Blame Prediction: Early detection of type errors in dynamically typed programming languages

Dries Harnie

September 2015

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel / fax : +32 2 629 33 44
crazycopy@vub.ac.be
www.crazycopy.be

ISBN : 9789492312051
NUR CODE : 989

Samenvatting

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 functieapplicatie 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.

Abstract

In the context of software development, programmers often use dynamically typed programming languages, which perform type tests at run-time and directly report 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.

Acknowledgements

This work, like many others, was not made in a vacuum. I owe a great deal of gratitude to Wolfgang De Meuter and Christophe Scholliers, without whom this book would be significantly shorter, less developed and utterly incomprehensible. I would like to thank the members of my jury who read this dissertation and challenged me with insightful questions and suggestions: Ann Dooms, Ann Nowé, Coen De Roover, Jacques Noyé, Tom Schrijvers, and Viviane Jonckers.

At home I was always given the chance to put away my work worries for a while, be it by watching TV, gaming, or going on vacation. I could always count on you for taking care of stuff when I could not, and you were there when I needed you.
Bedankt, papa, mama, Lia en Wiebrecht, alsook mijn meter en mijn peter.

At the lab I enjoyed the support and inspiration of my colleagues, specifically those in "SOFT prime": Eline (for being my role model); Joeri, Kevin and Lode (for the Asian dinner dates); Laure (for her animal-friendly view on things); Mattias (for the interesting puzzles); Nathalie (for being a fantastic officemate and enthousiastic supporter); Reinout (for the coffee and the carry); and Simon and Yves (for the chats about games).

Finally, I would like to thank my friends for being aboard the PhD train: both on sunny days and when it passed through dark tunnels. In alphabetical order: Ben, Bregt, Evert, Helder, Jonas, Kevin, Rein, Robbe, Romy and Toon. You each gave me your own brand of diversions, good times, causes for celebrations and causes for worries.
I thank you all.

Tenslotte, als dit de enige pagina is die je leest moet je hier waarschijnlijk ook tussen staan. Bedankt!

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1

INTRODUCTION

1.1 Research Context

Developing software is still predominantly a manual process: the programmer adds code to a program or rewrites existing code in order to implement a feature and then runs the program. If any errors occur, the programmer must diagnose the cause and fix the error. For the programmer, this edit-run-test cycle is repeated until the desired functionality is achieved. This process is the same regardless of which programming language is used. One large category of errors is *type errors*: passing values of the wrong type to primitive functions and built-in operations. When it comes to finding and fixing type errors, the programming languages community is divided between two approaches: statically typed programming languages and dynamically typed programming languages. The chief difference between these two approaches is when type errors are detected and reported.

Under the first approach we find statically typed programming languages such as Java, C#, Haskell, ML, ... [Gosling, 2000; Hejlsberg et al., 2006; Milner, 1997; Peyton Jones, 2003]. These languages statically verify the program before it is allowed to run. If the type system of such a statically typed programming language detects an ill-typed expression, a type error is reported to the programmer and compilation is aborted. Historically, type systems were used to prevent illegal operations, such as indexing an array with a function pointer or adding an integer to a string, but in recent years these type systems have scaled up to verify more complex program properties. [Aldrich et al., 2009; Clarke et al., 1998; Girard, 1995; Honda et al., 1998]

Under the second approach we find dynamically typed languages such as Python, JavaScript, Lua, Scheme, ... [Abelson et al., 1998; Flanagan, 2006; Ierusalimschy et al., 1996; Lutz, 1996]. Programs written in these languages are typically not compiled, but interpreted or at least executed on a virtual machine. Every time an interpreter evaluates a primitive function application such as `x + 3`, it must check the preconditions of the function. If the preconditions are not satisfied, the interpreter reports a type error

to the programmer. The dynamic nature of these languages makes it possible to express a large number of programming concepts. For example, some dynamically typed object-oriented languages allow changing the class of objects at run-time, or even alter the structure of class hierarchies. This flexibility comes at a cost, however: dynamically typed languages are harder to compile to efficient machine code. One viewpoint on dynamically typed languages [Wrigstad, 2009] states that these languages "optimize development time rather than machine time".

The debate between proponents of both approaches has been raging for decades. Proponents of the static typing approach claim that the safety offered by the type system allows for better productivity, while proponents of dynamic typing claim that the increased flexibility of dynamic typing allows for better productivity. For example, Hanenberg et al. [Hanenberg and Stuchlik, 2012; Kleinschmager et al., 2012; Mayer et al., 2012] have performed empirical studies in an attempt to determine which approach — if any — is better. One tentative result from this research is that smaller programs are faster to write in a dynamically typed language, but for larger programs there is no difference in productivity. Another result of this research, however, is that programmers find it easier to maintain programs in a statically typed language. Research into this question will undoubtedly continue.

In recent years, there have been attempts to marry the two approaches in an attempt to get the benefits of both worlds. One example of these hybrid approaches is the *retrofitting* of static typing onto dynamically typed programs [An et al., 2011; Bierman et al., 2014; Guha et al., 2010; Lerner et al., 2013; Tobin-Hochstadt and Felleisen, 2010; Tobin-Hochstadt et al., 2011]. In theory, this allows programmers to write code using the flexibility granted to them by dynamic typing, while still having the safety guarantees offered by static typing. In practice, these approaches often work for a subset of the language and require programmers to avoid some code patterns or rewrite them such that type errors are reported at compile time.

Another hybrid approach is *gradual typing* [Siek and Taha, 2006, 2007; Siek and Vachharajani, 2008]: it allows programmers to provide type annotations for parts of their program, and annotate others with the "unknown" type, written $\star$. Type errors in the statically typed part are reported as normal, whereas the interaction between both parts of the program is mediated by run-time type casts. This overcomes the requirement of making sure the entire program typechecks, while still providing safety guarantees for the statically typed part of the program.

Finally, one approach that goes the other way is *deferred type errors* [Bayne et al., 2011; Peyton Jones et al., 2012]. The authors transform their programs such that type errors are still detected and reported, but the program is allowed to run. The type errors are effectively *deferred* to run-time, where they only abort the program if evaluation reaches the expression(s) involved. In these papers, and especially in [Bayne et al., 2011], the authors report that being able to introduce and test modifications without having to appease the type system is good for both prototyping programs and making invasive API changes.

