Autoreferat

1 Imię i nazwisko

Dariusz Biernacki

2 Posiadane dyplomy i stopnie naukowe

Stopień doktora informatyki
BRICS PhD School, DAIMI, University of Aarhus, Aarhus, Dania, 2005

Tytuł rozprawy doktorskiej:
The Theory and Practice of Programming Languages with Delimited Continuations

Promotor: prof. Olivier Danvy

Dyplom nostryfikowany w 2006 roku na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego.

Tytuł magistra matematyki ze specjalnością informatyczną
Uniwersytet Marii Curie-Skłodowskiej w Lublinie, 2001

Tytuł pracy magisterskiej:
Struktury wielowartościowe a wnioskowanie niemonotoniczne

Promotor: dr Jerzy Mycka

3 Informacje o dotychczasowym zatrudnieniu w jednostkach naukowych

01.10.2007 – : adiunkt w Instytucie Informatyki Uniwersytetu Wrocławskiego

01.09.2006 – **31.08.2007**: staż podoktorski w INRIA Futurs (Team Démons), Orsay, Francja

01.02.2006 – **31.08.2006**: asystent na Wydziale Matematyki, Fizyki i Informatyki Uniwersytetu Marii Curie-Skłodowskiej w Lublinie

01.09.2002 – **31.08.2005**: doktorant, BRICS PhD School, DAIMI, University of Aarhus, Aarhus, Dania

01.02.2001 – **31.08.2002**: asystent na Wydziale Matematyki, Fizyki i Informatyki Uniwersytetu Marii Curie-Skłodowskiej w Lublinie

01.10.2000 – **31.06.2001**: student-asystent na Wydziale Matematyki, Fizyki i Informatyki Uniwersytetu Marii Curie-Skłodowskiej w Lublinie

4 Wskazanie osiągnięcia wynikającego z art. 16 ust. 2 ustawy z dnia 14 marca 2003 r. o stopniach naukowych i tytule naukowym oraz o stopniach i tytule w zakresie sztuki (Dz. U. nr 65, poz. 595 ze zm.)

4.1 Tytuł

Operatory sterowania w językach programowania wyższego rzędu:
struktura typów oraz równoważność programów

4.2 Publikacje

[22] Małgorzata Biernacka, Dariusz Biernacki. A context-based approach to proving termination of evaluation. *25th Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXV)*, ENTCS 249, 169–192, Oksford, Wielka Brytania, 2009.

[23] Małgorzata Biernacka, Dariusz Biernacki. Context-based proofs of termination for typed delimited-control operators. *11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2009)*, 289–300, Coimbra, Portugalia, 2009.

[26] Małgorzata Biernacka, Dariusz Biernacki, Sergueï Lenglet, Marek Materzok. Proving termination of evaluation for system F with control operators. *1st Workshop on Control Operators and their Semantics (COS 2013)*, EPTCS 127, 15–27, Eindhoven, Holandia, 2013.

[25] Małgorzata Biernacka, Dariusz Biernacki, Sergueï Lenglet. Typing control operators in the CPS hierarchy. *13th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2011)*, 149–160, Odense, Dania, 2011.

[134] Marek Materzok, Dariusz Biernacki. Subtyping delimited continuations. *16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011)*, 81–93, Tokio, Japonia, 2011.

Rozszerzona wersja artykułu została przyjęta do druku w czasopiśmie *Higher-Order and Symbolic Computation*.

[135] Marek Materzok, Dariusz Biernacki. A dynamic interpretation of the CPS hierarchy. *10th Asian Symposium on Programming Languages and Systems (APLAS 2012)*, LNCS 7705, 296–311, Kioto, Japonia, 2012.

[42] Dariusz Biernacki, Piotr Polesiuk. Logical relations for coherence of effect subtyping. *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*, LIPIcs 38, 107–122, Warszawa, Polska, 2015.

[38] Dariusz Biernacki, Sergueï Lenglet. Applicative bisimulations for delimited-control operators. *15th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2012)*, LNCS 7213, 119–134, Tallin, Estonia, 2012.

Pełna wersja artykułu: http://arxiv.org/abs/1201.0874.

[39] Dariusz Biernacki, Sergueï Lenglet. Normal form bisimulations for delimited-control operators. *13th International Symposium on Functional and Logic Programming (FLOPS 2012)*, LNCS 7294, 47–61, Kobe, Japonia, 2012.

Pełna wersja artykułu: http://arxiv.org/abs/1202.5959.

[40] Dariusz Biernacki, Sergueï Lenglet. Environmental bisimulations for delimited-control operators. *11th Asian Symposium on Programming Languages and Systems (APLAS 2013)*, LNCS 8301, 333–348, Melbourne, Australia, 2013.

Pełna wersja artykułu: http://hal.inria.cffr/hal-00862189.

[41] Dariusz Biernacki, Sergueï Lenglet. Applicative bisimilarities for call-by-name and call-by-value $\lambda\mu$-calculus. *30th Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXX)*, ENTCS 308, 49–64, Ithaca, USA, 2014.

Pełna wersja artykułu: http://hal.inria.fr/hal-00926100.

4.3 Omówienie wyników

4.3.1 Wstęp

Moje najważniejsze osiągnięcia naukowe dotyczą semantyki języków programowania wyższego rzędu z kontynuacjami oraz jej zastosowań w zakresie implementacji języków programowania oraz wnioskowania o programach z efektami sterowania. Prowadzone przeze mnie badania nad operatorami sterowania opierają się na rozważaniu rozmaitych rozszerzeń rachunku lambda traktowanego jako prototypowy język programowania [81] z wykorzystaniem metod semantyki operacyjnej (maszyny abstrakcyjne, semantyka redukcyjna) [81], transformacji programów (translacje do stylu kontynuacyjnego, defunkcjonalizacja [164, 158, 59, 61]), systemów typów (à la Church i à la Curry) [184], relacji logicznych [185, 90] oraz technik bisymulacji [2, 169, 171].

4.3.1.1 Kontynuacje

Pojęcie kontynuacji jest wszechobecne w literaturze dotyczącej zarówno teorii, jak i praktyki języków programowania wyższego rzędu, a także semantyki języków programowania w ogóle. Intuicyjnie kontynuację można rozumieć jako obiekt reprezentujący „obliczenia

pozostałe do wykonania" w danym punkcie programu. Przy takiej definicji nie dziwi popularność kontynuacji – w każdej sytuacji, kiedy mowa jest o obliczeniach, można mówić o „obliczeniach pozostałych do wykonania".

Kontynuacje „odkrywano" wielokrotnie w latach 60. i 70. XX wieku [163], ponieważ pojawiały się one w różnych kontekstach, które nie od razu kojarzono ze sobą. Na nich opiera się styl programowania zwany stylem kontynuacyjnym (ang. continuation-passing style, w skrócie CPS), w którym kontynuacje są reprezentowane jawnie jako funkcje, czy to w języku źródłowym (otrzymujemy wówczas techniki transformacji programów [87, 164, 158] oraz techniki programowania [186]), czy to w meta-języku (otrzymujemy wówczas styl semantyki denotacyjnej [188] oraz tzw. interpreter definiujący [164]). Ponadto, przełomowe prace Landina [120] oraz Reynoldsa [164] zapoczątkowały traktowanie kontynuacji jako pierwszorzędnych elementów języka, mających taki sam status jak funkcje czy inne rodzaje danych w językach wyższego rzędu. Takie podejście doprowadziło do odkrycia istotnych zastosowań praktycznych kontynuacji, a także ich niespodziewanych związków z systemami logicznymi. Konstrukcje języka, które pozwalają programiście na dostęp do bieżącej kontynuacji i manipulowanie nią, nazywane są w literaturze operatorami sterowania.

4.3.1.2 Operatory sterowania

Bezpośredni dostęp do kontynuacji za pomocą operatorów sterowania wzbogaca język programowania o efekty sterowania, które inaczej można osiągnąć jedynie przez transformację całego programu do stylu kontynuacyjnego. Operatory sterowania dla kontynuacji abortywnych[1], takich jak `callcc`, mają długą historię w językach funkcyjnych Scheme [190] oraz Standard ML of New Jersey [95] i są używane rutynowo przez biegłych programistów funkcyjnych. Liczne zastosowania operatorów abortywnych obejmują m. in. modelowanie efektów sterowania, takich jak nielokalne wyjścia z rekursji i programowanie z nawrotami [95], jak również modelowanie konstrukcji współbieżnych, takich jak koprocedury (ang. coroutines) [98] i procesy lekkie (ang. lightweight processes) [201].

Z drugiej strony Griffin w swojej przełomowej pracy [93] opisał, w jaki sposób poprzez izomorfizm Curry'ego-Howarda można otrzymać logiczną interpretację dla języków z abortywnymi operatorami sterowania. Wiadomo, że rachunek lambda z typami prostymi odpowiada w tej interpretacji minimalnej logice intuicjonistycznej [184]; Griffin pokazał, że dodanie do języka operatora `C` Felleisena [82, 84, 81] pozwala na eleganckie rozszerzenie izomorfizmu Curry'ego-Howarda do pełnej logiki klasycznej. Korzystając z tego istotnego odkrycia Murthy z kolei pokazał, w jaki sposób można rozszerzyć znany mechanizm ekstrakcji programów z dowodów logicznych do logiki klasycznej: otrzymuje się w ten sposób programy, które wykorzystują operatory sterowania takie jak `C` i `callcc` [144, 143]. Niezależnie od Griffina Parigot zaproponował formalizm znany jako rachunek $\lambda\mu$, który odpowiada na tej samej zasadzie pewnemu wariantowi systemu naturalnej dedukcji dla logiki klasycznej, używającemu sekwentów z wieloma konkluzjami [149], natomiast obliczeniowo

[1] Pojęcie kontynuacji abortywnych oznacza tutaj zwykłe kontynuacje (określane również mianem nieograniczonych lub skokowych), które reprezentują *całe* obliczenie pozostałe do wykonania w programie, dla odróżnienia od kontynuacji ograniczonych (określanych również mianem kontynuacji składalnych lub częściowych), które reprezentują *pewien początkowy fragment* obliczenia pozostałego do wykonania w programie.

jest równoważny rachunkowi lambda z abortywnymi operatorami sterowania [66]. Zarówno podejście Griffina jak i podejście Parigot do opisu języków z operatorami abortywnymi były następnie intensywnie badane i rozwijane, m. in. w pracach [65, 67, 51, 10].

4.3.1.3 Kontynuacje ograniczone

Operatory sterowania dla kontynuacji ograniczonych, wprowadzone przez Felleisena [80] oraz przez Danvy'ego i Filinskiego [57, 58], pozwalają na manipulację sterowaniem w programach funkcyjnych za pomocą jawnej reprezentacji bieżącej *ograniczonej* kontynuacji. W przeciwieństwie do abortywnego operatora `callcc`, operatory sterowania dla kontynuacji ograniczonych modelują złożenie kontynuacji (ograniczonych), a nie skoki do kontynuacji abortywnych. W szczególności, o ile za pomocą `callcc` czy `C` programista może wyrazić bezpośrednio wszystko to, co da się opisać stylem kontynuacyjnym, to użycie operatorów `shift` i `reset` Danvy'ego i Filinskiego pozwala na bezpośrednie wyrażenie wzorców programowania charakteryzujących styl składania kontynuacji (ang. continuation-composing style) używany m. in. w programowaniu z nawrotami przy użyciu kontynuacji sukcesu i porażki [58]. Operator `shift` umożliwia przechwycenie bieżącej kontynuacji ograniczonej oraz złożenie jej z inną, podczas gdy ogranicznik sterowania (ang. control delimiter) `reset` służy do zresetowania (ograniczenia) bieżącej kontynuacji.

Filinski pokazał, że `shift` i `reset` należą do fundamentalnych pojęć w obszarze gorliwych języków funkcyjnych, ponieważ język wyposażony w te operatory jest monadycznie zupełny, tj. dowolna monada obliczeniowa może być wyrażona bezpośrednio za pomocą tych operatorów [85, 86]. Aby otrzymać tę samą wyrażalność za pomocą operatora `callcc`, język musi być dodatkowo wyposażony w obsługę mutowalnego stanu [85]. Inne istotne teoretyczne wyniki wykorzystujące operatory sterowania dla kontynuacji ograniczonych to m. in.: pełna abstrakcja dla języków ze sterowaniem [183] i algorytm normalizacji przez ewaluację (ang. normalization by evaluation, NbE) dla rachunku lambda z typami wariantowymi [13]. Operatory sterowania dla kontynuacji ograniczonych znajdują też zastosowania bardziej praktyczne, m. in. w następujących obszarach i technikach programowania: programowanie niedeterministyczne [58], ewaluacja częściowa [54], generacja kodu [198], programowanie obliczeń mobilnych [189], programowanie internetowe [159], a także w lingwistyce [179], systemach operacyjnych [113] czy obliczeniach probabilistycznych [115]. Popularne języki funkcyjne, takie jak Haskell [78], Ocaml [112], Scala [166], Scheme [88] i SML [85] oferują już także różne warianty operatorów sterowania dla kontynuacji ograniczonych, co powoduje wzrost ich popularności wśród programistów.

Teoretyczne własności operatorów dla kontynuacji ograniczonych są wciąż przedmiotem aktywnych badań, które prowadzą do nowych odkryć w zakresie m. in. teorii typów i teorii dowodu [10, 12, 114, 205, 99, 103, 104], jak również teorii równościowych [100, 173, 73, 75, 74] dotyczących tych operatorów.

4.3.2 Osiągnięcia

W czasie studiów doktoranckich zajmowałem się zagadnieniami związanymi z semantyką operacyjną, wyrażalnością i zastosowaniami statycznych i dynamicznych kontynuacji ogra-

niczonych oraz ich hierarchii. Wyniki tych badań zostały opisane w pracach [4, 34, 24, 36, 37, 35] oraz w mojej rozprawie doktorskiej [31]. Na podstawie tych wyników można sformułować tezę, że transformacje programów (takie jak transformacja do stylu kontynuacyjnego i defunkcjonalizacja) stanowią techniki kluczowe dla zrozumienia, formalnego opisu i implementacji operatorów sterowania dla kontynuacji ograniczonych, jak również do efektywnego używania ich w praktyce programistycznej. Większość wyników uzyskanych w opisanych pracach dotyczy języków nietypowanych.

Moje główne osiągnięcie naukowe po uzyskaniu stopnia doktora dotyczy kontynuacji badań nad teorią i praktyką operatorów sterowania, które prowadziłem w ramach dwóch ścieżek badawczych:

1. We współpracy z Małgorzatą Biernacką i Sergueïem Lenglet, a także z moimi doktorantami Markiem Materzokiem i Piotrem Polesiukiem, opracowaliśmy szereg systemów typów dla języków programowania wyposażonych w kontynuacje ograniczone. Systemy te stanowią rozwinięcie i uogólnienie prac Danvy'ego i Filinskiego [57], którzy zaproponowali kanoniczny system typów z efektami dla tych operatorów, a także prac innych autorów. Zaproponowane przez nas systemy typów odzwierciedlają warstwową strukturę opartą na stylu kontynuacyjnym, charakteryzują się dużą ekspresywnością i stanowią dobry fundament dla możliwych implementacji typowanych języków funkcyjnych wyposażonych w operatory sterowania dla kontynuacji ograniczonych. Ponadto w ramach tych prac opracowaliśmy dwie ogólne (stosowalne nie tylko dla języków z operatorami sterowania) metody dowodzenia pewnych istotnych własności programów i systemów typów. Pierwsza z nich to kontekstowa metoda dowodzenia terminacji ewaluacji programów dobrze typowanych, a druga to metoda dowodzenia koherencji dla systemów typów ze złożonymi regułami podtypowania (ang. subsumption rules). Istniejące metody dowodzenia tych własności nie dają się wprost zastosować dla rozważanych przez nas języków, stąd potrzeba znalezienia innego podejścia. W rezultacie otrzymaliśmy metody, które same w sobie stanowią nowe i interesujące podejście do rozważanych zagadnień, zasługujące na osobne potraktowanie.

