Praktycznie we wszystkich dziedzinach współczesnego społeczeństwa – od zarządzania i administracji po biznes, naukę i media społecznościowe – zapotrzebowanie na wydajne obliczenia i przetwarzanie danych rośnie w niespotykanym dotąd tempie. Ten wzrost nie dotyczy jedynie ilości (mocy obliczeniowej, liczby potrzebnych serwerów czy programistów), ale przede wszystkim złożoności zadań stawianych przed systemami komputerowymi. Rodzi to ważne pytanie, jak zarządzać tą złożonością oraz jakie narzędzia i metodologie są możliwe, by wspomóc budowanie i utrzymanie skomplikowanych systemów.

Po stronie oprogramowania kluczowym słowem jest *abstrakcja*, czyli możliwość myślenia w kategoriach wysokopoziomowej logiki systemu, bez potrzeby zaprzątania sobie głowy szczegółami technicznymi niskiego poziomu. Jednym z przykładów, dlaczego narzędzia zapewniające abstrakcję wydają się niezbędne, jest wzrost zrównoleglenia i rozproszenia systemów. Oznacza to, że działają one na wielu niezależnych jednostkach obliczeniowych, na przykład na różnych rdzeniach procesora, jak w przypadku komputerów PC lub smartfonów, lub tysiącach zdalnych serwerów w chmurze obliczeniowej. Jeśli zarządzamy wszystkim „ręcznie", musimy radzić sobie z wieloma problemami niezwiązanymi bezpośrednio z wysokopoziomową logiką systemu, takimi jak organizowanie komunikacji i synchronizacji różnych części systemu, ich konkurowaniem o wspólne zasoby i zakleszczeniami (do których dochodzi, gdy aktywne są dwa obliczenia, każde czekające, aż drugie wykona akcję, co powoduje zacięcie całego systemu).

Narzędziem, które pośredniczy między logiką programu i niskopoziomową implementacją systemu, jest *język programowania*. W ostatnich latach, aby pomóc programistom konstruować i zarządzać skomplikowanymi systemami, główny nurt przemysłowego wytwarzania oprogramowania zwrócił swoją uwagę na *programowanie funkcyjne*, które co prawda nie jest nowym pomysłem (powstało w latach 50. XX wieku wraz z językiem programowania LISP), ale zapewnia wyższy poziom abstrakcja niż, dla przykładu, programowanie obiektowe. Programowanie funkcyjne jest paradygmatem deklaratywnym, co oznacza, że programista definiuje, *co* jest zamierzonym wynikiem programu, a nie *jak* go uzyskać manipulując wewnętrznym stanem maszyny. Zaletą stosowania programowania funkcyjnego na większą skalę jest też to, że jest to jeden z głównych punktów akademickich badań nad teorią języków programowania i semantyką obliczeń, co oznacza, że istnieje obfitość wyników dotyczących zawiłości semantyki, systemów typów, testowania, formalnego rozumowania o programach i tak dalej.

Proponowany projekt badawczy dotyczy dość świeżego pomysłu w dziedzinie programowania funkcyjnego – **efektów algebraicznych**. Ogólnie mówiąc, *efekty* to nazwa dla konstrukcji w językach programowania, które pozwalają programowi robić więcej niż tylko obliczyć wartość na podstawie początkowych argumentów. Przykłady obejmują wykonanie operacji wejścia/wyjścia, zmianę wartości zmiennej w pamięci, komunikację z innym wątkiem czy procesem, zgłoszenie wyjątku. Jedną z zalet programowania funkcyjnego jest zniechęcanie do używania efektów, jeśli nie są one konieczne. Jest to pożądane, ponieważ fragment kodu, który jest *czysty* (czyli nie wykonuje żadnych efektów) zachowuje się w sposób bardzo przewidywalny, może być swobodnie użyty w innym kontekście, jest łatwy do zrozumienia, utrzymania, testowania i formalnego rozumowania. Dlatego jednym z najważniejszych pytań w dziedzinie badań nad językami programowania jest to, czy da się używać efektów, nie tracąc wszystkich tych pożądanych własności. W tej chwili efekty algebraiczne wydają się być najbardziej obiecującym sposobem, by to uzyskać.

Bardziej konkretnym celem tego projektu jest zbadanie dwóch aspektów, które są ważne dla szerszego zastosowania efektów algebraicznych w praktyce programowania i zrozumienie ich matematycznych podstaw. Pierwszym aspektem jest **kompozycja** efektów, czyli to, jak można programować z wieloma różnymi efektami jednocześnie. Wiadomo, że różne efekty mogą ze sobą wchodzić w interakcję na wiele różnych sposobów, a sprawy stają się jeszcze trudniejsze, gdy wykorzystujemy pełną moc abstrakcji – na przykład, gdy nie możemy statycznie powiedzieć, które efekty będą faktycznie używane, gdy wykonamy program.

Kolejnym celem jest zbadanie wymiaru **koindukcyjnego** efektów algebraicznych, czyli zrozumienie programu z efektami nie jako obliczenia produkującego pewną ostateczną wartość z początkowych parametrów, ale jako systemu interaktywnego. Patrząc pod tym kątem, samo zdefiniowanie, co to oznacza, że program jest "poprawny" jest dużo trudniejsze, ponieważ interesuje nas bardziej jego obserwowalne zachowanie, które jest trudniejsze do uchwycenia z formalnego, matematycznego punktu widzenia.

Ogólnym oczekiwanym wynikiem tego projektu jest zbadanie matematycznych podstaw tych dwóch wymiarów efektów algebraicznych. Da to solidną podstawę do budowania lepiej zachowujących się i bardziej ekspresywnych języków programowania. Ponieważ efekty algebraiczne już mają duży wpływ na sposób projektowania i rozumienia języków programowania, rozwiązanie tych dwóch problemów powinno mieć widoczny wpływ na teorię i praktykę programowania.