$

jest elementem tropu, zaś

$$\text{PUT\_INT}((*_1, \text{INIT}(*)) \searrow, 1, 3)$$

oblicza wartość przekazaną przez pierwszy argument. W TAM 2000 nie ma takiego rozróżnienia — każde wywołanie programu dostępu typu obcego jest odwołaniem do semantyki tego programu. Zatem:

$$\text{PUT\_INT}(T; 1, 3)$$

oznacza stan tablicy po wykonaniu na niej operacji PUT_INT$(1, 3)$, w chwili gdy była w stanie T. Natomiast:

$$\text{GET\_INT}(T, 1)$$

to wartość przekazana przez wywołanie GET_INT(1) dla tablicy będącej w stanie T. Dla porównania przedstawimy również, jak te same wyrażenia należałoby zapisać w TAM'97:

$$\text{PUT\_INT}((*_1, T) \searrow, 1, 3) \quad \text{GET\_INT}(T, 1) \searrow$$

W TAM'97 problem opisany w poprzenim akapicie był szczególnie kłopotliwy w przypadku typów wbudowanych. W szczególności, występowały problemy z interpretacją wyrażeń arytmetycznych. Rozważmy wyrażenie:

$$x + y$$

Jeśli "+" jest programem dostępu, to $x+y$ jest elementem tropu i jego wartością jest jednowywołaniowy trop $+(x, y)$, a nie wynik tego działania. Z tego powodu w TAM'97 zdefiniowano "+" jako funkcję pomocniczą. Takie rozwiązanie jest bardzo nienaturalne. W TAM 2000 ta nienaturalność nie występuje, ponieważ "+" jest programem dostępu i poza specyfikacją typu int wywołanie $x + y$ oznacza liczbę powstałą przez dodanie y do x. Oczywiście wewnątrz specyfikacji typu int $x + y$ nadal oznacza jednowywołaniowy trop $+(x, y)$.

8.6 Dopasowanie wzorca

Dopasowywanie wzorca jest bardzo przydatnym udogodnieniem przy tworzeniu specyfikacji tropowych. Ten mechanizm był bardzo często wykorzystywany w specyfikacjach, zwykle występował w wyrażeniu "*where*". Na przykład w specyfikacji typu index zapisanej za pomocą TAM'97 występowałaby następująca definicja:

$$\text{PUT\_INT}((m, T) \searrow, p, n) =$$

Condition	Value
$length(T) = p$	$T.\text{PUT\_INT}(p, n)$
$length(T) > p$	$T_1.\text{PUT\_INT}(p, n).T_2$ *where* T_1, T_2 : <<index>>; x : <int> [$T = T_1.\text{PUT\_INT}(p, x).T_2$]

Dzięki mechanizmowi dopasowywania wzorca w TAM 2000 możemy tę definicję zapisać w postaci dwóch znacznie prostszych równań (por. dodatek E):

$$transition(T_1.\text{PUT\_INT}(p, x).T_2; \text{PUT\_INT}(p, n)) = T_1.\text{PUT\_INT}(p, n).T_2$$

$$transition(T; \text{PUT\_INT}(p, n)) = T.\text{PUT\_INT}(p, n)$$

Dzięki podziałowi na dwa równania, łatwiej jest odnaleźć konkretną informację. Czytelność specyfikacji poprawia też wyeliminowanie dość rozwlekłej konstrukcji "*where*".

Dopasowywanie wzorca upraszcza również semantykę metody — funkcja rozszerzenia i relacja wyjścia są traktowane jako zwykłe funkcje, które definiuje się za pomocą wielu równań (dla różnych wzorców). W TAM'97 trzeba było rozważać pod-funkcje rozszerzenia dla każdego programu dostępu i z nich dopiero budowano funkcję rozszerzenia. Warto tu także zwrócić uwagę, że w TAM 2000 wszystkie funkcje są definiowane w ten sam sposób. W TAM'97 tak nie było — sposób zapisu i znaczenie funkcji było inne w każdej sekcji specyfikacji.

Dopasowywanie wzorca szczególnie poprawia czytelność definicji tropów kanonicznych. Wzorce w kolejnych równaniach określają dopuszczalną postać tropów kanonicznych. W TAM'97 trop kanoniczny jest definiowany jedną formułą logiczną, z której trzeba wywnioskować to, jak wygląda trop kanoniczny. Na przykład, dla typu index:

$$\mathit{canonical}(T) \quad \Leftrightarrow \quad (T = \_) \vee \exists s : \text{<int>}; a_{1\ldots s} : \text{<int>} \\ [\ T = \text{INIT}().[\text{PUT\_INT}(i, x_i)]_{i=1}^{s} \wedge s \leq \text{maxSize}\ ]$$

W TAM 2000 zdefiniujemy zbiór tropów kanonicznych w sposób nieuwikłany, wprost podając ich format (por. dodatek E):

$\mathit{canonical}(\_) \Leftrightarrow \textbf{true}$

$\mathit{canonical}(\text{INIT}().[\text{PUT\_INT}(i, x_i)]_{i=1}^{s}) \Leftrightarrow s \leq \text{maxSize}$

$\mathit{canonical}(\mathit{other}) \Leftrightarrow \textbf{false}$

Tropy kanoniczne są bardzo ważne w specyfikacji; z tego powodu za niezwykle istotną należy uznać możliwość szybkiego odnalezienia ich formatu.

W dodatku F przedstawiliśmy rekurencyjną (a zatem uwikłaną) definicję tropów kanonicznych, aby zilustrować możliwość użycia indukcji w takich definicjach. Można tę definicję zapisać nie korzystając z rekurencji:

$\mathit{canonical}(\_) \Leftrightarrow \textbf{true}$

$\mathit{canonical}([\text{PICK}()\uparrow a_i]_{i=1}^{s}) \Leftrightarrow \text{first} \leq a_1 \wedge \forall\, i : \text{int}\ (0 < i \wedge i < s)[\ a_i < a_{i+1}\ ] \wedge a_s \leq \text{last}$

$\mathit{canonical}(\mathit{other}) \Leftrightarrow \textbf{false}$

8.7 Sygnały

W TAM'97 występują dwa pojęcia, których semantyki nigdy nie udało się zdefiniować w satysfakcjonujący sposób. Są to zmienne wyjściowe i żetony statusu. W TAM 2000 zastąpiono je sygnałami. Postaramy się wykazać, że problemy związane z owymi dwoma zastąpionymi pojęciami nie występują w przypadku sygnałów. Wykorzystamy w tym celu przykładową specyfikację typu receiver. Oryginał tej specyfikacji znajduje się w pracy [BIMO94], która ukazała się wcześniej niż raport TAM'97 [IKM+97]. Jednak zmienne wyjściowe w [BIMO94] mają takie same właściwości jak w TAM'97.

Pierwsza kłopotliwa sytuacja ma miejsce, gdy w danej chwili za pomocą zmiennej wyjściowej nie chcemy przekazać żadnej informacji. W TAM'97 potrzebna jest specjalna wartość, która wskazuje „brak komunikatu". W oryginalnej specyfikacji typu receiver, gdy nie chcemy wysłać żadnej ramki przez zmienną wyjściową N_IND_SAP, nadajemy jej wartość równą

tropowi pustemu tego samego typu co typ tej zmiennej. Po zastąpieniu zmiennych wyjściowych sygnałami, ten problem znika — po prostu nie wysyłamy sygnału N_IND_SAP, gdy nie mamy do przekazania żadnej nowej informacji dla warstwy wyższej protokołu.

Druga kłopotliwa sytuacja jest podobna do pierwszej — ma miejsce, gdy za pomocą zmiennej wyjściowej chcemy przekazać dwukrotnie tę samą informację. W TAM'97 zdarzenia związane ze zmiennymi wejściowymi następują jedynie wtedy, gdy zmienia się wartość zmiennej. Dwukrotne nadanie zmiennej wyjściowej tej samej wartości nie spowoduje zatem zdarzenia związanego ze sprzężoną zmienną wejściową. W takiej sytuacji musimy dodać do interejsu dodatkową zmienną wyjściową typu logicznego, która zmieniając swoją wartość na przeciwną poinformuje sprzężony obiekt o zajściu zdarzenia. W oryginalnej specyfikacji typu receiver, nie dostrzeżono tego problemu. Nie ma możliwości wysłania dwukrotnie pod rząd tej samej ramki przez zmienną wyjściową N_IND_SAP. Po zastąpieniu zmiennych wyjściowych sygnałami, ten problem znika — po prostu wysyłamy sygnał N_IND_SAP wielokrotnie pod rząd. Jest to interpretowane przez wyższą warstwę protokołu jako wielokrotne wystąpienie tej samej ramki.

Zmienne wyjściowe można sprzęgać ze zmiennymi wejściowymi — modyfikacja wartości wyjściowej powoduje zmianę wartości wejściowej. Sygnały również mają naturalnego partnera sprzężenia — modyfikatory. Sygnał pewnego obiektu może zostać podłączony do modyfikatora innego obiektu; wówczas wysłanie tego sygnału powoduje wywołanie tego modyfikatora. Argumetami tego wywołania są argumenty wysłanego sygnału. Po zastąpieniu żetonów statusu sygnałami, otrzymujemy intuicyjną semantykę osługi błędów, do której były powołane żetony statusu. Wystąpienie błędu powoduje wysłanie sygnału, co z kolei doprowadza do wywołania pewnego modyfikatora. Ten modyfikator może należeć do obiektu odpowiedzialnego za wykrywanie awarii.

8.8 Przykłady złożonych specyfikacji

W dodatkach H i I znajdują się specyfikacje dwóch modułów systemu Unix; są to tablica plików i tablica deskryptorów plików. Bardzo dobry opis działania tych modułów można znaleźć w książce [Bac95]. Specyfikacje we wspomnianych dodatkach są wzorowane na analogicznych specyfikacjach opublikowanych w pracy magisterskiej [KM96].

Opracowanie tych specyfikacji w TAM 2000 zajęło autorowi niniejszej rozprawy niecały dzień pracy. Uważamy, że nie jest to wiele, zwłaszcza, że nie posługiwano się żadnym narzędziem wspomagającym pracę nad specyfikacjami, a także znaleziono kilka błędów w oryginalnych specyfikacjach z [KM96].

Otrzymano specyfikacje o mniejszej objętości, niż ich odpowiedniki w [KM96]. Naszym zdaniem, specyfikacje zapisane w TAM 2000 są bardziej przejrzyste i czytelne. Zauważyliśmy, że szczególnie cennym mechanizmem jest dopasowywanie wzorca — znacznie ułatwia ono manipulację skomplikowanymi tropami. Nie mniej istotne było wyeliminowanie gwiazdek, które niosąc niewiele informacji występowały w każdym wywołaniu programu dostępu. To znacznie utrudniało zrozumienie treści specyfikacji.

Podsumowując, stwierdzamy, że opracowanie specyfikacji tablicy plików i tablicy deskryptowów plików było łatwiejsze w TAM 2000 niż w poprzednich wersjach metody tropów.

Rozdział 9

Podsumowanie

Celem niniejszej rozprawy było wykazanie, że istnieje abstrakcyjna metoda specyfikacji, która zachowując intuicje metody tropów w jej oryginalnym ujęciu, ma mniejszą złożoność niż metoda TAM'97. Dodatkowym wymaganiem było istnienie ścisłej definicji takiej metody. Postawiony cel został osiągnięty — zaprezentowana w rozprawie metoda TAM 2000 spełnia wymagania wyrażone w tezie. Niestety nie udało się zachować wszystkich udogodnień występujących w TAM'97 — TAM 2000 nie pozwala na definiowanie programów, których działanie zależy od występowania aliasów wśród argumentów wywołania. Przyjmując to ograniczenie dokonaliśmy niełatwego wyboru pomiędzy zachowaniem siły wyrazu metody, a znacznym uproszczenie postaci tropu i zarazem całej metody. Uznaliśmy, że uproszczenie metody jest ważniejsze; jedną z przesłanek była obserwacja, że uzależnianie definicji programu od aliasów wśród jego argumentów jest przydatne w niewielkiej liczbie przypadków.

Istotnym celem pracy było również poszukiwanie satysfakcjonującej koncepcji zmiennych wejściowych i wyjściowych. Cel ten został osiągnięty — proponujemy zastąpić takie zmienne sygnałami. Otrzymane rozwiązanie jest spójne i zgodne z intuicją. Istotną cechą sygnałów jest też pełna symetria między nimi i programami dostępu. Uważamy, że interesującą propozycję dalszych badań stanowi kontynuacja prac nad specyfikowaniem protokołów komunikacyjnych. Wierzymy, że pojęcia zawarte w TAM 2000 (sygnały) nadają się do tego celu lepiej niż odpowiednie pojęcia z TAM'97 (zmienne wejściowe i wyjściowe).

Idee wykorzystane przy opracowywaniu metody TAM 2000 nie są oryginalnym wkładem autora niniejszej rozprawy. W literaturze pojawiają się one w rozmaitych kontekstach, niektóre także w pracach poświęconych metodzie tropów. Abstrahowanie od nazw poszczególnych obiektów i hermetyzacja zawartości specyfikacji dotyczą niemal wszystkich metod specyfikacji poza metodą tropów. W pracach nad metoda tropów pojawiały się już: wychodzące wywołanie [Hof85], które jest protoplastą sygnału w TAM 2000, dopasowywanie wzorca [WP92], a także definiowanie legalności wywołań bez użycia żetonów statusu [EKMD98]. Oryginalnymi osiągnięciami niniejszej rozprawy są natomiast wykorzystanie wszystkich tych idei przy opracowaniu spójnej pojęciowo metody, a także określenie jej semantyki. Sposób definicji semantyki pochodzi z raportów [Plo81, Kah88]. Warto tu również wspomnieć o tym, że w trakcie prac nad rozprawą doktorską [Eng99] zidentyfikowano podobne niedoskonałości w TAM'97 do tych, których usunięciu poświęcona jest niniejsza rozprawa.

Definicja nowej metody nie oznacza końca jej rozwoju. Doprowadzenie do jej praktycznych zastosowań wymaga m.in. zbudowania narzędzi ułatwiających stosowanie metody. Do takich narzędzi należy zaliczyć: edytor strukturalny, generator prototypów, a także system

wspomagający automatyczne dowodzenie własności specyfikacji. Opracowanie takiego oprogramowania dla metody TAM 2000 będzie ułatwione, ponieważ istnieją analogiczne narzędzia dla TAM'97: edytor FUN–SPEC [IKM94, DIKR94], generator prototypów specyfikacji tropowych [Zio99]. Wypracowano również techniki weryfikacji poprawności specyfikacji tropowych przy użyciu systemu PVS [EKMD98, Kre98]. Obecnie są prowadzone prace nad zgodnością semantyki konstrukcji zawartych w PVS z semantyką metody TAM'97 [Eng99]. Jeśli teza postawiona w rozprawie [Eng99] okaże się prawdziwa, system PVS będzie niezwykle przydatny dla osób stosujących metodę tropów.

Z pracami nad metodą tropów nieodłącznie są związane prace nad projektem implementacji oraz jego weryfikacją względem specyfikacji tropowej. Prace w tym kierunku były utrudnione za sprawą wysokiej komplikacji obowiązującej wersji metody tropów, którą była TAM'97. Wierzymy, że niniejsza rozprawa tchnie nowego ducha w prace nad rozwojem projektu implementacji dzięki uproszczeniu specyfikacji interfejsu. Jest to szczególnie istotne, ponieważ metoda tropów nie może być jedyną techniką tworzenia dokumentacji w czasie trwania projektu. Potrzebna jest metodyka, która połączy specyfikacje interfejsu z kodem w języku programowania (w jej skład wchodzi projekt implementacji). Istotnym elementem badań powinna być weryfikacja praktycznej użyteczności tej metodyki. Można to osiągnąć jedynie poprzez stosowanie jej przy tworzeniu różnorakiego oprogramowania.

Literatura

[Bac95] Maurice J. Bach. *Budowa systemu operacyjnego UNIX*. Wydawnictwo Naukowo-Techniczne, Warszawa, Drugie wydanie edition, 1995.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[BD87] S. Budkowski and P. Dembiński. An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3–23, 1987.

[Bea88] S. Bear. Structuring for the VDM specification language. In *VDM'88: VDM — The Way Ahead*, number 328, pages 2–25. Springer-Verlag, 1988.

[BG77] R. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058, Cambridge, Massachussets, August 1977.

[BG81] R. Burstall and J. A. Goguen. An informal introduction to specifications using CLEAR. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, pages 185–213. Academic Press, 1981.

[BH97] Ramesh Bharadwaj and Constance L. Heitmeyer. Model checking complete requirements specifications using abstraction. NRL Memorandum Report NRL/MR/5540–97–7999, United States Naval Research Laboratory, Washington, DC, USA, November 1997.

[BIMO94] Jakub Bojanowski, Michał Iglewski, Jan Madey, and Abdellatif Obaid. Functional approach to protocols specification. In *Proceedings of the 14th International IFIP Symposium on Protocol Specification, Testing and Verification, PSTV'94, Vancouver, B.C.*, pages 371–378, 1994.

[Bjø81] D. Bjørner. The VDM principles of software specifications and program design. In *Formalizations of Programming Concepts*, number 107, pages 44–74. Springer-Verlag, 1981.

[BP78] Wolfram Bartussek and David Lorge Parnas. Using traces to write abstract specifications for software modules. In *Proceedings of 2nd Conference of European Cooperation in Informatics*, number 65, pages 211–236. Springer-Verlag, 1978.

[BP81] K. H. Britton and David Lorge Parnas. A–7E software module guide. NRL Memorandum Report 4702, United States Naval Research Laboratory, Washington, DC, USA, 1981.

[BP83] D. Bjørner and S. Prehn. Software engineering aspect of VDM. *Theory and Practice of Software Technology*, pages 85–134, 1983.

[Bri86] E. Brinksma. A tutorial on LOTOS. In V. M. Diaz, editor, *Protocol Specification, Testing and Verification*, pages 171–194, North Holland, 1986.

[BST99] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. In *Proceedings of 7th Intenational Conference on Algebraic Methodology and Software Technology (AMAST'98)*, number 1548, pages 341–357. Springer-Verlag, 1999.

[CP93] François Courtois and David Lorge Parnas. Formally specifying a communications protocol using the trace assertion method. CRL Report No. 269, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1993.

[DIKR94] Bernard Desruisseaux, Michał Iglewski, Marcin Kubica, and Patric Roy. Working notes on the requirements specification for the trace assertion method. Technical Report RR 94/09–5, Département d'Informatique, Université du Québec à Hull, Québec, Canada, 1994.

[Doo99] Andrew C. P. Dookhan. Improvements to the trace assertion method for software engineering. CRL Report No. 372, McMaster University, CRL, Hamilton, Ontario, Canada, 1999.

[EKM+93] Marcin Engel, Marcin Kubica, Jan Madey, David Lorge Parnas, Anders P. Ravn, and A. John van Schouwen. A formal approach to computer systems requirements documentation. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, number 736, pages 452–474. Springer-Verlag, 1993.

[EKMD98] Marcin Engel, Artur Kret, and Janina Mincer-Daszkiewicz. Increasing automation capabilities of the pvs theorem prover. In Hans-Dieter Burkhard, Ludwik Czaja, and Peter Starke, editors, *Proceedings of the CS&P Workshop*, pages 64–75, Berlin, September 1998.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I.* Springer-Verlag, 1985.

[Eng99] Marcin Engel. *Metoda badania poprawności specyfikacji tropowych.* PhD thesis, Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki, w przygotowaniu.

[Ers92] Neil Stuart Erskine. The usefulness of the trace assertion method for specifying device module interfaces. CRL Report No. 258, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1992.

[FGJM85] K. Fatatsugi, J. A. Goguen, Jean-Pierre Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proceedings of 12th Symposium on Principles of Programming Languages*, pages 52–66, New Orleans, January 1985.

[Gaw93] Michał Gawryś. *Unix: Narzędzia programistyczne yacc i lex*. Wydawnictwo PLJ, Warszawa, 1993.

[GH83] J. V. Guttag and J. J. Horning. An introduction to the Larch Shared Language. In R. E. A. Mason, editor, *Proceedings of the IFIP 9th World Computer Congress*, pages 809–814, Paris, France, September 1983.

[GH86a] J. V. Guttag and J. J. Horning. A Larch Shared Language handbook. *Science of Computer Programming*, 6:135–157, 1986.

[GH86b] J. V. Guttag and J. J. Horning. Report on the Larch Shared Language. *Science of Computer Programming*, 6:103–134, 1986.