2. We współpracy z Sergueïem Lenglet opracowaliśmy behawioralne teorie wnioskowania koindukcyjnego o równoważności programów z operatorami sterowania. Jedna z nich, która opiera się na podstawach operacyjnych opisanych w mojej pracy doktorskiej, dotyczy różnych technik bisymulacji (bisymulacje aplikatywne, postaci normalnej, środowiskowe) dla kontynuacji ograniczonych i dostarcza praktycznych narzędzi do ustalania równoważności kontekstowej między programami z efektami zawierającymi operatory `shift` i `reset`. Ten wynik rozszerza dotychczas istniejący arsenał narzędzi wnioskowania, ponieważ równoważność kontekstowa jest pojęciem silniejszym niż ta oparta na semantyce kontynuacyjnej wspomnianych operatorów. Otrzymane rezultaty można również w łatwy sposób zaadaptować do innych operatorów sterowania rozważanych w literaturze.

   Druga z opracowanych przez nas teorii behawioralnych dostarcza odpowiedzi na otwarty problem, w jaki sposób można scharakteryzować równoważność kontekstową w rachunku $\lambda\mu$ za pomocą bisymulacji. W ramach odpowiedzi na to pytanie opracowa-

liśmy poprawne i pełne techniki bisymulacji aplikatywnych oraz środowiskowych dla różnych strategii redukcji w rachunku $\lambda\mu$, częściowo korzystając z wyników otrzymanych wcześniej dla kontynuacji ograniczonych. Ten wynik stanowi pierwszą pełną teorię koindukcyjną dla abortywnych operatorów sterowania.

W dalszej części tej sekcji znajduje się krótki opis każdego z osiągniętych wyników; dla każdego z nich podana jest motywacja, ogólny opis oraz jego kontekst, tj. opis literatury ściśle związanej z danym zagadnieniem. Szczegóły techniczne można znaleźć we wskazanych niżej publikacjach.

Sekcje 4.3.2.1 – 4.3.2.3 przedstawiają wyniki dotyczące systemów typów dla operatorów sterowania oraz ich własności:

4.3.2.1: Kontekstowe dowody normalizacji dla języków z efektami sterowania – wyniki zaprezentowane w pracach [22, 23, 25, 134, 26];

4.3.2.2: Ekspresywne systemy typów z efektami dla kontynuacji ograniczonych – wyniki zaprezentowane w pracach [23, 25, 134, 135, 26];

4.3.2.3: Relacje logiczne w dowodzeniu koherencji podtypowania efektów – wyniki zaprezentowane w pracy [42];

natomiast Sekcja 4.3.2.4 jest poświęcona równoważności programów wykorzystujących efekty sterowania:

4.3.2.4: Zastosowanie bisymulacji do dowodzenia równoważności programów wykorzystujących efekty sterowania – wyniki zaprezentowane w pracach [38, 39, 40, 41].

4.3.2.1 Kontekstowe dowody normalizacji dla języków z efektami sterowania

W podejściu opartym na przepisywaniu termów (ang. term-rewriting), rachunek lambda jako archetypiczny język programowania zwykle przedstawiany jest za pomocą gramatyki termów oraz relacji redukcji zdefiniowanej na termach. Felleisen i in. wprowadzili pojęcie kontekstu redukcyjnego/ewaluacyjnego [82, 83, 84], bazującego na intuicyjnym pojęciu „termu z dziurą” [14], które okazało się niezwykle przydatne do zwięzłego opisu różnych strategii redukcji. Kontekst Felleisena reprezentuje term otaczający bieżący podterm, albo inaczej „resztę obliczenia pozostałą do wykonania”, i bezpośrednio odpowiada kontynuacji: tę ostatnią można rozumieć jako sposób reprezentacji kontekstu za pomocą funkcji. Ściślej, Danvy zauważył, że konteksty redukcyjne można otrzymać poprzez defunkcjonalizację kontynuacji występujących w definicji funkcji implementującej redukcję jednokrokową, natomiast konteksty ewaluacyjne można otrzymać poprzez defunkcjonalizację kontynuacji użytych w definicji funkcji ewaluacji, implementującej semantykę dużych kroków [55, 56]. (Ponieważ w obu przypadkach zdefunkcjonalizowane kontynuacje prowadzą do dokładnie tego samego typu danych, pojęcia kontekstów redukcyjnych i ewaluacyjnych są zwykle używane zamiennie.)

Przez swój ścisły związek z kontynuacjami, korzyści z zastosowania reprezentacji kontekstowej języków programowania są największe w przypadku języków wyposażonych w operatory sterowania, tj. konstrukcje językowe pozwalające na manipulowanie bieżącą kontynuacją (kontekstem), takie jak `callcc` [82], czy `shift` i `reset` [24]. Ponadto, jak pokazali Wright i Felleisen [204], semantyka redukcyjna oparta na kontekstach stanowi wygodny formalizm dla opisu i wnioskowania o poprawności typowej języka (ang. type soundness).

W serii prac [22, 23, 25, 134, 26] opracowaliśmy ogólną, kontekstową metodę dowodzenia innej kluczowej własności języków opartych na rachunku lambda – terminacji ewaluacji. Pokazaliśmy zastosowanie tej metody dla różnych strategii ewaluacji, dla języków z typami prostymi i polimorficznymi, zawierającymi operatory sterowania dla kontynuacji abortywnych i ograniczonych.

Typy proste W przypadku języków z operatorami sterowania typowanymi za pomocą systemów typów prostych standardowym sposobem dowodzenia terminacji ewaluacji (i własności normalizacji w ogóle) jest translacja, w odpowiedni sposób zachowująca kroki redukcji, z języka źródłowego do innego języka, o którym wiadomo, że ma własność normalizacji [93, 176]. Jednak, tego typu pośrednie metody w ogólności są kłopotliwe i nieodporne na błędy, jak zauważyli Ikeda i Nakazawa [102].

W pracy [22] zaprezentowaliśmy nową, bezpośrednią metodę dowodzenia terminacji ewaluacji dla rachunku lambda z typami prostymi, dla strategii gorliwej (ang. call-by-value) oraz leniwej (ang. call-by-name), która wykorzystuje konteksty redukcyjne w sposób kluczowy. Następnie rozszerzyliśmy czysty rachunek lambda o popularne abortywne operatory sterowania: `callcc`, `abort` oraz `C`, i zaadaptowaliśmy opracowaną metodę do tak rozszerzonego języka, wykorzystując jego kontekstową semantykę redukcyjną.

Metoda dowodu, którą stosujemy w tej pracy wykorzystuje wariant kontekstowy metody Taita opartej na predykatach redukowalności [193] i jest modyfikacją metody użytej we wcześniejszej pracy Pierce'a [154] oraz metody użytej w pracy Biernackiej i in. korzystających z „bezkontekstowych" (ang. direct-style) predykatów redukowalności [30]. W tej metodzie predykat redukowalności zdefiniowany na dobrze typowanych termach wyrażał następującą własność: jeśli term redukowalny zaaplikujemy do redukowalnego argumentu odpowiedniego typu, to otrzymany term jest także redukowalny. Ponadto każdy term redukowalny ma własność normalizacji. Dowód terminacji w tym przypadku polega na pokazaniu, że wszystkie dobrze typowane termy są redukowalne, z czego wynika, że mają własność normalizacji. W podejściu kontekstowym natomiast definiujemy predykat redukowalności tylko na wartościach (rozszerzenie do wszystkich termów jest możliwe, ale niekonieczne) i mówimy, że wartość jest redukowalna, jeśli ma następującą własność: jeśli zaaplikujemy ją do innej redukowalnej wartości odpowiedniego typu, i włożymy w redukowalny kontekst odpowiedniego typu, to otrzymany program ma własność normalizacji. Jednocześnie definiujemy własność redukowalności kontekstu, tj. takiego, który po włożeniu weń redukowalnej wartości odpowiedniego typu, ma własność normalizacji jako program. Przyjmując takie kontekstowe definicje redukowalności otrzymujemy metodę dowodzenia normalizacji, która jest prosta i bezpośrednia, i wykorzystuje naturalny, kontekstowy sposób definiowania semantyki redukcyjnej. Ponadto ta metoda daje się rozszerzać do języków wyposażonych

w operatory sterowania, co nie jest możliwe przy użyciu wspomnianej wyżej oryginalnej metody Taita.

Pewne warianty metody dowodzenia własności normalizacji z użyciem predykatów w stylu Taita były w literaturze stosowane dla czystego rachunku lambda, zarówno dla słabej, jak i silnej normalizacji [18, 193, 199], jak również dla normalizacji do słabej czołowej postaci normalnej przy użyciu strategii leniwej (podejście Martin-Löfa) oraz gorliwej (podejście Hoffmanna) [30]. Rozszerzenie tej metody dla operatorów sterowania było rozważane przez Parigot, który zmodyfikował metodę kandydatów redukowalności Girarda w celu udowodnienia silnej normalizacji dla rachunku $\lambda\mu$ drugiego rzędu, odpowiadającego systemowi naturalnej dedukcji dla logiki klasycznej [150]. Z kolei Berger i Schwichtenberg zauważyli, że zawartość obliczeniowa ich konstruktywnego dowodu silnej normalizacji przy użyciu predykatów redukowalności stanowi algorytm będący instancją tzw. normalizacji przez ewaluację. Ta obserwacja została następnie wykorzystana w dowodzie normalizacji do słabej czołowej postaci normalnej dla logiki kombinatorycznej przez Coquanda i Dybjera [48], oraz dla rachunku lambda przez Biernacką i in. [30].

Niektóre z dowodów normalizacji zostały sformalizowane w systemach wspierających dowodzenie twierdzeń (ang. proof assistants), a następnie z tych sformalizowanych dowodów otrzymano w sposób automatyczny poprzez ekstrakcję programy (funkcyjne) normalizujące termy do odpowiednich postaci normalnych [17, 20]. W przypadku dowodów przeprowadzonych metodą kontekstową, uzyskane przez ekstrakcję programy stanowią przykłady algorytmów w stylu kontynuacyjnym, w których kontynuacje otrzymane są poprzez ekstrakcję kodu z predykatów redukowalności dla kontekstów. W ten sposób otrzymujemy także potwierdzenie ścisłego związku między kontekstami a kontynuacjami od strony logicznej.

W przypadku operatorów sterowania `shift` i `reset` dla kontynuacji ograniczonych standardowa semantyka redukcyjna ma dwie warstwy kontekstów [24]. W pracy [23] pokazaliśmy w jaki sposób można rozszerzyć metodę kontekstową do języków używających tych operatorów, zarówno przy użyciu strategii gorliwej, jak i leniwej. Rozważane w tej pracy systemy typów bazują na kanonicznym systemie typów z efektami Danvy'ego i Filinskiego [57] (więcej szczegółów na temat tych systemów typów znajduje się w sekcji 4.3.2.2). Dla każdego z dowodów normalizacji wyprowadziliśmy jego zawartość obliczeniową w postaci odpowiedniego ewaluatora, który w tym przypadku jest zapisany w stylu kontynuacyjnym z dwiema warstwami kontynuacji i jest instancją normalizacji przez ewaluację [16, 18, 30]. Otrzymane programy obliczają słabą czołową postać normalną dla programów dobrze typowanych zawierających operatory `shift` i `reset` według odpowiedniej strategii ewaluacji i są dowodliwie poprawne względem rozważanej semantyki redukcyjnej. W tej pracy pokazujemy również, że inny przydatny system typów z efektami dla operatorów `shift` i `reset`, tj. system z ustalonym typem odpowiedzi kontynuacji, również posiada własność normalizacji, jeśli tylko typ odpowiedzi jest typem bazowym [11].

Metoda kontekstowa została również zastosowana dla dwóch innych systemów typów z efektami. W pracy [25] udowodniliśmy w ten sposób terminację ewaluacji dla typowanego wariantu hierarchii CPS [58], w której kontynuacje ułożone są w hierarchię warstw, którą można manipulować za pomocą rodziny operatorów będących uogólnieniem operatorów `shift` i `reset` (więcej szczegółów na temat systemów typów dla hierarchii CPS znajduje się w sekcji 4.3.2.2). Z kolei w pracy [134] otrzymaliśmy analogiczny wynik dla najbar-

dziej ekspresywnego znanego monomorficznego systemu typów z efektami dla kontynuacji ograniczonych: systemu typów dla operatorów `shift0` i `reset0` z podtypowaniem efektów, który wprowadziliśmy w innej pracy [134] (więcej szczegółów na temat tego systemu typów znajduje się w sekcji 4.3.2.2).

Polimorfizm parametryczny W pracy [26] pokazaliśmy, w jaki sposób metodę kontekstową można uogólnić tak, by dało się ją zastosować do udowodnienia terminacji ewaluacji w systemie F, poprzez rozważenie kontekstowych kandydatów redukowalności à la Girard (będących uogólnieniem predykatów redukowalności) [90]. Tak, jak w przypadku rachunku lambda z typami prostymi, istniejące w literaturze dowody terminacji dla polimorficznego rachunku lambda z operatorami sterowania wykorzystują głównie metody pośrednie, oparte na translacjach do innych silnie normalizowalnych języków: Harper i Lillibridge pokazali redukcję tego problemu dla systemu F_ω z operatorami `abort` i `callcc` przy strategii gorliwej do problemu normalizacji w czystym systemie F_ω [96] za pomocą odpowiedniej translacji do stylu kontynuacyjnego. Parigot pokazał redukcję problemu silnej normalizacji dla rachunku $\lambda\mu$ drugiego rzędu do problemu silnej normalizacji dla rachunku lambda z typami prostymi [150], Danos i in. pokazali redukcję problemu silnej normalizacji dla logiki klasycznej drugiego rzędu do silnej normalizacji w logice liniowej [52]. Wreszcie Kameyama i Asai pokazali redukcję problemu silnej normalizacji w systemie F z operatorami `shift` i `reset` (ze standardową semantyką) do problemu silnej normalizacji w czystym systemie F [108].

Z drugiej strony Parigot pokazał również bezpośredni dowód silnej normalizacji dla rachunku $\lambda\mu$ drugiego rzędu, który wykorzystuje pewien wariant metody kandydatów redukowalności [150]. Później pojawiły się kolejne wyniki wykorzystujące różne adaptacje metody Taita-Girarda w zastosowaniu do logik drugiego rzędu [53, 119, 128, 136]. W szczególności, Girard zapoczątkował wykorzystanie metod opartych na pojęciu ortogonalności do wyrażenia metod bazujących na redukowalności. To podejście pośrednio wprowadza pojęcie kontekstu rozumianego jako ciąg termów, z grubsza odpowiadającego kontekstom w strategii leniwej. Kandydat redukowalności w tym ujęciu jest definiowany za pomocą tzw. zbioru TT-zamkniętego. W przeciwieństwie do tego podejścia, w naszej pracy rozważamy konkretne strategie ewaluacji i używane przez nas konteksty pochodzą bezpośrednio z semantyki redukcyjnej (gorliwej lub leniwej; w przypadku operatorów sterowania dla kontynuacji ograniczonych pojawiają się także konteksty warstwowe), a używane przez nas pojęcia kandydatów redukowalności zawierają wyłącznie wartości i nie są TT-zamknięte. Podobne podejście zostało użyte w pracy Lindleya i Starka [130], którzy dowiedli silnej normalizacji dla rachunku Moggiego wprowadzając specjalną interpretację typów obliczeniowych wykorzystującą operację TT-liftingu. Ta operacja wydaje się w pewien sposób odpowiadać naszej definicji redukowalności dla jednej warstwy kontekstów. Jednak w przypadku silnej normalizacji pojęcie kontekstu należy rozumieć jako narzędzie służące do syntaktycznego podziału termu na części, a nie „pozostałą do wykonania resztę obliczenia". W szczególności, nasza metoda nie wymaga analizowania redukowalności termów składowych kontekstów.

Rozważane przez nas języki to system F z operatorem `callcc` przy strategiach gorliwej i leniwej, a także system F z operatorami `shift` i `reset` przy tychże strategiach. W każdym

z rozważanych wariantów używamy standardowej semantyki, gdzie (w przeciwieństwie do semantyki w stylu języka ML) ewaluacja nie jest wykonywana wewnątrz abstrakcji polimorficznych [96]. W przypadku operatora `callcc` używamy systemu typów inspirowanego systemem typów Harpera i Lillibridge'a [96], natomiast w przypadku operatorów `shift` i `reset` wprowadzamy nowe systemy typów (osobny dla każdej ze strategii), które uogólniają system typów Asaiego i Kameyamy [12], ponieważ pozwalają na abstrakcję polimorficzną nad dowolnymi wyrażeniami, a nie tylko czystymi (więcej szczegółów na temat tego systemu typów można znaleźć w sekcji 4.3.2.2). Tak jak w przypadku typów prostych, także i w tym przypadku zawartość obliczeniowa prezentowanych dowodów ma strukturę ewaluatorów w stylu kontynuacyjnym.