Apart from the advances on academic front, people in industry have not stopped developing software or inventing new programming languages either. When starting a new project, the programmer needs to choose a language. This depends on many factors, such as the programmer's experience with the language, the existence of libraries and modules in that language, and any constraints imposed by the environment. A survey of the languages used for new projects on the GitHub code collaboration site [Bard, 2014] shows that more than half of all new projects in 2014 used dynamically typed languages.

To summarize, both statically and dynamically typed programming languages have their merits. Both approaches are used, each with a distinct approach to type errors. Both the research and the debate on which is better are still ongoing. In this dissertation, we will focus on debugging type errors in dynamically typed languages.

1.2 Problem Statement: Locating Dynamic Type Errors

In this section we identify and name two problems which occur during development of programs in dynamically typed programming languages. In the course of developing a program in a dynamically typed programming language, the programmer will find (and fix) many type errors. The edit-run-test cycle is then as follows: the programmer edits the program, runs it, and is presented with a type error on a specific line. The first problem the programmer faces is finding which expression caused the type error and why. We call this problem the "*root cause analysis*" problem. Depending on the programming language used the type error can point to the exact expression, or it can be very terse like `Argument "hi" isn't numeric in addition (+) at line 1` or `String can't be coerced into Fixnum in +`.

Once the cause is found, the programmer still needs to figure out where the ill-typed value was computed or passed erroneously throughout the program's abstraction boundaries. There are two main strategies for finding the cause. The first strategy to debug these errors is to insert print statements to inspect intermediate values used in the program. The purpose of these print statements is to identify which values match the programmer's expectations and which do not. In order to find the source of the error, the programmer must — over several iterations — walk upwards in the stack trace to see where program deviates from expectations. This strategy tends to work well if the programmer starts the debugging process with a good idea of where the error can be located. The second strategy is to use a debugger which can stop the program at key points: either at program locations chosen by the programmer or whenever a specific memory location is altered. There the programmer can follow execution until the program state no longer matches expectations. However, the granularity of debugger steps can vary; according to [Boothe, 2000] it is easy to step past the program location that harbors the cause of the error. If that is the case, the programmer must restart the debugging process and again slowly inch towards the new probable

location of the error. Recent developments like omniscient debugging can help the programmer avoid restarting the process, but at a huge cost to performance [Lewis, 2003; Pothier and Tanter, 2011].

Another problem related to finding errors is actually running the program up to the point where it fails. Depending on the program, such a run might take a considerable amount of time. We call this problem the "*long time to crash*" problem. Some of the code that is evaluated will not be relevant to the problem the programmer is trying to fix, for example setting up data structures or pre-processing data. Ideally, program slicing [Silva, 2012] can be used to reduce the program to the essentials, but this requires a program analysis which itself takes time. In general, however, the programmer has no choice but to execute the entire program and wait for it to succeed or fail.

To summarize, we put forth two problems in finding and fixing type errors in dynamically typed programming languages:

Root cause analysis: When a type error is reported, the programmer has to trace backwards through the program in order to find where the epxression that *causes* the type error is defined.

Long time to crash: During the course of debugging this type error, the programmer might modify and run the program several times. The *longer the program takes* before it reports an error, the higher the time cost of debugging.

1.3 Research Goals

In this dissertation we propose to tackle both problems by *performing dynamic type tests earlier*. As we stated in the beginning, in a dynamically typed programming language a primitive operation only performs type tests for its arguments when its result is actually needed. Sometimes these arguments are computed long in advance, so the type of their value can already be tested there. By testing the type earlier, the programmer can start tracing back from the program location of the type test, rather than that of the primitive operation which caused the error message. Depending on how early the type test is performed, the root cause analysis problem is (partially) solved. In addition, the earlier type test can skip over potentially expensive expressions, also alleviating the long time to crash problem. On the other hand, some type tests may be performed even though they are not needed.

We will illustrate this concept using the example in listing 1.1, presented in the programming language Scheme. This code was taken verbatim from the well-known Gabriel benchmarks [Gabriel, 1985], with a single variable reference replaced by another. It is intended to produce a solution to the well-known "n queens" problem, where a program must determine in how many ways one can place n queens on an $n \times n$ chess board without attacking each other. The algorithm works by attempting to place one queen in every column of the board, from left to right.

```
1  (define (nqueens n)
2
3    (define (one-to n)
4      (let loop ((i n) (l '()))
5        (if (= i 0) l (loop (- i 1) (cons i l)))))
6
7    (define (try-it x y z)
8      (if (null? x)
9        (if (null? y)
10         1
11         0)
12       (+ (if (ok? (car x) 1 z)
13            (try-it (append (cdr x) y) '() (cons (car x) z))
14            0)
15          (try-it (cdr x) (cons (car z) y) z))))      ; Error is reported here
16
17   (define (ok? row dist placed)
18     (if (null? placed)
19       #t
20       (and (not (= (car placed) (+ row dist)))
21            (not (= (car placed) (- row dist)))
22            (ok? row (+ dist 1) (cdr placed)))))
23
24   (try-it (one-to n) '() '()))
25
26 (nqueens 8)
```

Listing 1.1: Motivating example: "n queens" solver with an error

```
7  (define (try-it x y z)
8    (if (null? x)
9      (if (null? y)
10       1
11       0)
12     (if (pair? z)
13       (+ (if (ok? (car x) 1 z)
14            (try-it (append (cdr x) y) '() (cons (car x) z))
15            0)
16          (try-it (cdr x) (cons (car z) y) z))
17       (error "z is not a pair" z))))
```

Listing 1.2: Introducing an explicit type test in the motivating example

The three functions in the program work together as follows:

one-to (lines 3–5) Generates a list of numbers 1 through n.

try-it (lines 7–15) Tries to place queens one column at a time. The `z` parameter contains a list of the queens already placed; whenever a new queen is placed it is checked against these queens. Initially, `x` contains a list of all queens that need to be placed and `y` is empty. The `+` expression on lines 12–15 represents two branching paths: the first (lines 12–14) attempts to place the queen at the head of `x` and continues. In the second path (line 15), the queen at the head of `x` is skipped, but remembered in `y`. Once a queen is placed, all queens in `y` become eligible for placement again. After a number of iterations, `x` must become empty: a successful board is recorded if all queens were placed (i. e. `y` is also empty).

ok? (lines 17–22) Determines whether a queen to be placed diagonally attacks any queen already on the board. By construction, horizontal and vertical attacks are already excluded.

