Genetic Programming in Typed Languages

Master Thesis

Łukasz Stafiniak

Politechnika Wrocławska, WPPT

Email: lukstafi@wp.pl

promotor dr Zdzisław Spławski

June 15, 2005
Translated "as is" October, 2007

Table of contents

Chapter 1
Introduction

1.1 Abstract.

This work explores the possibilities of using rich type systems in Genetic Programming (GP). GP uses evolutionary algorithms to search through the program space. This requires development of genetic operators: 0-ary operator of creation (generating random progarm), 1-ary operator of mutation (tiny random modification), 2-ary operator of recombination (exchange of program fragments). Term generation requires/implies type inference. This work argues for using implicit polymorphism, and basing recombination on anti-unification. Genetic operators reduce to term generation for given type, and finding subterms with "homomorphic" typings. This work illustrates the key properties of solutions: soundness and completeness of term generation, soundness of term recombination, with proofs for simple cases. New light is shed on second-order anti-unification. Practical algorithms for type systems richer than that of SML lanugage need further work.

1.2 Issues.

GP is a metaheuristic for solving problems by automatic construction of programs, which uses the domain knowledge by the evaluating function (fitness function). The generality of GP follows from the fact, that semantic analysis of programs is not needed, because programs are judged by execution (e.g. in a simulated environment). An introduction to evolutionary methods can be found in [6]. The term Genetic Programming was founded by John Koza [10].

1.2.1 GP and genes.

GP is distinguished among evolutionary algorithms by the fact, that its search space is essentially infinit; programs cannot be approximated by fixed finite dimension vectors of features. With such vector representation, the features are called genes. The equivalent of "building block hypothesis" in GP is that GP leads to automatic identification of useful, functional subprograms, which spread through the population and are refined in different contexts. This is counterparted in the domain of natural selection by a concept of the unit of selection, due to Dawkins [3]. This suggests thinking about the pool of genes as something fluent, evolving. Genes are pieces of programs which have huge chance of being preserved during recombination. C.f. [1].

1.2.2 Advantage of using type systems: deductive synthesis.

Richer type systems give an alternative (to fitness function) possibility to include domain knowledge, available when this knowledge is logically strict. They restrict the search space, eliminating silly constructions and allowing to give specifications for solutions. Using the most of knowledge at synthesis time is important especially when evaluation of solutions is expensive. The goal is given as a goal type, local conditions as types of primitive constructs (e.g. built-in functions).

1.2.3 Automatic programming – program synthesis: inductive and deductive synthesis.

Hasło "automatyczne programowanie" obejmuje różne techniki komputerowego wspierania programowania. Przegląd zagadnień zawiera np. [15]. Synteza programów oznacza automatyczne (lub półautomatyczne, interaktywne) generowanie programów. Synteza dedukcyjna oznacza syntezę programu na podstawie jego pełnej specyfikacji w pewnym systemie formalnym. Systemy syntezy dedukcyjnej dzielimy na teorio-dowodowe i transformacyjne. W tych pierwszych specyfikacja jest pewną formułą logiczną, dowód której odpowiada znalezieniu programu: w logikach konstruktywnych dowód istnienia oznacza podanie odpowiedniego obiektu, specyfikacja stwierdza istnienie szukanych obiektów. Systemy transformacyjne przekształcają specyfikację przy pomocy zbioru reguł, i gdy nie ma już reguł do zastosowania, wynik transformacji jest szukanym programem. Systemy teorio-dowodowe podzieliłbym dalej na systemy regułowe i systemy teoriotypowe. Pierwsze przekształcają formułę przy pomocy zbioru reguł logicznych, otrzymując w wyniku formułę (formuły) wyrażającą odpowiedź; fragment tej formuły jest szukanym programem. Np. formułą wejściową jest tautologia $P(?x) \Rightarrow P(?x)$, gdzie $?x$ jest zmienną egzystencjalną, P jest specyfikacją; reguły logiczne zachowują prawdziwość; formułą wynikową jest $P(t)$, gdzie t jest szukanym programem. Drugie konstruują dowód w postaci drzewa (węzły drzewa identyfikują stosowane reguły); na mocy izomorfizmu Curry'ego-Howarda, dowód jest szukanym programem.

Synteza indukcyjna oznacza generowanie programu na podstawie niepełnej wiedzy wyrażającej nasze oczekiwania; wiedzy "słabo ustrukturowanej", tzn. nie dającej wskazówek co do struktury programu. Mamy do dyspozycji między innymi: albo ustalone pary wejście-wyjście (uczenie się z nauczycielem, program dla zadanych wejść ma zwracać określone wyjścia), albo metodę zwracającą poprawne wyjścia dla danych wejść (uczenie się z wyrocznią, program ma być prostszą metodą aproksymującą wyrocznię), albo metodę określającą jakość danego wyjścia dla danego wejścia (uczenie się ze wzmocnieniem przez eksperymentowanie), albo sposób oceny jakości całego programu (uczenie się ze wzmocnieniem; nazwa pochodzi stąd, że możemy ocenić, czy modyfikacja rozwiązania polepsza go czy pogarsza). Uczenie się ze wzmocnieniem daje użytkownikowi największą swobodę co do informacji użytej w ocenie programów, ale najmniej wskazówek systemowi co do struktury programu. Z kolei możemy podzielić systemy indukcyjnego uczenia się na lokalne i globalne (nielokalne). Metody lokalne rozpatrują atrakcyjność pojedynczego rozwiązania (np. metody optymalizacji gradientowej). Metody globalne (nielokalne) rozpatrują hipotezę opisującą atrakcyjne obszary przestrzeni rozwiązań (np. metody optymalizacji dziel-i-rządź).

Programowanie genetyczne to metoda syntezy indukcyjnej programów będąca nielokalnym uczeniem się ze wzmocnieniem. Generowanie programów w językach typizowanych to teoriodowodowa, teorio-typowa metoda syntezy dedukcjyjnej. Podsumowując:

1. Deductive synthesis
    a. transformational
    b. proof-theoretical
        i. rule-based
        ii. **type-theoretical**
2. Synteza indukcyjna
    a. uczenie się z nauczycielem
    b. uczenie się z wyrocznią
    c. uczenie się ze wzmocnieniem przez eksperymentowanie
    d. uczenie się ze wzmocnieniem
        i. lokalne
        ii. **globalne (nielokalne)**

Praca proponuje połączenie metod syntezy dedukcyjnej i indukcyjnej, aby najpełniej wykorzystać wiedzę o problemie. Tą część specyfikacji, która ma charakter "logiczny" i jest na tyle prosta, że pozwala na efektywną dedukcję programów, zawiera się w systemie typów i typie zadanym. Pozostałą część wiedzy o problemie zawiera się w funkcji oceniającej programy.

Algorytmy genetyczne w szerokim sensie można nazwać uczeniem się "co prawda bez nauczyciela, ale z systemem edukacyjnym". Operatory rekombinacji próbuje się tworzyć tak, aby, jeśli rodzice (argumenty operatora) mają ten sam wynik dla danego wejścia, to potomek (rezultat operatora) też miał ten wynik dla tego wejścia – rodzice są nauczycielami potomka. Niestety, w GP stworzenie takiego operatora rekombinacji jest zbyt trudne.

1.2.4 Genetic operators and term generation.

Aby algorytm genetyczny (ewolucyjny) mógł rozpocząć pracę, potrzebuje co najmniej trzech operatorów genetycznych: 0-argumentowego operatora kreacji (dla budowy populacji początkowej), 1-argumentowego operatora mutacji (dla przeszukiwania lokalnego), 2-argumentowego operatora rekombinacji (dla przeszukiwania nielokalnego). Podstawą operatorów kreacji i mutacji jest algorytm generowania termów. Rozdzielamy zagadnienia przeszukiwania i konstrukcji, redukując generowanie termów do przeszukiwania drzew. Algorytm konstrukcji termu $\boldsymbol{C}(E, \tau, \vec{w})$ dla pewnych ciągów wyborów $\vec{w}$ (ciągów liczb naturalnych określających pojedynczą ścieżkę wykonania algorytmu niedeterministycznego) zwraca term e o typie τ w środowisku E; dla pozostałych $\vec{w}$ zawodzi (zgłasza niemożność zbudowania termu). Należy wtedy podjąć próbę dla innego ciągu $\vec{w}'$, np. zachowując możliwie wiele z wcześniejszej pracy algorytmu dzięki mechanizmowi nawrotów (kontynuacji) – jest to część zagadnienia przeszukiwania. Operator kreacji

zwraca $\boldsymbol{C}(E_0, \tau_0, \vec{w})$, gdzie E_0 jest środowiskiem początkowym (środowiskiem konstrukcji pierwotnych), a τ_0 jest typem zadanym – specyfikacją problemu, a $\vec{w}$ odpowiednim (znalezionym) ciągiem wyborów. Operator mutacji (mutacja podtermu) polega na zamianie w termie e podtermu e_1 na inny. Wiemy, że zachodzi sąd typizujący $E_0 \vdash e\colon \tau_0$; niech podterm e_1 będzie wprowadzony do termu e w wyprowadzeniu tego sądu typizującego z przesłanki $E_1 \vdash e_1\colon \tau_1$. Wtedy operator mutacji podstawia pod e_1 wynik $\boldsymbol{C}(E_1, \tau_1, \vec{w})$ dla odpowiedniego ciągu wyborów $\vec{w}$. Informację o typie podtermów i środowisku jego wyprowadzenia można przechowywać w tych samych strukturach, które wykorzystuje mechanizm nawrotów.

1.2.5 Term generation and "answer substitution".

Nawet, jeśli będziemy zadowoleni z algorytmu zwracającego term dokładnie typu τ, to w jego implementacji będzie potrzebny algorytm pozwalający na konkretyzację typu zadanego. Nie musimy, a wewnątrz algorytmu generującego często nie możemy, wiedzieć wszystko o typie termu, którego szukamy. Niech więc $\boldsymbol{C}(E, \tau, \vec{w})$ znajduje term e i podstawienie T takie, że $TE \vdash e\colon T\tau$. Porównaj podrozdział 3.4 "Proof-search in Logical Frameworks" artykułu [5], czy też wprowadzenie do SLD-rezolucji w [17]. Zmienne wolne pozostawiamy w τ jako niewiadome, które ma znaleźć algorytm.

Gdy rozpatrujemy operator mutacji, należy pamiętać, aby zablokować możliwość podstawiania pod zmienne, które w nadtermie typu zostają zgeneralizowane (traktując je jak stałe). Następnie modyfikujemy wyprowadzenie typu dla całego programu, stosując odpowiednio podstawienie zwrócone przez wygenerowanie nowego podtermu. Wadą takiego stosowania mutacji jest nadokreślenie typu programu (jest bardziej skonkretyzowany niż to wynika z termu), dlatego można retypizować cały program po zastosowaniu mutacji.

1.2.6 Tightening the correspondence between the sequence of choices and (properly) typed terms; the search.

Ciąg wyborów wyraża niedeterminizm, czy też stochastyczność algorytmu $\boldsymbol{C}$. Idealnie byłoby, gdyby dla każdego ciągu wyborów algorytm zwracał term poprawnie typowany, a wyliczeniu ciągów wyborów odpowiadałoby wyliczenie termów zamieszkujących zadany typ. Jednak algorytm wykorzystuje ciąg wyborów do generowania termu, i wyliczeniu ciągów wyborów odpowiada wyliczenie wszystkich termów odpowiedniej klasy zadanej syntaktycznie.

1.2.6.1 Term generation as a search in program space.

Rzeczywisty algorytm uzyskujemy więc z $\boldsymbol{C}$ dzięki zastosowaniu przeszukiwania grafów (tutaj zasadniczo drzew). Oczywiście, łatwiej przeszukiwać w głąb, ale lepiej przeszukiwać wszerz: krótsze termy mają większą szansę być lepszymi programami, $\boldsymbol{C}$ z przeszukiwaniem wszerz daje rekurencyjną przeliczalność problemu znajdowania mieszkańca zadanego typu. Dla operatorów genetycznych zadających ciąg wyborów losowo i oczekujących odpowiedzi, zwracamy term najbliższy wskazanemu węzłowi (w sensie obranej strategii przeszukiwania) (lub odpowiadamy, że taki nie istnieje, jeśli wyczerpaliśmy przestrzeń). Możemy również zwracać rzeczywisty ciąg wyborów, dla ewentualnego wznowienia przeszukiwania (przestrzeni programów). W praktycznej implementacji pamiętamy o wymuszaniu stopu, gdy przeszukiwanie trwa zbyt długo; zawodzi wtedy próba zastosowania operatora genetycznego.

1.2.6.2 On the need of type inference.

Generowanie termu obejmuje inferencję typu: wyprowadzamy termy typizowane. Jeśli nie mamy mechanizmów inferencji (typizacja bazuje na annotacjach termów typami), zastosowanie przeszukiwania grafów de facto rozwiązuje zadanie inferencji typu dla wygenerowanego termu. W ten sposób struktura przeszukiwań zostaje zdominowana przez problem inferencji typu, podczas gdy chcemy, aby wyrażała semantykę operacyjną programów (jak działają). Dlatego konieczne jest odizolowanie termów i typów, nawet niewyszukany algorytm inferencji dla systemów nierozstrzygalnych (z konieczności algorytm bez własności stopu, w praktyce zaś ze stopem wymuszonym, czyli algorytm niepełny) będzie lepszy od naiwnej enumeracji wszystkich możliwych typizowań. Możemy przywrócić "przeliczalną pełność" dodając w momencie stopowania typizacji jej kontynuację jako (odłożony na później) węzeł struktury przeszukiwań.

1.2.6.3 "Head-driven" algorithm.

Już w latach trzydziestych wiadomo było, że intuicjonistyczny rachunek zdań jest rozstrzygalny. W interpretacji w lambda-rachunku oznacza to, że dla każdego typu z systemu typów prostych możemy znaleźć term tego typu, albo pokazać, że takich termów nie ma. Wykorzystamy standardowy algorytm znajdowania takiego termu, opisany np. w pracy [14]. Idea polega na wykorzystaniu własności "cut elimination", czyli w terminologii λ-rachunku faktu, że jeśli istnieje term danego typu, to istnieje term bez β-redeksów, również tego typu. Można więc szukać termu wśród termów normalnych (bez β-redeksów), a pomijając zapętlenia w wyprowadzeniach, jest wtedy tylko skończenie wiele ścieżek do sprawdzenia, kończących się zadanym typem. Oczywiście, możemy zastosować ten mechanizm dla poszukiwania programów funkcyjnych, ponieważ β-redukcja zachowuje wtedy sens programu. Dla systemu typów prostych mamy więc algorytm generowania, zatrzymujący się dla każdego zadanego typu, z termem tego typu lub stwierdzeniem pustości typu.

1.2.6.4 Decidability of non-emptiness of types (inhabited types) for ML.

Dla ML bez algebraicznych typów danych, czyli dla ML bez dopasowywania wzorca (lub innych mechanizmów typu CASE), łatwo znaleźć algorytm zwracający term zadanego typu z odpowiadającym programem z własnością stopu, lub odpowiadający, że jeśli typ zamieszkują jakieś termy, to odpowiadające im programy nie zatrzymują się dla żadnego argumentu. Można przeprowadzić eliminację definicji rekurencyjnych i lokalnych i następnie zastosować argument taki sam, jak dla systemu typów prostych. (Jeśli program zatrzymuje się dla pewnych danych, to musi istnieć gałąź programu licząca wynik bez wywołania rekurencyjnego.) Eliminacja zmienia sens programu, ale nie zmienia typu programu. Pełny ML wydaje się nie być rozstrzygalny w tym sensie, ale nie potrafię tego pokazać. Oczywiście, rozstrzygalność ogólnego problemu niepustości typu (czy istnieje term danego typu) jest równoważna (przez izomorfizm Curry'ego-Howarda) rozstrzygalności odpowiadającej systemowi logiki. Pozytywne wyniki wiążą się z pełnością programowania w Prologu.

Algorytm generujący β-normalne polimorficzne λ-termy (wykorzystując mechanizm unifikacji) odpowiada strategii SLD-rezolucji. Rozszerzamy go o pozostałe konstrukcje ML (czy obszerniejszych systemów). Generowanie (wyliczanie) termów zgodnie z tym algorytmem jest równoważne strategii ewaluacji użytej w Prologu.

1.2.7 Recursive definitions and local definitions.

Zasygnalizowane w 1.2.6.4 problemy teoretyczne mają swój praktyczny odpowiednik w zagadnieniu użyteczności definicji rekurencyjnych i definicji lokalnych. Definicje rekurencyjne są użyteczne, jeśli odpowiadające im funkcje przynajmniej czasami się zatrzymują (nie będę się zajmował korekurencją). Zapewnienie mocniejszego warunku użyteczności jest opisane w 1.2.10.2. Natomiast definicje lokalne są użyteczne, jeśli przynajmniej raz są użyte w zakresie defi¿icji, są ce¿owe, jeśli są użyte więcej niż raz. Skąd jednak wiedzieć, jakie definicje będą przydatne w budowanym podtermie? Następujące rozwiązanie wydaje się optymalne: dopuszczaj, aby każdy wygenerowany podterm był możliwą definicją lokalną, o zakresie będącym najszerszym kontekstem zawierającym zmienne lokalne podtermu. Jeśli definicja będzie użyta (poza miejscem podtermu definiującego), to rzeczywiście ją wprowadzamy. W przeciwnym wypadku pozostawiamy podterm "na miejscu", nie wprowadzając definicji. To rozwiązanie ma jednak wadę, jeśli stosuje się je dla termów bez β-redeksów: wprowadza tylko definicje lokalne o typach, które są podtermami typów ze środowiska lub typu zadanego (właściwie, generalizacje tych typów). (Podobny problem pojawia się dla definicji rekurencyjnych.) Ogólnie, chcemy, aby definicje lokalne mogły tworzyć po zredukowaniu β-redeksy, ale nie chcemy wprowadzać bezużytecznych λ-abstrakcji, których zmienne nie będą następnie wykorzystane. Rozwiążemy to, "patternizując" wygenerowany podterm (o ile ma stanowić definicję lokaną), czyli wycinając podtermy tego podtermu i formując β-redeksy, następnie aplikacje zostawiając w zakresie definicji, a λ-abstrakcje w ciele definicji.

Pojawia się jeszcze jeden problem techniczny. Dla definicji lokalnej należy wyznaczyć najogólniejszy typ. Ale tworzona jest ona w innym środowisku, niż ostatecznie środowisko jej zakresu.

1.2.8 Recombination and the homomorphism of typings.

Recombination is generally based on substituting a subterm e_1 of given program e_0 with a subterm e_2 from another program; this requires the substituting subterm be of a type (typing) somehow concretizing the type (typing) of substituted subterm. Afterwards, typing for the whole term should be reconstructed. In simple types with subtyping, the subtyping relation is enough. In parametric polymorphism systems, where type parameters might be "convoluted" in the context of occurrence of the subterm, we need to build the correspondence between typing judgements $E_1 \vdash e_1 : \tau_1$ and $E_2 \vdash e_2 : \tau_2$. The domain of environments of parent programs is common, but the types can be concretized differently by „answer substitutions". The substitutions need to be unifiable, the environment E_2 must be injectable into E_1, $h : \mathrm{Dome}(E_2) \to \mathrm{Dom}(E_1)$, which is an identity on $\mathrm{Dom}(E_0)$ (renaming of free variables on e_2). If we base reconstruction of typing on $E_2 \vdash e_2 : \tau_2$, then renaming must "argee" with types: $H_L\, E_1(x) = E_2(h(x))$, and the type $H_R \tau_2 = \tau_1$, while H_R does not substitute for variables occuring in E_2 ($\mathrm{Dom}(H_R) \cap \boldsymbol{F}(E_2) = \varnothing$). We call he triple h, H_L, H_R the E_0-homomorphism of typings $E_1 \vdash e_1 : \tau_1$ and $E_2 \vdash e_2 : \tau_2$.

1.2.9 Recombination by anti-unification.

Jak widzimy, rekombinacja w przypadku bogatszych systemów typów jest złożonym problemem znajdowania analogii (między podtermami). Wielu autorów argumentuje, że podstawowym narzędziem znajdowania, czy też konstrukcji, analogii jest generalizacja (jako znajdowanie kategorii zawierającej wskazane obiekty) (np. [7]). Użyjemy generalizacji programów dla znajdowania analogicznych pozycji podtermów. Mechanizmem syntaktycznej generalizacji termów jest anty-unifikacja. W kracie termów z porządkiem danym przez podstawienia (jeśli $(\exists T) T e_1 \equiv e_2$, to $e_1 \geqslant e_2$, gdzie T jest podstawieniem) unifikacja jest operacją infimum, a anty-unifikacja operacją supremum dwu termów. Jednak przy pewnych naturalnych definicjach równoważności i podstawień anty-unifikacja nie jest dobrze określona (supremum nie istnieje). Artykuły dotyczące anty-unifikacji: [12], [7], [9]. Ogólnie pomysł stosowania anty-unifikacji do rekombinacji termów polega na krzyżowaniu (ang. crossing-over) podstawień. Dla termów e_1 i e_2 wynikiem anty-unifikacji jest term e i podstawienia A_1, A_2 t. że $e_1 = A_1 e$ i $e_2 = A_2 e$. Jeśli $A_i = [e_{i,1}/x_1; \ldots; e_{i,n}/x_n]$, rekombinacją e_1, e_2 będzie $[e_{w_1,1}/x_1;\ \ldots;\ e_{w_n,n}/x_n]e$, gdzie $\vec{w}$ jest dowolnym ciągiem jedynkowo-dwójkowym. Jednak w bogatszych systemach typów jedne podstawienia determinują typ dla innych podstawień i rekombinacja wymaga dostosowania tej techniki (ograniczenia dopuszczalnych ciągów $\vec{w}$ tak, by rekombinant był typizowalny).

1.2.9.1 Generalization and GP.

Jak podpowiada rozsądek, wspólne cechy obiecujących rodziców należy zachować u potomstwa. Oznacza to, że rekombinacja powinna respektować maksymalnie wiele wspólnych cech; często nie może respektować (zachowywać) wszystkich (porównaj analizy w [13]), w podanej wyżej formalizacji oznacza to, że supremum zawiera wiele termów. Artykuł [4] pokazuje użyteczność rekombinacji zachowującej generalizacje pierwszego rzędu (nie jest to rekombinacja krzyżująca podstawienia anty-unifikatora, zachowuje się ona bardziej "swobodnie") używanej łącznie z rekombinacją swobodną. Ograniczanie się do rekombinacji zachowującej kontekst wymienianych podtermów (jak rekombinacja krzyżująca anty-unifikatory) ma oczywiste wady, gdy system GP nie ma wsparcia dla duplikacji kodu i wykorzystywania podprogramów (code reuse). Jednak proponowany w niniejszej pracy system wspiera te mechanizmy ewolucji dzięki obsłudze definicji lokalnych: mutacja może wprowadzić wykorzystanie w danym miejscu dowolnego podtermu, który w razie potrzeby zostanie "podniesiony" do definicji lokalnej; odpowiedni operator genetyczny zajmuje się rozwijaniem definicji. Rekombinacja zachowująca kontekst ma duże znaczenie w kontekscie teori "samolubnego genu" nadając tożsamość jednostkom selecji: wymieniany podterm nie trafi w przypadkowe miejsce programu, pozostanie w tej samej pozycji anty-unifikatora. W ten sposób selekcja będzie następowała ze względu na spełnianie funkcji wyznaczonej przez anty-unifikator dla poszczególnych pozycji. Rekombinacja swobodna (dodatkowo dla języków nietypizowanych, gdzie jest ona rzeczywiście swobodna) powoduje nabywanie przez programy dużych partii "niekodujących" – nigdy nie wykonywanych; problem ten jest nazywany "code bloat". Jest to mechanizm obronny zmniejszający prawdopodobieństwo przecinania przez rekombinację cennych fragmentów kodujących, co zazwyczaj zaburzy ich funkcję (patrz m. in. [2] znakomicie pokazujący, że co jest dobre dla genów, nie musi być dobre dla szybkości ewolucji).

1.2.9.2 Recombination agreeing with mutation and second-order anti-unification.

Najpierw przedstawimy techniczne pojęcie kompatybilnych mutacji próbkowych, żeby następnie podać obrazowe pojęcie rekombinacji zgodnej z mutacją, dodatkowo uzasadniające oparcie rekombinacji programów na anty-unifikacji.

Pozycja podtermu obejmuje pozycje podtermów tego podtermu. Pozycje w termie nazywamy rozdzielonymi, jeśli żaden ich podzbiór nie obejmuje wspólnie wszystkich pozycji obejmowanych przez pewną pozycję obejmującą te pozycje, oprócz tej pozycji. Tzn. nie istnieje pozycja, obejmująca pozycje z tego podzbioru, i nie obejmująca żadnej pozycji nie obejmowanej przez pozycje z podzbioru poza samą sobą. Zdefiniujmy zbiór kompatybilnych mutacji próbkowych dla programu $E_0 \vdash e\colon \tau_0$ (odpowiednio dobranego). Mówimy, że zastosowano mutację $p_i, \vec{w}_i$, jeśli dany program $E_0 \vdash e_0\colon \tau_0$ ma w pozycji p_i podterm z typizacją $E_i \vdash e_i\colon \tau_i$, a wynikowy program ma dla T_i, $e_i' = \boldsymbol{C}(E_i, \tau_i, \vec{w}_i)$ postać $T_i\ E_0 \vdash [e_i'/p]e_0\colon T_i\tau_0$. Mutacja jest próbkowa, jeśli $\mathrm{FV}(e_1') \subset \mathrm{Dom}(E_0)$ oraz żadna zmienna ani stała e_i' nie występuje w e_0. Mutacje próbkowe są kompatybilne, jeśli pozycje p_i są rozdzielone, $T_i, e_i' = \boldsymbol{C}(T_{\pi(1)}...T_{\pi(k)}E_i, T_{\pi(1)}...T_{\pi(k)}\tau_i, \vec{w}_i)$ dla dowolnego wyboru indeksów π nie zawierającego i.

Mówimy, że operator rekombinacji jest zgodny z operatorem mutacji, jeśli dla dowolnego zbioru M kompatybilnych mutacji próbkowych dowolnego odpowiedniego programu e, dla programu e_K powstałego z zastosowania mutacji $K \subseteq M$ i programu e_L powstałego z zastosowania mutacji $L \subseteq M$, zbiór wszystkich rekombinacji e_K, e_L składa się z wszystkich programów powstałych przez zastosowanie mutacji $N \subseteq M$, gdzie $K \cap L \subseteq N \subseteq K \cup L$. Porównaj warunki "respect" i "assortment" z pracy [13].

Definicja pozycji rozdzielonych została dobrana tak, aby zgodny z mutacją operator rekombinacji mógł wogóle istnieć. Gdyby pozycje nie były rozdzielone, rekombinacja nie potrafiłaby określić, ile właściwie mutacji zastosowano. Sprawdźmy, co wymusza powyższy warunek na operatorze rekombinacji. Rekombinacja musi określić pozycje, w których mutację zastosowano tylko w jednym z rodziców. W pierwszym kroku oznaczmy pozycje, minimalne w odległości od korzenia, o różnych etykietach (w grafie termu) u obu rodziców. W kolejnych krokach oznaczajmy pozycje, których wszyscy potomkowie są już oznaczeni. Na koniec wybierzmy minimalne w odległości od korzenia pozycje oznaczone. Rekombinacja polegać będzie na wymianie podtermów rodziców na tych pozycjach. Jeśli podstawimy zmienne w miejscu wybranych pozycji, uzyskamy maksymalnie szczegółową anty-instancję drugiego rzędu względem porządku danego przez podstawienia ograniczone do podtermów anty-unifikowanych programów. Jest to najprostsza postać anty-unifikacji drugiego rzędu. Dużo mocniejszą postać anty-unifikacji drugiego rzędu podaje [7]. Odpowiada ona operatorowi rekombinacji zgodnemu z mutacjami: mutacją podtermu, insercją (a więc i delecją) oraz permutacją podtermów, przy odpowiednio określonej niezależności mutacji. Podrozdział 3.3 opisuje własności anty-unifikacji Haskera i wyżej opisanej. W bogatszych systemach typów, określenie anty-unifikacji, a szczególnie rekombinacji, może być trudne.

1.2.10 Type systems with constraints.

Kluczowym dla naszej sprawy artykułem jest według mnie [16]. Jak wcześniej pisałem, potrzebujemy możliwie sprawnej metody inferencji typów dla systemu pozwalającego wyrażać logiczne własności programów. Systemy z "Guarded Algebraic Data Types", czyli typami indukcyjnymi zawierającymi charakterystykę logiczną poszczególnych przypadków, są pod tym względem bardzo wygodne, co pokazuje chociażby sukces systemu Coq, stosującego system typów Calculus of Inductive Constructions, rozszerzający system CC o właśnie "Guarded Algebraic Data Types" (GADT). Artykuł Simoneta i Pottiera podaje "minimalne" rozszerzenie systemu ML pozwalające na wygodne posługiwanie się GADT – sparametryzowany językiem wyrażania więzów X system HMG(X). Jednocześnie podaje sprawny algorytm inferencji typu. (Dla potrzeb syntezy programów można rzeczy dodatkowo uprościć rezygnując z pełnego mechanizmu "pattern matching" na rzecz prostszej analizy przez przypadki.) Jedyne annotacje, jakich potrzebuje system HMG(X), to typy dla funkcji rekurencyjnych. Potrzebuje ich dlatego, że aby wykorzystać możliwości GADT potrzebujemy rekurencji polimorficznej, a problem inferencji typu dla rekurencji polimorficznej jest nierozstrzygalny.