4.3.2.2 Ekspresywne systemy typów z efektami dla kontynuacji ograniczonych

Struktura kontynuacyjna operatorów `shift` i `reset` narzuca system typów z efektami, wprowadzony przez Danvy'ego i Filinskiego, który przez ponad 15 lat był najbardziej ekspresywnym znanym monomorficznym systemem typów dla tych operatorów [57]. Sądy typowe w tym systemie mają postać $\Gamma; B \vdash e : A; C$, co oznacza, że przy założeniach Γ wyrażenie e można wstawić do kontekstu oczekującego wartości typu A i zwracającego wartość typu B oraz do meta-kontekstu oczekującego wartości typu C, przy czym typy B i C mogą być różne. Chociaż możliwość modyfikacji typu odpowiedzi kontekstu jest kluczową cechą większości systemów typów dla operatorów sterowania dla kontynuacji ograniczonych, system Danvy'ego i Filinskiego nie jest wystarczająco mocny, by zagwarantować poprawność typową dla wielu ciekawych i praktycznych programów, np. dla klasycznego przykładu programu generującego listy prefiksów z danej listy [24]. Z tego powodu opracowaliśmy, z zachowaniem poprawności, bardziej ekspresywne systemy typów bazujące na systemie Danvy'ego i Filinskiego, oraz uogólniliśmy ten system typów do bardziej ekspresywnych operatorów sterowania niż `shift` i `reset`, jak wyszczególniono poniżej.

shift i reset Wprowadzając konteksty typowane w pracy [23] otrzymaliśmy dwa nowe systemy typów dla operatorów `shift` i `reset`: jeden dla strategii gorliwej i jeden dla strategii leniwej. Systemy te zostały wyprowadzone z odpowiednich translacji do stylu kontynuacyjnego i są poprawne względem redukcji (własność zachowania typu przez redukcję jest spełniona w każdym przypadku) i mają własność terminacji jak wspomniano w sekcji 4.3.2.1. System dla strategii gorliwej jest udoskonaloną wersją oryginalnego systemu Danvy'ego i Filinskiego [57] w tym sensie, że pozwala na pewną postać niejawnego polimorfizmu w typie odpowiedzi kontekstu, która jest niezbędna do zapewnienia typowania pewnych programów, które nie dają się otypować w oryginalnym systemie. Z kolei przedstawiony przez nas system dla strategii leniwej jest nowy. Użyta przez nas translacja do stylu kontynuacyjnego dla operatorów `shift` i `reset` przy strategii leniwej została ponadto następnie wykorzystana przez Kameyamę i Tanakę jako punkt wyjścia dla aksjomatyzacji kontynuacji ograniczonych przy strategii leniwej [110], natomiast sam system typów dla tej strategii został wykorzystany przez tych samych autorów jako podstawa systemu typów dla ich hierarchii kontynuacyjnej [195].

W pracy [26] zaprezentowaliśmy wariant systemu F z efektami oparty na systemie monomorficznym dla strategii gorliwej, wprowadzonym we wcześniejszej pracy [23]. Ten system typów został wyprowadzony na podstawie translacji do stylu kontynuacyjnego, co oznacza, że typy abstrakcji polimorficznych są dodatkowo wyposażone w typy dla kontynuacji (kontekstów), podobnie jak typy lambda abstrakcji w systemie monomorficznym z efektami à la Danvy i Filinski są wyposażone w dodatkowe informacje o typie kontynuacji. Opisany przez nas system typów jest bardziej liberalny niż ten wprowadzony przez Asaiego i Kameyamę [12]. W ich pracy typy abstrakcji polimorficznych nie zawierają dodatkowych informacji o typach i mogą być użyte tylko dla wyrażeń bez efektów. Natomiast w naszym systemie pozwalamy na otypowanie dowolnej abstrakcji kosztem dodatkowej dekoracji w typach abstrakcji polimorficznych. Nasz system jest poprawny (redukcje zachowują typy) oraz ma własność terminacji (por. sekcja 4.3.2.1).

Hierarchia CPS Iterowanie translacji do stylu kontynuacyjnego pozwala na wprowadzenie hierarchii kontynuacji i na uogólnienie pojęcia kontynuacji i meta-kontynuacji, czyli pojęć używanych do zdefiniowania semantyki operatorów `shift` i `reset`. Posługując się tak otrzymaną hierarchią kontynuacji (ang. CPS hierarchy) Danvy i Filinski wprowadzili hierarchię operatorów sterowania $\texttt{shift}_i$ i $\texttt{reset}_i$ ($i \geq 1$), stanowiącą uogólnienie operatorów `shift` i `reset`. Hierarchia CPS pozwala na rozdzielenie efektów sterowania, które powinny istnieć niezależnie od siebie w programie [58]. Przykładowo, aby zebrać wszystkie rozwiązania znalezione przez algorytm z nawrotami zaimplementowany z użyciem operatorów $\texttt{shift}_1$ i $\texttt{reset}_1$, należy użyć operatorów $\texttt{shift}_2$ i $\texttt{reset}_2$, aby rozdzielić operacje szukania i emitowania znalezionych rozwiązań. Hierarchia CPS znajduje również zastosowanie w opisie zagnieżdżonych obliczeń w strukturach hierarchicznych, np. w pracy Biernackiej i in. [24] pokazaliśmy, jak za pomocą hierarchii zapisać w sposób naturalny algorytmy NbE (w technice normalizacji przez ewaluację) dla hierarchicznych języków zawierających operatory produktu i jedności, będące uogólnieniem problemu znajdowania dyzjunkcyjnej i koniunkcyjnej postaci normalnej dla formuł logicznych.

Hierarchia CPS była badana głównie w wariancie beztypowym. Danvy i Filinski zdefiniowali ją używając beztypowej translacji do stylu kontynuacyjnego oraz za pomocą funkcji ewaluacyjnej semantyki denotacyjnej [58], Danvy i Yang zdefiniowali semantykę operacyjną dla hierarchii i na tej podstawie zaimplementowali ją w języku SML [63], Kameyama podał aksjomatyzację hierarchii, która jest poprawna i zupełna względem translacji CPS [107], a Biernacka i in. wyprowadzili maszyny abstrakcyjne i semantykę redukcyjną zgodne z ewaluatorem definiującym hierarchię [24].

Efektem ubocznym implementacji w języku ML jest uzyskanie systemu typów dla hierarchii, który jednak jest dość restrykcyjny: na każdym poziomie typ odpowiedzi kontynuacji jest ustalony raz na zawsze. Ten system typów stanowi uogólnienie systemu Filinskiego dla operatorów `shift` i `reset` [85], ale nie był formalnie badany. Formalne systemy typów dla hierarchii pojawiły się w pracach Murthy'ego [145] i Shana [179]. Murthy zaproponował bardziej liberalne typy niż te z implementacji Danvy'ego i Yanga w tym sensie, że kontynuacje ograniczone na poziomie i-tym mogą mieć różne typy odpowiedzi, pod warunkiem że zgadzają się one z typem oczekiwanym przez kontynuację na poziomie $i + 1$. Z kolei

system Shana stanowi uogólnienie systemu Danvy'ego i Filinskiego [57], w którym efekty sterowania mogą zmieniać typ odpowiedzi kontekstu, w którym występują; oznacza to, że statycznie typ odpowiedzi kontynuacji na poziomie i-tym może być różny od tego oczekiwanego przez kontynuację na poziomie $i+1$. Praca Shana jest motywowana zastosowaniami hierarchii w lingwistyce, i rozważa on nieco inaczej zorganizowaną hierarchię niż ta zdefiniowana przez Danvy'ego i Filinskiego (poziom 0 jest u niego najwyżej, a w oryginalnej hierarchii jest najniżej). Ponadto, Shan nie pokazuje żadnych metateoretycznych własności swojego systemu.

W pracy [25] zaproponowaliśmy system typów, który również stanowi uogólnienie systemu Danvy'ego i Filinskiego, ale który został wyprowadzony bezpośrednio z iterowanej translacji do stylu kontynuacyjnego definiującej oryginalną hierarchię CPS. Ponadto operatory, które badamy, są nieco bardziej elastyczne (chociaż tak samo ekspresywne) niż oryginalna rodzina operatorów $\texttt{shift}_i$ i $\texttt{reset}_i$ w tym sensie, że kolejne warstwy kontynuacji są przechwytywane w osobnych zmiennych oraz możliwe jest przekazywanie wartości do krotek kontynuacji (które to kontynuacje mogły zostać przechwycone przez różne operatory). Taka forma operatorów pojawia się naturalnie w strukturze kontynuacji, jeśli rozważa się operacje przechwytywania kontynuacji oraz przekazywania wartości do przechwyconych kontynuacji jako dwie niezależne operacje.

Celem naszej pracy było opracowanie systemu typów dla operatorów sterowania w hierarchii CPS, który nie ogranicza programisty i oferuje możliwość zaprogramowania w hierarchii wszystkich tych programów, które są poprawnie typowane w stylu kontynuacyjnym za pomocą typów prostych, co jednak oznacza, że powstały system jest dość złożony. Ponadto chcieliśmy w ten sposób podać naturalną interpretację typów w hierarchii CPS. Jak zaobserwował Shan [179] najogólniejsze typy à la Danvy i Filinski są konieczne w niektórych praktycznych zastosowaniach operatorów, np. w lingwistyce do obsługi niejednoznaczności zasięgu kwantyfikatorów. Istnieją także przykłady programów wymagających modyfikacji typu odpowiedzi na pierwszym poziomie hierarchii, np. generowanie prefiksów listy [24] czy implementacja funkcji printf [12]. Typowy scenariusz, w którym występuje niezgodność typów między typem odpowiedzi na poziomie i-tym i typem oczekiwanym na poziomie $i+1$ występuje np. wtedy, gdy porzucamy obliczenie typu A na poziomie i i zwracamy wartość typu B do poziomu $i+1$, co jest standardowym wzorcem programowania w CPS.

W rezultacie przeprowadzonych badań opracowaliśmy podstawy teorii typów dla hierarchii CPS i zaproponowaliśmy ogólny schemat badania typowanych operatorów sterowania definiowalnych w ramach hierarchii CPS. Szczegółowe osiągnięte wyniki są następujące:

- zdefiniowanie nowej rodziny operatorów sterowania w hierarchii CPS; te operatory są nieco bardziej elastyczne niż oryginalna hierarchia operatorów $\texttt{shift}_i$ i $\texttt{reset}_i$, zadanych za pomocą translacji do CPS oraz semantyki redukcyjnej zgodnej z tą translacją;
- system typów à la Danvy i Filinski dla tych operatorów oraz dowód własności zachowania typu przez redukcję i poprawności systemu względem translacji CPS;
- dowód terminacji ewaluacji w semantyce redukcyjnej przeprowadzony za pomocą wariantu metody kontekstowej (wspomnianej w sekcji 4.3.2.1);

- symulacja zaproponowanych operatorów za pomocą oryginalnej rodziny operatorów $\texttt{shift}_i$ i $\texttt{reset}_i$;
- uogólnienie tych wyników do hierarchii jeszcze bardziej elastycznych operatorów wyrażalnych w hierarchii CPS.

shift0 i reset0 W swojej przełomowej pracy na temat operatorów sterowania dla kontynuacji ograniczonych [57] Danvy i Filinski krótko zarysowali warianty operatorów `shift` i `reset`, odpowiednio znane jako `shift0` i `reset0`. Semantyka operatora `reset0` jest taka sama jak operatora `reset`, natomiast `shift0`, w przeciwieństwie do operatora `shift`, umożliwia dostęp do kontekstów ograniczonych dowolnie głęboko zagnieżdżonych w metakontekście poprzez kolejne wielokrotne przechwytywanie kontekstów. Dzięki tej możliwości operator `shift0` okazuje się interesującym narzędziem o dużej sile wyrazu, które – w porównaniu do operatora `shift` – nie było wcześniej tak szczegółowo studiowane. W przypadku nietypowanym `shift0` i `reset0` były badane przez Shana [180], który zaprezentował translację do stylu kontynuacyjnego dla tych operatorów, stanowiącą konserwatywne rozszerzenie translacji Plotkina przy strategii gorliwej [158], która umożliwiła pokazanie, w jaki sposób operatory `shift` i `reset` mogą symulować `shift0` i `reset0`. Zarówno pokazana translacja, jak i symulacja wymagają, by kontynuacje były reprezentowane jako funkcje rekurencyjne, które jako jeden z argumentów przyjmują listę kontynuacji.

Kiselyov i Shan [114] zaproponowali substrukturalny system typów dla operatorów `shift0` and `reset0`, który pozwala na abstrakcyjną interpretację (w sensie formalizmu Cousot i Cousot [49]) ich semantyki małych kroków. Ten system typów wspiera modyfikację typu odpowiedzi kontynuacji i w pewnym sensie jest rozszerzeniem systemu typów Danvy'ego i Filinskiego.

W pracy [134] zaprezentowaliśmy nowy system typów z efektami dla operatorów `shift0` i `reset0`, który stanowi uogólnienie systemu typów Danvy'ego i Filinskiego. W tym celu najpierw zdefiniowaliśmy dwie nowe translacje do CPS dla tych operatorów: w wersji rozwiniętej (ang. curried), która jest konserwatywnym rozszerzeniem translacji Plotkina dla strategii gorliwej, i w wersji zwiniętej (ang. uncurried), która wymaga, by język docelowy obsługiwał operacje na listach. Z translacji w wersji zwiniętej wyprowadziliśmy nową maszynę abstrakcyjną, która wraz ze standardową semantyką redukcyjną stanowi operacyjną podstawę teoretyczną dla operatora `shift0`. Z kolei z wersji rozwiniętej wyprowadziliśmy wspomniany system typów z efektami à la Danvy i Filinski, który dodatkowo poprawiliśmy poprzez dodanie możliwości podtypowania efektów. Pokazaliśmy, że ten system typów spełnia pożądane własności, takie jak silna poprawność typów [204] oraz terminacja ewaluacji (jak wspomniano w sekcji 4.3.2.1), a ponadto opisaliśmy i udowodniliśmy poprawność algorytmu inferencji typów w tym systemie.

W porównaniu do systemu typów Kiselyova i Shana, nasz jest bardziej konwencjonalny: odpowiada bezpośrednio translacji do CPS, z której został wyprowadzony i istotnie korzysta z mechanizmu podtypowania. Ponadto wykorzystując fakt, że w naszym systemie wyrażenia czyste i wyrażenia z efektami są rozróżniane, pokazaliśmy selektywną translację do CPS opartą na typach, która tłumaczy termy z języka z operatorami `shift0` i `reset0` do rachunku lambda z typami prostymi. Ta translacja przekształca tylko wyrażenia z efek-

tami, natomiast nie zmienia wyrażeń czystych (bez efektów). Ma ona praktyczne znaczenie, ponieważ może być wykorzystana do implementacji rozważanych operatorów w istniejących językach programowania, takich jak Scala [166].

Ponadto pokazaliśmy, że oryginalny system typów Danvy'ego i Filinskiego dla operatorów `shift` i `reset` można w naturalny sposób zanurzyć w naszym systemie typów, oraz że pewne warunki nałożone na nasz system typów pozwalają uzyskać system typów z podtypowaniem à la Danvy i Filinski dla operatorów `shift` i `reset`. Podobnie, beztypowa translacja do CPS dla `shift0` i `reset0`, którą zdefiniowaliśmy, prowadzi do otrzymania nowej translacji do CPS dla operatorów `shift` i `reset` oraz do nowej symulacji operatorów `shift0` i `reset0` za pomocą operatorów `shift` i `reset`.

