Uniwersytet Warszawski

Praca magisterska z informatyki

Analiza statyczna w Extended ML

Mikołaj Konarski

22 sierpnia 1997

Praca napisana pod kierunkiem
dra hab. Andrzeja Tarleckiego

Spis rzeczy

Część I
Wstęp

1 Rzut oka na dziedzinę

W ostatnich latach programowanie funkcyjne i metody specyfikacji, czerpiąc z siebie nawzajem, szybko się rozwijały. Owocami tego pożytecznego zbliżenia są między innymi formalizm Extended ML, a w dalszym rzędzie system EML Kit. W tym rozdziale postaram się je opisać na tle środowiska, w którym się uformowały.

1.1 Standard ML

Standard ML (SML) [16] jest funkcyjnym językiem programowania, co oznacza, że działanie programu nie polega na stopniowej zmianie stanu maszyny, jak to jest w przypadku języków imperatywnych, lecz na obliczaniu wartości wyrażeń. Do opisywania wyrażeń służy część Standard ML zwana językiem Jądra.

Matematyczną podstawą modelu obliczeń SML jest rachunek λ [1] wraz z rozszerzonym systemem typów ML [3]. Uczynienie aplikowania funkcji głównym motorem ewaluacji ułatwia „matematyczne" myślenie o działaniu programu. Z kolei silny system typów SML, dzięki swojej elastyczności i ogólności nie krępując użytkownika, pozwala wykrywać zdecydowaną większość błędów jeszcze przed wykonaniem programu, podczas procesu zwanego analizą statyczną.

SML posiada bardzo silne mechanizmy modularyzacji, do których można się odwoływać przy pomocy języka Modułów. Do budowania globalnej struktury programu służą struktury, które są atomowymi modułami, i funktory, wyrażające hierarchiczne zależności między modułami i służące do ich składania.

Podstawowym narzędziem do opisu własności modułów, czyli do ich specyfikacji, są sygnatury:

```
signature TOTAL_PREORDER =
sig
    type elem
   val leq : elem * elem -> bool
    (* leq should be a total preorder *)
end
```

TOTAL_PREORDER to nazwa sygnatury, zdanie „type elem” postuluje istnienie typu o nazwie elem, zaś zdanie „val leq : elem * elem -> bool” stwierdza, że powinna istnieć funkcja leq o typie elem * elem -> bool. Komentarz w języku naturalnym zamykający sygnaturę opisuje żądane własności funkcji leq.

Zarówno struktury jak i funktory mogą być opatrzone sygnaturami. Ta przykładowa struktura jest zadeklarowana jako pasująca do zdefiniowanej wcześniej sygnatury TOTAL_PREORDER:

```
structure OddRational : TOTAL_PREORDER =
struct
    type elem = int * int
    fun leq ((m, n), (k, l)) =
        if n = 0 andalso l = 0
            then m < 0 orelse (m > 0 andalso k > 0)
        else if (n < 0 andalso l >= 0)
                orelse (n >= 0 andalso l < 0)
                 then k * n <= m * l
             else m * l <= k * n
end
```

Typ elem jest tu parą liczb całkowitych, zaś funkcja leq ma pewne cechy naturalnego porządku na liczbach wymiernych. Można pokazać, że wszystkie wymogi wyrażone w TOTAL_PREORDER zostały spełnione (również te opisane w komentarzu).

1.2 Extended ML

Połączenie prostoty modelu obliczeń Jądra z siłą języka Modułów czyni z SML wspaniałe narzędzie do tworzenia i pielęgnowania dużych i złożonych systemów oprogramowania. Narzucającymi się sposobami pracy z SML są warianty metodologii specyfikacji algebraicznych, choćby takie w których używa się języka naturalnego, jak w przykładzie powyżej.

Dzięki prostocie Jądra SML istnieje duża swoboda wyboru języków do wyrażania własości programów. Można używać języków bardzo słabych, jak na przykład logika równościowa, które w wypadku programowania imperatywnego nie miałyby szans zastosowania. Można używać także języków silnych, pozwalających wyrażać skomplikowane własności w sposób bardzo zwarty. Do takich języków należy na przykład bardzo mocne rozszerzenie logiki pierwszego rzędu, za pomocą którego buduje się ciała aksjomatów w Extended ML.

Extended ML (EML) [10] jest formalizmem do wywodzenia programów w Standard ML z ich specyfikacji. Proces konstrukcji oprogramowania odbywa się stopniowo, począwszy od czystej specyfikacji, poprzez stadia, w których przeplatają się fragmenty specyfikacji i kodu w SML, a skończywszy na w pełni wykonywalnym programie w SML. Wszystkie kroki wyrażane są w Extended ML, a ich poprawność dowodzona jest według reguł Semantyki Weryfikacyjnej EML.

Jako język, Extended ML jest rozszerzeniem (dużego podzbioru) Standard ML. Można go podzielić, podobnie jak w przypadku SML, na język Jądra i język Modułów. Szczególną rolę w procesie tworzenia oprogramowania odgrywa system modułów EML dopuszczający, oprócz elementów znanych z SML, sygnatury w których występują aksjomaty:

```
signature TOTAL_PREORDER =
sig
    type elem
    val leq : elem * elem -> bool
    axiom forall (a, b) => leq (a, b) orelse leq (b, a)
    and forall (c, d, e) =>
        (leq (c, d) andalso leq (d, e)) implies leq (c, e)
end
```

Dzięki swojemu całkowitemu sformalizowaniu oraz sile mechanizmów modularyzacyjnych i języka specyfikacji, Extended ML daje możliwość konstruowania oprogramowania o niezwykłej niezawodności i łatwości rozbudowy oraz pielęgnacji. O wygodzie stosowania EML decydują dwie rzeczy: czytelność i klarowność formalnej definicji oraz dostępność i jakość narzędzi wspomagających.

1.3 Semantyka naturalna

Język Standard ML jest ściśle opisany w swojej definicji [16]. Opis semantyki składa się z dwóch głównych części: Semantyki Statycznej i Semantyki Dynamicznej. Każda z nich podzielona jest z kolei na fragment dotyczący języka Jądra oraz fragment dotyczący języka Modułów.

Formalizm wykorzystany do przedstawienia semantyki SML zwie się semantyką naturalną [5]. W skład definicji wchodzą zbiory reguł postaci:

$$\frac{\nu_1 \quad \cdots \quad \nu_k}{\phi}$$

gdzie konkluzja ϕ jest zdaniem, zaś przesłanki ν_i są albo zdaniami albo regułami. W przypadku Semantyki Statycznej zdania mówią o możliwości przypisania elementowi języka pewnego typu, w przypadku Semantyki Dynamicznej — wartości.

Extended ML również posiada formalną definicję [10]. Podobnie jak EML jest w pewnym sensie rozszerzeniem SML, tak i definicja EML jest niejako nadbudowana na definicji SML. Choć miejscami zmodyfikowany jest sposób prezentacji, wprowadzone nowe elementy, poprawione błędy, to jednak ogólna struktura odziedziczona z definicji SML pozostaje zachowana. Jedyną zupełnie nową częścią definicji EML jest Semantyka Weryfikacyjna, opisana tym samym formalizmem co Semantyka Statyczna i Semantyka Dynamiczna, a służąca wyrażeniu własności spełnienia specyfikacji przez implementację.

Semantyka naturalna czyni możliwym opisanie tak wielkiego i skomplikowanego formalizmu jak EML we względnie zwarty i elegancki sposób. Ta forma przezentacji ma jednak również swoje złe strony. Niektóre drobne błędy w definicji potrafią długo pozostawać niezauważone. Niektóre mechanizmy wydają się proste i jednoznaczne jedynie dotąd, dopóki nie zostanie podjęta pierwsza próba ich implementacji. Niektóre globalne relacje pomiędzy częściami definicji są niejasne, dopóki jej hierarchia modułów nie zostanie opisana w poważnym formalizmie do definiowania struktury modularnej takim jak na przykład język Modułów SML.

1.4 EML Kit

O systemie EML Kit można myśleć (z pewną dozą dobrej woli) jako o w pełni deterministycznym, algorytmicznym i szczegółowym opisie Semantyki Statycznej i Semantyki Dynamicznej EML.

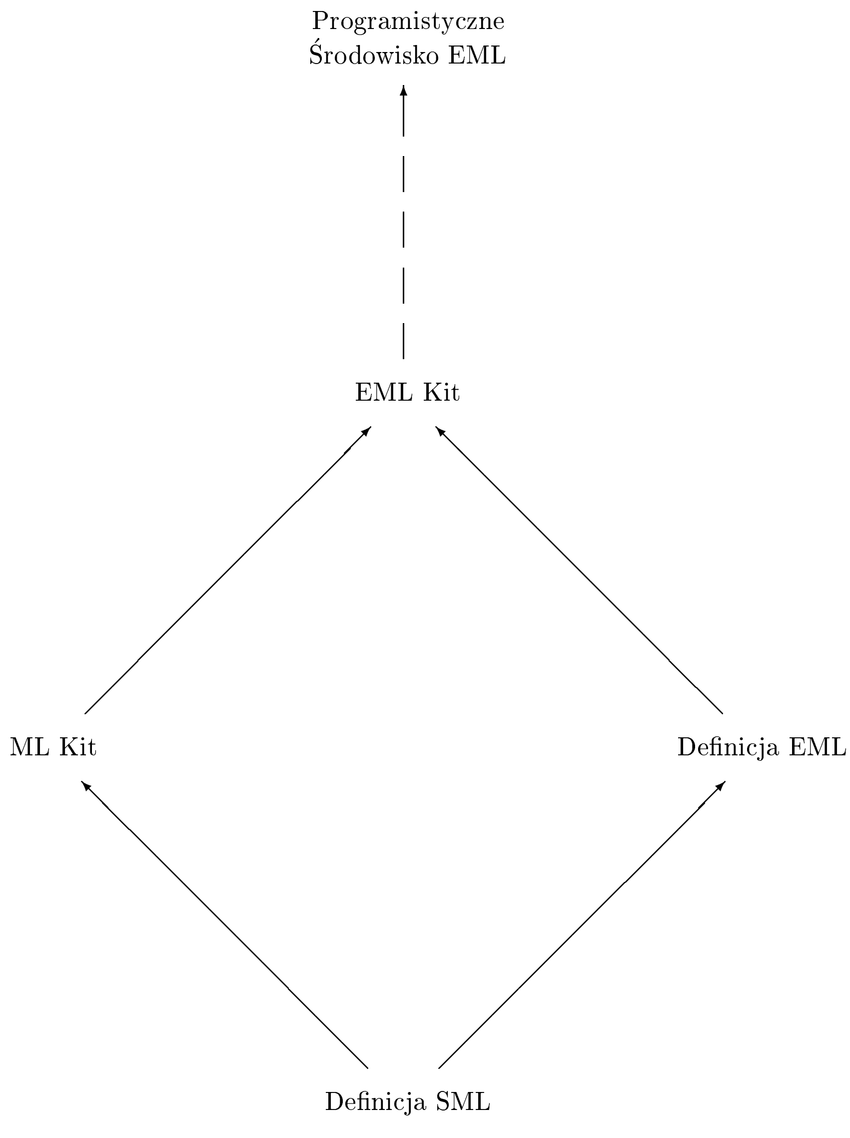
EML Kit zmierza do tego, by pełnić funkcje (podstawy i miejca do testowania) wygodnego środowiska do formalnego wywodzenia programów, przy użyciu języka, formalizmu i metodologii Extended ML.

Wersja 1.0 stworzona przez Marcina Jurdzińskiego, Mikołaja Konarskiego, Sławomira Leszczyńskiego i Aleksego Schuberta, jest kompletną implementacją Extended ML jako „języka programowania". Pozwala ona dokonywać analizy składniowej, sprawdzać poprawność typową i wyliczać wartość dowolnych programów EML, czy będzie to czysty Standard ML, czy też pełny EML z aksjomatami i innymi konstrukcjami specyfikacyjnymi. Proces ten odbywa się w ścisłej zgodności z Semantyką Statyczną i Sematyką Dynamiczną EML.

EML Kit bazuje na systemie ML Kit [2]; elastycznym, eleganckim i zmodularyzowanym interpreterze Standard ML, napisanym w Standard ML. ML Kit jest wierną i tak bezpośrednią, jak to możliwe, implemetacją języka zdefiniowanego w definicji SML. Składnia abstakcyjna jest niemal dosłownie wzięta z definicji, a projekt wielu innych detali, takich jak na przykład konwencje nazewnicze, jest ściśle zainspirowany definicją.

1.5 Podsumowanie i diagram zależności

Dwa fakty — ten, że definicja EML jest rozszerzeniem definicji SML, i ten, że ML Kit jest tak bliski tej ostatniej — pomogły nam w naszych wysiłkach uczynienia systemu EML Kit wierną i klarowną implementacją Extended ML. Byliśmy w stanie rozszerzyć ML Kit analogicznie do sposobu w jaki definicja EML rozszerza definicję SML, zachowując styl programowania właściwy dla systemu ML Kit. W ten sposób stworzyliśmy podstawę do rozwoju przyszłych narzędzi Środowiska Programistycznego Extended ML, takich jak generatory musików dowodowych, czy prototypowe systemy dowodzenia. Ilustruje to znajomo wyglądający diagram poniżej.