Running the program results in a type error pointing to `(car z)` on line 15, with a message reflecting the fact that `car` expects a non-empty list:

```
car: Contract violation
  expected: pair?
  given: '()
```

In terms of the "root cause analysis" problem, the programmer needs to determine whether `z` was supposed to be a non-empty list and, if so, find out why it is empty instead. To make matters worse, this error does not appear instantly: the program must first enumerate *all boards where the first queen has been successfully placed in the first row of the first column*. If the source of the error is not immediately obvious, the programmer will have to insert print statements to record information along the way or slowly step through the program to find the error.

As a start, however, the programmer can decide to perform the type test for `z` as soon as possible. Where should the type test on `z` go, or equivalently: how much earlier can the type test for `(car z)` be performed? Looking at line 15, the call to `car` is an argument to **cons**, the result of which is an argument to `try-it`, which in turn is an argument to `+`. In Scheme, arguments to a function are evaluated before the function is entered, so any type tests on `z` can be performed as early as the invocation of `+` on line 12. If the type test were moved upwards one more level, the program semantics would be different: the first branch of the conditional does not require that `z` is a list. For example, the expression `(try-it '() '() 42)` normally returns `1`, but would suddenly produce errors if the type test were to be moved out of the branch.

Listing 1.2 presents a modified `try-it` function, where the programmer inserted a type test on line 12. If the type test fails, the call to `error` on line 17 stops the program and reports the value of `z`. Running the program again now still results in an error

message, but much earlier. With this reduced run-time, the programmer can focus on the `try-it` function and discover that he made a typo on line 16: `(car z)` should have been `(car x)` all along. With the fix in place, the programmer runs the program again and finds that it produces the desired output. The programmer can then remove the type test and submit the code.

In terms of the "long time to crash" problem, the erroneous program performs 427,601 primitive operations before it reports the error, whereas the program with the type test inserted now only needs 38 primitive operations. To put these numbers in perspective, the corrected program performs 249,739 primitive operations in total. In terms of real-world clock times, the erroneous program runs for 2.30 seconds before it reports a type error, whereas the modified program only runs for 0.51 seconds.

In this thesis, we propose to **automate part of this process by means of a program transformation** called blame prediction. This transformation *decouples* the type tests of primitive operations and *moves* them upwards in the program. We can summarize the essential properties of blame prediction as follows:

Blame prediction may not alter the semantics of a program If the input program terminates successfully, a program which has undergone the blame prediction transformation must do so as well. Vice versa, if the original program raises a type error, the transformed program must also predict blame.

Blame prediction must accept unaltered programs We envision that blame prediction is applied to a program just before it is evaluated. Any program that is accepted by the unmodified interpreter *must also be accepted* by blame prediction.

This stands in stark contrast to many program analyses which only accept subsets of the languages they study and report errors otherwise. Accepting unaltered programs enables programmers to turn blame prediction on or off in their integrated development environment (IDE) without changing any code. This in turn avoids the need for separate, unfamiliar tools, which can have a negative impact on the edit–run–debug cycle.

Blame prediction is a must-fail analysis A program that has undergone blame prediction can stop and *predict blame* at a certain point. This means that *all* paths under that point *must* contain an expression that fails. The failing expressions in these paths are thus "predicted".

This is contrary to conventional type systems, which reject a program if an expression is ill-typed or one path *might* fail. For example, conventional type systems must be conservative and reject the expression

```
(if some-condition 'ok (sqrt "flower"))
```

because both paths have a different type, and the application of `sqrt` expects a number but is given a string. However, if `some-condition` is always true at

runtime this expression will never go wrong as the alternative will never be executed. Therefore, a must-fail analysis can only report an error once it knows which branch will be taken.

The thesis statement of this dissertation is:

> *In a dynamically typed program, the preconditions to primitive operations can be made explicit and tested long before the primitive operation expression is evaluated. When these preconditions do not hold, "blame" will be predicted for an expression, which stops the program. This means that, if the program were to proceed, evaluation of the blamed expression would result in a type error.*

The case we presented in this example is not synthetic, we will present it again in section 6.4 as part of our validation. In addition to raising type errors much earlier in the program's execution, the blame prediction transformation is able to significantly reduce the number of type tests which must be performed. In concordance with the first property of blame prediction, the transformation is allowed to remove type tests which do not provably alter the semantics of the program. For example, type tests on literals or identical type tests on variables can be resolved during processing and elided. In general, this leads to a significant reduction (compared to an un-transformed program) in the number of type tests written in the program text, as well as the number of type tests performed dynamically. For the `nqueens` example above, only 28.57 percent of the type tests in the program text remain, and only 23.83 percent of dynamic type tests are performed. This discrepancy is explained by the presence of recursion: a single type test in the program may be executed multiple times dynamically.

1.4 Contributions

This dissertation makes the following contributions in the fields of static and dynamic program analysis:

An error-tolerant type system for inferring type tests alongside types Our type system infers types for expressions in a dynamically typed language. It is *error-tolerant* because it incorporates potentially wrongly-typed terms in its analysis. For every expression, it records not only the resulting type but also any type tests which must be made during evaluation, which is then used to generate explicit type tests for both primitive operations and user-defined functions. For example, the expression `(+ x 3)` has type $(\mathsf{int} \mathbin{?=} \tau_x) \cdot \mathsf{int}$, which records the type test on `x`. The blame prediction transformation produces a new program regardless of whether τ_x is not equal to int.

A program transformation which performs type test mobility This dissertation describes a program transformation which makes type tests explicit and moves them

upwards in a program without affecting the program semantics with respect to reporting errors. To the best of our knowledge, the blame prediction transformation is the first of its kind to incorporate type test mobility.

Correct blame assignment and additional context for run-time type errors If a blame predicted program performs a type test and detects a type error, it *predicts blame* for the faulty expression. Additionally, values of all variables in the dynamic scope of the type test are reported to aid in debugging.