W pewnym sensie można uważać typy kontynuacyjne, które badaliśmy w pracy [23] za bardziej ogólne niż typy funkcji: kontynuacje mogą być aplikowane w dowolnym kontekście, natomiast funkcje mogą być aplikowane tylko w kontekstach o zgodnym typie odpowiedzi. System wprowadzony przez nas w pracy [134] usuwa syntaktyczne rozróżnienie między kontynuacjami i zwykłymi funkcjami poprzez podtypowanie efektów. Relacja podtypowania pozwala na wywołanie funkcji w sytuacji, gdy wiadomo więcej o typach kontekstu w meta-kontekście niż funkcja faktycznie wymaga. W szczególności, każda czysta funkcja – być może reprezentująca przechwyconą kontynuację – może być zawsze potraktowana jako funkcja z efektami.

Ponieważ `shift0` i `reset0` mogą dowolnie eksplorować i manipulować stosem kontekstów, naturalnie pojawia się pytanie o ich związek z hierarchią CPS. W pracy [135] odpowiedzieliśmy na to pytanie pokazując, że za pomocą operatorów `shift0` i `reset0` można w pełni wyrazić hierarchię CPS. W tym celu pokazaliśmy formalny związek między semantyką operacyjną, translacjami do CPS oraz systemami typów dla tych operatorów. Ponadto pokazaliśmy kilka typowych przykładów programów w hierarchii CPS zapisanych za pomocą tych operatorów oraz jeden przykład, który wykracza poza ramy hierarchii.

Przedstawione wyniki demonstrują znaczącą siłę wyrazu operatorów `shift0` i `reset0`, a ponadto sugerują nowe spojrzenie na teorię i praktykę programowania w hierarchii CPS, dostarczając metod wnioskowania oraz technik implementacji dla hierarchii przy użyciu operatorów `shift0` i `reset0`.

Praca, której wyniki zaprezentowaliśmy w artykułach [134, 135] była następnie kontynuowana przez Materzoka, który pod moją opieką promotorską opracował poprawną i pełną aksjomatyzację dla operatorów `shift0` i `reset0` w wersji zarówno typowanej (z użyciem podtypowania), jak i beztypowej [132], jak również wykonał ich niskopoziomową implementację w języku C. Pełne studium operatorów `shift0` i `reset0` zostało następnie zawarte w rozprawie doktorskiej Materzoka [133].

Osiągnięte przez nas wyniki dotyczące operatorów `shift0` i `reset0` były intensywnie wykorzystywane przez Downena i Ariolę w ich pracy nad rachunkiem $\Lambda\mu\tilde{\mu}_v$ [73, 75, 74], jak również przez Munch-Macagnoniego w pracy nad polaryzacją w pewnych wariantach rachunku $\Lambda\mu\hat{\mathtt{tp}}$ [142].

4.3.2.3 Relacje logiczne i koherencja podtypowania efektów

Semantyka języków programowania wyposażonych w mechanizm podtypowania (umożliwiający koercję typu wyrażenia do innego typu) zwykle zadawana jest za pomocą semantyki podzbiorów, gdzie dany typ może być uważany za podzbiór innego typu, albo za pomocą semantyki koercji, w której wyrażenia są jawnie konwertowane z jednego typu do drugiego. W językach z podtypowaniem drzewo wyprowadzenia typu zwykle zawiera wystąpienia sądów o koercjach, a zatem sądy typowe nie są jednoznacznie wyznaczone przez drzewa wyprowadzenia typu. Co za tym idzie, semantyka koercji, która interpretuje sądy o koercjach poprzez jawne operacje koercji między typami, jest definiowana na drzewach wyprowadzenia zamiast na sądach typowych. W takiej sytuacji naturalnie pojawia się pytanie o koherencję takiej semantyki, tj. czy semantyka wyrażenia nie zależy od drzewa wyprowadzenia typu.

Problem koherencji był rozważany dla rozmaitych wariantów typowanego rachunku lambda. Reynolds udowodnił koherencję semantyki denotacyjnej dla systemu typów iloczynowych [162]. Breazu-Tannen i in. udowodnili koherencję translacji z rachunku lambda z typami polimorficznymi, rekurencyjnymi i wariantowymi do systemu F [45] pokazując, że dowolne dwa wyprowadzenia tego samego sądu typowego da się znormalizować do wspólnej postaci normalnej, a poprawność normalizacji wynika z teorii równościowej w rachunku docelowym. Curien i Ghelli wprowadzili translację z systemu $F_{\leq}$ do rachunku z jawnymi koercjami i pokazali, że dowolne dwa wyprowadzenia tego samego sądu typowego są tłumaczone przez tę translację na termy, które są normalizowalne do jednoznacznej postaci normalnej [50]. Wreszcie Schwinghammer wykorzystał podejście Breazu-Tannen i in. do pokazania koherencji dla translacji koercji z obliczeniowego rachunku lambda Moggiego z podtypowaniem [177].

Dowody oparte na normalizacji polegają na znajdowaniu postaci normalnych dla reprezentacji wyprowadzenia, a następnie na pokazaniu, że taka postać normalna jest jednoznacznie wyznaczona dla danego sądu typowego. Kiedy rachunek źródłowy jest przedstawiony w stylu rachunku lambda à la Church, tzn. abstrakcje są udekorowane typami, jak to ma miejsce w przypadku przywołanych wyżej prac wykorzystujących tę metodę, wówczas term i kontekst typowania jednoznacznie wyznaczają kształt wyprowadzenia normalnego (z wyjątkiem być może koercji na typie całego termu) [140]. Jednak w rachunkach à la Curry taka własność nie zachodzi i metody opartej na normalizacji nie można bezpośrednio zastosować. Mimo tego, jeśli rachunek ma własność przynajmniej słabej normalizacji, wówczas można mieć nadzieję na uzyskanie jednoznaczności postaci normalnej dla wyprowadzenia typu dla termów w postaci normalnej, przy założeniu, że normalizacja termów zachowuje semantykę koercji. Na przykład w przypadku rachunku lambda z typami prostymi kontekst typowania jednoznacznie określa typ termu w pozycji funkcji dla aplikacji tworzących postaci β-normalne, a zatem wyprowadzenia normalne dla takich termów są wyznaczone jednoznacznie. Tego rozumowania nie można jednak zastosować w przypadku, gdy w języku występuje rekursja.

W pracy [42] rozważaliśmy problem koherencji dla rachunku lambda z typami prostymi wyposażonego w rekursję ogólną, efekty sterowania oraz bez żadnych dodatkowych informacji o typach dla termów, przy założeniu, że semantyka koercji przekłada wyprowadzenia typów w języku źródłowym do odpowiadającego mu języka docelowego z jawną koercją

(która w większości przypadków może zostać zastąpiona przez odpowiednie reprezentacje w postaci termów rachunku lambda). Użytym przez nas kryterium koherencji dla takiej translacji jest kontekstowa równoważność w rachunku docelowym [141].

Głównym uzyskanym przez nas wynikiem jest konstrukcja relacji logicznych, dzięki którym udowodniliśmy tak rozumianą koherencję selektywnej translacji do CPS sterowanej typami, z gorliwego rachunku lambda z operatorami sterowania dla kontynuacji ograniczonych i podtypowaniem efektów z wcześniej przytoczonej pracy [134], wyposażonego dodatkowo w rekursję. Chociaż Materzok pokazał wcześniej koherencję translacji dla rachunku z jawnymi oznaczeniami typów korzystając z teorii równościowej w rachunku docelowym [132], taka własność nie została pokazana dla translacji CPS oryginalnego języka, nie mówiąc już o rozszerzeniu o rekursję.

Rozważany przez nas język źródłowy, jak i sama translacja, są dość skomplikowane, przez co wymagają wprowadzenia odpowiednio złożonych binarnych relacji logicznych, które są: heterogeniczne, biortogonalne [117, 156, 77] i wyposażone w indeksowanie krokowe (ang. step-indexed) [8, 7, 6]. Heterogeniczność pozwala na powiązanie ze sobą termów różnych typów, w szczególności termów w stylu kontynuacyjnym z ich bezkontekstowymi odpowiednikami (ang. direct-style terms). Ta własność jest kluczowa, ponieważ ten sam term może mieć czysty typ, który nie jest przekształcany do stylu kontynuacyjnego przez naszą translację, a jednocześnie inny typ (z efektami), który powoduje translację do stylu kontynuacyjnego. Zdefiniowanie relacji między takimi termami wymaga kwantyfikowania po typach, więc żeby cała konstrukcja była dobrze określona, wprowadziliśmy indeksowanie krokowe, które ponadto wspiera wnioskowanie o rekursji. Sposób, w jaki używamy indeksowania krokowego jest inspirowany podejściem Dreyera i in. [76]. Z kolei biortogonalność, która wymusza pewien ustalony porządek ewaluacji wyrażeń, pozwala na uproszczenie konstrukcji relacji logicznych, jak również umożliwia wnioskowanie o kontynuacjach reprezentowanych w postaci kontekstów ewaluacyjnych.

Poza rozważaniem rachunku z podtypowaniem efektów, podejście zaprezentowane w pracy [42] zastosowaliśmy również do pokazania koherencji podtypowania dla szeregu innych rachunków, m. in. rachunku lambda z typami prostymi i podtypowaniem [140] wyposażonego w rekursję, dla rachunku z typami iloczynowymi [162], oraz rachunku lambda z podtypowaniem i operatorem `callcc`.

4.3.2.4 Zastosowanie bisymulacji do dowodzenia równoważności programów z operatorami sterowania

Translacje CPS dla języków źródłowych z operatorami sterowania, takimi jak `shift` i `reset`, do języka docelowego bez efektów są bez wątpienia narzędziem niezwykle przydatnym, ale w wielu sytuacjach pożądane są narzędzia operujące bezpośrednio na kodzie źródłowym, np. do wnioskowania o równoważności programów. W tym celu Kameyama i in. zaksjomatyzowali szereg języków z operatorami sterowania dla kontynuacji ograniczonych [109, 107, 110] i pokazali, że ich aksjomaty są poprawne i pełne względem odpowiednich translacji do CPS. Istnieje wiele innych prac dotyczących wnioskowania równościowego w rachunkach z kontynuacjami ograniczonymi [167, 10, 100, 132], pokazujących, że ten temat jest aktywnie badany.

Teorie równościowe dla operatorów sterowania, takich jak `shift` i `reset`, wynikają naturalnie z ich translacji do CPS, lecz w wielu przypadkach nie są one wystarczające do udowodnienia równoważności programów operacyjnie nierozróżnialnych (np. takich jak dwa różne kombinatory stałopunktowe). Zatem w celu zbudowania silniejszych narzędzi do wnioskowania o równoważności programów z kontynuacjami ograniczonymi skorzystaliśmy z podstaw operacyjnych dla takich operatorów opracowanych przez nas wcześniej [81, 24] i zajęliśmy się badaniem kryteriów równoważności programów, które można wyrazić w terminach operacyjnych.

Za najbardziej naturalne pojęcie równoważności programów opartych na rachunku lambda uważa się *równoważność kontekstową* [141]. Intuicyjnie oznacza ona, że dwa programy są równoważne wtedy, gdy można zastąpić jeden przez drugi w dowolnym większym programie bez zmiany zachowania tego większego programu. Zachowanie programu można opisywać za pomocą *obserwacji*, które chcemy wziąć pod uwagę dla danego języka, np. operacje wejścia/wyjścia dla komunikujących się systemów [172], odczyt i zapis do pamięci, itp. W przypadku czystego rachunku lambda obserwacją jest zwykle terminacja programu (czy program się zatrzymuje) [3]. „Większy program", o którym mowa wyżej, można rozważać jako kontekst (czyli term z dziurą), a zatem mówimy, że dwa termy t_0 i t_1 są równoważne kontekstowo, jeśli nie potrafimy ich rozróżnić w żadnym kontekście C, tzn. gdy programy $C[t_0]$ i $C[t_1]$ zwracają te same obserwacje.

Występująca w tej definicji kwantyfikacja po kontekstach C sprawia, że równoważność kontekstowa jest trudna do zastosowania w praktyce do udowodnienia, że dwa konkretne termy są równoważne. Zwykle zatem rozważa się łatwiejsze w użyciu alternatywne formalizmy, takie jak relacje logiczne (p. [157]), aksjomatyzacje [121] i bisymulacje. Bisymulacja ustala relację między dwoma termami t_0 i t_1 wymuszając, by każdy z nich umiał „naśladować" drugi term koindukcyjnie, tzn. jeśli t_0 redukuje się do t'_0, to t_1 musi redukować się do t'_1 takiego, że t'_0 i t'_1 są również w tej relacji bisymulacji (i symetrycznie dla redukcji termu t_1). Relacja równoważności na termach, zwana bipodobieństwem (ang. bisimilarity), jest definiowana na podstawie pojęcia bisymulacji w następujący sposób: dwa termy są bipodobne, jeśli istnieje bisymulacja, do której oba należą (bipodobieństwo jest największą bisymulacją). Zbudowanie właściwej teorii bisymulacji polega na znalezieniu warunków, przy których otrzymana relacja bipodobieństwa będzie *poprawna* i *pełna* względem równoważności kontekstowej, tzn. odpowiednio: będzie w niej zawarta i będzie ją zawierała.

Istnieją różne rodzaje bisymulacji dla języków opartych na rachunku lambda. Na przykład bisymulacje aplikatywne [3] ustalają relację między termami przez ich redukcję do wartości (jeśli istnieje) i wymóg, żeby otrzymane wartości zastosowane do dowolnego argumentu były z kolei termami należącymi do tej relacji bisymulacji. Jak widać z tej definicji, bisymulacje aplikatywne wymagają kwantyfikowania po argumentach w celu porównania wartości, są jednak znacznie łatwiejsze w użyciu niż równoważność kontekstowa dzięki swojej koindukcyjnej naturze, oraz dlatego, że nie musimy w nich rozważać wszystkich możliwych rodzajów kontekstów. Bisymulacje aplikatywne są zwykle poprawne i pełne względem równoważności kontekstowej, przynajmniej w przypadku języków deterministycznych, takich jak czysty rachunek lambda [3].

W przeciwieństwie do bisymulacji aplikatywnych, bisymulacje postaci normalnej (ang. normal-form bisimulation) [123], znane również jako bisymulacje otwarte [168], nie wyma-

gają kwantyfikowania po argumentach ani kontekstach. W tym podejściu termy są redukowane do postaci normalnej (jeśli istnieje), a następnie dekomponowane na podtermy, które również muszą być w tej samej relacji. W przeciwieństwie do bipodobieństwa aplikatywnego, generowane w ten sposób bipodobieństwo postaci normalnej zwykle nie jest relacją pełną, tj. istnieją termy kontekstowo równoważne, które nie są powiązane przez bipodobieństwo postaci normalnej. Jednak dzięki temu, że nie używa się tu żadnej kwantyfikacji, zwykle dość łatwo jest pokazać, że dwa termy są biopdobne w ten sposób, a same dowody mogą być jeszcze upraszczane za pomocą tzw. technik przybliżania (ang. up-to techniques). Są to takie techniki, w których definiuje się relacje, które nie są formalnie bisymulacjami, ale zawierają się w nich. Znajdowanie przybliżonej relacji ustalającej równoważność dwóch termów jest zwykle łatwiejsze niż znajdowanie właściwej bisymulacji.

Trzecim rodzajem bisymulacji są bisymulacje środowiskowe [170], które są podobne do aplikatywnych w tym, że termy są porównywane przez redukcję do wartości, a następnie otrzymane wartości są porównywane po zaaplikowaniu do pewnych argumentów. Różnica jednak polega na tym, że argumenty dostarczane do wartości nie są dowolne, ale pochodzą ze środowiska, które reprezentuje wiedzę na temat testowanych termów zgromadzoną dotychczas przez zewnętrznego obserwatora. Podobnie do bipodobieństwa aplikatywnego, bipodobieństwo środowiskowe jest zwykle relacją poprawną i pełną, ale umożliwia również użycie technik przybliżania (tak, jak bipodobieństwo postaci normalnej) w celu upraszczania dowodów równoważności. Natomiast zdefiniowanie praktycznych technik przybliżania dla bipodobieństwa aplikatywnego jest problemem otwartym.

Efekty sterowania mają dość złożoną naturę, co powoduje, że zagadnienie rozstrzygnięcia czy dane dwa programy z operatorami sterowania są operacyjnie równoważne jest trudne. Opracowanie poprawnych, pełnych i praktycznie użytecznych bisymulacji dla operatorów sterowania było jednym z głównych problemów naukowych, którymi zajmowałem się w ostatnich latach. W dalszej części pokrótce podsumuję otrzymane w tej dziedzinie wyniki: pełna behawioralna teoria dla kontynuacji ograniczonych oraz pierwsze poprawne i pełne bisymulacje dla kontynuacji abortywnych.

