Wyszukiwanie programów jako droga do sztucznej inteligencji ogólnej

Trudno jest opracować odpowiednią matematyczną definicję inteligencji. Dlatego rozważamy ogólny problem wyszukiwania programów o określonych właściwościach i twierdzimy, korzystając z tezy Churcha-Turinga, że obejmuje ona nieformalne znaczenie inteligencji. Algorytm wyszukiwania programów może być również używany do optymalizacji jego własnej struktury i uczenia się w ten sposób. Tak więc opracowanie praktycznego algorytmu wyszukiwania programów jest sposobem na stworzenie AI. Aby skonstruować działający algorytm wyszukiwania programów, pokazujemy model programów i logiki, w którym specyfikacje i dowody właściwości programu można zrozumieć w sposób naturalny. Łączymy go z rozbudowanym parserem i pokazujemy, jak wydajny kod maszynowy można wygenerować dla programów w tym modelu. W ten sposób konstruujemy system, który komunikuje się w precyzyjnym języku naturalnym i w którym programowanie i rozumowanie można skutecznie zautomatyzować.

Inteligencja i poszukiwanie programów

Inteligencję zazwyczaj obserwuje się, gdy wiedza jest wykorzystywana w inteligentny i kreatywny sposób do rozwiązania problemu. Nadal jednak wydaje się, że sednem inteligencji nie jest wiedza ani konkretna metoda jej wykorzystania, ale ogólny sposób uczenia się na podstawie poprzednich doświadczeń. Nie ogranicza się to do przyjmowania nowej wiedzy, ale obejmuje również naukę nowych sposobów wykorzystania tego, co wiemy, rozszerzanie jej poprzez rozumowanie, a nawet ulepszanie metod uczenia się w celu bardziej efektywnej nauki. Opracowywanie nowych sposobów rozwiązywania problemów jest lepszym wskaźnikiem inteligencji niż rozwiązywanie oddzielnych zadań, ponieważ jest to praca twórcza, w której nie mamy dokładnego opisu tego, co robić i oczekuje się, że znajdziemy właściwą metodę, wiedząc tylko, jakie cele chcemy osiągnąć. Przedstawimy nieformalną koncepcję uczenia się nowych sposobów rozwiązywania problemów jako poszukiwanie programów, które spełniają pewne właściwości i zaprojektujemy system, aby uczynić go praktycznym. Aby wyjaśnić, dlaczego wybraliśmy tę reprezentację, musimy przeanalizować, w jaki sposób metody rozwiązywania problemów w ogóle mogą być modelowane za pomocą pojęć abstrakcyjnych i w jaki sposób można określać problemy. Używamy ogólnej reprezentacji, która sięga początków AI i informatyki wraz z pracami Gödla, Turinga i Churcha. Twierdzimy, że nieformalne pojęcie metody rozwiązywania pewnych zadań można wyrazić w terminach matematycznych jako maszynę Turinga. Aby to uzasadnić, używamy tezy Churcha-Turinga, założenia, że wszystko, co jest obliczalne, każde złożone zachowanie systemu, można obliczyć lub modelować przy użyciu tylko małego zestawu prostych abstrakcyjnych operacji. Możemy wziąć różne zestawy takich operacji, użyć albo maszyn Turinga, albo rachunku lambda, funkcji rekurencyjnych lub dowolnego innego języka programowania. Mimo to wszystkie one mają tę samą moc obliczeniową i ponad pięćdziesiąt lat po sformułowaniu tej tezy nie udało nam się znaleźć żadnego systemu fizycznego, ani klasycznego, ani kwantowego, który byłby w stanie obliczyć więcej niż prosta maszyna Turinga. Zauważmy prostą konsekwencję tezy Churcha-Turinga: o ile założymy, że ludzie są normalnymi, choć bardzo złożonymi obiektami fizycznymi, procedura, która działa w naszych mózgach, może być również zaimplementowana na maszynach Turinga, a zatem również na zwykłych komputerach z wystarczającą ilością pamięci, gdy te staną się wystarczająco szybkie. Teza Churcha i Turinga uzasadnia, że każdą nieformalnie rozumianą metodę rozwiązywania problemu można zdefiniować jako algorytm, maszynę Turinga, która przyjmuje instancję problemu jako dane wejściowe i zwraca rozwiązanie. Oczywiście, aby być uznaną za wykonalne rozwiązanie danego problemu, metoda (teraz – maszyna Turinga) musi spełniać pewne wymagania, które zależą od problemu. Na przykład, jeśli chcemy znaleźć sposób na sortowanie kart, może być wiele lepszych lub gorszych sposobów, aby to zrobić, maszyny, które biorą karty i zwracają je w postaci wymieszanej, ale każde rozwiązanie musi zwracać karty we właściwej kolejności. Użyjemy logiki naturalnej (pierwszego rzędu) z językiem

odpowiednim do opisu maszyn Turinga, aby określić takie wymagania. Należy zauważyć, że w tej logice nie tylko jesteśmy w stanie określić, co jest dobrym rozwiązaniem; możemy również zdefiniować kolejność, definiując, kiedy jedno rozwiązanie jest lepsze od drugiego. Możemy na przykład powiedzieć, że rozwiązanie A jest lepsze od rozwiązania B, jeśli sortowanie kart zajmuje mniej czasu, a można to wyrazić za pomocą definicji liczby kroków w przebiegu maszyny Turinga. Musimy również wziąć pod uwagę, że często cele do osiągnięcia lub warunki pracy nie będą bezpośrednio określone, ale mogą odnosić się do wiedzy o podobnych zdarzeniach w przeszłości. Można to również uwzględnić w naszej specyfikacji wymagań, jeśli zakodujemy wiedzę z przeszłości wewnątrz wzoru. Ponieważ zakładamy tezę Churcha-Turinga, możemy również założyć, że maszyna Turinga może zweryfikować poprawność rozwiązania, a następnie wszystkie możliwe problemy, które inteligentny agent będzie kiedykolwiek musiał rozwiązać, mogą zostać określone w logice pierwszego rzędu lub nawet jej ograniczonej odmianie. Modelowaliśmy rozwiązywanie problemów jako wyszukiwanie maszyn Turinga o określonych właściwościach. Określenie, czy taka maszyna istnieje, jest oczywiście nierozstrzygalne, a problem jest ogólnie nierozwiązywalny, ale możemy poczynić pewne dodatkowe założenia. Po pierwsze, możemy założyć, że nie chcemy tylko maszyny, ale także dowodu, że spełnia ona formułę i że taka maszyna z dowodem istnieje. Jest to realistyczne założenie w kontekście sztucznej inteligencji, ponieważ agent zazwyczaj chce rozwiązać problem, który jest rozwiązywalny, a gdy rozwiązanie zostanie znalezione, powinno być jasne, że jest ono poprawne. Gdy nie można znaleźć żadnego rozwiązania lub agent nie wie nic o tym, czy jest ono poprawne, czy nie, nawet w sensie probabilistycznym, musi w każdym razie uciec się do innych metod, których tutaj nie badamy, takich jak poproszenie innego agenta o pomoc lub próba ponownego rozwiązania problemu później. Dlatego nie będziemy rozważać przypadków, gdy problem nie jest rozwiązywalny lub nie można udowodnić, że rozwiązanie jest poprawne, ponieważ w takich przypadkach agent AI musi określić, kiedy zakończyć poszukiwania rozwiązania, korzystając z wiedzy zewnętrznej i biorąc pod uwagę inne czynniki. Zamiast tego skupimy się na stworzeniu modelu programów i algorytmu wyszukiwania programów, który zachowuje ogólność, a jednocześnie jest wystarczająco prosty i wydajny, aby można go było stosować w praktyce w przypadku określonych klas problemów. Jak wspomnieliśmy omawiając inteligencję, nie chcemy tylko procedury rozwiązywania pewnych zadań, ale chcemy, aby agent się uczył. Uczenie się w tym przypadku sprowadza się do ulepszania procedury, tak aby po rozwiązaniu pewnej liczby instancji problemu rozwiązywała ona inne podobne instancje wydajniej. Przedstawimy samodoskonalący się algorytm, który wyszukuje maszyny Turinga o określonych właściwościach. Ponadto pokażemy innowacyjny system, który łączy programowanie i rozwiązywanie problemów z przetwarzaniem języka naturalnego. Zarys. W następnej sekcji poszukamy ogólnej procedury, która po podaniu formuły logicznej szuka maszyny Turinga spełniającej ją i która optymalizuje się po każdym udanym uruchomieniu. Przedstawimy taką teoretyczną metodę opartą na technice enumeracji programów i dowodów, która była już używana przez Gödla [4] i Turinga. Powstała w ten sposób procedura ma przyjemną właściwość samodoskonalenia, podobnie jak my poprawiamy nasze umiejętności uczenia się, i jest bardzo ogólna, więc po pewnym czasie stanie się tak dobra jak każda inna taka procedura pod względem dowolnej odpowiedniej miary wydajności. Pokażemy również, jak może być używana przez agenta AI w nieznanym środowisku, aby nauczyć się podejmować udane działania. Problem, z jakim się mierzymy w przypadku takiego teoretycznego rozwiązania, polega na tym, że nie byłoby ono użyteczne w praktyce, gdyby zostało wdrożone w sposób bezpośredni. Czas potrzebny na jego poprawę do poziomu wydajności, który dałby jakiekolwiek namacalne rezultaty, byłby ogromny. Dlatego w kolejnych sekcjach przedstawimy model obliczeń i logiki programu, który łączy programowanie funkcyjne z rozumowaniem za pomocą gier. Model ten jest wystarczająco wydajny, aby wyrażać algorytmy i dowody na tym samym poziomie abstrakcji, w jakim o nich myślimy, a jednocześnie kompilować programy do kodu binarnego. Tak więc, podczas uruchamiania procedury wyszukiwania programu w tym modelu, możemy oczekiwać, że implementacja będzie działać wydajnie i, nawet jeśli

nie znajdzie wyników automatycznie, nadal będziemy mogli zrozumieć podejmowane przez nią kroki i pokierować ją do poprawnego rozwiązania. W rozdziale 3 przedstawimy model i dodatkowo podamy metodę analizy złożonych wyrażeń, które pasują do modelu. Taka analiza poprawia prezentację programów i dowodów i może być rozszerzona o obsługę podstawowego przetwarzania języka naturalnego. Użyjemy również przykładów, aby pokazać kompilację programów z tego modelu do wydajnego kodu, przechodząc przez język C. W rozdziale 4 przeanalizujemy, w jaki sposób właściwości programów opisanych w modelu mogą być formalnie udowodnione na wysokim poziomie abstrakcji. Pokażemy, w jaki sposób dowody automatyczne mogą być kierowane przez użytkownika lub przez różne heurystyki, a także w jaki sposób podprocedury wnioskowania w mniej ogólnych przypadkach mogą być uwzględnione w modelu bez utraty ogólności. Należy pamiętać, że przedstawione przez nas wyniki teoretyczne są dobrze znane i nie omawiamy ich zbyt dokładnie. Model obliczeń, metoda analizy wyrażeń i logika przedstawiona później również opierają się na dobrze znanych pomysłach, ale ich połączenie jest innowacyjne. Dlatego podajemy więcej szczegółów na ten temat i opisujemy, jak stworzyć system, który pozwala pisać w języku naturalnym programy, o których możemy rozumować półautomatycznie w logice formalnej i które można skompilować do wydajnego kodu maszynowego.

Wyniki teoretyczne

W tej sekcji przedstawiamy przegląd wyników teoretycznych dotyczących wyszukiwania programów o określonych właściwościach i korzystania z wyszukiwania programów w standardowym modelu AI. Jako nasz model obliczeń przyjmujemy maszyny Turinga, ale tutaj można zastosować dowolny inny model Turinga-zupełny. Ponadto nie podajemy wyników w pełnym zakresie, ponieważ większość z nich jest już standardową wiedzą w informatyce, a my chcemy je po prostu umieścić w kontekście AGI lub rozszerzyć, a w takich przypadkach podajemy odniesienia do artykułów, w których te rozszerzenia są dokładnie omówione. Rozpoczynamy naszą teorię od ustalenia opisu programów i wybrania obliczalnego zestawu aksjomatów, z którego wywnioskujemy właściwości programów. Później przedstawimy model programów, które uważamy za proste i bardziej praktyczne, ale rozważmy teraz maszyny Turinga zdefiniowane w teorii mnogości wraz z aksjomatami teorii mnogości sformalizowanymi przez Zermelo i Fränkla, co jest szeroko stosowaną aksjomatyzacją. Problem wyszukiwania programu można przedstawić następująco: biorąc pod uwagę wzór $\varphi(x_1, \ldots, x_n)$ w logice pierwszego rzędu na strukturze zdefiniowanej powyżej ze zmiennymi wolnymi $x_1, \ldots, x_k$ oznaczającymi maszyny Turinga, znajdź dowód $\varphi(m_1, \ldots, m_k)$ dla pewnych maszyn Turinga $m_1, \ldots, m_k$. Stwierdzimy teraz ważny pozytywny fakt, który jest prostą konsekwencją enumeralności maszyn Turinga i dowodów.

Fakt 1. Istnieje algorytm, który oblicza rozwiązanie problemu wyszukiwania programu, jeśli jakieś rozwiązanie istnieje, więc biorąc pod uwagę $\varphi(x_1, \ldots, x_k)$ oblicza $m_1, \ldots, m_k$ i dowód $\varphi(m_1, \ldots, m_k)$, zakładając, że dla pewnych maszyn taki dowód istnieje.

Dowód. Ponieważ maszyny Turinga, programy i dowody są przeliczalne i można określić algorytmicznie, czy ciąg formuł stanowi dowód danego twierdzenia, możemy użyć następującego algorytmu, aby udowodnić ten fakt:

(1) Ustaw długość na 1.

(2) Wylicz wszystkie k-krotności $m_1, \ldots, m_k$ maszyn Turinga krótsze od długości i wszystkie dowody krótsze od długości i sprawdź, czy wśród nich jest jakiś dowód, który dowodzi $\varphi(m_1, \ldots, m_k)$.

(3) Jeśli znaleziono prawidłowe maszyny i dowód, zwróć je, w przeciwnym razie zwiększ długość o jeden i wróć do punktu (2). Oczywiście ten algorytm znajdzie rozwiązanie, nawet najkrótsze, jeśli ono istnieje. W przeciwnym razie algorytm nigdy się nie zatrzyma. Będziemy oznaczać ten algorytm przez PSP0.

Wyszukiwanie programów w standardowym modelu AI

Rozważymy teraz często używany model AI, w którym agent wchodzi w interakcję ze środowiskiem. Agent jest modelowany tak, aby miał czujniki, z których zbiera dane wejściowe, i efektory, których używa do wykonywania działań. Ponadto w dowolnym momencie agent może otrzymać dodatkową informację zwrotną, która oznacza jego własne szczęście lub skwantyfikowaną ocenę, którą otrzymuje od agenta-nauczyciela. Zadaniem agenta jest maksymalizacja całkowitej oceny, jaką otrzymuje przez całe swoje życie. Aby móc konstruować dobrze działających agentów, musimy założyć coś na temat środowiska lub przynajmniej coś na temat jego probabilistycznego zachowania. Jednym z rozsądnych założeń jest to, że środowisko lub przynajmniej rozkład prawdopodobieństwa zdarzeń jest napędzany przez jakiś program (maszynę Turinga). Chcemy stworzyć agenta, który będzie zachowywał się gorzej niż optymalny agent, jeśli taki istnieje, tylko przez pewien okres czasu, a później będzie działał optymalnie. Naszkicujmy możliwą konstrukcję takiego agenta, który używa wyszukiwania programu do znajdowania reguł w zachowaniu środowiska i używa tych reguł jako predyktorów, aby znaleźć najlepsze możliwe działania w założonym środowisku. Jest to bardzo naturalny ogólny sposób działania poprzez najpierw planowanie działań zgodnie z oczekiwanym przyszłym wynikiem, a następnie wybieranie najlepszych z nich. Niech nasz agent przechowuje następujące zmienne wewnętrzne:

(i) listę powiązanych zdarzeń i działań zwaną historią, początkowo pustą;

(ii) model programu, który modeluje środowisko, początkowo dowolne krótkie;

(iii) aktora programu, który modeluje podejrzewane optymalne zachowanie agenta, początkowo dowolny trywialny program;

(iv) dwie liczby: maksymalny rozmiar i maksymalny czas, początkowo ustawione na 1.