Proofs of correctness We prove that the blame prediction transformation is correct with respect to preservation of the program semantics. Concretely, this hinges on two equivalences, the latter formed by two entailments:

1. "value preservation": iff a program does not raise type errors, the transformed program produces the same value;
2. "use–blame entailment": if a program raises a type error, the transformed program must predict blame;
3. "blame–use entailment": if a transformed program predicts blame, the original program must raise a type error *or* diverge.

Evaluation and validation of the blame prediction transformation We apply the blame prediction transformation to programs from the well-known Gabriel benchmarks and the Computer Language Benchmark Game (CLBG). We measure how well our implementation of blame prediction can perform type tests earlier, both in terms of the program text and in terms of the program run-time. Additionally, we measure how many type tests the blame prediction transformation is able to eliminate statically, which is again reflected in the number of type tests remaining in the program text and the number of type tests performed at run-time. Finally, we apply a technique from mutation testing [Howden, 1982; Jia and Harman, 2011] in order to generate faulty variants of those in the Gabriel and CLBG sets, and apply the blame prediction transformation to these variants. This illustrates how blame prediction can help programmers find and fix problems in their programs.

1.5 Dissertation Outline

This dissertation is structured as follows:

Chapter 2: Functional Blame Prediction
In this chapter we define syntax and semantics for a minimal functional subset of the programming language Scheme called Scheme$_\beta$. Scheme$_\beta$ is closely related

to the λ-calculus without the Y combinator: it supports function abstractions, function application, variable binding and conditionals.

We also define the *blame prediction transformation* in four stages: The first stage uses a novel type system to identify type tests which are performed while evaluating a given expression in the program. The second stage uses these inferred type tests to make the preconditions of primitive operations explicit. In a third stage, the type tests are moved upwards in the program, with the guarantee that the program semantics are not changed, i. e. type tests always reference bound variables and they are only executed if the code they guard is *also* executed. The fourth and final stage eliminates redundant type tests, with the aim of consolidating type tests as much as possible and only keeping type tests which are critical to the program correctness.

We conclude the chapter by proving that the blame prediction transformation is semantics-preserving: a blame predicted program succeeds if the original succeeds, and it fails if the original program fails.

Chapter 3: Recursion

This chapter first extends the syntax and semantics of Scheme$_\beta$ to support recursive functions by means of an explicit mutable heap. Next, it introduces support for recursion into the core blame prediction transformation. Supporting recursion is non-trivial as the type inference produces "infinite types": types which refer to themselves and induce infinite loops in the rest of the type inference process. Instead, these types are "solved" first to finite types using techniques from abstract interpretation. We show that this process always terminates.

Chapter 4: Mutation

This chapter describes how the blame prediction transformation can be extended to cope with mutable variables. In this chapter we also speculate on supporting other common features of programming languages, such as compound data structures, objects, non-local control flow, and debug prints. Finally, this chapter proves that the extended blame prediction transformation is again semantics-preserving.

Chapter 5: Implementation

This chapter describes the architecture of our prototype implementation in broad strokes. Both the chapter and the implementation are patterned according to the stages given in chapter 2. Finally, we describe two bottlenecks in the implementation which were discovered while performing benchmarks and how we tackled them. The first bottleneck is related to a generic tree traversal library we used. The second bottleneck is caused by a naïve implementation of simplification which accidentally destroyed value sharing and rewrote types needlessly.

Chapter 6: Evaluation & Validation

The evaluation chapter applies the blame prediction transformation to programs from two well-known benchmarking sets. The aim of this experiment is to evaluate blame prediction along two criteria: first, how well blame prediction can move type tests upwards, and second, how many redundant type tests it can eliminate. These criteria are applied both to the program text and its run-time behavior. Finally, we apply the blame prediction transformation to mutated variants of the benchmarking programs to gauge how well it solves the "root cause analysis" and "long time to crash" problems.

Chapter 7: Related Work

This chapter situates the work presented in this dissertation in the general research context. We divide the related work into three approaches, which correspond to the time(s) when errors are reported to the programmer. We conclude by remarking that blame prediction occupies a niche in-between static analysis (error reports before the program is run) and dynamic analysis (error reports while the program runs).

Chapter 8: Conclusion and Future Work

The final chapter of this dissertation revisits the thesis and the contributions made. We highlight some key areas where blame prediction can be improved and speculate on other applications of blame prediction.

1.6 Supporting Publications and Technical Contributions

A number of publications, exploratory activities, and technical contributions directly support this dissertation. This section discusses them briefly to highlight their relevance to this work.

Publication

Dries Harnie, Christophe Scholliers, and Wolfgang De Meuter. Blame Prediction. *TFP '13: Trends in Functional Programming*, 8322:91–106, 2013

This publication served to introduce the concept of blame prediction for a small functional programming language. This paper already described much of the blame prediction transformation in its current form, as in chapter 2. It was presented at the "Trends in Functional Programming" symposium with the intent of gathering feedback and related work.

Implementation

Dries Harnie. Prototype Implementation of Blame Prediction. Technical report, September 2015

As a proof of concept, we have created a prototype implementation of the blame prediction transformation, which can be found at `https://github.com/Botje/crystal`. The result of the transformation is a program augmented with `check` expressions. This program can be executed in any R5RS-compatible interpreter with the addition of a macro definition for `check`. We will describe the implementation in more detail in chapter 5. For experimentation purposes, an on-line tool is available at `http://bit.ly/blame-sandbox`.

1.7 Other Publications

In this section we present other publications unrelated to this dissertation. The aim is to document our academic track record outside of blame prediction.

Much of our work was in the context of the AmbientTalk/2 project, a so-called "ambient-oriented programming language" [Mostinckx et al., 2005]. This programming language defines a programming model where connections between devices are established on ad-hoc discovery. These connections are very volatile as users can move about and establish new connections or break existing ones. Communication over such connections is thus done over so-called "far references", which uniquely identify an object hosted on a (mobile) device. In order to make applications be able to communicate in this hostile networking environment, all communication over far references must be done asynchronously. If two-way communication is needed, programmers can use the future pattern to (again asynchronously) capture a reply. Finally, far references can be created either through service discovery or by passing a far reference as part of a message.