2 Analiza statyczna

Każdy nowoczesny język programowania oparty jest na właściwym sobie prostym, klarownym matematycznym modelu. Proces analizy statycznej programu, przebiegający między jego napisaniem a wykonaniem, polega na przetworzeniu go w odpowiadający mu matematyczny obiekt. W tym rozdziale chciałbym przedstawić bliżej to ujęcie analizy statycznej, ilustrując je konkretnymi przykładami dotyczącymi specyfiki Extended ML.

2.1 Objaśnienie terminu

W skład analizy statycznej wchodzą działania, dokonywane w z góry ograniczonym czasie, a mające za podstawę tekst programu. Wynika stąd, że w przypadku języków dopuszczających zjawisko nieterminacji, samego procesu wykonywania programu nie można zaliczyć do analizy statycznej.

Analizę statyczną wykonuje się, by sprawdzić poprawność programów, na przykład poprawność składniową, oraz by przygotować program do wykonania, być może poddając go wyrafinowanym przekształceniom w celu optymalizacji. Jeśli potraktujemy Extended ML jako język programowania, to na podstawie systemu EML Kit będziemy mogli obejrzeć elementy przykładowej analizy statycznej tego języka.

Pierwszym stadium analizy statycznej w systemie EML Kit jest analiza składniowa, to znaczy przypisanie programowi jego drzewa rozbioru według gramatyki podanej w definicji Extended ML. Potem następuje szereg dodatkowych korekcji tego drzewa, wynikających ze specyfiki składni EML i sposobu jej prezentacji, a w rezultacie powstaje ostateczne drzewo składni abstrakcyjnej programu, które będzie użyte jako podstawa zarówno do dalszej analizy statycznej jak i procesu wykonywania. Następnym etapem jest elaboracja, to znaczy sprawdzanie poprawności programu jeśli chodzi o typy. Elaboracja odbywa się w oparciu o Semanykę Statyczną Jądra EML i Semantykę Statyczną Modułów EML, a przy przejściu z języka Jądra do Modułów dodatkowo odbywa się proces zwany rozwikływaniem przeładowania (ang.: overloading resolution).

Ponieważ programy napisane w EML mogą się pętlić, ich wykonywanie, które powinno się odbywać na podstawie Semantyki Dynamicznej EML, nie może być zaliczone do analizy statycznej. Z podobnych powodów weryfikacja programów w EML, która odbywałaby się na podstawie Semantyki Weryfikacyjnej EML, nie może być zaliczona do analizy statycznej. Ale już generowanie musików dowodowych (ang.: proof obligations) — pewna operacja wstępna względem weryfikacji, która być może zostanie w przyszłości zaimplementowana w ramach systemu EML Kit — zdecydowanie ma charakter statyczny i można ją przytoczyć jako przykład analizy statycznej, choć może nieco wyższego, niż się to zwykle spotyka, rzędu.

W pozostałej części rozdziału (i w całej pracy) skupimy się na przypadku analizy statycznej w najwęższym rozumieniu tego terminu, to znaczy na sprawdzania poprawności typowej fraz językowych, względem zadanej sematyki statycznej.

2.2 Podstawowe pojęcia

2.2.1 Obiekt semantyczny

Sematyka statyczna frazom języka, czyli obiektom syntaktycznym, przypisuje pewne twory matematyczne, zwane obiektami semantycznymi. Pełne zestawienie obiektów semantycznych Semantyki Statycznej EML znajduje się w dodatku B.

Najbardziej oczywistym przykładem obiektu semantycznego jest typ. W Semantyce Statycznej Jądra EML wyrażeniom przypisywane są obiekty semantyczne τ, należące do dziedziny semantycznej typów Type. Wyrażeniom będącymi rekordami przypisywane są typy rekordowe, wyrażeniom reprezentującym funkcje przypisywane są typy funkcyjne, itd.

Innym powszechnie spotykanym rodzajem obiektów semantycznych są środowiska. Reprezentują one skończony zbiór nazw, wraz z przypisanymi nazwom obiektami semantycznymi, na przykład typami. W Extended ML wynikiem analizy statycznej deklaracji są obiekty semantyczne oznaczane E, należące do dziedziny semantycznej środowisk Env.

W Semantyce Statycznej Modułów występują jeszcze bardziej złożone obiekty semantyczne. Strukturom odpowiadają semantyczne struktury, składające się z nazwy struktury i środowiska opisującego komponenty struktury, wraz z odpowiadającymi im obiektami semantycznymi:

$$S \text{ lub } (m, E) \quad \in \quad \mathrm{Str} = \mathrm{StrName} \times \mathrm{Env}$$

gdzie S to meta-zmienna przebiegająca dziedzinę Str, będącą produktem dziedzin StrName i Env, po których przebiegają z kolei meta-zmienne m i E.

Sygnaturom odpowiadają semantyczne sygnatury, składające się z semantycznej struktury i zbioru nazw, o których należy myśleć, że są związane:

$$\Sigma \text{ lub } (N)S \quad \in \quad \mathrm{Sig} = \mathrm{NameSet} \times \mathrm{Str}$$

gdzie Σ, N i S to znowu meta-zmienne.

2.2.2 Osąd

Osądy to zdania postaci:

$$C \vdash \mathit{fraza} \Rightarrow A$$

gdzie C i A to obiekty semantyczne, zaś *fraza* to element języka. Zdanie takie można czytać „w kontekście C, *fraza* daje A".

Przy pomocy Semantyki Statycznej Jądra EML można wywodzić na przykład zdania podobne do:

$$E \vdash \mathit{exp} \Rightarrow \tau$$

(„w środowisku E, wyrażenie *exp* daje typ τ"),

$$E_1 \vdash \mathit{dec} \Rightarrow E_2$$

(„w środowisku E_1, deklaracja *dec* daje środowisko E_2"). Zaś w Semantyce Statycznej Modułów pojawiają się między innymi osądy postaci:

$$B \vdash \mathit{sigexp} \Rightarrow S, \gamma$$

gdzie B to baza zawierająca w szczególności środowisko, *sigexp* to sygnatura, a γ to ślad.

Aby zasygnalizować, czym jest ślad, musimy przypomieć, że w Semantyce Statycznej EML osądami prawdziwymi są te, które można wywieść przy pomocy reguł. Każdy prawdziwy osąd posiada więc (być może więcej niż jedno) drzewo wywodu. W pewnym uproszczeniu ślady są właśnie drzewami wywodu pewnych osądów.

2.2.3 Obiekt główny

Niech $\mathcal{S}$ będzie semantyką statyczną i niech $\succ$ będzie częściowym porządkiem na zbiorze wywodów $\mathcal{S}$. Wtedy obiekt semantyczny A nazywamy głównym (ang.: principal) dla obiektu syntaktycznego *fraza* w kontekście obiektu semantycznego C, względem relacji $\succ$, w ramach semantyki $\mathcal{S}$, gdy

- istnieje wywód γ w ramach $\mathcal{S}$ osądu $C \vdash \mathit{fraza} \Rightarrow A$,
- jeśli γ' jest wywodem $C \vdash \mathit{fraza} \Rightarrow A'$ w ramach $\mathcal{S}$, to $\gamma \succ \gamma'$.

W Semantyce Statycznej EML pojawiają się definicje mające charakter definicji obiektu głównego. Przytoczę dwie spośród nich.

Typ τ wraz ze śladem γ jest główny dla wyrażenia *exp* w kontekście C, gdy (w trochę innym sformułowaniu niż oryginalne)

- $C \vdash \mathit{exp} \Rightarrow \tau, \gamma$,
- jeśli $C \vdash \mathit{exp} \Rightarrow \tau', \gamma'$, to $\tau \succ \tau'$ i $\gamma \succ \gamma'$.

gdzie „$\succ$” jest relacją uogólniania typów ($\tau \succ \tau'$, gdy typ τ jest ogólniejszy niż τ', czyli gdy typ τ' jest instancją typu τ), którą można w naturalny sposób rozszerzyć na ślady. Można udowodnić, że w SML (a także w EML) każde wyrażenie, które ma typ, ma również typ główny [14].

Mówimy, że sygnatura semantyczna $(N)S$ z γ jest główna dla sygnatury syntaktycznej *sigexp* w bazie B gdy (w uproszczeniu)

- $B \vdash \mathit{sigexp} \Rightarrow S, \gamma$,
- jeśli $B \vdash \mathit{sigexp} \Rightarrow S', \gamma'$, to $(N)S \geq S'$ i $\gamma \succ \gamma'$.

gdzie relacja „$\geq$” jest instancjacją sygnatur. W dodatku A do książki [15] zawarty jest dowód istnienia sygnatur głównych w SML. Można pokazać, że również w EML każda sygnatura, dla której można wywieść sygnaturę semantyczną, posiada sygnaturę semantyczną główną.

2.3 Ciekawsze zjawiska

2.3.1 Niedeterminizm

Semantykę statyczną $\mathcal{S}$ nazywamy niedeterministyczną, jeśli istnieje taki element języka *fraza* i takie obiekty semantyczne C, A_1, A_2, że zarówno osąd $C \vdash \mathit{fraza} \Rightarrow A_1$ jak i $C \vdash \mathit{fraza} \Rightarrow A_2$ są wyprowadzalne w $\mathcal{S}$.

Sematyka Statyczna EML jest niedeterministyczna. Stąd na przykład nie każdy typ, jaki można wywieść dla wyrażenia, jest jego typem głównym. W pustym kontekście wyrażeniu:

```
let
   fun S x y z = (x z) (y z)
in
   S
end
```

można przypisać zarówno typ $(\alpha \to \alpha \to \gamma) \to (\alpha \to \alpha) \to \alpha \to \gamma$, jak i typ $(\alpha \to \beta \to \beta) \to (\alpha \to \beta) \to \alpha \to \beta$, ale żaden z nich nie jest równy głównemu typowi tego wyrażenia: $(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$.

2.3.2 Wymuszanie

Ustalmy semantykę statyczną $\mathcal{S}$ i operację syntaktyczną insert, która w pewien sposób z dwóch obiektów syntaktycznych tworzy jeden. Niech T i P będą obiektami syntaktycznymi. Niech C będzie obiektem semantycznym i niech ϕ będzie własnością obiektów semantycznych.

Wtedy mówimy, że P wymusza ϕ w $\mathit{insert}(P, T)$ przy C, gdy jednocześnie

- istnieje obiekt semantyczny A taki, że można wywieść $C \vdash T \Rightarrow A$ oraz nie zachodzi własność ϕ dla A,

- dla każdego A', dla którego można wywieść $C \vdash insert(P,T) \Rightarrow A'$, zachodzi własność ϕ.

Ważnym przypadkiem wymuszania w Semantyce Statycznej EML jest wymuszenie przez aksjomat zrównania typów w sygnaturze. Ilustruje to następujący przykład:

```
sig
    type t
    type u
    val f : t -> u
    axiom forall x => f x == f (f x)
end
```

Niech $insert(P,T)$ będzie powyższą sygnaturą, P występującym w niej aksjomatem, a T sygnaturą bez aksjomatu. Niech C będzie pustą bazą, a ϕ własnością obiektu semantycznego, składającego się w szczególności ze środowiska, mówiącą, że powinno ono przypisywać typom syntaktycznym `t` i `u` ten sam typ semantyczny.

Widać, że obiekt powstały w wyniku głównego wywodu dla T nie spełnia własności ϕ. Natomiast obiekt powstały w wyniku głównego wywodu dla sygnatury $insert(P,T)$, a co za tym idzie również każdy obiekt, jaki można wywieść dla tej sygnatury, spełnia ϕ. Tak więc, w sygnaturze $insert(P,T)$ aksjomat P wymusza zrównanie typów `t` i `u`.

2.4 Analiza statyczna w Extended ML

Semantyka Statyczna EML jest niedeterministyczna. W dodatku niektóre z przesłanek występujących w regułach wyrażają własności postaci „dla każdego wywodu takiego że ϕ zachodzi, ψ musi zachodzić” — dobrymi przykładami są wymagania główności sygnatur pojawiające się w Semantyce Statycznej Modułów.

Z powodu niedeterminizmu i kwantyfikacji po nieskończonych zbiorach, niemożliwa jest bezpośrednia implementacja Semantyki Statycznej EML. Należy przetłumaczyć ją do bardziej „denotacyjnej” postaci (niektórzy wolą nazywać tę postać „deterministyczną”). Dopiero wtedy może ona zostać niemal dosłownie zaimplementowana w języku programowania, na przykład w SML.

Takie przeformułowanie jest nietrywialnym zadaniem. Na szczęście, podobny problem dotyczący SML został rozwiązany już jakiś czas temu. Algorytm Damasa i Milnera [3] jest dobrze znanym szkieletem narzędzia do sprawdzania poprawności typowej programów Jądra SML. Zaś dowód twierdzenia o sygnaturach głównych, z rozdziału A.2 klasycznej pozycji [15], jest przewodnikiem w zmaganiach z językiem Modułów SML. W końcu, sam ML Kit jest bardzo klarowną i skromną reformulacją i implementacją Semantyki Statycznej SML, opartą na wymienionych powyżej ideach.

Moja praca nad przeformułowaniem specyficznych dla EML fragmentów Semantyki Statycznej czasami sprowadzała się do trywialnych rozszerzeń na przykład algorytmu Damasa i Milnera, gdzie jedynym kłopotem było zmaganie z maszynerią systemu ML Kit. Tak było w przypadku prac nad kwantyfikatorami i pozostałymi konstrukcjami specyfikacyjnymi oraz w przypadku analizy innych niż podstawowe postaci aksjomatów.