[GH92] John V. Guttag and James J. Horning. Introduction to LCL, a Larch/C Interface Language. SRC Report 74, Digital Systems Research Center, November 1992.

[GH93] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[GT79] J. A. Goguen and J. J. Tardo. An introduction to OBJ: A language for writing and testing formal algebraic program specifications. In *Proceedings of the Conference on specifications of Reliable Software*, pages 170–189, Cambridge, Massachussets, 1979.

[GTW78] J. A. Goguen, J. W. Thatcher, and E. Wagner. An initial algebra approach to the specification and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology IV*, pages 80–184. Prentice-Hall, 1978.

[GTWW75] J. A. Goguen, J. W. Thatcher, E. Wagner, and J. Wright. Abstract data types as initial algebra and the correctness of data representations. In *Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Structure*, pages 89–93, May 1975.

[Gut77] John V. Guttag. Abstract data type and the development of data structures. *Communications of the ACM*, 20(6):396–404, 1977.

[HJL96] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[HKL+98] Constance L. Heitmeyer, James Kirby Jr., Bruce G. Labaw, Myla Archer, and Ramesh Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.

[Hof85] Daniel Hoffman. The trace specification of communication protocols. *IEEE Transactions on Computers*, C–34(12):1102–1113, December 1985.

[HU94] John E. Hopcroft and Jeffrey D. Ullman. *Wprowadzenie do teorii automatów, języków i obliczeń*. Wydawnictwo Naukowe PWN, Warszawa, 1994.

[IKM94] Michał Iglewski, Marcin Kubica, and Jan Madey. Editor for the trace assertion method. In Marek Zaremba, editor, *Proceedings of the 10th ISPE/IFAC International Conference on CAD/CAM, Robotics and Factories of the Future, CARs & FOF'94*, pages 876–881. OCRI, Ottawa, Canada, 1994.

[IKM95a] Michał Iglewski, Marcin Kubica, and Jan Madey. Trace specifications of non-deterministic multi-object modules. In Kanchana Kanchanasut and Jean-Jacques Levy, editors, *Algorithms, Concurrency and Knowledge. Proceedings*, number 1023, pages 381–395. Springer-Verlag, 1995.

[IKM+95b] Michał Iglewski, Marcin Kubica, Jan Madey, Janina Mincer-Daszkiewicz, and Krzysztof Stencel. The Fun-Project: From requirements specification to program presentation. Raport TR 95–18 (218), Instytut Informatyki, Uniwersytet Warszawski, Warszawa, 1995.

[IKM+97] Michał Iglewski, Marcin Kubica, Jan Madey, Janina Mincer-Daszkiewicz, and Krzysztof Stencel. TAM'97: the trace assertion method of module interface specification. Reference manual. Raport TR 97–01 (238), Instytut Informatyki, Uniwersytet Warszawski, Warszawa, 1997.

[IMD97] Michał Iglewski and Janina Mincer-Daszkiewicz. Internal design of modules specified in the trace assertion method. *Science of Computer Programming*, 28(2–3):139–170, 1997.

[IMDS95] Michał Iglewski, Janina Mincer-Daszkiewicz, and Krzysztof Stencel. Case study in trace specification of non-deterministic modules. In *Proceedings of the CS&P Workshop*, pages 96–110, Warszawa, December 1995.

[IMPK93] Michał Iglewski, Jan Madey, David Lorge Parnas, and Philip C. Kelly. Documentation paradigms (A progress report). CRL Report No. 270, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1993.

[IMS94] Michał Iglewski, Jan Madey, and Krzysztof Stencel. On fundamentals of the trace assertion method. Raport TR 94–09 (198), Instytut Informatyki, Uniwersytet Warszawski, Warszawa, 1994.

[Jan95] Ryszard Janicki. On foundations of the trace assertion method. Technical Report 95–04, McMaster University, Department of Computer Science and Systems, Hamilton, Ontario, Canada, 1995.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1986.

[Jon91] K. Jones. LM3: A Larch Interface Language for Modula–3. SRC Report 72, Digital Systems Research Center, June 1991.

[Kah88] G. Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258, North-Holland, 1988.

[KM96] Piotr Kołodziejski and Małgorzata Majewska. Specyfikacja unixowego systemu plików w metodzie tropów. Master's thesis, Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki, 1996.

[Kre98] Artur Kret. Strategie dowodzenia specyfikacji tropowych. Master's thesis, Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki, 1998.

[Kub94] Marcin Kubica. Metoda tropów — analiza i propozycje modyfikacji. Master's thesis, Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki, 1994.

[Kub97] Marcin Kubica. TAM'97: the trace assertion method of module interface specification. Formal semantics. Maszynopis, Instytut Informatyki, Uniwersytet Warszawski, Warszawa, 1997.

[Kub99] Marcin Kubica. *Specyfikacje wskaźnikowych struktur danych.* PhD thesis, Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki, w przygotowaniu.

[Las95] Sławomir Lasota. Semantyka specyfikacji w metodzie tropów. Master's thesis, Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki, 1995.

[Lin86] R. J. Linn, Jr. The features and facilities of Estelle. In V. M. Diaz, editor, *Protocol Specification, Testing and Verification*, pages 271–296, North Holland, 1986.

[Maj77] M. E. Majster. Limits of the "algebraic" specifications of abstract data types. *ACM SIGPLAN Notices*, pages 37–43, October 1977.

[McL84] John McLean. A formal method for the abstract specification of software. *Journal of the ACM*, 31(3):600–627, 1984.

[MCP93] C. Myers, C. Clark, and E. Poon. *Programming with Standard ML.* Prentice-Hall, New Yersey, 1993.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, September 1955.

[Nor95] Theodore S. Norvell. On trace specifications. CRL Report No. 305, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1995.

[ORS95] Sam Owre, John Rushby, and Natarajan Shankar. Analyzing tabular and state-transition requirements specifications in PVS. Technical Report CSL–95–12, Compuetr Science Laboratory, SRI International, Menlo Park CA, USA, June 1995.

[OSR93] Sam Owre, Natarajan Shankar, and John Rushby. *User Guide for the PVS Specification and Verification System, Language, and Proof Checker (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[PAM91] David Lorge Parnas, G. L. K. Asmis, and Jan Madey. Assessement of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2):189–198, 1991.

[Par71] David Lorge Parnas. Information distribution aspects of design methodology. In *Proceedings of IFIP Congress*, pages 26–30, 1971.

[Par72a] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Par72b] David Lorge Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(12):330–336, May 1972.

[Par90] David Lorge Parnas. Documentation of communications services and protocols. Technical Report 90–272, Queen's University, C&IS, Telecommunication Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, 1990.

[Par92] David Lorge Parnas. Tabular representations of relations. CRL Report No. 260, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1992.

[PCW85] David Lorge Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, 11:259–266, 1985.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Aarhus University, Aarhus, Denmark, 1981.

[PM95] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, 1995.

[PW89] David Lorge Parnas and Yabo Wang. The trace assertion method of module interface specification. Technical Report 89–261, Queen's University, C&IS, Telecommunication Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, 1989.

[RR76] O. Roubine and L. Robinson. SPECIAL reference manual. Technical Report CSL–45, Computer Science Laboratory, Stanford Research Institute, August 1976.

[RS82] A. Rockstrom and R. Saracco. SDL — CCITT specification and description language. *IEEE Transactions on Communications*, 30(6):1310–1317, June 1982.

[She97] Hong Shen. Canonical representation. Maszynopis, McMaster University, CRL, Hamilton, Ontario, Canada, 1997.

[SM90a] A. Sampaio and S. Meira. Modular extensions to Z. In *VDM'90: VDM and Z — Formal Methods in Software Development*, number 428, pages 211–232. Springer-Verlag, 1990.

[SM90b] J. Steensgaard-Madsen. Adding abstract datatypes to Meta-IV. In *VDM'90: VDM and Z — Formal Methods in Software Development*, number 428, pages 233–243. Springer-Verlag, 1990.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, Second edition edition, 1992.

[ST87a] Donald Sannella and Andrzej Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34(2/3):150–178, April/June 1987.

[ST87b] R. Saracco and P. A. J. Tilanus. CCITT SDL: Overview of the language and its applications. *Computer Networks and ISDN Systems*, 13(2):65–74, 1987.

[ST97] Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.

[Ste95] Krzysztof Stencel. Wyrażenia w metodzie tropów. Master's thesis, Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki, 1995.

[SW83] Donald Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. In *Proceedings, International Conference on Foundations of Computation Theory*, number 158, pages 413–427, Borgholm, Sweden, 1983. Springer-Verlag.

[vSPM93] A. John van Schouwen, David Lorge Parnas, and Jan Madey. Documentation of requirements in computer systems. In *IEEE International Symposium on Requirements Engineering*, pages 198–207, San Diego, CA, 1993. IEEE Computer Society Press.

[Wan94a] Yabo Wang. Formal and abstract software module interface specifications — a survey. CRL Report No. 238, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1994.

[Wan94b] Yabo Wang. Specifying and simulating the externally observable behavior of modules. CRL Report No. 292, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1994.

[WG89] Wiliam M. Waite and Gerhard Goos. *Konstrukcja kompilatorów*. Biblioteka Inżynierii Oprogramowania. Wydawnictwo Naukowo-Techniczne, Warszawa, 1989.

[Win87] J. M. Wing. Writing Larch Interface Language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[Wor92] J. B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.

[WP92] Yabo Wang and David Lorge Parnas. Trace rewriting systems. CRL Report No. 247, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1992.

[Zio99] Szymon Zioło. Generator prototypów specyfikacji tropowych. Master's thesis, Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki, 1999.

Dodatek A

Składnia specyfikacji tropowych

A.1 Plik wejściowy dla programy `lex`

```
digit   [0-9]
letter  [A-Za-z]

%{
#include "y.tab.h"
%}

%%

[,:;=\(\)\[\]#_\.+\-*/><~!^]    return *yytext;

"(0) CHARACTERISTICS"           return CHARACTERISTICS;
"(1) SYNTAX"                    return SYNTAX;
"(2) CANONICAL TRACES"          return CANONICAL;
"(3) SEMANTICS OF MUTATORS"     return SEMANTICS_OF_MUTATORS;
"(4) SEMANTICS OF OBSERVERS"    return SEMANTICS_OF_OBSERVERS;

"* type specified:"             return TYPE_SPECIFIED;
"* type parameter:"             return TYPE_PARAMETER;
"* value parameters:"           return VALUE_PARAMETERS;
"* foreign types:"              return FOREIGN_TYPES;
"mutator"                       return MUTATOR;
"observer"                      return OBSERVER;
"signal"                        return SIGNAL;
"args"                          return ARGS;
"result"                        return RESULT;
"non-det result"                return NON_DET_RESULT;
"->"                            return RIGHT_ARROW;
"::"                            return COLONS;
"..."                           return LDOTS;
```

```
"/\\"                           return AND;
"\\/"                           return OR;
"=>"                            return IMPLIES;
"<=>"                           return IFF;
"forall"                        return FORALL;
"exists"                        return EXISTS;
"false"                         return FALSE;
"true"                          return TRUE;
">="                            return GE;
"<="                            return LE;
"!="                            return NE;

"table"                         return TABLE;
"end"                           return END;

"legal"                         return LEGAL;
"feasible"                      return FEASIBLE;
"signals"                       return SIGNALS;
"transition"                    return TRANSITION;

{letter}({letter}|{digit}|_)*   return IDENT;
"-"?{digit}+                    return INTEGER;

[ \t\n+]                        ; /* ignoruj biale znaki */
.                               yyerror("Nieznany leksem");

%%

int yywrap() { return 1; }
```

A.2 Plik wejściowy dla programy yacc

```
%token AND
%token ARGS
%token CANONICAL
%token CHARACTERISTICS
%token END
%token EXISTS
%token FALSE
%token FEASIBLE
%token FORALL
%token FOREIGN_TYPES
%token GE
%token IDENT
%token IFF
%token IMPLIES
```

```
%token INTEGER
%token LDOTS
%token LE
%token LEGAL
%token MUTATOR
%token NE
%token NON_DET_RESULT
%token OBSERVER
%token OR
%token COLONS
%token RESULT
%token RIGHT_ARROW
%token SEMANTICS_OF_MUTATORS
%token SEMANTICS_OF_OBSERVERS
%token SIGNAL
%token SIGNALS
%token SYNTAX
%token TABLE
%token TRANSITION
%token TRUE
%token TYPE_PARAMETER
%token TYPE_SPECIFIED
%token VALUE_PARAMETERS
%token IDENT
%token INTEGER
%token CHARACTERISTICS
%token SYNTAX
%token CANONICAL
%token SEMANTICS_OF_MUTATORS
%token SEMANTICS_OF_OBSERVERS

%nonassoc PAR_REDUCE
%nonassoc GENERAL_LIST
%nonassoc SPECIFIC_LIST
%nonassoc ')'
%nonassoc '#'
%left AND
%left OR
%left IFF IMPLIES
%nonassoc '~'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
%left '.'
%left '^'

%{
```

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

%%

specification:
        characteristics syntax canonical semanticsMutators semanticsObservers;

characteristics:
        CHARACTERISTICS TYPE_SPECIFIED IDENT typeParameters valueParameters
        foreignTypes;

typeParameters:
                                                                /* empty word */
      | typeParameter typeParameters;

typeParameter:
        TYPE_PARAMETER IDENT mutatorList observerList;

valueParameters:
                                                                /* empty word */
      | VALUE_PARAMETERS parameterList;

parameterList:
        parameter
      | parameterList ';' parameter;

parameter:
        identList ':' IDENT;

foreignTypes:
                                                                /* empty word */
      | FOREIGN_TYPES identList;

identList:
        IDENT
      | identList ',' IDENT;

syntax:
        SYNTAX mutatorList observerList signalList;

mutatorList:
                                                                /* empty word */
      | mutatorList mutator;

mutator:
        MUTATOR IDENT argTypes nonDetResult;
```

```
observerList:
                                                        /* empty word */
      | observerList observer;

observer:
        OBSERVER IDENT argTypes RESULT IDENT;

signalList:
                                                        /* empty word */
      | signalList signal;

signal:
        SIGNAL IDENT argTypes;

argTypes:
                                                        /* empty word */
      | ARGS argTypeList;

argTypeList:
        IDENT
      | argTypeList IDENT;

nonDetResult:
                                                        /* empty word */
      | NON_DET_RESULT IDENT;

canonical:
        CANONICAL auxDefinitionList;

auxDefinitionList:
        auxDefinition
      | auxDefinitionList auxDefinition;

auxDefinition:
        signature
      | domainRestriction
      | valueDefinition;

signature:
        IDENT ':' argTypeList RIGHT_ARROW IDENT;

domainRestriction:
        LEGAL '(' IDENT '(' traceExprList ')' ')' IFF logExpr;

valueDefinition:
        IDENT '(' traceExprList ')' '=' traceExpr
```

```
      | IDENT '(' traceExprList ')' IFF logExpr;   /* if the result is bool */

semanticsMutators:
        SEMANTICS_OF_MUTATORS emptyEquivalence mutatorDefinitionList;

emptyEquivalence:
                                                                /* empty word */
      | '_' IMPLIES traceExpr;

mutatorDefinitionList:
        mutatorDefinition
      | mutatorDefinitionList mutatorDefinition;

mutatorDefinition:
        mutatorLegality
      | mutatorFeasibility
      | mutatorSignals
      | mutatorTransition;

mutatorLegality:
        LEGAL      '(' traceExpr ';' application ')' IFF logExpr;

mutatorFeasibility:
        FEASIBLE   '(' traceExpr ';' application ')' IFF logExpr;

mutatorSignals:
        SIGNALS    '(' traceExpr ';' application ')' '=' signalExpr;

mutatorTransition:
        TRANSITION '(' traceExpr ';' application ')' '=' traceExpr;

semanticsObservers:
                                                                /* empty word */
      | SEMANTICS_OF_OBSERVERS observerDefinitionList;

observerDefinitionList:
        observerDefinition
      | observerDefinitionList observerDefinition;

observerDefinition:
        observerLegality
      | observerSignals
      | observerResult;

observerLegality:
        LEGAL   '(' application ')' IFF logExpr;
```

```
observerSignals:
        SIGNALS '(' application ')' '=' signalExpr;

observerResult:
        RESULT  '(' application ')' '=' traceExpr
      | RESULT  '(' application ')' IFF logExpr;   /* if the result is bool */

application:
        IDENT '(' traceExprList ')'
      | IDENT '(' traceExprList ')' '^' traceExpr;

applicationList:
        application
      | applicationList ',' application;

foreignApplication:
        IDENT COLONS IDENT '('               traceExprList ')'
      | IDENT COLONS IDENT '(' traceExpr ';' traceExprList ')'
      | IDENT COLONS IDENT '(' traceExpr ';' traceExprList ')' '^' traceExpr;

varDeclarationList:
        varDeclaration
      | varDeclarationList ';' varDeclaration;

varDeclaration:
        variableDeclarationList ':' IDENT;

variableDeclarationList:
        variableDeclaration
      | variableDeclarationList ',' variableDeclaration;

variableDeclaration:
        IDENT
      | IDENT '[' traceExprList LDOTS traceExprList ']';

signalExpr:
        '[' ']'
      | '[' applicationList ']'
      | signalTable;

commonExpr:
        IDENT                              /* variable                    */
      | IDENT '[' nonEmptyTraceExprList ']' /* indexed variable            */
      | application                        /* invocation or aux. function  */
      | foreignApplication                 /* foreign invocation           */
      | '(' commonExpr ')'
      | commonTable;
```

```
logExpr:
        commonExpr                      %prec PAR_REDUCE
      | TRUE
      | FALSE
      | traceExpr '='   traceExpr
      | traceExpr '<'   traceExpr
      | traceExpr '>'   traceExpr
      | traceExpr GE    traceExpr
      | traceExpr LE    traceExpr
      | traceExpr NE    traceExpr
      | '(' logExpr ')'
      | logExpr AND     logExpr
      | logExpr OR      logExpr
      | logExpr IMPLIES logExpr
      | logExpr IFF     logExpr
      | '~' logExpr
      | FORALL varDeclarationList constraint '[' logExpr ']'
      | EXISTS varDeclarationList constraint '[' logExpr ']'
      | EXISTS '!' varDeclarationList constraint '[' logExpr ']'
      | logTable;

constraint:
                                                        /* empty word */
      | '(' logExpr ')';

traceExpr:
        commonExpr                      %prec PAR_REDUCE
      | INTEGER
      | '_'                             /* empty trace                    */
      | IDENT COLONS '_'                /* initial state of foreign object  */
      | traceExpr '.' traceExpr         /* trace concatenation            */
      | '(' traceExpr ')'
      | '[' traceExpr ']' '(' IDENT '=' traceExpr LDOTS traceExpr ')'
                                        /* iteration                      */
      | traceExpr '+' traceExpr
      | traceExpr '-' traceExpr
      | traceExpr '*' traceExpr
      | traceExpr '/' traceExpr
      | '-' traceExpr                   %prec UMINUS
      | traceTable;

traceExprList:
                                                        /* empty word */
      | nonEmptyTraceExprList;

nonEmptyTraceExprList:
```

```
        traceExpr
      | traceExprList ',' traceExpr;

traceTable:
        TABLE
          traceRowList
        END;

traceRowList:
        onlyTraceRowList                %prec GENERAL_LIST
      | commonRowList onlyTraceRowList  %prec GENERAL_LIST
      | traceRowList  onlyTraceRowList  %prec GENERAL_LIST
      | traceRowList  commonRowList     %prec GENERAL_LIST;

onlyTraceRowList:
        traceRow
      | onlyTraceRowList traceRow       %prec SPECIFIC_LIST;

traceRow:
        '#' logExpr '#' traceExpr '#';

logTable:
        TABLE
          logRowList
        END;

logRowList:
        onlyLogRowList                  %prec GENERAL_LIST
      | commonRowList onlyLogRowList    %prec GENERAL_LIST
      | logRowList commonRowList        %prec GENERAL_LIST
      | logRowList onlyLogRowList       %prec GENERAL_LIST;

onlyLogRowList:
        logRow
      | onlyLogRowList logRow           %prec SPECIFIC_LIST;

logRow:
        '#' logExpr '#' logExpr '#';

commonTable:
        TABLE
          commonRowList
        END;

commonRowList:
        commonRow
      | commonRowList commonRow         %prec SPECIFIC_LIST;
```