As the first part of our participation, we defined "ambient contracts" which allow programmers to declaratively define groups of services and automatically assemble them once all parties have been discovered. The contract remains established as long as all services remain connected. As long as this is the case, the programmer can deploy aspects on the nodes of the service which implement a cooperation (e. g. playing a tune on the sound system when a phone rings). The programmer can additionally install event handlers to react on the dissolution of the contract (when a service disconnects) and its re-establishment. All this taken together enables programmers to deal with groups of services much more easily than dealing with the services separately.

Publication

1. Dries Harnie, Christophe Scholliers, and Wolfgang De Meuter. Ambient Contracts. *Electronic Communications of the EASST*, 28(0), March 2010

2. Christophe Scholliers, Dries Harnie, Éric Tanter, Wolfgang De Meuter, and Theo D'Hondt. Ambient contracts: verifying and enforcing ambient object compositions à la carte. *Personal and Ubiquitous Computing*, 15(4), April 2011

Also in the context of the AmbientTalk/2 language, we developed a new kind of far reference which is aware of the actual network technology used to transport data. This allows applications to, for example, conserve bandwidth when on a mobile data plan or provide extra functionality when other nodes in the network are reachable via BlueTooth. Network-aware references (NARs) maintain bundles of references to the same device over multiple network technologies. The programmer can then attach policies to messages to indicate which technology to use, as well as register listeners to react on the (dis)connection of certain technologies.

Publication

3. Kevin Pinte, Dries Harnie, and Theo D'Hondt. Enabling Cross-Technology Mobile Applications with Network-Aware References. In *COORDINATION '11: Proceedings of the 13th International Conference on Coordination Models and Languages*, pages 142–156. Springer Berlin Heidelberg, 2011a

4. Kevin Pinte, Dries Harnie, Elisa Gonzalez Boix, and Wolfgang De Meuter. Network-aware references for pervasive social applications. *PerCol '11: Proceedings of the 2011 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 537–542, 2011b

Finally, we participated in formalizing the semantics of the AmbientTalk/2 language, particularly with regards to the interaction between futures and asynchronous message sending. This work was inspired by the semantics defined for a similar language JCoBox [Schäfer and Poetzsch-Heffter, 2010], which also supports futures.

Publication

5. Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures ()*, 40(3-4):112–136, October 2014

Next, we developed a hardware and software platform for deploying "city applications" on public transportation vehicles as part of an Innoviris project called "Urban area data structures for city applications in mobile nomadic networks". The aim of this project was to come up with a seamless networking infrastructure which enabled developers to create city applications which were accessible in the city of Brussels, regardless of whether the user actually has a mobile data plan on their phone. The key component of this platform was a so-called "urban-area tuple space", using techniques from delayed networking to propagate information across devices, be they users' phones, public transportation vehicles, or stationary network nodes.

Publication

6. Dries Harnie, Elisa Gonzalez Boix, Theo D'Hondt, and Wolfgang De Meuter. Programming urban-area applications. In *SAC '12: Proceedings of the 27th ACM Symposium on Applied Computing*, pages 1516–1521, 2012

7. Dries Harnie, Elisa Gonzalez Boix, Theo D'Hondt, and Wolfgang De Meuter. Programming Urban-Area Applications by Exploiting Public Transportation. *Transactions on Autonomous and Adaptive Systems (TAAS)*, 9(2), July 2014

Finally, we collaborated with the ExaScience Life Lab at Imec to produce a fault-tolerant, distributed coordination program for target prediction. Briefly summarized, (protein) target prediction is the first step towards drug discovery, which aims to identify candidate molecules which have an effect on the protein targets which are responsible for certain diseases. This is implemented by an expert system which consults a number of individual predictor programs, written in C++. We developed a coordination program using the Spark framework [Zaharia et al., 2012] which automatically runs these predictors on a cluster and aggregates the result. This coordination program allows for better checkpointing and monitoring of the target prediction process. In addition, it becomes easier to run multiple predictions in parallel and use the cluster optimally.

Publication

8. Dries Harnie, Alexander E Vapirev, Jörg Kurt Wegner, Andrey Gedich, Marvin Steijaert, Roel Wuyts, and Wolfgang De Meuter. Scaling Machine Learning for Target Prediction in Drug Discovery using Apache Spark. In *CCGRID Life '15: Proceedings of the 2015 Workshop on Clusters, Clouds and Grids for Life Sciences*, 2015

2

FUNCTIONAL BLAME PREDICTION

In the introductory chapter we gave a high level overview of the goal of this dissertation, namely making type tests in dynamically typed programs explicit and moving them upwards in the program. In this chapter we present the *blame prediction transformation*. The blame prediction transformation accepts programs written in the functional core of a Scheme-like language called Scheme$_\beta$, and introduces explicit type tests by way of check expressions.

In this chapter, we first define the syntax and semantics of Scheme$_\beta$ (section 2.1). Next, we define the blame prediction transformation in four stages (sections 2.2 to 2.5), which are explained below. Finally, section 2.6 and section 2.7 prove the correctness of the blame prediction transformation with regards to not introducing additional errors and preserving the program semantics.

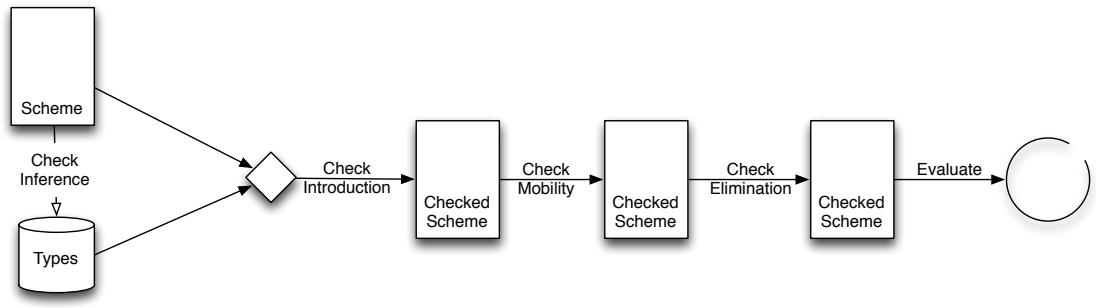

Figure 2.1: Overview of the blame prediction transformation

The introduction of check expressions in the blame prediction transformation happens in four stages. These stages are as follows, as illustrated in figure 2.1:

1. First, a type system infers the type tests made by the primitive operations in the program. These result in *conditional types* for various parts of the program.