Uważamy, że model środowiska m1 jest lepszy niż m_2, jeśli możemy udowodnić, że istnieje agent, który osiąga, używając m_1, lepszą ocenę niż jakikolwiek agent może osiągnąć, używając m_2. Agent będzie działał zgodnie z następującym algorytmem, gdy napotka nowe zdarzenie.

(1) Dołącz zdarzenie do historii.

(2) Wyszukaj dowolny program mniejszy od maksymalnego rozmiaru, który generuje historię w czasie krótszym niż maksymalny czas. Spośród takich modeli środowiskowych rozważ tylko najlepsze z nich, jak zdefiniowano powyżej, i zaktualizuj model, aby był jednym z najkrótszych z najlepszych programów

(3) Wyszukaj dowód, krótszy niż maksymalny rozmiar, który pokazuje, że jakiś program, mniejszy od maksymalnego rozmiaru i zatrzymujący się przy każdym wejściu, może osiągnąć lepszą ocenę w modelu środowiskowym niż aktor programu. W takim przypadku zaktualizuj aktora, aby był jednym z najkrótszych takich programów.

(4) Zwiększ maksymalny czas i maksymalny rozmiar o jeden.

(5) Oblicz odpowiedź aktora na zdarzenie wejściowe, dołącz odpowiedź do historii i wyprowadź ją.

Ponieważ w konstrukcji przeszukujemy wszystkie możliwe programy, możemy stwierdzić następujący prosty fakt.

Fakt 2. Jeśli maszyna Turinga może opisać zachowanie środowiska i istnieje udowodniony optymalny agent dla tego środowiska, to przedstawiony agent otrzymuje ocenę mniejszą od optymalnej tylko przez pewien okres czasu i zachowuje się optymalnie później.

Dowód. Rzeczywiście, jeśli środowisko jest programem, to po pewnym czasie działania wygeneruje wynik, który odróżnia go od każdego krótszego programu. Należy zauważyć, że zanim model będzie jasny, agent przyjmie optymistyczny i podejmie działania zgodnie z tym założeniem. Następnie, po przeanalizowaniu tego wyniku w kroku (2), zmienna model zostanie ustawiona na poprawny program środowiskowy. Gdy ta zmienna zostanie ustawiona poprawnie, agent będzie szukał w kroku (3) optymalnego agenta dla wykrytego środowiska. Ponieważ założyliśmy, że istnieje udowodniony optymalny agent, ten agent i dowód jego optymalności mają pewną długość. Gdy maksymalny rozmiar przekroczy tę długość, zmienna aktor zostanie ustawiona na optymalny program. Dlatego agent zacznie zachowywać się optymalnie po wykryciu prawidłowego środowiska i niezbędnym dowodzie. Konstrukcja agenta AIXI, oparta na podobnych pomysłach, ale rozszerzona i również określona w kontekście probabilistycznym, została przedstawiona szczegółowo i z pełnymi dowodami optymalności przez Huttera. Metodę definiowania różnych rzeczy jako najkrótszych możliwych programów opracował Levin w ramach teorii złożoności Kolmogorova , a Li i Vitanyi dają doskonały przegląd tych i podobnych metod.

Samodoskonalące się wyszukiwanie programów

Widzieliśmy, że problem wyszukiwania programów może być przydatny do budowy agenta AI, ale nadal nie wiemy, jak skutecznie wyszukiwać programy. Nie zamierzamy wyszukiwać żadnego konkretnego programu, ale nauczyć się wydajnych procedur wyszukiwania interesujących nas programów. Pokażemy, jak możemy zdefiniować, które programy są interesujące, w zależności od historii poprzednich zadań wyszukiwania, i pokażemy, jak w takim przypadku procedura wyszukiwania programów może się sama ulepszyć. Określmy zatem algorytm, który odbiera rozwiązywalne wystąpienia problemu wyszukiwania programów, rozwiązuje je i poprawia swoją wydajność w takich i podobnych wystąpieniach. Aby skonstruować tę procedurę, musimy zdefiniować, jak zdecydować, czy jeden algorytm wyszukiwania programów jest bardziej wydajny od innego w odniesieniu do historii obserwowanych wystąpień problemu, ale odłożymy dyskusję na temat takich definicji do następnej sekcji. Ponadto przedstawiony algorytm uruchamia kilka procesów jednocześnie, ale jasne jest, że taki paralelizm można symulować zarówno na maszynach Turinga, jak i na komputerach jednoprocesorowych. Najpierw algorytm inicjuje zmienną P do PSP0, algorytm wyszukiwania programów przedstawiony wcześniej, a P zostanie użyty zarówno do rozwiązania otrzymanych instancji problemów, jak i do samodoskonalenia. Inicjuje również historię do pustej sekwencji. Następnie dzieli dostępne zasoby na dwie części i uruchamia dwa procesy jednocześnie. Za każdym razem, gdy otrzymana zostanie nowa instancja problemu wyszukiwania programów, jest ona dołączana do historii. Algorytm działa w odniesieniu do miary wydajności μ, która w każdym momencie zależy od historii znanej w tym momencie. Gdy proces główny otrzyma instancję problemu, używa P do jej rozwiązania i zwraca rozwiązanie. Proces doskonalenia działa w następujący sposób:

(1) Dołącz formułę opisującą problem utworzenia algorytmu wyszukiwania programów bardziej wydajnego niż P w odniesieniu do μ do historii.

(2) Użyj P, aby znaleźć bardziej wydajny algorytm wyszukiwania programów, zgodnie z definicją zawartą we wzorze powyżej.

(3) Zaktualizuj P do nowej, bardziej wydajnej wersji.

(4) Powtórz, zaczynając od (1) z nowym P i być może rozszerzoną historią.

Można zauważyć, że ten algorytm nie tylko rozwiązuje problem wyszukiwania programów, ale także wykorzystuje swoją zdolność wyszukiwania programów do optymalizacji samego siebie. Dlatego nawet jeśli PSP_0 nie jest rozwiązaniem wydajnym, przedstawiona procedura automatycznie znajdzie lepsze

rozwiązanie, dzięki składnikowi poprawy. Założyliśmy, że relacja wydajności zależy od historii. Jeśli nie chcemy, aby ten algorytm popadał w cykle, myśląc, że jakiś algorytm wyszukiwania programów P_1 jest lepszy od P_2, a później, gdy historia się zmieni, decydując w inny sposób, musimy założyć, że definicja wydajności będzie w jakiś sposób monotoniczna. Jeśli nie jesteśmy w stanie poczynić takich założeń, przydatne może być oddzielenie historii instancji otrzymanych z zewnątrz od instancji samodoskonalenia i użycie dwóch oddzielnych algorytmów wyszukiwania programów, jednego do rozwiązywania problemów, a drugiego do poprawy wyszukiwania programów. Poniższy fakt można stwierdzić przy założeniu, że definicja wydajności jest odpowiednia, ale możliwe są również rozszerzenia na bardziej złożone sytuacje.

Fakt 3. Niech dany będzie algorytm wyszukiwania programu Q (nasz cel, algorytm efektywny) i załóżmy, że relacja efektywności jest taka, że istnieje tylko ograniczona liczba algorytmów, które są udowodnione jako bardziej efektywne niż PSP0 i mniej efektywne niż Q, w odniesieniu do wszelkich możliwych historii. Następnie dla dowolnej sekwencji otrzymanych instancji przedstawiony algorytm po pewnej liczbie kroków podstawi Q za swoją zmienną wewnętrzną P i stanie się co najmniej tak efektywny jak Q.

W ten sposób, jeśli znajdziemy jakąś rozsądną definicję wydajności, możemy po prostu uruchomić ten algorytm i czekać, aż znajdzie dobre rozwiązanie problemu wyszukiwania programu, które następnie może być użyte jako sztuczna inteligencja ogólna. Jedynym praktycznym problemem jest to, że jeśli zaczniemy od PSP0, to nawet przy najlepszych komputerach musielibyśmy czekać bardzo długo. Podobne algorytmy uczenia się i wyszukiwania programów zostały przeanalizowane przy użyciu narzędzi teorii złożoności Kolmogorova, aby uzyskać więcej informacji na ten temat. Schmidhuber szczegółowo omawia niedawno opracowaną optymalnie samodoskonalącą się maszynę, zwaną maszyną Gödla. Takie metody mogą być również istotne dla fizyki.

Omówienie definicji efektywności

Zajmijmy się teraz definicją efektywności algorytmów rozwiązujących problem wyszukiwania programu. Spróbujemy porównać takie algorytmy w odniesieniu do historii wystąpień problemu, który rozwiązują. Zwykłe definicje złożoności, nawet w sensie asymptotycznym, nie mogą być użyte w tym przypadku, ponieważ wiele wystąpień w ogóle nie jest rozwiązywalnych. Spójrzmy ponownie na problem z nieformalnej i intuicyjnej perspektywy. Po zdobyciu doświadczenia w przeszłości na klasie wystąpień, zazwyczaj powiemy, że algorytm jest wydajny, jeśli szybko rozwiązuje wystąpienia z tej klasy i inne podobne wystąpienia. Pozostały problem polega na zdefiniowaniu, które wystąpienia są podobne. Wydaje się rozsądne stwierdzenie, że dwa wystąpienia są podobne, jeśli jeden można przekształcić w drugi za pomocą kilku prostych przekształceń, na przykład zmieniając niektóre parametry lub przesuwając je w jakiś sposób. Załóżmy, że dany jest zestaw prostych przekształceń. Następnie możemy zdefiniować poziom podobieństwa między dwoma instancjami jako liczbę transformacji, które należy zastosować, aby przejść z jednej instancji do drugiej. Ze względów praktycznych moglibyśmy również założyć, że jeśli ta liczba jest większa od pewnej stałej, to instancje wcale nie są podobne. Korzystając z tego, możemy powiedzieć, że jeden algorytm wyszukiwania programu jest bardziej wydajny niż inny w odniesieniu do historii, jeśli jest szybszy we wszystkich instancjach w historii i we wszystkich podobnych instancjach. Możemy również użyć alternatywnej definicji i powiedzieć, że waga algorytmu A w odniesieniu do historii H wynosi

$$w(A,H) = \Sigma_{\{i \text{ similar to some } j \in H\}} \text{time}(A,i) \cdot 2^{\text{similarity}(i,H)},$$

gdzie podobieństwo(i,H) oznacza najmniejszy poziom podobieństwa między i a dowolnym wystąpieniem z H, a czas(A, i) oznacza czas, w którym A rozwiązuje i. Zakładamy, że suma jest brana

tylko po rozwiązywalnych wystąpieniach i. Te dwie definicje wydają się rozsądne, a pierwsza spełnia wymagania przedstawione w Faktach 3, ponieważ jest monotoniczna w odniesieniu do historii.

Jednak w praktyce druga definicja może być bardziej użyteczna, ponieważ wydaje się praktyczne zmniejszenie wydajności algorytmu w kilku przypadkach, jeśli może to prowadzić do dużych ulepszeń w innych przypadkach. Praktyczne mogłoby być również użycie innej wagi do definicji wydajności, na przykład uwzględnienie heurystyki, która mogłaby nieco pogorszyć wydajność w większości przypadków, ale znacznie ją poprawić w przypadku pewnej wąskiej klasy przypadków. Podobne problemy w kontekście wyszukiwania programów są rozważane bardziej szczegółowo przez Schmidhubera , gdzie przedstawiono więcej przykładów. Nadal wydaje się jednak, że funkcje wydajności będą musiały zostać dostrojone eksperymentalnie, gdy takie procedury zaczną być stosowane w praktyce.

Wygodny model obliczeń

Pokazaliśmy, jak skonstruować procedurę wyszukiwania programu uczącego się, ale gdybyśmy próbowali zaimplementować ją bezpośrednio przy użyciu PSP0, nie byłoby to praktyczne. Dlatego naszym celem jest teraz przedstawienie bardziej użytecznego rozwiązania. Ponieważ jest to wciąż praca w toku, a wiele szczegółów jest aktywnie dopracowywanych, należy zapoznać się ze stroną internetową w celu wprowadzenia poprawek i zapoznania się z najnowszą wersją. Wiele z tych definicji i metod jest już standardem w programowaniu funkcyjnym i przepisywaniu terminów . Powtórzmy naszą motywację: potrzebujemy modelu obliczeń, który pozwoli nam łatwo pisać programy i jednocześnie rozumować na ich temat. Aby skonstruować taki model, skupimy się tylko na dwóch podstawowych operacjach stosowanych w programowaniu, a mianowicie możliwości definiowania i stosowania funkcji oraz możliwości tworzenia złożonych typów danych. Dlatego w naszym modelu będziemy operować na obiektach, które reprezentują pewne dane, np. 1, 2, [T, F] i na funkcjach takich jak +, ·, i. Możemy składać funkcje z danymi i zapisywać wyrazy w ten sposób, na przykład 1 + 2, T i F lub (1+2) · (3 + 4). Aby zdefiniować funkcje w tym modelu, piszemy reguły mówiące, jak jeden wyraz powinien się zmieniać w inny, np. T i F → F. W takich regułach możemy używać zmiennych, na przykład możemy zapisać x + 0 → x. Należy zauważyć, że nie wszystkie wyrazy mają jakiekolwiek znaczenie, na przykład 1+T nic nie znaczy. Aby uniknąć takich wyrazów wprowadzimy typy, tak że na przykład 1 będzie miało typ int, a + będzie miało typ int, int → int, więc nie będziemy mogli zastosować go do wartości logicznej T. Model, który prezentujemy, jest znany jako przepisywanie wyrazów z typami polimorficznymi. Najpierw podamy podstawowe definicje szczegółowo, aby pokazać, że formalne rozumowanie na temat tych obiektów jest rzeczywiście wykonalne i uniknąć zamieszania później, gdy podamy przykłady mniej formalnie. Pokażemy również, jak analizować terminy z wyrażeń w języku półnaturalnym i jak generować wydajny kod maszynowy dla programów w tym modelu. W ten sposób zbudujemy system komputerowy, w którym dane wejściowe języka naturalnego mogą być używane do programowania i rozumowania bez utraty wydajności tworzonych programów. Aby zdefiniować model, potrzebujemy następujących klas, gdzie arność jest zawsze funkcją, która przypisuje liczbę naturalną każdemu elementowi rozpatrywanego zbioru:

(i) nieskończony przeliczalny zbiór zmiennych typu, oznaczany jako α, β, γ;

(ii) skończony zbiór Γ nazw typów z arnością, oznaczany jako T, R, S;

(iii) nieskończony przeliczalny zbiór V zmiennych terminów z arnością, oznaczany jako x, y, z;

(iv) skończony zbiór Θ nazw konstruktorów z arnością, oznaczany jako A, B, C;

(v) skończony zbiór Σ nazw funkcji z arnością, oznaczany jako f, g, h.