Czasami jednak rozszerzenie wymagało dowodu poprawności, albo bardzo dobrej znajomości budowy systemu. Było to potrzebne na przykład przy okazji prac nad semantyką funktorów EML, oraz nad ustaleniem poprawności rozwiązań dotyczących rozwikływania przeładowania i zbiorów jawnych zmiennych typowych.

Jeszcze kiedy indziej trzeba było znaleźć zupełnie nowe podejście, udowodnić jego poprawność i zmieścić w ramach systemu ML Kit. Niektóre z takich przedsięwzięć doprowadziły do propozycji zmian w samej definicji EML. Tak było w przypadku prac nad zasięgiem zmiennych typowych, nad warunkami poprawności aksjomatów, czy nad reprezentacją śladów.

Jednak zdecydowanie największe trudności sprawiały elaboracja aksjomatów w sygnaturach oraz zbieranie śladów analizy statycznej. Te właśnie dwa zagadnienia zostaną omówione ze szczególną dokładnością w następnych rozdziałach. Opis innych nietrywialnych problemów, w szczególności tych wymienionych powyżej, znajduje się w pracy [4]. Natomiast wyczerpujący obraz implementacji analizy statycznej EML, w ramach systemu EML Kit, można uzyskać jedynie poprzez lekturę jego kodu źródłowego, wraz z definicją Extended ML.

Część II
Teoria

3 Jak badać poprawność aksjomatów

W Extended ML aksjomaty mogą występować zarówno w strukturach, jak i w sygnaturach. Poprawność statyczną aksjomatów w strukturach można dość prosto sprowadzić do poprawności wyrażenia logicznego, stanowiącego najważniejszą część ciała tych aksjomatów. Z kolei poprawność tego wyrażenia, choć zawierać ono może konstrukcje językowe specyficzne dla EML, łatwo sprawdzić metodami analogicznymi co w SML.

Dla odmiany poprawności statycznej aksjomatów w sygnaturach nie można sprawdzić metodami analizy statycznej SML. Podstawową cechą tych metod jest liniowa widoczność, to znaczy ocenianie poprawności frazy tylko na podstawie informacji, uzyskanych z badania programu do tej frazy włącznie, bez „wybiegania naprzód". Natomiast, jak będzie widać z dalszych rozważań, poprawność aksjomatu może zależeć nie tylko od początkowego fragmentu sygnatury, ale od dowolnie dużej porcji sygnatury za aksjomatem.

3.1 Szkic algorytmu

W tym rozdziale podejmę próbę naszkicowania algorytmu elaboracji sygnatur Extended ML, potencjalnie zawierających aksjomaty.

3.1.1 Sygnatury z aksjomatami

Reguła 65 Semantyki Statycznej EML (rozpatruję tu nieco uproszczoną wersję) opisuje znaczenie syntaktycznej sygnatury, jako element dziedziny semantycznej Sig. Ta reguła jest trochę bardziej złożona niż jej odpowiednik w Sematyce Statycznej SML. Jak można łatwo pokazać, dodatkowa druga i trzecia przesłanka zapewnia, że aksjomaty nie wymuszają zrównania typów, ani uznania typu za równościowy w sygnaturze. To ograniczenie jest narzucone przy użyciu operacji strip, „obdzierającej" sygnaturę z aksjomatów:

$$\frac{\begin{array}{c}(N)S \text{ with } \gamma \text{ principal for } \mathit{sigexp} \text{ in } B \\ \mathrm{strip}(\mathit{sigexp}, \gamma) = (\mathit{sigexp}', \gamma') \\ (N)S \text{ with } \gamma' \text{ principal for } \mathit{sigexp}' \text{ in } B\end{array}}{B \vdash \mathit{sigexp} \Rightarrow (N)S, (N)\gamma} \tag{65}$$

Te specyficzne dla EML, dodatkowe warunki na poprawną postać sygnatur wprowadzone są głównie po to, by programy Extended ML po zakończeniu procesu ich konstruowania, automatycznie stawały się poprawnymi programami języka Standard ML. Inymi słowy, gdy nie ma konstrukcji specyfikacjnych zamiast wykonywalnego kodu, i gdy tylko wszystkie aksjomaty, pełniące już jedynie rolę komentarzy, zostają usunięte, program EML powinien być jednczeście programem SML. Rozpatrzmy przykład:

```
functor F (Arg : sig
                     type t
                     val f : t -> int
                     type u
                     val a : u
                     axiom f a = 0
                 end) :
sig
    val b : int
end =
struct
    val b = Arg.f Arg.a
end
```

Ten program nie jest poprawny, ponieważ aksjomat w sygnaturze wymusza zrównanie typów `t` i `u`. Gdyby nie ograniczenia w regule 65, powyższy program byłby poprawny. Jednak jego wersja „obdarta" z aksjomatów jest niepoprawnym programem języka SML, z powodu aplikacji funkcji `Arg.f`, która ma typ `t -> int`, do argumentu `Arg.a` o typie `u`.

3.1.2 Analiza statyczna modułów w QUASI-EML

Niech QUASI-EML będzie dialektem EML, w którym druga i trzecia przesłanka są nieobecne w regule 65. W QUASI-EML reguła 65 wygląda wobec tego następująco:

$$\frac{(N)S \text{ with } \gamma \text{ principal for } \mathit{sigexp} \text{ in } B}{B \vdash \mathit{sigexp} \Rightarrow (N)S, (N)\gamma}$$

Lemat 3.1. *W QUASI-EML skróty typowe (ang.: type abbreviations) mogą zostać zdefiniowane jako formy pochodne (ang.: derived forms).*

Zamiast formalnego dowodu rozpatrzmy przykładowy program:

```
sig
    type 'a t
    type u = int t (* a type abbreviation *)
    sharing type u = int
end
```

Pochodzi on z książki [15], strona 66, i zawiera skrót typowy w miejscu oznaczonym komentarzem. W języku QUASI-EML można napisać program równoważny temu programowi, ale z formą pochodną w miejscu skrótu typowego:

```
sig
    type 'a t
    type u
    axiom true orelse (forall (c : u, d : int t) => c == d)
    sharing type u = int
end
```

Wniosek 3.2. *W QUASI-EML rekonstrukcja sygnatur głównych jest nierozstrzygalna.*

Powodem jest to, że skróty typowe w połączeniu z postulatami równości typów (ang.: sharing equations) czynią rekonstrukcję sygnatur głównych tak trudną jak unifikacja drugiego rzędu. (W powyższym przykładzie unifikacja drugiego rzędu byłaby potrzebna by zdecydować czy Λ`'a.int`, czy też Λ`'a.'a` jest najogólniejszą funkcją typową, jaka może być przypisana `t`.) Jako że unifkacja drugiego rzędu jest nierozstrzygalna [15], dostajemy nierozstrzygalność rekonstrukcji sygnatur głównych w QUASI-EML.

3.1.3 Naiwna próba rozwiązania problemu

Teraz powróćmy do problemu elaboracji sygnatur języka EML. Patrząc na regułę 65 Semantyki Statycznej EML:

$$\frac{\begin{array}{c}(N)S \text{ with } \gamma \text{ principal for } \mathit{sigexp} \text{ in } B \\ \mathrm{strip}(\mathit{sigexp}, \gamma) = (\mathit{sigexp}', \gamma') \\ (N)S \text{ with } \gamma' \text{ principal for } \mathit{sigexp}' \text{ in } B\end{array}}{B \vdash \mathit{sigexp} \Rightarrow (N)S, (N)\gamma} \tag{65}$$

można dojść do wniosku, że poprawny jest najprostszy sposób jej algorytmizacji (dla uproszczenia nie uwzględniam tutaj śladów):

1. dokonaj elaboracji oryginalnej *sigexp* dostając sygnaturę główną $(N)S$,

2. obedrzyj *sigexp* z aksjomatów dostając *sigexp*$'$,

3. dokonaj elaboracji *sigexp*$'$ uzyskując jej sygnaturę główną $(N')S'$,

4. jeśli $(N)S$ i $(N')S'$ są identyczne wtedy $(N)S$ jest rezultatem, w przeciwnym przypadku sygnatura była niepoprawna.

Niestety implementacja pierwszego kroku jest niemożliwa, gdyż byłaby również rozwiązaniem (nierozstrzygalnego) problemu rekonstrukcji głównych sygnatur w QUASI-EML.

3.1.4 Pocieszający lemat

Na szczęście zachodzi następujący lemat.

Lemat 3.3 (Pocieszający Lemat). *Aksjomaty nie mają wkładu w zawartość środowiska powstałego w wyniku elaboracji sygnatury. Co więcej, obecność i tożsamość śladów powstałych jako rezultat elaboracji aksjomatów nie wpływa na elaborację pozostałych komponentów sygnatury.*

Dowód. Prawdziwość pierwszej części lematu wynika z reguły 74.1 Semantyki Statycznej EML:

$$\frac{B \vdash \mathit{axdesc} \Rightarrow \gamma}{B \vdash \texttt{axiom}\ \mathit{axdesc} \Rightarrow \{\}\ \text{in Env},\ \gamma}$$

Dowód drugiej części to żmudna indukcja po regułach 63–90. ☐

Wniosek 3.4. *Aksjomaty mogą wpływać na elaborację sygnatury tylko przez wymuszanie zrównania typów, albo uznania typu za równościowy, a nie przez nietrywialny wkład, to znaczy wkład do wynikowego środowiska, lub zmianę śladu inną niż proste wstawienie komponentu.*

3.1.5 Poprawne rozwiązanie

Bazując na Lemacie 3.3 i Wniosku 3.4 można teraz sformułować szkic algorytmu implementującego regułę 65:

1. obedrzyj *sigexp* z aksjomatów dostając sigexp',
2. dokonaj elaboracji sigexp' dostając sygnaturę główną $(N')S'$ i ślad γ',
3. sprawdź, że aksjomaty nie wymuszają zrównania typów, ani uznania typu za równościowy w *sigexp*,
4. dokonaj elaboracji aksjomatów zgodnie z regułą 74.1, uzyskując ślady $\gamma_1, \ldots, \gamma_n$,
5. jeśli wszystko przebiegło poprawnie, wynikiem jest $(N')S'$ i dodatkowo γ' ze śladami $\gamma_1, \ldots, \gamma_n$ wstawionymi w odpowiednich miejscach.

(Intuicje prowadzące do podobnego rozwiązania zostały sformułowane przez Stefana Kahrsa w [9].)

Twierdzenie 3.5. *Powyższa procedura jest poprawna.*

Dowód. Ślady elaboracji komponentów sygnatury są składane w „wolny" sposób (patrz np. reguła 81 cytowana w rozdziale 3.2.2 poniżej). Dzięki temu proste wstawienie $\gamma_1, \ldots, \gamma_n$ do γ' w kroku 5 jest wystarczające, by poprawnie zrekonstruować γ, czyli ślad, jaki powstałby w czasie elaboracji *sigexp*. W takim razie, gdy kontrola przeprowadzona w kroku 3 upewnia nas, że aksjomaty nie wymuszają zrównania typów, ani uznania typu za równościowy, wiemy z Wniosku 3.4, iż sygnatura semantyczna $(N')S'$, wynikająca z elaboracji $sigexp'$, wraz ze śladem γ jest główna dla *sigexp*. □

Zanalizujmy kroki naszkicowane powyżej. Krok 1 jest prostą syntaktyczną operacją. Ponieważ $sigexp'$ jest pozbawione aksjomatów, krok 2 jest tak łatwy jak elaboracja sygnatur SML (pomijając zbieranie śladów opisane w rozdziale 4). Wstawianie śladów $\gamma_1, \ldots, \gamma_n$ do γ' w kroku 5 jest jedynie prostym problemem natury technicznej. Pozostaje implementacja kroków 3 i 4.

3.1.6 Badanie dopuszczalności aksjomatów

Załóżmy, że mamy sygnaturę *sigexp* ze znajdującym się wewnątrz niej aksjomatem. Niech $sigexp'$ będzie sygnaturą podobną do *sigexp*, ale z pustą specyfikacją w miejscu, gdzie w *sigexp* znajdował się aksjomat. Załóżmy, że istnieje wywód Der' głównej sygnatury dla $sigexp'$ i niech Der_e będzie jego podwywodem, odpowiadającym pustej specyfikacji. Niech B_e będzie bazą, jaka znajduje się w korzeniu Der_e.

Twierdzenie 3.6. *Aksjomat nie wymusza zrównania typów ani uznania typu za równościowy w sigexp wtedy i tylko wtedy, gdy istnieje derywacja Der_{ax} głównego śladu dla aksjomatu w bazie B_e.*

Dowód ($\Leftarrow$). Przypuśćmy, że istnieje Der_{ax}. Wtedy Der' z Der_{ax} w miejscu Der_e jest szkieletem głównego wywodu Der dla *sigexp*. Aby ten szkielet przekształcić w formalnie poprawne drzewo wywodu, wystarczy na mocy Lematu 3.3 wykonać dodatkowo dwie operacje na każdym przodku węzła Der_e. Pierwsza operacja to poprawne wstawienie śladu wynikającego z Der_{ax} do śladu węzła. Druga to zastąpienie pustej specyfikacji, występującej w węźle, przez aksjomat.