1.2.10.1 Undecidability of type inference for polymorphic recursion.

Do inferencji typu dla języka ML rozszerzonego o rekurencję polimorficzną (tzw. język ML+, albo system Milnera-Mycrofta) proponowano wiele podejść. Ważną pracą jest [8], pokazująca redukcję do problemu semi-unifikacji, oraz podająca użyteczny algorytm semi-unifikacji, pomimo nierozstrzygalności samego problemu. Aktualnie prace nad użytecznością inferencji typu dla rekurencji polimorficznej prowadzone są np. w ramach projektu badawczego [Research Project: "SEMI-UNIFICATION and Periodicity of Turing Machines" Carlos Camarao, Lucilia Figueiredo, Luigi Laporte, http://www.dcc.ufmg.br/~camarao/SUP/]. Przeprowadzenie inferencji typu dla rekurencji polimorficznej w systemie HMG(X) jest oczywiście trudniejsze, ale wydaje się nie mniej osiągalne niż w przypadku systemu ML+.

1.2.10.2 Type systems for program termination.

Kluczowym zagadnieniem poprawności programu jest posiadanie przez niego własności stopu. Autorka pracy [18] stosuje iteratory (czy też rekursory) jako funkcje pierwotne do manipulacji indukcyjnymi typami danych, nie pozwalając na rekurencję bezpośrednią. Przypomina to sytuację w systemie Coq (przed wprowadzeniem konstrukcji Fix), gdzie przy definicji typu indukcyjnego system od razu generuje odpowiednie rekursory (funkcje realizujące schematy rekursji). Jednak najodpowiedniejszy schemat rekursji dla danego problemu może być bardzo różny od ogólnego schematu rekursji dla danego typu danych – warto pozwolić systemowi zrekonstruować odpowiedni schemat rekursji. Można wymusić własność stopu przy pomocy systemu typów. Bardzo prostym, a jednocześnie ekspresyjnym systemem jest $\lambda\hat{}$ przedstawiony w [2]. Hongwei Xi konstruuje bardziej ekspresyjny system na bazie swojego języka DML(X) (DML(X) jest obejmowany przez HMG(X)) [17]. Pisząc funkcję rekurencyjną, w prosty sposób wprowadza się metrykę, względem której kolejne wywołania rekurencyjne mają maleć. (Metryki dopuszczają m. in. rekurencję względem wielu zmiennych.)

1.2.11 Overview of chapters.

1.2.11.1 Chapter 2 – Generating terms.

First a term generation algorithm for simple type systems, for normal forms, is presented (it corresponds to theorem proving in intuitionistic propositional calculus). Next, type inference and term generation algorithms for Damas-Milner type system (ML) are presented, which base on "operational" interpretation of system rules, and on unification; properties of these algorithms are given, with proofs. The system $\lambda\hat{}$ from [2] is presented, original version and one modified to agument the operational interpretation of rules; together with term generation algorithms, based on algorithms for ML. We describe the similarity between generating β-normal terms and SLD-resolution. Finally, I discuss the problem of optimal introduction of local definitions.

1.2.11.2 Chapter 3 – Generalization.

I present an algorithm for second-order anti-unification restricted to substitutions of subterms fo generalized terms, I show maximality of generalization and (conditions for) correctness of recombination which is based on it, for the Damas-Milner system. Next I discuss the solution for generalization from the work [7] and point to an oversight in it. I present a generalization algorithm based on that solution. I give a mechanism for finding the least general generalization and account for its correctness.

1.2.11.3 Chapter 4 – Contributions and future work.

Rozdział prezentuje rezultaty pracy i zadania postawione do rozwiązania przez pracę. W pierwszej części zebrane są aspekty nowatorskie pracy. W drugiej części wskazane są dalsze prace, jakie należy podjąć, na drodze do praktycznej implementacji systemu programowania genetycznego w językach z bogatszymi systemami typów.

Chapter 2
Term Generation.

2.1 The system of simple types and intuitionistic propositional calculus.

W tym podrozdziale zaprezentuję wyniki z pracy [14]. Stanowią one dobre wprowadzenie w problem genrowania termów, w dobrze zbadanym usytuowaniu intuicjonistycznego rachunku zdań.

2.1.1 The system of natural deduction and the typed λ-calculus.

Natural deduction (implicational fragment). Dla uproszczenia, ograniczamy się do implikacyjnego fragmentu logiki intuicjonistycznej.

Definition 2.1. *Zbiór formuł (implikacyjnego rachunku zdań) jest określony regułami:*

1. *Jeśli A jest formułą atomową, to A jest formułą.*
2. *Jeśli X, Y są formułami, to $X \to Y$ jest formułą.*

W tym podrozdziale używamy zamiennie pojęć formuła i typ.
Długość formuły $|A|$ definiujemy

$$|A| \;=\; \begin{cases} 1 & \text{jeśli } A \text{ jest atomowa} \\ |B|+|C|+1 & \text{jeśli } A = B \to C \end{cases}$$

Definition 2.2. *System dedukcji naturalnej **NJ** jest określony następującymi regułami:*

1. *(Aksjomat) Brak.*
2. *(I)*
$$\frac{B}{A \to B}$$
3. *(E) Modus ponens*
$$\frac{A \to B \qquad A}{B}$$

Definition 2.3. *Niech Φ będzie skończonym zbiorem formuł i α będzie formułą. Piszemy $\Phi \vdash_{NJ} \alpha$ jeśli istnieje dowód α którego zbiorem niezdjętych (undischarged) założeń jest Φ.*

Type assignment for λ-terms.

Definition 2.4. *λ-term jest zdefiniowany jako:*

1. *Zmienne $x_1, x_2, \ldots$ są λ-termami.*
2. *Jeśli M, N są λ-termami, to (MN) jest λ-termem.*
3. *Jeśli M jest λ-termem i x jest zmienną, to $(\lambda x.M)$ jest λ-termem.*

(MN) jest nazywany aplikacją, a $(\lambda x.M)$ abstrakcją.

Definition 2.5. *Zbiór zmiennych wolnych* $\boldsymbol{F}(M)$ *jest określony przez:*

1. $\boldsymbol{F}(x) = \{x\}$
2. $\boldsymbol{F}(MN) = \boldsymbol{F}(M) \cup \boldsymbol{F}(N)$
3. $\boldsymbol{F}(\lambda x.M) = \boldsymbol{F}(M) \backslash \{x\}$

Definition 2.6. *Jeśli* M *jest* λ*-termem i* α *jest typem, to opis* $M\colon \alpha$ *jest nazywany przypisaniem typu.*

Definition 2.7. *Założenie jest przypisaniem typu* $x\colon \alpha$ *zmiennej* x*. Zbiorem założeń nazywamy zbiór* $\Phi = \{x_1\colon \alpha_1, ..., x_n\colon \alpha_n\}$ *taki, że* $x_i \neq x_j$ *dla* $i \neq j$*.*

Definition 2.8. *System typów* TA_λ *jest określony przez reguły:*

1. *(Aksjomat)*
$$\Phi \vdash_\lambda x\colon \alpha \qquad (x\colon \alpha \in \Gamma)$$
2. $$\frac{\Phi \vdash_\lambda M\colon \alpha \to \beta \qquad \Phi \vdash_\lambda N\colon \alpha}{\Phi \vdash_\lambda MN\colon \beta}$$
3. $$\frac{\Phi \vdash_\lambda M\colon \beta}{\Phi \backslash \{x\colon \alpha\} \vdash_\lambda \lambda x.M\colon \alpha \to \beta}$$

Curry-Howard isomorphism.

Theorem 2.9.

$$\Phi \vdash_{NJ} \alpha \iff \begin{array}{l} \exists M\colon \lambda\text{-}term\ taki,\ że \\ \{x_1\colon \alpha_1, ..., x_n\colon \alpha_n\} \vdash_\lambda M\colon \alpha \\ oraz\ \boldsymbol{F}(M) \subseteq \{x_1, ..., x_n\} \end{array}$$

Theorem 2.10. $\vdash_{NJ} \alpha \iff \exists M\colon$ *domknięty (bez zmiennych wolnych)* λ*-term taki, że* $\vdash M\colon \alpha$*.*

2.1.2 The proofs in normal form.

Beta reduction.

Definition 2.11. *(Beta redukcja) Niech* M *i* N *będą* λ*-termami i* x *będzie zmienną.* λ*-term otrzymany z* M *przez zastąpienie wszystkich wystąpień* x *przez* N *oznaczamy* $[N/x]M$*.* β*-redukcja jest rekurencyjnie określona przez:*

1. $M \longrightarrow_\beta M$*.*
2. $(\lambda x.M)N \longrightarrow_\beta [N/x]M$*.*
3. *Jeśli* $M \longrightarrow_\beta N$*, wtedy* $\lambda x.M \longrightarrow_\beta \lambda x.N$*,* $MP \longrightarrow_\beta NP$ *oraz* $PM \longrightarrow_\beta PN$*.*
4. *Jeśli* $L \longrightarrow_\beta M$ *oraz* $M \longrightarrow_\beta N$*, to* $L \longrightarrow_\beta N$*.*

λ*-term postaci* $(\lambda x.M)N$ *jest nazywany* β*-redeksem.*

Definition 2.12. *(Postać beta normalna) Jeśli* λ*-term* M *nie zawiera* β*-redeksów, wtedy* M *jest w postaci* β*-normalnej. Zbiór wszystkich* λ*-termów w postaci normalnej jest oznaczany przez* βnf*. Mówimy, że term* M *ma postać* β*-normalną* N *wtw. gdy* $M \longrightarrow_\beta N$ *i* $N \in \beta$nf*.*

Theorem 2.13. *(Twierdzenie "subject reduction") Jeśli* $\Phi \vdash M\colon \alpha$ *i* $M \longrightarrow_\beta N$*, to* $\Phi \vdash N\colon \alpha$*.*

Normalization theorem.

Theorem 2.14. *(Twierdzenie o normalizacji) Jeśli* $\Phi \vdash M\colon \alpha$, *to* $\Phi \vdash N\colon \alpha$ *dla pewnego* λ*-termu* $N \in \beta\text{nf}$ *takiego, że* $M \longrightarrow_\beta N$.

2.1.3 Searching for proofs.

The sequent calculus NJ_β. Kolejne zastosowania reguły (I) dają

$$\frac{\begin{array}{c} A_1 \ldots A_n \\ | \\ B \end{array}}{A_1 \to \ldots \to A_n \to B} \quad \text{I}^*$$

Kolejne zastosowania reguły (E) dają

$$\frac{A_1 \to \ldots \to A_n \to B \qquad A_1 \qquad \ldots \qquad A_n}{B} \quad \text{E}^*$$

W szczególności, B staje się formułą atomową w (I*) oraz (E*). Kiedy żadna inferencja nie pojawia się powyżej $A_1 \to \ldots \to A_n \to B$ w (E*), jest ona w zbiorze Γ założeń. Kiedy A_i nie jest atomowa, mamy następujący dowód ponad A_i:

$$\frac{\begin{array}{c} A_{i_1} \ldots A_{i_k} \\ | \\ A_{i_0} \end{array}}{A_{i_1} \to \ldots \to A_{i_k} \to A_{i_0}} \quad \text{I}^*$$

Zdefiniujemy rachunek sekwentów NJ_β. Intuicyjnie, dowód w NJ_β otrzymuje się przez koncentrację wystąpień formuł atomowych jak B i A_{i_0}. Dowód w NJ_β odpowiada "długiej postaci normalnej" (ang. long normal form) w λ-rachunku z systemem typów prostych oraz "rozszerzonej postaci normalnej" (ang. expanded normal form) Systemu Dedukcji Naturalnej Prawitza. (Zamiast oznaczeń H oraz T z pracy [14] użyję oznaczeń B jak "body" oraz H jak "head".)

Definition 2.15. *Mając daną formułę* $X = A_1 \to \ldots \to A_n \to A$ *z atomową* A, *definiujemy* $B(X) = \{A_1, \ldots, A_n\}$ *oraz* $H(X) = A$. *Jeśli* $n = 0$, *to* $H(X) = \varnothing$.

Używamy liter greckich Φ, Δ etc. na oznaczenie zbiorów formuł. Φ, Δ oznacza sumę zbiorów Φ i Δ.

Definition 2.16. *System dedukcji naturalnej* NJ_β *jest zdefiniowany następującymi regułami:*

1. *(Aksjomat)*
$$\Phi \vdash A, \qquad \text{jeśli } A \in \Phi \text{ i } A \text{ jest atomowa}$$
2. *(Inferencja)*
$$\frac{B(A_1), \Phi \vdash H(A_1) \qquad \ldots \qquad B(A_n), \Phi \vdash H(A_n)}{\Phi \vdash A}$$
jeśli $A_1 \to \ldots \to A_n \to A$ *należy do* Φ *oraz* A *jest atomowa.*

Theorem 2.17. *Niech* Φ *będzie zbiorem formuł i* A *formułą.*

$$\Phi \vdash_{NJ} A \iff \Phi \vdash_{NJ_\beta} A$$

The deduction tree. Szukamy dowodu sekwentu $\Phi \vdash A$ (dowód formuły A dany jest przy $\Phi = \varnothing$). Algorytm Search bierze jako wejście sekwent $\Phi \vdash A$ z atomową A oraz listę ξ sekwentów śledzącą ścieżkę dowodu, aby wykryć zapętlenie. Algorytm zwraca drzewo sekwentów których wnioskami są formuły atomowe. Liście drzewa mają trzy rodzaje etykiet: "aksjomat", "pętla", "stop". Nazywamy je drzewem dedukcyjnym. Drzewem dowodowym albo dowodem nazywamy drzewo, w którym węzeł powstaje przez zastosowanie reguły inferencji systemu dedukcyjnego do jego synów, a liście powstają przez zastosowanie reguły "aksjomat".

Definition 2.18. Search$(\Phi \vdash A; \xi)$*:*

1. $A \in \Phi$. *Zwróć węzeł* $A \in \Phi$ *z etykietą "aksjomat".*
2. $A \notin \Phi$.
   a. $\Phi \vdash A \in \xi$. *Zwróć węzeł* $\Phi \vdash A$ *z etykietą "pętla".*
   b. $\Phi \vdash A \notin \xi$.
      i. Φ *nie zawiera żadnej* A_i *takiej, że* $T(A_i) = A$. *Zwróć węzeł* $\Phi \vdash A$ *z etykietą "stop".*
      ii. Φ *zawiera* A_i *takie, że* $T(A_i) = A$. *Niech* A_1, ..., A_m *będą wszystkimi takimi formułami oraz* $A_i = B_1^i \to ... \to B_{n_i}^i \to A\,(i = 1, ..., m)$. *Mamy* $H(A_i) = \{B_1^i, ..., B_{n_i}^i\}$. *Stosując algorytm rekurencyjnie, mamy*

$$u_j^i = \text{Search}((B(B_j^i), \Phi) \vdash H(B_j^i); (\Phi \vdash A) + \xi)$$

dla $i = 1, ..., m$ *oraz* $j = 1, ..., n_i$.

1. *Dla pewnych (niedeterminizm)* $1 \leqslant i \leqslant m$, *wszystkie* $u_1^i, ..., u_{n_i}^i$ *są dowodami. Zwróć dowód postaci*

$$\frac{u_1^i \quad ... \quad u_{n_i}^i}{\Phi \vdash A}$$

2. *Dla każdego* $i = 1, ..., m$, *pewne (niedeterminizm) drzewo* $u_{j_i}^i$ *nie jest dowodem* $(1 \leqslant j_i \leqslant n_i)$. *Zwróć drzewo, którego korzeniem jest* $\Phi \vdash A$ *a synami korzenia są* $u_{j_1}^1, ..., u_{j_m}^m$.

Niedeterminizm algorytmu nie dotyczy rodzaju odpowiedzi: dowód czy obalenie, a jedynie tego, który konkretnie dowód (które obalenie) zostanie zwrócony. ϵ oznacza ciąg pusty.

Theorem 2.19. *Niech* A *będzie formułą i* $t = \text{Search}(\varnothing \vdash A; \epsilon)$. *Jeśli* A *jest dowodliwa w logice intuicjonistycznej, to wszystkie liście* t *są etykietowane przez "aksjomat" i drzewo jest dowodem* A.

Definition 2.20. *Jeśli drzewo zawiera etykietę "stop" albo "pętla", nazywamy je drzewem obalenia.*

Theorem 2.21. *(Zatrzymywanie się poszukiwania) Dla każdego sekwentu* $\Phi \vdash A$, *algorytm* Search *zatrzymuje się (kończy pracę).*

Theorem 2.22. *Niech* A *będzie formułą,* $t = \text{Search}(\varnothing \vdash A; \epsilon)$. *Jeśli* A *nie jest dowodliwa w logice intuicjonistycznej, to* t *jest drzewem obalenia.*

Drzewo obalenia pozwala wtedy łatwo skonstruować kontr-model.

2.1.4 The algorithm generating λ-terms.

λ-termy w długiej postaci normalnej (ang. η-long β-normal form) są podstawą działania algorytmu Search: izomorfizm Curry'ego-Howarda wykorzystuje się w dowodzie twierdzenia 2.17. Oto wersja zwracająca termy; nie potrzebujemy wtedy drzewa obalenia. Niech Φ będzie zbiorem założeń, a A typem. Wtedy Search$(\Phi, A; \xi)$:

1. $x\colon A \in \Phi$. Zwróć x.
2. $A \notin \Phi$.
   a. $\Phi \vdash A \in \xi$. Wywołaj wyjątek "pętla".
   b. $\Phi \vdash A \notin \xi$.
      i. Φ nie zawiera żadnego $x\colon A_i$ takiego, że $T(A_i) = A$. Wywołaj wyjątek "stop".

ii. Φ zawiera x_i: A_i takie, że $T(A_i) = A$. Niech x_1: A_1, ..., x_m: A_m będą wszystkimi takimi przypisaniami typu oraz $A_i = B_1^i \to ... \to B_{n_i}^i \to A$ $(i = 1, ..., m)$. Mamy $H(A_i) = \{B_1^i, ..., B_{n_i}^i\}$. Niech $B(B_j^i) = \{B_j^{i,1}, ..., B_j^{i,k_{i,j}}\}$. Niech $x_j^{i,k}$ będą nowymi zmiennymi. Stosując algorytm rekurencyjnie, mamy

$$u_j^i = \text{Search}((\{x_j^{i,1}: B_j^{i,1}, ..., x_j^{i,k_{i,j}}: B_j^{i,k_{i,j}}\}, \Phi), H(B_j^i); (\Phi \vdash A) + \xi)$$

dla $i = 1, ..., m$ oraz $j = 1, ..., n_i$.

1. Dla pewnych (niedeterminizm) $1 \leqslant i \leqslant m$, wszystkie u_1^i, ..., $u_{n_i}^i$ są obliczone. Zwróć term postaci

$$(...(x_i\,(\lambda x_1^{i,1}...\lambda x_1^{i,k_{i,1}}.u_1^i)) \qquad ... \qquad (\lambda x_1^{i,1}...\lambda x_1^{i,k_{i,1}}.u_{n_i}^i))$$

2. Dla każdego $i = 1$, ..., m, pewne $u_{j_i}^i$ nie jest obliczone (wywołano wyjątek) $(1 \leqslant j_i \leqslant n_i)$. Wywołaj wyjątek.

2.2 Damas-Milner Type System: type inference and term generation.

Generowanie termu dla danego typu jest "odwrotnością" inferencji typu dla danego termu. Obie operacje są dowodzeniem, w pierwszym przypadku, że typ jest zamieszkały, w drugim, że term jest poprawnie typowany (ang. well-typed). Regułami dowodzenia są reguły obranego systemu typów, które łączą termy i typy. W tym rozdziale opieram się na książce [11], rozdział 1. "A polymorphic applicative language".

Wprowadźmy oznaczenia:

Definition 2.23. *Generalization*

$$\boldsymbol{G}(\tau, E) \;=\; \forall\alpha_1, ..., \alpha_n.\tau$$

where $\alpha_1, ...\alpha_n$ *are exactly these free variables of* τ*, which do not occur in* E*.*

If $E = [x_1\colon \tau_1; x_2\colon \tau_2; ...; x_k\colon \tau_k]$*, and* $1 \leqslant n \leqslant k$ *then we write* $E(n) = x_n\colon \tau_n$ *and* $E(x_n) = \tau_n$*. For length we write* $\bar{E} = k$*.*

Definition 2.24. *Concretization of a type for a variable from environment:* τ *is a concretization of a type of variable* x *from environment* E

$$\tau \leqslant E(x) \;\equiv\; E = ...; x\colon \forall\alpha_1...\alpha_n.\sigma; ... \text{ and } \tau = [\tau_1/\alpha_1; ...; \tau_n/\alpha_n]\sigma$$

for some $\tau_1, ..., \tau_n$*. A type* σ *is more general than* τ

$$\tau \leqslant \sigma \;\equiv\; \tau = \forall\gamma_1...\gamma_m.\tau' \wedge \sigma = \forall\alpha_1...\alpha_n.\sigma' \wedge \tau' = [\tau_1/\alpha_1; ...; \tau_n/\alpha_n]\sigma'$$

for some $\tau_1, ...\tau_n$*. Substitution* $T = [\tau_1/\alpha_1; ...; \tau_n/\alpha_n]$ *denotes replacement of free occurrences of variables* α_i *by corresponding types* τ_i*.*

Definition 2.25. *Typing rules for the Mini-ML language:*

1. *VAR: Variables.*
$$\frac{\tau \leqslant E(x)}{E \vdash x\colon \tau}$$
2. *FIX: Functions.*
$$\frac{E.f\colon \sigma \to \tau.x\colon \sigma \vdash e\colon \tau}{E \vdash (\text{fix } fx.e)\colon \sigma \to \tau}$$
3. *APP: Applications.*
$$\frac{E \vdash e\colon \sigma \to \tau \qquad E \vdash e'\colon \sigma}{E \vdash ee'\colon \tau}$$
4. *LET: Local bindings.*
$$\frac{E \vdash e'\colon \sigma \qquad E.x\colon \boldsymbol{G}(\sigma, E) \vdash e\colon \tau}{E \vdash (\text{let } x = e' \text{ in } e)\colon \tau}$$

Dopuszczam n-arne konstruktory typów, np. nularny int, unarny list α, binarny pair(α, β). O stałych można założyć, że są zmiennymi początkowego środowiska. Za [Leroy92] pomijam konstrukcję λ-abstrakcji ABS

$$\frac{E.x\colon \sigma \vdash e\colon \tau}{E \vdash \lambda x.e\colon \sigma \to \tau}$$

ponieważ jest ona równoważna FIX dla nowej nazwy f nie występującej w e. W rozważanych w tym rozdziale systemach typów (systemie Mini-ML, rozszerzonym następnie o indukcyjne typy danych) kwantyfikator występuje tylko na zewnątrz, tzn. każdy typ ma postać $\forall\alpha_1...\alpha_n.\tau$ (lub τ dla $n = 0$) oraz $\forall$ nie występuje w τ. Jeśli typ zadany dla generowanego termu ma taką postać, to do algorytmu generującego przekazujemy $[d_1/\alpha_1; ...; d_n/\alpha_n]\tau$, gdzie d_i są nowymi stałymi typu (nularnymi konstruktorami typu). W algorytmach zakładamy, że kwantyfikator nie występuje w zadanym typie.

Lemma 2.26. *When $E \vdash \tau$ and type constants $d_1, ..., d_n$ do not occur in E, then*

$$E \vdash \forall \alpha_1 ... \alpha_n . [\alpha_1/d_1; ...; \alpha_n/d_n]\tau$$

2.2.1 Type inference algorithm $\mathcal{W}$ and term generation algorithm $\mathcal{C}$.

Przyjrzyjmy się algorytmowi $\mathcal{W}$ tak jak jest on podany w [11]. Kolejne kroki algorytmu odpowiadają kolejnym konstrukcjom języka. Jeśli E jest zbiorem założeń (środowiskiem), e wyrażeniem, V zbiorem nowych zmiennych typu i $\mathcal{W}$ kończy z powodzeniem, to $\mathcal{W}(E, e, V) = (T, \tau, V')$, gdzie τ jest najogólniejszym typem e, T takie że TE jest odpowiednim podstawieniem pod zmienne w E, V' są pozostałymi zmiennymi. $U = \boldsymbol{U}(\tau, \sigma)$ jest (najogólniejszym) unifikatorem typów τ, σ.

Definition 2.27. *$\mathcal{W}(E, e, V) = (T, \tau, V')$, where match e with:*

1. *$e = x$ (VAR: Variables). $T = \varnothing$ and for $x: \forall \alpha_1 ... \alpha_n . \sigma \in E$ let*

$$\begin{aligned} \tau &= [\vec{\beta}/\vec{\alpha}]\sigma \\ V' &= V' \setminus \vec{\beta} \\ T &= \varnothing \end{aligned}$$

2. *$e = \mathrm{fix}\, fx.e_1$ (FIX: Functions). Let*

$$\begin{aligned} V &= \{\beta, \beta_1\} \dot{\cup} V'' \\ (R, \rho, V') &= \mathcal{W}(E.f: \beta_1 \to \beta.x: \beta_1, e_1, V'') \\ U &= \boldsymbol{U}(R\beta, \rho) \\ T &= UR \\ \tau &= UR(\beta_1 \to \beta) \end{aligned}$$

3. *$e = fg$ (APP: Applications). Let*

$$\begin{aligned} (R, \rho, V_1) &= \mathcal{W}(E, f, V) \\ (S, \sigma, V_2) &= \mathcal{W}(RE, g, V_1) \\ U &= \boldsymbol{U}(S\rho, \sigma \to \beta) \\ V' &= V_2 \setminus \{\beta\} \\ T &= USR \\ \tau &= U\beta \end{aligned}$$

4. *$e = \mathrm{let}\, x = f \,\mathrm{in}\, g$ (LET: Local bindings). Let*

$$\begin{aligned} (R, \rho, V_1) &= \mathcal{W}(E, f, V) \\ (S, \sigma, V') &= \mathcal{W}(RE.x: \boldsymbol{G}(\rho, RE), g, V_1) \\ T &= SR \\ \tau &= \sigma \end{aligned}$$

Algorytm ten najpierw rozgałęzia się na poszczególne zmienne, a następnie unifikuje zgromadzoną informację. Algorytm generujący termy będzie rozdzielał informacje o typie, tak że gdy Rzdecyduje się" na zmienną, jej typ będzie już (w odpowiednim stopniu) określony. Algorytm zawodzi dla danej ścieżki wyborów, jeśli nie odnajdzie odpowiedniej zmiennej w środowisku. Zwrócony term ma typ bardziej konkretny niż typ zadany, zwrócone jest też odpowiednie podstawienie. Jeśli E jest środowiskiem, τ typem, $\vec{w}$ ciągiem liczb naturalnych nazywanym ciągiem albo ścieżką wyborów, to albo $\mathcal{C}(E, \tau, V, \vec{w}) = (T, e, V', \vec{w}')$, albo $\mathcal{C}$ nie jest określone, gdzie e jest typu $T\tau$ w środowisku $T\,E$, V' są niewykorzystanymi zmiennymi typu z V, $\vec{w}'$ są niewykorzystanymi wyborami: $w'_i = w_{i+k}$ dla pewnego k. Zazwyczaj będziemy pomijać ciąg wyborów i pisać $\mathcal{C}(E, \tau, V) = (T, e, V')$, co oznacza, że równość zachodzi dla pewnego ustalonego ciągu wyborów $\vec{w}$ i zakładając odpowiedni $\vec{w}'$. Można przyjąć dowolny nieskończony zbiór zmiennych V i pisać $\mathcal{C}(E, \tau) = (T, e)$ zakładając odpowiedni V'.

Definition 2.28. $\mathcal{C}(E,\tau,V,\vec{w})=(T,e,V',\vec{w}')$, *where for* $(w_1 \bmod 4)$ *equal*

1. *VAR: Variables. For* $E(w_2 \bmod \bar{E})=x\colon\forall\alpha_1...\alpha_n.\sigma$ *let*

$$\begin{aligned} U &= \boldsymbol{U}(\tau,[\beta_1/\alpha_1]...[\beta_n/\alpha_n]\sigma)\\ V &= \{\beta_i|1\leqslant i\leqslant n\}\dot{\cup}V'\\ e &= x\\ T &= U\\ w_i' &= w_{i+2}\end{aligned}$$

   If the unifier does not exist, $\mathcal{C}$ *is undefined.*

2. *FIX: Functions. If* $\tau=\sigma\to\rho$, *let*

$$\begin{aligned}(R,e_1,V',\vec{w}_1) &= \mathcal{C}(E.f\colon\tau.x\colon\sigma,\rho,V,(w_{i+1}))\\ T &= R\\ e &= \text{fix}\, fx.e_1\\ \vec{w}' &= \vec{w}_1\end{aligned}$$

   where f,x *are new variables. If* $\tau=\alpha$, *where* α *is a type variable, for* $\beta_1,\beta\in V$ *let*

$$\begin{aligned}R &= [\beta_1\to\beta/\alpha]\\ (S,e_1,V',\vec{w}_1) &= \mathcal{C}(R\,E.f\colon\beta_1\to\beta.x\colon\beta_1,\beta,V\setminus\{\beta_1,\beta\},(w_{i+1}))\\ T &= S\,R\\ e &= \text{fix}\, fx.e_1\end{aligned}$$

   where f,x *are new variables. If* τ *has a different form,* $\mathcal{C}$ *is undefined.*

3. *APP: Applications. For* $\beta\in V$ *let*

$$\begin{aligned}(R,f,V_1,\vec{w}_1) &= \mathcal{C}(E,\beta\to\tau,V\setminus\{\beta\},(w_{i+1}))\\ (S,g,V',\vec{w}_2) &= \mathcal{C}(R\,E,R\beta,V_1,\vec{w}_1)\\ T &= S\,R\\ e &= fg\\ \vec{w}' &= \vec{w}_2\end{aligned}$$

4. *LET: Local bindings. For* $\beta\in V$ *let*

$$\begin{aligned}(R,f,V_1,\vec{w}_1) &= \mathcal{C}(E,\beta,V\setminus\{\beta\},(w_{i+1}))\\ (S,g,V',\vec{w}_2) &= \mathcal{C}(R\,E.x\colon\boldsymbol{G}(R\beta,R\,E),R\tau,V_1,\vec{w}_1)\\ T &= S\,R\\ e &= \text{let}\, x=f\,\text{in}\, g\\ \vec{w}' &= \vec{w}_2\end{aligned}$$

Algorytmy dokonują podstawień w środowisku dla komunikowania informacji o konstruowanym typie: wynikowym typie wyrażenia dla algorytmu inferencji oraz konkretyzacji zadanego typu dla algorytmu $\mathcal{C}$.