Kontynuacje ograniczone W serii prac [38, 39, 40], zbudowaliśmy pierwszą w literaturze behawioralną teorię rachunku lambda rozszerzonego o operatory sterowania dla kontynuacji ograniczonych `shift` i `reset`. W naszych badaniach rozważaliśmy semantykę dwojakiego rodzaju: *oryginalną* (wyprowadzoną z interpretera definiującego `shift` i `reset` [24]), w której zakłada się, że ewaluowane termy zawierają zewnętrzny reset (są ograniczone) oraz *liberalną* (częściej stosowaną w implementacjach), przy której ten wymóg nie występuje, co wymaga uwzględnienia termów niedomkniętych ze względu na sterowanie (ang. control stuck term). Dla każdej z semantyk zdefiniowaliśmy pojęcie kontekstowej równoważności, a następnie podjęliśmy próbę scharakteryzowania jej w terminach różnych rodzajów bisymulacji (aplikatywnych, postaci normalnej i środowiskowych). Otrzymane relacje poddaliśmy również wnikliwemu porównaniu z relacją równoważności CPS, która zrównuje termy o $\beta\eta$-równych obrazach przez CPS translację, scharakteryzowaną przez aksjomatyzację Kameyamy i Hasegawy [109].

Istnieje bardzo niewiele prac poświęconych bisymulacjom aplikatywnym dla rachunków

z efektami sterowania. W pracy [137] Merro i Biasi definiują aplikatywne bisymulacje charakteryzujące kontekstową równoważność w rachunku CPS [196] – minimalnym rachunku zawierającym struktury sterowania znane z języków funkcyjnych z imperatywnymi skokami. Z kolei w pracy [122], Lassen definuje poprawne, ale niepełne aplikatywne bisymulacje dla rachunku $\lambda\mu$ w wersji leniwej.

W pracy [38] przedstawiliśmy poprawne i pełne bisymulacje dla rachunku lambda przy ewaluacji gorliwej, rozszerzonego o operatory `shift` i `reset`, przy liberalnej semantyce. Zaprezentowana technika bisymulacji została oparta na nieznanej wcześniej etykietowanej semantyce małych kroków. Wykorzystując adaptację metody Howe'a dowodzenia kongruencji bisymulacji [101], udowodniliśmy, że nasze bisymulacje są poprawne i pełne względem kontekstowej równoważności. Dodatkowo udowodniliśmy lemat kontekstowy Milnera [139] dla rozważanego rachunku. Omówiliśmy również szczegółowo związek naszej teorii z równoważnością CPS.

Pojęcie bisymulacji postaci normalnej zdefiniowano dla całego szeregu wariantów rachunku lambda z efektami sterowania, w tym dla rachunku lambda z operatorem `amb` [124], dla rachunku $\lambda\mu$ [125], a także dla rachunku $\lambda\mu\rho$ [187], w którym bisymulacje postaci normalnej w pełni charakteryzują równoważność kontekstową. W pracy [39], zaproponowaliśmy kilka podejść do bisymulacji postaci normalnej dla rachunku lambda w wersji gorliwej, rozszerzonego o operatory `shift` i `reset`, przy liberalnej semantyce, które okazują się bardziej praktyczne niż samo pojęcie kontekstowej równoważności czy bisymulacje aplikatywne. Udowodniliśmy, że nasze bisymulacje postaci normalnej są poprawne względem równoważności kontekstowej, ale nie są pełne. Zaproponowaliśmy też pewne techniki przybliżania, które okazują się niezwykle pomocne w dowodzeniu równoważności programów z użyciem bisymulacji postaci normalnej.

W pracy [40] uzupełniliśmy naszą teorię bisymulacji dla operatorów `shift` i `reset` o bisymulacje środowiskowe i to zarówno przy semantyce liberalnej, jak i oryginalnej. W obu przypadkach pokazaliśmy poprawność i pełność otrzymanych bisymulacji, a także wprowadziliśmy standardowe techniki przybliżania (m. in. przybliżanie z dokładnością do środowiska i kontekstu [170]), demonstrując możliwości ich zastosowania. Według naszej wiedzy praca ta jest pierwszym w literaturze (udanym) podejściem do bisymulacji środowiskowych dla języka z operatorami sterowania.

Każda z technik bisymulacji zdefiniowanych przez nas ma swoje mocne i słabe strony, ale wszystkie trzy wspólnie tworzą poprawną, pełną i dość praktyczną teorię pozwalającą dowodzić równoważności programów używających kontynuacji ograniczonych. Zaletą bisymulacji postaci normalnej jest niewątpliwie łatwość ich użycia, która prowadzi do względnie prostych dowodów równoważności programów, przede wszystkim dlatego, że nie wymagają one w swojej definicji kwantyfikowania po argumentach funkcji czy kontekstach ewaluacyjnych. Co więcej, bisymulacje postaci normalnej można łączyć z technikami przybliżania, które znacząco upraszczają konstruowane relacje. Jednakże, bisymulacje postaci normalnej są niewystarczające ze względu na to, że nie są pełne i w wielu przypadkach rozróżniają nawet bardzo proste kontekstowo równoważne termy. Dodatkowo, bisymulacje postaci normalnej są zdefiniowane na termach otwartych (zawierających wolne zmienne), co prowadzi do konieczności rozważania dodatkowych postaci normalnych w konstruowanych bisymulacjach. Bisymulacje aplikatywne i środowiskowe nie posiadają tych wad: są pełne i operują

na termach zamkniętych. W rezultacie, część dowodów okazuje się prostsza przy użyciu bisymulacji aplikatywnych niż postaci normalnej, z możliwością dalszych optymalizacji przy użyciu środowiskowych bisymulacji z techniką przybliżania z dokładnością do kontekstu. Dodatkowo, pokazanie równoważności termów niedomkniętych ze względu na sterowanie z innymi, np. będącymi wartością, jest możliwe wyłącznie przy użyciu bisymulacji środowiskowych dla semantyki oryginalnej.

Kontynuacje abortywne W pracy [41] zaprezentowaliśmy behawioralną teorię rachunku $\lambda\mu$ [149]. Typowana wersja rachunku $\lambda\mu$ dostarcza obliczeniową interpretację systemu klasycznej dedukcji naturalnej i w związku z tym stanowi rozszerzenie izomorfizmu Curry'ego-Howarda z logiki intuicjonistycznej do klasycznej. Operacyjnie, reguły redukcji w rachunku $\lambda\mu$ wyrażają nie tylko aplikacje funkcji ale również operacje przechwycenia bieżącej kontynuacji. Stąd też, rozważany w ujęciu beztypowym rachunek $\lambda\mu$ wyraża semantykę sterowania operatorów dla kontynuacji abortywnych, takich jak `callcc` z języka Scheme. Jest on również blisko związany z rachunkiem kontynuacji abortywnych Felleisena i Hieba [84].

Żadna z istniejących wcześniej w literaturze prac nie prezentowała pełnej względem równoważności kontekstowej teorii bisymulacji dla rachunku $\lambda\mu$ przy jakiejkolwiek strategii ewaluacji. Lassen zdefiniował bisymulacje postaci normalnej dla redukcji do słabej czołowej postaci normalnej przy strategii leniwej [122], dla redukcji do postaci czołowej [125] oraz, wspólnie ze Støvringiem, dla redukcji do słabej czołowej postaci normalnej przy gorliwej strategii ewaluacji [187], z których żadne nie są pełne. Okazuje się jednak, że bisymulacje postaci normalnej są pełne dla rachunku $\Lambda\mu$ [66] (wariant rachunku $\lambda\mu$) przy redukcji do czołowej postaci normalnej [125], jak również dla rachunku $\lambda\mu$ z mutowalnym stanem [187] przy redukcji do słabej czołowej postaci normalnej i strategii gorliwej. Lassen zdefiniował też niepełne aplikatywne bisymulacje dla rachunku $\lambda\mu$ przy redukcji do słabej czołowej postaci normalnej i strategii leniwej [122]. Zaproponowano również definicję aplikatywnych bisymulacji dla typowanego rachunku μPCF w wersji gorliwej [147], ale okazała się ona niepoprawna (i niepełna).

W naszej pracy przedstawiliśmy pierwsze w literaturze bisymulacje (aplikatywne oraz środowiskowe) dla rachunku $\lambda\mu$, które są poprawne i pełne, przy ewaluacji do słabej czołowej postaci normalnej, zarówno w wersji leniwej, jak i gorliwej. Zaproponowane bisymulacje są trudniejsze w użyciu niż bisymulacje postaci normalnej Lassena, ale ze względu na to, że są pełne, pozwalają na udowodnienie równoważności termów, które definicja Lassena rozróżnia, włączając tak ważne przypadki, jak klasyczny już kontrprzykład Davida i Py [64].

Podstawowym powodem, dla którego skonstruowane przez nas bisymulacje są pełne, w przeciwieństwie do pracy Lassena [122], jest szczególne potraktowanie tzw. termów nazwanych jako pierwotnych w rachunku, a następnie rozszerzenie definicji na pozostałe termy. Nazwy reprezentujące kontynuacje pozwalają na kontrolowanie tego, w jaki sposób konteksty ewaluacyjne zostają przechwycone i użyte w porównywanych termach. W tej sytuacji wystarczy testować zachowanie nazwanych wartości w kontekstach elementarnych [84] (reprezentujących pojedyncze ramki stosu).

Ze względu na to, że struktura kontekstów elementarnych przy strategii gorliwej jest

bogatsza niż przy strategii leniwej, technika bisymulacji aplikatywnych w wersji gorliwej jest trudniejsza w użyciu niż ta w wersji leniwej. W naszej pracy wskazujemy przykłady, które pokazują, że uproszczenie bisymulacji w wersji gorliwej tak by były podobne do tych w wersji leniwej, prowadzi do niepoprawnych definicji.

Mamy podstawy sądzić, że nasze podejście może być dość łatwo zaadaptowane, zachowując pełność bisymulacji, do innych wariantów rachunku $\lambda\mu$ takich jak $\lambda\mu$ z alternatywnym zestawem reguł redukcji [64], rachunek $\lambda\mu$ w wersji typowanej [149], czy też rachunek $\Lambda\mu$ de Groote'a [66].

5 Omówienie pozostałych osiągnięć naukowo-badawczych

Poza moimi głównymi badaniami, których wyniki zostały omówione w sekcji 4, po uzyskaniu stopnia doktora byłem zaangażowany w szereg innych projektów naukowych dotyczących podstaw i implementacji języków programowania, przeprowadzonych wspólnie z moimi współpracownikami oraz studentami. Projekty te w głównej mierze skupiają się na formalizacji semantyki operacyjnej języków funkcyjnych, jak również ich wariantów i rozszerzeń, na derywacyjnym podejściu do konstrukcji maszyn abstrakcyjnych z zapewnieniem ich poprawności, a także na formalizacji modelowych kompilatorów dla synchronicznych języków funkcyjnych (ang. synchronous data-flow languages). Większość z tych projektów została sformalizowana w teorii typów asystenta dowodzenia Coq, którego język taktyk Ltac również był przedmiotem naszych badań jako język programowania.

W dalszej części tej sekcji znajduje się krótki opis każdego z osiągniętych wyników; dla każdego z nich podana jest motywacja, ogólny opis oraz jego kontekst, tj. opis literatury ściśle związanej z danym zagadnieniem. Szczegóły techniczne można znaleźć we wskazanych niżej publikacjach.

Sekcja 5.1 jest poświęcona modularnej kompilacji synchronicznego języka funkcyjnego:

5.1: Formalizacja kompilatora synchronicznego języka funkcyjnego – wyniki zaprezentowane w pracach [33, 32];

sekcja 5.2 zawiera omówienie wyników dotyczących konstrukcji maszyn abstrakcyjnych dla języków programowania wyższego rzędu:

5.2.1: Refocusing w systemie Coq – wyniki zaprezentowane w pracach [21, 181];

5.2.2: Formalizacja maszyny abstrakcyjnej STG – wyniki zaprezentowane w pracy [155];

natomiast sekcja 5.3 dotyczy formalnej semantyki języka taktyk w systemie Coq:

5.3: Semantyka formalna języka Ltac (języka taktyk w systemie Coq) – wyniki zaprezentowane w pracy [105].

5.1 Formalizacja kompilatora synchronicznego języka funkcyjnego

5.1.1 Publikacje

[33] Dariusz Biernacki, Jean-Louis Colaço, Marc Pouzet. Clock-directed modular code generation from synchronous block diagrams. *2007 Workshop on Automatic Program*

Generation for Embedded Systems (APGES 2007), 12–20, Salzburg, Austria, 2007.

[32] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hammon, Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. *2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*, 121–130, Tucson, USA, 2008.

5.1.2 Opis wyników

Formalne podejście do diagramów blokowych takie jak w językach Scade/Lustre [174] czy Simulink [182] jest powszechnie spotykane w systemach wbudowanych. W szczególności, synchroniczne diagramy blokowe oparte są na dyskretnym modelu czasu, w którym sygnały są reprezentowane jako nieskończone strumienie danych, a bloki przetwarzające sygnały definiują funkcje na takich strumieniach. Generowanie sekwencyjnego imperatywnego kodu dla synchronicznych diagramów blokowych stanowi znane zagadnienie, któremu poświęcono uwagę już we wczesnym okresie istnienia języka Lustre [46], a temat ten jest obecnie uważany za element folkloru w dziedzinie programowania synchronicznego [15].

Dla danej funkcji na strumieniach f oraz równania $y = f(x)$, generowanie kodu polega na wyprodukowaniu funkcji przejścia f_t oraz stanu początkowego s_0 takich, że $\forall n \in \mathbb{N}.(y_n, s_{n+1}) = f_t(s_n, x_n)$, jeśli $x = (x_i)_{i\in\mathbb{N}}$ i $y = (y_i)_{i\in\mathbb{N}}$. Funkcja przejścia przyjmuje jako swoje argumenty stan oraz bieżący element strumienia wejściowego, a zwraca bieżący element strumienia wyjściowego oraz nowy stan. Nieskończona iteracja funkcji przejścia produkuje strumień wyjściowy. W praktycznych implementacjach funkcja przejścia jest zrealizowana imperatywnie z użyciem mutowalnej pamięci do reprezentowania stanu. Synchronizacja znajduje w takim podejściu praktyczne uzasadnienie: nieskończony strumień wartości typu T jest reprezentowany za pomocą skalarnej wartości typu T, bez konieczności użycia dodatkowej pamięci czy złożonych mechanizmów buforowania. Zasada ta zostaje uogólniona na funkcje operujące na więcej niż jednym strumieniu wejściowym oraz wyjściowym. Generowanie kodu otrzymuje się w wyniku statycznego uporządkowania równań zgodnie z występującymi zależnościami między danymi. Niezależne czy też modularne generowanie kodu ma na celu wyprodukowanie funkcji dla każdej definicji bloku w diagramie oddzielnie oraz złożenie otrzymanych funkcji tak, by otrzymać główną funkcję przejścia. Okazuje się, że nawet w przypadku braku nierozwiązywalnych zależności przyczynowych między blokami (ang. causality loop), modularne generowanie kodu nie zawsze jest wykonalne.

Istnieją w literaturze dwa podejścia do kompilacji języków synchronicznych. Pierwsze z nich ma na celu uzyskanie maksymalnie dużej siły wyrazu w języku źródłowym i polega na kompilacji programu po pełnym wstawieniu (ang. inlining) wywołań funkcji. Otrzymany zestaw równań może być następnie przetłumaczony do kodu imperatywnego za pomocą technik takich, jak wyliczenie w przód lub wstecz zmiennych stanu, by otrzymać jawny automat skończony, co prowadzi do bardzo wydajnego kodu [46, 94]. Niestety, podejście to ma swoją cenę – wyklucza kompilację modularną, a co więcej rozmiar wygenerowanego kodu może w praktyce być ogromny. Z tego powodu wyliczenie musi zostać ograniczone do pewnego wybranego podzbioru zmiennych stanu, na przykład tak, jak w akademickim

kompilatorze języka Lustre [192]. Wybranie właściwego podzbioru zmiennych stanu okazuje się jednak niepraktyczne zarówno ze względu na złożoność czasową, jak i pamięciową.