Po tych operacjach Der jest poprawnym wywodem dla *sigexp* i, co więcej, głównym wywodem, gdyż Der' i Der_{ax} są główne. Teraz należy zauważyć, że Der ma tę samą wynikową sygnaturę co Der', a jego ślad jest prostym rozszerzeniem śladu Der'. W takim razie z definicji wymuszania wynika, że aksjomat nie wymusza zrównania typów, ani uznania typu za równościowy w *sigexp*.

(⇒). Skoro *sigexp*′ ma główny wywód i aksjomat niczego nie wymusza, to z Wniosku 3.4 istnieje główny wywód dla *sigexp*. Oczywiście jego podwywód odpowiadający aksjomatowi jest również główny. Pozostaje udowodnić, że ten wywód dla aksjomatu zaczyna się w bazie B_e.

Z Lematu 3.3, aksjomat (który jak wiemy niczego nie wymusza) nie może wpływać na bazy, występujące w głównym wywodzie dla *sigexp*. A więc wywód główny dla aksjomatu jest wykonywany w tej samej bazie, co wywód dla Der_e, czyli w bazie B_e. □

Sprawdzenie, że istnieje wywód główny dla aksjomatu w danej bazie, nie jest bardziej skomplikowane, niż elaboracja aksjomatu występującego w strukturze. W dodatku, jak można było zobaczyć z dowodu, ślady wyprodukowane jako rezultat tej elaboracji są właśnie tymi wymaganymi w kroku 4 z rozdziału 3.1.5.

Teraz jedyną pozostającą trudnością w zaimplemetowaniu kroku 3 z rozdziału 3.1.5, a w ten sposób całego już algorytmu, jest obliczanie baz B_e dla danych aksjomatów.

3.2 Interludium

Tak naprawdę, wiemy już skąd dostać bazę B_e. Jest ona jednym z komponentów śladu γ' wynikającego z elaboracji *sigexp*′ (w kroku 2, rozdział 3.1.5). Istnieją jednak dwa problemy.

Po pierwsze, dla prostoty założyliśmy w rozdziale 2.2.2, że ślady są pełnymi drzewami wywodu. W rzeczywistości, chociaż można zrekonstruować drzewo wywodu dla frazy ze śladu elaboracji tej frazy, to jednak zadanie to jest niewiele łatwiejsze, niż zbudowanie drzewa wywodu dla frazy bez opierania się na śladzie.

Po drugie ślady są niezbędne jedynie Semantyce Weryfikacyjnej EML. Jeśli udało by się znaleźć wygodny sposób obliczania B_e bez śladów, nie było by potrzeby wprowadzania śladów do implementacji Sematyki Statycznej. Może to być o tyle pożyteczne, że implementacja śladów jest cokolwiek skomplikowana i kłopotliwa (patrz rozdział 4).

Aby zobaczyć, jak należy obliczać bazy B_e bez użycia śladów, powinniśmy najpierw zagłębić się w szczegóły elaboracji sygnatur w SML (albo równoważnie, elaboracji sygnatur bez aksjomatów w EML).

3.2.1 Semantyka statyczna sygnatur

Zarówno Semantyka Statyczna Modułów Extended ML jak i Standard ML silnie polega na niedeterminizmie. W ramach budowania wywodu, tożsamość występujących w nim typów i struktur jest „niedeterministycznie zgadywana”. Na przykład w regule 83 (nieco uproszczonej):

$$\frac{\mathit{tyvarseq} = \alpha^{(k)} \qquad \mathrm{arity}\,\theta = k}{C \vdash \mathit{tyvarseq}\ \mathit{tycon} \Rightarrow \{\mathit{tycon} \mapsto (\theta, \{\})\}} \tag{83}$$

θ może być dobrana tak, by spełnione były warunki na równość typów. Albo w regule 63:

$$\frac{B \vdash \mathit{spec} \Rightarrow E, \gamma}{B \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow (m, E), m \cdot \gamma} \tag{63}$$

m może zostać tak dobrane, by spełniona była równość odpowiednich struktur.

3.2.2 Elaboracja sygnatur

Aby pozbyć się niedeterminizmu tkwiącego w Semantyce Statycznej Modułów, należy użyć algorytmu do obliczania sygnatury głównej, przypominającego algorytm do rekonstrukcji typów głównych opisany przez Damasa i Milnera w pracy [3].

Zamiast być poprawnie ustalane od razu, nazwy typów i struktur należących do sygnatury są uznawane za świeże, czyli różne od wszystkich dotychczas spotkanych. Potem, w procesie elaboracji sygnatury, systematycznie gromadzona jest realizacja, czyli (w uproszczeniu) skończony automorfizm nazw typów i struktur.

Za każdym razem, gdy okazuje się, że pewne nazwy powinny być równe, na przykład dlatego, że tak stanowią postulaty równości (ang.: sharing equations), wzbogaca się realizację. Jednocześnie cały czas aplikuje się bieżącą realizację do obiektów semantycznych, co ulepsza pierwsze ostrożne przybliżenia poprzez zrównywanie odpowiednich nazw. Etap „zgadywania” zostaje w ten sposób odłożony do momentu, gdy wiadomo już dokładnie co powinno być odgadnięte.

Proces gromadzenia i aplikowania realizacji szczególnie dobrze widoczny jest w kroku algorytmu dotyczącym reguły 81, opisującej elaborację specyfikacji złożonej:

$$\frac{B \vdash \mathit{spec}_1 \Rightarrow E_1, \gamma_1 \qquad B + E_1 \vdash \mathit{spec}_2 \Rightarrow E_2, \gamma_2}{B \vdash \mathit{spec}_1\ \langle \texttt{;} \rangle\ \mathit{spec}_2 \Rightarrow E_1 + E_2, \gamma_1 \cdot \gamma_2} \tag{81}$$

Oglądając kod źródłowy implementacji tego kroku w systemie EML Kit, szczególną uwagę zwróćmy na realizacje (`rea1`, `rea2`), ich złożenie (`oo`) oraz ich aplikację do obiektów semantycznych (`onB`, `onE`):

```
fun elab_spec (B: Env.Basis, spec: IG.spec) :
    (Stat.Realisation * Env.Env * OG.spec) =
    case spec of
    ...
    (* Sequential specification *)
    | IG.SEQspec(i, spec1, spec2) =>
        let
            val (rea1, E1, out_spec1) = elab_spec(B, spec1)
            val B' = (rea1 onB B) B_plus_E E1
            val (rea2, E2, out_spec2) = elab_spec(B', spec2)
        in
            (rea2 oo rea1,
             (rea2 onE E1) E_plus_E E2,
             OG.SEQspec(okConv i, out_spec1, out_spec2))
        end
    ...
```

3.3 W poszukiwaniu bazy B_e

Znając podstawy algorytmu elaboracji sygnatur pozbawionych aksjomatów, spróbujmy zgadnąć, jak dostać bazę B_e dla danej sygnatury *sigexp* zawierającej aksjomat.

3.3.1 Baza początkowa

Czy powinna to być baza używana przez algorytm, gdy zaczyna on elaborację całego *sigexp*? Nie, i następujący przykład pokazuje, jakie mogą być tego niedobre skutki:

```
signature S1 =
sig
    type t
    axiom forall (a : t) => true
end
```

W tej bazie nie ma typu `t`, a więc aksjomat zostałby niepoprawnie uznany za źle skonstruowany.

3.3.2 Baza bieżąca

Może w takim razie użyć bazy, która pojawia się w miejscu tuż przed aksjomatem? Popatrzmy na przykład:

```
signature S2 =
sig
    type t
    type u
    axiom forall ( a : t, f : u -> bool) => f a
    sharing type t = u
end
```

Aksjomat występujący tutaj nie wymusza zrównania typów, ponieważ w sygnaturze z aksjomatem usuniętym, typy `t` i `u` byłyby zrównane. Niemniej elaboracja aksjomatu w bazie otrzymanej, kiedy algorytm skończy elaborować pierwsze dwie specyfikacje, nie powiedzie się, gdyż świeże semantyczne typy zostaną przypisane `t` i `u`, bez uwzględnienia pojawiającego się dalej postulatu równości typów.

3.3.3 Baza końcowa

Wydaje się więc, że baza zebrana na końcu elaboracji będzie dobrym kandydatem na B_e. Rzeczywiście elaboracja aksjomatu z poprzedniego przykładu przebiegnie tu pomyślnie, gdyż typy przypisane `t` i `u` są w niej zrównane. Niestety, w następującym przykładzie aksjomat, który jest wyraźnie źle skonstruowany, daje się elaborować w takiej bazie:

```
signature S3 =
sig
    axiom forall (a : t) => true
    type t
end
```

3.3.4 Spostrzeżenie

Jak widzimy, w bazie B_e powinny być dokładnie te komponenty, co w bazie tuż sprzed aksjomatu, ale ich tożsamość powinna odzwierciedlać informację zebraną podczas elaboracji całej sygnatury.

Spostrzeżenie 3.7. *Okazuje się, że aby dostać bazę B_e wystarczy zbobyć końcową realizację* `rea`*, wykonując standardowy algorytm elaboracji dla sygnatury sigexp$'$, a następnie zaaplikować* `rea` *do bazy, pobranej gdy algorytm zakończył analizowanie specyfikacji tuż przed aksjomatem.*

Wniosek 3.8. *Możliwe jest otrzymywanie bazy B_e — a co za tym idzie realizowanie kroku 3 z rozdziału 3.1.5 i dzięki temu przeprowadzanie poprawnej elaboracji sygnatur zawierających aksjomaty — bez użycia śladów.*

4 Jak gromadzić ślady

Skomplikowanie operacji zbierania śladów wynika z faktu, że drzewa wywodu, odpowiadające poszczególnym stadiom elaboracji wyrażenia, nie są poddrzewami ostatecznego, poprawnego drzewa wywodu tego wyrażenia. Przed przystąpieniem do rozważań na temat śladów spróbuję wyjaśnić to zjawisko.

4.1 Preludium

Przedstawiane w tym rozdziale przykłady pochodzą z części wspólnej EML i SML. Elaboracją konstrukcji specyfikacyjnych Jądra EML rządzą podobne do opisanych poniżej zasady.

4.1.1 Semantyka statyczna wyrażeń

Podobnie jak w Semantyce Statycznej Modułów (porównaj rozdział 3.2.1), w Semantyce Statycznej Jądra dużą rolę gra niedeterminizm. W wielu węzłach wywodów, typy przypisywane zmiennym są „niedeterministycznie zgadywane”, tak aby mogły być zastosowane reguły, zakładające równość niektórych typów pojawiających się w przesłankach. Niedeterminizm występuje na przykład w regule 35 (uproszczonej):

$$\frac{}{C \vdash \mathit{var} \Rightarrow (\{\mathit{var} \mapsto \tau\}, \tau)} \tag{35}$$

Formalnemu argumentowi funkcji może tu zostać przypisany dowolny typ. Zaś w regule 2 (nieco uproszczonej):

$$\frac{C(\mathit{longvar}) \succ \tau}{C \vdash \mathit{longvar} \Rightarrow \tau} \tag{2}$$

typ zmiennej może być dowolnie wybrany spośród instancji typu tej zmiennej przechowywanego w środowisku.

4.1.2 Elaboracja wyrażeń

Lekarstwem na niemożliwy do bezpośredniego zaimplementowania niedeterminizm jest algorytm Damasa i Milnera [3].

Elaborując wyrażenie przy pomocy tego algorytmu, wywodzi się typy główne podwyrażeń, konkretyzując w razie potrzeby konteksty, w których odbywa się elaboracja. Zamiast „zgadywać” typy, zakłada się że są one najbardziej ogólne, czyli że są świeżymi zmiennymi typowymi. W wypadku reguł, zakładających równość pewnych typów w przesłankach, przeprowadza się unifikację. Aplikacja podstawień wynikłych z unifikacji reprezentuje precyzowanie wiedzy na temat tego, jakie typy powinny były być „zgadnięte" w poprzednich stadiach elaboracji.

W regule typowania aplikacji (nieco uproszczonej):

$$\frac{C \vdash \mathit{exp} \Rightarrow \tau' \rightarrow \tau, \gamma \qquad C \vdash \mathit{atexp} \Rightarrow \tau', \gamma'}{C \vdash \mathit{exp}\ \mathit{atexp} \Rightarrow \tau, \gamma \cdot \gamma'} \tag{10}$$

widać wymaganie równości typu τ' w obu przesłankach. A oto fragment systemu EML Kit odpowiadający tej regule. `S1`, `S2` i `S3` są tu podstawieniami, `oo` ich złożeniem, a `onC` i `on` aplikacją podstawienia odpowiednio do kontekstu i typu:

```
fun elab_exp (C : Environments.Context, exp : IG.exp) :
    (Substitution * StatObject.Type * OG.exp) =
    case exp of
    ...
    (* Application expression *)                      (* rule 10 *)
    | IG.APPexp(i, exp, atexp) =>
        let
            val (S1, tau1, out_exp)   = elab_exp(C, exp)
            val (S2, tau2, out_atexp) = elab_atexp(S1 onC C, atexp)
            val new   = freshType()
            val arrow = StatObject.mkTypeArrow(tau2,new)
            val (S3, i') = Unify(arrow, S2 on tau1, i)
        in
            (S3 oo S2 oo S1, S3 on new,
             OG.APPexp(i', out_exp, out_atexp))
        end
```

Widać, jak zamiast „niedeterministycznego zgadywania" używa się tu świeżego typu (`freshType`). Można zauważyć unifikację (`Unify`) i aplikacje podstawień. Wyraźna jest też rola umowy, że każda elaboracja podwyrażenia daje w wyniku w szczególności wytworzone przez siebie podstawienie.