2.2.1.1 Soundness.

Dla algorytmów generujących zadany język, własność *algorytm* $\subseteq$ *język* nazywamy poprawnością, a własność *język* $\subseteq$ *algorytm* pełnością. Dowody dla $\mathcal{W}$ pochodzą z [11]. Pokażemy najpierw poprawność:

Theorem 2.29. *Let* e *be an expression,* E *an environment,* V *a set of type variables. If* $(T,\tau,V')=\mathcal{W}(E,e,V)$ *is defined, then we can derive* $TE\vdash e\colon\tau$.

Proof. Przez indukcję względem budowy termu:

1. $e=x$ (VAR: Variables). Mamy $T=\varnothing$ i $\tau=[\beta_1/\alpha_1;\,...;\,\beta_n/\alpha_n]\sigma$, gdzie $E(x)=\forall\alpha_1...\alpha_n.\sigma$, czyli $\tau\leqslant E(x)$, skąd $E\vdash x\colon\tau$.

2. $e = \text{fix}\, f x.e_1$ (FIX: Functions). Z zał. ind. mamy:

$$\begin{aligned} (R, \rho, V_1) &= \mathcal{W}(E, e_1, V \setminus \{\beta_1, \beta\}) \\ R(E.f\colon \beta_1 \to \beta.x\colon \beta_1) &\vdash e_1\colon \rho \end{aligned}$$

Z lematu o podstawianiu, podstawiamy U po obu stronach, korzystamy z $T = UR$:

$$T(E.f\colon \beta_1 \to \beta.x\colon \beta_1) \vdash e_1\colon U\rho$$

gdzie U jest unifikatorem $R\beta$ i ρ, mamy więc $T\beta = U\rho$. Pokazaliśmy w ten sposób, że:

$$TE.f\colon (T\beta_1 \to T\beta).x\colon T\beta_1 \vdash e_1\colon T\beta$$

Stosując regułę typizowania funkcji (FIX), mamy:

$$TE \vdash (\text{fix}\, f x \,\text{in}\, e_1)\colon T\beta_1 \to T\beta$$

To oczekiwany rezultat, ponieważ $\tau = T(\beta_1 \to \beta) = T\beta_1 \to T\beta$.

3. $e = f\,g$ (APP: Applications). Z zał. ind. mamy:

$$\begin{aligned} RE &\vdash f\colon \rho \\ SRE &\vdash g\colon \sigma \end{aligned}$$

Stosując lemat o podstawianiu:

$$SRE \vdash f\colon S\rho$$

Ponieważ U unifikuje $S\rho$ i $\sigma \to \beta$, mamy

$$USRE \vdash f\colon U(\sigma \to \beta)$$

Biorąc $T = USR$, z lematu o podstawianiu (drugie równanie) mamy:

$$\begin{aligned} TE &\vdash f\colon U\sigma \to U\beta \\ TE &\vdash g\colon U\sigma \end{aligned}$$

Teraz z reguły typowania APP mamy

$$TE \vdash f\,g\colon U\beta$$

co jest oczekiwanym wynikiem $\tau = U\beta$.

4. $e = \text{let}\, x = f \,\text{in}\, g$ (LET: Local bindings). Z zał. ind. mamy:

$$\begin{aligned} RE &\vdash f\colon \rho \\ S(RE.x\colon \boldsymbol{G}(\rho, RE)) &\vdash g\colon \sigma \end{aligned}$$

Przemianowując zmienne związane w $\boldsymbol{G}(\rho, RE)$ można pokazać

$$\boldsymbol{G}(S\rho, SRE) = S(\boldsymbol{G}(\rho, RE))$$

Biorąc $T = SR$ mamy:

$$\begin{aligned} TE &\vdash f\colon S\rho \\ TE.x\colon \boldsymbol{G}(S\rho, TE) &\vdash g\colon \sigma \end{aligned}$$

Stosując regułę typowania LET:

$$TE \vdash \text{let}\, x = f \,\text{in}\, g\colon \sigma$$

i $\tau = \sigma$ jak oczekiwaliśmy.

□

Theorem 2.30. *Let τ be a type, E an environment, V a set of type variables. If $(T, e, V') = \mathcal{C}(E, \tau, V)$ is defined, we can derive $TE \vdash e\colon T\tau$.*

Proof. Przez indukcję na złożoność generowanego termu:

1. $e = x$ (VAR: Variables). Gdy $\sigma \leqslant E(x)$, to $T\sigma \leqslant TE(x)$. Ponieważ T jest unifikatorem σ oraz τ, a więc $T\tau \leqslant TE(x)$, stąd z reguły typowania VAR mamy $TE \vdash x\colon T\tau$.

2. $e = \text{fix}\, fx.e_1$ (FIX: Functions). $\tau = \sigma \to \rho$ albo $\tau = \alpha$. W pierwszym przypadku, korzystając z zał. ind. mamy

$$\begin{aligned} R(E.f\colon \tau.x\colon \sigma) &\vdash e_1\colon R\rho \\ RE.f\colon R\sigma \to R\rho.x\colon R\sigma &\vdash e_1\colon R\rho \end{aligned}$$

Stosując regułę typizowania funkcji (FIX), mamy:

$$RE \vdash (\text{fix}\, fx \,\text{in}\, e_1)\colon R\sigma \to R\rho$$

To oczekiwany rezultat, ponieważ $T\tau = R\tau = R\sigma \to R\rho$ dla $T = R$. W drugim przypadku analogicznie: korzystając z zał. ind. mamy

$$\begin{aligned} S(RE.f\colon \beta_1 \to \beta.x\colon \beta_1) &\vdash e_1\colon S\beta \\ SRE.f\colon S\beta_1 \to S\beta.x\colon S\beta_1 &\vdash e_1\colon S\beta \end{aligned}$$

Stosując regułę typizowania funkcji (FIX), mamy:

$$SE \vdash (\text{fix}\, fx \,\text{in}\, e_1)\colon S\beta_1 \to S\beta$$

To oczekiwany rezultat, ponieważ $T\tau = SR\tau = S\beta_1 \to S\beta$ dla $T = SR$.

3. $e = fg$ (APP: Applications). Z zał. ind. mamy:

$$\begin{aligned} RE &\vdash f\colon R(\beta \to \tau) \\ SRE &\vdash g\colon SR\beta \end{aligned}$$

Stosując lemat o podstawianiu, biorąc $T = SR$:

$$TE \vdash f\colon T\beta \to T\tau$$

Teraz z reguły typowania APP mamy

$$TE \vdash fg\colon T\tau$$

co jest oczekiwanym wynikiem.

4. $e = \text{let}\, x = f \,\text{in}\, g$ (LET: Local bindings). Z zał. ind. mamy:

$$\begin{aligned} RE &\vdash f\colon R\beta \\ S(RE.x\colon \boldsymbol{G}(R\beta, RE)) &\vdash g\colon SR\tau \end{aligned}$$

Przemianowując zmienne związane w $\boldsymbol{G}(R\beta, RE)$ można pokazać:

$$\boldsymbol{G}(SR\beta, SRE) = S(\boldsymbol{G}(R\beta, RE))$$

Biorąc $T = SR$ mamy:

$$\begin{aligned} TE &\vdash f\colon S\rho \\ TE.x\colon \boldsymbol{G}(SR\beta, TE) &\vdash g\colon T\tau \end{aligned}$$

Stosując regułę typowania LET:

$$TE \vdash (\text{let}\, x = f \,\text{in}\, g)\colon T\tau$$

□

2.2.1.2 Completeness.

Teraz pokażemy pełność, najpierw algorytmu $\mathcal{W}$, za [11].

Theorem 2.31. *Let e be an expression, E an environment, V an infinite set of variables such, that $V \cap \boldsymbol{F}(E) = \varnothing$. If there exists type τ' and substitution T' such that $T'E \vdash e\colon \tau'$, then $(T, \tau, V') = \mathcal{W}(E, e, V)$ is defined, and there exists a substitution P such that*

$$\tau' = P\tau \qquad \textit{and} \qquad T' = PT \textit{ outside } V$$

Proof. Przez indukcję względem złożoności e:

1. $e = x$ (VAR: Variables). Ponieważ $T'E \vdash x\colon \tau'$, mamy $x \in \mathrm{Dom}(T'E)$ i $\tau' \leqslant T'E(x)$. $x \in \mathrm{Dom}(E)$, więc $\mathcal{W}(E, x, V)$ jest określony i zwraca
$$\begin{aligned} \tau &= [\beta_1/\alpha_1; ...; \beta_n/\alpha_n]\tau_x \\ T &= \varnothing \\ V' &= V \setminus \{\beta_1, ..., \beta_n\} \end{aligned}$$
Niech $E(x) = \forall\alpha_1...\alpha_n.\tau_x$, gdzie $\alpha_i \in V'$ nie występują w T'. Mamy $T'E(x) = \forall\alpha_1...\alpha_n.T'\tau_x$. Niech R będzie podstawieniem pod α_i t. że $\tau' = RT'\tau_x$. Weźmy
$$P = RT'[\alpha_1/\beta_1; ...; \alpha_n/\beta_n]$$
Mamy $P\tau = RT'\tau_x = \tau'$. Co więcej, $\alpha \notin V$ nie są ani α_i, ani β_i, więc $P\alpha = RT'\alpha = T'\alpha$. To daje oczekiwany rezultat, bo $PT = P$ dla $T = \varnothing$.

2. $e = \mathrm{fix}\ fx.e_1$ (FIX: Functions). Wyprowadzenie typu τ' kończy się zastosowaniem reguły FIX:
$$\frac{T'E.f\colon(\rho_1' \to \rho').x\colon\rho_1' \vdash e_1\colon\rho'}{T'E \vdash (\mathrm{fix}\ fx\ \mathrm{in}\ e_1)\colon\rho_1' \to \rho'}$$
Wybierzmy $\beta_1, \beta \in V$ jak w algorytmie. Określmy środowisko E_1 i podstawienie R' przez
$$\begin{aligned} E_1 &= E.f\colon\beta_1 \to \beta.x\colon\beta_1 \\ R' &= T'[\rho_1'/\beta_1; \rho'/\beta] \end{aligned}$$
Mamy $R'E_1 = T'E.f\colon\rho_1' \to \rho'.x\colon\rho_1'$. Stosując zał. ind. do e_1, E_1, R', ρ' mamy:
$$\begin{aligned} (R, \rho, V_1) &= \mathcal{W}(E_1, e_1, V \setminus \{\beta_1, \beta\}) \\ \rho' &= P_1\rho \\ R' &= P_1R \text{ poza } V \setminus \{\beta_1, \beta\} \end{aligned}$$
W szczególności, $P_1R\beta = R'\beta = \rho'$, stąd P_1 jest unifikatorem $R\beta$ i ρ. Stąd istnieje najbardziej ogólny unifikator tych typów: nazwijmy go U – i $\mathcal{W}(E, e, V)$ (tzn. wywołanie $\boldsymbol{U}(R\beta, \rho)$) jest dobrze określony. Niech P będzie podstawieniem t. że $P_1 = PU$. Pokażemy teraz, że P spełnia żądanie twierdzenia. Mamy:
$$\begin{aligned} P\tau = {} & PUR(\beta_1 \to \beta) && \text{z definicji } \tau \text{ w algorytmie} \\ = {} & P_1R(\beta_1 \to \beta) && \text{z definicji } P \\ = {} & R'(\beta_1 \to \beta) && \text{bo } \beta_1, \beta \notin V \setminus \{\beta_1, \beta\} \\ = {} & \rho_1' \to \rho' && \text{przez konstrukcję } R' \end{aligned}$$
(T' nie działa na ρ_1' i ρ'). Co więcej, dla każdej zmiennej $\gamma \notin V$:
$$\begin{aligned} PT\gamma = {} & PUR\gamma && \text{z definicji } T \text{ w algorytmie} \\ = {} & P_1R\gamma && \text{z definicji } P \\ = {} & R'\gamma && \text{bo } \gamma \notin V \\ = {} & T'\gamma && \text{bo } \gamma \notin V \text{ pociąga } \gamma \neq \beta_1 \wedge \gamma \neq \beta \end{aligned}$$
c.b.d.o.

3. $e = fg$ (APP: Applications). Wyprowadzenie τ' kończy się z
$$\frac{T'E \vdash f\colon\sigma' \to \nu' \qquad T'E \vdash g\colon\sigma'}{T'E \vdash fg\colon\nu'}$$
Stosując zał. ind. do $f, E, \sigma' \to \nu', T'$ otrzymujemy
$$\begin{aligned} (R, \rho, V_1) &= \mathcal{W}(E, f, V) \\ \sigma' \to \nu' &= P_1\rho \\ T' &= P_1R \text{ poza } V \end{aligned}$$

W szczególności, $T'E = P_1 R\,E$. Stosujemy zał. ind. do $g, R\,E, \sigma', P_1, V_1$. P_1 jest dobre, bo $T'E = P_1 R\,E$.

$$\begin{aligned}(S, \sigma, V_2) &= \mathcal{W}(R\,E, g, V_1)\\ \sigma' &= P_2\sigma\\ P_1 &= P_2 S \text{ poza } V_1\end{aligned}$$

Mamy $P_1\rho = P_2 S\rho$. Niech $P_3 = P_2[\nu'/\beta]$. Mamy:

$$\begin{aligned}P_3 S\rho &= P_2 S\rho = P_1\rho = \sigma' \to \nu'\\ P_3(\sigma \to \beta) &= P_2\sigma \to \nu' = \sigma' \to \nu'\end{aligned}$$

P_3 jest więc unifikatorem $S\rho$ i $\sigma \to \beta$. Te typy mają więc najogólniejszy unifikator, $\mathcal{W}(E,\ f\ g,\ V)$ jest dobrze określone. W dodatku, mamy $P_3 = P_4 U$ dla pewnego P_4. Teraz pokażemy, że $P = P_4$ spełnia wymagania stwierdzenia. Mamy:

$$P\tau = P_4 U\beta = P_3\beta = \nu'$$

a dla wszystkich $\gamma \notin V$

$$\begin{aligned}PT\gamma = \ & P_4 U S R\gamma && \text{z definicji } T\\ = \ & P_3 S R\gamma && \text{z definicji } P_4\\ = \ & P_2 S R\gamma && \text{bo } \gamma \neq \beta, \beta \text{ nie wyst. w } R, S\\ = \ & P_1 R\gamma && \text{bo } R\gamma \notin V_1\\ = \ & T'\gamma && \text{bo } \gamma \notin V\end{aligned}$$

4. $e = \text{let } x = f \text{ in } g$ (LET: Local bindings). Wyprowadzenie τ' kończy się z:

$$\frac{T'E \vdash f : \rho' \qquad T'E.x : \boldsymbol{G}(\rho', T'E) \vdash g : \sigma'}{T'E \vdash (\text{let } x = f \text{ in } g) : \sigma'}$$

Stosujemy zał. ind. do f, E, V, ρ', T'. Otrzymujemy

$$\begin{aligned}(R, \rho, V_1) &= \mathcal{W}(E, f, V)\\ \rho' &= P_1\rho\\ T' &= P_1 R \text{ poza } V\end{aligned}$$

W szczególności, $T'E = P_1 R\ E$. (Teraz sfaktoryzujemy środowisko wyprowadzenia σ'.) Łatwo sprawdzić, że $P_1\boldsymbol{G}(\rho,\ R\ E)$ jest bardziej (nie mniej) ogólne niż $\boldsymbol{G}(P_1\rho,\ P_1 R\ E)$.[2.1] Ponieważ możemy wyprowadzić

$$T'E.x : \boldsymbol{G}(\rho', P_1 R\,E) \vdash g : \sigma'$$

z następującego lematu:

Lemma 2.32. *Niech E, E' będą środowiskami t. że $\text{Dom}(E) = \text{Dom}(E')$ i $E'(x) \geqslant E(x)$ dla wszystkich $x \in \text{Dom}(E)$. Jeśli $E \vdash e : \tau$, to $E' \vdash e : \tau$.*

wynika, że możemy również wyprowadzić

$$T'E.x : P_1\boldsymbol{G}(\rho, R\,E) \vdash g : \sigma'$$

czyli

$$P_1(R\,E.x : \boldsymbol{G}(\rho, R\,E)) \vdash g : \sigma'$$

Stosujemy teraz zał. ind. do $g, R\,E.x : \boldsymbol{G}(\rho, R\,E), \sigma', P_1$, otrzymując

$$\begin{aligned}(S, \sigma, V') &= \mathcal{W}(R\,E.x : \boldsymbol{G}(\rho, R\,E), g, V_1)\\ \sigma' &= P_2\sigma\\ P_1 &= P_2 S \text{ poza } V_1\end{aligned}$$

2.1. Niech $P_1\boldsymbol{G}(\rho,\ R\ E) = P_1\forall\alpha_1 ... \alpha_n.\rho_1 = \forall\alpha_1 ... \alpha_n.P_1\rho_1$; natomiast $\boldsymbol{G}(P_1\rho,\ P_1 R\ E) = \boldsymbol{G}(\forall\alpha_i ... \alpha_n.P_1\rho_1,\ P_1 R\,E)$ oraz $\alpha_1 ... \alpha_{i-1}$ nie występują w $R\,E$; zmienne nie występujące w $P_1 R\,E$` które występują w $R\,E$, to te, pod które podstawia P_1; ale one nie występują w $P_1\rho_1$.

Algorytm bierze $\tau = \sigma$ i $T = S\,R$. Pokażemy, że $P = P_2$ spełnia wymagania stwierdzenia. Mamy $P\tau = \sigma'$. I jeśli $\gamma \notin V$, czyli $\gamma \notin V_1$:

$$\begin{aligned} PT\gamma &= P_2 S R\gamma \quad \text{z definicji } T \\ &= P_1 R\gamma \quad \text{bo } R\gamma \notin V_1 \\ &= T'\gamma \quad \text{bo } \gamma \notin V \end{aligned}$$

Stąd $T' = PT$ poza V, jak oczekiwano.

□

Teraz sprawdzimy pełność algorytmu $\mathcal{C}$. Wprowadźmy najpierw inne pojęcie ogólności typu:

Definition 2.33. *Type σ is more general than τ*

$$\tau \preccurlyeq \sigma \equiv \textit{ there exists substitution } P\colon P\sigma = \tau$$

Substitution S is more general than R

$$\begin{aligned} R \preccurlyeq S &\equiv \textit{ there exists substitution } P\colon PS = R \\ &\equiv \mathrm{Dom}(S) \subseteq \mathrm{Dom}(R) \textit{ i } \forall \alpha \in \mathrm{Dom}(S)\colon R\alpha \preccurlyeq S\alpha \end{aligned}$$

Theorem 2.34. *Let e be an expression, E an environment. For any type τ and substitution T' such that $T'E \vdash e\colon T'\tau$ and $\boldsymbol{F}(T') \cap \mathrm{Dom}(T') = \varnothing$, for infinite set of variables V disjoint with $\boldsymbol{F}(E)$ and $\boldsymbol{F}(T')$, for some path of choices the following holds: $(T, e, V') = \mathcal{C}(E, \tau, V)$ and $T' \preccurlyeq T$ outside V.*

Proof. Przez indukcję względem złożoności e:

1. $e = x$ (VAR: Variables). Ponieważ $T'E \vdash x\colon \tau'$, mamy $x \in \mathrm{Dom}(E)$ i $T'\tau \leqslant T'E(x)$. Wybieramy VAR w $\mathcal{C}(E, \tau, V)$ i zmienną x z E. Niech $E(x) = \forall\alpha_1 ... \alpha_n.\sigma$. Ponieważ $T'\tau \leqslant T'E(x)$, więc istnieje unifikator

$$U = \boldsymbol{U}(\tau, [\beta_1/\alpha_1; ... \beta_n/\alpha_n]\sigma)$$

gdzie $\beta_i \in V$, $\beta_i \notin V'$. Ponieważ U jest najogólniejszym unifikatorem a T' rozszerzony o podstawienia nad β_i (oznaczone wcześniej $[\tau_i/\alpha_i]$, ale $\alpha_i \leftrightarrow \beta_i$) też jest unifikatorem, więc $T' = SU$ poza V. Tak więc dla $T = U$ mamy $T' \preccurlyeq T$ poza V.

2. $e = \mathrm{fix}\, fx.e_1$ (FIX: Functions). Wyprowadzenie typu $T'\tau$ kończy się z

$$\frac{T'E.f\colon (\sigma' \to \rho').x\colon \sigma' \vdash e_1\colon \rho'}{T'E \vdash (\mathrm{fix}\, fx \,\mathrm{in}\, e_1)\colon \sigma' \to \rho'}$$

Typ τ jako ogólniejszy ma postać $\sigma \to \rho$ albo α (gdzie α jest zmienną typu). Rozpatrzmy najpierw przypadek $\sigma \to \rho$. Z zał. ind. dla T' i ρ' ($T'\rho = \rho'$) mamy: istnieje ciąg wyborów dający

$$(R, e_1, V_1) = \mathcal{C}(E.f\colon \tau.x\colon \sigma, \rho, V)$$

oraz $T' \preccurlyeq R$ poza V. Teza wynika z $T = R$.

W przypadku $\tau = \alpha$, $R = [\beta_1 \to \beta/\alpha]$, gdzie $\beta_1, \beta \in V$ wprowadzone przez algorytm. Ponieważ $T'\alpha = \sigma' \to \rho'$, T' można zmodyfikować: $U = \boldsymbol{U}(T'\alpha, \beta_1 \to \beta)$, $T'' = (T' \setminus \{\alpha \mapsto \sigma' \to \rho'\})U$. Z zał. ind. dla T'' i $\beta_1 \to \beta$ mamy

$$(S, e_1, V_1) = \mathcal{C}(R\,E.f\colon \beta_1 \to \beta.x\colon \beta_1, \beta, V \setminus \{\beta_1, \beta\})$$

oraz $T'' \preccurlyeq S$ poza $V \setminus \{\beta_1, \beta\}$. Ale $T''(\beta_1 \to \beta) = T'\alpha$, więc $T''R = T'$, stąd $T' \preccurlyeq S\,R = T$.

3. $e = fg$ (APP: Applications). Wyprowadzenie $T'\tau$ kończy się z

$$\frac{T'E \vdash f\colon \sigma' \to \tau' \qquad T'E \vdash g\colon \sigma'}{T'E \vdash fg\colon \tau'}$$

Niech $T_1 = [\sigma'/\beta]T'$ (co nie stwarza problemów, bo β nie wyst. w T'), stosując zał. ind. do T_1 i $\beta \to \tau$ (zachodzi $T_1(\beta \to \tau) = T_1\beta \to T_1\tau = \sigma' \to T'\tau = \sigma' \to \tau'$)

$$(R, f, V_1) = \mathcal{C}(E, \beta \to \tau, V \setminus \{\beta\})$$

oraz $T_1 \preccurlyeq R$ poza $V \setminus \{\beta\}$. Niech T_2: $T_1 = T_2R$ poza $V \setminus \{\beta\}$, zastosujmy zał. ind. do T_2, $RE, R\beta$ ($T_2R\beta = T_1\beta = \sigma'$):

$$(S, g, V_2) = \mathcal{C}(RE, R\beta, V_1)$$

$T_2 \preccurlyeq S$ poza V_1 (niech T_3: $T_2 = T_3S$ poza V_1), a więc $T_1 \preccurlyeq SR$ poza $V \setminus \{\beta\}$ (bo $V_1 \subseteq V \setminus \{\beta\}$), czyli $T' \preccurlyeq \mathrm{SR} = T$ poza V (bo $\beta \in V$).

4. $e = \mathrm{let}\, x = f \,\mathrm{in}\, g$ (LET: Local bindings). Wyprowadzenie $T'\tau$ kończy się z

$$\frac{T'E \vdash f : \rho' \qquad T'E.x : \boldsymbol{G}(\rho', T'E) \vdash g : \tau'}{T'E \vdash (\mathrm{let}\, x = f \,\mathrm{in}\, g) : \tau'}$$

Stosujemy zał. ind. do $[\rho'/\beta]T'$

$$(R, f, V_1) = \mathcal{C}(E, \beta, V \setminus \{\beta\})$$

mamy $[\rho'/\beta]T' \preccurlyeq R$, czyli $[\rho'/\beta]T' = T_1R$, poza $V \setminus \{\beta\}$. Analogicznie jak przy dowodzie dla algorytmu $\mathcal{W}$ pokazuje się, że możemy wyprowadzić

$$T'E.x : T_1\boldsymbol{G}(R\beta, RE) \vdash g : \tau'$$

Stosujemy więc zał. ind. do powyższego, biorąc $E' = RE.x : \boldsymbol{G}(R\beta, RE)$

$$(S, g, V_2) = \mathcal{C}(RE.x : \boldsymbol{G}(R\beta, RE), \tau, V_1)$$

oraz $T_1 \preccurlyeq S$ poza V_1. Stąd $T' \preccurlyeq SR = T$ poza V, bo $V_1 \subset V$ i $\beta \in V$.

□

Zobaczmy, że $\mathcal{C}$ w pewnym sensie odtwarza, podobnie jak $\mathcal{W}$, najogólniejszy typ skonstruowanego termu.

Corollary 2.35. *If β is a type variable not occurring in E, $\beta \notin V$, and $\mathcal{C}(E, \beta, V) = (T, e, V_1)$, then for any type τ and substitution T', $\beta \notin \boldsymbol{F}(\tau) \cup \boldsymbol{F}(T')$, for which $T'E \vdash e : \tau$, we have $T' \preccurlyeq T$ outside $V \cup \{\beta\}$ i $\tau \preccurlyeq T\beta$.*

Proof. Załóżmy, że dla wyborów $\vec{w}_1$ i $\vec{w}_2$ zachodzi $\mathcal{C}(E, \beta, V, \vec{w}_1) = (R, e, V_1)$ oraz $\mathcal{C}(E, \beta, V, \vec{w}_2) = (S, e, V_2)$. Sprawdzając, że przebiegi algorytmu się dokładnie pokrywają (przez indukcję względem e), czyli że $\mathcal{C}(E, \beta, V, \vec{w}_1) = \mathcal{C}(E, \beta, V, \vec{w}_2)$, otrzymujemy, że $R = S$.

Rozszerzmy T' do $T'' = [\tau/\beta]T'$. Mamy $T''E \vdash e : T''\beta$. Ze stwierdzenia 2.34 dla pewnej ścieżki wyborów $\vec{w}_1$ i $\mathcal{C}(E, \beta, V, \vec{w}_1) = (R, e, V_1)$ mamy $T'' \preccurlyeq R$ poza V, ale $R = T$, więc $T'' \preccurlyeq T$ poza V, a stąd $T' \preccurlyeq T$ poza $V \cup \{\beta\}$ i $\tau = T''\beta \preccurlyeq T\beta$. □

2.2.2 Extending the language with the construct case.

Dotychczasowy system typów nie traktował w żaden sposób struktur danych. Uściślimy go pod tym względem i wprowadzimy do języka manipulację tzw. indukcyjnymi strukturami danych: konstruktory oraz dekonstruktor analizy przez przypadki case. Zaprezentowany system typów odpowiada systemowi $\lambda^{\hat{}}$ z [2] z pominięciem etapów (stages), które wprowadzimy w następnym rozdziale. W kwestii szczegółów definicji odsyłam do [2].

Definition 2.36. *Typing rules for the inductive structures:*

1. *VAR: Variables...*
2. *FIX: Functions...*
3. *APP: Applications...*

4. *LET: Local bindings...*
5. *CASE: Deconstruction*

$$\frac{E\vdash e'\colon d\vec{\tau} \qquad E\vdash e_i\colon \mathrm{Inst}_{c_i}\vec{\tau}\to\theta\,(1\leqslant i\leqslant n)}{E\vdash \mathrm{case}\,e'\,\mathrm{of}\,\{c_1\Rightarrow e_1|...|c_n\Rightarrow e_n\}\colon\theta}\ \textit{if}\ \boldsymbol{C}(d)=\{c_1,...,c_n\}$$

6. *Instead of the rule: CONS: Constructor*

$$\frac{}{E\vdash c\colon \mathrm{Inst}_c\vec{\tau}\to d\vec{\tau}}\ \textit{if}\ c\in\boldsymbol{C}(d)$$

we introduce an assumption on environment E:

$$(\forall d)(\forall c\in\boldsymbol{C}(d))\ \ \textit{let}\ c\colon\forall\vec{\alpha}.\mathrm{Inst}_c\vec{\alpha}\to d\vec{\alpha}\in E,\ \textit{where}\ \#\vec{\alpha}\ \textit{is the arity of}\ d$$

where $\boldsymbol{C}(d)$ *is a set of constructors of the type* d, $\vec{\tau}$ *are parameters of the type,* $\mathrm{Inst}_c\vec{\tau}$ *is a vector of argument types of constructor* c, *when the parametric type* d *takes parameters* $\vec{\tau}$. *Notation: if* $\vec{\sigma}=(\sigma_1,...,\sigma_n)=(\sigma_i)_{1\leqslant i\leqslant n}$, *then* $\#\vec{\sigma}=n$ *and*

$$\vec{\sigma}\to\tau=\sigma_1\to...\to\sigma_n\to\tau=\sigma_1\to(...\to(\sigma_n\to\tau)...)$$

Wypada rozszerzyć algorytmy $\mathcal{W}$ i $\mathcal{C}$.