Typy. Zaczynamy od formalnych definicji typów. Mogą być one trudne do zrozumienia na początku, ale podane przez nas przykłady powinny wystarczyć do intuicyjnego zrozumienia. Zbiór typów jest definiowany indukcyjnie jako najmniejszy zbiór G taki, że:

(1) każda zmienna typu $\alpha \in G$;

(2) jeśli $T \in \Gamma$ z arnością n i $R_1, \ldots, R_n \in G$ to $T(R_1, \ldots, R_n) \in G$;

(3) dla dowolnej liczby n i typów $T_1, \ldots, T_n \in G$ i typ wyniku $R \in G$ typ funkcyjny $(T_1, \ldots, T_n \to R) \in G$.

Dopuszczamy typy funkcyjne dla n = 0, aby zachować spójną notację, ale typy R i $\emptyset \to R$ uważamy za identyczne i nie będziemy ich rozróżniać. Zdefiniujmy na przykład typy wartości boolowskich, par i list. Ustawimy:

Γ = {wartości boolowskie, listy, pary},

gdzie wartości boolowskie mają arność 0, listy arność 1, a pary arność 2. Następnie przykładowy typ E par składających się z wartości boolowskiej i listy dowolnego innego typu można przedstawić jako:

$$E = \mathrm{pairs}(\mathrm{booleans}, \mathrm{lists}(\alpha)) \in \mathcal{G}.$$

Zbiór TVar(T) zmiennych typu występujących w typie T jest również definiowany indukcyjnie przez $TVar(\alpha) = \{\alpha\}$, $TVar(T(R_1, \ldots, R_n)) = TVar(R_1) \cup \cdots \cup TVar(R_n)$ i $TVar(T_1, \ldots, T_n \to R) = TVar(T_1) \cup \cdots \cup TVar(T_n) \cup TVar(R)$, więc $TVar(E) = \{\alpha\}$.

Zwyczajowa intuicja stojąca za typami polega na tym, że traktuje się je jako drzewa etykietowane, dlatego wprowadzamy pojęcie pozycji w typach. Zbiór Λ pozycji jest zbiorem sekwencji dodatnich liczb naturalnych. Przez $\lambda \in \Lambda$ będziemy oznaczać pustą sekwencję lub pozycję górną (korzeń) w typie. Dla danego typu T i położenia p albo mówimy, że p nie istnieje w T, albo definiujemy typ na położeniu p w T (oznaczony jako $T|p$) w następujący indukcyjny sposób:

(1) λ istnieje w każdym typie i $T|\lambda = T$;

(2) $p = (n, q)$ istnieje w $S = T(R_1, \ldots, R_m)$ jeśli $m \geq n$ i q istnieje w Rn i w takim przypadku $S|_p = R_n|_q$;

(3) $p = (n, q)$ istnieje w $S = T_1, \ldots, T_m \to R$, jeśli albo $m \geq n$ i q istnieje w T_n, a w takim przypadku $S|p = T_n|_q$, albo m+ 1 = n i q istnieje w R, a wtedy $S|_p = R_q$.

Pozycja p jest powyżej pewnej pozycji q, jeśli istnieje ciąg r liczb taki, że q = (p, r). W tym przypadku mówimy również, że q jest poniżej p. Wysokość pozycji to jej długość, a wysokość typu to maksymalna wysokość pozycji istniejącej w tym typie. W przykładzie typu E widać, że pozycja 3 nie istnieje w E, ale $E|_{2,1} = \mathrm{lists}(\alpha)|_1 = \alpha$, więc E ma wysokość 2. Podstawienia i unifikatory. Czasami chcemy zmienić część typu, a wtedy mówimy, że podstawiamy typ S w typie T w pozycji p. W rezultacie otrzymujemy typ R = T [S]p, taki, że dla wszystkich pozycji q nie mniejszych niż p, które istnieją w T, zachodzi, że $R|_q = T|_q$ i $R|_p = S$. Mniej formalnie, R to po prostu T z poddrzewem na pozycji p zastąpionym przez S. Podstawienie typu S w typie T na zmienną α jest zdefiniowane jako podstawienie S w T na wszystkich pozycjach p, gdzie $T|_p = \alpha$. Podstawienie typu, zwykle oznaczane literami σ, τ, ρ, jest zbiorem par, z których każda składa się ze zmiennej typu i typu, a takie pary są oznaczane przez $\alpha \leftarrow T$. W przypadku podstawienia $\sigma = \{\alpha_1 \leftarrow T_1; \ldots; \alpha_n \leftarrow T_n\}$ oznaczymy zbiór zmiennych podstawionych przez $TVar(\sigma) = \{\alpha_1, \ldots, \alpha_n\}$ i powiemy, że stosując σ do typu T otrzymujemy typ $R = T\sigma$, który jest wynikiem podstawienia, dla każdego i, typu T_i w T dla zmiennej α_i. W niektórych algorytmach konieczne jest upewnienie się, że zmienne podstawiane są rozłączne ze zmiennymi w terminach, które podstawiamy. Jako przykład zastosujmy $\{\alpha \leftarrow \mathrm{booleans}\}$ do typu E zdefiniowanego wcześniej i otrzymamy

pairs(booleans, lists(α)){α ← booleans} = pairs(booleans, lists(booleans)). Czasami musimy zmienić nazwy zmiennych typu w typie T; albo wszystkich zmiennych, albo tylko zmiennych z danego zbioru V. Ustawmy:

$$\sigma = \{\alpha \leftarrow \alpha\underbrace{'''\ldots'''}_{k} \ : \ \alpha \in \mathrm{TVar}(T) \cap \mathcal{V}\}$$

gdzie k jest najpierw ustawione na 1 i podwaja się za każdym razem, gdy zmieniamy nazwę dowolnego typu. Następnie możemy zdefiniować zmienioną nazwę typu $\overline{T}^{\mathcal{V}} = T\sigma$ i jeśli chcemy zmienić nazwę $\overline{T}$ for $\overline{T}^{\mathrm{TVar}(T)}$ wszystkich zmiennych typu, po prostu napiszemy T zamiast . Ponieważ nazwy podstawionych zmiennych zmieniają się wraz z liczbą k przy każdej zmianie nazwy, możemy być pewni, że dowolne dwa typy R i S mają rozłączne zmienne po zmianie nazwy, $\mathrm{TVar}(\overline{R}) \cap \mathrm{TVar}(\overline{S}) = \emptyset$.

. = ∅. Możemy zastosować podstawienie typu σ do innego podstawienia typu ρ = {$\alpha_1 \leftarrow T_1; \ldots ; \alpha_n \leftarrow Tn$} i uzyskać podstawienie:

$$\rho\sigma = \{\alpha_1 \leftarrow T_1\sigma; \ldots ; \alpha_n \leftarrow T_n\sigma\}.$$

Powiemy, że podstawienie typu σ jest bardziej ogólne niż ρ, jeśli istnieje inne podstawienie τ, dla którego στ ⊆ ρ. Weźmy teraz zbiór krotek typów

$$\{(T_1, R_1, \ldots, S_1), \ldots, (T_n, R_n, \ldots, S_n)\}.$$

Każde podstawienie ρ takie, że $T_i\rho = R_i\rho = \ldots = S_i\rho$ dla każdego i nazywane jest unifikatorem tego zbioru, a dobrze znanym i ważnym faktem jest, że jeśli istnieje jakiś unifikator, to istnieje ten najogólniejszy, który oznaczymy następująco:

$$\mathrm{mgu}\{(T_1, R_1, \ldots, S_i), \ldots, (T_n, R_n, \ldots, S_n)\}$$

Najbardziej ogólny unifikator można obliczyć w czasie wielomianowym, jeśli możemy reprezentować typy w formie grafów acyklicznych, i w czasie wykładniczym, jeśli ograniczymy reprezentację do drzew, gdzie identyczne poddrzewa nie mogą być kompresowane. Na przykład łatwo zauważyć, że nie ma unifikatora dla:

$$\{(\mathrm{pairs}(\mathrm{booleans}, \alpha), \mathrm{pairs}(\mathrm{lists}(\beta), \gamma)\},$$

ale parę typów (pary(α, wartości logiczne), pary(listy(β), γ) można zunifikować i:

$$\mathrm{mgu}\{(\mathrm{pairs}(\alpha, \mathrm{booleans}), \mathrm{pairs}(\mathrm{lists}(\beta), \gamma)\} = \{\alpha \leftarrow \mathrm{lists}(\beta), \gamma \leftarrow \mathrm{booleans})\}$$

Mając dany zbiór podstawień typu {$\sigma_1, \ldots, \sigma_n$}, użyjemy również najogólniejszego unifikatora tych podstawień, τ = mgu{$\sigma_1, \ldots, \sigma_n$}, zdefiniowanego jako unifikator zbioru krotek ($T^{\alpha}_1, \ldots, T^{\alpha}_k$) wszystkich takich typów, że $\alpha \leftarrow T^{\alpha}_i \in \sigma_{li}$ dla pewnego σ_{li} , więc wszystkie typy podstawione za tę samą zmienną we wszystkich podstawieniach σ_i będą zunifikowane. Oznaczmy również, dla każdej zmiennej typu α, zunifikowany typ $T^{\alpha}_1\tau$ przez T^{α} i niech:

$$\text{subst}\{\sigma_1, \ldots, \sigma_n\} = \{\alpha \leftarrow T^{\alpha} \ : \ \alpha \in \text{TVar}(\sigma_1) \cup \ldots \cup \text{TVar}(\sigma_n)\}.$$

Terminy wpisane. Założymy teraz, że każda zmienna terminowa x ∈ V , każdy konstruktor C ∈ Θ i każda funkcja f ∈ Σ o arności n ma skojarzony typ funkcyjny

$$\text{type}(f) \ (\text{type}(C), \ \text{type}(x)) \ = \ T_1, \ldots, T_n \rightarrow R \ \in \mathcal{G}.$$

Przyjmiemy dodatkowe założenie, że dla konstruktorów typ R nie jest ani zmienną typu, ani typem funkcyjnym i ma wysokość co najwyżej jeden. Wykorzystując te informacje o typie możemy zdefiniować indukcyjnie zbiór dobrze typowanych terminów T , podając jednocześnie definicję typu terminu, type(t) ∈ G, zbiór zmiennych terminu, Var(t) ⊆ V , oraz podstawienie ρ(t) rekonstruujące zmienne typu w Var(t). Aby definicja była łatwiejsza do zrozumienia, przeanalizujemy typowanie terminu Pair(y, y) z type(y) = α, oraz konstruktora Pair ∈ Γ z typem α, β → pairs(α, β). Używamy tej nieco niestandardowej definicji z rekonstrukcją, ponieważ ułatwia ona późniejszą prezentację algorytmu parsowania. Po pierwsze, każda zmienna x ∈ V , konstruktor C ∈ Θ i symbol funkcji f ∈ Σ należą do T ze skojarzonym typem Var(C) = Var(f) = ρ(C) = ρ(f) = ∅, i

$$\text{Var}(x) = \{x\}, \ \rho(x) = \{ \ \alpha \leftarrow \alpha \text{ for } \alpha \in \text{TVar}(\text{type}(x)) \ \}$$

Tak więc w naszym przykładzie mamy ρ(y) = {α ← α}. Niech zmienna x ∈ V , konstruktor C ∈ Θ lub symbol funkcji f ∈ Σ mają arność n > 0. Najpierw zmienimy nazwę powiązanego typu S i oznaczymy $\overline{S} = S_1, \ldots, S_n \rightarrow R$. W tym momencie naszego przykładu zmieniamy nazwę typu Pair na α' , β' → pairs(α' , β'), wyrażając w ten sposób fakt, że zmienna typu α jest tylko przypadkowo taka sama w typie y i Pair. Ponadto weźmy wyrazy $t_1, \ldots, t_n \in T$ z typem(t_i) = R_i i zmieńmy nazwy wszystkich zmiennych, które nie są rekonstruowane, więc niech

$$T_i = \overline{R_i}^{\text{TVar}(R_i) \setminus \text{TVar}(\rho(t_i))}$$

W naszym przykładzie nie zmieniamy nazw niczego, ponieważ przyjmujemy $t_1 = t_2 = x$, a w x wszystkie zmienne typu są rekonstruowane. Następnie $f(t_1, \ldots, t_n)$, $C(t_1, \ldots, t_n)$ lub $x(t_1, \ldots, t_n)$ jest dobrze typizowane, jeśli istnieje

$$\rho = \text{mgu}\{\rho(t_1), \ldots, \rho(t_n)\} \text{ and } \sigma = \text{mgu}\{(T_1\rho, S_1), \ldots, (T_n\rho, S_n)\}.$$

i w takim przypadku, jeśli $\tau = \text{subst}\{\rho(t_1), \ldots, \rho(t_n)\}$ wtedy

$$f(t_1, \ldots, t_n) \in \mathcal{T}, \ \text{type}(f(t_1, \ldots, t_n)) = R\sigma$$

i $\text{Var}(f(t_1, \ldots, t_n)) = \text{Var}(t_1) \cup \cdots \cup \text{Var}(t_n)$, $\rho(f(t_1, \ldots, t_n)) = \tau\sigma$, i podobnie w przypadku konstruktora C. W naszym przykładzie unifikator ρ podstawień zmiennych dla y jest pustym podstawieniem, a σ unifikuje zarówno α , jak i β z przemianowanego typu Pair z α. Następnie, podstawiając go w typie pair, otrzymujemy typ wyniku pairs(α, α). W przypadku zmiennej musimy rozszerzyć definicje, więc mamy $\text{Var}(x(t_1, \ldots, t_n)) = \{x\} \cup \text{Var}(t_1) \cup \cdots \cup \text{Var}(t_n)$, i

$$\rho(x(t_1, \ldots, t_n)) = \tau\sigma \ \cup \ \{\alpha \leftarrow \alpha \text{ for } \alpha \in \text{TVar}(R) \setminus \text{TVar}(S_1) \cup \cdots \cup \text{TVar}(S_n)\}$$

Możemy również definiować pozycje w terminach i podstawieniach terminów w sposób analogiczny do definicji typów, i powiemy, że termin t jest bazowy, jeśli Var(t) = ∅, i że jest liniowy, jeśli żadna zmienna nie występuje w nim na więcej niż jednej pozycji. Weźmy na przykład dwa konstruktory T i F z arnością 0 i boolowskimi jako przypisanym typem. Weźmy również konstruktor Pair, który już znamy, i dwa konstruktory dla list, Nil z arnością 0 i typem lists(α), i Cons z arnością 2 i typem α, lists(α) → lists(α). W przykładach użyjemy symbolu : do oznaczenia typu danego terminu. Teraz możemy tworzyć terminy o określonych typach, na przykład:

Cons (T, Nil) : lists (booleans),

Pair (T, F) : pairs (booleans, booleans),

ale nie wolno nam używać terminów, które nie są dobrze typizowane, takich jak

```
Cons (F, T) or Cons (Pair (T, F), Cons (T, Nil))
```

W pierwszym przypadku termin z booleanów jest używany tam, gdzie oczekiwany jest termin z lists(α). W drugim przypadku nie ma poprawnego typu do utworzenia instancji zmiennej typu α w definicji typu Cons, ponieważ na liście znajdują się zarówno terminy z booleanów, jak i pary(booleanów, booleanów). Ponieważ będziemy kontynuować ten przykład, uprośćmy naszą notację. Oznaczymy Cons (x, y) jako x :: y, a Pair (x, y) jako (x, y), więc cztery powyższe terminy będą oznaczone:

```
T :: Nil , (T, F) , F :: T and (T, F) :: (T :: Nil)
```

Aby wyjaśnić potrzebę rekonstrukcji podstawień, załóżmy, że mamy trzy zmienne x, y, z ∈ V z typem(x) = listy(β), type(y) = booleans, type(z) = γ. Teraz weźmy następujący przykład dwóch wyrazów:

```
(x :: Nil, y :: Nil),
(x :: z :: Nil, y :: z :: Nil)
```

Pierwszy człon jest dobrze typizowany, ponieważ Nil jest konstruktorem, a jego typ może być zunifikowany z lists(β) w jednym miejscu i boolami w innym, ponieważ zmienna typu w type(Nil) zostanie przemianowana. Drugi człon nie jest dobrze typizowany, ponieważ nie można zrekonstruować typu zmiennej z, która nie może mieć typu boolowskiego w jednym miejscu i lists(β) w innym.