4.2 Ślady

Ślady służą akumulowaniu obiektów semantycznych pojawiających się podczas analizy statycznej. Do informacji przechowywanej w śladach odwołuje się następnie Semantyka Weryfikacyjna EML. Ślady grają również pewną aktywną rolę w Semantyce Statycznej EML, chociaż postępując z wystarczającą ostrożnością można, jak pokazałem w rozdziale 3.3, poprawnie elaborować programy bez ich użycia.

4.2.1 Ślad elaboracji

W rozdziale 2.2.2 ślady zostały przedstawione, na potrzeby rozważań o analizie statycznej, jako drzewa wywodu osądów semantyki statycznej. Jest to o tyle usprawiedliwione, iż ze śladu można zrekonstruować reprezentowany przez niego wywód, choć jak wspominałem w rozdziale 3.2, nie jest to proste.

Od tej pory, przez ślad analizy statycznej programu P w kontekście C, będę rozumiał główny ślad, jaki według Semantyki Statycznej EML (patrz rozdział 4.2.2) daje się wywieść dla P w C. Natomiast przez ślad elaboracji programu P w kontekście C, będę rozumiał ślad analizy statycznej P w $\varphi(C)$, gdzie φ jest podstawieniem wynikłym z elaboracji P w C.

Na przykład, podczas elaboracji programu:

```
fun f x = (x 5) < 25
```

wyrażenie P, występujące po prawej stronie znaku równości, elaborowane jest w pewnym kontekście C. W tym kontekście zmiennej `x` przypisany jest typ, będący zmienną typową, nazwijmy ją α. Widać, że nie istnieje ślad analizy statycznej P w kontekście C, ponieważ typ przypisany w C zmiennej `x` nie jest typem funkcyjnym, a więc nie da się wywieść typu dla aplikacji `(x 5)`. Natomiast ślad elaboracji P w kontekście C istnieje, gdyż równy jest on śladowi analizy statycznej P w kontekście $\varphi(C)$, gdzie podstawienie φ, wynikające z elaboracji P, przypisuje zmiennej α typ $\mathtt{int} \to \mathtt{int}$.

4.2.2 Ślad jako obiekt semantyczny

Ślady posiadają pewną dodatkową strukturę, wynikającą z możliwości ich domykania. Operator Clos jest używany w kilku miejscach Semantyki Statycznej Jądra EML (reguły 11.1, 11.2, 11.3 i 17), aby domykać ślad względem kontekstu:

$$\mathrm{Clos}_C\gamma = \forall\alpha^{(k)}.\gamma$$

gdzie $\forall$ to kwantyfikator wiążący zmienne typowe, zaś $\alpha^{(k)}$ to zmienne typowe wolne w γ, a nie występujące wolno w C. W śladach Semantyki Statycznej Modułów EML można natomiast wiązać nazwy struktur lub nazwy typów.

A oto pełna definicja śladów Jądra EML:

$$\begin{aligned}
&\mathrm{Trace} = \mathrm{Tree}(\mathrm{SimTrace} \uplus \mathrm{TraceScheme})\\
&\mathrm{SimTrace} = \mathrm{Type} \uplus \mathrm{Env} \uplus (\mathrm{Context} \times \mathrm{Type}) \uplus (\mathrm{Context} \times \mathrm{Env}) \uplus\\
&\qquad (\mathrm{Context} \times \mathrm{TyName}) \uplus \mathrm{TyEnv} \uplus (\mathrm{VarEnv} \times \mathrm{TyRea})\\
&\mathrm{TraceScheme} = \uplus_{k\geq 0}\mathrm{TraceScheme}^{(k)}\\
&\mathrm{TraceScheme}^{(k)} = \mathrm{TyVar}^k \times \mathrm{Trace}
\end{aligned}$$

gdzie Tree(A) jest dziedziną skończonych binarnych drzew elementów z A (stąd też pochodzi spotykany dalej operator składania śladów, pojawiający się we frazach w rodzaju $\gamma \cdot \gamma'$). Pozostałe nazwy oznaczają albo dziedziny obiektów Semantyki Statycznej EML, albo zbiory śladów szczególnej postaci, jak np. TraceScheme, który zawiera ślady będące domknięciami innych śladów.

Prezentacja śladów Modułów EML w definicji Extended ML [10] nie jest w pełni satysfakcjonująca, głównie z punktu widzenia osoby zajmującej się implementacją. W nowej wersji definicji Extended ML [11] autorzy zdefiniowali ślady zgodnie z moją propozycją zawartą w [4]:

$$\begin{aligned}
&\mathrm{Trace} = \mathrm{Tree}(\mathrm{Trace}_{\mathrm{COR}} \uplus \mathrm{SimTrace} \uplus \mathrm{BoundTrace})\\
&\mathrm{BoundTrace} = \mathrm{NameSet} \times \mathrm{Trace}\\
&\mathrm{SimTrace} = \mathrm{StrName} \uplus \mathrm{Rea} \uplus \mathrm{VarEnv} \uplus \mathrm{TyEnv} \uplus \mathrm{Env}
\end{aligned}$$

gdzie $\mathrm{Trace}_{\mathrm{COR}}$ to ślady Jądra EML, opisane powyżej. Zgodnie z tą definicją przeprowadziłem implementację zbierania śladów i ta definicja posłuży mi za podstawę rozważań o śladach, metodzie uzyskiwania śladów elaboracji programów i poprawności tej metody.

4.3 Zbieranie śladów

4.3.1 Lokalna i globalna główność

Zbieranie śladów prostych jest trywialne. Nieco kłopotów nastręczają natomiast ślady, które uzyskuje się przez domknięcie innych śladów. Przykładem może być reguła 17 (tutaj nieco uproszczona):

$$\frac{C \vdash \mathit{valbind} \Rightarrow \mathit{VE}, \gamma \qquad \mathit{VE}' = \mathrm{Clos}_C \mathit{VE}}{C \vdash \texttt{val}\ \mathit{valbind} \Rightarrow \mathit{VE}' \text{ in Env}, \mathrm{Clos}_C \gamma} \tag{17}$$

gdzie śladem jaki należy wywieść dla frazy „`val` *valbind*" jest domknięcie śladu γ, wywiedzionego dla jedynego podtermu tej frazy — „*valbind*". Twierdzenia podobne do poniższego, choć nieco bardziej skomplikowane, można dowieść również dla innych reguł, w których następuje domykanie śladów.

Twierdzenie 4.1. *Niech program P ma postać* `val` P'. *Niech C będzie kontekstem, γ śladem elaboracji P' w kontekście C, zaś φ niech będzie podstawieniem wynikłym z tej elaboracji.*

Wtedy śladem elaboracji P w kontekście C jest $\mathrm{Clos}_{\varphi(C)}\gamma$.

Dowód. Wynikiem elaboracji P w kontekście C jest wywiedzenie głównego obiektu semantycznego dla P w kontekście $\varphi(C)$ według reguły 17, lecz pomijając ślady. Skoro tak, to można zastosować regułę 17 dla programu P i kontekstu $\varphi(C)$ i otrzymać tezę. $\square$

Jednak najciekawszy jest przypadek śladów złożonych. Reguła 8, dotycząca typowania elementów rekordu, jest jedną z reguł, w których ślad programu jest otrzymywany przez złożenie śladów podprogramów:

$$\frac{C \vdash exp \Rightarrow \tau, U, \gamma \qquad \langle C \vdash exprow \Rightarrow \varrho, U', \gamma' \rangle}{C \vdash lab \texttt{ = } exp \ \langle \ \texttt{,} \ exprow \rangle \Rightarrow \{lab \mapsto \tau\}\langle + \ \varrho\rangle, \ U\langle \cup U'\rangle, \ \gamma\langle \cdot \gamma'\rangle} \tag{8}$$

Reguła ta jest bardzo skomplikowana. Zamiast niej, za podstawę naszych rozważań przyjmijmy, nie występującą w definicji EML, regułę typowania pary wyrażeń, będącej uproszczeniem szczególnego przypadku reguły 8:

$$\frac{C \vdash exp \Rightarrow \tau, \gamma \qquad C \vdash exp' \Rightarrow \tau', \gamma'}{C \vdash (exp, exp') \Rightarrow \{1 \mapsto \tau, 2 \mapsto \tau'\}, \gamma \cdot \gamma'}$$

Spostrzeżenie 4.2. *Niech program T ma postać (P, Q). Niech C będzie kontekstem. Wtedy elaboracja T w kontekście C ma dwie główne fazy:*

- *elaborowany jest podprogram P w kontekście C, przy czym otrzymywane jest podstawienie φ_P i ślad tej elaboracji γ_P,*
- *elaborowany jest podprogram Q w kontekście $\varphi_P(C)$, przy czym otrzymywane jest podstawienie φ_Q i ślad tej elaboracji γ_Q.*

Powstaje pytanie jak uzyskać ślad elaboracji T w kontekście C, zakładając że γ_P i γ_Q są poprawnymi śladami elaboracji P i Q? Narzucająca się odpowiedź, że ślad T równy jest $\gamma_P \cdot \gamma_Q$, niestety nie jest prawdziwa. Ślad γ_P uzyskiwany jest w potencjalnie niegłównym dla T kontekście, więc mimo iż γ_P jest główny dla P, może być zbyt ogólny, by stać się częścią śladu T. Ilustruje to następujący prosty przykład:

```
fun g fib = (fib, fib 5)
```

gdzie P to `fib`, zaś Q to `fib 5`. W γ_P zmienna `fib` nie ma przypisanego typu funkcyjnego, mimo że w śladzie T zmienna `fib` może mieć jedynie typ funkcyjny, gdyż z elaboracji Q wynika, że `fib` jest funkcją (w szczególności φ_Q przypisuje zmiennej typowej, która jest typem `fib`, typ funkcyjny).

Twierdzenie 4.3. *Przyjmijmy oznaczenia ze Spostrzeżenia 4.2. Wtedy ślad elaboracji T w kontekście C równy jest $\widehat{\gamma} \cdot \gamma_Q$, gdzie $\widehat{\gamma}$ jest śladem elaboracji P w kontekście $\varphi_Q(\varphi_P(C))$.*

Dowód. Ponieważ γ_Q jest śladem elaboracji Q w kontekście $\varphi_P(C)$, więc z definicji śladu elaboracji, γ_Q jest śladem analizy statycznej Q w kontekście $\varphi_Q(\varphi_P(C))$.

Z drugiej strony wynikiem elaboracji T w kontekście C jest wywiedzenie głównego obiektu semantycznego dla T w kontekście $\varphi_Q(\varphi_P(C))$, według reguły typowania pary wyrażeń, ale z pominięciem śladów. Ale w takim razie z reguły typowania pary wyrażeń wynika teza. $\square$

Odpowiedniki tego prostego twierdzenia dla innych reguł, w których następuje złożenie śladów, są również dość oczywiste. Co więcej podobne twierdzenia można udowodnić również dla języka Modułów EML, z tym że zamiast podstawień będą w nich występować realizacje.

Niestety bezpośrednie zastosowanie w algorytmie zbierania śladów procedury jaka wynika z Twierdzenia 4.3, prowadzi do podwójnej elaboracji P. Pierwszy raz w kontekście C, aby uzyskać podstawienie φ_P i drugi raz w kontekście $\varphi_Q(\varphi_P(C))$, aby uzyskać ślad $\widehat{\gamma}$.

4.3.2 Konkretyzacja śladu

Jednym z najbardziej niepokojących zjawisk, występujących podczas budowania śladu programu ze śladów podprogramów, jest to, iż ślady podprogramów czasami powstają na skutek domykania niegłównych dla całego programu śladów, względem również niegłównych kontekstów. Dowód następującego twierdzenia pokazuje, że zjawisko to nie musi być groźne.

Twierdzenie 4.4. *Niech P będzie programem napisanym w języku Jądra Extended ML. Niech D będzie kontekstem. Niech γ_P będzie śladem elaboracji P w kontekście D. Niech φ będzie podstawieniem.*

Wtedy ślad elaboracji P w kontekście $\varphi(D)$ równy jest $\varphi(\gamma_P)$.

Dowód. Indukcja po strukturze γ_P. Przypadki bazowe i przypadek składania śladów wynikają z podstawowych własności procesu elaboracji. Pozostaje przypadek, gdy $\gamma_P \in$ TraceScheme. Wtedy, w ostatnim kroku elaboracji P, musiał występować pewien ślad γ, domykany względem pewnego kontekstu C, przy czym $\gamma_P = \mathrm{Clos}_C\gamma$.

Niech P' będzie podprogramem P odpowiadającym γ. Dla uproszczenia załóżmy, że P jest postaci `val` P', a więc elaboracja P' jako części P odbyła się w kontekście D. Niech ta elaboracja daje podstawienie $\varphi_{P'}$. W takim razie z Twierdzenia 4.1 wiemy, że $C = \varphi_{P'}(D)$.

W razie potrzeby można przemianować w γ, P' i P zmienne typowe występujące wolno w γ, a nie występujące wolno w C tak, aby znalazły się one poza zasięgiem φ. Z uzyskanej z dalszego rozumowania tezy, że dla przemianowanego P i γ, ślad elaboracji P w kontekście $\varphi(D)$ równy jest $\varphi(\mathrm{Clos}_C\gamma)$, łatwo dostać tezę dla oryginalnych P i γ.