Definition 2.37. $\mathcal{W}(E,e,V)=(T,\tau,V')$, *where match* e *with:*

1. $e=x$ *(VAR: Variables)...*
2. $e=\mathrm{fix}\,fx.e_1$ *(FIX: Functions)...*
3. $e=fg$ *(APP: Applications)...*
4. $e=\mathrm{let}\,x=f\,\mathrm{in}\,g$ *(LET: Local bindings)...*
5. $e=\mathrm{case}\,e'\,\mathrm{of}\,\{c_1\Rightarrow e_1|...|c_n\Rightarrow e_n\}$ *(CASE: Deconstruction) For type* d *with arity* k *(*k*-parametric) such that* $c_1,...,c_n\in\boldsymbol{C}(d)$

$$\begin{aligned}
V &= \{\beta_i|1\leqslant i\leqslant k\}\cup\{\theta\}\dot{\cup}V_r\\
(T_0,\rho,V_0) &= \mathcal{W}(E,e',V_r)\\
U_0 &= \boldsymbol{U}(\rho,d\vec{\beta})\\
(T_1,\sigma_1,V_1) &= \mathcal{W}(U_0T_0E,e_1,V_0)\\
U_1 &= \boldsymbol{U}(T_1U_0T_0(\mathrm{Inst}_{c_1}\vec{\beta}\to\theta),\sigma_1)\\
&...\\
(T_n,\sigma_n,V_n) &= \mathcal{W}(U_{n-1}T_{n-1}...U_0T_0E,e_n,V_{n-1})\\
U_n &= \boldsymbol{U}(T_nU_{n-1}T_{n-1}...U_0T_0(\mathrm{Inst}_{c_1}\vec{\beta}\to\theta),\sigma_n)\\
T &= U_nT_n...U_0T_0\\
\tau &= T\theta\\
V' &= V_n
\end{aligned}$$

Note 2.38. Dla dowolnego podstawienia T, typu indukcyjnego d, typów $\vec{\tau}$, $\#\vec{\tau}=\mathrm{ar}(d)$ i konstruktora $c, c\in\boldsymbol{C}(d)$ zachodzi

$$\begin{aligned}
d(T\vec{\tau}) &= Td\vec{\tau}\\
\mathrm{Inst}_cT\vec{\tau} &= T\,\mathrm{Inst}_c\vec{\tau}
\end{aligned}$$

Definition 2.39. $\mathcal{C}(E,\tau,V,\vec{w})=(T,e,V',\vec{w}')$, *where for* $(w_1 \bmod 5)$ *equal*

1. *VAR: Variables...*
2. *FIX: Functions...*
3. *APP: Applications...*
4. *LET: Local bindings...*

5. *CASE: Deconstruction. Let d be $(w_{i+1} \bmod D)$-th inductive type, where D is the number of inductive types, and k is the arity of type d. Let*

$$\begin{aligned}
V &= \{\beta_i | 1 \leqslant i \leqslant k\} \cup \{\theta\} \dot{\cup} V_r \\
(R, e', V_0, \vec{w}_0) &= \mathcal{C}(E, d\vec{\beta}, V_r, (w_{i+2})) \\
(T_1, e_1, V_1, \vec{w}_1) &= \mathcal{C}(R\,E, R\,(\mathrm{Inst}_{c_1}\vec{\beta} \to \tau), V_0, \vec{w}_0) \\
&\ldots \\
(T_n, e_n, V_n, \vec{w}_n) &= \mathcal{C}(T_{n-1}...T_1 R\,E, T_{n-1}...T_1 R\,(\mathrm{Inst}_{c_n}\vec{\beta} \to \tau), V_{n-1}, \vec{w}_{n-1}) \\
T &= T_n...T_1 R \\
e &= \mathrm{case}\, e' \,\mathrm{of}\, \{c_1 \Rightarrow e_1 | ... | c_n \Rightarrow e_n\} \\
V' &= V_n \\
\vec{w}\,' &= \vec{w}_n
\end{aligned}$$

2.2.2.1 Soundness with case.

Theorem 2.40. *Let e be an expression, E an environment, V a set of type variables. If $(T, \tau, V') = \mathcal{W}(E, e, V)$ is defined, then we can derive $TE \vdash e : \tau$.*

Proof. Przez indukcję względem budowy termu:

1. $e = x$ (VAR: Variables)...
2. $e = \mathrm{fix}\, fx.e_1$ (FIX: Functions)...
3. $e = f\,g$ (APP: Applications)...
4. $e = \mathrm{let}\, x = f \,\mathrm{in}\, g$ (LET: Local bindings)...
5. e = case e' of $\{c_1 \Rightarrow e_1 | ... | c_n \Rightarrow e_n\}$ (CASE: Deconstruction). Z algorytmu i z zał. ind. mamy:

$$\begin{aligned}
T_0 E &\vdash e' : \rho \\
U_0 &= \boldsymbol{U}(\rho, d\vec{\beta}) \\
T_1 U_0 T_0 E &\vdash e_1 : \sigma_1 \\
U_1 &= \boldsymbol{U}(T_1 U_0 T_0(\mathrm{Inst}_{c_1}\vec{\beta} \to \theta), \sigma_1) \\
&\ldots \\
T_n U_{n-1} T_{n-1} ... U_0 T_0 E &\vdash e_n : \sigma_n \\
U_n &= \boldsymbol{U}(T_n U_{n-1} T_{n-1} ... U_0 T_0(\mathrm{Inst}_{c_n}\vec{\beta} \to \theta), \sigma_n) \\
T &= U_n T_n ... U_0 T_0 \\
\tau &= T\theta
\end{aligned}$$

Ponieważ

$$U_0 \rho = U_0 d\vec{\beta}$$

oraz

$$U_i \sigma_i = U_i(\mathrm{Inst}_{c_i}\vec{\beta} \to \theta)$$

z lematu o podstawianiu mamy (podstawiając po obu stronach U_i, stosując powyższą równość, podstawiając po obu stronach $T_{i+1}, U_{i+1}, ..., T_n, U_n$)

$$\begin{aligned}
TE &\vdash e' : U_n T_n ... U_0 d\bar{\beta} \\
TE &\vdash e_1 : T(\mathrm{Inst}_{c_1}\vec{\beta} \to \theta) \\
&\ldots \\
TE &\vdash e_n : T(\mathrm{Inst}_{c_n}\vec{\beta} \to \theta)
\end{aligned}$$

Stosując fakt 2.38 mamy

$$\begin{aligned}
TE &\vdash e' : d(T\vec{\beta}) \\
TE &\vdash e_1 : \mathrm{Inst}_{c_1}(T\vec{\beta}) \to \tau \\
&\ldots \\
TE &\vdash e_n : \mathrm{Inst}_{c_n}(T\vec{\beta}) \to \tau
\end{aligned}$$

Z reguły dekonstrukcji (CASE) wnioskujemy (dla tamtejszych $\vec{\tau} := T\vec{\beta}$ i $\theta := \tau$)

$$TE \vdash (\text{case } e' \text{ of } \{c_1 \Rightarrow e_1 | ... | c_n \Rightarrow e_n\}) : \tau$$

□

Theorem 2.41. *Let τ be a type, E an environment, V a set of type variables. If $(T, e, V') = \mathcal{C}(E, \tau, V)$ is defined, we can derive $TE \vdash e : T\tau$.*

Proof. Przez indukcję na złożoność generowanego termu:

1. $e = x$ (VAR: Variables)...
2. $e = \text{fix } fx.e_1$ (FIX: Functions)...
3. $e = fg$ (APP: Applications)...
4. $e = \text{let } x = f \text{ in } g$ (LET: Local bindings)...
5. $e = \text{case } e' \text{ of } \{c_1 \Rightarrow e_1 | ... | c_n \Rightarrow e_n\}$ (CASE: Deconstruction). Z algorytmu i zał. ind. mamy:

$$\begin{aligned} \rho &= d\vec{\beta} \\ RE &\vdash e' : R\rho \\ T_1 R E &\vdash e_1 : T_1 R(\text{Inst}_{c_1}\vec{\beta} \to \tau) \\ &... \\ T_n ... T_1 R E &\vdash e_n : T_n ... T_1 R\,(\text{Inst}_{c_n}\vec{\beta} \to \tau) \\ T &= T_n ... T_1 R \end{aligned}$$

Stosując lemat o podstawianiu i fakt 2.38, jak w dowodzie dla algorytmu $\mathcal{W}$, otrzymujemy

$$\begin{aligned} TE &\vdash e' : d(T\vec{\beta}) \\ TE &\vdash e_1 : \text{Inst}_{c_1}(T\vec{\beta}) \to T\tau \\ &... \\ TE &\vdash e_n : \text{Inst}_{c_n}(T\vec{\beta}) \to T\tau \end{aligned}$$

Z reguły dekonstrukcji (CASE) wnioskujemy (dla tamtejszych $\vec{\tau} := T\vec{\beta}$ i $\theta := T\tau$)

$$TE \vdash (\text{case } e' \text{ of } \{c_1 \Rightarrow e_1 | ... | c_n \Rightarrow e_n\}) : T\tau$$

□

2.2.2.2 Completeness with case.

Theorem 2.42. *Let e be an expression, E an environment, V an infinite set of type variables such that $V \cap \boldsymbol{F}(E) = \varnothing$. If there exists a type τ' and a substitution T' such that $T'E \vdash e : \tau'$, then $(T, \tau, V') = \mathcal{W}(E, e, V)$ is defined, and there exists a substituion P such that*

$$\tau' = P\tau \qquad i \qquad T' = PT \text{ outside } V$$

Proof. Przez indukcję względem złożoności e:

1. $e = x$ (VAR: Variables)...
2. $e = \text{fix } fx.e_1$ (FIX: Functions)...
3. $e = fg$ (APP: Applications)...
4. $e = \text{let } x = f \text{ in } g$ (LET: Local bindings)...
5. $e = \text{case } e' \text{ of } \{c_1 \Rightarrow e_1 | ... | c_n \Rightarrow e_n\}$ (CASE: Deconstruction). Wyprowadzenie typu τ' kończy się z

$$\frac{T'E \vdash e' : d\vec{\gamma} \qquad T'E \vdash e_i : \text{Inst}_{c_i}\vec{\gamma} \to \tau' \; (1 \leqslant i \leqslant n)}{T'E \vdash \text{case } e' \text{ of } \{c_1 \Rightarrow e_1 | ... | c_n \Rightarrow e_n\} : \tau'}$$

Stosujemy zał. ind. $n+1$-krotnie

$$\begin{aligned}
(T_0,\rho,V_0) &= \mathcal{W}(E,e',V_r)\\
d\vec{\gamma} &= P_0\rho\\
T' &= P_0T_0 \text{ poza } V_r\\
U_0 &= \boldsymbol{U}(\rho,d\vec{\beta})\\
P_0 &= P_0'U_0 \text{ poza } \vec{\beta}\\
T'E &= P_0T_0E = P_0'U_0T_0E\\
(T_1,\sigma_1,V_1) &= \mathcal{W}(U_0T_0\,E,e_1,V_0)\\
\mathrm{Inst}_{c_1}\vec{\gamma}\to\tau' &= P_1\sigma_1\\
P_0' &= P_1T_1 \text{ poza } V_0\\
U_1 &= \boldsymbol{U}(T_1U_0T_0(\mathrm{Inst}_{c_1}\vec{\beta}\to\theta),\sigma_1)\\
P_1 &= P_1'U_1 \text{ poza } \theta\\
T'E &= P_1'U_1T_1U_0T_0E\\
(T_2,\sigma_2,V_2) &= \mathcal{W}(U_1T_1U_0T_0E,e_2,V_1)\\
\mathrm{Inst}_{c_2}\vec{\gamma}\to\tau' &= P_2\sigma_2\\
P_1' &= P_2T_2 \text{ poza } V_1\\
U_2 &= \boldsymbol{U}(T_2U_1T_1U_0T_0(\mathrm{Inst}_{c_2}\vec{\beta}\to\theta),\sigma_2)\\
P_2 &= P_2'U_2 \text{ poza } \theta\\
&\ldots\\
(T_n,\sigma_n,V_n) &= \mathcal{W}(U_{n-1}T_{n-1}...U_0T_0E,e_n,V_{n-1})\\
\mathrm{Inst}_{c_n}\vec{\gamma}\to\tau' &= P_n\sigma_n\\
P_{n-1}' &= P_nT_n \text{ poza } V_{n-1}\\
U_n &= \boldsymbol{U}(T_nU_{n-1}T_{n-1}...U_0T_0(\mathrm{Inst}_{c_n}\vec{\beta}\to\theta),\sigma_n)\\
P_n &= P_n'U_n \text{ poza } \theta\\
T &= U_nT_n...U_0T_0\\
T' &= P_n'T \text{ poza } V\\
\tau &= T\theta
\end{aligned}$$

Unifikator U_0 istnieje, bo $P_0\rho = d\vec{\gamma} = [\vec{\gamma}/\vec{\beta}]d\vec{\beta}$, gdzie $[\vec{\gamma}/\vec{\beta}] = [\gamma_1/\beta_1;...;\gamma_{\mathrm{ar}(d)}/\beta_{\mathrm{ar}(d)}]$. $P_0 = P_0'U_0$ poza $\vec{\beta}$ dla pewnego P_0', bo $[\vec{\gamma}/\vec{\beta}]P_0$ jest innym unifikatorem; (jeśli U_0 podstawia pod ρ, które jest wtedy zmienną, to P_0' podstawia pod β_i; jeśli U_0 podstawia tylko pod β_i, to $P_0' = P_0$; jeśli $U_0\alpha = \beta_i$, to $P_0'\beta_i = P_0\alpha$). Istnieje unifikator U_1, bo $P_1\sigma_1 = \mathrm{Inst}_{c_1}\vec{\gamma}\to\tau'$, a $P_1T_1U_0T_0(\mathrm{Inst}_{c_1}\vec{\beta}\to\theta) = P_0'U_0T_0(\mathrm{Inst}_{c_1}\vec{\beta}\to\theta) = \mathrm{Inst}_{c_1}\vec{\gamma}\to\theta$, bo $d(P_0'U_0\vec{\beta}) = P_0'(U_0d\vec{\beta}) = P_0'U_0\rho = P_0\rho = d\vec{\gamma}$. Stąd też $P_1 = P_1'U_1$ poza <the4a>, bo $[\tau'/\theta]P_1$ jest unifikatorem. Niech σ_1^θ oznacza fragment typu $U_1\sigma_1$ odpowiadający $U_1\theta$. Istnieje unifikator U_2, bo $P_2\sigma_2 = \mathrm{Inst}_{c_2}\vec{\gamma}\to\tau'$, a

$$\begin{aligned}
P_2T_2U_1T_1U_0T_0(\mathrm{Inst}_{c_2}\vec{\beta}\to\theta) &= P_1'U_1T_1U_0T_0(\mathrm{Inst}_{c_2}\vec{\beta}\to\theta)\\
&= P_1'U_1T_1U_0T_0\mathrm{Inst}_{c_2}\vec{\beta}\to P_1'U_1T_1U_0T_0\theta\\
&= P_1T_1U_0T_0\mathrm{Inst}_{c_2}\vec{\beta}\to P_1'(U_1\theta)\\
&= P_0'U_0T_0\mathrm{Inst}_{c_2}\vec{\beta}\to P_1'\sigma_1^\theta\\
&= \mathrm{Inst}_{c_2}\vec{\gamma}\to(\tau' \text{ albo } \theta)
\end{aligned}$$

Także $P_2 = P_2'U_2$ poza θ, bo jeśli nie P_2, to napewno $[\tau'/\theta]P_2$ jest unifikatorem. Podobnie istnieją unifikatory U_i dla $i = 3, ..., n$. Mamy $P_n'T = P_n'U_nT_n...U_0T_0 = P_nT_n...U_0T_0 = P_{n-1}'U_{n-1}T_{4-1}...U_0T_0 = P_0'U_0T_0 = P_0T_0 = T'$ poza V, bo $\vec{\beta}$, $\{\theta\}$, V_r, $V_i \subset V$. Także $P_n'U_nT_n...U_0T_0\theta = \tau'$ jeśli któreś spośród U_i podstawia pod θ, w przec. przyp. $P_n'T = \theta$. Określmy $P = [\tau'/\theta]P_n'$. P spełnia warunki stwierdzenia.

□

Theorem 2.43. *Let e be an expression, E an environment. For any type τ and substitution T' such that $T'E \vdash e : T'\tau$ and $\boldsymbol{F}(T') \cap \mathrm{Dom}(T') = \varnothing$ (?), for an infinite set of variables V disjoint with $\boldsymbol{F}(E)$ and $\boldsymbol{F}(T')$, for some path of choices the following holds: $(T, e, V') = \mathcal{C}(E, \tau, V)$ and $T' \preccurlyeq T$ outside V.*

Proof. Przez indukcję względem złożoności e:

1. $e = x$ (VAR: Variables)...
2. $e = \mathrm{fix}\, fx.e_1$ (FIX: Functions)...
3. $e = f\,g$ (APP: Applications)...
4. $e = \mathrm{let}\, x = f \,\mathrm{in}\, g$ (LET: Local bindings)...
5. $e = \mathrm{case}\, e' \,\mathrm{of}\, \{c_1 \Rightarrow e_1 | ... | c_n \Rightarrow e_n\}$ (CASE: Deconstruction). Wyprowadzenie $T'\tau$ kończy się z
$$\frac{T'E \vdash e' : d\vec{\gamma} \qquad T'E \vdash e_i : \mathrm{Inst}_{c_i}\vec{\gamma} \to \tau' \; (1 \leqslant i \leqslant n)}{T'E \vdash \mathrm{case}\, e' \,\mathrm{of}\, \{c_1 \Rightarrow e_1 | ... | c_n \Rightarrow e_n\} : \tau'}$$
gdzie $\tau' = T'\tau$. Wybieramy typ indukcyjny d. Stosując $n+1$-krotnie zał. ind. mamy
$$\begin{aligned}
(R, e', V_0, \vec{w}_0) &= \mathcal{C}(E, d\vec{\beta}, V_r, (w_{i+2})) \\
[\vec{\gamma}/\vec{\beta}]T' &\preccurlyeq R \text{ poza } V_r \\
[\vec{\gamma}/\vec{\beta}]T' &= P_0 R \text{ poza } V_r \\
(T_1, e_1, V_1, \vec{w}_1) &= \mathcal{C}(R\,E, R\,(\mathrm{Inst}_{c_1}\vec{\beta} \to \tau), V_0, \vec{w}_0) \\
P_0 &\preccurlyeq T_1 \text{ poza } V_0 \\
P_0 &= P_1 T_1 \text{ poza } V_0 \\
(T_2, e_2, V_2, \vec{w}_2) &= \mathcal{C}(T_1 R\,E, T_1 R(\mathrm{Inst}_{c_2}\vec{\beta} \to \tau), V_1, \vec{w}_1) \\
P_1 &\preccurlyeq T_2 \text{ poza } V_1 \\
P_1 &= P_2 T_2 \text{ poza } V_1 \\
&... \\
(T_n, e_n, V_n, \vec{w}_n) &= \mathcal{C}(T_{n-1}...T_1 R\,E, T_{n-1}...T_1 R\,(\mathrm{Inst}_{c_n}\vec{\beta} \to \tau), V_{n-1}, \vec{w}_{n-1}) \\
P_{n-1} &\preccurlyeq T_n \text{ poza } V_{n-1} \\
P_{n-1} &= P_n T_n \text{ poza } V_{n-1} \\
T &= T_n...T_1 R \\
T' &= P_n T \text{ poza } V
\end{aligned}$$
$[\vec{\gamma}/\vec{\beta}]T'd\vec{\beta} = [\vec{\gamma}/\vec{\beta}]d\vec{\beta} = d\vec{\gamma}$. $P_nT = P_nT_n...T_1R = P_{n-1}T_{n-1}...T_1R = ... = P_0R = [\vec{\gamma}/\vec{\beta}]T'$ poza $V_r \cup V_0 \cup ... \cup V_{n-1} \subset V$, oraz $\vec{\beta} \subset V$, czyli $P_nT = T'$ poza V.

□

2.2.2.3 Practical issues: CASE driven by use.

Aby zmniejszyć przypadkowość użyć reguły CASE, wprowadzamy do środowiska specjalne zmienne $x_{c_i,j}$: $\forall\vec{\beta}.(\mathrm{Inst}_{c_i}\vec{\beta})_j$ oznaczające wyjęcie j-tego argumentu konstruktora c_i. Dla każdego wyboru zmiennej generycznej $x_{c_i,j}$ generowane i wprowadzane do środowiska są nowe zmienne $x^k_{c_i,j}$: $(\mathrm{Inst}_{c_i}\vec{\beta})_j$ dla $j = 1, ..., \mathrm{ar}(c_i)$, odpowiadające argumentom konstruktora c_i z odpowiedniej dekonstrukcji CASE (gdzie $\vec{\beta}$ są świeżymi zmiennymi z V); użyta zostaje zmienna $x^k_{c_i,j}$. W fazie postprocessingu konstrukcje CASE są zastosowane w najwęższych podtermach obejmujących $x^k_{c_i,j}$ dla ustalonych i, k, tak, żeby żadne $x^k_{c_i,j}$ nie było głową "swojego" podtermu. Stworzony podterm nazywamy gałęzią główną odpowiedniego zastosowania CASE; rekurencyjnie generowane są pozostałe gałęzie CASE. W jednoprzebiegowym algorytmie należy przyjąć jakąś arbitralną strategię limitowania rozmiaru gałęzi.

2.3 Monotonicity and termination.

Teraz zaprezentuję system kontrolujący głębokość rekurencji, zapewniający w ten sposób własność stopu. System $\lambda^\wedge$ został przedstawiony w [2]. Typy danych w $\lambda^\wedge$ są indeksowane informacją o głębokości struktur, tzw. etapami: $d^s\vec{\tau}$, d jest typem danych, ^{s} jest etapem, $\vec{\tau}$ są parametrami typu. Reguła wprowadzająca rekurencję zapewnia, że argument funkcji w wywołaniach rekurencyjnych będzie malał, a ponieważ rozmiary danych są skończone, więc program rekurencyjny w końcu się zatrzyma.

Etapy konstruuje się ze zmiennych etapu, unarnego operatora $\hat{}$ oznaczającego jednostkowe zwiększenie rozmiaru, oraz etapu pochłaniającego ∞, oznaczającego dowolny rozmiar. $\hat{\iota}$ oznacza wszystkie rozmiary nie większe więcej niż o jeden od rozmiarów oznaczanych przez ι. $d^s\vec{\tau}$ oznacza wszystkie dane "typu $d\vec{\tau}$" o rozmiarze należącym do s (alternatywnie można mówić: nie większym niż s).

2.3.1 System typów i algorytm $\mathcal{C}$.

Tak wygląda prezentacja w [2]:

Definition 2.44. *Porównywanie etapów $\preccurlyeq$ i podtypowanie $\sqsubseteq$. Reguły porównywania etapów:*

1. *(REFL)*
$$\frac{}{s \preccurlyeq s}$$
2. *(TRANS)*
$$\frac{s \preccurlyeq r \qquad r \preccurlyeq p}{s \preccurlyeq p}$$
3. *(HAT)*
$$\frac{}{s \preccurlyeq \hat{s}}$$
4. *(INFTY)*
$$\frac{}{s \preccurlyeq \infty}$$

Reguły podtypowania:

1. *(REFL)*
$$\frac{}{s \sqsubseteq s}$$
2. *(DATA)*
$$\frac{s \preccurlyeq r \qquad \tau_i \sqsubseteq \tau_i' \qquad (1 \leqslant i \leqslant \mathrm{ar}(d))}{d^s\vec{\tau} \sqsubseteq d^r\vec{\tau}'}$$
3. *(FUNC)*
$$\frac{\tau' \sqsubseteq \tau \qquad \sigma \sqsubseteq \sigma'}{\tau \to \sigma \sqsubseteq \tau' \to \sigma'}$$

Definition 2.45. *Reguły typowania.*

1. *(VAR)*
$$\frac{}{E \vdash x : \sigma}$$
gdy $(x:\sigma) \in \Gamma$.
2. *(ABS)*
$$\frac{E.x:\tau \vdash e : \sigma}{E \vdash \lambda x.e : \tau \to \sigma}$$
3. *(APP)*
$$\frac{E \vdash e : \tau \to \sigma \qquad E \vdash e' : \tau}{E \vdash e\,e' : \sigma}$$
4. *(CONS)*
$$\frac{}{E \vdash \mathrm{Inst}_c^{s\vec{\tau}} \to d^{\hat{s}}\vec{\tau}}$$

gdy $c \in \boldsymbol{C}(d)$.

5. *(CASE)*

$$\frac{E \vdash e' : d^{\hat{s}}\vec{\tau} \qquad E \vdash e_i : \mathrm{Inst}^{s}_{c_i}\vec{\tau} \to \theta \qquad (1 \leqslant i \leqslant n)}{E \vdash \mathrm{case}\, e' \,\mathrm{of}\, \{c_1 \Rightarrow e_1 | ... | c_n \Rightarrow e_n\} : \theta}$$

gdy $\boldsymbol{C}(d) = \{c_1, ..., c_n\}$.

6. *(REC)*

$$\frac{E.f : d^{\iota}\vec{\tau} \to \theta \vdash e : d^{\hat{\iota}}\vec{\tau} \to [\hat{\iota}/\iota]\theta \qquad \iota \,\mathrm{pos}\, \theta}{E \vdash (\mathrm{letrec}\ f = e) : d^{s}\vec{\tau} \to [s/\iota]\theta}$$

gdy ι *nie występuje w* $E, \vec{\tau}$.

7. *(SUB)*

$$\frac{E \vdash e : \sigma \qquad \sigma \sqsubseteq \sigma'}{E \vdash e : \sigma'}$$

Pozytywne-negatywne wystąpienia zmiennej etapu:

1. *(SP1)*

$$\frac{}{\iota \,\mathrm{pos}\, \alpha}$$

2. *(SP2)*

$$\frac{\iota \,\mathrm{neg}\, \tau \qquad \iota \,\mathrm{pos}\, \sigma}{\iota \,\mathrm{pos}\, \tau \to \sigma}$$

3. *(SP3)*

$$\frac{\iota \,\mathrm{pos}\, \tau_i \qquad (1 \leqslant i \leqslant \mathrm{ar}(d))}{\iota \,\mathrm{pos}\, d^{s}\vec{\tau}}$$

4. *(SN1)*

$$\frac{}{\iota \,\mathrm{neg}\, \alpha}$$

5. *(SN2)*

$$\frac{\iota \,\mathrm{pos}\, \tau \qquad \iota \,\mathrm{neg}\, \sigma}{\iota \,\mathrm{neg}\, \tau \to \sigma}$$

6. *(SN3)*

$$\frac{\iota \,\mathrm{nocc}\, s \qquad \iota \,\mathrm{neg}\, \tau_i \qquad (1 \leqslant i \leqslant \mathrm{ar}(d))}{\iota \,\mathrm{neg}\, d^{s}\vec{\tau}}$$

Jednak my potrzebujemy w regułach typowania wskazówek co do podstawień, dlatego będziemy jawnie obsługiwać polimorfizm, wprowadzając explicite kwantyfikację zmiennych etapu. Ponieważ rekurencja jest dopuszczalna tylko względem argumentu będącego typem danych, musimy przywrócić podział na abstrakcję i rekurencję, jednak dostosujemy system do tego z pierwszego rozdziału, zastępując letrec $f = e$ (REC) przez konstrukcję fix $fx.e$ (FIX). Wprowadzimy też definicje lokalne (LET). Zamiast trudnej do kontroli reguły (SUB), wprowadzamy podtypowanie obok konkretyzacji typu w regule (VAR). Co istotniejsze, wprowadzimy stałe etapu (albo zmienne uniwersalne etapu), ponieważ zmienne etapu (albo zmienne egzystencjalne, zmienne unifikacji) w typie zadanym zostałyby potraktowane jako niewiadome. Podobnie, wprowadzimy stałe typu (albo zmienne uniwersalne typu), w odróżnieniu od konstruktorów typów danych d i od zmiennych typu (egzystencjalnych, zmiennych unifikacji). (Egzystencjalne) zmienne wolne typu lub etapu nazywamy niewiadomymi typu lub etapu, odpowiednio.

Definition 2.46. *Generalizacja*

$$\boldsymbol{G}(\tau, E) \;=\; \forall \alpha_1, ..., \alpha_n.\tau$$

gdzie $\alpha_1, ...\alpha_n$ *są dokładnie tymi zmiennymi wolnymi typu lub etapu (niewiadomymi) w* τ*, które nie występują w* E.

Definition 2.47. *Konkretyzacja typu zmiennej ze środowiska:* τ *jest konkretyzacją typu zmiennej* x *ze środowiska* E, α_i *są zmiennymi typu, a* ι_i *są zmiennymi etapu,*

$$\tau \leqslant E(x) \;\equiv\; E = ...; x : \forall \alpha_1 ... \alpha_n \iota_1 ... \iota_m.\sigma; ... \;\; i \;\; \tau = [\tau_1/\alpha_1; ...; \tau_n/\alpha_n; s_1/\iota_1; ...; s_m/\iota_m]\sigma$$

dla pewnych typów $\tau_1, ..., \tau_n$ *oraz etapów* $s_1, ..., s_m$.