```
commonRow:
        '#' logExpr '#' commonExpr '#';

signalTable:
        TABLE
          signalRowList
        END;

signalRowList:
        signalRow
      | signalRowList signalRow;

signalRow:
        '#' logExpr '#' signalExpr '#';

%%

int yyerror(char *s)
{
  fprintf(stderr, "%s\n", s);
  exit(1);
}

int main()
{
  yyparse();
  return 0;
}
```

Dodatek B

Reguły statycznej poprawności specyfikacji

B.1 Specyfikacja

specification ::= *characteristics syntax canonical semanticsMutators semanticsObservers*

$$\frac{\Theta \vdash \mathit{characteristics} \Longrightarrow \Phi_1 \qquad \Phi_1 \vdash \mathit{syntax} \Longrightarrow \Phi_2 \quad \Phi_2 \vdash \mathit{canonical} \Longrightarrow \Phi \qquad \Phi \vdash \mathit{semanticsMutators} \Longrightarrow \mathbf{OK} \qquad \Phi \vdash \mathit{semanticsObservers} \Longrightarrow \mathbf{OK}}{\Theta \vdash \mathit{specification} \Longrightarrow \Phi}$$

B.2 Sekcja CHARACTERISTICS

characteristics ::= `(0) CHARACTERISTICS * type specified:` *IDENT*
typeParameter$_1$ ... *typeParameter*$_n$ *valueParameters foreignTypes*

$$\frac{\mathit{IDENT} \neq \text{“traces”} \qquad \Theta[\mathbf{specified} := \mathit{IDENT}][\mathbf{programs}(\mathit{IDENT}) := \lambda x.\mathbf{error}] = \Phi_0 \qquad \Phi_0 \vdash \mathit{typeParameter}_1 \Longrightarrow \Phi_1 \quad \ldots \quad \Phi_{n-1} \vdash \mathit{typeParameter}_n \Longrightarrow \Phi_n \qquad \Phi_n \vdash \mathit{valueParameters} \Longrightarrow \Psi \qquad \Psi \vdash \mathit{foreignTypes} \Longrightarrow \Phi}{\Theta \vdash \mathit{characteristics} \Longrightarrow \Phi}$$

typeParameter ::= `* type parameter:` *IDENT* *mutator*$_1$ ... *mutator*$_n$ *observer*$_1$ ... *observer*$_m$

$$\frac{\mathit{IDENT} \neq \text{“traces”} \qquad \Phi.\mathbf{programs}(\mathit{IDENT}) = \mathbf{error} \qquad \Phi[\mathbf{specified} := \mathit{IDENT}][\mathbf{programs}(\mathit{IDENT}) := \lambda x.\mathbf{error}] = \Phi_0 \qquad \Phi_0 \vdash \mathit{mutator}_1 \Longrightarrow \Phi_1 \quad \ldots \quad \Phi_{n-1} \vdash \mathit{mutator}_n \Longrightarrow \Phi_n \qquad \Phi_n = \Psi_0 \quad \Psi_0 \vdash \mathit{observer}_1 \Longrightarrow \Psi_1 \quad \ldots \quad \Psi_{m-1} \vdash \mathit{observer}_m \Longrightarrow \Psi_m}{\Phi \vdash \mathit{typeParameter} \Longrightarrow \Psi_m[\mathbf{specified} := \Phi.\mathbf{specified}]}$$

$\boxed{\textit{valueParameters} ::= \varepsilon}$

$$\frac{}{\Phi \vdash \textit{valueParameters} \Longrightarrow \Phi}$$

$\boxed{\textit{valueParameters} ::= \texttt{* value parameters:}\ \textit{parameter}_1; \ldots; \textit{parameter}_n}$

$$\frac{\Phi_0 \vdash \textit{parameter}_1 \Longrightarrow \Phi_1 \quad \ldots \quad \Phi_{n-1} \vdash \textit{parameter}_n \Longrightarrow \Phi_n}{\Phi_0 \vdash \textit{valueParameters} \Longrightarrow \Phi_n}$$

$\boxed{\textit{parameter} ::= \textit{IDENT}_1, \ldots, \textit{IDENT}_n : \textit{IDENT}_{\textit{type}}}$

$$\frac{\begin{array}{cc} & \Phi_0.\mathbf{programs}(\textit{IDENT}_{\textit{type}}) \neq \mathbf{error} \\ \Phi_0.\mathbf{variables}(\textit{IDENT}_1) = \mathbf{error} & \Phi_1 = \Phi_0[\mathbf{variables}(\textit{IDENT}_1) := \langle \textit{IDENT}_{\textit{type}}, 0\rangle] \\ \vdots & \vdots \\ \Phi_{n-1}.\mathbf{variables}(\textit{IDENT}_n) = \mathbf{error} & \Phi_n = \Phi_{n-1}[\mathbf{variables}(\textit{IDENT}_n) := \langle \textit{IDENT}_{\textit{type}}, 0\rangle] \end{array}}{\Phi_0 \vdash \textit{parameter} \Longrightarrow \Phi_n}$$

$\boxed{\textit{foreignTypes} ::= \varepsilon}$

$$\frac{}{\Phi \vdash \textit{foreignTypes} \Longrightarrow \Phi}$$

$\boxed{\textit{foreignTypes} ::= \texttt{* foreign types:}\ \textit{IDENT}_1, \ldots, \textit{IDENT}_n}$

$$\frac{\begin{array}{cc} \vdash \textit{specification}_{\textit{IDENT}_1} \Longrightarrow \Phi_1 & \ldots \quad \vdash \textit{specification}_{\textit{IDENT}_n} \Longrightarrow \Phi_n \\ \textit{IDENT}_1 \neq \text{“traces”} & \ldots \quad \textit{IDENT}_n \neq \text{“traces”} \\ \Psi_0.\mathbf{programs}(\textit{IDENT}_1) = \mathbf{error} & \\ \vdots & \Psi_1 = \Psi_0[\mathbf{programs}(\textit{IDENT}_1) := \Phi_1.\mathbf{programs}(\textit{IDENT}_1)] \\ \vdots & \vdots \\ \Psi_{n-1}.\mathbf{programs}(\textit{IDENT}_n) = \mathbf{error} & \vdots \\ & \Psi_n = \Psi_{n-1}[\mathbf{programs}(\textit{IDENT}_n) := \Phi_n.\mathbf{programs}(\textit{IDENT}_n)] \end{array}}{\Psi_0 \vdash \textit{foreignTypes} \Longrightarrow \Psi_n}$$

Uwaga: Wyrażenie $\textit{specification}_{\textit{IDENT}}$ oznacza specyfikację typu $\textit{IDENT}$.

B.3 Sekcja SYNTAX

$\boxed{\mathit{syntax} ::= \texttt{(1) SYNTAX}\ \mathit{mutator}_1\ \ldots\ \mathit{mutator}_n\ \mathit{observer}_1\ \ldots\ \mathit{observer}_m\ \mathit{signal}_1\ \ldots\ \mathit{signal}_p}$

$$\frac{\begin{array}{ccccc} & \Phi_0 \vdash \mathit{mutator}_1 \Longrightarrow \Phi_1 & \ldots & \Phi_{n-1} \vdash \mathit{mutator}_n \Longrightarrow \Phi_n \\ \Phi_n = \Psi_0 & \Psi_0 \vdash \mathit{observer}_1 \Longrightarrow \Psi_1 & \ldots & \Psi_{m-1} \vdash \mathit{observer}_m \Longrightarrow \Psi_m \\ \Psi_m = \Gamma_0 & \Gamma_0 \vdash \mathit{signal}_1 \Longrightarrow \Gamma_1 & \ldots & \Gamma_{p-1} \vdash \mathit{signal}_p \Longrightarrow \Gamma_p \end{array}}{\Phi_0 \vdash \mathit{syntax} \Longrightarrow \Gamma_p}$$

$\boxed{\mathit{mutator} ::= \mathit{IDENT}\ \mathit{argTypes}\ \mathit{nonDetResult}}$

$$\frac{\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(\mathit{IDENT}) = \mathbf{error} \quad \Phi \vdash \mathit{argTypes} \Longrightarrow \mathit{tList} \quad \Phi \vdash \mathit{nonDetResult} \Longrightarrow \langle\rangle}{\Phi \vdash \mathit{mutator} \Longrightarrow \Phi[\mathbf{programs}(\Phi.\mathbf{specified})(\mathit{IDENT}) := \langle \mathbf{m}, \mathit{tList} \rangle]}$$

$$\frac{\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(\mathit{IDENT}) = \mathbf{error} \quad \Phi \vdash \mathit{argTypes} \Longrightarrow \mathit{tList} \quad \Phi \vdash \mathit{nonDetResult} \Longrightarrow \langle rt \rangle}{\Phi \vdash \mathit{mutator} \Longrightarrow \Phi[\mathbf{programs}(\Phi.\mathbf{specified})(\mathit{IDENT}) := \langle \mathbf{m}, \mathit{tList}, rt \rangle]}$$

$\boxed{\mathit{observer} ::= \mathit{IDENT}_{name}\ \mathit{argTypes}\ \texttt{result}\ \mathit{IDENT}_{result}}$

$$\frac{\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(\mathit{IDENT}_{name}) = \mathbf{error} \quad \Phi \vdash \mathit{argTypes} \Longrightarrow \mathit{tList} \quad \Phi.\mathbf{programs}(\mathit{IDENT}_{result}) \neq \mathbf{error}}{\Phi \vdash \mathit{observer} \Longrightarrow \Phi[\mathbf{programs}(\Phi.\mathbf{specified})(\mathit{IDENT}_{name}) := \langle \mathbf{o}, \mathit{tList}, \mathit{IDENT}_{result} \rangle]}$$

$\boxed{\mathit{signal} ::= \texttt{signal}\ \mathit{IDENT}\ \mathit{argTypes}}$

$$\frac{\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(\mathit{IDENT}) = \mathbf{error} \quad \Phi \vdash \mathit{argTypes} \Longrightarrow \mathit{tList}}{\Phi \vdash \mathit{signal} \Longrightarrow \Phi[\mathbf{programs}(\Phi.\mathbf{specified})(\mathit{IDENT}) := \langle \mathbf{s}, \mathit{tList} \rangle]}$$

$\boxed{\mathit{argTypes} ::= \varepsilon}$

$$\frac{}{\Phi \vdash \mathit{argTypes} \Longrightarrow [\,]}$$

$\boxed{\textit{argTypes} ::= \texttt{args}\ \textit{IDENT}_1\ \ldots\ \textit{IDENT}_n}$

$$\frac{\Phi.\textbf{programs}(\textit{IDENT}_1) \neq \textbf{error} \quad \ldots \quad \Phi.\textbf{programs}(\textit{IDENT}_n) \neq \textbf{error}}{\Phi \vdash \textit{argTypes} \Longrightarrow [\textit{IDENT}_1, \ldots, \textit{IDENT}_n]}$$

$\boxed{\textit{nonDetResult} ::= \varepsilon}$

$$\frac{}{\Phi \vdash \textit{nonDetResult} \Longrightarrow \langle\rangle}$$

$\boxed{\textit{nonDetResult} ::= \texttt{non-det result}\ \textit{IDENT}}$

$$\frac{\Phi.\textbf{specified} \neq \textit{IDENT} \qquad \Phi.\textbf{programs}(\textit{IDENT}) \neq \textbf{error}}{\Phi \vdash \textit{nonDetResult} \Longrightarrow \langle \textit{IDENT} \rangle}$$

B.4 Sekcja CANONICAL

Uwaga: W opisie sekcji CANONICAL środowisko jest rozszerzone o nazwę właśnie definiowanej funkcji. Nazwę tej funkcji zapisujemy po symbolu oznaczającym środowisko i przecinku.

$\boxed{\textit{canonical} ::= \texttt{(2) CANONICAL TRACES}\ \textit{auxDefinition}_1\ \ldots\ \textit{auxDefinition}_n}$

$$\frac{\begin{array}{c}\Phi_0, \textbf{error} \vdash \textit{auxDefinition}_1 \Longrightarrow \Phi_1, f_1 \quad \ldots \quad \Phi_{n-1}, f_{n-1} \vdash \textit{auxDefinition}_n \Longrightarrow \Phi_n, f_n \\ \Phi_n.\textbf{functions}(\text{“canonical”}) \neq \textbf{error}\end{array}}{\Phi_0 \vdash \textit{canonical} \Longrightarrow \Phi_n}$$

$\boxed{\textit{auxDefinition} ::= \textit{signature}}$

$$\frac{\Phi \vdash \textit{signature} \Longrightarrow \Psi, g}{\Phi, f \vdash \textit{auxDefinition} \Longrightarrow \Psi, g}$$

$\boxed{\textit{auxDefinition} ::= \textit{domainRestriction}}$

$$\frac{\Phi, f \vdash \textit{domainRestriction} \Longrightarrow \textbf{OK}}{\Phi, f \vdash \textit{auxDefinition} \Longrightarrow \Phi, f}$$

auxDefinition ::= *valueDefinition*

$$\frac{\Phi, f \vdash \textit{valueDefinition} \Longrightarrow \textbf{OK}}{\Phi, f \vdash \textit{auxDefinition} \Longrightarrow \Phi, f}$$

signature ::= IDENT_{name} : $\mathit{IDENT}_1 \ \ldots \ \mathit{IDENT}_n$ -> IDENT_{n+1}

$$\frac{\begin{array}{l}\Phi.\textbf{functions}(\mathit{IDENT}_{name}) = \textbf{error} \qquad \mathit{IDENT}_{name} \neq \text{“canonical”} \\ \Phi.\textbf{programs}(\Phi.\textbf{specified})(\mathit{IDENT}_{name}) = \textbf{error} \\ \text{dla } i = 1, 2, \ldots, n+1 \quad \Phi.\textbf{programs}(\mathit{IDENT}_i) \neq \textbf{error} \text{ i } \Phi.\textbf{specified} \neq (\mathit{IDENT}_i) \\ \qquad\qquad \text{lub} \quad \mathit{IDENT}_i = \text{“traces”} \\ \qquad\qquad \text{lub} \quad \mathit{IDENT}_i = \Phi.\textbf{specified} \text{ i } \Phi.\textbf{functions}(\text{“canonical”}) \neq \textbf{error}\end{array}}{\begin{array}{r}\Phi \vdash \textit{signature} \Longrightarrow \Phi[\textbf{functions}(\mathit{IDENT}_{name}) := \langle[\mathit{IDENT}_1, \ldots, \mathit{IDENT}_n], \mathit{IDENT}_{n+1}\rangle], \\ \mathit{IDENT}_{name}\end{array}}$$

$$\frac{\begin{array}{lll}\Phi.\textbf{functions}(\text{“canonical”}) = \textbf{error} & & \mathit{IDENT}_{name} = \text{“canonical”} \\ n = 1 \qquad \mathit{IDENT}_1 = \text{“traces”} & & \mathit{IDENT}_2 = \text{“bool”}\end{array}}{\Phi \vdash \textit{signature} \Longrightarrow \Phi[\textbf{functions}(\text{“canonical”}) := \langle[\text{“traces”}], \text{“bool”}\rangle], \text{“canonical”}}$$

domainRestriction ::= `legal ( IDENT (` $\mathit{traceExpr}_1, \ldots, \mathit{traceExpr}_n$ `) ) <=>` *logExpr*

$$\frac{\begin{array}{l}\mathit{IDENT} \neq \text{“canonical”} \qquad \Phi.\textbf{functions}(\mathit{IDENT}) = \langle[s_1, \ldots, s_n], s_{n+1}\rangle \\ \text{dla } i = 1, \ldots, n \text{ jeśli } s_i \neq \text{“traces”, to } t_i = s_i, \text{ jeśli } s_i = \text{“traces”, to } t_i = \Phi.\textbf{specified} \\ \Psi_{min} \vdash \textit{logExpr} \Longrightarrow \textbf{OK} \\ \Psi_{min} \text{ jest najmniejszym wg relacji “}\preceq\text{” środowiskiem, o następujących własnościach} \\ \quad (i) \quad \Phi \preceq \Psi_{min} \\ \quad (ii) \quad \Psi_{min} \vdash \mathit{traceExpr}_1 \Longrightarrow t_1 \quad \ldots \quad \Psi_{min} \vdash \mathit{traceExpr}_n \Longrightarrow t_n\end{array}}{\Phi, \mathit{IDENT} \vdash \textit{domainRestriction} \Longrightarrow \textbf{OK}}$$

valueDefinition ::= *IDENT* `(` $\mathit{traceExpr}_1, \ldots, \mathit{traceExpr}_n$ `) =` $\mathit{traceExpr}_{result}$

$$\frac{\begin{array}{l}\Phi.\textbf{functions}(\mathit{IDENT}) = \langle[s_1, \ldots, s_n], s_{n+1}\rangle \qquad s_{n+1} \neq \text{“bool”} \\ \text{dla } i = 1, \ldots, n+1 \text{ jeśli } s_i \neq \text{“traces”, to } t_i = s_i, \text{ jeśli } s_i = \text{“traces”, to } t_i = \Phi.\textbf{specified} \\ \Psi_{min} \vdash \mathit{traceExpr}_{result} \Longrightarrow t_{n+1} \\ \Psi_{min} \text{ jest najmniejszym wg relacji “}\preceq\text{” środowiskiem, o następujących własnościach} \\ \quad (i) \quad \Phi \preceq \Psi_{min} \\ \quad (ii) \quad \Psi_{min} \vdash \mathit{traceExpr}_1 \Longrightarrow t_1 \quad \ldots \quad \Psi_{min} \vdash \mathit{traceExpr}_n \Longrightarrow t_n\end{array}}{\Phi, \mathit{IDENT} \vdash \textit{valueDefinition} \Longrightarrow \textbf{OK}}$$

valueDefinition ::= *IDENT* `(` *traceExpr*$_1$, ..., *traceExpr*$_n$ `)` `<=>` *logExpr*

$$\Phi.\textbf{functions}(\mathit{IDENT}) = \langle [s_1, \ldots, s_n], \text{“bool”} \rangle$$

dla $i = 1, \ldots, n$ jeśli $s_i \neq$ “traces”, to $t_i = s_i$, jeśli $s_i =$ “traces”, to $t_i = \Phi.\textbf{specified}$

$$\Psi_{min} \vdash \mathit{logExpr} \Longrightarrow \textbf{OK}$$

Ψ_{min} jest najmniejszym wg relacji “$\preceq$” środowiskiem, o następujących własnościach

(i) $\quad \Phi \preceq \Psi_{min}$

(ii) $\quad \Psi_{min} \vdash \mathit{traceExpr}_1 \Longrightarrow t_1 \quad \ldots \quad \Psi_{min} \vdash \mathit{traceExpr}_n \Longrightarrow t_n$

$$\Phi, \mathit{IDENT} \vdash \mathit{valueDefinition} \Longrightarrow \textbf{OK}$$

B.5 Sekcja SEMANTICS OF MUTATORS

semanticsMutators ::= `(3) SEMANTICS OF MUTATORS` *emptyEquivalence* *mutatorDefinition*$_1$ ... *mutatorDefinition*$_n$

$$\frac{\Phi \vdash \mathit{emptyEquivalence} \Longrightarrow \textbf{OK} \quad \Phi \vdash \mathit{mutatorDefinition}_1 \Longrightarrow \textbf{OK} \quad \ldots \quad \Phi \vdash \mathit{mutatorDefinition}_n \Longrightarrow \textbf{OK}}{\Phi \vdash \mathit{semanticsMutators} \Longrightarrow \textbf{OK}}$$

emptyEquivalence ::= ε

$$\frac{}{\Phi \vdash \mathit{emptyEquivalence} \Longrightarrow \textbf{OK}}$$

emptyEquivalence ::= `_ =>` *traceExpr*

$$\frac{\Phi \vdash \mathit{traceExpr} \Longrightarrow \Phi.\textbf{specified}}{\Phi \vdash \mathit{emptyEquivalence} \Longrightarrow \textbf{OK}}$$

mutatorDefinition ::= *mutatorLegality*

$$\frac{\Phi \vdash \mathit{mutatorLegality} \Longrightarrow \textbf{OK}}{\Phi \vdash \mathit{mutatorDefinition} \Longrightarrow \textbf{OK}}$$

mutatorDefinition ::= *mutatorFeasibility*