Stosując hipotezę indukcyjną do P', D, γ i φ dostaję, że ślad elaboracji P' w kontekście $\varphi(D)$ równy jest $\varphi(\gamma)$. Niech ta elaboracja daje podstawienie $\phi_{P'}$. Z Twierdzenia 4.1 otrzymuję, że ślad elaboracji P w kontekście $\varphi(D)$ równy jest $\mathrm{Clos}_{\phi_{P'}(\varphi(D))}\varphi(\gamma)$. Ale można pokazać, że

$$\mathrm{Clos}_{\phi_{P'}(\varphi(D))}\varphi(\gamma) = \mathrm{Clos}_{\varphi(\varphi_{P'}(D))}\varphi(\gamma) = \mathrm{Clos}_{\varphi(C)}\varphi(\gamma)$$

Pierwszą z tych równości uzyskuje się przez żmudną indukcję po strukturze P', korzystając z faktu że odpowiednie zmienne typowe znajdują się poza zasięgiem φ (być może na skutek przemianowania zmiennych), druga wynika z tego, że $C = \varphi_{P'}(D)$.

Pozostaje udowodnić, że

$$\mathrm{Clos}_{\varphi(C)}\varphi(\gamma) = \varphi(\mathrm{Clos}_C\gamma)$$

Dla uproszczenia założę do końca dowodu, iż w γ występuje dokładnie jedna typowa zmienna wolna — α. Łatwo to rozumowanie przenieść na przypadki z inną liczbą zmiennych.

Rozpatrzę następujące możliwości:

- α występuje wolno w kontekście C. Wtedy $\mathrm{Clos}_C\gamma = \gamma$, a stąd

  $$\varphi(\mathrm{Clos}_C\gamma) = \varphi(\gamma)$$

  Z drugiej zaś strony jedyne zmienne wolne $\varphi(\gamma)$ to zmienne wolne występujące w $\varphi(\alpha)$, co wraz z faktem, iż w C występuje wolno α, czyli w $\varphi(C)$ występują wolno zmienne wolne termu $\varphi(\alpha)$, daje

  $$\mathrm{Clos}_{\varphi(C)}\varphi(\gamma) = \varphi(\gamma)$$

- α nie występuje wolno w kontekście C. Wtedy $\mathrm{Clos}_C\gamma = \forall\alpha.\gamma$ a ponieważ $\forall\alpha.\gamma$ nie posiada zmiennych wolnych, dostaję

  $$\varphi(\mathrm{Clos}_C\gamma) = \forall\alpha.\gamma$$

  Z drugiej strony ponieważ α jest poza zasięgiem φ, to $\varphi(\gamma) = \gamma$ oraz $\varphi(C)$ nie zawiera α, a stąd

  $$\mathrm{Clos}_{\varphi(C)}\varphi(\gamma) = \forall\alpha.\gamma$$

□

Teraz wreszcie możemy pokazać, w jaki sposób z poprawnych śladów elaboracji podprogramów można zbudować ślad elaboracji ich pary.

Twierdzenie 4.5. *Niech program T ma postać (P, Q). Niech C będzie kontekstem w którym przeprowadzana jest elaboracja T. Niech φ_P będzie podstawieniem, wynikłym z elaboracji P jako części T, i niech γ_P będzie śladem tej elaboracji. (Zauważmy, że P jest elaborowane w kontekście C.) Niech φ_Q będzie podstawieniem, wynikłym z elaboracji Q jako części T, i niech γ_Q będzie śladem tej elaboracji.*

Wtedy $\varphi_Q(\gamma_P) \cdot \gamma_Q$ jest śladem elaboracji T.

Dowód. Z Twierdzenia 4.3 wiemy, iż ślad elaboracji T w kontekście C równy jest $\widehat{\gamma} \cdot \gamma_Q$, gdzie $\widehat{\gamma}$ jest śladem elaboracji P w kontekście $\varphi_Q(\varphi_P(C))$.

Zastosujmy Twierdzenie 4.4 z $\varphi = \varphi_P$ i $D = C$. Dostajemy, że śladem elaboracji P w kontekście $\varphi_P(C)$ jest $\varphi_P(\gamma_P)$. Ale ponieważ φ_P zawiera tylko te konkretyzacje typów, które są niezbędne by udała się elaboracja P w C (porównaj rozdział 4.1.2), a γ_P jest śladem tej właśnie elaboracji, mamy $\varphi_P(\gamma_P) = \gamma_P$. A więc ślad elaboracji P w kontekście $\varphi_P(C)$ równy jest γ_P.

Jeśli teraz postawimy $\varphi = \varphi_Q$ i $D = \varphi_P(C)$ to ponownie spełnione zostaną założenia Twierdzenia 4.4, i uzyskamy $\widehat{\gamma} = \varphi(\gamma_P) = \varphi_Q(\gamma_P)$. □

Podobne twierdzenia można udowodnić dla wszystich postaci programów Jądra EML, w których składane są ślady i może występować zjawisko domykania niedoprecyzowanego śladu, względem niegłównego kontekstu. Co więcej, twierdzenia podobne do 4.4 i 4.5 można udowodnić również w przypadku języka Modułów, gdy rolę podstawień spełniają realizacje.

Wniosek 4.6. *Zbierania śladów można dokonywać wraz z elaboracją, przy pomocy jedoprzebiegowego algorytmu.*

Część III
Implementacja

W tej części wielokrotnie odwołuję się do kodu źródłowego systemu EML Kit. Staram się pokazywać przykładowy kod w postaci jak najmniej zmienionej i pociętej. Jednocześnie zdaję sobie sprawę, że nie jestem w stanie zastąpić narracją przeplataną przykładami, lektury pełnych źródeł i eksperymentów przy użyciu pracującego systemu. Dlatego też w dodatku A podaję informacje na temat dostępności najnowszego wydania systemu EML Kit, zawierającego w szczególności ponad półtora megabajta kodu źródłowego.

5 Aksjomaty w sygnaturach

Elaboracja sygnatur zawierających aksjomaty zbudowana została na podstawie algorytmu naszkicowanego w rozdziale 3.1.5, korzystając z wniosków sformułowanych w rozdziale 3.1.6 i 3.3.4 oraz bazując na elaboracji sygnatur SML zaimplementowanej w systemie ML Kit.

Przypomnę szkic algorytmu z rozdziału 3.1.5:

1. obedrzyj *sigexp* z aksjomatów dostając *sigexp*′,
2. dokonaj elaboracji *sigexp*′ dostając sygnaturę główną $(N')S'$ i ślad γ',
3. sprawdź, że aksjomaty nie wymuszają zrównania typów, ani uznania typu za równościowy w *sigexp*,
4. dokonaj elaboracji aksjomatów zgodnie z regułą 74.1, uzyskując ślady $\gamma_1, \ldots, \gamma_n$,
5. jeśli wszystko przebiegło poprawnie, wynikiem jest $(N')S'$ i dodatkowo γ' ze śladami $\gamma_1, \ldots, \gamma_n$ wstawionymi w odpowiednich miejscach.

Widać, że rezultatem kroków 1 i 2 ma być semantyczna sygnatura $(N')S'$, główna dla *sigexp*′ oraz ślad γ'. Ze Spostrzeżenia 3.7 wiadomo, że przydatna byłaby również realizacja φ, będąca wynikiem elaboracji wykonanej w kroku 2.

Algorytm elaboracji sygnatur w systemie EML Kit musi być przygotowany na możliwość znalezienia w analizowanej przez siebie sygnaturze aksjomatów. Jeśli nie zażąda się inaczej, to napotykane aksjomaty są elaborowane w bieżącej bazie, a ewentualne błędy ich elaboracji są zapisywane w drzewie składni abstrakcyjnej i pod innymi względami ignorowane. Dzięki temu mogę uzyskać $(N')S'$ oraz realizację φ elaborując oryginalne *sigexp* zamiast jej obdartej wersji. Ponieważ ślad takiej elaboracji jest tworem sztucznym i nieprzydatnym, nie zachowam go, a zbieranie γ' odłożę na później.

Dokonuje się to w pierwszym przebiegu algorytmu, którego wywołanie można wyróżnić w kodzie po tym, że dodatkowy parametr elaboracji ma wartość `FIRST_PASS`:

```
fun elab_psigexp (B: Env.Basis, psigexp: IG.psigexp)
    : (Stat.Sig * OG.psigexp) =
  let
    val IG.PRINCIPpsigexp(i, sigexp) = psigexp
    val A = Env.mkAssembly B
    val NofB = Env.N_of_B B
    val (rea, _, S_first, _) =
        (StrId.backup_state();
        elab_sigexp'(B, sigexp, A, FIRST_PASS))
    val (_, rea1) = Stat.equality_principal(NofB, S_first)
    val (_, _, S, out_sigexp) =
        (StrId.restore_state();
        elab_sigexp'(B, sigexp, A, SECOND_PASS(rea1 oo rea)))
  in
    ... Stat.closeStr(NofB, S) ...
  end
```

Historia i znaczenie `StrId.backup_state` oraz `StrId.restore_state` pojawiających się powyżej, opisana jest w rozdziale 8. Znaczenie `mkAssembly` i wielu rozbudowanych poleceń znajdujących się w wykropkowanej części kodu, jest takie samo jak w przypadku elaboracji sygnatur SML, opisanej w dokumentacji systemu ML Kit [2].

Po pierwszym przebiegu tworzona jest realizacja, której aplikacja do sygnatury uczyni ją równościowo główną (ang.: equality principal). Równościowa główność jest pojęciem, które dla uproszczenia pominęliśmy w rozważaniach teoretycznych i do którego również tutaj nie będziemy już więcej wracali, gdyż nie przeszkadza ono w poprawnej elaboracji aksjomatów, a jego implementacja przebiega identycznie jak w przypadku SML.

Teraz należy wykonać krok 3, czyli sprawdzić czy aksjomaty czegoś nie wymuszają. Na mocy Twierdzenia 3.6 wystarczy w tym celu dokonać elaboracji aksjomatów w odpowiadających im bazach B_e. Najłatwiejszym sposobem, aby to osiągnąć jest elaboracja *sigexp* po raz drugi, z dodatkowym parametrem, którym jest otrzymana w pierwszym przebiegu realizacja φ. Kiedy podczas tej elaboracji napotykany jest aksjomat, jest on elaborowany w bieżącej bazie zmodyfikowanej realizacją φ. Ta baza jest, na mocy Spostrzeżenia 3.7, równa bazie B_e aksjomatu:

```
fun elab_spec (B: Env.Basis, spec: IG.spec, A: Env.Assembly, p : pass):
   (Stat.Realisation * Env.Assembly * Env.Env * OG.spec) =
  case spec of
  ...
   (* Axiom specification *)
  | IG.AXIOMspec(i, axdesc) =>
      let
        val (out_axdesc) =
          case p
            of FIRST_PASS => elab_axdesc(B, axdesc)
             | (SECOND_PASS(rea)) => elab_axdesc(rea onB B, axdesc)
      in
        (Stat.Id, Env.emptyA, Env.emptyE, OG.AXIOMspec(okConv i, out_axdesc))
      end
```

Podczas drugiego przebiegu elboracji *sigexp* następuje zbieranie śladu, nazwijmy go γ, a w nim zbierane są również ślady $\gamma_1, \ldots, \gamma_n$ zgodnie z krokiem 4. Co więcej, jak łatwo zobaczyć, γ równy jest śladowi γ', z $\gamma_1, \ldots, \gamma_n$ wstawionymi w odpowiednich miejscach, co oznacza, że elaboracja została zakończona.

6 Ślady języka Modułów

W systemie EML Kit ślady są gromadzone w drzewie składni abstrakcyjnej programu. Każdy węzeł takiego drzewa zawiera pole, w którym przechowywane są dodatkowe informacje. W wypadku programów, które przeszły już proces elaboracji, typ tego pola ma następującą definicję:

```
datatype PostElabGrammarInfo =
  POST_ELAB_GRAMMAR_INFO of {preElabGrammarInfo: PreElabGrammarInfo,
                             errorInfo: ErrorInfo Option,
                             typeInfo: TypeInfo Option,
                             overloadingInfo: OverloadingInfo Option,
                             trace : Trace Option}
```

Przy czym pojawiający się tu i w innych miejscach typ `Option`, jest zdefiniowany następująco:

```
datatype 'a Option = None | Some of 'a
```

Ściśle mówiąc, w węzłach nie ma prawdziwych śladów, czyli obiektów budowanych przy pomocy operacji właściwych dziedzinom postaci Tree(A) (patrz rozdział 4.2.2). Zamiast tego i w pewnym sensie równoważnie, w węzłach znajdują się komponenty śladów. W ten sposób ślad podprogramu, mającego swój korzeń w danym węźle, reprezentowany jest przez całe poddrzewo składni abstrakcyjnej, zaczynające się w tym węźle i zawierające składniki tego śladu.