Innym podejściem jest to występujące w kompilatorach przemysłowych, m. in. w kompilatorze języka Scade, w których, z oczywistych względów, podstawowym założeniem jest modularność procesu kompilacji. W takich kompilatorach nie występuje wstawienie wywołań funkcji, chyba, że na żądanie programisty, ale za to narzuca się silne wymogi przyczynowości między blokami, zwykle polegające na tym by każda pętla w diagramie zależności danych była jawnie przesunięta (opóźniona). Wymogi te zostały zaakceptowane przez użytkowników języka Scade, ale również, wpływając na uproszczenie procesu generowania kodu, pozwoliły spełnić wymagania władz certyfikujących oprogramowanie w zastosowaniach, w których bezpieczeństwo jest elementem najważniejszym (komputery pokładowe w samochodach czy samolotach, systemy sterujące elektrowniami atomowymi etc.) Uzupełnieniem kompilacji modularnej, np. jako krok wstępny, jest dekompozycja ciała funkcji na pewną minimalną liczbę funkcji, z których każda może być przetłumaczona na atomową funkcję przejścia [160].

Modularna kompilacja synchronicznych diagramów blokowych, mimo, że powszechnie używana w przemysłowych kompilatorach języka Lustre nie była wcześniej ani precyzyjnie opisana, ani sformalizowana w całości. Ze względu na zastosowania języka Lustre/Scade taka formalizacja wydaje się niezwykle potrzebna, szczególnie jako punkt wyjścia dla długoterminowego celu jakim jest zbudowanie matematycznie zweryfikowanego kompilatora synchronicznego języka funkcyjnego w systemie Coq, ale również jako podstawa do rewizji i usprawnień istniejących implementacji. Co więcej, formalnie zdefiniowany kompilator mógłby w przyszłości wpłynąć na poprawę procesu certyfikacji krytycznego ze względu na bezpieczeństwo oprogramowania. Mając na myśli wszystkie wyżej wymienione cele, w pracach [33, 32] zaprezentowaliśmy minimalny, ale pełny opis generatora kodu, który może stanowić najważniejsze ogniwo w łańcuchu kompilacji programów synchronicznych. Obie prace szczegółowo przedstawiają modularną kompilację synchronicznych diagramów blokowych do kodu sekwencyjnego.

Językiem źródłowym, który rozważamy jest deklaratywny język pierwszego rzędu przypominający Lustre, wystarczająco ogólny, by mógł pełnić rolę odpowiedniego języka pośredniego dla kompilacji automatów [131, 47]. Język zawiera ogólny operator `merge` pozwalający scalać uzupełniające się strumienie, operator `reset` pozwalający na restart poszczególnych komponentów systemu w modularny sposób, a także uogólnione pojęcie zegara używane do wyrażania rozmaitych warunków aktywacji komponentów. W prezentowanym przez nas kompilatorze kluczową rolę odgrywa obiektowy język pośredni służący do reprezentowania sekwencyjnych funkcji przejścia, które następnie tłumaczone są do języków Java oraz C. Programy synchroniczne są tłumaczone modularnie do programów w języku pośrednim z wykorzystaniem zegarów odgrywających główną rolę w zapewnieniu wydajności generowanych struktur sterowania.

Nasze podejście znacząco różni się od klasycznych metod kompilacji opartych na technikach związanych z wyliczaniem zmiennych. Użycie odpowiedniego języka pośredniego, a także wykorzystanie zegarów prowadzi do zwięzłego opisu procesu kompilacji, produkując wydajny kod sekwencyjny, który może rywalizować z tym otrzymanym z użyciem kompilatorów przemysłowych. Kompilator, opisany w naszych pracach, został zaimplementowany

w języku OCaml, a także w języku programowania systemu Coq.

5.2 Formalizacje konstrukcji maszyn abstrakcyjnych dla języków programowania wyższego rzędu

5.2.1 Refocusing w systemie Coq

5.2.1.1 Publikacje

[21] Małgorzata Biernacka, Dariusz Biernacki. Formalizing constructions of abstract machines for functional languages in Coq. *7th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2007)*, Paryż, Francja, 2007.

[181] Filip Sieczkowski, Małgorzata Biernacka, Dariusz Biernacki. Automating derivations of abstract machines from reduction semantics: A generic formalization of refocusing in Coq. *22nd Symposium on Implementation and Application of Functional Languages (IFL 2010)*, LNCS 6647, 72–88, Alphen aan den Rijn, Holandia, 2010.

5.2.1.2 Opis wyników

Technika *refocusingu* została zaproponowana przez Danvy'ego i Nielsena [62] jako metoda mechanicznej derywacji maszyny abstrakcyjnej z (nieefektywnego) ewaluatora implementującego semantykę redukcyjną (semantykę operacyjną małych kroków wykorzystującą jawną reprezentację kontekstów redukcyjnych) i polega na złożeniu pewnych prostych transformacji programów. Oryginalnie refocusing został opracowany dla semantyki redukcyjnej opartej na operacji podstawienia i takiej, w której kontrakcja redeksów jest lokalna, tzn. nie bierze pod uwagę kontekstu redukcyjnego, który jest drugim, oprócz redeksu, wynikiem dekompozycji programu. W późniejszych pracach Biernacka i Danvy rozwinęli metodę refocusingu w dwóch istotnych kierunkach. Po pierwsze, bazując na pojęciu jawnego podstawienia [1] opracowali oni rozszerzenie refocusingu pozwalające na konstrukcję maszyn abstrakcyjnych używających środowiska zamiast zdecydowanie mniej efektywnego podstawienia [27]. Po drugie, uogólniając pojęcie redeksu, umożliwili wyprowadzenie maszyn abstrakcyjnych odpowiadających semantyce wrażliwej na kontekst [28], a więc np. takiej, która definiuje języki z operatorami sterowania czy mutowalnym stanem. Metoda refocusingu stale zyskuje na popularności, a jej aplikacje obejmują między innymi derywacje maszyn abstrakcyjnych dla języka Scheme [29], jak również dla różnych wersji rachunku lambda w wersji call-by-need [60, 89, 9]. Technika ta może służyć nie tylko jako sposób na poprawną derywację nowych maszyn abstrakcyjnych, ale również jako narzędzie umożliwiające weryfikację równoważności różnych specyfikacji semantycznych danego języka programowania, które były zaproponowane niezależnie. Przykładowo, Biernacka i Danvy zaprezentowali semantykę redukcyjną odpowiadającą kilku znanym maszynom abstrakcyjnym, weryfikując ich poprawność przy użyciu refocusingu [28].

Celem naszego projektu, którego wyniki zaprezentowaliśmy w pracy [181] była formalizacja metody refocusingu w asystencie dowodzenia Coq, zapewniająca jej poprawność. W

pracy wprowadzającej podstawową wersję transformacji Danvy i Nielsen zdefiniowali zestaw warunków nałożonych na semantykę redukcyjną, które wystarczają do skonstruowania funkcji ewaluacyjnej maszyny abstrakcyjnej, wraz ze szkicem dowodu poprawności otrzymanej funkcji względem wyjściowej semantyki. Jednakże, ich podejście jest skoncentrowane wyłącznie na ostatecznej definicji wydajnej funkcji ewaluacyjnej, a rozważana przez nich reprezentacja semantyki redukcyjnej nie wydaje się odpowiednia do komputerowej formalizacji. My natomiast prezentujemy refocusing jako ciąg prostych intensjonalnych transformacji relacji ewaluacji wynikającej z semantyki redukcyjnej, dowodząc poprawności każdego z kroków transformacji.

Punktem wyjścia naszej formalizacji jest aksjomatyzacja semantyki redukcyjnej, wystarczająca do automatycznego zastosowania metody refocusingu. Pokazujemy następnie, że każda semantyka redukcyjna spełniająca aksjomaty może być automatycznie przetransformowana do równoważnej jej maszyny abstrakcyjnej. Każdy z pośrednich kroków jest oddzielnie sformalizowany z zapewnieniem jego poprawności. Praca ta stanowi uogólnienie naszych wcześniejszych rezultatów [21]. Oprócz formalizacji podstawowej wersji refocusingu przeprowadziliśmy formalizację obu jej rozszerzeń zaprezentowanych przez Biernacką i Danvy'ego [27, 28].

Całość formalizacji została przeprowadzona w asystencie dowodzenia Coq. Języki, na których została przetestowana to m. in.: język wyrażeń arytmetycznych, rachunek lambda (zarówno czysty jak i rozszerzony o operator sterowania `callcc`) przy gorliwej strategii ewaluacji (otrzymana maszyna abstrakcyjna to maszyna CEK Felleisena [82]), jak również przy leniwej strategii ewaluacji (otrzymana maszyna abstrakcyjna to maszyna Krivine'a [118]), a także Mini-ML. Powyższe instancje ogólnej metody służą ilustracji, jak używać formalizacji, ale również stanowią potwierdzenie właściwego doboru aksjomatów dla semantyki redukcyjnej.

Nasza implementacja w istotny sposób wykorzystuje system modułów Coqa [19], który oparty jest na systemach modułów znanych z języków z rodziny ML. W Coqu jednakże typ modułu (nazywany sygnaturą w języku SML) może zawierać nie tylko deklaracje dotyczące danych, ale również logiczne aksjomaty wyrażające własności wymagane od tych danych. W rezultacie implementacja modułu musi dostarczać dowody, że moduł spełnia wymagane własności. W naszej formalizacji wszystkie własności charakteryzujące semantykę redukcyjną są zebrane w typie modułu. Podobnie, każda z semantyk pośrednich opisujących kolejne kroki refocusingu jest opisana przez taką sygnaturę, a każdy z kroków derywacji jest zdefiniowany jako funktor, tj. transformacja z jednego modułu do drugiego. Formalizacja jest zaprojektowana jako ogólne narzędzie pozwalające użytkownikowi na przetransformowanie jego własnej semantyki redukcyjnej do odpowiadającej jej maszyny abstrakcyjnej, o ile tylko wyspecyfikuje semantykę redukcyjną zgodnie z aksjomatyzacją. Otrzymana maszyna abstrakcyjna jest ekstensjonalnie równoważna początkowej semantyce redukcyjnej.

5.2.2 Formalizacja maszyny abstrakcyjnej STG

5.2.2.1 Publikacje

[155] Maciej Piróg and Dariusz Biernacki. A systematic derivation of the STG machine verified in Coq. *3rd ACM Haskell Symposium (Haskell 2010)*, 25–36, Baltimore, USA, 2010.

5.2.2.2 Opis wyników

Język *Shared Term Graph (STG)* wraz z maszyną abstrakcyjną *Spineless Tagless G-machine (maszyna STG)*, opracowane przez Peyton-Jonesa i Salkilda [152, 151], stanowią trzon kompilatora *Glasgow Haskell Compiler (GHC)* – flagowego kompilatora języka Haskell [97]. STG jest czystym leniwym funkcyjnym językiem wyższego rzędu opartym na znormalizowanym rachunku lambda z konstrukcją wiązania wielu zmiennych, konstruktorami typów algebraicznych i mechanizmem dopasowania wzorca. STG występuje jako język pośredni w kompilatorze GHC i jest kompilowany do kodu, który opisuje wykonanie obliczeń przez maszynę STG. Maszyna STG definiuje semantykę operacyjną, a jednocześnie wyznacza modelową implementację języka STG. Ponieważ maszyna STG zawiera sporo szczegółów implementacyjnych, niespecjalnie nadaje się do wnioskowania o operacyjnych aspektach języka źródłowego. Znacznie bardziej intuicyjnym formalizmem, który abstrahuje od szczegółów implementacyjnych, jest semantyka naturalna, w przypadku języków leniwych pierwotnie zaproponowana przez Launchbury'ego [126], a następnie ulepszona przez Sestofta [178]. Mimo że wyniki Launchbury'ego oraz Sestofta są niezwykle ważne, nie opisują one pełnego języka STG, ale jego uproszczoną wersję. Z kolei, Encina i Peña w swoich pracach prezentują semantykę naturalną języka, który mocno przypomina STG [68, 69, 70], jednakże ich semantyka nie odzwierciedla modelu ewaluacji zdefiniowanego przez maszynę STG w sposobie, w jaki alokowana i aktualizowana jest sterta. Różnica ta znajduje odzwierciedlenie w ich maszynach abstrakcyjnych, które ekstensjonalnie są zgodne z zaproponowaną semantyką naturalną, ale różnią się od oryginalnej maszyny STG.

Żadna z istniejących wcześniej w literaturze semantyk naturalnych nie definiowała pełnego języka STG, a w szczególności jego mechanizmu wiązania wielu zmiennych oraz nietrywialnego protokołu aktualizowania sterty opartego na tzw. flagach aktualizacji. Co więcej, żadna z zaproponowanych semantyk naturalnych ewaluacji leniwej nie jest w pełni zgodna z modelem ewaluacji zdefiniowanym przez oryginalną maszynę STG.

W celu uzupełnienia tej luki, w pracy [155] zaprezentowaliśmy semantykę naturalną języka STG, która stanowi rozszerzenie semantyki Sestofta, a także przeprowadziliśmy systematyczną i mechaniczną derywację odpowiadającej jej maszyny abstrakcyjnej. Metoda derywacji, której użyliśmy, składa się ze standardowych kroków, takich jak wprowadzenie stosu argumentów czy wprowadzenie środowiska, ale kluczową transformacją z semantyki dużych kroków do równoważnej maszyny abstrakcyjnej jest transformacja do stylu kontynuacyjnego [161] połączona z defunkcjonalizacją kontynuacji [161, 61]. Oznacza to, że nasza derywacja w istocie opiera się na funkcyjnej odpowiedniości ewaluatorów wyższego rzędu i maszyn abstrakcyjnych, opracowanej przez Danvy'ego (m. in., przy moim udziale) [4], która już wcześniej okazywała się przydatna w konstruowaniu maszyn abstrakcyjnych dla

leniwego rachunku lambda [5], z tą różnicą, że w pracy [155] przedmiotem transformacji są relacje definiujące semantykę języka, a nie ewaluatory funkcyjne. Ponieważ zastosowana metoda przekształca wyłącznie formę semantyki, pozostawiając zawarty w niej model ewaluacji bez zmian, zaproponowana przez nas semantyka naturalna oraz otrzymana maszyna abstrakcyjna ściśle sobie odpowiadają.

Zgodnie z oczekiwaniami otrzymaną maszyną abstrakcyjną jest maszyna STG. W rezultacie, maszyna STG, chociaż zaprojektowana w celu zapewnienia efektywnej ewaluacji, a co ważniejsze, efektywnej implementacji języka Haskell, może być postrzegana jako naturalny odpowiednik wysokopoziomowej semantyki języka STG, z której została wyprowadzona za pomocą systematycznej i uniwersalnej metody. Przy takim podejściu do konstrukcji maszyny abstrakcyjnej, wszelkie rozszerzenia języka STG mogą zostać wprowadzone na poziomie semantyki naturalnej, a zmodyfikowana maszyna abstrakcyjna powstanie niejako automatycznie przez powtórzenie kroków derywacji.

Cała derywacja zaprezentowana w naszej pracy została sformalizowana w systemie Coq, co stanowi komputerowo zweryfikowany dowód poprawności maszyny STG względem semantyki naturalnej. Wynik ten może być uznany za punkt wyjścia dla znacznie większego projektu, mającego na celu skonstruowanie certyfikowanego kompilatora języka Haskell w Coqu.

5.3 Semantyka formalna języka Ltac (języka taktyk w systemie Coq)

5.3.1 Publikacje

[105] Wojciech Jedynak, Małgorzata Biernacka, Dariusz Biernacki. An operational foundation for the tactic language of Coq. *15th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'13)*, 25–36, Madryt, Hiszpania, 2013.