$$\frac{\Phi \vdash \textit{mutatorFeasibility} \Longrightarrow \textbf{OK}}{\Phi \vdash \textit{mutatorDefinition} \Longrightarrow \textbf{OK}}$$

mutatorDefinition ::= *mutatorSignals*

$$\frac{\Phi \vdash \textit{mutatorSignals} \Longrightarrow \textbf{OK}}{\Phi \vdash \textit{mutatorDefinition} \Longrightarrow \textbf{OK}}$$

mutatorDefinition ::= *mutatorTransition*

$$\frac{\Phi \vdash \textit{mutatorTransition} \Longrightarrow \textbf{OK}}{\Phi \vdash \textit{mutatorDefinition} \Longrightarrow \textbf{OK}}$$

mutatorLegality ::= `legal (` *traceExpr* `;` *application* `) <=>` *logExpr*

$$\frac{\begin{array}{c}\Psi_{min} \vdash \textit{logExpr} \Longrightarrow \textbf{OK} \\ \Psi_{min} \text{ jest najmniejszym wg relacji “}\preceq\text{” środowiskiem, o następujących własnościach} \\ (i) \quad \Phi \preceq \Psi_{min} \\ (ii) \quad \Psi_{min} \vdash \textit{traceExpr} \Longrightarrow \Phi.\textbf{specified} \\ (iii) \quad \Psi_{min} \vdash \textit{application} \Longrightarrow \textbf{m without result}\end{array}}{\Phi \vdash \textit{mutatorLegality} \Longrightarrow \textbf{OK}}$$

mutatorFeasibility ::= `feasible (` *traceExpr* `;` *application* `) <=>` *logExpr*

$$\frac{\begin{array}{c}\Psi_{min} \vdash \textit{logExpr} \Longrightarrow \textbf{OK} \\ \Psi_{min} \text{ jest najmniejszym wg relacji “}\preceq\text{” środowiskiem, o następujących własnościach} \\ (i) \quad \Phi \preceq \Psi_{min} \\ (ii) \quad \Psi_{min} \vdash \textit{traceExpr} \Longrightarrow \Phi.\textbf{specified} \\ (iii) \quad \Psi_{min} \vdash \textit{application} \Longrightarrow \textbf{m with result}\end{array}}{\Phi \vdash \textit{mutatorFeasibility} \Longrightarrow \textbf{OK}}$$

mutatorSignals ::= `signals ( traceExpr ; application ) = signalExpr`

$$\Psi_{min} \vdash \mathit{signalExpr} \Longrightarrow \mathbf{OK}$$

Ψ_{min} jest najmniejszym wg relacji "$\preceq$" środowiskiem, o następujących własnościach

(i) $\quad \Phi \preceq \Psi_{min}$

(ii) $\quad \Psi_{min} \vdash \mathit{traceExpr} \Longrightarrow \Phi.\mathbf{specified}$

(iii) $\quad \Psi_{min} \vdash \mathit{application} \Longrightarrow \mathbf{m\ complete}$

$$\Phi \vdash \mathit{mutatorSignals} \Longrightarrow \mathbf{OK}$$

mutatorTransition ::= `transition ( traceExpr`$_{arg}$ `; application ) = traceExpr`$_{value}$

$$\Psi_{min} \vdash \mathit{traceExpr}_{value} \Longrightarrow \Phi.\mathbf{specified}$$

Ψ_{min} jest najmniejszym wg relacji "$\preceq$" środowiskiem, o następujących własnościach

(i) $\quad \Phi \preceq \Psi_{min}$

(ii) $\quad \Psi_{min} \vdash \mathit{traceExpr}_{arg} \Longrightarrow \Phi.\mathbf{specified}$

(iii) $\quad \Psi_{min} \vdash \mathit{application} \Longrightarrow \mathbf{m\ complete}$

$$\Phi \vdash \mathit{mutatorTransition} \Longrightarrow \mathbf{OK}$$

B.6 Sekcja SEMANTICS OF OBSERVERS

semanticsObservers ::= ε

$$\Phi \vdash \mathit{semanticsObservers} \Longrightarrow \mathbf{OK}$$

semanticsObservers ::= `(4) SEMANTICS OF OBSERVERS`
observerDefinition$_1$ ... *observerDefinition*$_n$

$$\Phi \vdash \mathit{observerDefinition}_1 \Longrightarrow \mathbf{OK} \quad \ldots \quad \Phi \vdash \mathit{observerDefinition}_n \Longrightarrow \mathbf{OK}$$
$$\Phi \vdash \mathit{semanticsObservers} \Longrightarrow \mathbf{OK}$$

observerDefinition ::= *observerLegality*

$$\Phi \vdash \mathit{observerLegality} \Longrightarrow \mathbf{OK}$$
$$\Phi \vdash \mathit{observerDefinition} \Longrightarrow \mathbf{OK}$$

observerDefinition ::= *observerSignals*

$$\frac{\Phi \vdash \textit{observerSignals} \Longrightarrow \textbf{OK}}{\Phi \vdash \textit{observerDefinition} \Longrightarrow \textbf{OK}}$$

observerDefinition ::= *observerResult*

$$\frac{\Phi \vdash \textit{observerResult} \Longrightarrow \textbf{OK}}{\Phi \vdash \textit{observerDefinition} \Longrightarrow \textbf{OK}}$$

observerLegality ::= `legal (` *application* `) <=>` *logExpr*

$$\Psi_{min} \vdash \textit{logExpr} \Longrightarrow \textbf{OK}$$

Ψ_{min} jest najmniejszym wg relacji "$\preceq$" środowiskiem, o następujących własnościach

(*i*) $\Phi \preceq \Psi_{min}$

(*ii*) $\Psi_{min} \vdash \textit{application} \Longrightarrow \mathbf{o}\ \textit{resultType}$

$$\Phi \vdash \textit{observerLegality} \Longrightarrow \textbf{OK}$$

observerSignals ::= `signals (` *application* `) =` *signalExpr*

$$\Psi_{min} \vdash \textit{signalExpr} \Longrightarrow \textbf{OK}$$

Ψ_{min} jest najmniejszym wg relacji "$\preceq$" środowiskiem, o następujących własnościach

(*i*) $\Phi \preceq \Psi_{min}$

(*ii*) $\Psi_{min} \vdash \textit{application} \Longrightarrow \mathbf{o}\ \textit{resultType}$

$$\Phi \vdash \textit{observerSignals} \Longrightarrow \textbf{OK}$$

observerResult ::= `result (` *application* `) =` *traceExpr*

$$\Psi_{min} \vdash \textit{traceExpr} \Longrightarrow \textit{resultType}$$

Ψ_{min} jest najmniejszym wg relacji "$\preceq$" środowiskiem, o następujących własnościach

(*i*) $\Phi \preceq \Psi_{min}$

(*ii*) $\Psi_{min} \vdash \textit{application} \Longrightarrow \mathbf{o}\ \textit{resultType}$

(*iii*) $\textit{resultType} \neq$ "bool"

$$\Phi \vdash \textit{observerResult} \Longrightarrow \textbf{OK}$$

observerResult ::= `result ( ` *application* ` ) <=> ` *logExpr*

$$\Psi_{min} \vdash \mathit{logExpr} \Longrightarrow \mathbf{OK}$$

Ψ_{min} jest najmniejszym wg relacji "$\preceq$" środowiskiem, o następujących własnościach

(i) $\quad \Phi \preceq \Psi_{min}$

(ii) $\quad \Psi_{min} \vdash \mathit{application} \Longrightarrow \mathbf{o}$ "bool"

$$\Phi \vdash \mathit{observerResult} \Longrightarrow \mathbf{OK}$$

B.7 Wywołania specyfikowanych programów dostępu i funkcji pomocniczych

application ::= *IDENT* `(` $\mathit{traceExpr}_1, \ldots, \mathit{traceExpr}_n$ `)`

$$\frac{\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(\mathit{IDENT}) = \langle \mathbf{s}, [t_1, \ldots, t_n] \rangle \qquad \Phi \vdash \mathit{traceExpr}_1 \Longrightarrow t_1 \quad \ldots \quad \Phi \vdash \mathit{traceExpr}_n \Longrightarrow t_n}{\Phi \vdash \mathit{application} \Longrightarrow \mathbf{s}}$$

$$\frac{\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(\mathit{IDENT}) = \langle \mathbf{o}, [t_1, \ldots, t_n], t_{n+1} \rangle \qquad \Phi \vdash \mathit{traceExpr}_1 \Longrightarrow t_1 \quad \ldots \quad \Phi \vdash \mathit{traceExpr}_n \Longrightarrow t_n}{\Phi \vdash \mathit{application} \Longrightarrow \mathbf{o}\ t_{n+1}}$$

$$\frac{\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(\mathit{IDENT}) = \langle \mathbf{m}, [t_1, \ldots, t_n] \rangle \qquad \Phi \vdash \mathit{traceExpr}_1 \Longrightarrow t_1 \quad \ldots \quad \Phi \vdash \mathit{traceExpr}_n \Longrightarrow t_n}{\begin{array}{c}\Phi \vdash \mathit{application} \Longrightarrow \mathbf{m\ complete} \\ \Phi \vdash \mathit{application} \Longrightarrow \mathbf{m\ without\ result} \\ \Phi \vdash \mathit{application} \Longrightarrow \Phi.\mathbf{specified}\end{array}}$$

$$\frac{\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(\mathit{IDENT}) = \langle \mathbf{m}, [t_1, \ldots, t_n], t_{n+1} \rangle \qquad \Phi \vdash \mathit{traceExpr}_1 \Longrightarrow t_1 \quad \ldots \quad \Phi \vdash \mathit{traceExpr}_n \Longrightarrow t_n}{\Phi \vdash \mathit{application} \Longrightarrow \mathbf{m\ without\ result}}$$

$$\frac{\begin{array}{c}\Phi.\mathbf{functions}(\mathit{IDENT}) = \langle [s_1, \ldots, s_n], s_{n+1} \rangle \\ \text{dla } i = 1, \ldots, n+1 \text{ jeśli } s_i \neq \text{"traces"}, \text{ to } t_i = s_i, \text{ jeśli } s_i = \text{"traces"}, \text{ to } t_i = \Phi.\mathbf{specified} \\ \Phi \vdash \mathit{traceExpr}_1 \Longrightarrow t_1 \quad \ldots \quad \Phi \vdash \mathit{traceExpr}_n \Longrightarrow t_n\end{array}}{\Phi \vdash \mathit{application} \Longrightarrow t_{n+1}}$$

$$\boxed{\mathit{application} ::= \mathit{IDENT}\ \texttt{(}\ \mathit{traceExpr}_1, \ldots, \mathit{traceExpr}_n\ \texttt{)}\ \texttt{\^{}}\ \mathit{traceExpr}_{result}}$$

$$\frac{\Phi.\mathbf{programs}(\Phi.\mathbf{specified})(\mathit{IDENT}) = \langle \mathbf{m}, [t_1, \ldots, t_n], t_{n+1} \rangle \quad \Phi \vdash \mathit{traceExpr}_1 \Longrightarrow t_1 \quad \ldots \quad \Phi \vdash \mathit{traceExpr}_n \Longrightarrow t_n \quad \Phi \vdash \mathit{traceExpr}_{result} \Longrightarrow t_{n+1}}{\begin{array}{c}\Phi \vdash \mathit{application} \Longrightarrow \mathbf{m\ complete} \\ \Phi \vdash \mathit{application} \Longrightarrow \mathbf{m\ with\ result} \\ \Phi \vdash \mathit{application} \Longrightarrow \Phi.\mathbf{specified}\end{array}}$$

B.8 Wywołania obcych programów dostępu

$$\boxed{\mathit{foreignApplication} ::= \mathit{IDENT}_{type}\ \texttt{::}\ \mathit{IDENT}_{prg}\ \texttt{(}\ \mathit{traceExpr}_1, \ldots, \mathit{traceExpr}_n\ \texttt{)}}$$

$$\frac{\mathit{IDENT}_{type} \neq \Phi.\mathbf{specified} \quad \Phi.\mathbf{programs}(\mathit{IDENT}_{type})(\mathit{IDENT}_{prg}) = \langle \mathbf{o}, [t_1, \ldots, t_n], t_{n+1} \rangle \quad \Phi \vdash \mathit{traceExpr}_1 \Longrightarrow t_1 \quad \ldots \quad \Phi \vdash \mathit{traceExpr}_n \Longrightarrow t_n}{\Phi \vdash \mathit{foreignApplication} \Longrightarrow t_{n+1}}$$

$$\boxed{\begin{array}{lcll}\mathit{foreignApplication} & ::= & \mathit{IDENT}_{type}\ \texttt{::}\ \mathit{IDENT}_{prg}\ \texttt{(} & \mathit{traceExpr}_0\ \texttt{;} \\ & & & \mathit{traceExpr}_1, \ldots, \mathit{traceExpr}_n\ \texttt{)}\end{array}}$$

$$\frac{\mathit{IDENT}_{type} \neq \Phi.\mathbf{specified} \quad \Phi.\mathbf{programs}(\mathit{IDENT}_{type})(\mathit{IDENT}_{prg}) = \langle \mathbf{m}, [t_1, \ldots, t_n] \rangle \quad \Phi \vdash \mathit{traceExpr}_0 \Longrightarrow \mathit{IDENT}_{type} \quad \Phi \vdash \mathit{traceExpr}_1 \Longrightarrow t_1 \quad \ldots \quad \Phi \vdash \mathit{traceExpr}_n \Longrightarrow t_n}{\Phi \vdash \mathit{foreignApplication} \Longrightarrow \mathit{IDENT}_{type}}$$

$$\boxed{\begin{array}{lcll}\mathit{foreignApplication} & ::= & \mathit{IDENT}_{type}\ \texttt{::}\ \mathit{IDENT}_{prg}\ \texttt{(} & \mathit{traceExpr}_0\ \texttt{;} \\ & & & \mathit{traceExpr}_1, \ldots, \mathit{traceExpr}_n \\ & & \texttt{)} & \texttt{\^{}}\ \mathit{traceExpr}_{n+1}\end{array}}$$

$$\frac{\mathit{IDENT}_{type} \neq \Phi.\mathbf{specified} \quad \Phi.\mathbf{programs}(\mathit{IDENT}_{type})(\mathit{IDENT}_{prg}) = \langle \mathbf{m}, [t_1, \ldots, t_n], t_{n+1} \rangle \quad \Phi \vdash \mathit{traceExpr}_0 \Longrightarrow \mathit{IDENT}_{type} \quad \Phi \vdash \mathit{traceExpr}_1 \Longrightarrow t_1 \quad \ldots \quad \Phi \vdash \mathit{traceExpr}_{n+1} \Longrightarrow t_{n+1}}{\Phi \vdash \mathit{foreignApplication} \Longrightarrow \mathit{IDENT}_{type}}$$

B.9 Wyrażenia sygnałowe

$\boxed{\mathit{signalExpr} ::= \texttt{[}\ \mathit{application}_1, \ldots, \mathit{application}_n\ \texttt{]}}$

$$\frac{\Phi \vdash \mathit{application}_1 \Longrightarrow \mathbf{s} \quad \ldots \quad \Phi \vdash \mathit{application}_n \Longrightarrow \mathbf{s}}{\Phi \vdash \mathit{signalExpr} \Longrightarrow \mathbf{OK}}$$

$\boxed{\mathit{signalExpr} ::= \mathit{signalTable}}$

$$\frac{\Phi \vdash \mathit{signalTable} \Longrightarrow \mathbf{OK}}{\Phi \vdash \mathit{signalExpr} \Longrightarrow \mathbf{OK}}$$

B.10 Wyrażenia tropowe

$\boxed{\mathit{traceExpr} ::= \mathit{IDENT}}$

$$\frac{\Phi.\mathbf{variables}(\mathit{IDENT}) = \langle \mathit{type}, 0 \rangle}{\Phi \vdash \mathit{traceExpr} \Longrightarrow \mathit{type}}$$

$\boxed{\mathit{traceExpr} ::= \mathit{IDENT}\ \texttt{[}\ \mathit{traceExpr}_1, \ldots, \mathit{traceExpr}_n\ \texttt{]}}$

$$\frac{\begin{array}{c}\Phi.\mathbf{variables}(\mathit{IDENT}) = \langle \mathit{type}, n \rangle \\ \Phi \vdash \mathit{traceExpr}_1 \Longrightarrow \text{“int”} \quad \ldots \quad \Phi \vdash \mathit{traceExpr}_n \Longrightarrow \text{“int”}\end{array}}{\Phi \vdash \mathit{traceExpr} \Longrightarrow \mathit{type}}$$

$\boxed{\mathit{traceExpr} ::= \mathit{application}}$

$$\frac{\Phi \vdash \mathit{application} \Longrightarrow \mathit{type}}{\Phi \vdash \mathit{traceExpr} \Longrightarrow \mathit{type}}$$

$\boxed{\mathit{traceExpr} ::= \mathit{foreignApplication}}$

$$\frac{\Phi \vdash \mathit{foreignApplication} \Longrightarrow \mathit{type}}{\Phi \vdash \mathit{traceExpr} \Longrightarrow \mathit{type}}$$

$\boxed{traceExpr ::= INTEGER}$

$$\frac{}{\Phi \vdash traceExpr \Longrightarrow \text{“int”}}$$

$\boxed{traceExpr ::= \texttt{\_}}$

$$\frac{}{\Phi \vdash traceExpr \Longrightarrow \Phi.\textbf{specified}}$$

$\boxed{traceExpr ::= IDENT\ \texttt{::}\ \texttt{\_}}$

$$\frac{IDENT \neq \Phi.\textbf{specified} \quad \Phi.\textbf{programs}(IDENT) \neq \textbf{error}}{\Phi \vdash traceExpr \Longrightarrow IDENT}$$

$\boxed{traceExpr ::= traceExpr_{left}\ \texttt{.}\ traceExpr_{right}}$

$$\frac{\Phi \vdash traceExpr_{left} \Longrightarrow \Phi.\textbf{specified} \quad \Phi \vdash traceExpr_{right} \Longrightarrow \Phi.\textbf{specified}}{\Phi \vdash traceExpr \Longrightarrow \Phi.\textbf{specified}}$$

$\boxed{traceExpr ::= \texttt{(}\ traceExpr_{inside}\ \texttt{)}}$

$$\frac{\Phi \vdash traceExpr_{inside} \Longrightarrow type}{\Phi \vdash traceExpr \Longrightarrow type}$$

$\boxed{traceExpr ::= \texttt{[}\ traceExpr_{inside}\ \texttt{]}\ \texttt{(}\ IDENT\ \texttt{=}\ traceExpr_{begin}\ \texttt{...}\ traceExpr_{end}\ \texttt{)}}$

$$\frac{\Phi \vdash traceExpr_{begin} \Longrightarrow \text{“int”} \qquad \Phi \vdash traceExpr_{end} \Longrightarrow \text{“int”} \qquad \Phi[\textbf{variables}(IDENT := \langle \text{“int”}, 0 \rangle] \vdash traceExpr_{inside} \Longrightarrow \Phi.\textbf{specified}}{\Phi \vdash traceExpr \Longrightarrow \Phi.\textbf{specified}}$$

$\boxed{\mathit{traceExpr} ::= \mathit{traceExpr}_{left}\ \texttt{+}\ \mathit{traceExpr}_{right}}$

$$\frac{\begin{array}{l}\Phi \vdash \mathit{traceExpr}_{left} \Longrightarrow \mathit{type}_{left} \qquad\qquad \Phi \vdash \mathit{traceExpr}_{right} \Longrightarrow \mathit{type}_{right} \\ \text{istnieje dokładnie jeden } \mathit{type}_{defines}\text{, taki że dla pewnego } \mathit{type}_{result} \text{ zachodzi} \\ \quad (i)\ \ \mathit{type}_{defines} \neq \Phi.\mathbf{specified} \\ \quad (ii)\ \ \Phi.\mathbf{programs}(\mathit{type}_{defines})(\text{“+”}) = \langle \mathbf{o}, [\mathit{type}_{left}, \mathit{type}_{right}], \mathit{type}_{result}\rangle\end{array}}{\Phi \vdash \mathit{traceExpr} \Longrightarrow \mathit{type}_{result}}$$