Ślady języka Modułów EML składają się ze śladów języka Jądra EML (tu oznaczonych `TraceCOR` — ich zbieraniu poświęcony jest rozdział 7) oraz śladów prostych (`SimTrace`). Ponieważ nigdy nie zdarza się jednoczesne dodanie komponentu do śladu i związanie w nim nazw, zjawisko związywania nazw w śladach może być reprezentowane jako trzeci rodzaj składnika śladu (`BoundTrace`):

```
datatype Trace =
    TRACE_COR of TraceCOR
  | SIM_TRACE of SimTrace
  | BOUND_TRACE of BoundTrace
and BoundTrace =
    BOUND of NameSet
and SimTrace =
    STRNAME of StrName
  | REA of Rea
  | VARENV of VarEnv
  | TYENV of TyEnv
  | ENV of Env
```

Dodawania śladów można teraz dokonywać przy pomocy prostych funkcji pomocniczych:

```
(* Auxiliary functions for trace collection *)
fun addTrace (tr : Trace.Trace) (i : PostElabGrammarInfo) =
    GrammarInfo.addPostElabTrace(i, tr)
fun addCoreTrace (tr : Trace.TraceCOR) (i : PostElabGrammarInfo) =
    addTrace (Trace.TRACE_COR(tr)) i
fun addSimTrace (sim : Trace.SimTrace) (i : PostElabGrammarInfo) =
    addTrace (Trace.SIM_TRACE(sim)) i
fun addBoundTrace (bound : Trace.BoundTrace) (i : PostElabGrammarInfo) =
    addTrace (Trace.BOUND_TRACE(bound)) i
```

Elaboracja sygnatur odbywa się w dwóch przebiegach, po to by poprawnie traktować aksjomaty (patrz rozdział 5). Mechanizmy, skonstruowane na potrzeby tej elaboracji, pozwalają również bez dodatkowych wysiłków opanować niedeterminizm, przeszkadzający w zbieraniu śladów języka Modułów.

Poprawne musi być jedynie zbieranie śladów podczas drugiego przebiegu elaboracji. Ale wtedy właśnie dostępna jest już wynikowa realizacja całego programu, dzięki której można „zgadywać” ostateczną konkretyzację pojawiających się obiektów semantycznych. Dzięki temu nie ma potrzeby uściślania zebranych uprzednio śladów, a do każdego komponentu aplikowana jest tylko jedna realizacja i tylko jeden raz.

Ilustrują to dwie reguły Semantyki Statycznej Modułów EML, które są źródłami niedetermizmu w wyborze własności podstruktur pojawiających się w sygnaturach:

$$\frac{B \vdash \mathit{spec} \Rightarrow E, \gamma}{B \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow (m, E), m \cdot \gamma} \tag{63}$$

$$\frac{B(\mathit{sigid}) \geq_{\varphi} S}{B \vdash \mathit{sigid} \Rightarrow S, \varphi} \tag{64}$$

oraz implementacja kroku elaboracji odpowiadającego tym regułom:

```
fun elab_sigexp' (B: Env.Basis, sigexp: IG.sigexp, A: Env.Assembly, p : pass):
  (Stat.Realisation * Env.Assembly * Stat.Str * OG.sigexp) =
   case sigexp of
     (* Generative *)
     IG.SIGsigexp(i, spec) =>
       let
         val (rea, A1, E, out_spec) = elab_spec(B, spec, A, p)
         val m = Stat.freshStrName()
         val S  = Stat.mkStr(m, E)
         val A2 = A1 union Env.singleA_Str(S)
       in
         (rea, A2, S, OG.SIGsigexp(addSimTrace
          (Trace.STRNAME(case p
                         of FIRST_PASS => m
                          | (SECOND_PASS(rea1)) => M.onStrName rea1 m))
          (okConv i), out_spec))
       end
     (* Signature identifier *)
   | IG.SIGIDsigexp(i, sigid) =>
       case Env.lookup_sigid(B, sigid)
         of Some sigma =>
              let
                val (S, A', rea) = Env.Sig_instance_for_traces sigma
              in
                (Stat.Id, A', S, OG.SIGIDsigexp(addSimTrace
                 (Trace.REA(case p
                            of FIRST_PASS => rea
                             | (SECOND_PASS(rea1)) => rea1 oo rea))
                (okConv i), sigid))
              end
          | None => ... ErrorInfo.LOOKUP_SIGID sigid ...
```

7 Ślady języka Jądra

Z Wniosku 4.6 wynika, że zbierania śladów języka Jądra EML można dokonywać, dodając do bieżących śladów nowe komponenty i ciągle konkretyzując stare, poprzez aplikowanie podstawień.

Inną możliwością jest wstrzymanie się od poprawiania śladów podczas elaboracji, a zamiast tego zaaplikowanie końcowego podstawienia do śladu otrzymanego na końcu procesu. Aby dowieść, że jest to poprawne, potrzebne jest twierdzenie podobne do Lematu 3.3 z rozdziału 3.1.4. Dowód lematu następuje poprzez indukcję po wszystkich regułach Semantyki Statycznej Jądra EML.

Rzeczą, która wymaga ostrożności jest to, iż skoro wstrzymujemy się od konkretyzacji śladów, nie powinniśmy ich również za wcześnie domykać. W związku z tym implementacja śladów elaboracji Jądra EML zawiera dwie formy TraceScheme. Jedna z nich to lista zmiennych, reprezentująca zmienne znajdujące się pod kwantyfikatorem. Druga to kontekst, jaki występował podczas elaboracji, gdy pojawiła się potrzeba domknięcia śladu, ale domknięcie zostało odłożone na później:

```
datatype Trace =
    TRACE of SchemeTrace Option * SimTrace Option
and SimTrace =
    TYPE of Type
  | ENV of Env
  | CONTEXTxTYPE of Context * Type
  | CONTEXTxENV of Context * Env
  | TYENV of TyEnv
  | VARENVxTYREA of VarEnv * TyRea
  | CONTEXTxTYNAME of Context * TyName
and SchemeTrace =
    SCHEME_C of Context
  | TYVARS of TyVar list
```

Zauważmy, że ponieważ czasami jednocześnie dodaje się komponent do śladu i domyka cały ślad, typ `Trace` nie może być sumą rozłączną możliwych postaci komponentów śladu, jak było to w rozdziale 6.

Podczas zbierania śladów, każda napotkana sytuacja, w której potrzebne by było domknięcie śladu, powoduje umieszczenie bieżącego kontekstu w komponencie `SchemeTrace`. Natomiast po zebraniu całego śladu elaboracji, wykonywana jest operacja `process_core_trace_and_tvs`, która aplikuje wynikowe podstawienie elaboracji do każdego atomowego śladu, dokonując jednocześnie właściwego domykania na podstawie zgromadzonych kontekstów:

```
fun onScheme (S, SCHEME_C(C)) = SCHEME_C(Environments.onC (S, C))
  | onScheme (S, TYVARS(_))   = Crash.impossible "CoreTrace.onScheme"
fun onScheme_opt (S, Some(scheme)) = Some(onScheme (S, scheme))
  | onScheme_opt (S, None)          = None
fun onSim (S, TYPE(tau)) = TYPE(Environments.on (S, tau))
  | onSim (S, ENV(E)) = ENV(Environments.onE (S, E))
  | onSim (S, CONTEXTxTYPE(C, tau)) = CONTEXTxTYPE(Environments.onC (S, C),
                                                   Environments.on (S, tau))
  | onSim (S, CONTEXTxENV(C, E)) = CONTEXTxENV(Environments.onC (S, C),
                                               Environments.onE (S, E))
  | onSim (S, TYENV(TE)) = TYENV(TE)
  | onSim (S, VARENVxTYREA(VE, phi_Ty)) = VARENVxTYREA(VE, phi_Ty)
  | onSim (S, CONTEXTxTYNAME(C, t)) = CONTEXTxTYNAME(Environments.onC (S, C), t)
fun onSim_opt (S, Some(sim)) = Some(onSim (S, sim))
  | onSim_opt (S, None)      = None
fun onCoreTrace (S, TRACE(scheme_opt, sim_opt)) =
    TRACE(onScheme_opt (S, scheme_opt), onSim_opt (S, sim_opt))
fun tyvars_sim (TYPE(tau)) = Environments.tyvarsTy tau
  | tyvars_sim (ENV(E)) = Environments.tyvarsE E
  | tyvars_sim (CONTEXTxTYPE(C, tau)) = (Environments.tyvarsC C)
                                        @ (Environments.tyvarsTy tau)
  | tyvars_sim (CONTEXTxENV(C, E)) = (Environments.tyvarsC C)
                                     @ (Environments.tyvarsE E)
  | tyvars_sim (TYENV(TE)) = []
  | tyvars_sim (VARENVxTYREA(VE, phi_Ty)) = []
  | tyvars_sim (CONTEXTxTYNAME(C, t)) = Environments.tyvarsC C
fun tyvars_sim_opt (Some(sim)) = tyvars_sim sim
  | tyvars_sim_opt None = []
fun process_core_trace_and_tvs (core_trace, tvs) S =
  let
    val core_trace' as (TRACE(scheme_opt, sim_opt)) = onCoreTrace (S, core_trace)
    val sim_opt_tvs = tyvars_sim_opt sim_opt
    val free_tvs = sim_opt_tvs @ tvs
  in
    case scheme_opt
      of None => (core_trace', free_tvs)
       | Some(SCHEME_C(C)) =>
          let val tvs_to_bind = minus (free_tvs, Environments.tyvarsC C)
          in (TRACE(Some(TYVARS(tvs_to_bind)), sim_opt),
              minus (free_tvs, tvs_to_bind))
          end
       | Some(TYVARS(_)) =>
          Crash.impossible "CoreTrace.process_core_trace_and_tvs"
  end
```

8 Anegdota

Kiedy skończyłem kodowanie algorytmu elaboracji aksjomatów w sygnaturach, przez kilka dni byłem pewien, mogłem dowieść, że jest on poprawny — ale algorytm uparcie produkował dziwne i niewyjaśnialne rezultaty. Byłem nieco zirytowany. Jednak to, co uczyniło mnie naprawdę złym, było odkrycie, że winowajcą są efekty uboczne. Jestem ich fanatycznym przeciwnikiem i nie przyjmowałem do wiadomości, że autorzy tak eleganckiego systemu, jak ML Kit, mogli zniżyć się do ich stosowania. Tymczasem okazało się, że w systemie ML Kit, a co za tym idzie również w systemie EML Kit, efekty uboczne są używane. Między innymi po to, aby uzyskiwać świeże nazwy typów i struktur podczas elaboracji sygnatur.

Rozwiązałem problem w raczej brutalny sposób, nie przebierając w środkach, jak można zobaczyć w kodzie funkcji `elab_psigexp` powyżej oraz w odpowiedzialnym za szkody module poniżej:

```
functor Timestamp(): TIMESTAMP =
struct
  type stamp = int
  val r = ref 0
  fun new() = (r := !r + 1; !r)
  fun print i = "$" ^ Int.string i
(* mikon#1.44 *)
  val backup = ref 0
  fun backup_state() = backup := !r
  fun restore_state() = r := !backup
(* end mikon#1.44 *)
end;
```

Część IV
Podsumowanie

Formalizm Extended ML został w 1994 roku opisany w swojej definicji [10]. Od tego czasu Extended ML wykorzystywany jest do tworzenia oprogramowania i do celów edukacyjnych. Trwają prace nad teorią dowodu EML. Rozwijane są prototypowe narzędzia wspomagające proces konstrukcji oprogramowania. Ciągle ulepszana jest i poprawiana sama definicja formalizmu, w czym autor tej pracy chlubi się mieć swój malutki udział.

Od samego początku wyraźne jest znaczenie analizy statycznej EML. Zdefiniowana ona została w części definicji zwanej Semantyką Statyczną. Jej wstępne implementacje były jednym z najważniejszych elementów kolejnych wersji systemu EML Kit, stanowiąc o jego użyteczności w praktycznych zastosowaniach. Pobieżny opis ostatecznej implementacji zawartej w systemie EML Kit 1.0, wraz z dokładnym omówieniem niektórych szczegółów, został zamieszczony w raporcie na temat systemu EML Kit [4]. Ponadto Stefan Kahrs poświęcił pracę [9] opisowi różnic między analizą statyczną Extended ML i analizą statyczną Standard ML. A ta z kolei rozpatrywana była w wielu pozycjach, na przykład w [15] czy w [14].

Bieżąca aktywność naukowa wokół Extended ML skupiona jest na Semantyce Weryfikacyjnej EML i zagadnieniach, które można by nazwać jej „implementacją". Z kolei Semantyka Weryfikacyjna często odwołuje się do Semantyki Statycznej, na przykład w celu wyrażenia własności posiadania typu. Co więcej, Semantyka Statyczna jest obciążona odpowiedzialnością za zbieranie różnorakich informacji na potrzeby Semantyki Weryfikacyjnej. Wszystko to sprawia, iż niezbędna jest pełna i godna zaufania implementacja Semantyki Statycznej EML, co stało się motywacją do opisanych tu przedsięwzięć.

Niniejszą pracą mam nadzieję zamknąć temat własności analizy statycznej EML i metod jej przeprowadzania. W pracy tej zdaję również sprawę z najciekawszych aspektów jej implementacji w ramach systemu EML Kit. Szczególną uwagą obdarzyłem dwa problemy, powstające podczas sprawdzania poprawności typowej programów napisanych w języku Extended ML. Pierwszy problem to analiza statyczna sygnatur, w których mogą jednocześnie występować aksjomaty i postulaty równości typów. Drugi to zbieranie śladów podczas analizy statycznej. Wskazałem źródła groźnych błędów, jakie mogą być popełnione jeśli te problemy nie są potraktowane z wystarczającą ostrożnością. Na tym tle naszkicowałem poprawne rozwiązania, wspominając również o pewnych ich wariantach. Udowodniłem także szereg własności Semantyki Statycznej EML, które potem posłużyły mi do dowodu poprawności podanych algorytmów. Na końcu przedstawiłem rzeczywistą implementację opisanych rozwiązań, w tej postaci w jakiej jest ona częścią systemu EML Kit.