2. Conditional types are then used as a guide to insert explicit check expressions into the program directly around function application expressions.
3. The explicit check expressions are moved upwards in the program, without changing its semantics.
4. Finally, redundant check expressions are eliminated and the remaining check expressions are simplified.

2.1 Language Description

Before explaining the blame prediction transformation in detail, we describe the language Scheme$_\beta$, a functional subset of Scheme [Abelson et al., 1998] extended with check expressions.

Syntax Figure 2.2 presents the syntax of Scheme$_\beta$. An expression is either:

- a simple expression (a variable reference, a constant or a lambda expression). Constants are either booleans, numbers, or primitive operations;
- a function application, where operator and operands are all simple expressions;
- a conditional expression which chooses between two expressions depending on the result of the condition, also a simple expression;
- a let expression;
- a check expression.

We require that the input program to the blame prediction transformation is well-formed, i. e. a *closed* expression which does not contain free variables apart from primitive operations. Additionally, every expression in the program must be uniquely associated with a source location (or *label*) ℓ. Two functions exist that map labels to expressions and vice versa: Label returns the label for a given expression, and Expr returns the expression pointed at by a label.

check expressions serve to explicitly verify preconditions of primitive operations. They evaluate their body e only if the precondition p holds. Preconditions, noted p, are trees of conjunctions and disjunctions, with either type tests or no-ops (#t) at the leaves. These type tests verify whether the given variable or constant, denoted $\times/c$, is of the specified type. If the precondition p does not hold, a *blame prediction error* is raised, which predicts blame for the set of labels L, which correspond to expressions in the program. The blame prediction transformation will introduce check expressions into the program, the programmer does not have to write them.

The distinction between ordinary expressions e and simple expression s guarantees that any Scheme$_\beta$ expression is automatically in Administrative Normal Form

$\mathsf{e} \in \text{Exp}$	$::=$	s	Simple expressions
	$\mid$	$(\mathsf{s}\ \mathsf{s}_1 \ldots \mathsf{s}_n)$	Application
	$\mid$	$(\mathsf{if}\ \mathsf{s}\ \mathsf{e}\ \mathsf{e})$	Conditional
	$\mid$	$(\mathsf{let}\ ([\mathsf{x}\ \mathsf{e}])\ \mathsf{e})$	Let
	$\mid$	$(\mathsf{check}\ \mathsf{p}\ \mathsf{e})$	Check
$\mathsf{s} \in \text{Simp}$	$::=$	x	Variables
	$\mid$	c	Constants and literals
	$\mid$	$(\mathsf{lambda}\ (\mathsf{x}_1 \ldots \mathsf{x}_n)\ \mathsf{e})$	Lambda expressions
$\mathsf{c} \in \text{Const}$	$::=$	$\#\mathsf{f} \mid \#\mathsf{t}$	Boolean constants
	$\mid$	n	Integer literals
	$\mid$	"..."	String literals
	$\mid$	o	Primitive operations
$\mathsf{o} \in \text{PrimOp}$	$::=$	$+ \mid - \mid * \mid \ldots$	
$\mathsf{p} \in \text{Pred}$	$::=$	$(\tau? \times/\mathsf{c})^{\mathsf{L}}$	Simple type test
	$\mid$	$\mathsf{p} \wedge \mathsf{p}$	Conjunction
	$\mid$	$\mathsf{p} \vee \mathsf{p}$	Disjunction
	$\mid$	$\#\mathsf{t}$	No-op
$\times/\mathsf{c}$	$::=$	$\mathsf{x} \mid \mathsf{c}$	
$\ell \in \mathsf{L}$	$\subset$	Lab	
Label	$:$	$\text{Exp} \mapsto \text{Lab}$	Labels of expressions
Expr	$:$	$\text{Lab} \mapsto \text{Exp}$	Expressions at labels

Figure 2.2: Syntax of Scheme$_\beta$

(ANF), presented in [Sabry and Felleisen, 1993]. This formulation makes the order of evaluation explicit, which is commonly done in program transformations. The specification in terms of ANF does not diminish the generality of our approach, as Sabry and Felleisen [1993] demonstrates a general transformation of arbitrary expressions to ANF.

Semantics The semantics of Scheme$_\beta$ are presented in figure 2.3, using the notation in Felleisen and Hieb [1992]. Evaluation of a well-formed program proceeds by repeatedly reducing an applicable expression e inside an evaluation context E, written $\mathrm{E}\langle \mathsf{e} \rangle$. This evaluation context is either the whole program, or the expression part of a let expression. Because there is exactly one evaluation context that can be reduced further at all times during the evaluation process, evaluation is deterministic. When no evaluation context applies, the remaining expression must be either a run-time value (such as #f, 98, "hello" or a function), or an error. We write $P \rightsquigarrow \mathsf{v}$ for the reduction of program P to a value v and $P \rightsquigarrow \text{err-}\omega$ in case of an error.

$$
\begin{array}{rll}
\mathsf{v} & ::= \#\mathsf{f} \mid \#\mathsf{t} \mid \mathsf{n} \mid \lambda \mathsf{x}_1 \ldots \mathsf{x}_n.\mathsf{e} & \text{Run-time values} \\
\mathsf{E} & ::= \Box \mid (\mathsf{let}\ ([\mathsf{x}\ \mathsf{E}])\ \mathsf{e}) & \text{Evaluation contexts}
\end{array}
$$