Uwaga: Analogiczne reguły obowiązują dla dwuargumentowych operatorów “-”, “*” i “/”.

$\boxed{\mathit{traceExpr} ::= \texttt{-}\ \mathit{traceExpr}_{arg}}$

$$\frac{\begin{array}{c}\Phi \vdash \mathit{traceExpr}_{arg} \Longrightarrow \mathit{type}_{arg} \\ \text{istnieje dokładnie jeden } \mathit{type}_{defines}\text{, taki że dla pewnego } \mathit{type}_{result} \text{ zachodzi} \\ \begin{array}{l}(i)\ \ \mathit{type}_{defines} \neq \Phi.\mathbf{specified} \\ (ii)\ \ \Phi.\mathbf{programs}(\mathit{type}_{defines})(\text{“unary -”}) = \langle \mathbf{o}, [\mathit{type}_{arg}], \mathit{type}_{result}\rangle\end{array}\end{array}}{\Phi \vdash \mathit{traceExpr} \Longrightarrow \mathit{type}_{result}}$$

$\boxed{\mathit{traceExpr} ::= \mathit{traceTable}}$

$$\frac{\Phi \vdash \mathit{traceTable} \Longrightarrow \mathit{type}}{\Phi \vdash \mathit{traceExpr} \Longrightarrow \mathit{type}}$$

B.11 Deklaracje zmiennych

$\boxed{\mathit{varDeclarationList} ::= \mathit{varDeclaration}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{varDeclaration}_n}$

$$\frac{\Phi_0 \vdash \mathit{varDeclaration}_1 \Longrightarrow \Phi_1 \quad \ldots \quad \Phi_{n-1} \vdash \mathit{varDeclaration}_n \Longrightarrow \Phi_n}{\Phi_0 \vdash \mathit{varDeclarationList} \Longrightarrow \Phi_n}$$

$$\boxed{\textit{varDeclaration} ::= \textit{variableDeclaration}_1, \ldots, \textit{variableDeclaration}_n \ : \ \textit{IDENT}}$$

$$\frac{\begin{array}{c}\Phi_0.\textbf{programs}(\textit{IDENT}) \neq \textbf{error} \\ \Phi_0, \textit{IDENT} \vdash \textit{variableDeclaration}_1 \Longrightarrow \Phi_1 \\ \ldots \\ \Phi_{n-1}, \textit{IDENT} \vdash \textit{variableDeclaration}_n \Longrightarrow \Phi_n\end{array}}{\Phi_0 \vdash \textit{varDeclaration} \Longrightarrow \Phi_n}$$

$$\frac{\begin{array}{c}\textit{IDENT} = \text{“traces”} \\ \Phi_0, \Phi.\textbf{specified} \vdash \textit{variableDeclaration}_1 \Longrightarrow \Phi_1 \\ \ldots \\ \Phi_{n-1}, \Phi.\textbf{specified} \vdash \textit{variableDeclaration}_n \Longrightarrow \Phi_n\end{array}}{\Phi_0 \vdash \textit{varDeclaration} \Longrightarrow \Phi_n}$$

$$\boxed{\textit{variableDeclaration} ::= \textit{IDENT}}$$

$$\frac{\Phi.\textbf{variables}(\textit{IDENT}) = \textbf{error}}{\Phi, \textit{type} \vdash \textit{variableDeclaration} \Longrightarrow \Phi[\textbf{variables}(\textit{IDENT}) := \langle \textit{type}, 0 \rangle]}$$

$$\boxed{\begin{array}{lcll}\textit{variableDeclaration} & ::= & \textit{IDENT} \ \texttt{[} & \textit{traceExpr}_1^{\textit{begin}}, \ldots, \textit{traceExpr}_n^{\textit{begin}} \\ & & \ldots & \\ & & & \textit{traceExpr}_1^{\textit{end}}, \ldots, \textit{traceExpr}_n^{\textit{end}} \quad \texttt{]}\end{array}}$$

$$\frac{\begin{array}{c}\Phi.\textbf{variables}(\textit{IDENT}) = \textbf{error} \\ \Phi \vdash \textit{traceExpr}_1^{\textit{begin}} \Longrightarrow \text{“int”} \quad \ldots \quad \Phi \vdash \textit{traceExpr}_n^{\textit{begin}} \Longrightarrow \text{“int”} \\ \Phi \vdash \textit{traceExpr}_1^{\textit{end}} \Longrightarrow \text{‘int”} \quad \ldots \quad \Phi \vdash \textit{traceExpr}_n^{\textit{end}} \Longrightarrow \text{‘int”}\end{array}}{\Phi, \textit{type} \vdash \textit{variableDeclaration} \Longrightarrow \Phi[\textbf{variables}(\textit{IDENT}) := \langle \textit{type}, n \rangle]}$$

B.12 Wyrażenia logiczne

$$\boxed{\textit{logExpr} ::= \textit{IDENT}}$$

$$\frac{\Phi.\textbf{variables}(\textit{IDENT}) = \langle \text{“bool”}, 0 \rangle}{\Phi \vdash \textit{logExpr} \Longrightarrow \textbf{OK}}$$

$\boxed{\mathit{logExpr} ::= \mathit{IDENT}\ \texttt{[}\ \mathit{traceExpr}_1, \ldots, \mathit{traceExpr}_n\ \texttt{]}}$

$$\frac{\Phi.\textbf{variables}(\mathit{IDENT}) = \langle\text{“bool”}, n\rangle \quad \Phi \vdash \mathit{traceExpr}_1 \Longrightarrow \text{“int”} \quad \ldots \quad \Phi \vdash \mathit{traceExpr}_n \Longrightarrow \text{“int”}}{\Phi \vdash \mathit{logExpr} \Longrightarrow \textbf{OK}}$$

$\boxed{\mathit{logExpr} ::= \mathit{application}}$

$$\frac{\Phi \vdash \mathit{application} \Longrightarrow \text{“bool”}}{\Phi \vdash \mathit{logExpr} \Longrightarrow \textbf{OK}}$$

$\boxed{\mathit{logExpr} ::= \mathit{foreignApplication}}$

$$\frac{\Phi \vdash \mathit{foreignApplication} \Longrightarrow \text{“bool”}}{\Phi \vdash \mathit{logExpr} \Longrightarrow \textbf{OK}}$$

$\boxed{\mathit{logExpr} ::= \texttt{true}}$

$$\frac{}{\Phi \vdash \mathit{logExpr} \Longrightarrow \textbf{OK}}$$

Uwaga: Analogiczna reguła obowiązuje dla symbolu “`false`”.

$\boxed{\mathit{logExpr} ::= \mathit{traceExpr}_{left}\ \texttt{=}\ \mathit{traceExpr}_{right}}$

$$\frac{\Phi \vdash \mathit{traceExpr}_{left} \Longrightarrow \mathit{type} \quad \Phi \vdash \mathit{traceExpr}_{right} \Longrightarrow \mathit{type}}{\Phi \vdash \mathit{logExpr} \Longrightarrow \textbf{OK}}$$

Uwaga: Analogiczna reguła obowiązuje dla operatora “`!=`”.

$\boxed{\mathit{logExpr} ::= \mathit{traceExpr}_{left}\ \texttt{<}\ \mathit{traceExpr}_{right}}$

$$\frac{\begin{array}{l}\Phi \vdash \mathit{traceExpr}_{left} \Longrightarrow \mathit{type}_{left} \qquad \Phi \vdash \mathit{traceExpr}_{right} \Longrightarrow \mathit{type}_{right} \\ \text{istnieje dokładnie jeden } \mathit{type}_{defines}\text{, taki że zachodzi} \\ \quad (i)\ \ \mathit{type}_{defines} \neq \Phi.\textbf{specified} \\ \quad (ii)\ \ \Phi.\textbf{programs}(\mathit{type}_{defines})(\text{“<”}) = \langle \textbf{o}, [\mathit{type}_{left}, \mathit{type}_{right}], \text{“bool”}\rangle\end{array}}{\Phi \vdash \mathit{logExpr} \Longrightarrow \textbf{OK}}$$

Uwaga: Analogiczne reguły obowiązują dla operatorów “`>`”, “`>=`” i “`<=`”.

$\boxed{logExpr ::= \texttt{(}\ logExpr_{inside}\ \texttt{)}}$

$$\frac{\Phi \vdash logExpr_{inside} \Longrightarrow \mathbf{OK}}{\Phi \vdash logExpr \Longrightarrow \mathbf{OK}}$$

$\boxed{logExpr ::= logExpr_{left}\ \texttt{=>}\ logExpr_{right}}$

$$\frac{\Phi \vdash logExpr_{left} \Longrightarrow \mathbf{OK} \quad \Phi \vdash logExpr_{right} \Longrightarrow \mathbf{OK}}{\Phi \vdash logExpr \Longrightarrow \mathbf{OK}}$$

Uwaga: Analogiczne reguły obowiązują dla spójników "`/\`", "`\/`" i "`<=>`".

$\boxed{logExpr ::= \texttt{\~{}}\ logExpr_{arg}}$

$$\frac{\Phi \vdash logExpr_{arg} \Longrightarrow \mathbf{OK}}{\Phi \vdash logExpr \Longrightarrow \mathbf{OK}}$$

$\boxed{logExpr ::= \texttt{forall}\ varDeclarationList\ constraint\ \texttt{[}\ logExpr_{inside}\ \texttt{]}}$

$$\frac{\Phi \vdash varDeclarationList \Longrightarrow \Psi \quad \Psi \vdash constraint \Longrightarrow \mathbf{OK} \quad \Psi \vdash logExpr_{inside} \Longrightarrow \mathbf{OK}}{\Phi \vdash logExpr \Longrightarrow \mathbf{OK}}$$

Uwaga: Analogiczne reguły obowiązują dla kwantyfikatorów "`exists`" i "`exists !`".

$\boxed{logExpr ::= logTable}$

$$\frac{\Phi \vdash logTable \Longrightarrow \mathbf{OK}}{\Phi \vdash logExpr \Longrightarrow \mathbf{OK}}$$

$\boxed{constraint ::= \varepsilon}$

$$\frac{}{\Phi \vdash constraint \Longrightarrow \mathbf{OK}}$$

constraint ::= `(` *logExpr* `)`

$$\frac{\Phi \vdash \mathit{logExpr} \Longrightarrow \mathbf{OK}}{\Phi \vdash \mathit{constraint} \Longrightarrow \mathbf{OK}}$$

B.13 Jednowymiarowe normalne tabele funkcyjne

traceTable ::= `table` $\mathit{traceRow}_1 \ldots \mathit{traceRow}_n$ `end`

$$\frac{\Phi \vdash \mathit{traceRow}_1 \Longrightarrow \mathit{type} \quad \ldots \quad \Phi \vdash \mathit{traceRow}_n \Longrightarrow \mathit{type}}{\Phi \vdash \mathit{traceTable} \Longrightarrow \mathit{type}}$$

traceRow ::= `#` *logExpr* `#` *traceExpr* `#`

$$\frac{\Phi \vdash \mathit{logExpr} \Longrightarrow \mathbf{OK} \quad \Phi \vdash \mathit{traceExpr} \Longrightarrow \mathit{type}}{\Phi \vdash \mathit{traceRow} \Longrightarrow \mathit{type}}$$

logTable ::= `table` $\mathit{logRow}_1 \ldots \mathit{logRow}_n$ `end`

$$\frac{\Phi \vdash \mathit{logRow}_1 \Longrightarrow \mathbf{OK} \quad \ldots \quad \Phi \vdash \mathit{logRow}_n \Longrightarrow \mathbf{OK}}{\Phi \vdash \mathit{logTable} \Longrightarrow \mathbf{OK}}$$

logRow ::= `#` $\mathit{logExpr}_{\mathit{condition}}$ `#` $\mathit{logExpr}_{\mathit{value}}$ `#`

$$\frac{\Phi \vdash \mathit{logExpr}_{\mathit{condition}} \Longrightarrow \mathbf{OK} \quad \Phi \vdash \mathit{logExpr}_{\mathit{value}} \Longrightarrow \mathbf{OK}}{\Phi \vdash \mathit{logRow} \Longrightarrow \mathbf{OK}}$$

signalTable ::= `table` $\mathit{signalRow}_1 \ldots \mathit{signalRow}_n$`end`

$$\frac{\Phi \vdash \mathit{signalRow}_1 \Longrightarrow \mathbf{OK} \quad \ldots \quad \Phi \vdash \mathit{signalRow}_n \Longrightarrow \mathbf{OK}}{\Phi \vdash \mathit{signalTable} \Longrightarrow \mathbf{OK}}$$

signalRow ::= `#` *logExpr* `#` *signalExpr* `#`

$$\frac{\Phi \vdash \mathit{logExpr} \Longrightarrow \mathbf{OK} \quad \Phi \vdash \mathit{signalExpr} \Longrightarrow \mathbf{OK}}{\Phi \vdash \mathit{signalRow} \Longrightarrow \mathbf{OK}}$$

Dodatek C

Reguły wyliczania wartości wyrażeń

Poniższe reguły odnoszą się jedynie do specyfikacji, które są statycznie poprawne (por. p. 7.3.2). Zakładamy, że środowisko Φ jest statyczną semantyką rozważanej specyfikacji, zaś dla każdego typu obcego *type* $\mathcal{S}_{type}$ oznacza denotację specyfikacji tego typu.

C.1 Wywołania specyfikowanych programów dostępu i funkcji pomocniczych

$$\boxed{\mathit{application} ::= \mathit{IDENT}\ (\ \mathit{traceExpr}_1, \ldots, \mathit{traceExpr}_n\ )}$$

$$\frac{\begin{array}{c}\text{nie istnieje } l, r \text{ takie, że } \Phi.\mathbf{functions}(\mathit{IDENT}) = \langle l, r\rangle^1 \\ \mathcal{W} \vdash \mathit{traceExpr}_1 \Longrightarrow v_1 \quad \ldots \quad \mathcal{W} \vdash \mathit{traceExpr}_n \Longrightarrow v_n\end{array}}{\mathcal{W} \vdash \mathit{application} \Longrightarrow \mathit{IDENT}(v_1, \ldots, v_n)}$$

$$\frac{\begin{array}{c}\Phi.\mathbf{functions}(\mathit{IDENT}) = \langle [t_1, \ldots, t_n], r\rangle \\ \mathcal{W} \vdash \mathit{traceExpr}_1 \Longrightarrow v_1 \quad \ldots \quad \mathcal{W} \vdash \mathit{traceExpr}_n \Longrightarrow v_n \\ \langle v_1, \ldots, v_n\rangle \text{ należy do dziedziny funkcji } \mathit{IDENT} \\ v \text{ jest wartością funkcji } \mathit{IDENT} \text{ dla argumentów } \langle v_1, \ldots, v_n\rangle\end{array}}{\mathcal{W} \vdash \mathit{application} \Longrightarrow v}$$

$\langle v_1, \ldots, v_n\rangle$ **należy do dziedziny funkcji** *IDENT* jedynie jeśli:

1. dla każdego $i = 1, \ldots, n$ jeśli $t_i = \Phi.\mathbf{specified}$, to da się wyprowadzić

   $$\mathcal{W}[\langle T, [\ ]\rangle := v_i] \vdash \mathit{canonical}(T) \Longrightarrow \mathbf{true}$$

2. oraz jeśli w definicji tej funkcji występuje ograniczenie dziedziny (*domainRestriction*), musi dać się wyprowadzić

   $$\mathcal{E} \vdash \mathit{logExpr} \Longrightarrow \mathbf{true}$$

[1] Tzn. *IDENT* nie jest nazwą funkcji pomocniczej.

przy czym *logExpr* pochodzi z pierwszego w specyfikacji *domainRestriction*:

`legal (` *IDENT* `(`$traceExpr_1^{constr}, \ldots, traceExpr_n^{constr}$`)` `<=>` *logExpr*

takiego, że istnieje wartościowanie $\mathcal{F}$, przy którym dla każdego $i = 1, \ldots, n$ zachodzi

$$\mathcal{F} \vdash traceExpr_i^{constr} \Longrightarrow v_i$$

$\mathcal{E}$ jest najmniejszym wg relacji "$\preceq$" wartościowaniem spełniającym ten warunek.

v **jest wartością funkcji** *IDENT* **dla argumentów** $\langle v_1, \ldots, v_n \rangle$ jedynie jeśli da się wyprowadzić

$$\mathcal{E} \vdash logExpr \Longrightarrow v \qquad \text{albo} \qquad \mathcal{E} \vdash traceExpr_{result} \Longrightarrow v$$

przy czym *logExpr* (albo odpowiednio $traceExpr_{result}$) pochodzi z pierwszego w specyfikacji *valueDefinition*:

IDENT `(`$traceExpr_1^{def}, \ldots,\ traceExpr_n^{def}$`)` `<=>` *logExpr*

albo odpowiednio:

IDENT `(`$traceExpr_1^{def}, \ldots,\ traceExpr_n^{def}$`)` `=` $logExpr_{result}$

takiego, że istnieje wartościowanie $\mathcal{F}$, przy którym dla każdego $i = 1, \ldots, n$ zachodzi

$$\mathcal{F} \vdash traceExpr_i^{def} \Longrightarrow v_i$$

$\mathcal{E}$ jest najmniejszym wg relacji "$\preceq$" wartościowaniem spełniającym ten warunek.

application ::= *IDENT* `(` $traceExpr_1, \ldots, traceExpr_n$ `)` `^` $traceExpr_{result}$

$$\frac{\mathcal{W} \vdash traceExpr_1 \Longrightarrow v_1 \quad \ldots \quad \mathcal{W} \vdash traceExpr_n \Longrightarrow v_n \quad \mathcal{W} \vdash traceExpr_{result} \Longrightarrow v_{result}}{\mathcal{W} \vdash application \Longrightarrow IDENT(v_1, \ldots, v_n)\hat{}v_{result}}$$

C.2 Wywołania obcych programów dostępu

foreignApplication ::= $IDENT_{type}$ `::` $IDENT_{prg}$ `(` $traceExpr_1, \ldots, traceExpr_n$ `)`

$$\frac{\begin{array}{c}\mathcal{W} \vdash traceExpr_1 \Longrightarrow v_1 \quad \ldots \quad \mathcal{W} \vdash traceExpr_n \Longrightarrow v_n \\ IDENT_{prg}(v_1, \ldots, v_n) \in \mathcal{S}_{IDENT_{type}}.\mathbf{legal} \\ r = \mathcal{S}_{IDENT_{type}}.\mathbf{result}(IDENT_{prg}(v_1, \ldots, v_n))\end{array}}{\mathcal{W} \vdash foreignApplication \Longrightarrow r}$$

$$\boxed{\begin{array}{lcll} \textit{foreignApplication} & ::= & \textit{IDENT}_{type}\ \texttt{::}\ \textit{IDENT}_{prg}\ \texttt{(} & \textit{traceExpr}_0\ \texttt{;} \\ & & & \textit{traceExpr}_1,\ \ldots,\ \textit{traceExpr}_n\ \texttt{)} \end{array}}$$

$$\frac{\begin{array}{c} \mathcal{W} \vdash \textit{traceExpr}_0 \Longrightarrow v_0 \quad \mathcal{W} \vdash \textit{traceExpr}_1 \Longrightarrow v_1 \quad \ldots \quad \mathcal{W} \vdash \textit{traceExpr}_n \Longrightarrow v_n \\ \langle v_0, \textit{IDENT}_{prg}(v_1, \ldots, v_n)\rangle \in \mathcal{S}_{\textit{IDENT}_{type}}.\textbf{legal} \\ r = \mathcal{S}_{\textit{IDENT}_{type}}.\textbf{transition}(\langle v_0, \textit{IDENT}_{prg}(v_1, \ldots, v_n)\rangle) \end{array}}{\mathcal{W} \vdash \textit{foreignApplication} \Longrightarrow r}$$

$$\boxed{\begin{array}{lcll} \textit{foreignApplication} & ::= & \textit{IDENT}_{type}\ \texttt{::}\ \textit{IDENT}_{prg}\ \texttt{(} & \textit{traceExpr}_0\ \texttt{;} \\ & & & \textit{traceExpr}_1,\ \ldots,\ \textit{traceExpr}_n \\ & & \texttt{)} & \texttt{\^{}}\ \textit{traceExpr}_{n+1} \end{array}}$$