Poniżej, podtypowanie jest tak jak określono powyżej.

Definition 2.48. *Dostosowane reguły typowania dla* $\lambda^\wedge$.

1. *VAR: Variables.*
$$\frac{\tau \leqslant E(x) \qquad \sigma \sqsubseteq \tau}{E \vdash x : \sigma}$$
2. *ABS: Abstractions.*
$$\frac{E.x : \tau \vdash e : \sigma}{E \vdash \lambda x.e : \tau \to \sigma}$$
3. *FIX: Recursive functions.*
$$\frac{E.f : d^\iota \vec{\tau} \to \theta.x : d^{\hat{\iota}} \vec{\tau} \vdash e : [\hat{\iota}/\iota]\theta \qquad \iota \operatorname{pos} \theta}{E \vdash (\operatorname{fix} fx = e) : d^s \vec{\tau} \to [s/\iota]\theta}$$
gdzie ι *nie występuje w* E *ani w* $\vec{\tau}$.
4. *APP: Applications.*
$$\frac{E \vdash e : \sigma \to \tau \qquad E \vdash e' : \sigma}{E \vdash e e' : \tau}$$
5. *LET: Local bindings.*
$$\frac{E \vdash e' : \sigma \qquad E.x : \boldsymbol{G}(\sigma, E) \vdash e : \tau}{E \vdash (\operatorname{let} x = e' \operatorname{in} e) : \tau}$$
6. *CASE: Deconstruction*
$$\frac{E \vdash e' : d^{\hat{s}} \vec{\tau} \qquad E \vdash e_i : \operatorname{Inst}^s_{c_i} \vec{\tau} \to \theta \, (1 \leqslant i \leqslant n)}{E \vdash \operatorname{case} e' \operatorname{of} \{c_1 \Rightarrow e_1 | ... | c_n \Rightarrow e_n\} : \theta} \text{ jeśli } \boldsymbol{C}(d) = \{c_1, ..., c_n\}$$
7. *Zamiast reguły: CONS: Constructor*
$$\frac{}{E \vdash c : \operatorname{Inst}^s_c \vec{\tau} \to d^{\hat{s}} \vec{\tau}} \text{ jeśli } c \in \boldsymbol{C}(d)$$
wprowadzamy założenie na temat środowiska E:
$$\forall d \forall c \in \boldsymbol{C}(d) \text{ niech } c : \forall \alpha_1 ... \alpha_n \iota . \operatorname{Inst}^\iota_c \vec{\alpha} \to d^{\hat{\iota}} \vec{\alpha} \in E, \text{ gdzie } n \text{ jest arnością } d$$

Aby opracować algorytm generowania termów $\mathcal{C}$ dla $\lambda^\wedge$, potrzebujemy mechanizmu szukającego możliwości zajścia relacji podtypowania, tak jak unifikacja jest ü mechanizmem szukania konkretyzacji. Odpowiedni algorytm nazwiemy $\boldsymbol{U}_\sqsubseteq$. Zachodzi $U = \boldsymbol{U}_\sqsubseteq(\tau, \sigma)$, jeśli $U\tau \sqsubseteq U\sigma$, przy czym U jest podstawieniem zarówno pod zmienne typu, jak i pod zmienne etapu.

Definition 2.49. $\mathcal{C}(E, \tau, V, \vec{w}) = (T, e, V', \vec{w}')$, *gdzie dla* $(w_1 \bmod 4)$ *równego*

1. *VAR: Variables. Dla* $E(w_2 \bmod \bar{E}) = x : \forall \alpha_1 ... \alpha_n . \sigma$ *niech*
$$\begin{aligned} U &= \boldsymbol{U}_\sqsubseteq(\tau, [\beta_1/\alpha_1]...[\beta_n/\alpha_n]\sigma) \\ V &= \{\beta_i | 1 \leqslant i \leqslant n\} \dot\cup V' \\ e &= x \\ T &= U \\ w'_i &= w_{i+2} \end{aligned}$$
Jeśli unifikator nie istnieje, $\mathcal{C}$ *jest nieokreślone.*
2. *ABS: Abstractions. Jeśli* $\tau = \sigma \to \rho$, *niech*
$$\begin{aligned} (R, e_1, V', \vec{w}_1) &= \mathcal{C}(E.x : \sigma, \rho, V, (w_{i+1})) \\ T &= R \\ e &= \lambda x.e_1 \\ \vec{w}' &= \vec{w}_1 \end{aligned}$$

gdzie f, x są nowymi zmiennymi. Jeśli $\tau = \alpha$, gdzie α jest zmienną typu, dla $\beta_1, \beta \in V$ niech

$$\begin{aligned} R &= [\beta_1 \to \beta/\alpha] \\ (S, e_1, V', \vec{w}_1) &= \mathcal{C}(R\,E.x\colon \beta_1, \beta, V \setminus \{\beta_1, \beta\}, (w_{i+1})) \\ T &= SR \\ e &= \lambda x.e_1 \end{aligned}$$

gdzie f, x są nowymi zmiennymi. Jeśli τ ma inną postać, $\mathcal{C}$ jest nieokreślone.

3. *FIX: Functions. Jeśli $\tau = d^s\vec{\sigma} \to \theta$, niech κ dowolne takie, że $[s/\iota]\kappa = \theta$, oraz*

$$\begin{aligned} (R, e_1, V', \vec{w}_1) &= \mathcal{C}(E.f\colon d^\iota\vec{\sigma} \to \kappa.x\colon d^{\hat{\iota}}\vec{\sigma}, [\hat{\iota}/\iota]\kappa, V, (w_{i+1})) \\ T &= R \\ e &= \operatorname{fix} fx.e_1 \\ \vec{w}\,' &= \vec{w}_1 \end{aligned}$$

gdzie f, x są nowymi zmiennymi. Jeśli $\tau = \alpha$, gdzie α jest zmienną typu, dla $\beta_i, \beta', \beta'', \beta''', \iota, \eta \in V$ niech

$$\begin{aligned} R &= [d^\eta\vec{\beta} \to \beta'''/\alpha] \\ (S, e_1, V', \vec{w}_1) &= \mathcal{C}(R\,E.f\colon d^\iota\vec{\beta} \to \beta''.x\colon d^{\hat{\iota}}\vec{\beta}, \beta', V \setminus \{\beta_i, \beta', \beta'', \beta''', \iota, \eta\}, (w_{i+1})) \\ U &= \boldsymbol{U}([\hat{\iota}/\iota]\beta'', \beta') \\ W &= \boldsymbol{U}([\eta/\iota]\beta'', \beta''') \\ T &= WUSR \\ e &= \operatorname{fix} fx.e_1 \end{aligned}$$

gdzie f, x są nowymi zmiennymi. Jeśli τ ma inną postać, $\mathcal{C}$ jest nieokreślone.

4. *APP: Applications. Niech dla $\beta \in V$*

$$\begin{aligned} (R, f, V_1, \vec{w}_1) &= \mathcal{C}(E, \beta \to \tau, V \setminus \{\beta\}, (w_{i+1})) \\ (S, g, V', \vec{w}_2) &= \mathcal{C}(R\,E, R\beta, V_1, \vec{w}_1) \\ T &= SR \\ e &= fg \\ \vec{w}\,' &= \vec{w}_2 \end{aligned}$$

5. *LET: Local bindings. Niech dla $\beta \in V$*

$$\begin{aligned} (R, f, V_1, \vec{w}_1) &= \mathcal{C}(E, \beta, V \setminus \{\beta\}, (w_{i+1})) \\ (S, g, V', \vec{w}_2) &= \mathcal{C}(R\,E.x\colon \boldsymbol{G}(R\beta, R\,E), R\tau, V_1, \vec{w}_1) \\ T &= SR \\ e &= \operatorname{let} x = f \operatorname{in} g \\ \vec{w}\,' &= \vec{w}_2 \end{aligned}$$

6. *CASE: Deconstruction. Niech d będzie $(w_{i+1} \bmod D)$-ym typem indukcyjnym, gdzie D jest ilością typów indukcyjnych, oraz k arnością typu d. Niech*

$$\begin{aligned} V &= \{\beta_i | 1 \leqslant i \leqslant k\} \dot\cup \{\iota, \theta\} \dot\cup V_r \\ (R, e', V_0, \vec{w}_0) &= \mathcal{C}(E, d^{\hat{\iota}}\vec{\beta}, V_r, (w_{i+2})) \\ (T_1, e_1, V_1, \vec{w}_1) &= \mathcal{C}(R\,E, R\,(\operatorname{Inst}^\iota_{c_1}\vec{\beta} \to \tau), V_0, \vec{w}_0) \\ &\dots \\ (T_n, e_n, V_n, \vec{w}_n) &= \mathcal{C}(T_{n-1}...T_1R\,E, T_{n-1}...T_1R\,(\operatorname{Inst}^\iota_{c_n}\vec{\beta} \to \tau), V_{n-1}, \vec{w}_{n-1}) \\ T &= T_n...T_1R \\ e &= \operatorname{case} e' \operatorname{of} \{c_1 \Rightarrow e_1 | ... | c_n \Rightarrow e_n\} \\ V' &= V_n \\ \vec{w}\,' &= \vec{w}_n \end{aligned}$$

Uwagi: zmienna etapu ι, podobnie jak parametry typu $\vec{\beta}$, w regule (CASE) i (FIX), są zmiennymi wolnymi, które mogą zunifikować, "dowiadując się" (np. od typu τ w przypadku (CASE)) jaki etap bądź typ jest potrzebny.

2.3.1.1 Unifikacja z podtypowaniem $U_{\sqsubseteq}$.

Potrzebujemy najogólniejszego unifikatora, w następującym sensie: jeśli $U = \boldsymbol{U}_{\sqsubseteq}(\tau, \sigma)$, to $U\tau \sqsubseteq U\sigma$, oraz jeśli $V\tau \sqsubseteq V\sigma$, to $V\tau \sqsubseteq U\tau$ oraz $U\sigma \sqsubseteq V\sigma$. Odpowiednio, jeśli $U = \boldsymbol{U}_{\mathrm{st}}(r, s)$, to $U\,r \preccurlyeq U\,s$ oraz jeśli $Vr \preccurlyeq Vs$, to $Vr \preccurlyeq Ur$ oraz $Us \preccurlyeq Vs$.

Najpierw porównajmy etapy.

Definition 2.50. $\boldsymbol{U}_{\mathrm{st}}(r,s) =$ *dopasuj* r, s *dobierając pierwszy pasujący przypadek*

1. $\#, \infty$*: zwróć* [] *(reguła INFTY)*
2. $\hat{r}, \hat{s}$*: zwróć* $\boldsymbol{U}_{\mathrm{st}}(r,s)$
3. $r, \hat{s}$*: zwróć* $\boldsymbol{U}_{\mathrm{st}}(r,s)$ *(reguła HAT)*
4. $\iota, \#$*, gdzie* ι *jest zmienną etapu: zwróć* $[s/\iota]$ *(*s *jest zmienną lub stałą etapu)*
5. $\#, \iota$*, gdzie* ι *jest zmienną etapu: jeśli* ι *nie występuje w* r*, zwróć* $[r/\iota]$

Definition 2.51. $\boldsymbol{U}(\tau,\sigma) =$ *dopasuj* τ, σ

1. $\alpha, \#$ *i* α *jest zmienną typu: jeśli* α *nie występuje w* σ*, zwróć* $[\sigma/\alpha]$
2. $\#, \alpha$ *i* α *jest zmienną typu: jeśli* α *nie występuje w* τ*, zwróć* $[\tau/\alpha]$
3. α, α*: zwróć* []
4. $\tau_1 \to \tau_2, \sigma_1 \to \sigma_2$*: niech* $U_1 = \boldsymbol{U}(\tau_2, \sigma_2)$*, zwróć* $\boldsymbol{U}(U_1\sigma_1, U_1\tau_1)$
5. $d^r\vec{\tau}, d^s\vec{\sigma}$*: niech*

$$\begin{aligned} U_0 &= \boldsymbol{U}_{\mathrm{st}}(r,s) \\ U_1 &= \boldsymbol{U}(U_0\tau_1, U_0\sigma_1) \\ &\ldots \\ U_{\#\vec{\tau}} &= U_{\#\vec{\tau}}(U_0 \ldots U_{\#\vec{\tau}-1}\tau_{\#\vec{\tau}}, U_0 \ldots U_{\#\vec{\tau}-1}\sigma_{\#\vec{\tau}}) \end{aligned}$$

2.4 Generowanie: mechanizmy do zastosowania.

Wykonamy krok wstecz od języka ML z poprzedniego paragrafu, pomijając dekonstrukcję (case), definicje rekurencyjne i lokalne, aby do generowania termów wykorzystać doskonałą maszynerię, opracowaną dla języka Prolog. Przy okazji poczynimy krok w kierunku języka HMG(X), wprowadzając typy indukcyjne z więzami unifikacyjnymi.

2.4.1 System typów z typami indukcyjnymi z więzami unifikacyjnymi.

Modyfikacja, dająca system typów z typami indukcyjnymi z więzami unifikacyjnymi, dotyczy reguł CASE i CONS. Jednak, aby móc zastosować mechanizm z korzyścią dla choćby najprostszych funkcji rekurencyjnych, potrzebna jest rekurencja polimorficzna.

Definition 2.52. *Reguły typowania dla typów indukcyjnych z więzami unifikacyjnymi:*

1. *VAR: Variables...*
2. *ABS: λ-abstractions...*
3. *APP: Applications...*
4. *LETREC: Polymorphic Recursion.*

$$\frac{E.x\colon \rho \vdash e'\colon \sigma \qquad \rho = \boldsymbol{G}(\sigma, E) \qquad E.x\colon \rho \vdash e\colon \tau}{E \vdash (\text{letrec } x = e' \text{ in } e)\colon \tau}$$

5. *CASE: Deconstruction*

$$\frac{E \vdash e'\colon d\vec{\tau} \qquad T_i E \vdash e_i\colon T_i \text{Inst}_{c_i}\vec{\alpha}_i \to T_i\theta \; (1 \leqslant i \leqslant n)}{E \vdash \text{case } e' \text{ of } \{c_1 \Rightarrow e_1 | \ldots | c_n \Rightarrow e_n\}\colon \theta} \text{ jeśli } \boldsymbol{C}(d) = \{c_1, \ldots, c_n\}$$

   gdzie $T_i = \boldsymbol{U}(\vec{\tau}, \text{Tyargs}_{c_i}\vec{\alpha}_i)$, dla wszystkich $1 \leqslant i \leqslant n$ takich, że $\vec{\tau}$ i $\text{Tyargs}_{c_i}\vec{\alpha}$ unifikują, $\vec{\alpha} \cap (\boldsymbol{F}(\vec{\tau}) \cup \boldsymbol{F}(E) \cup \boldsymbol{F}(\theta)) = \varnothing$ ($\vec{\alpha}$ są nowymi zmiennymi). Pozostałe T_i są dowolne.

6. *CONS: Constructor*

$$\frac{}{E \vdash c\colon \text{Inst}_c\vec{\sigma} \to d(\text{Tyargs}_c\vec{\sigma})} \text{ jeśli } c \in \boldsymbol{C}(d)$$

   odpowiednio założenie na temat środowiska E:

$$(\forall d)(\forall c \in \boldsymbol{C}(d)) \;\; \text{niech } c\colon \forall\vec{\alpha}.\text{Inst}_c\vec{\alpha} \to d(\text{Tyargs}_c\vec{\alpha}) \in E, \;\; \text{gdzie } \#\vec{\alpha} \text{ jest arnością } d$$

gdzie $\boldsymbol{C}(d)$ jest zbiorem konstruktorów typu d, $\text{Tyargs}_c\vec{\sigma}$ jest wektorem argumentów typu zadanym przez konstruktor c przy parametrach $\vec{\sigma}$, $\text{Inst}_c\vec{\sigma}$ jest wektorem typów argumentów konstruktora c dla parametrów $\vec{\sigma}$, oraz $\text{Tyargs}_c\vec{\sigma} = [\vec{\sigma}/\vec{\alpha}]\text{Tyargs}_c\vec{\alpha}$, $\text{Inst}_c\vec{\sigma} = [\vec{\sigma}/\vec{\alpha}]\text{Inst}_c\vec{\alpha}$.

Gałęzie case, dla których unifikator nie istnieje, są tzw. martwym kodem, nie osiągalnym podczas wykonania programu. T_i komunikuje typowi θ dokonany wybór. Przedstawiony system, jeśli pominąć fakt, że używamy prostej analizy przez przypadki zamiast pełnego dopasowywania wzorca, odpowiada minimalnej wersji HMG, obejmowanej przez HMG(X) dla wszystkich języków więzów X.

Example 2.53. Zobaczmy definicję typu lista z parametrem oznaczającym długość.

$$\begin{aligned} \text{Nil} &: \forall\alpha.\text{list}(\alpha, 0) \\ \text{Cons} &: \forall\alpha\beta.\alpha \to \text{list}(\alpha, \beta) \to \text{list}(\alpha, 1+\beta) \end{aligned}$$

2.4.1.1 Generowanie η-długich β-normalnych polimorficznych λ-termów: Prolog.

Niestety, "czysty" prolog odpowiada generowaniu termów tylko dla języków pierwszego rzędu (nie mylić z systemami typów pierwszego rzędu; nawet system przedstawiony powyżej jest pierwszego rzędu: pod zmienne typu nie można podstawiać skwantyfikowanych typów, czyli schematów typu). Podstawowym krokiem, który trzeba podjąć, jest przejście od klauzul hornowskich, "definite clauses" ($a \to b \to ... \to c$, gdzie $a, b, ..., c$ są atomowe) do dowolnych wyrażeń implikacyjnych (czyli zawierających $\to$ jako jedyny spójnik logiczny). System typów, dla którego generowanie termów η-długich β-normalnych można osiągnąć dostosowując metodę SLD-rezolucji, to:

Definition 2.54. *Reguły typowania dla polimorficznych λ-termów:*

1. *VAR: Variables.*
$$\frac{E(x) = \forall\vec{\alpha}.\tau}{E \vdash x : [\vec{\sigma}/\vec{\alpha}]\tau}$$
2. *ABS: λ-abstractions.*
$$\frac{E.x : \sigma \vdash e : \tau}{E \vdash \lambda x.e : \sigma \to \tau}$$
3. *APP: Applications.*
$$\frac{E \vdash e : \sigma \to \tau \qquad E \vdash e' : \sigma}{E \vdash ee' : \tau}$$
4. *CONS: Constructor*
$$\frac{c \in \boldsymbol{C}(d)}{E \vdash c : \mathrm{Inst}_c\vec{\sigma} \to d(\mathrm{Tyargs}_c\vec{\sigma})}$$

odpowiednio założenie na temat środowiska E:

$$(\forall d)(\forall c \in \boldsymbol{C}(d)) \;\; \textit{niech } c : \forall\vec{\alpha}.\mathrm{Inst}_c\vec{\alpha} \to d(\mathrm{Tyargs}_c\vec{\alpha}) \in E, \;\; \textit{gdzie } \#\vec{\alpha} \textit{ jest arnością } d$$

przy oznaczeniach jak powyżej.

2.4.1.2 W stronę pełnego HMG(X).

Definicje lokalne nie wpływają na logikę, a jedynie pozwalają wielokrotnie wykorzystywać wyniki pośrednie przeszukiwania.

Definicje rekurencyjne w postaci nieograniczonej są paradoksalne: rozszerzają zbiór aksjomatów o fałszywy schemat zdaniowy $(X \to X) \to X$. (Dlatego w GP system typów powinien być rozszerzony by wymuszać własność stopu.) W przypadku rekurencji "zwykłej" zmienne wolne są skwantyfikowane wspólnie: $\forall((X \to X) \to X)$, a w przypadku rekurencji polimorficznej są skwantyfikowane niezależnie: $\forall((\forall(X) \to X) \to X)$.

Dekonstrukcja (analiza przez przypadki) z punktu widzenia logiki jest eliminacją spójnika "lub", a następnie eliminacją kwantyfikatora egzystencjalnego w każdym dysjunkcie. Jej wprowadzenie odpowiada wprowadzeniu aksjomatów:

$$(\exists\vec{\sigma}.d(\mathrm{Tyargs}_{c_1}\vec{\sigma}) \to \bigwedge \mathrm{Inst}_{c_1}\vec{\sigma}) \vee ... \vee (\exists\vec{\sigma}.d(\mathrm{Tyargs}_{c_n}\vec{\sigma}) \to \bigwedge \mathrm{Inst}_{c_n}\vec{\sigma})$$

gdzie $\boldsymbol{C}(d) = \{c_1, ..., c_n\}$. Z punktu widzenia przeszukiwania wprowadzenie definicji lokalnej odpowiada rozgałęzieniu się ze względu na możliwe wartości pewnych zmiennych wolnych w $d\vec{\tau}$: w różnych gałęziach zmienne te konkretyzują się (niezależnie), a (pomimo to) w wejściowym $d\vec{\tau}$ pozostają nieskonkretyzowane (porównaj regułę (CASE) z definicji 2.52, warunek na podstawienia unifikujące). Następnie pozyskiwana jest informacja specyficzna dla danego kostruktora.

2.4.2 Programy bez nieużytecznych definicji lokalnych.

Odpowiedni mechanizm definicji lokalnych, właściwie współpracujący z operatorem mutacji, jest kluczowy dla ponownego wykorzystywania raz wyewoluowanego kodu przez programy, bez angażowania mechanizmów rekombinacji. Nieużytecznych definicji nie będziemy generować, jeśli wielokrotne użycie kodu będzie się opierało na kodzie już wygenerowanym w jakimś celu.

Dość radykalne rozwiązanie wspierające "code reuse" polegać może na dodawaniu, dla każdego skonstruowanego podtermu, odpowiadającej mu zmiennej do środowiska. W momencie, gdy z podtermu "uciekają" zmienne wolne (np. wychodzimy poza odpowiednią λ-abstrakcję), typ zmiennej jest względem ich typów "podnoszony" i jeśli zmienna zostanie użyta, to będzie zaaplikowana do wartości zastępującej zmienne, które "uciekły". Cały mechanizm jednak trudno zintegrować z polimorficznym systemem typów, żeby typy zmiennych były możliwie ogólne.

Chapter 3
Rekombinacja i generalizacja

3.1 Rekombinacja swobodna.

Niech $\mathrm{Dom}(E_0) \subseteq \mathrm{Dom}(E_1)$ i $\mathrm{Dom}(E_0) \subseteq \mathrm{Dom}(E_2)$ oraz istnieją T_1, T_2, że $T_1E_0 = E_1|_{\mathrm{Dom}(E_0)}$ i $T_2E_0 = E_2|_{\mathrm{Dom}(E_0)}$ oraz $T_1E_0 \vdash e_0\colon \tau_0$. Znajdźmy podtermy e_1, e_2 wprowadzone do typizacji rodziców z typizacjami $E_1 \vdash e_1\colon \tau_1$ oraz $E_2 \vdash e_2\colon \tau_2$ (program e_0 – nadterm e_1 nazywa się matką, a program-nadterm e_2 – ojcem), dla których istnieje H_L $E_1(x) = E_2(h(x))$, oraz typ $H_R\tau_2 = \tau_1$, przy czym H_R nie podstawia pod zmienne występujące w E_2 ($\mathrm{Dom}(H_R) \cap \boldsymbol{F}(E_2) = \varnothing$). Trójkę h, H_L, H_R nazwiemy E_0-homomorfizmem typizowań $E_1 \vdash e_1\colon \tau_1$ oraz $E_2 \vdash e_2\colon \tau_2$. Rekombinacją swobodną matki e_0 na pozycji podtermu e_1 z ojcem na pozycji podtermu e_2 nazywamy program $H_L\ T_1E_0 \vdash [h(e_2)/e_1]e_0\colon\ H_L\tau_0$. Typizacja ta może być (podobnie, jak w wyniku mutacji) nadokreślona względem środowiska E_0; możemy więc przeprowadzić retypizację.

3.2 Anty-unifikacja drugiego rzędu – prosty przypadek.

Przedstawimy w tym podrozdziale anty-unifikację pierwszego rzędu w algebrze termów odpowiadającej danemu językowi rozszerzającemu λ-rachunek. Ta anty-unifikacja odpowiada pewnym prostym przypadkom anty-unifikacji drugiego rzędu. Termy algebry termów nazywam też λ-strukturami: są to termy, w których konstrukcje λ-rachunku (czy też języków rozszerzających język λ-termów) są symbolami funkcyjnymi, a stałe i zmienne λ-termu są symbolami stałymi. Jedyna modyfikacja względem sformułowań zwykłych algebr termów i anty-unifikacji pierwszego rzędu w nich wiąże się z obsługą α-równoważności i zmiennych związanych.

Warto porównać niniejsze sformułowanie z tym zaproponowanym w [9]. Nasza generalizacja radzi sobie ze zmiennymi związanymi przez bezpośrednią obsługę kontekstu (i ograniczenie do generalizacji "ground terms"), generalizacja w [9] przez "variable freezing" (eliminację zmiennych przez podstawienie stałych) i zabronienie λ-abstrakcji w argumentach. Generalizacja w [9] czy też w systemie CC jak w pracy Pfenninga [12] jest dodatkowo skomplikowana przez występowanie typów w termach ("explicit polymorphism"). Nasze generalizacje są czysto syntaktyczne, w sensie, że nie rozpatrują typów termów. Wynika to z założenia stosowania systemów typów bez annotacji typami w termach; cały ciężar spada oczywiście na dowody zgodności z odpowiednim systemem typów.

W kwestii skierowania porządku, "co kraj to obyczaj". W pracy [7] generalizację oznacza się $\sqcap$, a w pracach [12] oraz [9] przez $\sqcup$.

3.2.1 Definicja.

Zamiast formalnej definicji λ-struktur przykład: programowi

$$\mathrm{fix}\, fx.\mathrm{case}\, x\, \mathrm{of}\, \{\mathrm{Nil} \Rightarrow 0; \mathrm{Cons} \Rightarrow \lambda y.\lambda z. +1(fz)\}$$

odpowiada λ-struktura:

$$\mathrm{fix}_{f,x}(\mathrm{case}^{\mathrm{list}}(x, 0, \lambda_y(\lambda_z(@(@(+,1), @(f,z))))))$$

Stosujemy adaptację algorytmu Gerarda Hueta obsługującą zmienne związane (kontekst):

Definition 3.1. *Anty-unifikacja pierwszego rzędu λ-struktur:*

$$\begin{aligned} F_{\vec{x}}(s_1,...,s_n)\sqcap_{\vec{z}}F_{\vec{y}}(t_1,...,t_n) &= F_{\vec{u}}([\vec{u}/\vec{x}]s_1\sqcap_{\vec{z}.\vec{u}}[\vec{u}/\vec{y}]t_1,...,[\vec{u}/\vec{x}]s_n\sqcap_{\vec{z}.\vec{u}}[\vec{u}/\vec{y}]t_n), \qquad F\in\mathcal{C} \\ s\sqcap_{\vec{z}}s &= s \\ s\sqcap_{\vec{z}}t &= \phi([\vec{v}/\vec{z}]s,[\vec{v}/\vec{z}]t), \qquad \textit{w przec. przyp.} \end{aligned}$$

gdzie $\vec{x}$ i $\vec{y}$ są ciągami tej samej długości zera lub więcej zmiennych związanych przez konstrukcję F, $\vec{u}$ jest ciągiem nowych (nie występujących w $\vec{s}$ ani w $\vec{t}$) zmiennych języka, $\vec{v}$ jest odcinkiem początkowym długości $\#\vec{z}$ ustalonego ciągu zmiennych $v_1, v_2, ...$ rozłącznego z pozostałymi zmiennymi używanymi przez algorytm. ϕ jest bijekcją między parami λ-struktur modulo α-równoważność a zbiorem zmiennych M (rozłącznym ze zmiennymi języka). Niech

$$s\sqcap t \;=\; s\sqcap_{\epsilon}t$$

gdzie ϵ jest ciągiem pustym.

Dla języka ML mamy $\mathcal{C} = \{@,\ \lambda,\ \text{fix},\ \text{let}\} \cup \{\text{case}^d\}_{d\in D}$ gdzie D jest zbiorem typów algebraicznych; @,case^d wiążą zero zmiennych, λ, let wiążą jedną zmienną, fix wiąże dwie zmienne. Dalej utożsamiamy termy języka i odpowiednie struktury, przekształcając jedne w drugie w miarę potrzeby. Oznacza to, że język rozszerzamy o tzw. meta-zmienne ze zbioru M. Podstawienie $\vec{v}$ zapewnia, że podtermy różniące się tylko nazwami zmiennych wolnych będą utożsamiane.

3.2.2 Własność: maksymalnie specyficzna generalizacja.

Pokażemy teraz, że powyżej zdefiniowana operacja jest maksymalnie specyficzną generalizacją (anty-unifikacją) dla porządku podstawieniowego.

Zdefiniujmy teraz podstawienie obsługujące kontekst. Pamiętajmy, że zwykłe podstawienie $[\vec{t}/\vec{x}]s$ nie podstawia pod zmienne związane.

Definition 3.2. *Podstawienie obsługujące kontekst:*

$$\begin{aligned} \%(T,\vec{x};F_{\vec{y}}(t_1,...,t_n)) &= F_{\vec{u}}(\%(T,\vec{x}.\vec{u};[\vec{u}/\vec{y}]t_1),...,\%(T,\vec{x}.\vec{u};[\vec{u}/\vec{y}]t_n)) \qquad F\in\mathcal{C} \\ \%(T,\vec{x};Y) &= [\vec{x}/\vec{y}]s \qquad \textit{jeśli}\ \ Y\in M, T(Y)=\vec{y};s \\ \%(T,\vec{x};s) &= s \qquad \textit{w przeciwnym przypadku} \end{aligned}$$

gdzie $\vec{u}$ jest ciągiem nowych (nie występujących w $\vec{t}$) zmiennych języka.

