Om software te ontwikkelen gebruiken programmeurs vaak dynamisch getypeerde programmeertalen, die typetests uitvoeren en rapporteren terwijl het programma draait.Het ontwikkelen en debuggen van programma's in zulke talen is echter niet eenvoudig.Ten eerste, indien het programma een type error rapporteert moet de programmeur de waarde met het verkeerde type identifiëren en uitzoeken waar en hoe deze waarde werd berekend.Deze programmalocatie is niet noodzakelijk in dezelfde functie.De programmeur moet dus een handmatig zoekproces opstarten, wat meestal veel tijd vergt.Een tweede probleem is de tijd nodig om het programma uit te voeren na elke aanpassing, waardoor de programmeur moet wachten.

In deze doctoraatsthesis beschrijven we een programmatransformatie genaamd "blame prediction" voor een kleine programmeertaal die gelijkt op de programmeertaal Scheme.
Het kernidee is om de typetests van ingebouwde functies en -operatoren concreet te maken en deze zo vroeg mogelijk uit te voeren, erop lettend dat de semantiek van het programma niet verandert.
Indien zo'n typetest een fout ontdekt dan spreken we van "blame prediction" voor een primitieve operatie verderop in het programma.
Deze transformatie lost beide problemen op: ten eerste kan de programmeur naar de fout beginnen zoeken vanaf de typetest in plaats van de operatie die gaat falen.
Tegelijkertijd moet het programma minder stappen uitvoeren voor het een fout ontdekt, waardoor de programmeur minder lang moet wachten.

De "blame prediction" transformatie bestaat uit drie onderdelen.
Het eerste onderdeel is gebaseerd op een innovatief typesysteem dat niet alleen het type berekent van elke expressie in een programma, maar ook de typetests die "onderweg" nodig zijn.
De combinatie van types en typetests maakt het mogelijk om typetests te propageren over functiegrenzen heen.
Als we dit typesysteem zonder meer toepassen zorgt de analyse van recursieve functies voor recursieve types, welke eeuwig groeien.
We gebruiken technieken uit de abstracte interpretatie om dit probleem op te lossen.
Een tweede luik van dit onderdeel is een effectanalyse, die conservatief afschat welke variabelen aangepast kunnen worden door een gegeven expressie.
Het einderesultaat van dit onderdeel is een getransformeerd programma waar elke functie-applicatie omringd is door een typetest.

Het tweede onderdeel zal deze typetests zo hoog mogelijk in het programma verplaatsen zonder daarbij de betekenis van het programma aan te tasten.
Concreet wil dit zeggen dat typetests niet mogen verwijzen naar ongebonden variabelen, en dat het programma geen typetests mag uitvoeren voor expressies die anders niet zouden worden uitgevoerd.
De effectanalyse van het eerste onderdeel wordt gebruikt om deze verplaatsing bij te sturen: typetests mogen niet zonder meer over expressies springen indien deze variabelen wijzigen waarop de typetest van toepassing is.

Het derde en laatste onderdeel is een vereenvoudigingsstap welke overbodige typetests weer wegwerkt.
Het resultaat van deze stap — en dus van de blame prediction-transformatie — is een programma dat sterk gelijkt op het invoerprogramma, maar met de toevoeging van speciale expressies die

typetests uitvoeren en "blame predicten" voor gekenmerkte expressies als deze een typeerfout opmerken.

We hebben de blame prediction-transformatie toegepast op twee standaard programmacorpora. Het resultaat hiervan is dat typetests naar boven verplaatst werden in een aantal programma's, zowel statisch (in termen van de programmatekst, wat het eerste probleem oplost) en dynamisch (in termen van de uitvoeringstijd, wat het tweede probleem oplost).
Daarenboven merkten we dat de blame prediction-transformatie een groot aandeel van alle typetests in een dynamisch getypeerd programma kan elimineren. Tenslotte hebben we bewezen dat de blame prediction-transformatie geen extra fouten kan veroorzaken (zij het door ongebonden variabelen te refereren, zij het door extra typetests te introduceren), en dat het het gedrag van het invoerprogramma bewaart.

In the context of software development, programmers often use dynamically typed programming languages, which perform type tests at run-time and directlyreport any type errors. However, developing and debugging programs in such languages is difficult for a number of reasons. First, when a type error is reported, the programmer must identify the wrongly-typed value and trace back through the program to find out how and where it was computed. This place might be in a different part of a program and finding it requires a manual, time-consuming search. Second, there might be a large number of computational steps in between these two program locations, which slows down the edit-run-debug cycle by making the programmer wait.

In this dissertation we propose a program transformation called "blame prediction" for a small core language similar to the programming language Scheme. The driving idea is to make the type tests of built-in functions and operators explicit and perform them as early as possible without changing the program semantics. When such an early type test fails, the program "predicts blame" for primitive operations further down in the program. This solves both problems: first, the programmer can start tracing where and how the value is computed starting from the type test instead of the primitive operation, and second, some computational steps are bypassed, thereby reducing the wait.

The blame prediction transformation consists of three parts. The first part is built around a novel type system which not only infers types from dynamically typed programs, but also records any (run-time) type tests which must be made "along the way" in order for the program to succeed. Such types enable the propagation of type tests beyond function boundaries. Under this type system, recursive functions induce ever-growing recursive types, which we tackle using techniques from abstract interpretation. In addition, our type system makes a conservative estimation of which variables are mutated by expressions. Finally, every function application in the program is wrapped in a type test.

The second part moves these type tests upwards as far as possible, without changing the semantics of the input program. This means making sure no unbound variables are referenced and no type tests are performed which would otherwise not be performed. This part makes use of the mutation

information from the first step: since type tests inspect the type of values inside variables, they may not be moved over expressions which mutate those variables.

The third part is a simplification step which eliminates redundant type tests. The result of this step — and of the blame prediction transformation — is a program similar to the input program, but augmented with special expressions which perform a type test and predict blame for labeled future expressions if they detect a type error.

We have applied the blame prediction transformation to two standard program corpora. We showed that it is capable of moving type tests upwards in a variety of programs, both statically (solving the first problem) and dynamically (solving the second problem). In addition, we found that the blame prediction transformation is able to eliminate a large fraction of the type tests in a dynamically typed program. Finally, we have proved that the blame prediction transformation does not introduce additional errors (either by referencing unbound variables or introducing extra type tests) and that it preserves the behaviour of its input program.