Przepisywanie. Aby zdefiniować funkcję w programie, który chcemy wykonać, wprowadźmy koncepcję reguły przepisywania, pary członów l i r, lewej i prawej strony reguły, oznaczonych jako l → r. W regule przepisywania l → r musi być zachowane, że Var(l) ⊇ Var(r), type(l) = type(r) (modulo zmiana nazw zmiennych typu), a symbol na najwyższej pozycji w l musi być nazwą funkcji. Regułę przepisania l → r można zastosować do wyrazu t w pozycji p, jeśli istnieje podstawienie σ zmiennych w l takie, że $t|_p$ = lσ. Wynikiem zastosowania reguły jest $t[r\sigma]_p$, wyraz t przepisany w pozycji p. Należy zauważyć, że istnieje tylko jeden możliwy wynik zastosowania reguły do wyrazu w danej pozycji, a warunki gwarantują, że wyraz podstawowy pozostaje podstawowy po zastosowaniu do niego reguły w dowolnej pozycji, a w takim przypadku nadal ma ten sam typ po zastosowaniu reguły. Reguła jest podstawowa, jeśli r jest podstawowa i jest liniowa, jeśli r jest liniowa. Będziemy modelować programy za pomocą systemu wyrazów i typów zdefiniowanych powyżej oraz zestawu reguł przepisania, gdzie każdy podzbiór reguł z tym samym symbolem funkcji na górnej pozycji po lewej stronie jest uporządkowany liniowo, a jak zobaczysz, pierwsze reguły w tej kolejności będą uważane za ważniejsze. Biorąc pod uwagę zbiór R reguł przepisywania, mówimy, że wyraz t przepisuje się do wyrazu s w jednym kroku, jeśli istnieje reguła l → r ∈ R, którą można zastosować do t w pewnej pozycji p, aby uzyskać s, i spełnione są dwa dodatkowe warunki. Po pierwsze, żadna reguła z R nie może być zastosowana do t w

pozycji poniżej p, co nazywa się gorliwym przepisywaniem. Robimy tylko jeden wyjątek od tej reguły: funkcja if nie musi oceniać wszystkich gałęzi. Po drugie, żadna reguła l1 → r1 ∈ R z tą samą nazwą funkcji na najwyższej pozycji po lewej stronie i przed l → r w liniowej kolejności takich reguł, nie może być zastosowana do tτ w pozycji p dla żadnego podstawienia τ, które mogłoby wygenerować konflikt. Mówimy, że τ i l_1 → r_1 generuje konflikt z l → r na t, jeśli $l_1 = t\tau\rho_1$, $l = t\tau\rho_2$ i $r_1\rho_1 = r\rho_2$. W ten sposób zabraniamy stosowania reguł, które są mniej ważne, jeśli potencjalnie można by zastosować jakąś ważniejszą regułę i uzyskać inny wynik. Jeśli t zawiera symbole funkcji, traktujemy je jako zmienne, ponieważ wynik funkcji jest nieznany, jeśli nie można jej przepisać. Wyraz t przepisuje się do s w k krokach, jeśli istnieje wyraz u, do którego t przepisuje się w k − 1 krokach, a u przepisuje się do s w jednym kroku. Powiemy również, że wyraz t jest w postaci normalnej, jeśli nie można go dalej przepisać. Wynika to z liniowej kolejności reguł z tym samym symbolem funkcji i założenia chętnego przepisywania, że jeśli dowolny wyraz t przepisuje się w dowolnej liczbie kroków do postaci normalnej, to t nie przepisuje się do żadnej innej postaci normalnej. Teraz zdefiniujemy funkcję konkatenacji z lists(α) → lists(α), która przyjmuje dwie listy i tworzy konkatenację tych list, a aby to zrobić, potrzebujemy symbolu funkcji concat ∈ Σ i trzech zmiennych x, y, z z arnością 0, type(y) = α i type(x) = type(z) = lists(α). Następnie funkcję można zdefiniować za pomocą następujących dwóch reguł przepisywania:

```
concat (Nil, x) -> x,
concat (x :: y, z) -> x :: concat (y, z)
```

Aby zobaczyć jak wykonujemy tę funkcję, połączmy T :: Nil z F :: Nil poprzez przepisanie wyrazu, co robimy w następujący sposób:

concat(T :: Nil, F :: Nil) → T :: concat(Nil,F :: Nil) → T :: F :: Nil.

Przepisywanie terminów z typami, jak przedstawiono powyżej, jest używane jako podstawa dla języków programowania wysokiego poziomu, takich jak ML i Haskell, więc przedstawiony model jest nie tylko precyzyjnym bytem matematycznym, który może być używany do logicznego rozumowania; może być używany do pisania programów, które są łatwe do odczytania i zrozumienia. Programy w innych modelach obliczeniowych nadających się do logicznego rozumowania, takich jak maszyny Turinga, nie są bezpośrednio czytelne. Z drugiej strony, dość trudno jest skonstruować elegancki rachunek logiczny dla dowolnego języka programowania imperatywnego używanego w praktyce i rozumować na jego temat. Przepisywanie terminów I/O. Jednym z problemów przedstawionego modelu jest definicja wejścia i wyjścia, ponieważ w przepisywaniach terminów nie ma efektów ubocznych. Dlatego założymy, że system typów, terminów i reguł przepisywania, z którym pracujemy, jest programem komputerowym i może reagować na zestaw poleceń, które opiszemy. Istnieją polecenia, które pozwalają nam definiować nowe typy, konstruktory, symbole funkcji i zmienne; polecenia, które dodają nowe reguły przepisywania i przepisują terminy. Omówimy je w następnej sekcji wraz z rozszerzoną notacją, której będziemy używać. Przyjrzyjmy się teraz dodatkowym poleceniom, które umożliwiają przechowywanie ciągów lub sekwencji definicji w plikach, które również mogą być ładowane. Ponadto istnieje wewnętrzna baza wiedzy, w której można przechowywać terminy dowolnego typu. Niech ścieżka będzie ciągiem reprezentującym ścieżkę do pliku lub urządzenia wirtualnego, na przykład drukarki lub wyświetlacza, ciąg i nazwa będą ciągami, typ będzie typem, a termin będzie dowolnym terminem. Następujące polecenia zapewniają operacje wejściowe i wyjściowe, przy czym ostatnie trzy służą do definiowania i manipulowania przestrzenią pamięci masowej w wewnętrznej bazie danych:

```
Load string from [path].
Store [string] in [path].
Load definitions from [path].
Store system in [path].
Define data [name] in [type].
Load from [name].
Store [term] in [name].
```

Definiując przestrzeń pamięci za pomocą polecenia define data, ustawiamy również jej typ. Ten typ musi być bardziej ogólny niż typ czegokolwiek, co przechowujemy za pomocą polecenia store, i jest typem terminu, który otrzymujemy za pomocą polecenia load dla tego magazynu. Aby zaimplementować go bez utraty poprawności typu, musimy wygenerować odpowiednie polecenia load i store dla wewnętrznej bazy danych za każdym razem, gdy używane jest polecenie define data. W przedstawionym ustawieniu możliwe jest załadowanie lub zapisanie terminów tylko po wykonaniu pełnej sekwencji kroków przepisywania; nie można zmienić stanu żadnych zmiennych podczas przepisywania, co komplikowałoby rozumowanie dotyczące programów. Ponieważ zwykle przepisujemy terminy między analizami składniowymi, konieczne jest również dodanie specjalnego sposobu obsługi poleceń load podczas konstruowania funkcji systemowych, ponieważ musimy zapobiec umieszczaniu tych poleceń w wierszu przed rzeczywistym wywołaniem. Używamy specjalnego znacznika not inline w definicji funkcji, aby zapobiec ocenie tych funkcji przed właściwym czasem. Ponieważ terminy najlepiej nadają się do symbolicznej reprezentacji danych, najlepszym sposobem tworzenia programów graficznych w systemie jest użycie grafiki wektorowej. Możemy połączyć wejście i wyjście systemu przepisywania terminów z serwerem HTML i użyć standardów internetowych, takich jak XForms i SVG. Wtedy wystarczy wygenerować terminy z odpowiednimi typami odpowiadającymi specyfikacjom standardów internetowych, a do uruchamiania programów przepisywania terminów z interfejsami graficznymi można użyć dowolnej przeglądarki. Jest jeszcze lepiej, gdy połączy się to z pomysłami interfejsu użytkownika i gdy użytkownik może mieć całą swoją pracę, zarówno tekstową, jak i graficzną, na pulpicie jednocześnie dzięki interfejsowi z możliwością powiększania.

Rozszerzona notacja programu

Pokazaliśmy model obliczeń, który odpowiada naszym potrzebom, ale nadal istnieje problem z prezentacją programów, ponieważ nawet w przypadku krótkiego przykładu, który omówiliśmy, konieczne było zdefiniowanie dodatkowej notacji dla konstruktorów list i par. Dlatego opracujemy bardziej czytelną prezentację, która będzie zbliżona do języka naturalnego. Aby to umożliwić, przypiszmy definicje składni do nazw typów, konstruktorów, symboli funkcji i zmiennych. Musimy zdefiniować element składni jako ciąg lub typ i założyć, że typ typów jest ustawiony. Sekwencje elementów składni, po których następuje typ zwracany, stanowią definicje składni. Na przykład przypiszmy do list nazw typów następującą definicję składni: „lists”, „of”, typy z typami zwracanymi, a do konstruktora Cons następującą: ?a, „:” „:”, listy (?a) z listami typów zwracanych (?a). Znaczenie definicji składni w tym konstruktorze jest takie samo, jak w notacji, którą zdefiniowaliśmy wcześniej. Pokażmy przykład, jak nazwa typu, konstruktor, symbol funkcji i zmienna są podawane z odpowiednią definicją składni. Umieścimy ciągi znaków w elementach składni w gwiazdkach * i użyjemy poleceń jak w poniższych przykładach dla definicji, gdzie zakładamy, że każda definicja typu zwraca element z typów. W definicjach możemy użyć słowa class do oznaczenia typów i słowa element dla konstruktorów.

```
Define class *lists* *of* types.
Define element *Nil* in lists of ?a.
Define element ?a *:* *:* lists of ?a in lists of ?a.
Define variable *x* in lists of ?a.
Define function *concatenate* lists of ?a *with* lists of ?a
    into lists of ?a.
```

Pokażemy, jak wyrażenia mogą być analizowane przy użyciu definicji składni, ale zauważ, że polecenie define nie tylko dodaje zdefiniowany typ, konstruktor, zmienną lub nazwę funkcji do systemu i przypisuje odpowiednie typy i definicje składni. Ponadto, gdy określony jest tag funkcjonalny, dodaje definicję składni, która pozwala na użycie konstruktorów, funkcji lub zmiennych o arności większej niż 0 jako wartości funkcjonalnych przy użyciu tylko ich nazw. W tej sytuacji i podczas działania na funkcji na poziomie meta, co omówimy później, musimy mieć jedną nazwę odpowiadającą definicji składni. Te nazwy są konstruowane automatycznie, w taki sposób, że wszystkie ciągi w definicji są zachowywane, a wszystkie typy są zmieniane na wielkie pierwsze litery nazwy typu poprzedzone ’; A jest używane dla zmiennych typu. Na przykład dla konstruktora listy zamiast Cons będziemy teraz używać nazwy ’A : : ’L, a dla definicji typu listy nazwy lists ’T . Ponieważ nazwy muszą być jednoznaczne, zawsze kończą się podkreśleniem, jeśli istnieje tylko jedna definicja składni z odpowiadającą jej nazwą, i kończą się liczbą, np. listy ’T_1, listy ’T_ 2, jeśli więcej definicji odpowiada tej samej nazwie. Pokażmy teraz, jak analizujemy ciąg wejściowy i tworzymy z niego termin. Podczas analizy używamy rozszerzonych reguł przepisywania w postaci $l_1, l_2, \ldots, l_n \rightarrow r$, które oczekują ciągu wyrazów i tylko jeśli taki ciąg zostanie napotkany, przepisują go do wynikowego terminu r. Definicje składni są łatwo kodowane jako takie rozszerzone reguły przepisywania, gdy każdy ciąg jest kodowany jako termin typu strings, a każdy typ T jest reprezentowany przez zmienną x_T z type(x_T) = T. Przyjrzyjmy się na przykład rozszerzonym regułom przepisywania dla definicji typu lists i dla konstruktora „::”:

```
‘lists’ : strings, ‘of’ : strings, t : types ->
    lists_of_’T_ (t) : types,
x : ?a, ‘:’ : strings, ‘:’ : strings, y : lists (?a) ->
    ’A_:_:_’L_ (x, y) : lists (?a).
```

Przed analizą składniową ciąg wejściowy jest dzielony na wszystkie spacje i wszystkie symbole, które nie są literami, z wyjątkiem cyfr lub liter połączonych ze słowami za pomocą lub ^. Na przykład ciąg var10 *x _11* z: ?a^2 zostałby podzielony na var, 1, 0, *, x _11, *, z, :, ?, a^2. Następnie każda część jest kodowana jako termin typu strings i stosujemy rozszerzone reguły przepisywania wyprowadzone z definicji składni do sekwencji terminów ciągu zdekodowanych z ciągu wejściowego. Można pomyśleć o algorytmie stosowania tych rozszerzonych reguł przepisywania jako rozszerzeniu analizy składniowej od dołu gramatyk bezkontekstowych. W naszym przypadku jednak reguły obejmują typy polimorficzne, a nie tylko skończony zbiór nieterminali, więc trudniej jest stosować je wszędzie, a jednocześnie w ten sposób można łatwo wyrazić więcej rzeczy. Aby zastosować zestaw rozszerzonych reguł przepisywania do sekwencji terminów $t_1, \ldots, t_n$ będziemy przechowywać dla każdej pary pozycji $1 \le i \le j \le n$ w sekwencji wszystkie wyrazy t, które można wyprowadzić między tymi dwoma pozycjami, wraz z rekonstruującymi podstawieniami ρ (t). Czasami będą one rozszerzać poprzednio zdefiniowane podstawienia ρ(t), tak że typy przepisanych zmiennych wyrazów nie zostaną zapomniane. Zbiór wszystkich wyrazów wyprowadzonych między i i j oznaczymy jako d[i, j] i obliczymy wszystkie wyrazy wyprowadzalne między wszystkimi pozycjami. Zaczniemy od $d[i, i] = t_i$ i $d[i, j] = \emptyset$ w innych przypadkach i poszukamy punktu stałego następującego rozszerzenia zbiorów d[i, j]. Aby rozszerzyć zbiory wyrazów wyprowadzalnych, możemy wziąć dowolny ciąg pozycji $1 \le i1 \le i2 \le \ldots \le i_{m+1} \le n$ i wyrazy $u_k \in d[i_k, i_{k+1}]$