5.3.2 Opis wyników

Wiele lat pracy nad asystentami wspomagającymi dowodzenie twierdzeń sprawiło, że istniejące narzędzia osiągnęły na tyle wysoki poziom zaawansowania i dojrzałości, że stały się powszechnie używane. Literatura pokazuje, że możliwe jest już formalne przeprowadzenie w asystentach dowodzenia bardzo dużych realistycznych projektów zarówno w dziedzinie informatyki, jak i matematyki. W pierwszej z tych kategorii warto wymienić konstrukcję certyfikowanego kompilatora języka C (projekt CompCert [129]) w Coqu [19], weryfikację jądra systemu operacyjnego (projekt seL4 [116]) w Isabelle [146], czy też formalizację poprawności systemu typów języka Standard ML [127] w Twelfie [153]. Imponującymi projektami z drugiej kategorii są m. in. formalizacja twierdzenia o czterech barwach [91] oraz twierdzenie Feita-Thompsona z teorii grup (ang. Odd Order Theorem) [92], obie przeprowadzone w Coqu.

W pewnych dziedzinach informatyki wykorzystanie asystentów dowodzenia okazuje się szczególnie pomocne: nie tylko wspierają one zapewnienie poprawności, ale też niektóre z nich pozwalają na znaczną automatyzację dowodzenia twierdzeń, co w znacznym stopniu usprawnia proces formalnej weryfikacji i poszukiwanie dowodu wymaganej własności.

Im szersze i bardziej zaawansowane są zastosowania asystentów dowodzenia, tym większa pojawia się potrzeba narzędzi wspierających proces budowania i refaktoryzacji tworzonych z ich pomocą dowodów. Pojawiają się pierwsze prace opisujące trudności i wyzwania związane z bardzo dużymi projektami formalizacyjnymi [44].

Większość z dużych projektów dotyczących formalizacji i weryfikacji w informatyce i matematyce wykorzystuje proceduralne podejście oparte na wykorzystaniu w dowodzeniu twierdzeń tzw. taktyk. W podejściu tym użytkownik konstruuje dowód niejawnie, używając właśnie taktyk; w przypadku, gdy system jawnie buduje term reprezentujący dowód, jest on ukrywany przed użytkownikiem. Definiowane przez użytkownika taktyki złożone (ang. tacticals) znacząco poprawiają automatyzację, utrzymanie i czytelność tworzonych skryptów dowodów. W szczególności, dobrze zaprojektowane ogólne taktyki potrafią znacząco uprościć skrypt i sprawić, by był odporny na zmiany w projekcie.

Coq jest jednym z najbardziej popularnych asystentów dowodzenia wyposażonych w język taktyk Ltac o dużej sile wyrazu. Semantyka języka Ltac została opisana nieformalnie w pracach autora języka [71, 72] oraz w dokumentacji systemu Coq. Taki nieformalny opis jest wystarczający jako wprowadzenie do systemu, ale pozostawia wiele szczegółów niedospecyfikowanych lub omawia je nieprecyzyjnie. Eksperci potrafią bardzo sprytnie wykorzystać niedospecyfikowane fragmenty języka, jednakże pozostali użytkownicy systemu muszą opierać się wyłącznie na intuicji i eksperymentach, by zrozumieć wszystkie mechanizmy języka.

Jedyną wcześniejszą próbą formalnej definicji semantyki języka Ltac jest praca Kirchnera [111], który zaproponował operacyjną semantykę małych kroków języka w formie semantyki redukcyjnej. Jednakże wersja języka, którą rozważał Kirchner jest mocno przestarzała, a w dodatku jego podejście do obsługi wyjątków w Ltacu zawiera znaczące uproszczenia.

W pracy [105] podjęliśmy próbę uzupełnienia braków w formalnym podejściu do języka Ltac poprzez wyczerpujące studium semantyki operacyjnej tego języka. W tym celu zaproponowaliśmy operacyjne podstawy języka Ltac, na które składają się:

- semantyka naturalna, która może odgrywać rolę objaśniającą znaczenie różnych konstrukcji języka, ale również stanowić bazę dla wnioskowania o równoważności taktyk;
- maszyna abstrakcyjna, która stanowi model implementacji języka Ltac;
- semantyka redukcyjna opisująca obliczenia w małych krokach, dobrze nadająca się do śledzenia wykonania taktyk.

Punktem wyjścia naszej pracy jest semantyka naturalna podzbioru języka Ltac, z której pozostałe dwa formaty semantyczne zostały wyprowadzone z zachowaniem semantyki i z użyciem technik funkcyjnej odpowiedniości między ewaluatorem wyższego rzędu a maszyną abstrakcyjną [4] oraz syntaktycznej odpowiedniości między semantyką redukcyjną a maszyną abstrakcyjną (metody refocusingu) [62].

Otrzymane wyniki mogą stanowić podstawę dalszych badań języka Ltac oraz być pomocne w projektowaniu jego wariantów i rozszerzeń. Mogą również służyć jako punkt odniesienia dla użytkowników języka.

6 Literatura

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. A preliminary version was presented at the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL 1990).

[2] Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[3] Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105:159–267, 1993.

[4] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Miller [138], pages 8–19.

[5] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004.

[6] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 340–353. ACM, 2009.

[7] Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2006.

[8] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, 2001.

[9] Zena M. Ariola, Paul Downen, Hugo Herbelin, Keiko Nakata, and Alexis Saurin. Classical call-by-need sequent calculi: The unity of semantic artifacts. In Schrijvers and Thiemann [175], pages 32–46.

[10] Zena M. Ariola, Hugo Herbelin, and Amr Sabry. A proof-theoretic foundation of abortive continuations. *Higher-Order and Symbolic Computation*, 20(4):403–429, 2007.

[11] Zena M. Ariola, Hugo Herbelin, and Amr Sabry. A type-theoretic foundation of delimited continuations. *Higher-Order and Symbolic Computation*, 20(4):403–429, 2007.

[12] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In Zhong Shao, editor, *Proceedings of the 5th Asian Symposium on Programming Languages and Systems (APLAS'07)*, number 4807 in LNCS, pages 239–254, Singapore, December 2007. Springer-Verlag.

[13] Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In Xavier Leroy, editor, *Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages*, pages 64–76, Venice, Italy, January 2004. ACM Press.

[14] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, revised edition, 1984.

[15] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[16] Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.

[17] Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1):25–49, 2006.

[18] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Kahn [106], pages 203–211.

[19] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[20] Małgorzata Biernacka. Formalization of the proof of weak head normalization for System T and its extracted evaluator (an instance of normalization by evaluation), 2007. Available online at http://www.ii.uni.wroc.pl/~mabi/nbe/cbn-system-T-church.

[21] Małgorzata Biernacka and Dariusz Biernacki. Formalizing constructions of abstract machines for functional languages in Coq. In Jürgen Giesl, editor, *Preliminary proceedings of the Seventh International Workshop on Reduction Strategies in Rewriting and Programming (WRS'07)*, Paris, France, June 2007.

[22] Małgorzata Biernacka and Dariusz Biernacki. A context-based approach to proving termination of evaluation. In *Proceedings of the 25th Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXV)*, pages 169–192, Oxford, UK, April 2009. Elsevier.

[23] Małgorzata Biernacka and Dariusz Biernacki. Context-based proofs of termination for typed delimited-control operators. In Francisco J. López-Fraguas, editor, *Proceedings of the 11th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 289–300, Coimbra, Portugal, September 2009. ACM Press.

[24] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005.

[25] Małgorzata Biernacka, Dariusz Biernacki, and Serguei Lenglet. Typing control operators in the CPS hierarchy. In Michael Hanus, editor, *Proceedings of the 13th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'11)*, Odense, Denmark, July 2011. ACM Press.

[26] Małgorzata Biernacka, Dariusz Biernacki, Sergueï Lenglet, and Marek Materzok. Proving termination of evaluation for system F with control operators. In Ugo de'Liguoro and Alexis Saurin, editors, *Proceedings of the 1st Workshop on Control Operators and their Semantics (COS 2013)*, volume 127 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–27, Eindhoven, The Netherlands, June 2013.

[27] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 2006. To appear. Available as the research report BRICS RS-06-3.

[28] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007.

[29] Małgorzata Biernacka and Olivier Danvy. Towards compatible and interderivable semantic specifications for the Scheme programming language, Part II: Reduction semantics and abstract machines. In Jens Palsberg, editor, *Semantics and Algebraic Specification: Essays dedicated to Peter D. Mosses on the occasion of his 60th birthday*, number 5700 in Lecture Notes in Computer Science, pages 186–206. Springer, 2009.

[30] Małgorzata Biernacka, Olivier Danvy, and Kristian Støvring. Program extraction from proofs of weak head normalization. In Escardó et al. [79], pages 169–189.

[31] Dariusz Biernacki. *The Theory and Practice of Programming Languages with Delimited Continuations*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, December 2005.

[32] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hammon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In Krisztian Flautner and John Regehr, editors, *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*, pages 121–130, Tucson, AZ, June 2008. ACM Press.

[33] Dariusz Biernacki, Jean-Louis Colaço, and Marc Pouzet. Clock-directed modular code generation from synchronous block diagrams. In Paul H. J. Kelly and Kevin Hammond, editors, *Proceedings of the 2007 Workshop on Automatic Program Generation for Embedded Systems (APGES 2007)*, pages 12–20, Salzburg, Austria, October 2007.

[34] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, August 2003. Springer-Verlag.

[35] Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. Technical Report BRICS RS-05-16, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark, May 2005. To appear in ACM Transactions on Programming Languages and Systems.

[36] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the dynamic extent of delimited continuations. *Information Processing Letters*, 96(1):7–17, 2005.

[37] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006.

[38] Dariusz Biernacki and Sergueï Lenglet. Applicative bisimulations for delimited-control operators. In Lars Birkedal, editor, *Foundations of Software Science and Computation Structures, 15th International Conference, FOSSACS 2012*, number 7213 in LNCS, pages 119–134, Tallinn, Estonia, March 2012. Springer. An extended version containing appendices with proofs available at http://arxiv.org/abs/1201.0874.

[39] Dariusz Biernacki and Sergueï Lenglet. Normal form bisimulations for delimited-control operators. In Schrijvers and Thiemann [175], pages 47–61. An extended version containing appendices with proofs available at http://arxiv.org/abs/1202.5959.

[40] Dariusz Biernacki and Sergueï Lenglet. Environmental bisimulations for delimited-control operators. In Chung-chieh Shan, editor, *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13)*, volume 8301 of *Lecture Notes in Computer Science*, pages 333–348, Melbourne, VIC, Australia, December 2013. Springer. An extended version containing appendices with proofs available at http://hal.inria.fr/hal-00862189.

[41] Dariusz Biernacki and Sergueï Lenglet. Applicative bisimilarities for call-by-name and call-by-value $\lambda\mu$-calculus. In Bart Jacobs, Alexandra Silva, and Sam Staton, editors, *Proceedings of the 30th Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXX)*, volume 308 of *Electronic Notes in Theoretical Computer Science*, pages 49–64, Ithaca, NY, USA, June 2014. An extended version containing appendices with proofs available at http://hal.inria.fr/hal-00926100.

[42] Dariusz Biernacki and Piotr Polesiuk. Logical relations for coherence of effect subtyping. In Thorsten Altenkirch, editor, *Proceedings of the 13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, volume 38 of *Leibniz International Proceedings in Informatics*, pages 107–122, Warsaw, Poland, July 2015. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik.

[43] Matthias Blume and German Vidal, editors. *Functional and Logic Programming, 10th International Symposium, FLOPS 2010*, number 6009 in Lecture Notes in Computer Science, Sendai, Japan, April 2010. Springer.

[44] Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. Challenges and experiences in managing large-scale proofs. In *Proceedings of the 11th International Conference on Intelligent Computer Mathematics*, pages 32–48, Bremen, Germany, 2012. Springer-Verlag.

[45] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, 1991.

[46] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 178–188. ACM Press, 1987.

[47] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In Wayne Wolf, editor, *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*, pages 173–182. ACM, 2005.

[48] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

[49] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Ravi Sethi, editor, *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977. ACM Press.

[50] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1):55–91, 1992.

[51] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In Wadler [200], pages 233–243.

[52] Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. A new deconstructive logic: Linear logic. *Journal of Symbolic Logic*, 62(3):755–807, 1997.

[53] Vincent Danos and Jean-Louis Krivine. Disjunctive tautologies as synchronisation schemes. In Peter Clote and Helmut Schwichtenberg, editors, *Computer Science Logic, 14th Annual Conference of the EACSL, Proceedings*, volume 1862 of *Lecture Notes in Computer Science*, pages 292–301, Fischbachau, Germany, August 2000. Springer.

[54] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[55] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Thielecke [197], pages 13–23. Invited talk.

[56] Olivier Danvy. Defunctionalized interpreters for programming languages. In Peter Thiemann, editor, *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, Victoria, British Columbia, September 2008. ACM Press. Invited talk.

[57] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.

[58] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [202], pages 151–160.

[59] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[60] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. Defunctionalized interpreters for call-by-need evaluation. In Blume and Vidal [43], pages 240–256.

[61] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press.

[62] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. In Mark van den Brand and Rakesh M. Verma, editors, *Informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001)*, volume 59.4 of *Electronic Notes in Theoretical Computer Science*, Firenze, Italy, September 2001.

[63] Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 224–242, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[64] René David and Walter Py. $\lambda\mu$-calculus and Böhm's theorem. *Journal of Symbolic Logic*, 66(1):407–413, 2001.

[65] Philippe de Groote. A CPS-translation of the $\lambda\mu$-calculus. In Sophie Tison, editor, *19th Colloquium on Trees in Algebra and Programming (CAAP'94)*, number 787 in Lecture Notes in Computer Science, pages 47–58, Edinburgh, Scotland, April 1994. Springer-Verlag.

[66] Philippe de Groote. On the relation between the $\lambda\mu$-calculus and the syntactic theory of sequential control. In Frank Pfenning, editor, *5th International Conference on Logic Programming and Automated Reasoning*, number 822 in LNAI, pages 31–43, Kiev, Ukraine, July 1994. Springer-Verlag.

[67] Philippe de Groote. An environment machine for the lambda-mu-calculus. *Mathematical Structures in Computer Science*, 8:637–669, 1998.

[68] Alberto de la Encina and Ricardo Peña. Proving the correctness of the STG machine. In Ricardo Peña and Thomas Arts, editors, *Implementation of Functional Languages, 14th International Workshop, IFL 2002, Revised Selected Papers*, volume 2670 of *Lecture Notes in Computer Science*, pages 88–104, Madrid, Spain, September 2002. Springer.

[69] Alberto de la Encina and Ricardo Peña. Formally deriving an STG machine. In Miller [138], pages 102–112.

[70] Alberto de la Encina and Ricardo Peña. From natural semantics to C: A formal derivation of two STG machines. *Journal of Functional Programming*, 19(1):47–94, 2009.

[71] David Delahaye. A tactic language for the system Coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 85–95, Reunion Island (France), November 2000. Springer.

[72] David Delahaye. A Proof Dedicated Meta-Language. In Frank Pfenning, editor, *Logical Frameworks and Meta-Languages (LFM)*, volume 70(2) of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 96–109, Copenhagen (Denmark), July 2002. Elsevier.

[73] Paul Downen and Zena M. Ariola. A systematic approach to delimited control with multiple prompts. In Helmut Seidl, editor, *Programming Languages and Systems, 21st European Symposium on Programming, ESOP 2012*, Lecture Notes in Computer Science, pages 234–253, Tallinn, Estonia, March 2012. Springer.

[74] Paul Downen and Zena M. Ariola. Compositional semantics for composable continuations: from abortive to delimited control. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 109–122, Gothenburg, Sweden, September 2014. ACM Press.

[75] Paul Downen and Zena M. Ariola. Delimited control and computational effects. *Journal of Functional Programming*, 24(1):1–55, 2014.

[76] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2:16):1–37, 2011.

[77] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4-5):477–528, 2012.

[78] R. Kent Dybvig, Simon Peyton-Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, 2007.

[79] Martin Escardó, Achim Jung, and Michael Mislove, editors. *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI)*, volume 155 of *ENTCS*, Birmingham, UK, May 2005. Elsevier Science Publishers.

[80] Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988. ACM Press.

[81] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[82] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[83] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[84] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[85] Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994. ACM Press.

[86] Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press.

[87] Michael J. Fischer. Lambda-calculus schemata. In Talcott [194], pages 259–288. Earlier version available in the proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.

[88] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In Norman Ramsey, editor, *Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, pages 165–176, Freiburg, Germany, September 2007. ACM Press.

[89] Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. Lazy evaluation and delimited control. In Benjamin C. Pierce, editor, *Proceedings of the Thirty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 153–164. ACM Press, January 2009.