$$
\begin{array}{rlll}
\text{(E-If-False)} & \mathsf{E}\langle(\mathsf{if}\ \#\mathsf{f}\ \mathsf{e}_1\ \mathsf{e}_2)\rangle & \to \mathsf{E}\langle\mathsf{e}_2\rangle & \\
\text{(E-If-True)} & \mathsf{E}\langle(\mathsf{if}\ \mathsf{v}\ \mathsf{e}_1\ \mathsf{e}_2)\rangle & \to \mathsf{E}\langle\mathsf{e}_1\rangle & \text{if } \mathsf{v} \neq \#\mathsf{f} \\
\text{(E-Let)} & \mathsf{E}\langle(\mathsf{let}\ ([\mathsf{x}\ \mathsf{v}])\ \mathsf{e})\rangle & \to \mathsf{E}\langle\mathsf{e}[\mathsf{v}/\mathsf{x}]\rangle & \\
\text{(E-Lambda)} & \mathsf{E}\langle(\mathsf{lambda}\ (\mathsf{x}_1 \ldots \mathsf{x}_n)\ \mathsf{e})\rangle & \to \mathsf{E}\langle\lambda \mathsf{x}_1 \ldots \mathsf{x}_n.\mathsf{e}\rangle & \\
\text{(E-Apply)} & \mathsf{E}\langle(\mathsf{v}_f\ \mathsf{v}_1 \ldots \mathsf{v}_n)\rangle & \to \mathsf{E}\langle\delta(\mathsf{v}_f, \mathsf{v}_1, \ldots, \mathsf{v}_n)\rangle & \\
\text{(E-Error)} & \mathsf{E}\langle\mathsf{err}\text{-}\omega\rangle & \to \mathsf{err}\text{-}\omega &
\end{array}
$$

$$
\begin{array}{rlll}
\text{(E-Check-Pass)} & \mathsf{E}\langle(\mathsf{check}\ \mathsf{p}\ \mathsf{e})\rangle & \to \mathsf{E}\langle\mathsf{e}\rangle & \text{if p holds} \\
\text{(E-Check-Fail)} & \mathsf{E}\langle(\mathsf{check}\ \mathsf{p}\ \mathsf{e})\rangle & \to \mathsf{err}\text{-}\mathsf{blame}(\mathsf{p}) & \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcll}
\delta(\mathsf{o}, \mathsf{v}_1, \mathsf{v}_2) & = & \mathsf{err}\text{-}\mathsf{not}\text{-}\mathsf{int}(\mathsf{v}_i) & \text{if } \exists i : \neg\mathsf{int?}\ \mathsf{v}_i \\
\delta(\mathsf{o}, \mathsf{v}_1, \mathsf{v}_2) & = & \mathsf{o}(\mathsf{v}_1, \mathsf{v}_2) & \text{otherwise} \\
\delta(\mathsf{o}, \mathsf{v}_1, \ldots, \mathsf{v}_m) & = & \mathsf{err}\text{-}\mathsf{args}\text{-}\lambda(\mathsf{o}) & \\
\delta(\lambda \mathsf{x}_1 \ldots \mathsf{x}_m.\mathsf{e}, \mathsf{v}_1, \ldots, \mathsf{v}_m) & = & \mathsf{e}[\mathsf{v}_1 \ldots \mathsf{v}_m / \mathsf{x}_1 \ldots \mathsf{x}_m] & \\
\delta(\lambda \mathsf{x}_1 \ldots \mathsf{x}_m.\mathsf{e}, \mathsf{v}_1, \ldots, \mathsf{v}_n) & = & \mathsf{err}\text{-}\mathsf{args}\text{-}\lambda(\lambda \mathsf{x}_1 \ldots \mathsf{x}_m.\mathsf{e}) & \text{if } m \neq n \\
\delta(\mathsf{v}, \ldots) & = & \mathsf{err}\text{-}\mathsf{not}\text{-}\lambda(\mathsf{v}) & \text{if } \neg\mathsf{function?}(\mathsf{v})
\end{array}
$$

Figure 2.3: Run-time syntax and evaluation rules of Scheme$_\beta$

Evaluation of the various language constructs themselves is defined as usual. In a let expression, references to the bound variable in the body are replaced by the computed value. This is enforced by the separate evaluation contexts for let: $(\mathsf{let}\ ([\mathsf{x}\ \mathsf{e}_\mathsf{x}])\ \mathsf{e})$ cannot be reduced before e_x has been reduced to a value.

Evaluation of function applications (both primitive and user-defined) is handled by the δ operator. For simplicity, we assume all primitive operators o receive two integer-typed values and return an integer, but other operations can be added similarly. There are three possibilities:

1. The value in function position is a primitive operator such as o: the primitive operation is applied to the operands. If a primitive operation is called with a different amount of parameters, it raises an $\mathsf{err}\text{-}\mathsf{args}\text{-}\lambda(\mathsf{o})$. If one of the parameters is not a number, an $\mathsf{err}\text{-}\mathsf{not}\text{-}\mathsf{int}(\mathsf{v}_i)$ error is raised.

2. The function is user-defined: evaluation continues with a copy of the body, where every formal parameter $\mathsf{x}_1 \ldots \mathsf{x}_m$ is replaced by its corresponding value. If the number of actual parameters does not match the number of formal parameters, an $\mathsf{err}\text{-}\mathsf{args}\text{-}\lambda(\lambda \mathsf{x}_1 \ldots \mathsf{x}_m.\mathsf{e})$ error is raised.

3. Finally, if the value in function position is not a function, an $\mathsf{err}\text{-}\mathsf{not}\text{-}\lambda(\mathsf{v})$ error is raised.

In the next sections we describe the blame prediction transformation, which aims to insert check expressions into the program and move them upwards as high as possible.

2.2 Check Inference

The first stage of the blame prediction transformation determines types for all the expressions throughout the program. Unlike regular type systems, our type inference also encodes the various type tests made along the path that yields the final type. There are three reasons for this choice: first, we can still ascribe types to expressions which produce type errors at run-time. The second reason is that type tests can be resolved statically and will not result in run-time type tests. Finally, type tests in a function body can become part of its type, meaning that type tests become accessible outside of the function.

To give an example, the expression `(+ 7 5)` is assigned a *conditional type*

$$(\mathsf{int} \mathbin{?=} \tau_7) \cdot (\mathsf{int} \mathbin{?=} \tau_5) \cdot \mathsf{int}$$

Read as: "if τ_7 (the type of 7) is equal to type int and τ_5 (the type of 5) is equal to type int, this expression has type int". τ_7 and τ_5 are the types of the literals 7 and 5, i. e. int. Therefore, the type of `(+ 7 5)` can be simplified to int.

Recall that we stated in the introductory chapter that check inference **must accept any program**, even those with type errors. For example, most type systems would reject the expression `(+ 3 "hello")` as wrongly typed. In our system, this expression does not result in a failed compilation but gives rise to the type

$$(\mathsf{int} \mathbin{?=} \mathsf{string}) \cdot \mathsf{int}$$

A later part of the blame prediction transformation will introduce a $(\mathsf{check}\ (\mathsf{int?}\ \text{"hello"})\ \ldots)$ expression that always fails, assigning blame to the expression `(+ 3 "hello")`. If this expression never happens to be reached — for example because it resides in a lambda that is never called or a never-reached branch of an if expression — the program executes normally.