(k = 1, . . .,m), dla których ρ’ = subst{ρ’(u_1), . . . , ρ’(u_m)} istnieje. Ponadto potrzebujemy rozszerzonej reguły przepisywania l_1, . . . , l_m → r takiej, aby dla pewnego podstawienia wyrazów σ zachodziło, że l_kσ = u_k dla wszystkich k. Następnie możemy rozszerzyć zbiór d[i1, im+1] przez ustawienie d[i_1, i_{m+1}] := d[i_1, i_{m+1}] ∪ {rσ} z ρ’(rσ) = ρ(rσ) ∪ ρ’ . Będziemy kontynuować ten proces, aby osiągnąć wszystkie możliwe wyprowadzalne wyrazy dla całego wyrażenia i jeśli w d[1, n] jest tylko jeden wyraz, zwrócimy go jako wynik. Jeśli w d[1, n] jest więcej wyrazów, zgłosimy błąd niejednoznaczności, a błąd braku składni wystąpi, jeśli d[1, n] = ∅. Może się również zdarzyć, że zbiory zostaną rozszerzone w nieskończoność, ale możemy zapobiec takim przypadkom, stosując subsumcję, która jest opisana poniżej, i stosując dodatkowe sprawdzanie reguł przed rozpoczęciem składni. W praktyce, gdy istnieje wiele rozszerzonych reguł przepisywania, musimy najpierw sprawdzić, jakie ciągi pojawiają się w jakiej kolejności na wejściu i użyć tylko takich reguł, dla których wszystkie ciągi po lewej stronie reguły można znaleźć w zbiorze pochodnym, a zatem możliwe jest zastosowanie reguły. W tym przypadku, jeśli wyprowadzimy nowy ciąg dla pewnej pozycji, być może będziemy musieli zwiększyć liczbę rozważanych reguł. Przeanalizujmy na przykład, jak jest parsowany wyraz x :: Nil, gdzie zmienna x ma typ lists(α). Najpierw stosujemy dwukrotnie regułę pochodzącą z definicji zmiennej x, która zmienia ciąg „x” na wyraz x z lists(α), oraz regułę dla „Nil”, aby uzyskać wyraz Nil z lists(α). Następnie stosujemy definicję składni ::, aby uzyskać jedyny możliwy wynik parsowania Cons (x, Nil). Jest jeszcze jeden problem, ponieważ gdybyśmy próbowali użyć do zaprezentowanego rozwiązania w praktyce, bardzo często otrzymywalibyśmy błędy niejednoznaczności. Pierwszą rzeczą, którą musimy zrobić, aby tego uniknąć, jest zdefiniowanie, że wyraz t z rekonstruującym podstawieniem ρ (t) podporządkowuje inny wyraz r z innym podstawieniem ρ’(r), jeśli istnieje podstawienie typu τ takie, że ρ’(t)τ ⊆ ρ’(r) i podstawienie σ takie, że tσ = r. W ten sposób określamy, kiedy jeden pośredni wynik analizy składniowej jest bardziej ogólny niż inny, i będziemy rozważać tylko najogólniejsze pośrednie wyniki, tj. między dowolnymi dwoma pozycjami będziemy rozważać tylko takie terminy i podstawienia typów dla zmiennych rekonstruowanych, które nie są podporządkowane żadnym innym, dającym się wyprowadzić między tymi dwoma pozycjami. Rozważanie takich podporządkowań jest szczególnie przydatne, gdy mamy definicje składni dla rzutów z jednego typu na drugi, ponieważ w takich przypadkach często można wyprowadzić typy bardziej i mniej ogólne. Inną cechą, którą musimy dodać, jest możliwość zdefiniowania priorytetu reguł, siły wiązania i asocjatywności definicji składni, aby poprawnie przeanalizować 1 + 2 + 3 * 4 bez używania nawiasów. Możemy włączyć to do naszego algorytmu w taki sposób, że najpierw obliczymy wszystkie wyprowadzalne terminy za pomocą drzew derywacyjnych, a następnie wybierzemy tylko najlepsze wyprowadzenia zgodnie z pewnymi regułami priorytetów. Reguły priorytetów formalizują fakt, że gdy operator ◦ jest lewostronny, to (x◦ y)◦ z ma wyższy priorytet niż x ◦ (y ◦ z), a odwrotna sytuacja jest prawdziwa w przypadku operatorów prawostronnych, a jeśli jeden wiąże się silniej niż drugi, to jest to respektowane. Aby sprawdzić łączność i wiązanie, musimy obrócić drzewo reprezentujące termin, ale jeśli analizujemy dwa niejednoznaczne terminy, to używamy priorytetów reguł. Jeżeli symbol f ma większy priorytet niż g, to f(t_1, . . . , t_n) ma większy priorytet niż g(r_1, . . . , r_m). Jeżeli symbole f i g mają ten sam priorytet, to możemy powiedzieć, że f(t_1, . . . , t_n) jest większe niż g(r_1, . . . , r_m) tylko wtedy, gdy n ≥ m, t_1 ≥ r_1, . . . , t_n ≥ r_n i pewne t_i > r_i. Aby w praktyce używać rzutowań z jednego typu na inny, musimy dodać specjalny priorytet i regułę porównania priorytetów, co oznacza następujące. Aby porównać rzutowanie cast(t), wyprowadzone przy użyciu reguły ze specjalnym priorytetem, z wyrazem s, musimy najpierw porównać t i s, a jeżeli okaże się, że t jest większe niż s, to wybrać cast(t), w przeciwnym razie wybrać wyraz s bez rzutowania. Oczywiście, nawet przy tych regułach istnieje wiele nieporównywalnych wyprowadzeń i nadal możemy otrzymać niejednoznaczności, ale w praktyce zdarza się to rzadko.

Aby uczynić język zależnym od kontekstu i bardziej elastycznym, zakładamy, że każde polecenie wysłane do systemu jest najpierw przetwarzane przez funkcję polecenia preprocess. Gdy ta funkcja nie

jest zdefiniowana, polecenie pozostaje bez przetwarzania, ale możliwość zdefiniowania i ponownego zdefiniowania tej funkcji za pomocą reguł przepisywania pozwala na rozszerzenie języka. Innym ważnym dodatkiem do prostego algorytmu parsowania przedstawionego wcześniej jest obsługa zdań złożonych. Pozwalamy użytkownikowi zdefiniować, w jaki sposób zdania mogą być tworzone, na przykład

```
sentence1 "and" sentence2
sentence2 "where" sentence1
```

a następnie, przed rozpoczęciem parsowania, dzielimy tekst wzdłuż tych reguł kompozycji i parsujemy pierwsze zdanie przed parsowaniem następnego. Za pomocą przedstawionych metod nie tylko jesteśmy w stanie parsować złożone wyrażenia, które można wygodnie wykorzystać do pisania programów w przedstawionym modelu, ale możemy również użyć go do podstawowego przetwarzania języka naturalnego. Możemy bezpośrednio tłumaczyć ramki FrameNet [1] na typy w naszym modelu i używać reguł ramowych dla określonych jednostek leksykalnych jako definicji składni. Następnie wiele zdań napisanych w języku naturalnym zostanie przeanalizowanych na terminy oznaczające ich strukturę gramatyczną, a czasami także część struktury semantycznej. Następnie możemy zdefiniować funkcje działające na tych terminach i umożliwić interakcję ze zdefiniowanym systemem w języku naturalnym. W ten sposób proste programowanie i niektóre wyszukiwania programów opisane w następnej sekcji mogą być wykonywane przez osoby niebędące programistami, co sprawia, że system jest użyteczny w praktyce. Podczas pracy na większych zestawach funkcji, typów, konstruktorów i reguł przepisywania, przydatne jest ich oznaczenie w jakiś sposób i możliwość wybrania tych, których chcemy użyć w danym momencie. Dlatego przypiszemy każdej funkcji, typowi i konstruktorowi zestaw tagów w formie klucz = wartość, gdzie zarówno klucz, jak i wartość są ciągami znaków. Jak wspomniano wcześniej, niektóre specjalne tagi mogą być również używane do generowania dodatkowych reguł lub zatrzymywania inlineingu funkcji. Możemy aktywować i dezaktywować wszystkie symbole za pomocą danego zestawu tagów ustawionych na określoną lub dowolną możliwą wartość. Ponieważ nie przedstawiliśmy jeszcze poleceń niezbędnych do dodawania reguł przepisywania, ustawiania priorytetów, usuwania definicji typu, konstruktora i funkcji, przepisywania terminów i tagów, podajmy tutaj prosty przykład.

```
Define function integers *+* integers into integers
    priority normal associativity left
    with tags [context = arithmetics, system = true].
Let 0 + 0 be 0. Compute 0 + 0.
Remove function arg + arg. Remove class lists of ?a.
Activate with tags [system = ANYTHING].
Deactivate with tags [context = arithmetics].
Close context.
```

Zwróć uwagę na polecenie close context, które usuwa wszystkie definicje zmiennych i otwiera nowy zestaw nazw zmiennych, dzięki czemu możemy używać zmiennej o nazwie x w różnych kontekstach z różnymi typami. Ponadto, usuwając funkcje i typy, musimy sprawdzić, czy nie są one używane gdzieś w innych definicjach lub w wewnętrznej bazie danych.

Kompilowanie systemów przepisywania typowego

Zaprezentowaliśmy ładny model obliczeń i pokazaliśmy, jak reprezentować programy w czytelnej formie, ale potrzebujemy jakiegoś sposobu wykonywania programów. Oczywiście moglibyśmy z łatwością napisać interpreter przepisywania terminów, ale ponieważ spodziewamy się pracy ze

złożonymi i czasochłonnymi programami, musimy mieć bardziej wydajną metodę ich wykonywania. Ogólnie rzecz biorąc, nie jest trudno skompilować reguły przepisywania terminów do języka funkcyjnego z typami polimorficznymi, ale skompilowane programy mogą być dość nieefektywne. Jedną z metod poprawy wydajności jest umożliwienie pisania definicji funkcji i typów w języku, do którego kompilujemy, a następnie, podczas kompilacji, zastępowanie tych typów i funkcji ich bardziej wydajnymi odpowiednikami pisanymi ręcznie. W tym przypadku musimy powielić naszą pracę i napisać te same programy zarówno jako reguły przepisywania terminów, jak i w języku, do którego kompilujemy, i możemy popełniać błędy w tłumaczeniu. Aby tego uniknąć, wprowadzimy kilka optymalizacji reguł przepisywania i pokażemy, jak generować wydajny kod, tak aby pisanie tych samych fragmentów kodu ręcznie dla wydajności było konieczne tylko w rzadkich przypadkach lub dla specjalnych bardzo często używanych funkcji i typów systemowych, takich jak arytmetyka lub listy. Najlepszą metodą kompilacji byłoby posiadanie formalnego modelu języka docelowego i użycie zaawansowanych algorytmów wyszukiwania programów w celu znalezienia wydajnego równoważnego kodu. Nie jest to obecnie możliwe, ponieważ nasze metody analizy programów nie są wystarczająco zaawansowane, ani też nie jest łatwa konstrukcja prostego, ale wiarygodnego modelu głównego języka programowania. Ze względów praktycznych musimy trzymać się bardziej standardowych metod kompilacji. Języki programowania funkcyjnego są obecne w środowisku akademickim od dawna, a ostatnio niektóre z powiązanych z nimi pomysłów zaczęto wykorzystywać w przemyśle. Typy polimorficzne, pod nazwą „generyków", są już zawarte w Javie i C#, a trwają szeroko zakrojone prace komercyjne nad konstruowaniem wydajnych kompilatorów dla języków polimorficznie typowanych. Istnieje również wiele badań dotyczących tych zagadnień, na przykład [23], gdzie przedstawiono optymalizacje list, ale pokażemy tylko kilka prostych optymalizacji, które można dość łatwo zaimplementować i które w praktyce działają bardzo dobrze. Skupiają się one na poprawie zarządzania pamięcią i zwiększeniu liczby funkcji rekurencyjnych ogonowych i są podobne do optymalizacji liniowej. Pomimo swojej prostoty i wydajności, takie optymalizacje nie zostały zaimplementowane w powszechnie używanych kompilatorach, być może dlatego, że w dużym stopniu polegają na braku jakichkolwiek efektów ubocznych podczas obliczeń, co jest prawdą w przypadku programów w naszym modelu, ale rzadkością w innych modelach. Najpierw pokażmy, jak przetłumaczyć reguły przepisywania na kod C, używając jako przykładu funkcji konkatenacji zdefiniowanej przez reguły:

```
concat (Nil, x) -> x,

concat (x :: y, z) -> x :: concat (y, z)
```

Omówimy tutaj podstawowe tłumaczenie na kod C, aby pokazać idee, chociaż uważamy, że bardziej praktyczne jest użycie C++ i wygenerowanie osobnych klas dla każdego typu. Szablony mogą być używane do szybkiego polimorfizmu i przeciążania, aby funkcje kopiowania i porównywania działały na wszystkich terminach, nawet na wstępnie zdefiniowanych klasach, takich jak liczby całkowite. W ten sposób łatwiej jest również obsługiwać terminy ze zmiennymi i dodawać metafunkcjonalność do wygenerowanego kodu bez utraty wydajności, ponieważ można wygenerować dowolną funkcję specjalną o podanej nazwie dla każdego typu, dodać domyślną jako szablon i pozwolić kompilatorowi C++ obsługiwać przeciążanie, gdy funkcja jest używana. Powinno być jasne, jak zaimplementować dopasowanie terminów za pomocą drzewa wyrażeń if lub case, i założymy, że istnieje rekordowy termin t zdefiniowany w C, który przechowuje identyfikator symbolu w pozycji głównej w terminie i tablicę podterminów. Zdefiniujemy funkcję konkatenacji w C tak, aby przyjmowała dodatkowy argument, wskaźnik do terminu t, w którym zostanie zapisany wynik. Zatem biorąc pod uwagę dopasowanie, funkcja konkatenacji w języku C wygląda następująco:

```
void concat (term_t arg0, term_t arg1, term_t *result)
```

```
{
if (arg0.id == Nil_ID) {
code for the first rule
}
else {
code for the second rule
}
}
```

Teraz wygenerujemy kod dla reguł, ale będziemy traktować konstruktory inaczej niż symbole funkcji. W przypadku symbolu funkcji najpierw będziemy musieli wygenerować argumenty i zapisać je w zmiennych, a następnie wywołać funkcję, podczas gdy w przypadku konstruktorów najpierw je przydzielimy, a później będziemy kontynuować generowanie kodu ze zmienionymi wskaźnikami wyników. Przyjrzyjmy się, jak kod jest generowany dla konstruktora w drugiej regule, gdzie args0.subterms[0] odpowiada zmiennej x w regule przepisywania.

```
*result = NEW_TERM (Cons_ID, 2);
code for assigning arg0.subterms[0] to (*result).subterms[0]
code for assigning the other part to (*result).subterms[1]
```

Podczas konstruowania kodu dla drugiej części najpierw przypiszemy termin y do nowej zmiennej x0, a termin z do x1, a w ostatnim wierszu wywołamy funkcję konkatenacji za pomocą

```
concat (x0, x1, & (*result).subterms[1]);
```

W ten sposób udało nam się wykorzystać wiedzę o tym, który symbol jest konstruktorem, a który symbolem funkcji, aby utworzyć wersję rekurencyjną ogonową funkcji konkatenacji. Mogliśmy również wyeliminować konieczność przydzielania pamięci dla niektórych zmiennych, ponownie wykorzystując terminy z lewej strony. Zamiast przydzielać pamięć dla wyniku i ustawiać id za pomocą makra NEW TERM, mogliśmy po prostu ustawić wynik na arg0, a następnie zmienić id i wskaźniki, gdy było to konieczne. Łatwo jest ponownie wykorzystać pamięć przydzieloną po lewej stronie i zmienne, jeśli występują one taką samą liczbę razy po lewej i prawej stronie reguły przepisywania, ale nie przedstawiliśmy tego tutaj szczegółowo dla jasności. Dodanie możliwości obsługi niektórych typów i funkcji w języku zewnętrznym, np. liczb całkowitych bezpośrednio w C, wymaga dodatkowej pracy, szczególnie w celu zapobiegania pakowaniu i rozpakowywaniu, gdzie to możliwe, ponieważ wtedy musimy wygenerować osobną wersję każdej funkcji polimorficznej dla każdego typu specjalnego. Chociaż możemy przetłumaczyć nasz system przepisywania bezpośrednio na C, najpierw przeprowadzimy kilka optymalizacji, aby zwiększyć wydajność generowanego kodu. Pierwsza, dość techniczna, ma na celu zmniejszenie wykorzystania pamięci i konieczności realokacji pamięci. Postaramy się, aby jak najwięcej reguł przepisywania było liniowych, więc spróbujemy zwrócić wszystkie nieużywane argumenty. Na przykład funkcja konkatenacji jest liniowa, ale funkcja podwójna

```
double (x) -> Pair (x,x)
```

nie jest. Nie wszystkie funkcje mogą być liniowe, ale można wykonać pewne optymalizacje. Możemy podstawić funkcje, dla których odczytywany jest argument złożony, ale tylko argument prosty jest

zwracany przez równoważne funkcje zwracające również argument złożony. Na przykład funkcja, która oblicza długość listy, powinna zostać podstawiona przez funkcję, która oblicza długość i zwraca samą listę. Aby wyjaśnić metodę, rozważ następujący przykład:

length (Nil) -> 0

length (x :: xs) -> 1 + length (xs)

argument_length (x) -> (length (x), x)

W tym przypadku funkcja argument length będzie musiała sklonować termin x zanim będzie mogła wywołać length, co z kolei zniszczy jej kopię x. Aby tego uniknąć, moglibyśmy zoptymalizować funkcje i sprawić, aby length zwracał również argument, który przyjmuje, więc stałby się równoważny argument length. Aby to zdefiniować, potrzebujemy nowej funkcji increment append, która będzie działać na elemencie i parze i będzie robić to samo, co druga reguła dla length, ale akumulować nieużywaną listę.

increment_append (x, (n, xs)) -> (1 + n, x :: xs)

length (Nil) -> (0, Nil)

length (x :: xs) -> increment_append (x, length (xs))

W ten sposób jesteśmy w stanie poprawić efektywność alokacji pamięci i wprowadzić dodatkowe usprawnienia, aby zwiększyć możliwość ponownego wykorzystania konstruktorów, co może jeszcze bardziej zoptymalizować kod.