[90] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[91] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007. Revised and Invited Papers*, volume 5081, pages 333–333. Springer-Verlag, Singapore, December 2007.

[92] Georges Gonthier. Engineering mathematics: the odd order theorem proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–2, Rome, Italy, 2013. ACM.

[93] Timothy G. Griffin. A formulae-as-types notion of control. In Paul Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.

[94] Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. Generating efficient code from data-flow programs. In Jan Małuszyński and Martin Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 207–218, Passau, Germany, August 1991. Springer-Verlag.

[95] Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.

[96] Robert Harper and Mark Lillibridge. Operational interpretations of an extension of F_ω with control operators. *Journal of Functional Programming*, 6(3):393–418, 1996.

[97] Haskell homepage: `http://www.haskell.org`.

[98] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In Guy L. Steele Jr., editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, Texas, August 1984. ACM Press.

[99] Hugo Herbelin. An intuitionistic logic that proves Markov's principle. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010*, pages 50–56, Edinburgh, United Kingdom, July 2010. IEEE Computer Society.

[100] Hugo Herbelin and Silvia Ghilezan. An approach to call-by-name delimited continuations. In Philip Wadler, editor, *Proceedings of the Thirty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 383–394. ACM Press, January 2008.

[101] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[102] Satoshi Ikeda and Koji Nakazawa. Strong normalization proofs by CPS-translations. *Information Processing Letters*, 99(4):163–170, 2006.

[103] Danko Ilik. Delimited control operators prove double-negation shift. *Annals of Pure and Applied Logic*, 163(11):1549–1559, 2012.

[104] Danko Ilik. Proofs in continuation-passing style: normalization of gödel's system T extended with sums and delimited control operators: Distilled tutorial. In Olaf Chitil, Andy King, and Olivier Danvy, editors, *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP'14)*, pages 55–56. ACM Press, 2014.

[105] Wojciech Jedynak, Małgorzata Biernacka, and Dariusz Biernacki. An operational foundation for the tactic language of coq. In Ricardo Peña and Tom Shrijvers, editors, *Proceedings of the 15th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'13)*, pages 25–36, Madrid, Spain, September 2013. ACM Press.

[106] Gilles Kahn, editor. *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[107] Yukiyoshi Kameyama. Axioms for control operators in the CPS hierarchy. *Higher-Order and Symbolic Computation*, 20(4):339–369, 2007.

[108] Yukiyoshi Kameyama and Kenichi Asai. Strong normalization of polymorphic calculus for delimited continuations. In *Proceedings of the Austrian-Japanese Workshop on Symbolic Computation in Software Science (SCSS 2008), RISC-Linz Report Series No. 08-08*, pages 96–108, Hagenberg, Austria, July 2008.

[109] Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In Olin Shivers, editor, *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pages 177–188, Uppsala, Sweden, August 2003. ACM Press.

[110] Yukiyoshi Kameyama and Asami Tanaka. Equational axiomatization of call-by-name delimited control. In Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández, editors, *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'10)*, pages 77–86, Hagenberg, Austria, July 2010. ACM Press.

[111] Florent Kirchner. Coq tacticals and PVS strategies: A small-step semantics. In *Design and Application of Strategies/Tactics in Higher Order Logics*, pages 69–83, 2003.

[112] Oleg Kiselyov. Delimited control in OCaml, abstractly and concretely: System description. In Blume and Vidal [43], pages 304–320.

[113] Oleg Kiselyov and Chung-chieh Shan. Delimited continuations in operating systems. In Boicho Kokinov, Daniel C. Richardson, Thomas R. Roth-Berghofer, and Laure Vieu, editors, *Modeling and Using Context, 6th International and Interdisciplinary Conference, CONTEXT 2007*, number 4635 in Lecture Notes in Artificial Intelligence, pages 291–302, Roskilde, Denmark, August 2007. Springer.

[114] Oleg Kiselyov and Chung-chieh Shan. A substructural type system for delimited continuations. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007*, number 4583 in Lecture Notes in Computer Science, pages 223–239, Paris, France, June 2007. Springer-Verlag.

[115] Oleg Kiselyov and Chung-chieh Shan. Embedded probabilistic programming. In Walid Taha, editor, *Domain-Specific Languages, DSL 2009*, number 5658 in Lecture Notes in Computer Science, pages 360–384, Oxford, UK, July 2009. Springer.

[116] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*, pages 207–220, Big Sky, Montana, USA, October 2009. ACM Press.

[117] Jean-Louis Krivine. Classical logic, storage operators and second-order lambda-calculus. *Annals of Pure and Applied Logic*, 68(1):53–78, 1994.

[118] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[119] Jean-Louis Krivine. Realizability in classical logic. *Interactive models of computation and program behaviour*, 27:197–229, 2009.

[120] Peter J. Landin. A correspondence between Algol 60 and Church's lambda notation. *Communications of the ACM*, 8:89–101 and 158–165, 1965.

[121] Ivan Lanese, Jorge A. Pérez, Davide Sangiorgi, and Alan Schmitt. On the expressiveness and decidability of higher-order process calculi. *Information and Computation*, 209(2):198–226, 2011.

[122] Søren B. Lassen. Bisimulation for pure untyped $\lambda\mu$-caluclus (extended abstract). Unpublished note, January 1999.

[123] Søren B. Lassen. Eager normal form bisimulation. In Panangaden [148], pages 345–354.

[124] Søren B. Lassen. Normal form simulation for McCarthy's amb. In Escardó et al. [79], pages 445–465.

[125] Søren B. Lassen. Head normal form bisimulation for pairs and the $\lambda\mu$-calculus. In Rajeev Alur, editor, *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*, pages 297–306, Seattle, WA, August 2006. IEEE Computer Society Press.

[126] John Launchbury. A natural semantics for lazy evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, January 1993. ACM Press.

[127] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–184, Nice, France, 2007. ACM Press.

[128] Stéphane Lengrand and Alexandre Miquel. Classical fomega, orthogonality and symmetric candidates. *Annals of Pure and Applied Logic*, 153(1-3):3–20, 2008.

[129] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–54, Charleston, SC, USA, January 2006. ACM Press.

[130] Sam Lindley and Ian Stark. Reducibility and tt-lifting for computation types. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005*, volume 3461 of *Lecture Notes in Computer Science*, pages 262–277, Nara, Japan, 2005. Springer-Verlag.

[131] Florence Maraninchi and Yann Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[132] Marek Materzok. Axiomatizing subtyped delimited continuations. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013, CSL 2013*, volume 23 of *LIPIcs*, pages 521–539, Torino, Italy, September 2013. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[133] Marek Materzok. *Control Abstraction for Layered Continuations: Semantics, Types and Implementation*. PhD thesis, University of Wrocław, Wrocław, Poland, 2014.

[134] Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceedings of the 2011 ACM SIGPLAN International Conference on Functional Programming (ICFP'11)*, pages 81–93, Tokyo, Japan, September 2011. ACM Press.

[135] Marek Materzok and Dariusz Biernacki. A dynamic interpretation of the CPS hierarchy. In Ranjit Jhala and Atsushi Igarashi, editors, *Proceedings of the 10th Asian Symposium on Programming Languages and Systems (APLAS'12)*, number 7705 in Lecture Notes in Computer Science, pages 296–311, Kyoto, Japan, December 2012. Springer.

[136] Paul-André Melliès and Jerome Vouillon. Recursive polymorphic types and parametricity in an operational framework. In Panangaden [148], pages 82–91.

[137] Massimo Merro and Corrado Biasi. On the observational theory of the CPS-calculus: (extended abstract). In S. Brookse and M. Mislove, editors, *Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXII)*, volume 158 of *ENTCS*, pages 307–330, Genova, Italy, May 2006. Elsevier Science Publishers.

[138] Dale Miller, editor. *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, Uppsala, Sweden, August 2003. ACM Press.

[139] Robin Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4(1):1–22, 1977.

[140] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[141] James H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, Massachusets Institute of Technology, 1968.

[142] Guillaume Munch-Maccagnoni. Formulae-as-types for an involutive negation. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14*, pages 70:1–70:10, Vienna, Austria, July 2014. ACM Press.

[143] Chetan R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1990.

[144] Chetan R. Murthy. An evaluation semantics for classical proofs. In Kahn [106], pages 96–107.

[145] Chethan R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In Olivier Danvy and Carolyn L. Talcott, editors, *Proceedings of the First ACM SIGPLAN Workshop on Continuations (CW'92)*, Technical report STAN-CS-92-1426, Stanford University, pages 49–72, San Francisco, California, June 1992.

[146] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, 2002.

[147] C.-H. Luke Ong and Charles A. Stewart. A Curry-Howard foundation for functional computation with control. In Neil D. Jones, editor, *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 215–227, Paris, France, January 1997. ACM Press.

[148] Prakash Panangaden, editor. *Proceedings of the Twentieth Annual IEEE Symposium on Logic in Computer Science*, Chicago, IL, June 2005. IEEE Computer Society Press.

[149] Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, number 624 in LNAI, pages 190–201, St. Petersburg, Russia, July 1992. Springer-Verlag.

[150] Michel Parigot. Proofs of strong normalisation for second order classical natural deduction. *Journal of Symbolic Logic*, 62(4):1461–1479, 1997.

[151] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[152] Simon L. Peyton Jones and Jon Salkild. The spineless tagless G-machine. In Joseph E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 184–201, London, England, September 1989. ACM Press.

[153] Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction: Automated Deduction*, pages 202–206. Springer-Verlag, 1999.

[154] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[155] Maciej Piróg and Dariusz Biernacki. A systematic derivation of the STG machine verified in Coq. In *Proceedings of the 3rd ACM Haskell Symposium (Haskell 2010)*, pages 25–36, Baltimore, MD, September 2010. ACM Press.

[156] Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In Andrew Gordon and Andrew Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Publications of the Newton Institute, Cambridge University Press, 1998.

[157] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10(3):321–359, 2000.

[158] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[159] Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In Wadler [200], pages 23–33.

[160] Pascal Raymond. Compilation separee de programmes Lustre. Technical report, Projet SPECTRE, IMAG, July 1988.

[161] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [165].

[162] John C. Reynolds. The coherence of languages with intersection types. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software, International Conference TACS '91*, volume 526 of *Lecture Notes in Computer Science*, pages 675–700, Sendai, Japan, September 1991. Springer.

[163] John C. Reynolds. The discoveries of continuations. In Talcott [194], pages 233–247.

[164] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword [165].

[165] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[166] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In Andrew Tolmach, editor, *Proceedings of the 2009 ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, pages 317–328, Edinburgh, UK, August 2009. ACM Press.

[167] Amr Sabry. Note on axiomatizing the semantics of control operators. Technical Report CIS-TR-96-03, Department of Computer and Information Science, University of Oregon, 1996.

[168] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. In Andre Scedrov, editor, *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92)*, pages 102–109, Santa Cruz, California, June 1992. IEEE Computer Society.

[169] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2012.

[170] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 33(1):1–69, January 2011.

[171] Davide Sangiorgi and Jan Rutten. *Advanced Topics in Bisimulation and Coinduction*. Cambridge Tracts in Theoretical Computer Science (No. 52). Cambridge University Press, 2012.

[172] Davide Sangiorgi and David Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[173] Alexis Saurin. A hierarchy for delimited continuations in call-by-name. In Luke Ong, editor, *Foundations of Software Science and Computation Structures, 13th International Conference, FOSSACS 2010*, number 6014 in Lecture Notes in Computer Science, pages 374–388, Paphos, Cyprus, March 2010. Springer-Verlag.

[174] Scade homepage: http://www.esterel-technologies.com/scade/.

[175] Tom Schrijvers and Peter Thiemann, editors. *Functional and Logic Programming, 13th International Symposium, FLOPS 2012*, number 7294 in LNCS, Kobe, Japan, May 2012. Springer-Verlag.

[176] Helmut Schwichtenberg. Proofs, lambda terms and control operators. In Helmut Schwichtenberg, editor, *Logic of Computation, volume 157 of Series F: Computer and Systems Sciences, Proceedings of the NATO Advanced Study Institute on Logic of Computation, Marktoberdorf, Germany, July 25 - August 6, 1995*, pages 309–347. Springer Verlag, 1997.

[177] Jan Schwinghammer. Coherence of subsumption for monadic types. *Journal of Functional Programming*, 19(2):157–172, 2009.

[178] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[179] Chung-chieh Shan. Delimited continuations in natural language: quantification and polarity sensitivity. In Thielecke [197], pages 55–64.

[180] Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007.

[181] Filip Sieczkowski, Małgorzata Biernacka, and Dariusz Biernacki. Automating derivations of abstract machines from reduction semantics: A generic formalization of refocusing in Coq. In Juriaan Hage and Marco T. Morazán, editors, *Proceedings of the 22nd Symposium on Implementation and Application of Functional Languages (IFL 2010)*, number 6647 in Lecture Notes in Computer Science, pages 72–88, Alphen aan den Rijn, The Netherlands, September 2010. Springer.

[182] Simulink homepage: http://www.mathworks.com/products/simulink.

[183] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In Wand [202], pages 161–175.

[184] Morten H. Sørensen and Paweł Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, 2006.

[185] Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.

[186] Guy L. Steele Jr. and Gerald J. Sussman. Lambda, the ultimate imperative. AI Memo 353, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, March 1976.

[187] Kristian Støvring and Søren B. Lassen. A complete, co-inductive syntactic theory of sequential control and state. In Matthias Felleisen, editor, *Proceedings of the 34th Annual ACM Symposium on Principles of Programming Languages*, pages 161–172, Nice, France, January 2007. ACM Press.

[188] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, 2000. Reprint of the technical monograph PRG-11, Oxford University Computing Laboratory (1974), with a foreword.

[189] Eijiro Sumii. An implementation of transparent migration on standard Scheme. In Matthias Felleisen, editor, *Proceedings of the Workshop on Scheme and Functional Programming*, Technical Report 00-368, Rice University, pages 61–64, Montréal, Canada, September 2000.

[190] Gerald J. Sussman and Guy L. Steele Jr. Scheme: An interpreter for extended lambda calculus. AI Memo 349, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1975. Reprinted in Higher-Order and Symbolic Computation 11(4):405–439, 1998, with a foreword [191].

[191] Gerald J. Sussman and Guy L. Steele Jr. The first report on Scheme revisited. *Higher-Order and Symbolic Computation*, 11(4):399–404, 1998.

[192] Synchrone homepage: `http://www-verimag.imag.fr/Synchrone,30`.

[193] William W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32:198–212, 1967.

[194] Carolyn L. Talcott, editor. *Special issue on continuations (Part I)*, Lisp and Symbolic Computation, Vol. 6, Nos. 3/4, 1993.

[195] Asami Tanaka and Yukiyoshi Kameyama. A call-by-name CPS hierarchy. In Tom Schrijvers and Peter Thiemann, editors, *Functional and Logic Programming - 11th International Symposium, FLOPS 2012*, volume 7294 of *Lecture Notes in Computer Science*, pages 260–274, Kobe, Japan, May 2012. Springer.

[196] Hayo Thielecke. *Categorical Structure of Continuation Passing Style.* PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1997. ECS-LFCS-97-376.

[197] Hayo Thielecke, editor. *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations (CW'04)*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, Venice, Italy, January 2004.

[198] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.

[199] Anne S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics.* Springer-Verlag, 1973.

[200] Philip Wadler, editor. *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, Montréal, Canada, September 2000. ACM Press.

[201] Mitchell Wand. Continuation-based multiprocessing. In Ruth E. Davis and John R. Allen, editors, *Conference Record of the 1980 LISP Conference*, pages 19–28, Stanford, California, August 1980. Reprinted in Higher-Order and Symbolic Computation 12(3):285–299, 1999, with a foreword [203].

[202] Mitchell Wand, editor. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990. ACM Press.

[203] Mitchell Wand. Continuation-based multiprocessing revisited. *Higher-Order and Symbolic Computation*, 12(3):283, 1999.

[204] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

[205] Noam Zeilberger. Polarity and the logic of delimited continuations. In Jean-Pierre Jouannaud, editor, *Proceedings of the 25th IEEE Symposium on Logic in Computer Science (LICS 2010)*, pages 219–227, Edinburgh, UK, July 2010. IEEE Computer Society Press.