$$\frac{\begin{array}{c} \mathcal{W} \vdash \textit{traceExpr}_0 \Longrightarrow v_0 \quad \mathcal{W} \vdash \textit{traceExpr}_1 \Longrightarrow v_1 \quad \ldots \quad \mathcal{W} \vdash \textit{traceExpr}_{n+1} \Longrightarrow v_{n+1} \\ \langle v_0, \textit{IDENT}_{prg}(v_1, \ldots, v_n)\texttt{\^{}}v_{n+1}\rangle \in \mathcal{S}_{\textit{IDENT}_{type}}.\textbf{feasible} \\ r = \mathcal{S}_{\textit{IDENT}_{type}}.\textbf{transition}(\langle v_0, \textit{IDENT}_{prg}(v_1, \ldots, v_n)\texttt{\^{}}v_{n+1}\rangle) \end{array}}{\mathcal{W} \vdash \textit{foreignApplication} \Longrightarrow r}$$

C.3 Wyrażenia sygnałowe

$$\boxed{\textit{signalExpr} ::= \texttt{[}\ \textit{application}_1,\ \ldots,\ \textit{application}_n\ \texttt{]}}$$

$$\frac{\mathcal{W} \vdash \textit{application}_1 \Longrightarrow s_1 \quad \ldots \quad \mathcal{W} \vdash \textit{application}_n \Longrightarrow s_n}{\mathcal{W} \vdash \textit{signalExpr} \Longrightarrow [s_1, \ldots, s_n]}$$

$$\boxed{\textit{signalExpr} ::= \textit{signalTable}}$$

$$\frac{\mathcal{W} \vdash \textit{signalTable} \Longrightarrow ls}{\mathcal{W} \vdash \textit{signalExpr} \Longrightarrow ls}$$

C.4 Wyrażenia tropowe

$$\boxed{\textit{traceExpr} ::= \textit{IDENT}}$$

$$\frac{\mathcal{W}(\textit{IDENT}, [\,]) = v \quad v \neq \textbf{error}}{\mathcal{W} \vdash \textit{traceExpr} \Longrightarrow v}$$

$\boxed{traceExpr ::= IDENT\ \texttt{[}\ traceExpr_1, \ldots, traceExpr_n\ \texttt{]}}$

$$\frac{\mathcal{W} \vdash traceExpr_1 \Longrightarrow v_1 \quad \ldots \quad \mathcal{W} \vdash traceExpr_n \Longrightarrow v_n \qquad \mathcal{W}(IDENT, [v_1, \ldots, v_n]) = v \qquad v \neq \textbf{error}}{\mathcal{W} \vdash traceExpr \Longrightarrow v}$$

$\boxed{traceExpr ::= application}$

$$\frac{\mathcal{W} \vdash application \Longrightarrow v}{\mathcal{W} \vdash traceExpr \Longrightarrow v}$$

$\boxed{traceExpr ::= foreignApplication}$

$$\frac{\mathcal{W} \vdash foreignApplication \Longrightarrow v}{\mathcal{W} \vdash traceExpr \Longrightarrow v}$$

$\boxed{traceExpr ::= INTEGER}$

$$\frac{}{\mathcal{W} \vdash traceExpr \Longrightarrow INTEGER}$$

$\boxed{traceExpr ::= \_}$

$$\frac{}{\mathcal{W} \vdash traceExpr \Longrightarrow [\,]}$$

$\boxed{traceExpr ::= IDENT\ ::\ \_}$

$$\frac{v = \mathcal{S}_{IDENT}.\textbf{initial}}{\mathcal{W} \vdash traceExpr \Longrightarrow v}$$

$\boxed{traceExpr ::= traceExpr_{left}\ .\ traceExpr_{right}}$

$$\frac{\mathcal{W} \vdash traceExpr_{left} \Longrightarrow v_{left} \quad \mathcal{W} \vdash traceExpr_{right} \Longrightarrow v_{right}}{\mathcal{W} \vdash traceExpr \Longrightarrow v_{left} \bullet v_{right}}$$

$\boxed{traceExpr ::= \texttt{(}\ traceExpr_{inside}\ \texttt{)}}$

$$\frac{\mathcal{W} \vdash traceExpr_{inside} \Longrightarrow v}{\mathcal{W} \vdash traceExpr \Longrightarrow v}$$

$\boxed{traceExpr ::= \texttt{[}\ traceExpr_{inside}\ \texttt{]}\ \texttt{(}\ IDENT\ \texttt{=}\ traceExpr_{begin}\ \texttt{...}\ traceExpr_{end}\ \texttt{)}}$

$$\frac{\mathcal{W} \vdash traceExpr_{begin} \Longrightarrow b \quad \mathcal{W} \vdash traceExpr_{end} \Longrightarrow e \quad b = e + 1}{\mathcal{W} \vdash traceExpr \Longrightarrow [\,]}$$

$$\frac{\begin{array}{c}\mathcal{W} \vdash traceExpr_{begin} \Longrightarrow b \qquad \mathcal{W} \vdash traceExpr_{end} \Longrightarrow e \qquad b \leq e \\ \mathcal{W}[\langle IDENT, [\,]\rangle := b] \vdash traceExpr_{inside} \Longrightarrow v \\ \mathcal{W} \vdash \texttt{[}\ traceExpr_{inside}\ \texttt{]}\ \texttt{(}\ IDENT\ \texttt{=}\ \texttt{(}\ traceExpr_{begin}\ \texttt{+}\ \texttt{1)}\ \texttt{...}\ traceExpr_{end}\ \texttt{)} \Longrightarrow w\end{array}}{\mathcal{W} \vdash traceExpr \Longrightarrow v \bullet w}$$

$\boxed{traceExpr ::= traceExpr_{left}\ \texttt{+}\ traceExpr_{right}}$

$$\frac{\begin{array}{c}\mathcal{W} \vdash traceExpr_{left} \Longrightarrow v_{left} \qquad \mathcal{W} \vdash traceExpr_{right} \Longrightarrow v_{right} \\ \text{istnieje dokładnie jeden } type\text{, taki, że dla pewnego } r \text{ zachodzi} \\ r = \mathcal{S}_{type}.\mathbf{result}(\text{“+”}(v_{left}, v_{right}))\end{array}}{\Phi \vdash traceExpr \Longrightarrow r}$$

Uwaga: Analogiczne reguły obowiązują dla dwuargumentowych operatorów “-”, “*” i “/”.

$\boxed{traceExpr ::= \texttt{-}\ traceExpr_{arg}}$

$$\frac{\begin{array}{c}\mathcal{W} \vdash traceExpr_{arg} \Longrightarrow v_{arg} \\ \text{istnieje dokładnie jeden } type\text{, taki, że dla pewnego } r \text{ zachodzi} \\ r = \mathcal{S}_{type}.\mathbf{result}(\text{“unary -”}(v_{arg}))\end{array}}{\Phi \vdash traceExpr \Longrightarrow r}$$

$\boxed{traceExpr ::= traceTable}$

$$\frac{\mathcal{W} \vdash traceTable \Longrightarrow v}{\mathcal{W} \vdash traceExpr \Longrightarrow v}$$

C.5 Deklaracje zmiennych

$\boxed{\textit{varDeclarationList} ::= \textit{varDeclaration}_1 \ ; \ \ldots \ ; \ \textit{varDeclaration}_n}$

$$\frac{\mathcal{W}_0, \mathcal{W}_1 \vdash \textit{varDeclaration}_1 \Longrightarrow \textbf{OK} \quad \ldots \quad \mathcal{W}_{n-1}, \mathcal{W}_n \vdash \textit{varDeclaration}_n \Longrightarrow \textbf{OK}}{\mathcal{W}_0, \mathcal{W}_n \vdash \textit{varDeclarationList} \Longrightarrow \textbf{OK}}$$

$\boxed{\textit{varDeclaration} ::= \textit{variableDeclaration}_1, \ldots, \textit{variableDeclaration}_n \ : \ \textit{IDENT}}$

$$\frac{\mathcal{W}_0, \mathcal{W}_1 \vdash \textit{variableDeclaration}_1 \Longrightarrow \textbf{OK} \quad \ldots \quad \mathcal{W}_{n-1}, \mathcal{W}_n \vdash \textit{variableDeclaration}_n \Longrightarrow \textbf{OK}}{\textit{IDENT}, \mathcal{W}_0, \mathcal{W}_n \vdash \textit{varDeclaration} \Longrightarrow \textbf{OK}}$$

$\boxed{\textit{variableDeclaration} ::= \textit{IDENT}}$

$$\frac{\begin{array}{l}\text{dla każdego } id \text{ i } l \text{ takich, że } id \neq \textit{IDENT} \text{ lub } l \neq [\,] \text{ zachodzi } \mathcal{W}(k,l) = \mathcal{U}(k,l) \\ \text{jeśli } \textit{type} \notin \{\text{“traces”}, \Phi.\textbf{specified}\}, \text{ to } \mathcal{U}(\textit{IDENT}, [\,]) \in \mathcal{S}_{\textit{type}}.\textbf{states} \\ \text{jeśli } \textit{type} \in \{\text{“traces”}, \Phi.\textbf{specified}\}, \text{ to } \mathcal{U}(\textit{IDENT}, [\,]) \in \textbf{Traces} \\ \text{jeśli } \textit{type} = \Phi.\textbf{specified}, \text{ to } \mathcal{U}[\langle T, [\,]\rangle := \mathcal{U}(\textit{IDENT}, [\,])] \vdash \textit{canonical}(T) \Longrightarrow \textbf{true}\end{array}}{\textit{type}, \mathcal{W}, \mathcal{U} \vdash \textit{variableDeclaration} \Longrightarrow \textbf{OK}}$$

$$\boxed{\begin{array}{lcll}\textit{variableDeclaration} & ::= & \textit{IDENT} \ \texttt{[} & \textit{traceExpr}_1^{\textit{begin}}, \ldots, \textit{traceExpr}_n^{\textit{begin}} \\ & & \ldots & \\ & & & \textit{traceExpr}_1^{\textit{end}}, \ldots, \textit{traceExpr}_n^{\textit{end}} \quad \texttt{]}\end{array}}$$

$$\frac{\begin{array}{c}\mathcal{W} \vdash \textit{traceExpr}_1^{\textit{begin}} \Longrightarrow b_1 \quad \ldots \quad \mathcal{W} \vdash \textit{traceExpr}_n^{\textit{begin}} \Longrightarrow b_n \\ \mathcal{W} \vdash \textit{traceExpr}_1^{\textit{end}} \Longrightarrow e_1 \quad \ldots \quad \mathcal{W} \vdash \textit{traceExpr}_n^{\textit{end}} \Longrightarrow e_n \\ \text{dla każdych } id \text{ i } l \text{ takich, że } id \neq \textit{IDENT} \text{ zachodzi, } \mathcal{W}(k,l) = \mathcal{U}(k,l) \\ \text{dla każdej listy } l \text{ o długości innej niż } n, \text{ zachodzi } \mathcal{U}(\textit{IDENT}, l) = \textbf{error} \\ \text{dla każdej listy } [i_1, \ldots, i_n] \text{ takiej, że} \\ \qquad \text{dla co najmniej jednego } k = 1, \ldots, n \text{ zachodzi } i_k > e_k \text{ lub } i_k < b_k, \\ \text{spełnione jest} \quad \mathcal{U}(\textit{IDENT}, [i_1, \ldots, i_n]) = \textbf{error} \\ \text{dla każdej listy } l = [i_1, \ldots, i_n] \text{ takiej, że} \\ \qquad \text{dla każdego } k = 1, \ldots, n \text{ zachodzi } b_k \leq i_k \leq e_k, \\ \text{spełnione jest} \\ \qquad \text{jeśli } \textit{type} \notin \{\text{“traces”}, \Phi.\textbf{specified}\}, \text{ to } \mathcal{U}(\textit{IDENT}, l) \in \mathcal{S}_{\textit{type}}.\textbf{states} \\ \qquad \text{jeśli } \textit{type} \in \{\text{“traces”}, \Phi.\textbf{specified}\}, \text{ to } \mathcal{U}(\textit{IDENT}, l) \in \textbf{Traces} \\ \qquad \text{jeśli } \textit{type} = \Phi.\textbf{specified}, \text{ to} \\ \qquad\qquad \mathcal{U}[\langle T, [\,]\rangle := \mathcal{U}(\textit{IDENT}, l)] \vdash \textit{canonical}(T) \Longrightarrow \textbf{true}\end{array}}{\textit{type}, \mathcal{W}, \mathcal{U} \vdash \textit{variableDeclaration} \Longrightarrow \textbf{OK}}$$

C.6 Wyrażenia logiczne

$\boxed{\textit{logExpr} ::= \textit{IDENT}}$

$$\frac{\mathcal{W}(\textit{IDENT}, [\,]) = v \quad v \neq \textbf{error}}{\mathcal{W} \vdash \textit{logExpr} \Longrightarrow v}$$

$\boxed{\textit{logExpr} ::= \textit{IDENT}\ \texttt{[}\ \textit{traceExpr}_1, \ldots, \textit{traceExpr}_n\ \texttt{]}}$

$$\frac{\begin{array}{ll}\mathcal{W} \vdash \textit{traceExpr}_1 \Longrightarrow v_1 & \ldots \quad \mathcal{W} \vdash \textit{traceExpr}_n \Longrightarrow v_n \\ \mathcal{W}(\textit{IDENT}, [v_1, \ldots, v_n]) = v & v \neq \textbf{error}\end{array}}{\mathcal{W} \vdash \textit{logExpr} \Longrightarrow v}$$

$\boxed{\textit{logExpr} ::= \textit{application}}$

$$\frac{\mathcal{W} \vdash \textit{application} \Longrightarrow v}{\mathcal{W} \vdash \textit{logExpr} \Longrightarrow v}$$

$\boxed{\textit{logExpr} ::= \textit{foreignApplication}}$

$$\frac{\mathcal{W} \vdash \textit{foreignApplication} \Longrightarrow v}{\mathcal{W} \vdash \textit{logExpr} \Longrightarrow v}$$

$\boxed{\textit{logExpr} ::= \texttt{true}}$

$$\frac{}{\mathcal{W} \vdash \textit{logExpr} \Longrightarrow \textbf{true}}$$

Uwaga: Analogiczna reguła obowiązuje dla symbolu "`false`".

$\boxed{\textit{logExpr} ::= \textit{traceExpr}_{left}\ \texttt{=}\ \textit{traceExpr}_{right}}$

$$\frac{\mathcal{W} \vdash \textit{traceExpr}_{left} \Longrightarrow v_{left} \quad \mathcal{W} \vdash \textit{traceExpr}_{right} \Longrightarrow v_{right}}{\mathcal{W} \vdash \textit{logExpr} \Longrightarrow v_{left} = v_{right}}$$

$\boxed{logExpr ::= traceExpr_{left} \texttt{ != } traceExpr_{right}}$

$$\frac{\mathcal{W} \vdash traceExpr_{left} \Longrightarrow v_{left} \quad \mathcal{W} \vdash traceExpr_{right} \Longrightarrow v_{right}}{\mathcal{W} \vdash logExpr \Longrightarrow v_{left} \neq v_{right}}$$

$\boxed{logExpr ::= traceExpr_{left} \texttt{ < } traceExpr_{right}}$

$$\frac{\begin{array}{c}\mathcal{W} \vdash traceExpr_{left} \Longrightarrow v_{left} \qquad \mathcal{W} \vdash traceExpr_{right} \Longrightarrow v_{right} \\ \text{istnieje dokładnie jeden } type\text{, taki, że dla pewnego } r \text{ zachodzi} \\ r = \mathcal{S}_{type}.\textbf{result}(\text{“<”}(v_{left}, v_{right}))\end{array}}{\mathcal{W} \vdash logExpr \Longrightarrow r}$$

Uwaga: Analogiczne reguły obowiązują dla operatorów “>”, “>=” i “<=”.

$\boxed{logExpr ::= \texttt{(} logExpr_{inside} \texttt{)}}$

$$\frac{\mathcal{W} \vdash logExpr_{inside} \Longrightarrow v}{\mathcal{W} \vdash logExpr \Longrightarrow v}$$

$\boxed{logExpr ::= logExpr_{left} \texttt{ /\textbackslash } logExpr_{right}}$

$$\frac{\mathcal{W} \vdash logExpr_{left} \Longrightarrow \textbf{true} \quad \mathcal{W} \vdash logExpr_{right} \Longrightarrow \textbf{true}}{\mathcal{W} \vdash logExpr \Longrightarrow \textbf{true}}$$

$$\frac{\mathcal{W} \vdash logExpr_{left} \Longrightarrow \textbf{false} \quad \mathcal{W} \vdash logExpr_{right} \Longrightarrow v}{\mathcal{W} \vdash logExpr \Longrightarrow \textbf{false}}$$

$$\frac{\mathcal{W} \vdash logExpr_{left} \Longrightarrow v \quad \mathcal{W} \vdash logExpr_{right} \Longrightarrow \textbf{false}}{\mathcal{W} \vdash logExpr \Longrightarrow \textbf{false}}$$

$\boxed{logExpr ::= \texttt{~} logExpr_{arg}}$

$$\frac{\mathcal{W} \vdash logExpr_{arg} \Longrightarrow \textbf{true}}{\mathcal{W} \vdash logExpr \Longrightarrow \textbf{false}}$$

$$\frac{\mathcal{W} \vdash logExpr_{arg} \Longrightarrow \textbf{false}}{\mathcal{W} \vdash logExpr \Longrightarrow \textbf{true}}$$

$\boxed{logExpr ::= logExpr_{left} \texttt{ \textbackslash/ } logExpr_{right}}$

$$\frac{\mathcal{W} \vdash \texttt{~}(\texttt{~}logExpr_{left} \texttt{ /\textbackslash } \texttt{~}logExpr_{right}) \Longrightarrow v}{\mathcal{W} \vdash logExpr \Longrightarrow v}$$

$\boxed{logExpr ::= logExpr_{left} \texttt{ => } logExpr_{right}}$

$$\frac{\mathcal{W} \vdash \texttt{~}logExpr_{left} \texttt{ \textbackslash/ } logExpr_{right} \Longrightarrow v}{\mathcal{W} \vdash logExpr \Longrightarrow v}$$

$\boxed{logExpr ::= logExpr_{left} \texttt{ <=> } logExpr_{right}}$

$$\frac{\mathcal{W} \vdash logExpr_{left} \Longrightarrow v_{left} \quad \mathcal{W} \vdash logExpr_{right} \Longrightarrow v_{right}}{\mathcal{W} \vdash logExpr \Longrightarrow v_{left} = v_{right}}$$

$\boxed{logExpr ::= \texttt{forall } varDeclarationList\ constraint \texttt{ [} logExpr_{inside} \texttt{]}}$

$$\frac{\begin{array}{l}\text{dla każdego wartościowania } \mathcal{U} \text{ takiego, że} \\ \qquad (i) \quad \mathcal{W},\mathcal{U} \vdash varDeclarationList \Longrightarrow \mathbf{OK} \\ \qquad (ii) \quad \mathcal{U} \vdash constraint \Longrightarrow \mathbf{true} \\ \text{można wyprowadzić} \quad \mathcal{U} \vdash logExpr_{inside} \Longrightarrow \mathbf{true}\end{array}}{\mathcal{W} \vdash logExpr \Longrightarrow \mathbf{true}}$$

$$\frac{\begin{array}{l}\text{dla każdego wartościowania } \mathcal{U} \text{ takiego, że} \\ \qquad (i) \quad \mathcal{W},\mathcal{U} \vdash varDeclarationList \Longrightarrow \mathbf{OK} \\ \qquad (ii) \quad \mathcal{U} \vdash constraint \Longrightarrow \mathbf{true} \\ \text{można wyprowadzić} \quad \mathcal{U} \vdash logExpr_{inside} \Longrightarrow v \\ \text{dla co najmniej jednego takiego } \mathcal{U} \text{ zachodzi } \mathcal{U} \vdash logExpr_{inside} \Longrightarrow \mathbf{false}\end{array}}{\mathcal{W} \vdash logExpr \Longrightarrow \mathbf{false}}$$