Jest jeszcze jedna ważna i bardziej semantyczna optymalizacja, którą możemy wykonać. W naszym modelu obliczeń terminy są konstruowane z dobrze zdefiniowanych typów, więc jeśli argument funkcji ma typ niezmienny, możemy rozwinąć definicję funkcji, podstawiając wszystkie możliwe konstruktory tego typu za argument. Na przykład w definicji funkcji łączącej listy mieliśmy argument y z lists(α) w regule concat (x :: y, z) → x :: concat (y, z). Ponieważ lista, zgodnie z definicją naszych konstruktorów listy, jest albo pustą listą, albo jest konstruowana z elementu i listy, moglibyśmy podstawić te dwie możliwości i uzyskać dwie nowe reguły:

concat (x :: Nil, z) -> x :: concat (Nil, z),

concat (x :: (y :: ys), z) -> x :: concat (y :: ys, z).

Teraz prawą stronę tych reguł można symbolicznie zredukować i uzyskać nową definicję konkatenacji składającą się z następujących trzech reguł:

concat (Nil, x) -> x,

concat (x :: Nil, z) -> x :: z,

concat (x :: (y :: ys), z) -> x :: y :: concat (ys, z).

Należy pamiętać, że przy tej nowej definicji funkcja konkatenacji będzie wywoływana na długich listach tylko połowę razy, co miałoby miejsce przy starej definicji. Ceną jest to, że musimy wykonać większe dopasowanie, aby sprawdzić wszystkie trzy reguły, ale możemy wygenerować optymalne drzewa if dla wzorców, a kompilator na niższym poziomie, w naszym przypadku kompilator C, zazwyczaj może je zoptymalizować znacznie lepiej niż nadmierne wywołania funkcji. Ponadto, jeśli są wywoływane pewne pomocnicze funkcje nierekurencyjne, te wywołania mogą być czasami całkowicie usunięte w ten sposób, a wywołania funkcji dla określonych klas argumentów mogą być również zoptymalizowane.

Gdy definicje są rozwijane, możliwe jest, że niektóre wywołania funkcji wystąpią wiele razy. Jeśli reprezentujemy terminy jako skierowane grafy acykliczne (DAG), które nie mają izomorficznych pod-DAG, które nie są identyczne, wówczas takie wielokrotne wystąpienia zostaną wykryte i będzie można je zredukować do jednego wywołania funkcji. Ponadto, jeśli jedna funkcja wywołuje dwie funkcje w różnych podterminach, wówczas możemy wykonywać te funkcje w oddzielnych wątkach. Zwiększona liczba reguł uzyskana dzięki optymalizacji opisanej powyżej może zamortyzować koszt tworzenia nowych wątków. Możliwość automatycznego uczynienia programu współbieżnym, co nie jest praktyczne w przypadku programów imperatywnych, które muszą aktualizować globalny stan pamięci, jest bardzo ważna dla wydajności, ponieważ systemy komputerowe stają się coraz bardziej równoległe. Takie proste redukcje mogą czasami przyspieszyć wykonywanie o duży współczynnik, a duża liczba programów funkcjonalnych jest podatna na takie optymalizacje.

Rozumowanie za pomocą gier

Tworzenie i rozumienie dowodów jest złożonym zadaniem, a aby dogłębnie zrozumieć ten proces i spróbować zrobić to automatycznie, musimy zbudować pewien model dowodów, o którym będziemy myśleć. W logice matematycznej dowody przedstawiano jako sekwencje stwierdzeń, w których jedno stwierdzenie wynika z drugiego. W takim modelu łatwo sprawdzić, czy coś jest poprawnym dowodem, ale nawet w szkole można zauważyć, że bardzo trudno jest znaleźć dowód czegokolwiek. Dlatego rozważymy inną, bardziej intuicyjną reprezentację, w której dowody są modelowane przez gry między dwoma graczami: Eloise, mającą na celu udowodnienie żądanej własności, i Abelardem, który chce ją sfalsyfikować. Własność jest udowodniona, jeśli Eloise ma wygrywającą strategię w grze, tj. jeśli Abelard przegrywa bez względu na to, jak gra. Takie gry istnieją dla wielu logik i można znaleźć przegląd powiązanych wyników. W grach logicznych, kiedykolwiek widzimy kwantyfikator egzystencjalny lub alternatywę w rozważanym wzorze, Eloise porusza się i wybiera element struktury, aby podstawić zmienną ograniczoną przez kwantyfikator lub jeden składnik alternatywy. Odwrotnie, kiedykolwiek widzimy kwantyfikator uniwersalny lub koniunkcję, Abelard porusza się i wybiera element lub jeden składnik koniunkcji. Przyjrzyjmy się prostemu przykładowi i udowodnijmy własność, że istnieje liczba mniejsza od 3 i istnieje liczba mniejsza od 2. Liczby naturalne są naszą strukturą w tym przypadku, a ta własność jest koniunkcją dwóch stwierdzeń:

(1) istnieje liczba mniejsza od 3,

(2) istnieje liczba mniejsza od 2.

Ponieważ mamy koniunkcję, pierwszy ruch należy do Abelarda i wybiera on (1) lub (2). Następnie kolej na ruch Eloise, ponieważ w obu wzorach na najwyższej pozycji znajduje się kwantyfikator egzystencjalny. W pierwszym przypadku może wybrać liczbę 2 i wygrać, a w drugim przypadku może wybrać liczbę 0 i wygrać. Dlatego Eloise ma wygrywającą strategię, którą można opisać następująco: jeśli Abelard wybierze opcję (1), wybierz liczbę 2, a jeśli wybierze opcję (2), wybierz liczbę 0. Zauważ, że gdyby teza w drugim przypadku zakładała, że istnieje liczba mniejsza od 0, nie byłoby żadnej wygrywającej strategii dla Eloise, ponieważ nie byłaby w stanie wybrać takiej liczby, a Abelard wygrałby. Powinno być również jasne, że gdybyśmy chcieli dodać prostą indukcję do tej gry, moglibyśmy pozwolić graczom podstawić zmienną kwantyfikowaną x tylko przez 0 lub x+1. Jeśli spróbujemy zrobić to z większą liczbą kwantyfikatorów, pojawią się problemy, gdy będziemy chcieli najpierw wywołać na jednej zmiennej kwantyfikowanej uniwersalnie, a następnie na jednej zmiennej kwantyfikowanej egzystencjalnie. Aby rozwiązać takie problemy i uchwycić całą moc rozumowania indukcyjnego bez utraty kontroli nad skończonością, musimy zdefiniować na nowo gry, których używamy i dodać liczbę naturalną do każdej pozycji, oznaczającą poziom widoczności w tej pozycji. Następnie dla każdego poziomu widoczności musimy zdefiniować zbiór możliwych działań i dla każdej

pozycji na tym poziomie musimy przypisać każdej akcji dokładnie jedną wychodzącą krawędź na wykresie gry, a teraz gracze nie będą po prostu wybierać ruchów, ale będą wybierać działania. W ten sposób stan gry jest słowem nad alfabetem możliwych działań, ale gdy gracz jest na poziomie widoczności i, damy mu tylko niekompletne informacje o bieżącej grze – tylko litery, które pochodzą z poziomów widoczności niższych lub równych i. Aby powiedzieć, że gracz wygrywa taką grę z częściową informacją, nie możemy po prostu przedstawić zwycięskich strategii, ale musimy je podawać krok po kroku przez poziomy widoczności. Dlatego wymagamy, aby wygrywający gracz najpierw podał swoją strategię na pierwszy poziom widoczności, następnie przeciwnik odpowiedział strategią na pierwszy poziom, następnie pierwszy gracz podał swoją strategię na drugi poziom i tak dalej aż do ostatniego poziomu. Gry z poziomami widoczności mogą następnie ostatecznie użyć warunku wygranej parzystości lub Mullera i przechwycić sprawdzanie modelu na strukturach (drzewo, ω)-automatycznych lub innego rozumowania. Przez inne rozumowanie rozumiemy tutaj zwłaszcza rozszerzenia gry o reguły rozumowania składniowego, w tym generalizację lub specyficzne reguły eliminacji kwantyfikatorów. Należy zauważyć, że używając kwantyfikatorów egzystencjalnych i reprezentując funkcje za pomocą reguł przepisywania możemy użyć tego do wyszukiwania programów. Jednak chociaż w przypadku gier z poziomami widoczności może być nietrywialnie trudno określić zwycięzcę, zawsze można wygrać takie gry, używając strategii ze skończoną pamięcią, a zwycięzca jest zawsze określany. Zanim przedstawimy rozszerzoną grę dla terminów ogólnych z dodatkowymi możliwymi ruchami, pokażmy, jak logikę można zaimplementować w omawianym systemie przepisywania. Logika w systemie. Aby umożliwić zastosowanie logicznego rozumowania w systemie przepisywania, musimy zdefiniować typ formuł logicznych, na podstawie których będziemy prowadzić rozumowanie, a także typ terminów, aby logika mogła być reprezentowana za pomocą metareguł.

Wzory w naszym systemie są zdefiniowane względem typu T wyrazów bazowych w równościach w następujący sposób:

(1) jeśli t i r są wyrazami typu T, to t = r jest formułą T;

(2) jeśli φ i ψ są formułami, to ¬φ, φ ∧ ψ, φ ∨ ψ, φ → ψ są również formułami; (3) jeśli φ jest formułą, a s jest ciągiem znaków, który ma być nazwą zmiennej w φ, to również ∀s φ i ∃s φ są formułami.

Możemy również zdefiniować zbiór zmiennych wolnych wyrazów w formule przez Var(t = r) = Var(t) ∪ Var(r), Var(φ ∧ ψ) = Var(φ ∨ ψ) = Var(φ) ∪ Var(ψ) i Var(∀s φ) = Var(∃s φ) = Var(φ)\{s}, i odróżniamy je od zmiennych związanych, które pojawiają się za pomocą kwantyfikatorów ∃ i ∀. Jest to standardowy typ formuł, których będziemy używać i dla których zdefiniujemy reguły wnioskowania, ale można również zdefiniować inne logiki za pomocą metod opisanych poniżej. Możemy przechowywać formuły z różnymi atrybutami, na przykład ich dowody lub części dowodów, tak jak każdy inny wyraz w bazie danych systemu. Aby zmienić wyrażenia składające się z wyrazów i typów na różne sposoby, które nie są obsługiwane przez reguły bezpośredniego przepisywania, musimy uzyskać do nich dostęp zakodowany na bardziej bezpośrednim poziomie. W tym celu definiujemy w systemie typ terminów i typ typów oraz określamy, w jaki sposób terminy dowolnego typu odpowiadają zakodowanym terminom. Kodowanie używa T Term dla konstruktora terminów, T Variable dla zmiennej terminów i TT Type, TT Function, TT Type var do kodowania typów. Nazwy definicji składni są używane jako ciągi znaków w kodowaniu, a zmienne terminów są kodowane razem z ich typami. Na przykład termin (x : booleans)::[] reprezentujący listę z jedną wartością boolowską można zakodować jako:

```
T_Term ("'A_:_:_'L_",
    [T_Variable ("x_", TT_Type ("booleans_", []), []),
     T_Term ("[_]_", [])]).
```

Aby wykorzystać przedstawione kodowanie, dodajemy do systemu funkcje specjalne, które umożliwiają dostęp do informacji o już zdefiniowanych typach, funkcjach i ostatnio zdefiniowanych. Informacje o funkcjach umieszczane są w specjalnym typie, który podaje tagi, reguły przepisywania dla funkcji i typ funkcji; otrzymujemy analogiczne dane dla konstruktorów. Aby wybrać informacje w systemie, używamy zapytań tagów, które są listami nazw tagów i opcjonalnych wartości. Element z tagami spełnia zapytanie, jeśli ma tagi o odpowiednich nazwach zdefiniowanych, a gdy w zapytaniu podana jest wartość tagu, odpowiadający mu tag elementu musi mieć tę samą wartość. Funkcje

get constructors [tag query]

get functions [tag query]

pobierają z systemu wszystkie definicje konstruktorów lub funkcji, które spełniają podane zapytania. Wszystkie konstruktory typu o nazwie t mają specjalny tag #type = t dla łatwego dostępu. Aby faktycznie wykorzystać opisane kodowanie, musimy przenieść funkcje zdefiniowane pomiędzy wyrazami na poziom normalny, gdzie funkcje przyjmują argumenty i zwracają wyniki różnych typów. Załóżmy, że mamy funkcję, która przyjmuje N wyrazów i zwraca wyraz, f : term, . . . , term → term. Następnie możemy zdefiniować funkcję normalną o podanej nazwie i typach type1, . . ., typeN i zwrócić typ ret type w następujący sposób:

Define function [name] [type1] ... [typeN]

into [res_type] from meta function [f].

Taka funkcja robi to samo, co wykonywanie f z kodowaniem, tj. koduje wszystkie argumenty, wykonuje f na zakodowanych wyrazach i dekoduje wynik z powrotem. Ponadto ta definicja wprowadza do systemu nowe formuły logiczne, które należy udowodnić, aby zagwarantować, że f rzeczywiście ma zadeklarowany typ. Są to proste formuły, które można udowodnić, po prostu wykonując funkcję sprawdzającą typ. Musimy zdefiniować funkcję sprawdzającą typ, która używa definicji typu i funkcji pobranych z systemu, aby obliczyć typ podanych wyrazów, ale jest to tylko problem techniczny. Używanie funkcji meta sprawia, że kompilowanie funkcji do C lub C++ jest bardziej problematyczne. Czasami wymagane jest przepisanie wyrazów ze zmiennymi, więc musimy przygotować kod C++ na takie przypadki i sprawdzić go, gdy zostanie wykonane dopasowanie. Ponadto, podczas korzystania z wbudowanych typów C++ czasami musimy wykonać pakowanie, aby umożliwić reprezentację zmiennych wyrazów. Ponadto, musimy zachować w kodzie mapowanie przypisanych identyfikatorów symboli konstruktora i funkcji na ich nazwy ciągów, aby móc wykonywać funkcje meta. Możliwość operowania na poziomie meta nie jest trywialnie implementowana, ale mając ją możemy zdefiniować reguły logiki i rozumowania w jasny sposób. Ogólnie rzecz biorąc, będziemy reprezentować reguły rozumowania jako funkcje, zwykle implementowane na poziomie meta, które przyjmują przesłanki w formie formuły T i generują wnioski tego samego typu, zwykle oznaczane:

przesłanki : formuła |- wnioski : formuła.