Definition 3.3.

$$\vec{x};t\leqslant\vec{x};s \;\equiv\; (\exists T)\%(T,\vec{x};t)=_{\alpha}s$$

Pokażmy na rozgrzewkę

Theorem 3.4.

$$\begin{aligned} \vec{y};r\sqcap_{\vec{y}}s &\leqslant \vec{y};r \\ \vec{y};r\sqcap_{\vec{y}}s &\leqslant \vec{y};s \end{aligned}$$

Proof. Pokazujemy pierwszą nierówność przez indukcję względem złożoności r a drugą względem złożoności s. W odpowiednim miejscu algorytmu ⊓, wystarczy brać $T(\phi([\vec{v}/\vec{z}]s,[\vec{v}/\vec{z}]t))=\vec{v};[\vec{v}/\vec{z}]s$ dla pierwszej nierówności i $T(\phi([\vec{v}/\vec{z}]s,[\vec{v}/\vec{z}]t))=\vec{v};[\vec{v}/\vec{z}]t$ dla drugiej, gdzie $\vec{z},\vec{v},s,t$ jak w algorytmie ⊓. □

Theorem 3.5.

$$(\forall\vec{x};t)\ \ \vec{x};t\leqslant\vec{x};r\wedge\vec{x};t\leqslant\vec{x};s\ \Rightarrow\vec{x};t\leqslant\vec{x};r\sqcap_{\vec{x}}s$$

Proof. Ustalmy $\vec{x}; t$. Z założenia twierdzenia mamy podstawienia P, Q:

$$\%(P, \vec{x}; t) = r, \%(Q, \vec{x}; t) = s$$

. Niech $P(X_i) = \vec{x}^i$; p_i, $Q(X_i) = \vec{y}^i$; q_i dla $X_i \in \mathrm{Dom}(P) = \mathrm{Dom}(Q) = \boldsymbol{FM}(t)$. Pokażemy, że $T(X_i) = \vec{u}$; $[\vec{u}/\vec{x}^i]p_i \sqcap_{\vec{u}} [\vec{u}/\vec{y}^i]q_i$ daje $\%(T, \vec{x}; t) = r \sqcap_{\vec{x}} s$. Przeprowadźmy indukcję względem złożoności t, przez analizę wyprowadzenia $r \sqcap_{\vec{x}} s$.

1. $t = X_i \in M$. Wt?dy ? $[\vec{x}/\vec{x}^i]p_i = r$, $[\vec{x}/\vec{y}^i]q_i = s$ oraz $T(Y) = \vec{x}; r \sqcap_{\vec{x}} s$.
2. $t = F_{\vec{z}}(t_1, ..., t_n)$. Z założenia twierdzenia, musi być $r = F_{\vec{v}}(r_1, ..., r_n)$, $s = F_{\vec{w}}(s_1, ..., s_n)$; stąd

   $$r \sqcap_{\vec{x}} s \;=\; F_{\vec{u}}([\vec{u}/\vec{v}]r_1 \sqcap_{\vec{x}.\vec{u}} [\vec{u}/\vec{w}]s_1, ..., [\vec{u}/\vec{v}]r_n \sqcap_{\vec{x}.\vec{u}} [\vec{u}/\vec{w}]s_n$$

   Ponieważ $t_i \leqslant r_i$ i $t_i \leqslant s_i$, co wynika z definicji podstawienia $\%$ i α-równoważności, z zał. ind. mamy $\vec{x}.\vec{u}; [\vec{u}/\vec{z}]t_i \leqslant \vec{x}.\vec{u}; [\vec{u}/\vec{v}]r_i \sqcap_{\vec{x}.\vec{u}} [\vec{u}/\vec{w}]s_i$. Stąd

   $$r \sqcap_{\vec{x}} s \;=\; F_{\vec{u}}(\%(T_1, \vec{x}.\vec{u}; [\vec{u}/\vec{z}]t_1), ..., \%(T_n, \vec{x}.\vec{u}; [\vec{u}/\vec{z}]t_n))$$

   gdzie $T_j(X_i) = \vec{u}; [\vec{u}/\vec{x}^i]p_i \sqcap_{\vec{u}} [\vec{u}/\vec{y}^i]q_i = T(X_i)$.
3. $t = c$. Wtedy musi być $r = s = r \sqcap s = c$.

□

3.2.3 Zgodność rekombinacji z systemem typów.

Definition 3.6. *Niech dane będą programy s, t, oraz $T_1E_0 \vdash s : T_1\tau_0$ i $T_2E_0 \vdash t : T_2\tau_0$. Rekombinacją nazwiemy program $r = \%(R, \epsilon; s \sqcap_\epsilon t)$, gdzie $\%(S, \epsilon; s \sqcap_\epsilon t) = s$, $\%(T, \epsilon; s \sqcap_\epsilon t) = t$ oraz $R(X_i) = S(X_i)$ lub $R(X_i) = T(X_i)$. Rekombinację nazywamy poprawną, jeśli $T_3E_0 \vdash r : T_3\tau_0$ dla pewnego T_3.*

Zbadajmy teraz, dla jakich podstawień R, $r = \%(R, s \sqcap t)$ jest rekombinacją (tzn. r jest typizowalny). Otrzymamy w ten sposób algorytm rekombinacji. Badanie będzie polegało na generalizacji wyprowadzeń najogólniejszych typów dla s i t w środowisku E_0.

Zastosujmy algorytm inferencji typu (prototypem jest algorytm $\mathcal{W}$) do problemu E_0, $s \sqcap t$; niech zwraca wyznaczone typy meta-zmiennych. Modyfikujemy algorytm tak, że gdy ma wyznaczać typ któregoś z wystąpień meta-zmiennej X_i (w środowisku E_1), wyznacza typy dla $S(X_i)$ oraz $T(X_i)$ (z przemianowanymi zmiennymi kontekstu tak, aby odpowiadały zmiennym wprowadzonym do E_1) w środowisku E_1. Następnie generalizuje znalezione typy (zwykłym algorytmem anty-unifikacji pierwszego rzędu). Za zmienne generalizacji wstawia nowe konstruktory typu zaaplikowane do listy zmiennych typu: listy wszystkich zmiennych występujących w podtermach "różnicowych" (typach z podstawień generalizacji pod tą zmienną), wynik przekazuje jako znaleziony typ. Podobnie, generalizuje typy w zwracanych podstawieniach, i zwraca podstawienie z typami ze zmiennymi generalizacji zastąpionymi przez nowe konstruktory. (Nazwijmy te nowe konstruktory zmiennymi generalizacji typu, TGV.) Jeśli któreś z podstawień generalizacji jest zmienną typu, to nazywamy ją zmienną projekcyjną tej TGV. Podstawienia generalizacji danej TGV nazywamy gałęziami tej zmiennej. Unifikacja TGV z typem polega na osobnych unifikacjach gałęzi z tym typem, i jeśli pojawiają się w nim TGV, na unifikowaniu z odpowiednią gałęzią każdej TGV. Jeśli TGV traci w wyniku unifikacji wszystkie zmienne projekcyjne, jej gałęzie są ponownie generalizowane, i jeśli znaleziona anty-instancja jest nietrywialna, to jest podstawiana pod tą TGV.

Zmienne typu podzielimy na specyficzne, wprowadzone w inferencji typu wystąpienia meta-zmiennej, i niespecyficzne, pozostałe. Specyficzne zmienne typu będą się pojawiały tylko w odpowiednich gałęziach różnych TGV. Jeśli w wyniku unifikacji gałęzi powstają podstawienia niespecyficzne, to są one ujmowane w TGV, która na pozostałej gałęzi ma (nową, specyficzną) zmienną projekcyjną. Stąd, jeśli inna podgałąź będzie unifikować do tej zmiennej (zastąpionej przez skonstruowaną TGV), to podstawienia zostaną zgeneralizowane, i we wszystkie wystąpienia (najpierw zmiennej niespecyficznej, następnie zamienionej na TGV) trafia anty-instancja z odpowiednimi TGV.

Jeśli TGV jest unifikowana z podgałęzią innej TGV, to para tych zmiennych jest zapamiętywana w zbiorze unifikowanych TGV. Jeśli następuje podstawienie pod TGV, to pary zawierające tą zmienną w zbiorze unifikowanych TGV są zastępowane przez pary "zmienna unifikowana do podstawianej w zastępowanej parze" - "TGV występujące w podstawieniu". W zbiorze unifikowanych TGV uwzględniamy też podstawienia pod zwykłe zmienne typu, kiedy pewne TGV tracą zmienną projekcyjną. Jeśli zmienna typu zostaje skwantyfikowana (zgeneralizowana w sensie polimorfizmu), to w zbiorze unifikowanych TGV przestaje ona być zmienną projekcyjną.

Despecyfikacja zmiennych. Po zakończeniu inferencji, usuwamy TGV której obydwie gałęzie są zmiennymi specyficznymi (i albo obydwie są nieskwantyfikowane, albo obydwie są skwantyfikowane), zastępując te zmienne nową zmienną niespecyficzną (nie ograniczoną do gałęzi odpowiadających jednemu programowi). Usuwamy też pary zawierające tą zmienną ze zbioru unifikowanych TGV. (Usuwamy też wszystkie TGV, które też miały te zmienne jako projekcyjne, zgodnie z zasadą, że generalizujemy TGV które tracą swoje zmienne projekcyjne.) Postępujemy tak, dopóki są TGV z dwiema specyficznymi zmiennymi projekcyjnymi. Jest to krok niedeterministyczny algorytmu: wynik zależy od kolejności wyboru TGV.

Silna despecyfikacja konkretyzująca stanowi ostatnią, opcjonalną fazę algorytmu. Można pozbyć się jeszcze większej ilości TGV, kosztem konkretyzacji "answer substitution". Usuwamy TGV mającą zmienną projekcyjną (może być niespecyficzna) nie skwantyfikowaną (czyli wolną w całym typowaniu), podstawiając pod zmienną projekcyjną pozostałą gałąź TGV, o ile zmienna projekcyjna nie występuje w tej gałęzi. Pamiętamy, aby dodać to podstawienie do "answer substitution". Gdy któraś TGV w wyniku starci swoje zmienne projekcyjne, generalizujemy gałęzie i powtarzamy despecyfikację dla (ewentualnie) powstałych TGV. Jest to krok niedeterministyczny.

Lemma 3.7. *Tak zmodyfikowany algorytm kończy pracę zwracając pewien typ i podstawienie.*

Nazwijmy dwa wystąpienia meta-zmiennych zależnymi, jeśli typ jednego zawiera TGV unifikowaną z TGV występującą w typie drugiego (tzn. para tych TGV należy do zbioru unifikowanych TGV), a silnie zależnymi, jeśli żadna TGV z pary nie zawiera zmiennej projekcyjnej.

Definition 3.8. *Rekombinacja podstawień typów:* $T_3 \in \boldsymbol{R}(T_1, T_2)$ *jeśli*

$$\begin{array}{rcll} T_3\alpha & = & H(\tau \sqcap \sigma) & \alpha \in \mathrm{Dom}(T_1) \cap \mathrm{Dom}(T_2) \\ \tau = T_1\alpha & & \sigma = T_2\alpha & \\ H\beta = T\beta & \vee & H\beta = S\beta & \beta \in \boldsymbol{MV}(\tau \sqcap \sigma) \\ T(\tau \sqcap \sigma) = \tau & & S(\tau \sqcap \sigma) = \sigma & \end{array}$$

gdzie $\sqcap$ *jest generalizacją pierwszego rzędu,* $\boldsymbol{MV}$ *zwraca zmienne wstawione przez generalizację.*

Theorem 3.9. *Rekombinacja, która pod wystąpienia zależne bez despecyfikacji konkretyzującej podstawia z tego samego podstawienia (rodzica), jest poprawna. Jeśli* $T_1E_0 \vdash s\colon T_1\tau_0$ *i* $T_2E_0 \vdash t\colon T_2\tau_0$*, to* $T_3E_0 \vdash r\colon T_3\tau_0$ *i* $T_3 \in \boldsymbol{R}(T_1, T_2)$*, dodatkowo jeśli* $T_1\tau_0$ *i* $T_2\tau_0$ *są pryncypalnymi typami* s *i* t*, to* $T_3\tau_0$ *jest typem pryncypalnym (ang. principal type) programu* r*. Rekombinacja, która pod wystąpienia zależne po despecyfikacji konkretyzującej podstawia z tych samych podstawień, jest poprawna. Rekombinacja, która pod wystąpienia silnie zależne podstawia z różnych podstawień (rodziców), nie jest poprawna.*

Proof. (Szkicowe uzasadnienie twierdzenia.)

1. Jeśli pod TGV podstawimy odpowiednie podstawienia generalizacji, ze zmiennymi zastępionymi przez argumenty TGV, to uzyskamy przebiegi algorytmu odpowiadające przebiegom dla rodziców, jeśli staniemy przed fazą despecyfikacji konkretyzującej. Kolejne generalizacje nie wpływają na odpowiedniość z przebiegiem algorytmu $\mathcal{W}$ dla zagadnień E_0, r i E_0, s. Despecyfikacja zmiennych (niekonkretyzująca) odpowiada przemianowaniu zmiennych nie zmieniającemu wyprowadzenia (żadne zmienne nie zostają utożsamione).

2. Każda TGV jest związana z pewną meta-zmienną w $s \sqcap t$: każda TGV powstaje albo w generalizacji typizowań dla podstawień pod meta-zmienną, albo z podstawienia pod zmienną niespecyficzną powstałego przy typizowaniu podstawień pod meta-zmienną, wtedy jest związana z tą meta-zmienną, albo z generalizacji gałęzi innej TGV, wtedy jest związana z meta-zmienną macierzystej TGV.
3. Obierzmy pewną rekombinację R. Będziemy rekonstruować typizację dla rekombinanta $\%(R,\ s \sqcap t)$ przez wybieranie tych gałęzi TGV, które podstawienie ($S(X)$ czy $T(X)$) zostało wybrane dla meta-zmiennej X związanej z daną TGV.
4. Wyprowadzenie typu z tak podstawionymi TGV odpowiada wyprowadzeniu typu dla rekombinanta, o ile ono istnieje: część odpowiadająca $s \sqcap t$ jest wspólna dla wszystkich rekombinantów, jako część odpowiadającą meta-zmiennej X zostało wybrane pewne wyprowadzenie typu dla tego właśnie podstawienia $R(X)$.
5. Wyprowadzenie typu dla $\%(R,\ s \sqcap t)$ daje się zrekonstruować, ponieważ komunikacja wewnątrz wyprowadzenia nie jest zaburzona: jeśli pewne dwa typy są unifikowane, to zawierają TGV albo są częścią gałęzi TGV związanych z meta-zmiennymi zależnymi, czyli zgodnie z warunkami twierdzenia, R prowadzi do wybrania odpowiadających sobie gałęzi, i na mocy odpowiedniego wyprowadzenia dla s albo dla t, typy te są unifikowalne.
6. W zrekonstruowanej typizacji $T_3 \in \boldsymbol{R}(T_1,\ T_2)$ ponieważ algorytm (bez despecyfikacji konkretyzującej) zwraca generalizację podstawień. Zrekonstruowany typ jest pryncypalny, ponieważ jest wynikiem zwracanym przez algorytm inferencji dla rekombinanta.
7. Despecyfikacja konkretyzująca zachowuje poprawność, ponieważ podstawia pod zmienne typu wolne w typizacji.
8. Krzyżowanie podstawień silnie zależnych daje nietypizowalny program, ponieważ w (deterministycznym) przebiegu algorytmu inferencji napotyka się wtedy na niespełnialną unifikację różnych gałęzi TGV zaczynających się od stałych (konstruktorów typu). □

Zauważmy, że algorytm nie uwzględnia tożsamości meta-zmiennych przy różnych ich wystąpieniach w anty-instancji. Dla potrzeb programowania genetycznego wydaje się nawet korzystne ograniczenie generalizacji do termów liniowych ze względu na meta-zmienne; jednocześnie bardzo by to uprościło algorytm generalizacji.

3.2.4 Związek z algorytmami z prac [12] i [9].

Pod względem syntaktycznym zaprezentowany algorytm jest słabszy od tych ze wspomnianych prac przez brak obsługi permutacji kontekstu, jednak łatwo go rozszerzyć, aby taką obsługę zapewniał. Natomiast mocniejszy jest od [12] w tym, że stosuje generalizację drugiego rzędu z "subterm restriction", czyli ograniczeniem do podtermów generalizowanych termów (i odróżnieniem zmiennych termu od meta-zmiennych generalizacji), a nie "pattern restriction", czyli ograniczeniem do podtermów które są zmiennymi związanymi (obejmowaną przez "subterm restriction"). Od [9] mocniejszy jest w tym, że pozwala na zmienne związane lokalnie, czyli m.in. na λ-abstrakcje w pozycjach argumentowych. Pokazuje to, że wspomniane algorytmy są syntaktycznie bliskie algorytmom pierwszego rzędu; zasadniczą różnicą z syntaktycznego punktu widzenia jest obsługa permutacji (tutaj permutacji zmiennych związanych, kontekstu).

Pod względem obsługi typów algorytm z paragrafu 3.2.1 jest nie do przyjęcia. Jedynie w przypadku systemów z jawnym polimorfizmem ("explicit polymorphism") (kodujących typy wewnątrz termów), jak te użyte we wspomnianych pracach, algorytm syntaktyczny może zidentyfikować typy równoważne w różnych kontekstach. Właśnie dlatego opracowaliśmy algorytm "inferencji typu dla anty-instancji" na wzór systemów z "ad-hoc polymorphism". Czy da się ten wynik przenieść na grunt polimorfizmu parametrycznego, pozostaje do zbadania.

3.3 Generalizacja drugiego rzędu i ogólne struktury.

Teraz zajmiemy się algorytmem anty-unifikacji "prawdziwie" drugiego rzędu.

3.3.1 Generalizacje rekombinatorowe z pracy [7].

W tym paragrafie prezentuję definicje i wybrane fakty z rozdziałów 5 i 6 pracy Haskera.

3.3.1.1 Definicje ogólne i algorytm dla termów monadycznych.

Definition 3.10. *Zbiór generalizacji termów* a, b

$$\boldsymbol{G}(a,b) \;=\; \{\langle \theta_i\colon t_i \to a, \theta_i'\colon t_i \to b\rangle\}$$

Aplikacja podstawienia do generalizacji

$$\rho\colon \langle \theta\colon t \to a, \theta'\colon t \to b\rangle \to \langle \sigma\colon s \to a, \sigma'\colon s \to b\rangle \;\Leftrightarrow\; \rho(t) = s, \theta = \rho\circ\sigma, \theta' = \rho\circ\sigma'$$

Przez $g_1 \to g_2$ *oznaczamy fakt* $(\exists\rho)\rho\colon g_1 \to g_2$.

Definition 3.11. $\Gamma = \mathrm{MSG}(a,b)$ *jest zbiorem maksymalnie specyficznych generalizacji, jeśli*

1. *(Soundness:)* $\Gamma \subseteq \boldsymbol{G}(a,b)$
2. *(Completeness:)* $(\forall g \in \boldsymbol{G}(a,b))(\exists g' \in \Gamma)\, g \to g'$
3. *(Minimality:)* $(\forall g, g' \in \Gamma) g \to g' \Rightarrow g = g'$
4. *(Uniqueness:)* $(\forall g \in \boldsymbol{G}(a,b) \forall g' \in \Gamma)\, \rho\colon g \to g' \wedge \rho'\colon g \to g' \Rightarrow \rho = \rho'$

Definition 3.12. *Niech* $\boldsymbol{G}'(a,b)$ *będzie podzbiorem* $\boldsymbol{G}(a,b)$ *takim, że jeśli* $\langle \theta_1\colon s \to a, \theta_2\colon s \to b\rangle \in \boldsymbol{G}'(a,b)$ *i* $f \in \boldsymbol{F}(s)$, *to* $\theta_1(f) = \theta_2(f) \Rightarrow \theta_1(f) = f$.

Definition 3.13. *Para zmiennych* f, g *w termie* t *jest przyległa (ang. adjacent) jeśli* t *zawiera podterm równy* fg.

Definition 3.14. *Generalizacja* $\langle \theta_1\colon t \to a_1,\ \theta_2\colon t \to a_2\rangle$ *jest redundantna, jeśli* f *i* g *są przyległymi zmiennymi wolnymi w* t *oraz zachodzi jedno z:* $\theta_1(f) \neq f, \theta_2(f) \neq f, \theta_1(g) \neq g$, $\theta_2(g) \neq g$.

Definition 3.15. $\boldsymbol{CG}(a,b)$, *zbiór skondensowanych generalizacji, to podzbiór* $\boldsymbol{G}(a,b)$ *nie zawierający generalizacji redundantnych. Zbiór maksymalnie specyficznych generalizacji skondensowanych,* $\mathrm{MSC}(a,b)$, *to zbiór maksymalnie specyficznych generalizacji w* $\boldsymbol{CG}(a,b)$.

Definition 3.16. *Kiedy* $g_1 \to g_2$ *w* $\boldsymbol{CG}(a,b)$, *to mówimy, że* g_1 *jest bardziej ogólna (mniej specyficzna) niż* g_2, $g_1 \geqslant g_2$.

Oznaczmy przez

$$\frac{t, \theta_1, \theta_2}{t', \theta_1', \theta_2'}$$

regułę przekształcania generalizacji $\langle \theta_1\colon t \to a,\ \theta_2\colon t \to b\rangle$ w $\langle \theta_1'\colon t' \to a,\ \theta_2'\colon t' \to b\rangle$, gdzie "mianownik" jest ściśle bardziej specyficzny niż "licznik".

Definition 3.17. *Żeby policzyć* $\mathrm{MSC}(a,b)$ *zaczynamy z* $g_0 \in \boldsymbol{CG}(a,b)$ *i stosujemy nast. reguły, aż żadna nie będzie stosowalna:*

1. *(Delete)*

$$\frac{t, \theta_1 \cup \{s/f\}, \theta_2 \cup \{s/f\}}{\{s/f\}(t), \theta_1, \theta_2}$$

2. (Merge)

$$\frac{t, \theta_1 \cup \{r/f, r/f'\}, \theta_2 \cup \{s/f, s/f'\}}{\{f/f'\}(t), \theta_1 \cup \{r/f\}, \theta_2 \cup \{s/f\}}$$

3. (Factor)

$$\frac{t, \theta_1 \cup \{r\,K\,r'/f\}, \theta_2 \cup \{s\,K\,s'/f\}}{\{h\,K\,h'/f\}(t), \theta_1 \cup \{r/h, r'/h'\}, \theta_2 \cup \{s/h, s'/h'\}}$$

3.3.1.2 Relewantne kombinatory i algorytm dla termów poliadycznych.

Definition 3.18. *Kombinatory kartezjańskie:*

$$\pi\langle p, p'\rangle = p \quad \pi'\langle p, p'\rangle = p' \quad \langle \pi r, \pi' r\rangle = r \qquad !_A t = !_B$$

Można pokazać $\langle p,\ p'\rangle r = \langle p\ r,\ p'\ r\rangle$. $!_A$: $A \to u$ posyła argument typu A do typu "unit", tzn. ignoruje argument.

Problem dopasowywania (ang. matching) (oraz generalizacji) nie jest dobrze określony dla typów produktowych w ogólności; pod jedną zmienną można podstawiać coraz bardziej złożone struktury, a pod drugą odpowiednie projekcje pozbywające się ich. Wybranym ograniczeniem jest ograniczenie typów wynikowych funkcji do typów bez produktów. Ale to nie rozwiązuje jeszcze problemu określoności generalizacji, ponieważ projekcje ciągle pozwalają ignorować podtermy anty-instancji. Dlatego wprowadza się relewantne kombinatory.

Produkt typów i konstruktor pary oznaczamy przez kropkę, np. $A \cdot (B \cdot C) \cong (A \cdot B) \cdot C$.

Definition 3.19. *Restruktorem nazywamy parę dwóch wzorców $p \mapsto q$, gdzie wzorce p, q składają się ze zmiennych $x, y, z, ...$, symbolu specjalnego 1_u oraz operatora binarnego "$\cdot$". Restruktor nazywamy relewantnym, jeśli p jest liniowe, ozn.* $\mathrm{linear}(p)$, *oraz $\boldsymbol{F}(p) = \boldsymbol{F}(q)$, tzn. każda zmienna występująca w restruktorze pojawia się po prawej stronie. Zakładamy następujące równości między termami (po prawej stronie warunki zachodzenia):*

$$\begin{array}{rcll}
(s_1 \cdot s_2)(t_1 \cdot t_2) & = & s_1 t_1 \cdot s_2 t_2 & \\
\mathbf{1}_X & = & p \mapsto p & p{:}_P X \\
p \mapsto q & = & \sigma(p) \mapsto \sigma(q) & \mathrm{linear}(\sigma(p)) \\
(p \mapsto q)t & = & \theta(q)(r \mapsto \sigma(q)) & \theta(p) = t \wedge \sigma(p) = r \\
(p \mapsto q)(r \mapsto p) & = & r \mapsto q & \\
(p_1 \mapsto q_1) \cdot (p_2 \mapsto q_2) & = & p_1 \cdot p_2 \mapsto q_1 \cdot q_2 & \mathrm{linear}(p_1 \cdot p_2)
\end{array}$$

Powyżej, θ jest podstawieniem termów pod zmienne wzorca, a σ jest podstawieniem wzorców pod zmienne wzorca. Przykładowa interpretacja reguły czwartej: θ wyłuskuje wyniki t i $\theta(q)$ ustawia je we wzorzec q, następnie $\sigma(q)$ porządkuje argumenty według wzorca q, żeby je dostarczyć $\theta(q)$; jest to możliwe, bo argumenty rozdzielają się według uszczegółowienia wzorca p do funkcji atomowych t (których typ wynikowy nie jest produktem).

Wprowadzamy następujące skróty:

$$\begin{array}{lrcrcl}
\text{associate left:} & \alpha_{X,Y,Z} & = & p \cdot (q \cdot r) & \mapsto & (p \cdot q) \cdot r \\
\text{associate right:} & \alpha^{-1}_{X,Y,Z} & = & (p \cdot q) \cdot r & \mapsto & p \cdot (q \cdot r) \\
\text{delete left:} & \lambda_X & = & \mathbf{1_u} \cdot p & \mapsto & p \\
\text{insert left:} & \lambda^{-1}_X & = & p & \mapsto & \mathbf{1_u} \cdot p \\
\text{delete right:} & \varrho_X & = & p \cdot \mathbf{1_u} & \mapsto & p \\
\text{insert right:} & \varrho^{-1}_X & = & p & \mapsto & p \cdot \mathbf{1_u} \\
\text{commute:} & \gamma_{X,Y} & = & p \cdot q & \mapsto & q \cdot p \\
\text{duplicate:} & \delta_X & = & p & \mapsto & p \cdot p
\end{array}$$

Definition 3.20. *Niech $\mathcal{R}$ będzie zbiorem termów generowanych przez produkcję*

$$R ::= \mathbf{1} | \alpha | \alpha^{-1} | \lambda | \lambda^{-1} | \varrho | \varrho^{-1} | \gamma | \delta | R\,R | R \cdot R$$

Ponieważ zachodzą równania:

$$\begin{array}{rclcrcl}(\alpha\cdot\mathbf{1})\alpha(\mathbf{1}\cdot\alpha) &=& \alpha\alpha & & (\varrho\cdot\mathbf{1})\alpha &=& \mathbf{1}\cdot\lambda\\ \lambda_{\boldsymbol{u}} &=& \varrho_{\boldsymbol{u}} & & \gamma\gamma &=& \mathbf{1}\\ \lambda\gamma &=& \varrho & & (\gamma\cdot\mathbf{1})\alpha(\mathbf{1}\cdot\gamma) &=& \alpha\gamma\alpha\\ \alpha(\mathbf{1}\cdot\delta)\delta &=& (\delta\cdot\mathbf{1})\delta & & \gamma\delta &=& \delta\\ \lambda_{\boldsymbol{u}}\delta_{\boldsymbol{u}} &=& \mathbf{1}_{\boldsymbol{u}} & & \varrho_{\boldsymbol{u}}\delta_{\boldsymbol{u}} &=& \mathbf{1}_{\boldsymbol{u}}\end{array}$$

mamy fakt

Theorem 3.21. *Dla każdego relewantnego restruktora $p\mapsto q$ istnieje $r\in\mathcal{R}$, $r=p\mapsto q$.*

Dopasowywanie (ang. matching) relewantnych kombinatorów jest rozstrzygalne np. przez sprowadzenie do kombinatorów kartezjańskich.

Definition 3.22. *Zmienne wolne f, g w t są przyległe, jeśli istnieje podterm fs w t i relewantny restruktor $p\mapsto q$ taki, że $f(p\mapsto q)s=f(g\cdot s')$ dla pewnego s'.*

Czyli zmienne są przyległe, jeśli nie są rozdzielone przez stałą.