$\boxed{logExpr ::= \texttt{exists } varDeclarationList\ constraint \texttt{ [} logExpr_{inside} \texttt{]}}$

$$\frac{\mathcal{W} \vdash \texttt{~ forall } varDeclarationList\ constraint \texttt{ [~}logExpr_{inside} \texttt{]} \Longrightarrow v}{\mathcal{W} \vdash logExpr \Longrightarrow v}$$

logExpr ::= `exists !` *varDeclarationList constraint* `[` $logExpr_{inside}$ `]`

$$\frac{\begin{array}{rl}\text{dla każdego wartościowania } \mathcal{U} \text{ takiego, że} & \\ (i) & \mathcal{W},\mathcal{U} \vdash varDeclarationList \Longrightarrow \mathbf{OK} \\ (ii) & \mathcal{U} \vdash constraint \Longrightarrow \mathbf{true} \\ \text{można wyprowadzić} & \mathcal{U} \vdash logExpr_{inside} \Longrightarrow v \\ \text{dla dokładnie jednego takiego } \mathcal{U} \text{ zachodzi} & \mathcal{U} \vdash logExpr_{inside} \Longrightarrow \mathbf{true}\end{array}}{\mathcal{W} \vdash logExpr \Longrightarrow \mathbf{true}}$$

$$\frac{\begin{array}{rl}\text{dla każdego wartościowania } \mathcal{U} \text{ takiego, że} & \\ (i) & \mathcal{W},\mathcal{U} \vdash varDeclarationList \Longrightarrow \mathbf{OK} \\ (ii) & \mathcal{U} \vdash constraint \Longrightarrow \mathbf{true} \\ \text{można wyprowadzić} & \mathcal{U} \vdash logExpr_{inside} \Longrightarrow v \\ \text{dla żadnego lub co najmniej dwóch takich } \mathcal{U} \text{ zachodzi} & \mathcal{U} \vdash logExpr_{inside} \Longrightarrow \mathbf{true}\end{array}}{\mathcal{W} \vdash logExpr \Longrightarrow \mathbf{false}}$$

logExpr ::= *logTable*

$$\frac{\mathcal{W} \vdash logTable \Longrightarrow v}{\mathcal{W} \vdash logExpr \Longrightarrow v}$$

constraint ::= ε

$$\frac{}{\mathcal{W} \vdash constraint \Longrightarrow \mathbf{true}}$$

constraint ::= `(` *logExpr* `)`

$$\frac{\mathcal{W} \vdash logExpr \Longrightarrow v}{\mathcal{W} \vdash constraint \Longrightarrow v}$$

C.7 Jednowymiarowe normalne tabele funkcyjne

traceTable ::= `table` $traceRow_1 \ldots traceRow_n$ `end`

$$\frac{\begin{array}{rl}i \in \{1,\ldots,n\} & \mathcal{W} \vdash traceRow_i \Longrightarrow v \\ \text{dla każdego } j = 1,\ldots,n, \text{ jeśli } j \neq i, \text{ to} & \mathcal{W} \vdash traceRow_j \Longrightarrow \mathbf{No}\end{array}}{\mathcal{W} \vdash traceTable \Longrightarrow v}$$

$\boxed{\textit{traceRow} ::= \texttt{\#}\ \textit{logExpr}\ \texttt{\#}\ \textit{traceExpr}\ \texttt{\#}}$

$$\frac{\mathcal{W} \vdash \textit{logExpr} \Longrightarrow \textbf{true} \quad \mathcal{W} \vdash \textit{traceExpr} \Longrightarrow v}{\mathcal{W} \vdash \textit{traceRow} \Longrightarrow v}$$

$$\frac{\mathcal{W} \vdash \textit{logExpr} \Longrightarrow \textbf{false}}{\mathcal{W} \vdash \textit{traceRow} \Longrightarrow \textbf{No}}$$

$\boxed{\textit{logTable} ::= \texttt{table}\ \textit{logRow}_1 \ldots \textit{logRow}_n\ \texttt{end}}$

$$\frac{\begin{array}{ll} i \in \{1, \ldots, n\} & \mathcal{W} \vdash \textit{logRow}_i \Longrightarrow v \\ \text{dla każdego } j = 1, \ldots, n, \text{ jeśli } j \neq i, \text{ to } \mathcal{W} \vdash \textit{logRow}_j \Longrightarrow \textbf{No} & \end{array}}{\mathcal{W} \vdash \textit{logTable} \Longrightarrow v}$$

$\boxed{\textit{logRow} ::= \texttt{\#}\ \textit{logExpr}_{\textit{condition}}\ \texttt{\#}\ \textit{logExpr}_{\textit{value}}\ \texttt{\#}}$

$$\frac{\mathcal{W} \vdash \textit{logExpr}_{\textit{condition}} \Longrightarrow \textbf{true} \quad \mathcal{W} \vdash \textit{logExpr}_{\textit{value}} \Longrightarrow v}{\mathcal{W} \vdash \textit{logRow} \Longrightarrow v}$$

$$\frac{\mathcal{W} \vdash \textit{logExpr}_{\textit{condition}} \Longrightarrow \textbf{false}}{\mathcal{W} \vdash \textit{logRow} \Longrightarrow \textbf{No}}$$

$\boxed{\textit{signalTable} ::= \texttt{table}\ \textit{signalRow}_1 \ldots \textit{signalRow}_n\texttt{end}}$

$$\frac{\begin{array}{ll} i \in \{1, \ldots, n\} & \mathcal{W} \vdash \textit{signalRow}_i \Longrightarrow v \\ \text{dla każdego } j = 1, \ldots, n, \text{ jeśli } j \neq i, \text{ to } \mathcal{W} \vdash \textit{signalRow}_j \Longrightarrow \textbf{No} & \end{array}}{\mathcal{W} \vdash \textit{signalTable} \Longrightarrow v}$$

$\boxed{\textit{signalRow} ::= \texttt{\#}\ \textit{logExpr}\ \texttt{\#}\ \textit{signalExpr}\ \texttt{\#}}$

$$\frac{\mathcal{W} \vdash \textit{logExpr} \Longrightarrow \textbf{true} \quad \mathcal{W} \vdash \textit{signalExpr} \Longrightarrow v}{\mathcal{W} \vdash \textit{signalRow} \Longrightarrow v}$$

$$\frac{\mathcal{W} \vdash \textit{logExpr} \Longrightarrow \textbf{false}}{\mathcal{W} \vdash \textit{signalRow} \Longrightarrow \textbf{No}}$$

Dodatek D

Stack Module

(0) CHARACTERISTICS

- type specified: stack
- type parameter: *elem*

(1) SYNTAX

MUTATORS

Program Name	Arg #1
JOIN	stack
PUSH	*elem*
POP	
SPLIT_1	stack
SPLIT_2	stack

OBSERVERS

Program Name	Arg #1	Result Type
TOP	stack	*elem*

SIGNALS

Signal Name
EMPTY

(2) CANONICAL TRACES

$canonical$: $traces \rightarrow$ bool

$canonical([\text{PUSH}(x_i)]_{i=1}^{s}) \Leftrightarrow$ **true**

$canonical(other) \Leftrightarrow$ **false**

(3) SEMANTICS OF MUTATORS

$\mathit{legal}(T; \mathrm{JOIN}(U)) \Leftrightarrow \mathbf{true}$

$\mathit{transition}(T; \mathrm{JOIN}(U)) = T.U$

$\mathit{legal}(T; \mathrm{PUSH}(x)) \Leftrightarrow \mathbf{true}$

$\mathit{transition}(T; \mathrm{PUSH}(x)) = T.\mathrm{PUSH}(x)$

$\mathit{legal}(T; \mathrm{POP}()) \Leftrightarrow T \neq \_$

$\mathit{signals}(\_; \mathrm{POP}()) = [\mathrm{EMPTY}()]$

$\mathit{transition}(T.\mathrm{PUSH}(x); \mathrm{POP}()) = T$

$\mathit{legal}(T; \mathrm{SPLIT\_1}(U)) \Leftrightarrow \mathbf{true}$

$\mathit{transition}([\mathrm{PUSH}(x_i)]_{i=1}^{k}.[\mathrm{PUSH}(x_i)]_{i=k+1}^{2k}; \mathrm{SPLIT\_1}(U)) = [\mathrm{PUSH}(x_i)]_{i=1}^{k}$

$\mathit{transition}([\mathrm{PUSH}(x_i)]_{i=1}^{k}.[\mathrm{PUSH}(x_i)]_{i=k+1}^{2k+1}; \mathrm{SPLIT\_1}(U)) = [\mathrm{PUSH}(x_i)]_{i=1}^{k}$

$\mathit{legal}(T; \mathrm{SPLIT\_2}(U)) \Leftrightarrow \mathbf{true}$

$\mathit{transition}(T; \mathrm{SPLIT\_2}([\mathrm{PUSH}(x_i)]_{i=1}^{k}.[\mathrm{PUSH}(x_i)]_{i=k+1}^{2k})) = T.[\mathrm{PUSH}(x_i)]_{i=k+1}^{2k}$

$\mathit{transition}(T; \mathrm{SPLIT\_2}([\mathrm{PUSH}(x_i)]_{i=1}^{k}.[\mathrm{PUSH}(x_i)]_{i=k+1}^{2k+1})) = T.[\mathrm{PUSH}(x_i)]_{i=k+1}^{2k+1}$

(4) SEMANTICS OF OBSERVERS

$\mathit{legal}(\mathrm{TOP}(T)) \Leftrightarrow T \neq \_$

$\mathit{signals}(\mathrm{TOP}(\_)) = [\mathrm{EMPTY}()]$

$\mathit{result}(\mathrm{TOP}(T.\mathrm{PUSH}(x))) = x$

Dodatek E

Index Module

(0) CHARACTERISTICS

- type specified: index
- value parameters: maxSize : int

(1) SYNTAX

MUTATORS

Program Name	Arg #1	Arg #2
CATENATE	index	
COPY	index	
DELETE	int	
DESTROY		
INIT		
PUT_INT	int	int

OBSERVERS

Program Name	Arg #1	Arg #2	Result Type
GET_INT	index	int	int
IS_EQUAL	index	index	bool
NB_ELEMS	index		int

SIGNALS

Signal Name
BAD_ACCESS
NEXT_INIT
NO_INIT
OUT_OF_SPACE

(2) CANONICAL TRACES

$canonical\ : traces \rightarrow \text{bool}$

$canonical(\_) \Leftrightarrow \textbf{true}$

$canonical(\text{INIT}().[\text{PUT\_INT}(i, x_i)]_{i=1}^{s}) \Leftrightarrow s \leq \text{maxSize}$

$canonical(other) \Leftrightarrow \textbf{false}$

(3) SEMANTICS OF MUTATORS

$legal(T; \text{CATENATE}(\_)) \Leftrightarrow \textbf{false}$

$signals(T; \text{CATENATE}(\_)) = [\text{NO\_INIT}()]$

$legal(\_; \text{CATENATE}(U)) \Leftrightarrow \textbf{false}$

$signals(\_; \text{CATENATE}(U)) = [\text{NO\_INIT}()]$

$legal(\text{INIT}().T; \text{CATENATE}(\text{INIT}().U)) \Leftrightarrow length(T) + length(U) \leq \text{maxSize}$

$signals(\text{INIT}().T; \text{CATENATE}(\text{INIT}().U)) =$

Condition	Value
$length(T) + length(U) \leq \text{maxSize}$	[ ]
$length(T) + length(U) > \text{maxSize}$	[OUT_OF_SPACE()]

$transition(T; \text{CATENATE}(\text{INIT}().[\text{PUT\_INT}(i, x_i)]_{i=1}^{s})) =$

$$T.[\text{PUT\_INT}(i + count(T, \text{PUT\_INT}), x_i)]_{i=1}^{s}$$

$legal(T; \text{COPY}(\_)) \Leftrightarrow \textbf{false}$

$signals(T; \text{COPY}(\_)) = [\text{NO\_INIT}()]$

$legal(\_; \text{COPY}(U)) \Leftrightarrow \textbf{false}$

$signals(\_; \text{COPY}(U)) = [\text{NO\_INIT}()]$

$legal(\text{INIT}().T; \text{COPY}(\text{INIT}().U)) \Leftrightarrow \textbf{true}$

$transition(T; \text{COPY}(U)) = U$

$legal(\_; \text{DELETE}(n)) \Leftrightarrow \textbf{false}$

$signals(\_; \text{DELETE}(n)) = [\text{NO\_INIT}()]$

$legal(\text{INIT}().T; \text{DELETE}(n)) \Leftrightarrow n \leq length(T)$

$signals(\text{INIT}().T; \text{DELETE}(n)) =$

Condition	Value
$n \leq length(T)$	[ ]
$n > length(T)$	[BAD_ACCESS()]

$transition(T.\text{PUT\_INT}(n, x_n).[\text{PUT\_INT}(j, x_j)]_{j=n+1}^{s}; \text{DELETE}(n)) =$

$$T.[\text{PUT\_INT}(j-1, x_j)]_{j=n+1}^{s}$$

$legal(\_; \text{DESTROY}()) \Leftrightarrow \textbf{false}$

$signals(\_; \text{DESTROY}()) = [\text{NO\_INIT}()]$

$legal(\text{INIT}().T; \text{DESTROY}()) \Leftrightarrow \textbf{true}$

$transition(T; \text{DESTROY}()) = \_$

$legal(\text{INIT}().T; \text{INIT}()) \Leftrightarrow \textbf{false}$

$signals(\text{INIT}().T; \text{INIT}()) = [\text{NEXT\_INIT}()]$

$legal(\_; \text{INIT}()) \Leftrightarrow \textbf{true}$

$transition(\_; \text{INIT}()) = \text{INIT}()$

$legal(\_; \text{PUT\_INT}(p, n)) \Leftrightarrow \textbf{false}$

$signals(\_; \text{PUT\_INT}(p, n)) = [\text{NO\_INIT}()]$

$legal(\text{INIT}().T; \text{PUT\_INT}(p, n)) \Leftrightarrow p \leq \text{maxSize} \wedge p \leq count(T, \text{PUT\_INT}) + 1$

$signals(\text{INIT}().T; \text{PUT\_INT}(p, n)) =$

Condition	Value
$p > count(T, \text{PUT\_INT}) + 1$	[BAD_ACCESS()]
$p = count(T, \text{PUT\_INT}) + 1 \wedge p = \text{maxSize} + 1$	[OUT_OF_SPACE()]
$legal(T; \text{PUT\_INT}(p, n))$	[ ]

$transition(T_1.\text{PUT\_INT}(p, x).T_2; \text{PUT\_INT}(p, n)) = T_1.\text{PUT\_INT}(p, n).T_2$

$transition(T; \text{PUT\_INT}(p, n)) = T.\text{PUT\_INT}(p, n)$

(4) SEMANTICS OF OBSERVERS

$$\mathit{legal}(\text{GET\_INT}(T, n)) \Leftrightarrow n \leq \mathit{count}(T, \text{PUT\_INT}) \wedge T \neq \_$$

$$\mathit{signals}(\text{GET\_INT}(\_, n)) = [\text{NO\_INIT}()]$$

$\mathit{signals}(\text{GET\_INT}(\text{INIT}().T, n)) =$

Condition	Value
$n > \mathit{length}(T)$	[BAD_ACCESS()]
$n \leq \mathit{length}(T)$	[ ]

$$\mathit{result}(\text{GET\_INT}(T_1.\text{PUT\_INT}(n, x).T_2, n)) = x$$

$$\mathit{legal}(\text{NB\_ELEMS}(T)) \Leftrightarrow T \neq \_$$

$$\mathit{signals}(\text{NB\_ELEMS}(\_)) = [\text{NO\_INIT}()]$$

$$\mathit{result}(\text{NB\_ELEMS}(T)) = \mathit{count}(T, \text{PUT\_INT})$$

$$\mathit{legal}(\text{IS\_EQUAL}(T, U)) \Leftrightarrow \textbf{true}$$

$$\mathit{result}(\text{IS\_EQUAL}(T, U)) \Leftrightarrow (T = U)$$

Dodatek F

Name Manager Module

(0) CHARACTERISTICS

- type specified: nameManager
- type parameter: *name*
  OBSERVERS

Program Name	Arg #1	Arg #2	Result Type
$\leq$	*name*	*name*	bool

- value parameters: first, last : *name*

(1) SYNTAX

MUTATORS

Program Name	Arg #1	Non-det. Result
PICK		*name*
DISPOSE	*name*	

SIGNALS

Signal Name
NO_MORE_NAMES

(2) CANONICAL TRACES

$canonical$: $traces \rightarrow$ bool

$canonical(\_) \Leftrightarrow$ **true**

$canonical(\text{PICK}()\uparrow a) \Leftrightarrow \text{first} \leq a \wedge a \leq \text{last}$

$$canonical(T.\text{PICK}()\uparrow a.\text{PICK}()\uparrow b) \Leftrightarrow \text{first} \leq a \wedge a \leq b \wedge b \leq \text{last} \wedge a \neq b \wedge canonical(T.\text{PICK}()\uparrow a)$$

$canonical(other) \Leftrightarrow$ **false**

$greaterElems$: nameManager $\times$ $name$ $\rightarrow$ nameManager

$greaterElems(\_, b) = \_$

$greaterElems(T.\text{PICK}()\uparrow a, b) =$

Condition	Value
$\neg(a \leq b)$	$greaterElems(T, b).\text{PICK}()\uparrow a$
$a \leq b$	$\_$

$lessElems$: nameManager $\times$ $name$ $\rightarrow$ nameManager

$lessElems(\_, b) = \_$

$lessElems(\text{PICK}()\uparrow a.T, b) =$

Condition	Value
$\neg(b \leq a)$	$\text{PICK}()\uparrow a.lessElems(T, b)$
$b \leq a$	$\_$

(3) SEMANTICS OF MUTATORS

$legal(T; \text{PICK}()) \Leftrightarrow \exists a : name[\text{first} \leq a \wedge a \leq \text{last} \wedge \forall R, S : traces[T \neq R.\text{PICK}()\uparrow a.S]]$

$feasible(T_1.\text{PICK}()\uparrow a.T_2; PICK()\uparrow a) \Leftrightarrow$ **false**

$feasible(other; \text{PICK}()\uparrow a) \Leftrightarrow \text{first} \leq a \wedge a \leq \text{last}$

$transition(T; \text{PICK}()\uparrow a) = lessElems(T, a).\text{PICK}()\uparrow a.greaterElems(T, a)$

$signals(T; \text{PICK}()) =$

Condition	Value
$legal(T; \text{PICK}())$	[ ]
$\neg legal(T; \text{PICK}())$	[NO_MORE_NAMES()]

$legal(T; \text{DISPOSE}(a)) \Leftrightarrow$ **true**

$transition(T_1.\text{PICK}()\uparrow a.T_2; \text{DISPOSE}(a)) = T_1.T_2$

$transition(T; \text{DISPOSE}(a)) = T$

Dodatek G

Receiver Module

(0) CHARACTERISTICS

- type specified: receiver
- foreign types: message

(1) SYNTAX

MUTATORS

Program Name	Arg #1	Arg #2
N−1_IND_SAP	int	message
N−1_ACCEPTED		

SIGNALS

Signal Name	Arg #1
ALREADY_ACCEPTED	
LAST_NOT_ACCEPTED	
N_IND_SAP	message
N−1_RES_SAP	int

(2) CANONICAL TRACES

$canonical$: $traces \rightarrow$ bool

$canonical(\text{N}-1\_\text{IND}\_\text{SAP}(s, m)) \Leftrightarrow$ **true**

$canonical(\text{N}-1\_\text{IND}\_\text{SAP}(s, m).\text{N}-1\_\text{ACCEPTED}()) \Leftrightarrow$ **true**

$canonical(other) \Leftrightarrow$ **false**

(3) SEMANTICS OF MUTATORS

$\_ \Rightarrow \text{N}{-}1\_\text{IND}\_\text{SAP}(0, \text{message} :: \_).\text{N}{-}1\_\text{ACCEPTED}()$

$\mathit{legal}(T; \text{N}{-}1\_\text{ACCEPTED}()) \Leftrightarrow \mathit{length}(T) = 1$

$\mathit{signals}(T.\text{N}{-}1\_\text{ACCEPTED}(); \text{N}{-}1\_\text{ACCEPTED}()) = [\text{ALREADY}\_\text{ACCEPTED}()]$

$\mathit{transition}(T; \text{N}{-}1\_\text{ACCEPTED}()) = T.\text{N}{-}1\_\text{ACCEPTED}()$

$\mathit{signals}(\text{N}{-}1\_\text{IND}\_\text{SAP}(s, m); \text{N}{-}1\_\text{ACCEPTED}()) = [\text{N}\_\text{IND}\_\text{SAP}(m)]$

$\mathit{legal}(T; \text{N}{-}1\_\text{IND}\_\text{SAP}(t, n)) \Leftrightarrow \mathit{length}(T) = 2$

$\mathit{signals}(\text{N}{-}1\_\text{IND}\_\text{SAP}(s, i); \text{N}{-}1\_\text{IND}\_\text{SAP}(t, j)) = [\text{LAST}\_\text{NOT}\_\text{ACCEPTED}()]$

$\mathit{transition}(\text{N}{-}1\_\text{IND}\_\text{SAP}(s, m).\text{N}{-}1\_\text{ACCEPTED}(); \text{N}{-}1\_\text{IND}\_\text{SAP}(t, n)) =$