W takich funkcjach możemy wykorzystać informacje o konstruktorach typu, aby dokonać indukcji na tym typie, i możemy uzyskać dostęp do reguł przepisywania dla określonych funkcji, aby je rozpoznać i zaimplementować wyspecjalizowane procedury decyzyjne. Czasami musimy zdefiniować nowe funkcje lub typy w systemie, aby móc skonstruować wnioski. Aby to umożliwić, pozwalamy regułom rozumowania na tworzenie listy poleceń systemowych, które są wykonywane po kolei przed oceną reguły w sposób analogiczny do tego, w jaki są analizowane zdania złożone. Aby nadać sens regułom rozumowania, przedstawimy zbiór podstawowych reguł, które są uważane za prawdziwe, to znaczy przenoszą prawdziwe przesłanki na prawdziwe wnioski. Formuły udowodnione za pomocą tych reguł

są również prawdziwe i pozwalamy na rozszerzenie zbioru reguł używanych do rozumowania poprzez sprowadzenie udowodnionych formuł do poziomu meta. Dokładniej, jeśli udowodnimy formułę $\varphi \rightarrow \psi$ przy użyciu funkcji $f_1, \ldots, f_n$ i typów $t_1, \ldots, t_n$ wtedy możemy dodać regułę wnioskowania $\delta \wedge \varphi' + \psi'$. W tej regule φ' i ψ' różnią się od φ i ψ tylko tym, że funkcje i typy są zastępowane zmiennymi tego samego typu. W δ wykorzystujemy możliwość uzyskania konstruktorów typów i przepisujemy reguły dla funkcji, aby sprawdzić, czy definicje wszystkich f_i i t_i są równoważne definicjom zmiennych, którymi zostały zastąpione w φ' i ψ' . W ten sposób reguła wnioskowania zależy tylko od semantyki funkcji i typów, a nie od ich nazw. W następnej sekcji przedstawimy podstawowe reguły wnioskowania i grę służącą do znajdowania dowodów i ich zrozumienia. Wykorzystując możliwość konstruowania reguł dedukcji z udowodnionych wzorów możemy udowodnić poprawność logicznych procedur decyzyjnych. W ten sposób można udowodnić metody decyzyjne wykorzystujące automaty lub eliminację kwantyfikatorów i stosować je podczas rozumowania o odpowiadających sobie obiektach. Na przykład stare procedury dla teorii liczb rzeczywistych z dodawaniem i mnożeniem lub dla arytmetyki Presburgera i jej nowoczesnych wariantów.

Gra rozumowania i wyszukiwania terminów

Zdefiniujmy teraz grę, która pozwoli na wyszukiwanie programów i udowodnienie ich własności w modelu przepisywania terminów typowanych przedstawionym wcześniej. Pozycje w tej grze są wzorami i zakładamy, że zmienne wolne są niejawnie uniwersalnie kwantyfikowane. Nie będziemy identyfikować pozycji, które różnią się tylko nazwami zmiennych, ale ich tożsamość będzie ważna przy określaniu zwycięzcy, jeśli zostanie zastosowana indukcja. W tej grze każda reguła rozumowania $\varphi \vdash \psi$ opisuje możliwy ruch Eloise z ψ do φ i możliwy ruch Abelarda z $\neg\psi$ do $\neg\varphi$. Gdy wiemy, że $\varphi_1 \vdash \psi, \ldots, \varphi_k \vdash \psi$ i że $\psi \vdash \varphi_1 \vee \ldots \vee \varphi_k$, wówczas nazywamy zbiór ruchów Eloise z ψ do $\varphi_1, \ldots, \varphi_k$ kompletne i analogiczne dla ruchów Abelarda od $\neg\psi$ do $\neg\varphi_1, \ldots, \neg\varphi_k$. Założymy również, że wyrazy w równościach wewnątrz pozycji są zawsze przepisywane do ich normalnych form. Podczas dowodzenia własności funkcji, które się nie kończą, możemy nie być w stanie spełnić tego wymogu i wpaść w nieskończoną pętlę podczas próby normalizacji wyrazu po wykonaniu ruchu. Założymy, że takie ruchy są niedozwolone i nie będziemy ich rozważać. Najpierw opiszemy warunek wygranej w grze i podamy podstawowy zestaw prostych ruchów, które są wystarczające do przeprowadzenia indukcji na strukturze typu zdefiniowanej przez konstruktory, do uogólnienia wzoru i podstawienia części wzoru przy użyciu niektórych już udowodnionych równości. Aby uzyskać pełną moc niezbędną do wszystkich dowodów, musimy dodatkowo utworzyć nowe reguły rozumowania, jak opisano wcześniej, lub dodać nowe typy, funkcje i udowodnić lematy. Mimo to podstawowe reguły rozumowania odpowiadają pojęciu prostego dowodu i powinny wystarczyć do intuicyjnie łatwych własności. Przedstawiamy również proste ruchy, które nie są kompletne, ale często można je wykorzystać w praktyce, aby szybciej znaleźć dowody. Zauważ, że w wielu przypadkach dowody istnienia funkcji prowadzą do definicji tej funkcji i dlatego mówimy, że jest to również gra poszukiwawcza. Gdy obecne są zmienne funkcyjne, czasami dodamy reguły przepisywania dla tych funkcji do systemu w trakcie gry lub nawet zdefiniujemy nowe funkcje w trakcie dowodu. W przypadku stwierdzeń egzystencjalnych, które są dowodzone w sposób niekonstruktywny, pozwalamy również zdefiniować odpowiadające im funkcje w systemie za pomocą polecenia define ... from formula podobnego do tego, którego użyliśmy dla funkcji meta. Oczywiście, takie funkcje mogą być używane tylko do dowodów, nie można ich przepisywać i kompilować, ale mimo to czasami przydatne jest ich zdefiniowanie. Najpierw określmy, które pozycje są trywialnie wygrywające dla Eloise, a które dla Abelarda. Jedyne trywialnie wygrywające pozycje dla Eloise to pozycje $t = t$ dla pewnego wyrazu t, a pozycje trywialnie wygrywające dla Abelarda to te $s_1 = s_2$, gdzie s_1 i s_2 są wyrazami podstawowymi i nie są równe. Oczywiście, jeśli dowolny gracz może przejść z pozycji p do pozycji wygrywającej dla siebie, to pozycja p również jest wygrywająca i gwarantujemy, że każda pozycja będzie wygrywająca dla co najwyżej jednego gracza. Kiedy zestaw ruchów jest

kompletny, to jeśli gracz przegrywa przy wszystkich tych ruchach, to przegrywa w tej pozycji. Kiedy używamy reguł indukcyjnych, musimy sprawdzić, czy wracamy do pozycji identycznej z tą, od której zaczęliśmy, tylko z nowymi zmiennymi. Omówimy to później, kiedy zostaną przedstawione ruchy indukcyjne. Pierwszy rodzaj ruchów, które przeanalizujemy, jest bardzo prosty; Eloise może przejść z φ ∨ ψ, a Abelard może przejść z φ ∧ ψ do φ lub ψ. Ruchy te odpowiadają regule rozumowania: $\varphi \vdash \varphi \vee \psi.$

Dla Eloise jest to bezpośrednia odpowiedniość, podczas gdy dla Abelarda musimy podstawić regułę na ¬φ |- ¬φ ∨ ¬ψ i pamiętać, że ¬¬φ = φ. Dla każdej reguły, gdy chcemy wyciągnąć z niej możliwe ruchy Abelarda, powinniśmy również pamiętać o podstawieniu formuł negacyjnych. Dwie reguły φ |- φ ∨ ψ,ψ |- φ ∨ ψ są kompletne. Ponieważ uważamy, że ruchy w grze są bardziej intuicyjne niż reguły rozumowania używane w implementacji, będziemy trzymać się przedstawiania ruchów. Innym możliwym ruchem dla Abelarda w pozycji φ jest próba indukcyjnego opisu terminów, które można podstawić za zmienne w Var(φ) w zależności od ich typu, co jest możliwe dla x ∈ Var(φ) przez rozważenie wszystkich konstruktorów typu(x), jeśli nie jest to zmienna typu i nie jest typem funkcyjnym. Załóżmy na przykład, że istnieje pozycja f(x) = c, gdzie x jest zmienną z lists(α). Następnie Abelard może przejść albo do f(Nil) = c, albo do f(Cons(x_0, x_1)) = c. Należy wziąć pod uwagę wszystkie możliwe konstruktory, z których można skonstruować typ, a następnie taki zestaw ruchów jest kompletny. Podobna indukcja jest możliwa dla zmiennych funkcyjnych na typie dowolnych argumentów lub na typie wyniku, a następnie nowe reguły przepisywania muszą zostać dodane do systemu. Na przykład niech z będzie zmienną funkcyjną o typie α, lists(α) → lists(α). Następnie możemy dokonać indukcji na drugim argumencie w taki sposób, że zdefiniujemy nową funkcję w systemie o nazwie fz z regułami przepisywania:

$$f_z(x, \mathrm{Nil}, z_1, z_2) \to z_1(x) \quad , \quad f_z(x, \mathrm{Cons}(y_1, y_2), z_1, z_2) \to z_2(x, y_1, y_2).$$

Musimy objąć wszystkie możliwe konstruktory wybranego typu argumentu i dodać odpowiednią liczbę nowych zmiennych funkcyjnych (z_1, z_2) z typami przekazującymi. Następnie musimy zastąpić każde wystąpienie z parą (z_1, z_2) i każde wywołanie z(x, y) przez fz(x, y, z_1, z_2). Gdy wystąpienia zmiennej z jako wartości funkcyjnej zostaną podstawione, będziemy musieli zmienić funkcje, które jej używają, aby przyjmowały parę (z_1, z_2) jako argument zamiast z i używały f_z(x, y, z_1, z_2) zamiast z(x, y). Propagowanie tych zmian może wymagać od nas zdefiniowania innych nowych funkcji z odpowiednimi typami w systemie, ale powinno być jasne, że takie ruchy są poprawne i kompletne. Gdy wykonujemy indukcję na typie zwracanym, albo ustawiamy omawianą funkcję na stałą lub na jedną ze zmiennych, która ma ten sam typ co wynik, albo ustawiamy ją na wywołanie funkcji innej funkcji, która może używać dodatkowych pośrednich wyników obliczeń. Aby wykonać ten ruch, musimy najpierw sklonować wzór w naszej sytuacji, aby uzyskać odpowiednią liczbę zmiennych funkcyjnych, więc zamiast φ(z) rozważamy w naszym przykładzie φ(z_1) ∨ φ(z_2) ∨ φ(z_3) i konstruujemy nowe funkcje:

$$f_{z_1}(x, y) \to \mathrm{Nil} \quad , \quad f_{z_2}(x, y) \to y \quad , \quad f_{z_3}(x, y, v_1, v_2) \to \ v_1(x, y, v_2(x, y)).$$

Zauważ, że typ wyniku pośredniego v_2(x, y) będzie przyjmowany jako tak ogólny, jak to tylko możliwe, więc weźmiemy typ krotki dla wszystkich argumentów, a dodatkowo typ ciągu dla innych obliczonych informacji, więc ostatecznie będzie to: pairs(pairs(α, lists(α)), strings). Następnie zastępujemy zmienną z_1 przez f_{z1}, z_2 przez f_{z2}, a z_3 przez f_{z3} w taki sam sposób, jak zrobiliśmy to powyżej z f_z. W pierwszym wzorze φ(z_1) zmienna z_1 może zniknąć z powodu normalizacji do Nil, w drugim wzorze można ją zastąpić drugim argumentem. W trzecim będziemy teraz mieć dwie nowe zmienne funkcyjne v_1, v_2 i musimy poprawić typy i być może odpowiednio rozszerzyć system, aby dostać się do właściwej pozycji φ‘() ∧ φ’’(y) ∧ φ’’’(c, v_1, v_2). Ponieważ zawsze normalizujemy człony w pozycjach, do których się

przemieszczamy, może się zdarzyć, że podczas gry wrócimy do pozycji, w której już byliśmy, ale z innymi zmiennymi. Na przykład, jeśli mamy funkcję f zdefiniowaną przez reguły przepisywania f(Nil) → Nil i f(Cons(x, y)) → f(y), to możemy chcieć pokazać, że φ = (f(x) = Nil) jest prawdziwe. W pozycji φ Abelard musi wykonać jeden z kompletnych ruchów indukcyjnych opisanych powyżej, więc może albo przejść do f(Nil) = Nil, co zostanie przepisane w locie na Nil = Nil i trywialnie wygrywa dla Eloise, albo do f(Cons(x_1, x_2)) = Nil, co zostanie przepisane na f(x_2) = Nil, i może powtarzać ten ruch w nieskończoność. Eloise może mieć podobne problemy, próbując udowodnić ∃x f(x) = Cons(1, Nil). Aby poradzić sobie z takimi problemami, gdy identyczna pozycja modulo zmiana nazwy zmiennej jest powtarzana w cyklu lub jeśli mamy jakąkolwiek nieskończoną grę, musimy być bardziej ostrożni przy określaniu, kto wygrywa. W prostym przypadku, gdy tylko pozycja jednego gracza jest powtarzana nieskończenie często i ten gracz wykonuje ruch indukcyjny, wtedy gracz przegrywa. Ale z przeplatanymi kwantyfikatorami egzystencjalnymi i uniwersalnymi otrzymujemy większy problem. Na przykład, jeśli dla pewnej funkcji g analizujemy wzór ∃x ∀y g(x, y) = T, wtedy może się zdarzyć, że wykonamy kolejno indukcję na x i y. Ale aby zachować znaczenie kwantyfikatorów, musimy upewnić się, że żaden krok indukcyjny dla x nie zależy od poprzednich kroków dla y. Aby to zagwarantować, być może będziemy musieli rozważyć zbiory potęgowe pozycji i sprawdzić, czy strategie są tam skorelowane. Przy większej liczbie przeplatających się kwantyfikatorów mogą to być nawet zbiory potęgowe zbiorów potęgowych itd., ponieważ problem spełnialności dla struktur automatycznych, który można sprowadzić do tego, ma nieelementarną złożoność w liczbie wystąpień przeplatających się kwantyfikatorów. Istnieje inny ważny rodzaj ruchu indukcyjnego, który może wykonać Eloise i jest on również zupełny. Załóżmy, że mamy równość f(t_1, . . . , t_n) = t gdzieś wewnątrz wzoru φ i że funkcja f jest zdefiniowana przez zbiór reguł przepisywania l_1 → r_1, . . . , l_k → r_k. Kiedy mówimy, że f jest zdefiniowana przez zbiór reguł przepisywania R, zakładamy, że dla dowolnych wyrazów podstawowych u_1, . . . , u_n w postaci normalnej, wyraz f(u_1, . . . , u_n) można zapisać na najwyższej pozycji za pomocą pewnej reguły z R. Ponadto zakładamy, że kolejność stosowania reguł nie ma znaczenia dla reguł w zbiorze R. Załóżmy, że funkcje w naszym systemie są zdefiniowane wyczerpująco, więc spełniony jest pierwszy wymóg. Gdy mamy uporządkowane liniowe reguły przepisywania, zawsze możemy uczynić je niezależnymi od kolejności, wyliczając konstruktory, na przykład jeśli i zostało zdefiniowane przez and(T, T) → T, and(x, y) → F, to możemy zmienić reguły na and(T, T) → T, and(x, F) → F, and(F, x) → F, aby uczynić je niezależnymi od kolejności. Wróćmy teraz do równości f(t_1, . . . , t_n) = t i reguł l^i → r_i, które są wyczerpujące i niezależne od kolejności, i niech l_i = f(l^1_i , . . . , l^n_i). Ponieważ wyraz f(t_1, . . . , t_n) zostanie przepisany przez niektóre z tych reguł, gdy zostanie podstawiony jako podstawa, możemy poszukać poprawnej reguły i podstawień, aby go przepisać i sprawdzić wzór później. Odpowiada to możliwości, aby Eloise przesunęła się do pozycji $\psi_1 \vee \ldots \vee \psi_k$, gdzie:

$$\psi_i = \exists \mathrm{Var}(\mathrm{l_i})\ t_1 = l_i^1 \wedge \ldots \wedge t_n = l_i^n \wedge \varphi[f(t_1, \ldots, t_n) = t \leftarrow r_i = t],$$

gdzie φ[f(t_1, . . . , t_n) = t ← r_i = t] jest pozycją φ z równością f(t_1, . . . , t_n) = t zmienioną na r_i = t. Zauważ, że jeśli pozycja φ zawiera zmienne niezwiązane, nowe zmienne z l_i przyjmują zmienne niezwiązane jako argumenty, co odpowiada skolemizacji. Aby to wyjaśnić, rozważmy pozycję, która implikuje(t_1, t_2) = F dla pewnych terminów t_1 i t_2 z dwiema zmiennymi niezwiązanymi x i y, i niech implikuje będzie implikacją normalną zdefiniowaną przez implikuje(F, v)→ T, implikuje(T, T) → T, implikuje(T, F)→ F. W tym przypadku Eloise może przejść do wzoru:

$$(\exists v\ t_1 = \mathrm{F}\ \wedge\ t_2 = v(x, y)\ \wedge\ \mathrm{T} = \mathrm{F}) \vee$$

$$\vee\ (t_1 = \mathrm{T}\ \wedge\ t_2 = \mathrm{T}\ \wedge\ \mathrm{T} = \mathrm{F}) \vee (t_1 = \mathrm{T}\ \wedge\ t_2 = \mathrm{F}\ \wedge\ \mathrm{F} = \mathrm{F}),$$

co może być wygraną tylko dla ostatniego składnika, więc Eloise przechodzi do $t_1 = T \vee t_2 = F$. Zauważ, że v była zmienną funkcyjną i przyjęła x i y jako argumenty. Jest to kompletny ruch i można go wykonać wewnątrz formuły skwantyfikowanej lub formuły ze zmiennymi wolnymi, jak powyżej. Nie jest to możliwe na przykład dla $\varphi \vee \psi$ jako $\forall x\, \varphi \vee \psi$ $\forall x\, \varphi \vee \forall x\, \psi$. Jak mogłeś zauważyć, indukcja na zmiennych funkcyjnych dla Abelarda umożliwi udowodnienie czegokolwiek interesującego tylko w bardzo rzadkich przypadkach, ponieważ zwykle komplikuje problem jedynie indukcja na funkcjach. Ale dla Eloise może być to bardzo ważne, jeśli chce znaleźć funkcję o określonej własności. Umożliwiliśmy użycie wyników pośrednich i dodaliśmy typ string, indukując na typie wyniku funkcji, aby wszystkie funkcje obliczalne były w ten sposób reprezentowalne, ale często powinniśmy szukać ładniejszego rozwiązania, używając innych funkcji i typów, które już mamy w systemie. Ponadto, gdy szukamy wyrazu o typie niefunkcjonalnym, może być przydatne przedstawienie go jako wyniku obliczenia funkcji, która już istnieje. Dokładniej, załóżmy, że szukamy wyrazu typu T, aby podstawić go albo za zmienną związaną x, albo za wynik funkcji w ruchu indukcyjnym Eloise dla zmiennej funkcjonalnej. W drugim przypadku istnieją dodatkowe parametry $x_1, \ldots, x_n$, które są argumentami funkcji z typem $(x_i) = T_i$. Weźmy zatem dowolną funkcję f zdefiniowaną w systemie z typem $S_1, \ldots, S_k \to R$ taką, że istnieje podstawienie typu σ, dla którego $R\sigma = T$ i dla pewnych indeksów $\{i_1, \ldots, i_l\} \subseteq \{1, \ldots, k\}$ możemy przypisać liczby $p(i_m)$ tak, aby $Si_m\sigma = T_{p(im)}$. Za pomocą tej funkcji możemy przedstawić szukany wyraz jako $x = f(y_1, \ldots, y_k)$, gdzie dla $m \in \{i_1, \ldots, i_l\}$ mamy $y_m = x_{p(m)}$, a pozostałe argumenty to nowe zmienne, które ponownie zostaną poproszone o znalezienie. Mniej formalnie, po prostu przedstawiamy szukany wyraz jako wywołanie funkcji z dowolną kombinacją już istniejących lub nowych argumentów. W ten sposób możemy użyć dowolnej funkcji z systemu, która ma odpowiedni typ, aby znaleźć szukany wyraz. Takie ruchy są opcjonalne tylko dla Eloise, ale w praktyce bardzo często wykorzystuje się w ten sposób posiadaną wiedzę, a wiele naturalnych problemów można rozwiązać w zaledwie kilku krokach, jeśli odpowiednie funkcje są znane z góry i zostaną użyte we właściwym czasie. Opisane powyżej ruchy stanowią podstawę wszystkich dowodów i powinny wystarczyć w przypadku bardzo prostych własności i znajdowania programów, które nie są złożone. Jednak w przypadku nieco bardziej interesujących dowodów musimy użyć innych wzorów udowodnionych wcześniej, aby wejść w interakcję z tym, który chcemy udowodnić. Przedstawimy możliwe ruchy dla takiej interakcji; nie są one kompletne, a niektóre z użytych wzorów muszą być już znane jako prawdziwe, co wygrywa dla Eloise. Należy pamiętać, że przedstawiliśmy również sposób tworzenia nowych reguł rozumowania, gdy te tutaj nie są wystarczające do wydajnego rozwiązania problemu. Gdy Eloise gra w pozycji φ, może wybrać dowolny wyraz t z typem T, który pojawia się w pewnej pozycji w niektórych równościach w φ i nie ma zmiennych ograniczonych, a następnie przejść do pozycji ψ, która jest identyczna z φ, przy czym wszystkie wystąpienia t w dowolnej pozycji w dowolnym wyrazie w dowolnej równości są zastąpione zmienną x z typem T. Ten ruch nazwiemy uogólnieniem t. Aby wykonać inny ruch, załóżmy, że wiemy, że formuła $t = s_1 \vee \ldots \vee t = s_n$ jest prawdą i jesteśmy w pozycji φ, która zawiera wyraz u w pewnej równości u = s. Jeśli dla pewnej pozycji p w u i dla pewnego podstawienia σ mamy $u|_p = t\sigma$ i żadne zmienne w u|p nie są ograniczone, to zdefiniujmy ψi jako pozycję identyczną z φ z wyrazem u zastąpionym przez $u[si\sigma]_p$. Wtedy możemy pozwolić Eloise na przejście z φ do $\psi_1 \vee \ldots \vee \psi_n$. Umożliwiamy graczom inny sposób poruszania się lub zmiany systemu, który umożliwia definiowanie nowych typów, funkcji i konstruowanie nowych pozycji do analizy przy użyciu istniejących jako bloków konstrukcyjnych. Te ruchy są opisane w prosty sposób: każdy gracz może wybrać dowolny dobrze typizowany wyraz, zbudować dobrze skonstruowaną pozycję i wstawić ją do gry. Może również zbudować funkcję z argumentami i typami wyników, które już istnieją w systemie i wybrać dla niej szereg reguł przepisywania. Nowe typy można również konstruować, wybierając pewną liczbę dobrze uformowanych konstruktorów, a zarówno dla funkcji, jak i typów można budować ich kilka naraz i uczynić je wzajemnie rekurencyjnymi. Można również dowodzić lematów i tworzyć nowe reguły

wnioskowania. Opcjonalne ruchy połączone z prostymi umożliwiają dowodzenie złożonych własności programów.

Jak widać, istnieje ograniczony zestaw rozsądnych ruchów podstawowych i szersza możliwość wykonywania ruchów opcjonalnych przy użyciu wiedzy w systemie lub tworzenia nowych typów i funkcji, które mogą być przydatne później. Aby grać w grę w dobry sposób, tak aby wszystkie fałszywe formuły szybko okazały się fałszywe, a wszystkie prawdziwe zostały skutecznie udowodnione, Eloise i Abelard muszą używać rozsądnych strategii i wykonywać odpowiednie ruchy zgodnie z sytuacją. Oczywiście, każda strategia gracza, która nie pomija żadnego nieskończenie długiego możliwego ruchu, jest procedurą wyszukiwania programu i znajdzie programy, które udowodnią spełnienie określonego wzoru. Gdy sama gra jest zdefiniowana z odpowiednim typem wewnątrz systemu i możliwe ruchy są również zdefiniowane, możemy określić, że strategia jest funkcją, która wybiera możliwy ruch w danym stanie gry, i możemy użyć procedury wyszukiwania opartej na grze, aby znaleźć lepsze strategie i w związku z tym sprawić, aby strategia sama się udoskonalała poprzez naukę, jak opisano wcześniej w dyskusji teoretycznej. Wyrażenie rozumowania jako gry umożliwia zrozumienie heurystyk, których używamy do rozumowania, takich jak „zawsze najpierw spójrz na kilka prostych przykładów, zanim zaczniesz dowodzić" lub „nie używaj jednej indukcji po drugiej" jako prostych strategii w grze. W przypadku pewnych typów pozycji możemy użyć procedur decyzyjnych, które już istnieją, i uwzględnić je w grze, gdy tylko zostaną zaimplementowane, ponieważ podane są reguły rozumowania i dowody ich poprawności. Procedury te nie muszą być złożone i kompletne, mogą również reprezentować dobre heurystyki. Jako bardzo prosty przykład załóżmy, że jesteśmy w pozycji $\exists y\ x + y(x, z) = z$ lub podano bardziej złożone wyrażenie arytmetyczne, ale tylko ze stałymi i dodawaniem. Pierwszym ruchem, jaki powinna wykonać każda dobrze działająca strategia, jest podstawienie $y(x, z) = z-x$ lub użycie rozwiązywacza algebraicznego w celu znalezienia właściwej funkcji do podstawienia i w ten sposób włączenie prostej procedury decyzyjnej do gry rozumowania. Można również wdrożyć bardziej zaawansowane reguły wnioskowania, wykorzystując automaty i eliminację kwantyfikatorów.

Wnioski

Pokazaliśmy, jak metody wyszukiwania programów mogą być używane zarówno do rozwiązywania problemów, jak i do automatycznego konstruowania wydajniejszych rozwiązywaczy problemów. Po pokazaniu teoretycznego rozwiązania zademonstrowaliśmy wygodny model obliczeń i grę do rozumowania. Twierdziliśmy, że model obliczeń może być praktyczny i wydajny, a w naszej grze rozumowania możemy zrozumieć podejmowane działania i włączyć inne procedury decyzyjne. Przedstawiony model i metoda rozumowania są wystarczająco obszerne, aby objąć zadania sztucznej inteligencji ogólnej, i wystarczająco proste, aby używać ich do określonych zadań rozumowania podczas programowania. Obecnie pracujemy nad tym, aby system był przyjazny dla użytkownika i aby zbudować dla niego podstawową bibliotekę wiedzy. Warto byłoby mieć obszerną standardową bibliotekę typów, programów zakodowanych jako reguły przepisywania i dowody ważnych faktów na temat tych programów i powiązanych typów w przedstawionym modelu. Wraz z kilkoma prostymi heurystykami pisanymi ręcznie, wydajnymi metodami kompilacji dla reguł przepisywania z typowaniem i notacją programu rozszerzoną, aby była wygodna w użyciu, powinno to uczynić praktycznym tworzenie dowodów poprawności programów, a nawet generowanie prostych programów automatycznie. Jednocześnie byłaby to interesująca baza testów i dowodów formalnych w prostym modelu teoretycznym, więc mogłaby być potencjalnie wykorzystywana przez inne systemy, a także służyć jako zbiór przykładów do nauczania przyszłych samodoskonalących się procedur. Pytanie, czy możliwe będzie zdefiniowanie heurystyk rozumowania na tyle dobrze, aby działały one bezpośrednio w bardziej złożonych programach w przedstawionym modelu, pozostaje otwarte. Ale ponieważ ruchy, które podejmujemy w grze rozumowania, mają jasne, intuicyjne znaczenie, możemy

mieć nadzieję na sformalizowanie w ten sposób naszych własnych metod myślenia lub, jeśli nam się nie uda, przynajmniej na jasne zrozumienie, które z intuicyjnych kroków, które podejmujemy przy rozwiązywaniu problemów, są najbardziej problematyczne dla AI. Odkrywamy, że projekt systemu, który opisaliśmy i który dodatkowo obsługuje przetwarzanie języka naturalnego, stanowi podstawę AGI. Nie można oczekiwać od takiego systemu takich rzeczy, jak świadomość lub rozpoznawanie mowy, przynajmniej nie zanim nie zostaną w nim zaprogramowane. Można jednak rozwiązywać problemy, a nawet czasami pisać programy automatycznie, tylko określając, jakie właściwości muszą być zachowane. Ten system jest również żywotnym środowiskiem projektowania i rozwoju oprogramowania, w którym można wdrażać wydajne aplikacje z interfejsami graficznymi i w którym można automatyzować żmudne zadania programistyczne. Ponieważ włączono przetwarzanie języka naturalnego, można również poprosić ekspertów w określonych dziedzinach o zapisanie ich wiedzy bezpośrednio w systemie, a następnie wykorzystanie tej wiedzy w programach lub do rozumowania. Gdy włączy się dużą bazę wiedzy i szereg heurystyk rozumowania, system będzie również w stanie uczyć się od nich i optymalizować własną strukturę. Ponieważ przedstawiony system potrafi ustanowić korespondencję między naturalnym myśleniem i językiem człowieka a formalną notacją odpowiednią dla komputerów i generowania kodu, sprawia, że komunikacja i współpraca między ludźmi a komputerami są praktyczne w niemal każdej sytuacji, gdy problem wymaga rozwiązania. Dlatego uważamy, że przedstawiony sposób łączenia logiki formalnej z przetwarzaniem języka naturalnego i umożliwienia jej rozszerzenia za pomocą heurystyk numerycznych jest interesujący dla przyszłego rozwoju AGI. W tym rozdziale pominęliśmy wiele ważnych badań powiązanych ze sztuczną inteligencją. Nie omawialiśmy logiki rozmytej i probabilistycznej oraz modeli obliczeniowych, chociaż z pewnością należy je wykorzystać. Nadal jednak wolimy uwzględniać je i powiązane z nimi metody weryfikacji [6] jako procedury rozumowania w prezentowanym modelu, niż analizować je jako podstawowe elementy na tym samym poziomie co prymitywy programowania i logika. Ponieważ omawiany temat jest bardzo obszerny, z pewnością nie wspomnieliśmy i nie powołaliśmy się na wszystkie istotne publikacje, więc w celu bardziej szczegółowego zbadania należy zapoznać się z pierwszymi czterema książkami z listy literatury.