Definition 3.23. *Podstawienie $\{t/x\}$ jest przemianowaniem, jeśli t jest postaci $y\tau$ dla pewnej zmiennej wolnej y oraz obustronnie odwracalnego restruktora relewantnego τ.*

Theorem 3.24. *Dla stałej K w t istnieją r, s takie, że $t=r(K\cdot\mathbf{1})s$.*

Definition 3.25. *Asocjacyjne restruktory, $\mathcal{A}$, to restruktory generowane przez produkcję*

$$A::=\mathbf{1}|\alpha|\alpha^{-1}|A\,A|A\cdot A$$

Jeśli oznaczymy $\vec{s_n}=s_1\cdot(s_2\cdot ...(s_{n-1}\cdot s_n)...)$, to, jeśli rozumieć "podterm" jako "podterm pierwszego rzędu" (czyli np. s nie jest podtermem w st):

Theorem 3.26. *Dla termu t i podtermu Ks o m wystąpieniach w t, istnieją $r, \vec{s_n}, \mu\in\mathcal{A}$, $\tau\in\mathcal{R}$ takie, że $t=r(\delta^m K\mu(\vec{s_n})\cdot\mathbf{1})\tau$.*

Definition 3.27. *Żeby policzyć* MSC(a, b) *zaczynamy z $g_0\in\boldsymbol{CG}(a,b)$ i stosujemy nast. reguły, aż żadna nie będzie stosowalna:*

1. *(Delete)*
$$\frac{t,\theta_1\cup\{s/f\},\theta_2\cup\{s/f\}}{\{s/f\}(t),\theta_1,\theta_2}$$
2. *(Merge) Dla obustronnie odwracalnego restruktora τ*
$$\frac{t,\theta_1\cup\{r/f, r\tau/f'\},\theta_2\cup\{s/f, s\tau/f'\}}{\{f\tau/f'\}(t),\theta_1\cup\{r/f\},\theta_2\cup\{s/f\}}$$
3. *(Factor-Constant)*
$$\frac{t,\theta_1\cup\{r(\delta^m K\mu(\vec{r_n})\cdot\mathbf{1})\tau/f\},\theta_2\cup\{s(\delta^{m'}K\mu(\vec{s_n})\cdot\mathbf{1})\tau/f\}}{\{h\,(K\mu(\overrightarrow{h_n})\cdot\mathbf{1})\tau/f\}(t),\theta_1\cup\{r(\delta^m\cdot\mathbf{1})/h,\vec{r_n}/\overrightarrow{h_n}\},\theta_2\cup\{s(\delta^{m'}\cdot\mathbf{1})/h,\vec{s_n}/\overrightarrow{h_n}\}}$$
4. *(Factor-Restructor)*
$$\frac{t,\theta_1\cup\{r\tau/f\},\theta_2\cup\{s\tau/f\}}{\{h\tau/f\}(t),\theta_1\cup\{r/h\},\theta_2\cup\{s/h\}}$$
oraz τ nie ma lewostronnego elementu odwrotnego.

3.3.1.3 Praktyczny algorytm uwzględniający rozmiar generalizacji.

Przedstawimy efektywny algorytm, według [7] znajdujący użyteczny podzbiór MSC(a, b), przy założeniach:

1. Termy a i b nie zawierają zmiennych wolnych.

2. Termy a i b są pierwszego rzędu, tzn. są kombinatorami typu $\boldsymbol{u} \to X$.
3. Jest pojedynczy typ bazowy ι; dane mają typ $\boldsymbol{u} \to \iota$ a funkcje mają typ $\iota \cdot \iota \cdot ... \cdot \iota \to \iota$.

Definition 3.28. *Dla pewnej funkcji* size *i porządku na jej przeciwdziedzinie* $\leqslant$

$$\mathrm{MSC}_{\max}(a,b) \;=\; \{g\colon g \in \mathrm{MSC}(a,b) \wedge (\forall g' \in \mathrm{MSC}(a,b))\, \mathrm{size}(g') \leqslant \mathrm{size}(g)\}$$

Ustalmy funkcję mierzącą size: $\mathrm{MSC}(a,b) \to \langle N, N\rangle$

$$\mathrm{size}(\langle \theta_1 \colon t \to a, \theta_2 \colon t \to b\rangle) = \langle \mathrm{size}(t), \mathrm{size}(\theta_1) + \mathrm{size}(\theta_2)\rangle$$

gdzie dla termów t

$$\mathrm{size}(t) \;=\; \begin{cases} \mathrm{size}(r) + \mathrm{size}(s) & \text{jeśli } t = r\,s \vee t = r \cdot s \\ 0 & \text{jeśli } t = p \mapsto q \vee t = \mathbf{1} \vee t \text{ jest zmienną wolną} \\ 1 & \text{w przeciwnym przypadku} \end{cases}$$

i dla podstawień θ

$$\mathrm{size}(\theta) \;=\; \sum_{x \in \mathrm{Dom}(\theta)} \mathrm{size}(\theta(x))$$

Dla tej definicji size, określmy $\leqslant$ jako

$$\langle a,b\rangle \leqslant \langle c,d\rangle \iff a < c \vee (a = c \wedge b \geqslant d)$$

Definition 3.29. *Drzewo typizowane to zorientowane drzewo etykietowane takie, że jeśli węzeł t jest symbolem funkcyjnym arności n to t ma n synów.*

Definition 3.30. *Częściową permutacją z n do m jest włożenie ϕ z S do $\{1,...,m\}$ gdzie S jest k-elementowym podzbiorem $\{1,...,n\}$ oraz $k = \min(n,m)$.*

Definition 3.31. *Typizowane drzewo s jest objęte (ang. embedded) w t,*

$$s = f(s_1,...,s_m) \;\trianglelefteq\; g(t_1,...,t_n) = t$$

jeśli zachodzi dowolny z następujących warunków:

1. $s \trianglelefteq t_i, \qquad \exists i \in \{1,...,n\}$
2. $\mathrm{label}(f) = \mathrm{label}(g), \qquad s_i \trianglelefteq t_i \qquad \forall i \in \{1,...,n\}$
3. *f jest zmienną, oraz, dla pewnej injekcji ϕ z $\{1,...,m\}$ do $\{1,...,n\}$, $s_i \trianglelefteq t_{\phi(i)}\, \forall i \in \{1,...,m\}$.*

Jeśli drzewo s jest puste, to przyjmujemy $s \trianglelefteq t$ dla dowolnego t.

Hasker w [7] błędnie definiuje punkt 3, zamiast wymagać injekcji pozwala na permutację częściową. Nie jest wtedy prawdą, że poniższe gentrees odpowiada dalej zdefiniowanemu zbiorowi $\mathcal{M}$ maksymalnych dopasowań. Rozszerzenie definicji na przypadek drzewa s pustego ma charakter techniczny.

Definition 3.32.

$$\mathrm{gentrees}(a,b) \;=\; \{t | t \trianglelefteq a, t \trianglelefteq b, (\forall t' \colon t' \trianglelefteq a \wedge t' \trianglelefteq b)\, \mathrm{size}(t') \leqslant \mathrm{size}(t)\}$$

Numerujemy kolejno węzły drzew używając przejścia wszerz z indeksem korzenia ustawionym na 1. Jeśli węzeł jest bliżej korzenia, to ma mniejszy indeks. Indeks węzła p jest oznaczony $\mathrm{index}(p)$, a węzeł (i poddrzewo, kontekst rozróżnia) t o indeksie i przez t/i. Korzeń drzewa t oznaczamy odpowiednio $t/1$. Etykieta węzła p jest oznaczona przez $\mathrm{label}(p)$. Niech $|t|$ będzie ilością węzłów w t. Definiujemy macierze N i N' rozmiaru $|t_1| \times |t_2|$:

Definition 3.33. *Niech synami t_1/i będą $\vec{p_m}$, a synami t_2/j będą $\vec{q_n}$ (dla $m, n \geqslant 0$), gdzie $\vec{u_n} = u_1, ..., u_n$. Wtedy $N_{i,j}$ jest zbiorem zawierającym sumę rodzin zbiorów*

1. $\{\{\langle i, \mathrm{index}(q)\rangle\} | q \in \vec{q_m}\}$
2. $\{\{\langle \mathrm{index}(p), j\rangle\} | p \in \vec{p_n}\}$
3. $\{\{\langle \mathrm{index}(p_k), \mathrm{index}(q_{\phi(k)})\rangle | k \in \mathrm{Dom}(\phi)\} | \phi$ *jest częściową permutacją z n do m*$\}$

oraz $N'_{i,j}$ jest zbiorem

$$N'_{i,j} = \begin{cases} \{\langle i, j\rangle, \langle \mathrm{index}(p_k), \mathrm{index}(q_k)\rangle | 1 \leqslant k \leqslant n\} & \mathrm{label}(t_1/i) = \mathrm{label}(t_2/j) \\ \varnothing & \textit{w przeciwnym przypadku} \end{cases}$$

Definiujemy teraz macierze M, M' zawierające rozmiary maksymalnie objętych drzew otrzymanych z N i N'. Oznaczmy

$$M_{\Sigma P} = \sum_{\langle p,q\rangle \in P} M_{p,q}$$

Definition 3.34.

$$\begin{aligned} M'_{i,j} &= \begin{cases} 0 & N'_{i,j} = \varnothing \\ 1 + M_{\Sigma(N'_{i,j} \setminus \{\langle i,j\rangle\})} & \textit{w przeciwnym przypadku} \end{cases} \\ M_{i,j} &= \max\{M'_{i,j}, M_{\Sigma P} | P \in N_{i,j}\} \end{aligned}$$

Otrzymujemy

$$M_{1,1} = |t| \text{ dla } t \in \mathrm{gentrees}(a, b)$$

Stosując memoizację, można efektywnie policzyć (jak w metodach programowania dynamicznego)

```
for i=n downto 1
  for j=m downto 1
    compute M'_{i,j} and M_{i,j}
```

Żeby znaleźć maksymalnie duże objęte drzewa definiujemy

Definition 3.35.

$$I_{i,j} = \{P \in N_{i,j} | M_{i,j} = M_{\Sigma P}\} \cup \{N'_{i,j} | M_{i,j} = M'_{i,j}\}$$

Definition 3.36. *Niech $P \in N_{i,j}$, wtedy*

$$\mathrm{matches}_{\langle i,j\rangle} = \bigcup_{\langle p,q\rangle \in P \setminus \{\langle i,j\rangle\}} \mathrm{matches}_{\langle p,q\rangle} \cup \begin{cases} \{\langle i, j\rangle\} & \textit{jeśli } \langle i, j\rangle \in P \\ \varnothing & \textit{w przec. przyp.} \end{cases}$$

Definicja jest niedeterministyczna ze względu na wybór P. Oznaczamy zbiór wszystkich rozwiązań przez

$$\mathcal{M} = \{\mathrm{matches}_{\langle 1,1\rangle}\}$$

3.3.2 Algorytm rekonstrukcji generalizacji z relewantnymi λ-abstrakcjami.

Generalizacja drugiego rzędu w porządku danym przez podstawienia wyraża się w języku λ-termów pierwszego rzędu: zmienne związane λ-abstrakcji służą do "sklejania" podstawień z antyinstancją. (Ogólnie, rząd generalizacji w porządku podstawieniowym jest o 1 większy niż rząd potrzebnych λ-termów.) Dopiero w porządku aplikacyjnym (gdzie podstawienia są realizowane przez β-redukcję) rząd generalizacji jest taki sam jak rząd λ-termów.

Będziemy się posługiwać λ-rachunkiem, ponieważ jest wygodniejszy dla manipulacji lokalną strukturą aplikacyjną niż algebra kombinatorów: "nitki" łączące funkcję i argument mogą być jawnie nazwane; poza tym, jest bardziej naturalny przy traktowaniu termów jako drzew. Będziemy używać postaci "uncurried", bardziej odpowiedniej w języku drugiego rzędu. λ-termy używane w generalizacji będziemy uważać za różne od wszelkich konstrukcji języka, którego termy generalizujemy; zbiór zmiennych generalizacji, czyli meta-zmiennych, oznaczmy M.

Ograniczymy się do relewantnego λ-rachunku, tzn. do λ-abstrakcji relewantnych.

Definition 3.37. *λ-abstrakcja $\lambda x_1, ..., x_n.t$ jest relewantna, jeśli $\{x_1, ..., x_n\} = \boldsymbol{F}_M(t)$, gdzie $\boldsymbol{F}_M$ oznacza wolne meta-zmienne termu.*

Opracujemy algorytm rekonstrukcji generalizacji z macierzy I. Taki algorytm nie był podany bezpośrednio w pracy Haskera. Zbiory zawarte w $I_{i,j}$ oznaczają możliwe do podjęcia w pozycji i, j dopasowania kroki:

1. Dołącz korzeń drugiego termu do konstruowanego podstawienia, cały pierwszy term zachowaj dla dalszej generalizacji.
2. Dołącz korzeń pierwszego termu do konstruowanego podstawienia, cały drugi term zachowaj dla dalszej generalizacji.
3. Dołącz korzenie obu termów do konstruowanego podstawienia, przydziel podtermy, które w dalszym przebiegu będą tworzyć argumenty konstruowanego podstawienia (zmiennej).
4. Utwórz anty-instancję z korzeni obu podtermów (o ile mają wspólną etykietę), konstruuj generalizację (anty-instancję) dla każdego argumentu.

Algorytm generalizacji $\boldsymbol{A}_s(i,j) = \mathrm{Subst}(l, r, \theta_1, \theta_2, V, H, S)$ lub $\boldsymbol{A}_s(i,j) = \mathrm{AntiInst}(g, \theta_1, \theta_2)$, gdzie i, j to pozycje (indeksy) w generalizowanych programach, l, r lewe (dające s) i prawe (dające t) aktualnie konstruowane podstawienia, θ_i wynikowe podstawienia generalizacji, $\vec{V}$ zmienne związane aktualnych podstawień generalizacji, $\vec{H}$ podtermy anty-instancji w kolejności odpowiadającej V, S pozostałe do przetworzenia fragmenty programów ("skrawki"), g skonstruowaną anty-instancję.

Przy generalizacji programów s, t, $\boldsymbol{A}_s(i, j, \theta_1, \theta_2)$ zwraca: wybierz $P \in I_{i,j}$ i w zależności od kategorii rodzin podzbiorów, do której P należy

1. $P = \{\langle i,\ j'\rangle\}$, $t/j = f(t_1,\ ...,\ t_n)$, $t/j' = t_k$. Aby skrócić sformułowania, dopuszczamy wszędzie przypadek $n = 0$.

   `jeśli` $A_s(i,j') = \mathrm{Subst}(l, r, \theta_1, \theta_2, \vec{V}, \vec{H}, S)$

   $$\begin{aligned} r' &= f(r_1, ..., r_n) \\ r_m &= x_m \texttt{ dla } 1 \leqslant m \leqslant n, m \neq k \\ r_k &= r \\ S' &= S \cup \{t_m/x_m | 1 \leqslant m \leqslant n, m \neq k\} \end{aligned}$$

   `zwróć` $\mathrm{Subst}(l, r', \theta_1, \theta_2, \vec{V}, \vec{H}, S')$

   `w p.p.` $A_s(i,j') = \mathrm{AntiInst}(g, \theta_1, \theta_2)$

   $$\begin{aligned} r' &= f(r_1, ..., r_n) \\ r_m &= x_m \texttt{ dla } 1 \leqslant m \leqslant n, m \neq k \\ r_k &= X \\ \vec{V}' &= \vec{V}.X \\ \vec{H}' &= \vec{H}.g \\ S' &= S \cup \{t_m/x_m | 1 \leqslant m \leqslant n, m \neq k\} \end{aligned}$$

   `zwróć` $\mathrm{Subst}(X, r', \theta_1, \theta_2, \vec{V}', \vec{H}', S')$

   x_m oraz X są nowymi meta-zmiennymi.

2. $P = \{\langle i', j\rangle\}, s/i = f(s_1, ..., s_n), s/i' = s_k$.

`jeśli` $A_s(i',j) = \mathrm{Subst}(l, r, \theta_1, \theta_2, \vec{V}, \vec{H}, S)$

$$\begin{aligned} l' &= f(l_1, ..., l_n) \\ l_m &= x_m \texttt{ dla } 1 \leqslant m \leqslant n, m \neq k \\ l_k &= l \\ S' &= S \cup \{t_m/x_m | 1 \leqslant m \leqslant n, m \neq k\} \end{aligned}$$

`zwróć` $\mathrm{Subst}(l', r, \theta_1, \theta_2, \vec{V}, \vec{H}, S')$

`w p.p.` $A_s(i',j) = \mathrm{AntiInst}(g, \theta_1, \theta_2)$

$$\begin{aligned} l' &= f(l_1, ..., l_n) \\ l_m &= x_m \texttt{ dla } 1 \leqslant m \leqslant n, m \neq k \\ l_k &= X \\ \vec{V}' &= \vec{V}.X \\ \vec{H}' &= \vec{H}.g \\ S' &= S \cup \{s_m/x_m | 1 \leqslant m \leqslant n, m \neq k\} \end{aligned}$$

`zwróć` $\mathrm{Subst}(l', X, \theta_1, \theta_2, \vec{V}', \vec{H}', S')$

x_m oraz X są nowymi meta-zmiennymi.

3. $P = \{\langle i_1, j_1\rangle, ..., \langle i_k, j_k\rangle\}$, $s/i = f(s_1, ..., s_n)$, $t/j = g(t_1, ..., t_{n'})$, $s/i_m = s_{\psi_m}$, $t/j_m = t_{\phi(\psi_m)}$, $1 \leqslant m \leqslant k, \mathrm{Dom}(\phi) = \{\psi_1, ..., \psi_k\}$ w porządku rosnącym.

$\vec{V}' := \vec{V}, \vec{H}' := \vec{H}$

`dla` $1 \leqslant m \leqslant k$

`jeśli` $A_s(i_m, j_m) = \mathrm{Subst}(l^m, r^m, \theta_1^m, \theta_2^m, \vec{V}^m, \vec{H}^m, S^m)$

$$\begin{aligned} l_{\psi_m} &= l^m \\ r_{\phi(\psi_m)} &= r^m \end{aligned}$$

`w p.p.` $A_s(i_m, j_m) = \mathrm{AntiInst}(g^m, \theta_1^m, \theta_2^m)$

$$\begin{aligned} l_{\psi_m} &= X_m \\ r_{\phi(\psi_m)} &= X_m \\ \vec{V}' &:= \vec{V}'.X_m \\ \vec{H}' &:= \vec{H}'.g \end{aligned}$$

`następnie`

$$\begin{aligned} S' &= S \cup \{s_m/x_m, t_{m'}/y_{m'} | 1 \leqslant m \leqslant n, 1 \leqslant m' \leqslant n', m \notin \mathrm{Dom}(\phi), m' \notin \mathrm{Ran}(\phi)\} \\ l_m &= x_m, \qquad 1 \leqslant m \leqslant n, m \notin \mathrm{Dom}(\phi) \\ r_m &= y_m, \qquad 1 \leqslant m \leqslant n', m \notin \mathrm{Ran}(\phi) \\ l' &= f(l_1, ..., l_n) \\ r' &= g(r_1, ..., r_{n'}) \\ \theta_1' &= \bigcup_{1 \leqslant m \leqslant k} \theta_1^m \\ \theta_2' &= \bigcup_{1 \leqslant m \leqslant k} \theta_2^m \end{aligned}$$

`zwróć` $\mathrm{Subst}(l', r', \theta_1', \theta_2', \vec{V}', \vec{H}', S')$

x_m, y_m oraz X_m są nowymi meta-zmiennymi.

4. $P = \{\langle i, j\rangle, \langle i_1, j_1\rangle, ..., \langle i_n, j_n\rangle\}$, $s/i = f(s_1, ..., s_n)$, $t/j = f(t_1, ..., t_n)$, $s/i_m = s_m$, $t/j_m = t_m$, $1 \leqslant m \leqslant n$.

$\theta_1' := \theta_1, \theta_2' := \theta_2$

`dla` $1 \leqslant m \leqslant n$
`jeśli` $A_s(i_m, j_m) = \mathrm{Subst}(l^m, r^m, \theta_1^m, \theta_2^m, \vec{V}^m, \vec{H}^m, S^m)$

$$\begin{aligned} S_1'(x) &= A_p(\vec{V}^m, \theta_1^m \vec{H}^m, S^m(x)) \qquad x \in \mathrm{Dom}(S^m) \\ S_2'(x) &= A_p(\vec{V}^m, \theta_2^m \vec{H}^m, S^m(x)) \qquad x \in \mathrm{Dom}(S^m) \\ \theta_1' &:= \theta_1' \cup \theta_1^m \cup \{\lambda \vec{V}^m . S_1'(l^m)/Z_m\} \\ \theta_2' &:= \theta_2' \cup \theta_2^m \cup \{\lambda \vec{V}^m . S_2'(r^m)/Z_m\} \\ g_m &= Z_m(\vec{H}^m) \end{aligned}$$

`w p.p.` $A_s(i_m, j_m) = \mathrm{AntiInst}(g^m, \theta_1^m, \theta_2^m)$

$$\begin{aligned} \theta_1' &= \theta_1' \cup \theta_1^m \\ \theta_2' &= \theta_2' \cup \theta_2^m \\ g_m &= g^m \end{aligned}$$

`zwróć` $\mathrm{AntiInst}(f(g_1, ..., g_n), \vec{\theta}_1', \vec{\theta}_2')$

Z_m są nowymi meta-zmiennymi.

5. $P = \varnothing, s/i = c, t/j = d, c \neq d$, zwróć $\mathrm{Subst}(c, d, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing)$.

Żeby wyznaczyć generalizację (anty-unifikację) programów s, t:

`wyznacz macierze` N, N', M, M', I
`jeśli` $A_s(1,1) = \mathrm{Subst}(l, r, \theta_1, \theta_2, \vec{V}, \vec{H}, S)$

$$\begin{aligned} S_1'(x) &= A_p(\vec{V}, \theta_1 \vec{H}, S(x)) \qquad x \in \mathrm{Dom}(S) \\ S_2'(x) &= A_p(\vec{V}, \theta_2 \vec{H}, S(x)) \qquad x \in \mathrm{Dom}(S) \\ \theta_1' &= \theta_1 \cup \{\lambda \vec{V} . S_1'(l)/Z\} \\ \theta_2' &= \theta_2 \cup \{\lambda \vec{V} . S_2'(r)/Z\} \\ g &= Z(\vec{H}) \end{aligned}$$

`zwróć` g, θ_1', θ_2'
`w p. p.` $A_s(1,1) = \mathrm{AntiInst}(g, \theta_1, \theta_2)$
`zwróć` g, θ_1, θ_2

Pozostaje jeszcze zadanie wyznaczenia $A_p(\vec{V}, \vec{H}, t)$. Mając już rozwiązane zadanie maksymalizacji rozmiaru anty-instancji, zajmiemy się minimalizacją rozmiaru podstawień. Ponieważ postać anty-instancji determinuje argumenty podstawień, na ich podstawie trzeba optymalnie "wykroić skrawki". Wystarczy przeglądać term t od końców (od liści), szukając najdłużej pasującego argumentu z $\vec{H}$. Przy odpowiedniej reprezentacji, sprowadza się to do porównywania stringów. Jako ważne zadanie pozostaje zbadanie złożoności obliczeniowej powstałych algorytmów i ewentualna optymalizacja algorytmu $A_p(\vec{V}, \vec{H}, t)$.

Algorytm $A_p(\vec{V}, \vec{H}, t)$:

1. Sprowadź t i elementy $\vec{H}$ do postaci odwrotnej notacji polskiej (ang. RPN): $t_r, \vec{H}_r$.
2. Znajdź string z $\vec{H}_r$ o maksymalnej długości będący odcinkiem początkowym t_r. Jeśli takiego nie ma, zakończ (zakończ wywołanie rekurencyjne). (Krok niedeterministyczny.)
3. Zastąp znaleziony odc. początkowy t_0 w t_r odpowiednią zmienną x_0 z $\vec{V}$.
4. Określ odcinek początkowy s_r taki, że t_0 jest argumentem funkcji będącej ostatnim elementem s_r (tzn. najmniejszy podterm t_r, że t_0 jest jego ścisłym podtermem). Jeśli taki s_r istnieje, to wywołaj rekurencyjnie od punktu 2 dla s_r', gdzie $s_r = t_0.s_r'$.
5. Jeśli $s_r \neq t_r$ (a jeśli s_r nie istnieje, to jeśli $t_0 \neq t_r$), wywołaj rekurencyjnie od punktu 2 dla t_r', gdzie $t_r = s_r.t_r'$ (a jeśli s_r nie istnieje, $t_r = t_0.t_r'$).
6. Poskładaj odpowiedzi, zastępujące odpowiednie podtermy t_r przez zmienne, i zwróć tak przekształcony term.

3.3.3 Generalizacja przez znajdowanie maksymalnych wspólnych podstruktur.

W tym podrozdziale pokażemy, że efektywny algorytm Haskera nie oblicza $\mathrm{MSC}_{\max}$. Podamy też algorytm obliczający MSC. Powód, dla którego efektywny algorytm Haskera nie oblicza $\mathrm{MSC}_{\max}$ jest taki sam, jak powód, dla którego zrezygnowano z kombinatorów kartezjańskich na rzecz relewantnych: żeby wykorzystać tę samą informację w różnych miejscach, stosując algorytm rekursji wzdłuż struktury termów, trzeba ją skopiować, a następnie skasować w niepotrzebnych miejscach. Algorytm Haskera wyklucza kopiowanie, żeby nie było konieczne kasowanie. Dopuszczenie kasowania prowadzi do patologicznych pseudo-generalizacji.

Niekiedy mówiąc o drzewie, ma się na myśli nie tylko graf skierowany, ale jeszcze porządek określony dla każdego węzła na przyległych do niego krawędziach. Tak też niejawnie przyjmuje Hasker w swojej definicji drzewa typizowanego.

Porządkiem sub wyznaczonym przez drzewo typizowane t nazwiemy domknięcie przechodnie relacji danej przez graf t: $\mathrm{sub}(v_1, v_2)$, jeśli węzeł v_2 należy do poddrzewa zaczepionego w v_1. Zbiór węzłów drzewa oznaczmy przez $V(t)$. Niech V będzie wyróżnioną etykietą oznaczającą zmienne. Niech predykat $\mathrm{label}_k(v) \equiv (\mathrm{label}(v) = k \vee \mathrm{label}(v) = V)$. Niech relacja

$\mathrm{argn}_n(v_1, v_2) = (n \leqslant \mathrm{outdeg}(v_1)) \wedge \mathrm{sub}(v_1, v_2) \wedge (\mathrm{label}_V(v_1) \vee$ wierzchołek v_2 należy do poddrzewa zaczepionego w n-tym argumencie węzła v_1)

(Jeśli n jest większe od arności funkcji $\mathrm{label}(v_1)$, to $\{v_2|\mathrm{argn}_n(v_1, v_2)\} = \varnothing\}$).

Definition 3.38. *Niech* $G = V(t)\backslash\{p|\mathrm{label}_V(p)\}$. *Strukturą* $\mathrm{Struct}(t)$ *wyznaczoną przez drzewo typizowane* t *nazywamy system relacyjny* $\langle G, \{\mathrm{label}_k|k$ *jest etykietą*$\} \cup \{\mathrm{sub}\} \cup \{\mathrm{argn}_n|1 \leqslant n \leqslant$ *maksymalna arność symb. funkcyjnych* $t\}, \varnothing, \varnothing\rangle$ *(bez operacji i elementów wyróżnionych; relacje są obcięte do zbiorów* G^n*).*

Definition 3.39. *Monomorfizmem systemów relacyjnych* $\boldsymbol{A}$ *i* $\boldsymbol{B}$ *nazywamy różnowartościowy silny homomorfizm, tzn. odwzorowanie* $h: A \xrightarrow{1-1} B$ *takie, że* $r^A(a_1, ..., a_n) \equiv r^B(h(a_1), ..., h(a_n))$.

Definicja systemu relacyjnego i monomorfizmu, patrz [19]. Zamysłem twierdzenia jest wskazanie, że algorytm z [7] (a właściwie mój algorytm, na bazie struktury wyznaczonej przez Haskera) zwraca generalizację, ale nie $\mathrm{MSC}_{\max}$.

Theorem 3.40. *Jeśli drzewo typizowane* s *jest objęte przez drzewo typizowane* t, $s \trianglelefteq t$, *to istnieje monomorfizm* h *ze* $\mathrm{Struct}(s)$ *do* $\mathrm{Struct}(t)$, $h[\mathrm{Struct}(s)] \leqslant \mathrm{Struct}(t)$. *Jednak jeśli nawet istnieje monomorfizm* h *ze* $\mathrm{Struct}(s)$ *do* $\mathrm{Struct}(t)$, *to ciągle może nie zachodzić* $s' \trianglelefteq t'$ *dla żadnych* s', t': $\mathrm{Struct}(s) = \mathrm{Struct}(s'), \mathrm{Struct}(t) = \mathrm{Struct}(t')$.

Przypomnijmy definicję: Typizowane drzewo s jest objęte (ang. embedded) w t,

$$s = f(s_1, ..., s_m) \quad \trianglelefteq \quad g(t_1, ..., t_n) = t$$

jeśli zachodzi dowolny z następujących warunków:

1. $s \trianglelefteq t_i, \qquad \exists i \in \{1, ..., n\}$
2. $\mathrm{label}(f) = \mathrm{label}(g), \qquad s_i \trianglelefteq t_i \qquad \forall i \in \{1, ..., n\}$
3. f jest zmienną, oraz, dla pewnej injekcji ϕ z $\{1, ..., m\}$ do $\{1, ..., n\}$, $s_i \trianglelefteq t_{\phi(i)}\, \forall i \in \{1, ..., m\}$.

Proof. Część $s \trianglelefteq t \implies h[\mathrm{Struct}(s)] \leqslant \mathrm{Struct}(t)$. Niech drzewa typizowane spełniają $s \trianglelefteq t$. Przeprowadźmy indukcję ze względu na rozmiar t.

Niech t będzie liściem. Wtedy s jest liściem, może zachodzić 2. lub 3. punkt definicji obejmowania. Jeśli f jest zmienną, to monomorfizm jest trywialny (pusty). W przeciwnym przypadku $\mathrm{label}(s/1) = \mathrm{label}(t/1)$ i wszystkie relacje się pokrywają, $h(s/1) = t/1$ jest monomorfizmem.