Podziękowania

Szczególnie serdecznie chciałbym podziękować Andrzejowi Tarleckiemu, mojemu promotorowi.

Pozostałym autorom Extended ML: Stefanowi Kahrsowi i Donowi Sannelli, jestem wdzięczny za życzliwą uwagę i czas jaki mi poświęcili. Z wdzięcznością pozdrawiam przyjaciół, z którymi wspólnie pracowałem nad systemem EML Kit: Marcina Jurdzińskiego, Sławomira Leszczyńskiego, Roberta Marona i Aleksego Schuberta. Za dobre słowo dziękuję Marcinowi Benke, Sławomirowi Białeckiemu, Sławomirowi Lasocie i Michałowi Grabowskiemu.

Dodatki

A System EML Kit

EML Kit jest dostępny w sieci Internet. Poza wydaniem kodu źródłowego systemu EML Kit wersji 1.0, istnieje również wydanie zawierające binaria dla systemu Solaris na architekturze Sun SPARC, oraz wydanie zawierające binaria dla systemu Linux na architekture i386.

Wszystkie te zestawy są dostępne przez anonimowe ftp z:

`ftp://zls.mimuw.edu.pl/pub/mikon/EMLKit`

Można je również znaleźć na stronie domowej WWW systemu EML Kit:

`http://zls.mimuw.edu.pl/~mikon/ftp/EMLKit/README.html`

EML Kit ma następujący copyright:

EML Kit został wydany na warunkach opisanych w licencji GNU GPL, co wyjaśniają i przypieczętowują następujące informacje w języku angielskim załączane do każdego wydania systemu:

Autorzy systemu EML Kit są wdzięczni za wszelką pomoc. Prace nad systemem EML Kit były częściowo wspierane przez następujące granty KBN (Komitetu Badań Naukowych):

- „Formal Methods of Software Development”

  `http://wwwat.mimuw.edu.pl/~tarlecki/kbn93grant/index.html`

- „Logical Aspects of Software Specification and Development Methods”

  `http://wwwat.mimuw.edu.pl/~tarlecki/kbn96grant/index.html`

B Obiekty semantyczne

Dziedziny prostych obiektów semantycznych Semantyki Statycznej Jądra Extended ML, przestawione są w Tabeli 1. Dziedziny te: TyVar, TyName i StrName, to dowolne nieskończone zbiory. Ponadto każdy $t \in \mathrm{TyName}$ posiada arność $k \geq 0$, a klasa nazw typów o arności k znaczana jest $\mathrm{TyName}^{(k)}$.

α lub *tyvar*	$\in$	TyVar	zmienne typowe
t	$\in$	TyName	nazwy typów
m	$\in$	StrName	nazwy struktur

Tabela 1: Proste obiekty semantyczne

Dziedziny złożonych obiektów semantycznych Semantyki Statycznej Jądra EML, przestawione są w Tabeli 2.

Gdy A i B są zbiorami, $\mathrm{Fin}\,A$ oznacza zbiór skończonych podzbiorów A, zaś $A \stackrel{\mathrm{fin}}{\to} B$ oznacza zbiór skończonych odwzorowań (częściowych funkcji o skończonej dziedzinie) z A do B.

Dla dowolnej dziedziny obiektów semantycznych A definiujemy $\mathrm{Tree}(A)$ jako zbiór skończonych binarnych drzew o elementach z A. Innymi słowy $\mathrm{Tree}(A)$ jest najmniejszym rozwiązaniem równania dziedzinowego

$$\mathrm{Tree}(A) = \{\epsilon\} \uplus A \uplus \{x \cdot y \mid x, y \in \mathrm{Tree}(A)\}$$

Operacja $\uplus$ oznacza sumę rozłączną dziedzin. Zakładamy, że wszystkie definiowane dziedziny są rozłączne.

$$
\begin{array}{rcl}
\tau & \in & \mathrm{Type} = \mathrm{TyVar} \uplus \mathrm{RecType} \uplus \mathrm{FunType} \uplus \mathrm{ConsType} \\
(\tau_1, \cdots, \tau_k) \text{ lub } \tau^{(k)} & \in & \mathrm{Type}^k \\
(\alpha_1, \cdots, \alpha_k) \text{ lub } \alpha^{(k)} & \in & \mathrm{TyVar}^k \\
\varrho & \in & \mathrm{RecType} = \mathrm{Lab} \stackrel{\mathrm{fin}}{\to} \mathrm{Type} \\
\tau \to \tau' & \in & \mathrm{FunType} = \mathrm{Type} \times \mathrm{Type} \\
 & & \mathrm{ConsType} = \uplus_{k \geq 0} \mathrm{ConsType}^{(k)} \\
\tau^{(k)} t & \in & \mathrm{ConsType}^{(k)} = \mathrm{Type}^k \times \mathrm{TyName}^{(k)} \\
\theta \text{ lub } \Lambda\alpha^{(k)}.\tau & \in & \mathrm{TypeFcn} = \uplus_{k \geq 0} \mathrm{TyVar}^k \times \mathrm{Type} \\
\sigma \text{ lub } \forall\alpha^{(k)}.\tau & \in & \mathrm{TypeScheme} = \uplus_{k \geq 0} \mathrm{TyVar}^k \times \mathrm{Type} \\
S \text{ lub } (m, E) & \in & \mathrm{Str} = \mathrm{StrName} \times \mathrm{Env} \\
(\theta, CE) & \in & \mathrm{TyStr} = \mathrm{TypeFcn} \times \mathrm{ConEnv} \\
SE & \in & \mathrm{StrEnv} = \mathrm{StrId} \stackrel{\mathrm{fin}}{\to} \mathrm{Str} \\
TE & \in & \mathrm{TyEnv} = \mathrm{TyCon} \stackrel{\mathrm{fin}}{\to} \mathrm{TyStr} \\
CE & \in & \mathrm{ConEnv} = \mathrm{Id} \stackrel{\mathrm{fin}}{\to} \mathrm{TypeScheme} \\
VE & \in & \mathrm{VarEnv} = \mathrm{Id} \stackrel{\mathrm{fin}}{\to} \mathrm{TypeScheme} \\
E \text{ lub } (SE, TE, VE) & \in & \mathrm{Env} = \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{VarEnv} \\
T & \in & \mathrm{TyNameSet} = \mathrm{Fin}(\mathrm{TyName}) \\
U & \in & \mathrm{TyVarSet} = \mathrm{Fin}(\mathrm{TyVar}) \\
C \text{ lub } T, U, E & \in & \mathrm{Context} = \mathrm{TyNameSet} \times \mathrm{TyVarSet} \times \mathrm{Env} \\
\varphi_{\mathrm{Ty}} & \in & \mathrm{TyRea} = \mathrm{TyName} \to \mathrm{TypeFcn} \\
\gamma & \in & \mathrm{Trace} = \mathrm{Tree}(\mathrm{SimTrace} \uplus \mathrm{TraceScheme}) \\
 & & \mathrm{SimTrace} = \mathrm{Type} \uplus \mathrm{Env} \uplus (\mathrm{Context} \times \mathrm{Type}) \uplus \\
 & & \quad (\mathrm{Context} \times \mathrm{Env}) \uplus (\mathrm{Context} \times \mathrm{TyName}) \uplus \\
 & & \quad \mathrm{TyEnv} \uplus (\mathrm{VarEnv} \times \mathrm{TyRea}) \\
 & & \mathrm{TraceScheme} = \uplus_{k \geq 0} \mathrm{TraceScheme}^{(k)} \\
\forall\alpha^{(k)}.\gamma & \in & \mathrm{TraceScheme}^{(k)} = \mathrm{TyVar}^k \times \mathrm{Trace}
\end{array}
$$

Tabela 2: Złożone obiekty semantyczne

Obiekty semantyczne Semantyki Statycznej Modułów EML to obiekty semantyczne Semantyki Statycznej Jądra EML, oraz dodatkowe obiekty semantyczne przestawione w Tabeli 3.

$$
\begin{array}{rcl}
M & \in & \mathrm{StrNameSet} = \mathrm{Fin}(\mathrm{StrName}) \\
N \text{ lub } (M, T) & \in & \mathrm{NameSet} = \mathrm{StrNameSet} \times \mathrm{TyNameSet} \\
\Sigma \text{ lub } (N)S & \in & \mathrm{Sig} = \mathrm{NameSet} \times \mathrm{Str} \\
\Phi \text{ lub } (N)(S, (N')S') & \in & \mathrm{FunSig} = \mathrm{NameSet} \times (\mathrm{Str} \times \mathrm{Sig}) \\
G & \in & \mathrm{SigEnv} = \mathrm{SigId} \stackrel{\mathrm{fin}}{\to} \mathrm{Sig} \\
F & \in & \mathrm{FunEnv} = \mathrm{FunId} \stackrel{\mathrm{fin}}{\to} \mathrm{FunSig} \\
B \text{ lub } N, F, G, E & \in & \mathrm{Basis} = \mathrm{NameSet} \times \mathrm{FunEnv} \times \mathrm{SigEnv} \times \mathrm{Env} \\
\varphi_{\mathrm{Str}} & \in & \mathrm{StrRea} = \mathrm{StrName} \to \mathrm{StrName} \\
\varphi \text{ lub } (\varphi_{\mathrm{Ty}}, \varphi_{\mathrm{Str}}) & \in & \mathrm{Rea} = \mathrm{TyRea} \times \mathrm{StrRea} \\
\gamma & \in & \mathrm{Trace} = \mathrm{Tree}(\mathrm{Trace}_{\mathrm{COR}} \uplus \mathrm{SimTrace} \uplus \mathrm{BoundTrace}) \\
(N)\gamma & \in & \mathrm{BoundTrace} = \mathrm{NameSet} \times \mathrm{Trace} \\
 & & \mathrm{SimTrace} = \mathrm{StrName} \uplus \mathrm{Rea} \uplus \mathrm{VarEnv} \uplus \mathrm{TyEnv} \uplus \mathrm{Env}
\end{array}
$$

Tabela 3: Dalsze złożone obiekty semantyczne

Bibliografia

[1] H. P. Barendregt. *The Lambda Calculus, its Syntax and Semantics.* 2nd edition, North Holland (1984).

[2] L. Birkedal, N. Rothwell, M. Tofte and D. N. Turner. *The ML Kit, Version 1.* (1993).

[3] L. Damas and R. Milner. Principal type schemes for functional programs. *Proc. 9th ACM Symposium on Principles of Programming Languages*, pp. 207–212. ACM, New York (1982).

[4] M. Jurdziński, M. Konarski, A. Schubert. *The EML Kit, Version 1.* Technical Report TR 96-04 (225), Institute of Informatics, Warsaw University (1996).

[5] J. Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2), 123–152 (1993).

[6] R. Harper. *Introduction to Standard ML.* Report ECS-LFCS-86-14, Univ. of Edinburgh (1986).

[7] S. Kahrs. *Mistakes and ambiguities in the definition of Standard ML.* Report ECS-LFCS-93-257, Univ. of Edinburgh (1993).

[8] S. Kahrs. *Mistakes and ambiguities in the definition of Standard ML – Addenda.* Univ. of Edinburgh (1995).

[9] S. Kahrs. *On the Static Analysis of Extended ML.* Univ. of Edinburgh (1994).

[10] S. Kahrs, D. Sannella and A. Tarlecki. *The Definition of Extended ML.* Report ECS-LFCS-94-300, Univ. of Edinburgh (1994).

[11] S. Kahrs, D. Sannella and A. Tarlecki. *The Definition of Extended ML, Version 1.15 + 0.2i.*

[12] S. Kahrs and D. Sannella and A. Tarlecki. The semantics of Extended ML: a gentle introduction. *Proc. Workshop on Semantics of Specification Languages*, Utrecht, 1993. Springer Workshops in Computing, 186–215 (1994).

[13] X. Leroy. Manifest types, modules and separate compilation. In: *Proc. 21st ACM Symposium on Principles of Programming Languages, Portland.* ACM (1994).

[14] X. Leroy. *Polymorphic typing of an algorithmic language*. Research report 1778, INRIA (1992).

[15] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press (1991).

[16] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press (1990).

[17] C. Reade. *Elements of functional programming*. Addison-Wesley (1989).

[18] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Fundamental Aspects of Computing*, to appear.

[19] D. Sannella and A. Tarlecki. Extended ML: past, present and future. *Proc. 7th Workshop on Specification of Abstract Data Types*, Wusterhausen. Springer LNCS 534, 297–322 (1991).

[20] D. Sannella and M. Wirsing. Specification Languages. In: *Algebraic Foundations of Systems Specifications* (E. Astesiano, H.-J. Kreowski and B. Krieg-Brückner, editors). Chapman and Hall, to appear.

[21] M. Wirsing. Algebraic specification. In: *Handbook of Theoretical Computer Science, Vol. B* (J. van Leeuwen, ed.), 675-788. Elsevier and MIT Press (1990).