Condition	Value
$t < s + 1$	$\text{N}{-}1\_\text{IND}\_\text{SAP}(s, m)$
$t = s + 1$	$\text{N}{-}1\_\text{IND}\_\text{SAP}(t, n)$
$t > s + 1$	$\text{N}{-}1\_\text{IND}\_\text{SAP}(s, m).\text{N}{-}1\_\text{ACCEPTED}()$

$\mathit{signals}(\text{N}{-}1\_\text{IND}\_\text{SAP}(s, m).\text{N}{-}1\_\text{ACCEPTED}(); \text{N}{-}1\_\text{IND}\_\text{SAP}(t, n)) =$

Condition	Value
$t < s + 1$	$[\text{N}{-}1\_\text{RES}\_\text{SAP}(s)]$
$t = s + 1$	$[\text{N}\_\text{IND}\_\text{SAP}(n), \text{N}{-}1\_\text{RES}\_\text{SAP}(t)]$
$t > s + 1$	[ ]

Dodatek H

File Table Module

W niniejszym dodatku przedstawimy specyfikację tablicy plików z systemu Unix. Ta specyfikacja jest wzorowana na analogicznej specyfikacji zapisanej w jednej z poprzednich wersji metody tropów i opublikowanej w pracy magisterskiej [KM96]. Poniżej opisujemy każdy z programów dostępu w języku naturalnym.

CREATE$(fpos, inode)$ jest modyfikatorem zapamiętującym w tablicy informację o otwartym pliku, którego i-węzeł ma numer *inode*, zaś bieżąca pozycja w tym pliku to $fpos$; ten modyfikator wytwarza niedeterministyczny wynik — indeks w tablicy, pod którym zapamiętano dodaną informację.

SETPOS$(no, fpos)$ jest modyfikatorem zmieniającym na $fpos$ bieżącą pozycję w pliku przechowywanym pod indeksem *no*.

INCCOUNTER(no) jest modyfikatorem zwiększającym o jeden licznik odwołań do pliku przechowywanego pod indeksem *no*.

DECCOUNTER(no) jest modyfikatorem zmniejszającym o jeden licznik odwołań do pliku przechowywanego pod indeksem *no*.

GETINODETABLEINDEX(ft, no) jest obserwatorem przekazującym numer i-węzła pliku przechowywanego w tablicy ft pod indeksem *no*.

GETPOS(ft, no) jest obserwatorem przekazującym bieżącą pozycję w pliku przechowywanym w tablicy ft pod indeksem *no*.

EXISTS(ft, no) jest obserwatorem sprawdzającym, czy w tablicy ft pod indeksem *no* jest przechowywany jakikolwiek plik.

ISFULL(ft) jest obserwatorem sprawdzającym, czy w tablicy ft są jeszcze wolne miejsca.

(0) CHARACTERISTICS

- type specified: filetable
- value parameters: G_filetable_size, G_inodetable_size, G_sequence_size : int

(1) SYNTAX

MUTATORS

Program Name	Arg #1	Arg #2	Non-det. Result
CREATE	int	int	int
SETPOS	int	int	
INCCOUNTER	int		
DECCOUNTER	int		

OBSERVERS

Program Name	Arg #1	Arg #2	Result Type
GETINODETABLEINDEX	filetable	int	int
GETPOS	filetable	int	int
EXISTS	filetable	int	bool
ISFULL	filetable		bool

SIGNALS

Signal Name
EBADF
EINVAL
ENFILE

(2) CANONICAL TRACES

$canonical$: $traces \rightarrow$ bool

$$canonical([\text{CREATE}(pos_i, ino_i)\uparrow no_i.[\text{INCCOUNTER}(no_i)]_{j=0}^{m_i}]_{i=0}^{n}) \quad \Leftrightarrow$$

$$\begin{aligned} & n < \text{G\_filetable\_size} \\ \wedge \quad & \forall i : \text{int } (0 \leq l < n) \; [\quad 0 \leq no_i < no_{i+1} < \text{G\_filetable\_size }] \\ \wedge \quad & \forall i : \text{int } (0 \leq l \leq n) \; [\quad 0 \leq pos_i < \text{G\_sequence\_size} \\ & \qquad\qquad\qquad\qquad \wedge \; 0 \leq ino_i < \text{G\_inodetable\_size }] \end{aligned}$$

$$canonical(other) \Leftrightarrow \textbf{false}$$

$validArgs$: int $\times$ int $\times$ int $\rightarrow$ bool

$$\begin{aligned} validArgs(pos, ino, no) \Leftrightarrow \quad & 0 \leq pos < \text{G\_sequence\_size} \\ \wedge \quad & 0 \leq ino < \text{G\_inodetable\_size} \\ \wedge \quad & 0 \leq no < \text{G\_filetable\_size} \end{aligned}$$

$exists$: filetable $\times$ int $\rightarrow$ bool

$exists(T_1.\text{CREATE}(pos, ino)\uparrow no.T_2, no) \Leftrightarrow \textbf{true}$

$exists(other, no) \Leftrightarrow \textbf{false}$

$isFull$: filetable $\rightarrow$ bool

$isFull(T) \Leftrightarrow count(T, \text{CREATE}) = \text{G\_filetable\_size}$

$select$: filetable $\times$ int $\times$ int $\rightarrow$ filetable

$select(\_, l, h) = \_$

$select(T.\text{CREATE}(pos, ino)\uparrow no.[\text{INCCOUNTER}(no)]_{j=0}^{m}, l, h) =$

Condition	Value
$\neg(l \leq no \leq h)$	$select(T, l, h)$
$l \leq no \leq h$	$select(T, l, h).\text{CREATE}(pos, ino)\uparrow no.[\text{INCCOUNTER}(no)]_{j=0}^{m}$

(3) SEMANTICS OF MUTATORS

$legal(T; \text{CREATE}(pos, ino)) \Leftrightarrow validArgs(pos, ino, 0) \wedge \neg isFull(T)$

$signals(T; \text{CREATE}(pos, ino)) =$

Condition	Value
$\neg validArgs(pos, ino, 0)$	[EINVAL()]
$validArgs(pos, ino, 0) \wedge isFull(T)$	[ENFILE()]
$legal(T; \text{CREATE}(pos, ino))$	[ ]

$feasible(T; \text{CREATE}(pos, ino)\uparrow no) \Leftrightarrow validArgs(0, 0, no) \wedge \neg exists(T, no)$

$transition(T; \text{CREATE}(pos, ino)\uparrow no) =$

$$select(T, 0, no).\text{CREATE}(pos, ino)\uparrow no.select(T, no, \text{G\_filetable\_size})$$

$legal(T; \text{SETPOS}(no, pos)) \Leftrightarrow validArgs(pos, 0, no) \wedge exists(T, no)$

$signals(T; \text{SETPOS}(no, pos)) =$

Condition	Value
$\neg validArgs(pos, 0, no)$	[EINVAL()]
$validArgs(pos, 0, no) \wedge \neg exists(T, no)$	[EBADF()]
$legal(T; \text{SETPOS}(no, fpos)$	[ ]

$transition(T_1.\text{CREATE}(oldPos, ino)\uparrow no.T_2; \text{SETPOS}(no, pos)) =$

$$T_1.\text{CREATE}(pos, ino)\uparrow no.T_2$$

$legal(T; \text{INCCOUNTER}(no)) \Leftrightarrow exists(T, no)$

$signals(T; \text{INCCOUNTER}(no)) =$

Condition	Value
$\neg validArgs(0, 0, no)$	[EINVAL()]
$validArgs(0, 0, no) \wedge \neg exists(T, no)$	[EBADF()]
$legal(T; \text{INCCOUNTER}(no))$	[ ]

$transition(T_1.\text{CREATE}(pos, ino)\uparrow no.T_2; \text{INCCOUNTER}(no)) =$

$$T_1.\text{CREATE}(pos, ino)\uparrow no.\text{INCCOUNTER}(no).T_2$$

$legal(T; \text{DECCOUNTER}(no)) \Leftrightarrow exists(T, no)$

$signals(T; \text{DECCOUNTER}(no)) =$

Condition	Value
$\neg validArgs(0, 0, no)$	[EINVAL()]
$validArgs(0, 0, no) \wedge \neg exists(T, no)$	[EBADF()]
$legal(T; \text{DECCOUNTER}(no))$	[ ]

$transition(T_1.\text{CREATE}(pos, ino)\uparrow no.\text{INCCOUNTER}(no).T_2; \text{DECCOUNTER}(no)) =$

$$T_1.\text{CREATE}(pos, ino)\uparrow no.T_2$$

$transition(T_1.\text{CREATE}(pos, ino)\uparrow no.T_2; \text{DECCOUNTER}(no)) = T_1.T_2$

(4) SEMANTICS OF OBSERVERS

$legal(\text{GETINODETABLEINDEX}(T, no)) \Leftrightarrow exists(T, no)$

$signals(\text{GETINODETABLEINDEX}(T, no)) =$

Condition	Value
$\neg validArgs(0, 0, no)$	[EINVAL()]
$validArgs(0, 0, no) \wedge \neg exists(T, no)$	[EBADF()]
$legal(\text{GETINODETABLEINDEX}(T, no))$	[ ]

$result(\text{GETINODETABLEINDEX}(T_1.\text{CREATE}(pos, ino)\uparrow no.T_2, no)) = ino$

$legal(\text{GETPOS}(T, no)) \Leftrightarrow exists(T, no)$

$signals(\text{GETPOS}(T, no)) =$

Condition	Value
$\neg validArgs(0, 0, no)$	[EINVAL()]
$validArgs(0, 0, no) \wedge \neg exists(T, no)$	[EBADF()]
$legal(\text{GETPOS}(T, no))$	[ ]

$result(\text{GETPOS}(T_1.\text{CREATE}(pos, ino)\uparrow no.T_2, no)) = pos$

$legal(\text{EXISTS}(T, no)) \Leftrightarrow validArgs(0, 0, no)$

$signals(\text{EXISTS}(T, no)) =$

Condition	Value
$\neg validArgs(0, 0, no)$	[EINVAL()]
$validArgs(0, 0, no)$	[ ]

$result(\text{EXISTS}(T, no)) \Leftrightarrow exists(T, no)$

$legal(\text{ISFULL}(T)) \Leftrightarrow \mathbf{true}$

$result(\text{ISFULL}(T)) \Leftrightarrow isFull(T)$

Dodatek I

File Descriptor Table Module

W niniejszym dodatku przedstawimy specyfikację tablicy desktryptorów plików z systemu Unix. Ta specyfikacja jest wzorowana na analogicznej specyfikacji zapisanej w jednej z poprzednich wersji metody tropów i opublikowanej w pracy magisterskiej [KM96]. Poniżej opisujemy każdy z programów dostępu w języku naturalnym.

CREATE(no) jest modyfikatorem rezerwującym nowy deskryptor otwartego pliku odwołujący się do pozycji no w tablicy plików (filetable; por. dodatek H); numer nowego deskryptora można odczytać wywołując obserwator **CREATE_RES**(fno).

RELEASE(fd) jest modyfikatorem zwalniającym deskryptor o numerze fd.

DUP(fd) jest modyfikatorem rezerwującym nowy deskryptor otwartego pliku odwołujący się do tej samej pozycji w tablicy plików (filetable; por. dodatek H) co deskryptor fd; numer nowego deskryptora można odczytać wywołując obserwator **DUP_RES**(fd).

EXISTS(fdt, fd) jest obserwatorem sprawdzającym, czy deskryptor o numerze fd jest używany w tablicy fdt.

GETMINFREE(fdt) jest obserwatorem przekazującym numer najmniejszego wolnego deskryptora w tablicy fdt.

GETFILETABLEINDEX(fdt, fd) jest obserwatorem przekazującym indeks w tablicy plików (filetable; por. dodatek H), do którego odwołuje się deskryptor fd.

GETMAXFD(fdt) jest obserwatorem przekazującym numer największego używanego deskryptora w tablicy fdt.

ISFULL(fdt) jest obserwatorem sprawdzającym, czy w tablicy fdt są jeszcze wolne miejsca.

(0) CHARACTERISTICS

- type specified: fdtable
- value parameters: G_filetable_size, G_fdtable_size : int

(1) SYNTAX

MUTATORS

Program Name	Arg #1
CREATE	int
RELEASE	int
DUP	int

OBSERVERS

Program Name	Arg #1	Arg #2	Result Type
CREATE_RES	fdtable	int	int
DUP_RES	fdtable	int	int
EXISTS	fdtable	int	bool
GETMINFREE	fdtable		int
GETFILETABLEINDEX	fdtable	int	int
GETMAXFD	fdtable		int
ISFULL	fdtable		bool

SIGNALS

Signal Name
EBADFD
EINVAL
EMFILE

(2) CANONICAL TRACES

$canonical$: $traces \rightarrow$ bool

$canonical([\text{CREATE}(no_i)]_{i=0}^{n}.[\text{RELEASE}(fd_j)]_{j=0}^{m}) \quad \Leftrightarrow$

$$\begin{aligned} & n < \text{G\_fdtable\_size} \\ \wedge \quad & \forall j : \text{int } (0 \leq l < m) \; [\quad 0 \leq fd_j < fd_{j+1} < n - 1 \;] \\ \wedge \quad & \forall j : \text{int } (0 \leq j \leq m) \; [\quad no_{fd_j} = 0 \;] \\ \wedge \quad & \forall i : \text{int } (0 \leq i \leq n) \; [\quad 0 \leq no_i < \text{G\_filetable\_size} \;] \end{aligned}$$

$canonical(other) \Leftrightarrow$ **false**

$validArgs$: int $\times$ int $\rightarrow$ bool

$validArgs(no, fd) \Leftrightarrow 0 \leq no < \text{G\_filetable\_size} \wedge 0 \leq fd < \text{G\_fdtable\_size}$

$exists$: fdtable $\times$ int $\rightarrow$ bool

$exists(T_1.\text{RELEASE}(fd).T_2, fd) \Leftrightarrow \textbf{false}$

$exists([\text{CREATE}(no_i)]_{i=0}^{n}.[\text{RELEASE}(fd_j)]_{j=0}^{m}, fd) \Leftrightarrow fd \leq n$

$isFull$: fdtable $\rightarrow$ bool

$isFull(T) \Leftrightarrow count(T, \text{CREATE}) = \text{G\_fdtable\_size} \wedge count(T, \text{RELEASE}) = 0$

$getMinFree$: fdtable $\rightarrow$ int

$legal(getMinFree(T)) = \neg isFull(T)$

$getMinFree(T_1.\text{CREATE}(no).\text{RELEASE}(fd).T_2) = fd$

$getMinFree(T) = length(T)$

$normalize$: $traces \rightarrow traces$

$normalize([\text{CREATE}(no_i)]_{i=0}^{n}.T.\text{RELEASE}(n)) = normalize([\text{CREATE}(no_i)]_{i=0}^{n-1}.T)$

$normalize(T) = T$

$select$: $traces \times \text{int} \times \text{int} \rightarrow traces$

$select(T.\text{RELEASE}(fd), l, h) =$

Condition	Value
$\neg(l \leq fd \leq h)$	$select(T, l, h)$
$l \leq fd \leq h$	$select(T, l, h).\text{RELEASE}(fd)$

$select(T, l, h) = T$

(3) SEMANTICS OF MUTATORS

$legal(T; \text{CREATE}(no)) \Leftrightarrow validArgs(no, 0) \wedge \neg isFull(T)$

$signals(T; \text{CREATE}(no)) =$

Condition	Value
$\neg validArgs(no, 0)$	[EINVAL()]
$validArgs(no, 0) \wedge isFull(T)$	[EMFILE()]
$legal(T; \text{CREATE}(no))$	[ ]

$transition([\text{CREATE}(no_i)]_{i=0}^{n}.\text{RELEASE}(fd).T; \text{CREATE}(no)) =$

$$[\text{CREATE}(no_i)]_{i=0}^{fd-1}.\text{CREATE}(no).[\text{CREATE}(no_i)]_{i=fd+1}^{n}.T$$

$$transition(T; \text{CREATE}(no)) = T.\text{CREATE}(no)$$

$$legal(T; \text{RELEASE}(fd)) \Leftrightarrow exists(T, fd)$$

$$signals(T; \text{RELEASE}(fd)) = \begin{array}{|c|c|}\hline \text{Condition} & \text{Value} \\ \hline \neg validArgs(0, fd) & [\text{EINVAL}()] \\ \hline validArgs(0, fd) \wedge \neg exists(T, fd) & [\text{EBADFD}()] \\ \hline legal(T; \text{RELEASE}(fd)) & [\ ] \\ \hline \end{array}$$

$$transition([\text{CREATE}(no_i)]_{i=0}^{n}.[\text{RELEASE}(fd_j)]_{j=0}^{m}; \text{RELEASE}(n)) =$$
$$normalize([\text{CREATE}(no_i)]_{i=0}^{n-1}.[\text{RELEASE}(fd_j)]_{j=0}^{m}$$

$$transition([\text{CREATE}(no_i)]_{i=0}^{n}.[\text{RELEASE}(fd_j)]_{j=0}^{m}; \text{RELEASE}(fd)) =$$
$$[\text{CREATE}(no_i)]_{i=0}^{fd-1}.\text{CREATE}(0).[\text{CREATE}(no_i)]_{i=fd+1}^{n}.$$
$$select([\text{RELEASE}(fd_j)]_{j=0}^{m}, 0, fd).\text{RELEASE}(fd).$$
$$select([\text{RELEASE}(fd_j)]_{j=0}^{m}, fd, \text{G\_fdtable\_size})$$

$$legal(T; \text{DUP}(fd)) \Leftrightarrow exists(fd) \wedge \neg isFull(T)$$

$$signals(T; \text{DUP}(fd)) = \begin{array}{|c|c|}\hline \text{Condition} & \text{Value} \\ \hline \neg validArgs(0, fd) & [\text{EINVAL}()] \\ \hline validArgs(0, fd) \wedge \neg exists(T, fd) & [\text{EBADFD}()] \\ \hline exists(T, fd) \wedge isFull(T) & [\text{EMFILE}()] \\ \hline legal(T; \text{DUP}(fd)) & [\ ] \\ \hline \end{array}$$

$$transition([\text{CREATE}(no_i)]_{i=0}^{n}.\text{RELEASE}(copyFd).T; \text{DUP}(fd)) =$$
$$[\text{CREATE}(no_i)]_{i=0}^{copyFd-1}.\text{CREATE}(no_{fd}).[\text{CREATE}(no_i)]_{i=copyFd+1}^{n}.T$$

$$transition([\text{CREATE}(no_i)]_{i=0}^{n}; \text{DUP}(fd)) = [\text{CREATE}(no_i)]_{i=0}^{n}.\text{CREATE}(no_{fd})$$

(4) SEMANTICS OF OBSERVERS

$$legal(\text{CREATE\_RES}(T, no)) \Leftrightarrow validArgs(no, 0) \wedge \neg isFull(T)$$

$$result(\text{CREATE\_RES}(T, no)) = getMinFree(T)$$

$$legal(\text{DUP\_RES}(T, fd)) \Leftrightarrow exists(fd) \wedge \neg isFull(T)$$

$$result(\text{DUP\_RES}(T, fd)) = getMinFree(T)$$

$legal(\text{EXISTS}(T, fd)) \Leftrightarrow validArgs(0, fd)$

$signals(\text{EXISTS}(T, fd)) =$

Condition	Value
$\neg validArgs(0, fd)$	[EINVAL()]
$validArgs(0, fd)$	[ ]

$result(\text{EXISTS}(T, fd)) \Leftrightarrow exists(T, fd)$

$legal(\text{GETMINFREE}(T)) \Leftrightarrow \textbf{true}$

$result(\text{GETMINFREE}(T)) =$

Condition	Value
$isFull(T)$	-1
$\neg isFull(T)$	$getMinFree(T)$

$legal(\text{GETFILETABLEINDEX}(T, fd)) \Leftrightarrow exists(T, fd)$

$signals(\text{GETFILETABLEINDEX}(T, fd)) =$

Condition	Value
$\neg validArgs(0, fd)$	[EINVAL()]
$validArgs(0, fd) \wedge \neg exists(T, fd)$	[EBADFD()]
$legal(\text{GETFILETABLEINDEX}(T, fd))$	[ ]

$result(\text{GETFILETABLEINDEX}([\text{CREATE}(no_i)]_{i=0}^{fd}.T, fd)) = no_{fd}$

$legal(\text{GETMAXFD}(T)) \Leftrightarrow \textbf{true}$

$result(\text{GETMAXFD}(T)) = count(T, \text{CREATE}) - 1$

$legal(\text{ISFULL}(T)) \Leftrightarrow \textbf{true}$

$result(\text{ISFULL}(T)) \Leftrightarrow isFull(T)$