Niech teraz $g(t_1, ..., t_n) = t, n \geqslant 1$. Jeśli zachodzi punkt 1. definicji obejmowania, to monomorfizm mamy z zał. ind.

Jeśli zachodzi punkt 2., to z zał. ind. mamy n monomorfizmów, które składamy razem i dodajemy $h(f) = g$ otrzymując monomorfizm ze Struct(s) w Struct(t). Widać zgodność relacji label$_k$ i sub, natomiast zgodność relacji argn$_i$ wynika stąd, że sparowane są odpowiednie s_i oraz t_i.

Jeśli zachodzi punkt 3., to z zał. ind. mamy n monomorfizmów dla podsystemów Struct(s_i) w Struct($t_{\phi(i)}$). Relacje sub i argn$_l$ nie zachodzą dla żadnych par z Struct(s_i) $\times$ Struct(s_j), $i \neq j$, ani dla żadnych par z Struct(t_i) $\times$ Struct(t_j), $i \neq j$, a nośnikiem Struct(s) jest suma nośników Struct(s_i), więc składając otrzymane n monomorfizmów otrzymujemy szukany monomorfizm.

Część $s \trianglelefteq t \not\Leftarrow h[\text{Struct}(s)] \leqslant \text{Struct}(t)$. Niech będzie dany monomorfizm h ze Struct(s) w Struct(t). Przeprowadźmy próbę indukcji ze względu na liczebność nośnika Struct(t).

Niech nośnik Struct(t) będzie pusty. Wymusza to, że nośnik Struct(s) też jest pusty. Obierzmy jako s' drzewo puste, wtedy $s' \trianglelefteq t' = t$.

Jeśli nośnik Struct(s) jest pusty, obierzmy jako s' drzewo puste, wtedy $s' \trianglelefteq t' = t$.

Niech $s = f(s_1, ..., s_m), g(t_1, ..., t_n) = t$. Jeśli f nie jest zmienną oraz $h(f) = g$, obieramy punkt 2. derywacji i podderywacje obejmowania mielibyśmy z założenia indukcyjnego: ponieważ h jest zgodna z argn$_i$, więc jest homomorfizmem podsystemów Struct(s_i) w Struct(t_i). Tezę mielibyśmy dla $s' = f(s'_1, ..., s'_m), g(t'_1, ..., t'_n) = t'$.

Jeśli $h(f) \neq g$, to obieramy punkt 1. derywacji dla odpowiednich poddrzew t tak długo, aż zejdziemy do podtermu o korzeniu $t^1/1 = g^1 = h(f)$ (pamiętamy, że $h(f) \in \text{Struct}(t)$). Ponieważ h jest zgodna z sub, więc $h[\text{Struct}(s)] \subseteq \text{Struct}(t^1)$, stąd h jest monomorfizmem Struct(s) w Struct(t^1) i możemy wykorzystać rezultat dany wyżej (tzn. dla przypadku $h(f) = g$) dla uzyskania potrzebnej podderywacji: niech zwrócone drzewa będą s', $t^{1\prime}$. Tezę mielibyśmy dla s', $[t^{1\prime}/t^1]t$.

Jeśli f jest zmienną, to możemy obrać w derywacji relacji obejmowania punkt 1 lub 3. Potrzebujemy tak dobrać permutacje, aby pary: minimalna f_i należąca do nośnika Struct(s) i $h(f_i)$ się spotkały. Jednak h może łączyć minimalne f_i z dowolnie ulokowanymi nieporównywalnymi przez sub podtermami t.

Dla przykładu, niech x będzie zmienną funkcyjną:

$$\begin{aligned} s &= x(a, b, c) \\ t &= f(a, g(b, c)) \end{aligned}$$

Monomorfizm $h(a_s) = a_t, h(b_s) = b_t, h(c_s) = c_t$, ale $s \not\trianglelefteq t$.

□

Maksymalne izomorficzne podsystemy to takie, które nie są podsystemami innych izomorficznych podsystemów. Podobnie można udowodnić, że:

Theorem 3.41. *Zbiory z $\mathcal{M}$ wyznaczają izomorficzne podsystemy systemów relacyjnych* Struct(s) *i* Struct(t), *dla ustalonych drzew typizowanych s i t. Jednak nie muszą to być maksymalne podsystemy izomorficzne.*

Teraz przejdźmy do meritum:

Weźmy generalizację programów a i b. (Niech jej anty-instancja i podstawienia będą w postaci normalnej, tzn. bez restruktorów wewnątrz termów, tylko po prawej.) We wszystkich restruktorach, wyróżnijmy pewne wystąpienia zmiennych prawej strony tak, aby każdy restruktor był liniowy jeśli uwzględniać tylko te wystąpienia (eliminacja powtórzeń). Teraz dokonajmy podstawień generalizacji i wyróżnijmy te wystąpienia symboli (węzły drzewa), które pochodzą od korzenia anty-instancji, albo pochodzą od anty-instancji poprzez zmienną wyróżnioną. Można zauważyć, że każdemu wystąpieniu symbolu w anty-instancji odpowiada identyfikowalne wyróżnione wystąpienie symbolu w uzyskanych z podstawienia programach (wyróżniony węzeł drzewa). To pozwala zbudować odpowiedniość między wyróżnionymi węzłami a i b. Nazwijmy ją odpowiedniością przez uliniowienie.

Theorem 3.42. *Każda odpowiedniość przez uliniowienie dana przez generalizację* $\langle t, \theta_1, \theta_2\rangle$ *jest izomorfizmem odpowiednich podsystemów* Struct(a) *i* Struct(b). *Podsystemy te są izomorficzne z systemem relacyjnym* Struct(t).

Pamiętamy z definicji, że zmienne są usuwane z nośnika Struct(t).

Proof. Przeprowadźmy indukcję względem struktury anty-instancji.

Jeśli w korzeniu anty-instancji jest stała, to stosując zał. ind. do bezpośrednich podtermów a, b i anty-instancji (odpowiadających sobie, tak, że arg_i jest zachowana), otrzymujemy tezę, przyjmując $h(a/1) = b/1$, odpowiednio $h_1(a/1) = t/1$, $h_2(b/1) = t/1$. (Jeśli nie ma podtermów, to teza wynika natychmiast dla $h(a/1) = b/1$, $h_1(a/1) = t/1, h_2(b/1) = t/1$.)

Jeśli zaś w korzeniu anty-instancji jest zmienna, to ustalamy, jakie zmienne restruktorów są wyróżnione w podstawieniach. Następnie tworzymy trójki: argument zmiennej w korzeniu anty-instancji, podterm a uzyskany z podstawienia tego argumentu pod zmienną wyróżnioną w podstawieniu dającym a, podterm b tak samo. Stosujemy założenie indukcyjne do każdej trójki (z tymi samymi podstawieniami generalizacji). Szukany izomorfizm jest złożeniem otrzymanych izomorfizmów, wystarczy pokazać, że są zgodne. Ale jeśli dwa węzły leżą w dziedzinach różnych izomorfizmów, to leżą w rozłącznych podtermach, więc arg_i ani sub nie zachodzi między nimi; to samo dla dwóch węzłów w obrazach różnych izomorfizmów, w szczególności dla obrazów wspomnianych węzłów z dziedzin. Stąd złożenie izomorfizmów jest izomorfizmem. (Dotyczy to wszystkich trzech izomorfizmów, również tych na Struct(t), ponieważ $t/1$ nie należy do nośnika.) □

Theorem 3.43. *Jeśli pewna (dowolna) odpowiedniość przez uliniowienie wyznacza maksymalne izomorficzne podsystemy, to generalizacja jest sprowadzalna do* MSC(a, b) *przez co najwyżej łączenie zmiennych (reguła Merge).*

Proof. Pokażemy, że do tej generalizacji nie stosuje się żadna, poza (Merge), reguła spośród tych w definicji 3.27, czyli

1. (Delete)
$$\frac{t, \theta_1 \cup \{s/f\}, \theta_2 \cup \{s/f\}}{\{s/f\}(t), \theta_1, \theta_2}$$
2. (Merge) Dla obustronnie odwracalnego restruktora τ
$$\frac{t, \theta_1 \cup \{r/f, r\tau/f'\}, \theta_2 \cup \{s/f, s\tau/f'\}}{\{f\tau/f'\}(t), \theta_1 \cup \{r/f\}, \theta_2 \cup \{s/f\}}$$
3. (Factor-Constant)
$$\frac{t, \theta_1 \cup \{r(\delta^m K\mu(\vec{r_n}) \cdot \mathbf{1})\tau/f\}, \theta_2 \cup \{s(\delta^{m'} K\mu(\vec{s_n}) \cdot \mathbf{1})\tau/f\}}{\{h\,(K\mu(\overrightarrow{h_n}) \cdot \mathbf{1})\tau/f\}(t), \theta_1 \cup \{r(\delta^m \cdot \mathbf{1})/h, \vec{r_n}/\overrightarrow{h_n}\}, \theta_2 \cup \{s(\delta^{m'} \cdot \mathbf{1})/h, \vec{s_n}/\overrightarrow{h_n}\}}$$
4. (Factor-Restructor)
$$\frac{t, \theta_1 \cup \{r\tau/f\}, \theta_2 \cup \{s\tau/f\}}{\{h\tau/f\}(t), \theta_1 \cup \{r/h\}, \theta_2 \cup \{s/h\}}$$

   oraz τ nie ma lewostronnego elementu odwrotnego.

Załóżmy, że do generalizacji stosuje się reguła (Delete). Weźmy dowolną odpowiedniość przez uliniowienie. Ustalmy pewne wystąpienie zmiennej f w anty-instancji. Ponieważ podsystemy tej odpowiedniości są izomorficzne ze Struct(t), więc miejsce w porządku odpowiedniego wystąpienia s w drzewach a i b jest takie samo w obydwu podsystemach. Stąd można je rozszerzyć dodając odpowiednie wystąpienia s i zachowując własność izomorfizmu, czyli nie jest on maksymalny.

Teraz załóżmy, że do generalizacji stosuje się reguła (Merge). Jeśli przed jej zastosowaniem żadna inna reguła się nie stosuje, to po zastosowaniu (Merge) też nie będzie się stosować, bo jeśli stosuje się do f po (Merge), to stosuje się też do f przed (Merge). Stąd sprowadzając dowolny generalizator skondensowany do generalizatora MSC, można najpierw stosować pozostałe reguły, a gdy żadna się nie będzie stosować, same reguły (Merge).

Załóżmy, że do generalizacji stosuje się reguła (Factor-Constant). Ustalmy wystąpienie zmienej f, podobnie jak w przypadku (Delete). Wybierzmy pierwsze spośród m wyselekcjonowanych wystąpień stałej K z ustalonych pozycji w a, podobnie pierwsze spośród m' wystąpień w b. Ze względu na to, że stała K ma taką samą strukturę argumentów w obu podstawieniach w tych wystąpieniach, ma dokładnie te same elementy ponad ze względu na relacje sub i argn_i w obu podsystemach odpow. z a i z b izomorficznych ze $\mathrm{Struct}(t)$. Oczywiście ma też te same elementy pod (są to elementy $\mathrm{Struct}(t)$ pod f w t); stąd izomorfizm można rozszerzyć o odwzorowanie tych wystąpień K, nie jest więc maksymalny.

Załóżmy, że do generalizacji stosuje się reguła (Factor-Restructor). Reguła ta oznacza, że pewne argumenty anty-instancji zostaną powielone. Ustalmy dowolną odpowiedniość przez uliniowienie. Ustalmy jeden z argumentów, który ma być powielony w wyniku reguły. Oznacza to, że w restruktorze ma on co najmniej dwa wystąpienia po prawej. Wybierzmy po jednym niewyróżnionym wystąpieniu w każdym z podstawień. Możemy teraz rozszerzyć izomorfizm, chociażby odwzorowując korzeń podtermu a odpowiadającego argumetnowi wybranego podstawienia w tak określony węzeł b. To zachowa izomorfizm, ponieważ odwzorowane węzły nie mają niczego ponad w podsystemach do których będą dodane, a pod mają to, co "pod zmienną f". □

Zagadnienia praktyczne. Powyższe twierdzenie podpowiada, że można szukać maksymalnych izomorficznych podsystemów żeby znaleźć $\mathrm{MSC}(a, b)$; po znalezieniu izomorfizmu rekonstruujemy anty-instancję, która będzie liniowa; następnie łączymy zmienne i "kompresujemy skrawki" algorytmem A_p.

Jednak dla potrzeb programowania genetycznego, właśnie taka generalizacja, liniowa względem meta-zmiennych generalizacji i λ-abstrakcji generalizacji, wydaje się odpowiednia.

Możliwość określenia efektywnego algorytmu znajdowania maksymalnej wspólnej podstruktury w ogólnym przypadku zależy od systemu; nie może być zbyt bogaty w "niestrukturalny" sposób. Relacje mogą np. wyznaczać jakieś uporządkowanie, pozwalające na zastosowanie programowania dynamicznego, jak tutaj relacja sub.

Określenie generalizacji a następnie rekombinacji w terminologii podstruktur ma duże znaczenie dla programowania genetycznego, i szerzej dla syntezy (czy wnioskowania) indukcyjnej i analogicznej. Wiele (być może większość) skutecznych zastosowań GP wykorzystuje struktury bliskie problemowi: struktury grafowe i "wariacje na ich temat", np. zastosowania w projektowaniu obwodów elektrycznych filtrujących (sukcesy promowane przez Kozę), anten do telefonów komórkowych, stworzeń poruszających się w środowisku fizycznym (system Framsticks opracowywany na UAM w Poznaniu; GP konstruuje budowę fizyczną stworzenia i budowę jego mózgu). Obecnie wykorzystywane w tych systemach operatory rekombinacji najczęściej dokonują przypadkowej wymiany fragmentów reprezentacji. Dla przykładu, w systemie Framsticks trzeba sztucznie ograniczyć rekombinację do wymiany wag w sieci neuronowej, aby zapobiec psuciu przez rekombinację dobrze dostosowanych struktur neuronalnych.

Uwaga natury teoretycznej. Odpowiedniość między generalizacjami a maksymalnymi izomorficznymi podstrukturami (podsystemami) wynika stąd, że generalizacja to anty-instancja o r a z podstawienia. Generalizacja będąca MSC może mieć nie najbardziej specyficzną anty-instancję. Oznacza to, że maksymalne izomorficzne podsystemy $s' \sim t'$, $s' < s$, $t' < t$ mogą być parami izomorficzne z izomorficznymi podsystemami $s' \sim s''$, $t' \sim t''$, $s'' < s$, $t'' < t$, dla których istnieją $s''' \sim t'''$, $s'' < s''' \leqslant s$, $t'' < t''' \leqslant t$.

3.3.3.1 Znajdowanie maksymalnych izomorficznych podsystemów.

Najprostszym sposobem znajdowania maksymalnych izomorficznych podsystemów jest:

C := zbiór par pozycji o tych samych etykietach
$M := \varnothing$
dopóki $C \neq \varnothing$
  wybierz $(p, q) \in C$
  jeśli dla wszystkich $(p', q') \in M$
    $(\mathrm{sub}(p, p') \equiv \mathrm{sub}(q, q')) \wedge \bigwedge_{1 \leqslant i \leqslant \mathrm{ar}(p)} \mathrm{argn}_i(p, p') \equiv \mathrm{argn}_i(q, q')$
  to $M := M \cup \{(p, q)\}$
  $C := C \setminus \{(p, q)\}$

zwróć M

Ma on złożoność $O(n^4)$, gdzie n jest sumą długości generalizowanych termów: najpierw generowanych jest mniej niż n^2 par, następnie dla każdej z nich dokonywanych jest mniej niż n^2 sprawdzeń. Zaletą tego algorytmu jest, że można uwzględnić preferencje co do wybieranych par. W przypadku programowania genetycznego można np. preferować pary pozycji, dla których podejrzewa się „tożsamość genealogiczną" (pochodzenie od wspólnego przodka).

Wprowadźmy oznaczenia: parę $(p,q) \in C$ nazywamy Pareto-maksymalną w C jeśli

$$\neg(\exists(p',q') \in C)\, p' \neq p \wedge q' \neq q \wedge \mathrm{sub}(p,p') \wedge \mathrm{sub}(q,q')$$

Odpowiednio definiuje się parę Pareto-minimalną w C. Wprowadzimy teraz wydajniejsze algorytmy. Algorytm, przetwarzający pozycje od liści ku korzeniowi:

C := zbiór par pozycji o tych samych etykietach
$M := \varnothing$
dopóki $C \neq \varnothing$
  1: wybierz (p,q) Pareto-maksymalną w C
  2: $C := C \backslash \{(p',q') | \mathrm{sub}(p,p') \vee \mathrm{sub}(q,q')\}$
  3: $C := C \backslash \{(p',q') | (\exists i)\, \mathrm{argn}_i(p',p) \not\equiv \mathrm{argn}_i(q',q)\}$
  $M := M \cup \{(p,q)\}$
zwróć M

Algorytm przetwarzający pozycje od korzenia ku liściom (odpowiada klasycznym rekurencyjnym sformułowaniom generalizacji termów):

przetwórz(C) =
  $M := \varnothing$
  dopóki $C \neq \varnothing$
    1: wybierz (p,q) Pareto-minimalną w C
    dla i=1 do $\mathrm{ar}(p)$
      2: $M := M \cup$ przetwórz$(\{(p',q') \in C \,|\, (\exists i)\, \mathrm{argn}_i(p,p') \wedge \mathrm{argn}_i(q,q')\})$
    3: $C := C \backslash \{(p',q') | \mathrm{sub}(p,p') \vee \mathrm{sub}(q,q')\}$
  zwróć M

C := zbiór par pozycji o tych samych etykietach
zwróć przetwórz(C)

Algorytm od-korzenia-ku-liściom łatwo zaimplementować wydajniej – ze złożonością $O(n^2)$. Zbiór C można generować w sposób leniwy z poddrzew $s,t \colon C \subset s \times t$, ponieważ operacje na przynależności do tego zbioru to 3: usuwanie poddrzew oraz 2: przekazywanie poddrzew do dalszego przetwarzania. Wystarczy opracować mechanizm wyboru (punkt 1) nie przetwarzający par wielokrotnie. Można to zrobić przechowując przetworzone poddrzewa S, T: $S \ni s'$ (odpowiednio $T \ni t'$) oraz w pojedynczym kroku dodawać węzeł p do s' lub q do t' i szukać q (odpowiednio p) w $t' \in T$ (odpowiednio w $s' \in S$) o tej samej etykiecie; po znalezieniu, zastąpić t' w T przez poddrzewa powstałe przez usunięcie z t' węzłów od korzenia t' do węzła q (odpowiednio dla przypadku s', S, p).

Theorem 3.44. *Powyższe algorytmy generują izomorfimy maksymalnych izomorficznych podsystemów.*

Konstrukcja generalizacji ograniczonych do $\mathrm{MSC}_{\max}$, a więc nie tylko maksymalnie specyficznych, ale i największych rozmiarem, jest trudniejsza. Przy przetwarzaniu od-liści-ku-korzeniowi możemy zastosować metodę programowania dynamicznego: najlepiej dopasowane drzewa mają też najlepiej dopasowane poddrzewa. Jednak należy uwzględnić wszystkie możliwe dopasowania powyżej przetwarzanej pary. Dla zbioru C wszystkich par pozycji o takich samych etykietach:

Gdy wszystkie pary z C powyżej $(p, q) \in C$ zostały już ocenione, skonstruuj wszystkie maksymalnie liczne zbiory par Pareto-minimalnych z

$$\{(p', q') \in C \,|\, (\exists i)\, \mathrm{argn}_i(p, p') \wedge \mathrm{argn}_i(q, q')\}$$

takich, że elementy par w konstruowanym zbiorze są nieporównywalne, oceń zbiory na sumę ocen należących do nich par, oceń (p, q) na maksimum z ocen skonstruowanych zbiorów plus jeden. Skojarz z parą (p, q) zbiór, dla którego zachodzi maksimum, na potrzeby rekonstrukcji izomorfizmu.

Niestety, nawet przy wydajnej implementacji algorytm oparty na powyższej regule będzie miał dużą złożoność obliczeniową. Pozostaje mieć nadzieję, że w praktycznych przypadkach zbiory par Pareto-minimalnych z $\{(p', q') \in C \,|\, (\exists i)\, \mathrm{argn}_i(p, p') \wedge \mathrm{argn}_i(q, q')\}$ dla danych p, q nie będą bardzo liczne.

3.3.3.2 Rozszerzanie generalizacji na λ-termy z definicjami lokalnymi i rekurencyjnymi.

Anty-unifikacja z poprzednich podrozdziałłów bezpośrednio przenosi się na przypadek definicji lokalnych, ponieważ w reprezentacji grafowej wyznaczają one ciągle relację porządku. Natomiast zastosowanie do definicji rekurencyjnych wymaga dalszych badań, ale wiąże się z prowadzonymi przez innych badaniami nad indukcją funkcji rekurencyjnych.

3.3.3.3 Zgodność z systemem typów.

Opracowanie algorytmu z obsługą zmiennych związanych w języku (w przeciwieństwie do metazmiennych) na wzór potraktowania ich w pierwszym podrozdziale, oraz opracowanie rekombinacji zgodnej z systemem typów, to zadanie na bliską przyszłość.

Chapter 4
Zakończenie

4.1 Wkład pracy.

Postawione zadanie: Perspektywy programowania genetycznego w językach typizowanych – postanowiłem rozpatrzeć od strony teoretycznej, umieszczając zagadnienia GP w nowym dla nich kontekscie syntezy dedukcyjnej i indukcyjnej programów.

We wprowadzeniu przedstawiłem tematykę pracy, jej cele, oraz szerszy kontekst, w tym związki z teorią ewolucji darwinowskiej (pojęcie genu jako jednostki selekcji).

W pierwszej części pracy (rozdział 2) zajmowałem się zagadnieniami syntezy dedukcyjnej. Wskazałem na związek syntezy dedukcyjnej z inferencją typów i postawione jej wyzwanie: inferować pełną specyfikację logiczną konstruowanego programu. Opracowałem algorytmy generowania programów dla języka ML (system typów Damasa-Milnera) o charakterze teoretycznym, z pełnymi dowodami poprawności ($algorytm \subseteq język$) i pełności ($język \subseteq algorytm$). Algorytmy te uwypuklają analogię generowania termów i inferencji typów. Przedstawiłem system typów $\lambda\hat{}$ kontrolujący rekurencję (zapewniający własność stopu), razem z algorytmem generowania programów. Wskazałem na praktyczny mechanizm generowania termów: metodę rezolucji. Wskazałem na najodpowiedniejszego moim zdaniem kandydata na język dla programowania genetycznego: język HMG(X), przy czym uproszczony przez rezygnację z pełnego dopasowywania wzorca. Podałem proste sformułowanie "podatnego" na metodę rezolucji języka pozwalającego na kodowanie specyfikacji wyrażalnych mechanizmem unifikacji. Wspomniałem także o bardzo istotnym zagadnieniu maksymalizacji "code reuse" poprzez elastyczne mechanizmy wprowadzania definicji lokalnych.

W drugiej części pracy (rozdział 3) zająłem się kluczowym zagadnieniem syntezy indukcyjnej: generalizacją. Generalizacja ma szerokie zastosowania, natomiast zastosowanie jej, w postaci anty-unifikacji, jako techniki rekombinacji w programowaniu genetycznym jest nowatorskie. W usytuowaniu szeroko rozumianych algorytmów genetycznych używano (w pewnym sensie) generalizacji dla rekombinacji, w postaci rekombinacji zachowującej maksymalne wspólne schematy.

W kontekscie złożonych systemów typów zagadnienie znalezienia swego rodzaju homomorfizmu typizowań, pozwalającego wymieniać fragmenty programów, jest trudnym zadaniem. Nie wydaje się rozsądne poświęcać energii na poszukiwanie homomorfizmu typizowań dla czegoś tak przypadkowego jak rekombinacja swobodna; anty-unifikacja proponuje bardziej rozsądne pozycje dla rekombinacji. Opracowałem syntaktyczny algorytm anty-unifikacji obsługujący wiązania zmiennych (kontekst); jest on syntaktycznie mocniejszy (po prostej modyfikacji dodającej permutację kontekstu) od pozostałych przytoczonych podejść do anty-unifikacji, ale działa w języku nietypizowanym. Aby zastosować ten algorytm dla rekombinacji termów typizowanych (ale bez annotacji typami w termach) opracowałem algorytm inferencji typów dający typizację dla każdego poprawnego dziecka danej pary rodziców, określający, które dzieci są poprawnie typizowane.

Z myślą o przyszłych zastosowaniach do rekombinacji programów bogato wykorzystujących definicje lokalne, zająłem się pełną syntaktyczną unifikacją drugiego rzędu. Znakomite sformułowanie generalizacji w języku kombinatorów zawiera praca Haskera. Praca nie zawiera bezpośredniego sformułowania algorytmu znajdującego generalizacje, który dostarczyłem dla opracowanego tam praktycznego algorytmu konstrukcji anty-instancji. Pokazałem, że opracowany w tezie Haskera algorytm praktyczny nie jest tak mocny, jak się tam stwierdza: nie daje anty-instancji dla maksymalnie specyficznych generalizacji. Znalazłem obrazowe podejście do zagadnienia anty-unifikacji drugiego rzędu, które uogólnia się na dowolne systemy relacyjne: zadanie znajdowania maksymalnych izomorficznych podstruktur generalizowanych struktur. Zaproponowałem algorytm rozwiązujący to zadanie oraz przedstawiłem problem maksymalizacji rozmiaru szukanych podstruktur, odpowiadający problemowi znajdowania $\mathrm{MSC}_{\max}$ z rozdziału 6 pracy [7].

4.2 Postawione zadania.

Przeprowadzone badania wskazały kilka problemów, które trzeba bądź warto rozwiązać, nim przystąpi się do implementacji systemu programowania genetycznego opartego o systemy typów bardziej złożone niż system Curry'ego-Hindley'a (patrz [8]).

Pierwszym obszernym zadaniem jest stworzenie systemu GP optymalizującego "code reuse"; jest ono ściśle związane z wynikłym problemem (raczej technicznym), inferencji najogólniejszego typu jednocześnie z typem konkretyzującym zadany.

Drugim zadaniem jest pełna inferencja typu dla HMG(X) poprzez wykorzystanie mechanizmów inferencji typu dla rekurencji polimorficznej. Na bazie inferencji typu należy skonstruować algorytm generujący programy HMG(X).

Trzecim zadaniem jest zbadanie złożoności obliczeniowej problemu znajdowania generalizacji z $\mathrm{MSC}_{\max}$.

Czwartym zadaniem jest dokonanie dla pełnej anty-unifikacji drugiego rzędu tego, czego podrozdział 3.2 dokonuje dla anty-unifikacji pierwszego rzędu: wprowadzenie obsługi zmiennych związanych, opracowanie algorytmu inferencji typu dla rekombinantów.

Pozostaje do zbadania szeroka klasa dziedzin związanych z generowaniem termów: teorii rezolucji i programowania w logice, programowania więzów, poszukiwania dowodów i interaktywnego dowodzenia, dla spożytkowania w teorii programowania genetycznego.

Nie wolno też zapominać, że teoria jest wstępem do praktyki.

Bibliography

[1] Lee Altenberg. The evolution of evolvability in genetic programming. In *Advances in Genetic Programming*, 1994.

[2] Tobias Blickle and Lothar Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38, Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany, 1994. Max-Planck-Institut für Informatik (MPI-I-94-241).

[3] Richard Dawkins. *Samolubny gen*. Proszynski i S-ka, 1996.

[4] Patrik D'haeseleer. Context preserving crossover in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 256–261, Orlando, Florida, USA, 27-29 1994. IEEE Press.

[5] Didier Galmiche and David J. Pym. Proof-search in type-theoretic languages: an introduction. *Theoretical Computer Science*, 232(1–2):5–53, 2000.

[6] David E. Goldberg. *Algorytmy genetyczne i ich zastosowania*. Wydawnictwa Naukowo-Techniczne, Warszawa, 1995.

[7] Robert W. Hasker. *The Replay of Program Derivations*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[8] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[9] Masaami Hagiya Jianguo Lu, Masateru Harao. Generalization in $\lambda 2$. In *5th Workshop on Logic, Language, Information and Computation*, Sao Paulo, Brazil, July 1998.

[10] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[11] Xavier Leroy. Polymorphic typing of an algorithmic language. Technical report, INRIA, 1992.

[12] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, 1991.

[13] Nicolas J. Radcliffe. The algebra of genetic algorithms. Technical Report TR92-11, Edinburgh Parallel Computing Centre, University of Edinburgh, 1992.

[14] Daisuke Nagano Sachio Hirokawa. Long normal form proof search and counter-model generation. *Electronic Notes in Theoretical Computer Science*, 37:1–11, 2001.

[15] Ute Schmid. *Inductive Synthesis of Functional Programs, Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*. Springer, 2003.

[16] V. Simonet and F. Pottier. Constraint-based type inference for guarded algebraic data types. Research Report 5462, INRIA, January 2005.

[17] Hongwei Xi. Dependent types for program termination verification. In *16th Symposium on Logic in Computer Science*, pages 169–180, Florence, September 2001.

[18] Gwoing Tina Yu. *An Analysis of the Impact of Functional Programming Techniques on Genetic Programming*. PhD thesis, University College, London, Gower Street, London, WC1E 6BT, 1999.

[19] Pawel Zbierski Zofia Adamowicz. *Logika matematyczna*. Panstwowe Wydawnictwo Naukowe, Warszawa, 1991.