Another important property of check inference is that it must also make sure to **preserve** all type tests made "en route" to reach a value. For example, the snippet in listing 2.1 performs a `sqrt` operation on line 1 and binds the result to a variable `y`, which is not used in the body.

```
1 (let ([y (sqrt x)])
2   z)
```

Listing 2.1: Example of why preserving all types in a let is necessary

In a traditional type system, the type of the expression `(sqrt x)` is bound to the variable `y` during type checking of the body. If the body does not use the variable, as is the case here, the type is discarded. Thus, the type tests made by the expression `(sqrt x)`

$$
\begin{array}{rcll}
\tau & ::= & \star & \text{The “any” type} \\
 & | & \gamma & \text{ground types} \\
 & | & \tau \vee \tau & \text{union types} \\
 & | & (\tau_t \mathbin{?=} \tau)^{\ell}_{\ell} \cdot \tau & \text{conditional type} \\
 & | & \Pi_{(\alpha_1 \ldots \alpha_n)}.\tau & \text{type function} \\
 & | & \alpha & \text{type variable} \\
 & | & (\alpha\ \tau_1 \ldots \tau_n) & \text{type function application} \\
 & & & \\
\gamma & ::= & \mathsf{int} \mid \mathsf{string} \mid \mathsf{boolean} & \\
\tau_t & ::= & \gamma \mid \mathsf{function\text{-}arity\text{-}0} \mid \mathsf{function\text{-}arity\text{-}1} \mid \ldots & \\
\alpha, \beta, \ldots & \in & \text{TVar} &
\end{array}
$$

Figure 2.4: Overview of types

would not be included in the final type, making the final type of the expression τ_z, the type of the variable z. However, our type system preserves the type tests made by `(sqrt x)`, so this expression has type $(\mathsf{number} \mathbin{?=} \tau_x) \cdot \tau_z$. In our type system we use a technique we call “chaining”, which preserves type tests made in let-expressions by prepending any conditional types in the type of the expression to the type of the body.

Types The types that are employed in our type system are displayed in figure 2.4. The “any” type $\star$ is used for expressions of which no type information is known, they can contain any value at run-time. Next are ground types γ, which are either int[1], string or boolean. In our system, the two branches of an if expression can have different types. This is supported by combining them in a *union type* [Barbanera and Dezani-Ciancaglini, 1995; Pierce, 1991]. For example, the type of `(if some-test 3 "hi")` is $\mathsf{int} \vee \mathsf{string}$.

As stated above, recording type tests is done using *conditional types*: they consist of the type test (the $(\tau_t \mathbin{?=} \tau)$ part), and the type of the computation that is performed if the test succeeds. τ_t must be a concrete type which matches the given type τ, or one of $\mathsf{function\text{-}arity\text{-}0}, \mathsf{function\text{-}arity\text{-}1}, \ldots$ which match functions with zero, one, ... arguments. Each conditional type has two labels attached to it: a *blame label* (top) and a *cause label* (bottom). These labels refer to the source positions of the function application and the expression being tested, respectively. They are used to correctly assign blame in the last stage of the transformation. For example, in the case of `(+ 3 "hello")`, the blame label points to the application of +, and the cause label points to `"hello"`.

Finally, type functions — either from user-defined functions or wrapped primitive operators — are templates where type variables serve as placeholders for the types of their arguments and the labels of the expressions that give rise to them. Applying a

[1] R5RS Scheme actually only has a number type, but in this dissertation we will stay with int. We will discuss how blame prediction can support other kinds of numbers in section 2.8.2.

type function is done by substituting the actual types for the type variables, and updating the cause labels for any type tests involving these type variables. By performing type function application in this manner, any conditional types inside the function are made visible to its callers, where they are applied to the actual argument types.

```
(let ([f (lambda (x) (+ 3 x))])
  (f 5))
```

Listing 2.2: Example of type functions and type function applications

For example, in listing 2.2 the function `f` has type function $\Pi_{(\alpha)}.(\mathsf{int} \mathbin{?=} \alpha) \cdot \mathsf{int}$. The constant `5` has type int, so to calculate the type of `(f 5)` we need to substitute int for α. The result of this substitution is $(\mathsf{int} \mathbin{?=} \mathsf{int}) \cdot \mathsf{int}$, which can be simplified to just int.

This mechanism also supports higher-order functions, where function arguments are themselves invoked as functions. An example of such a higher-order function is

```
(lambda (f) (f 3 "hi"))
```

which applies its argument `f` to two arguments. Since nothing is known about this argument until it is supplied, `f` is bound to a type variable α_f. The body applies `f` to two arguments, therefore α_f must be a type function of two arguments, which results in the following final type:

$$\Pi_{(\alpha_f)}.(\mathsf{function\text{-}arity\text{-}2} \mathbin{?=} \alpha_f) \cdot (\alpha_f\ \mathsf{int}\ \mathsf{string})$$

where the conditional type tests whether α_f is a function of two arguments. As we stated above, applying this function will substitute an actual type for α_f. For example, passing in the function `(lambda (a b) b)` with type $\Pi_{(\delta,\epsilon)}.\epsilon$ yields the following type:

$$\begin{aligned}
&(\mathsf{function\text{-}arity\text{-}2} \mathbin{?=} \Pi_{(\delta,\epsilon)}.[\epsilon]) \cdot (\Pi_{(\delta,\epsilon)}.[\epsilon]\ \mathsf{int}\ \mathsf{string}) \\
&= (\Pi_{(\delta,\epsilon)}.[\epsilon]\ \mathsf{int}\ \mathsf{string}) \\
&= \mathsf{string}
\end{aligned}$$

Finally, we must remark that the reduction rules for applied type functions are *non-standard*. They are shown in figure 2.5. First, if the argument to a type function is a union type, the type function application is duplicated for each of the branches in the union. Next, if an argument is a conditional type, the conditional part is moved outside of the type function application. Finally, passing type variables, type functions and ground types as arguments to a function causes them to be substituted in the body of the type function and the arity of the type function to be decreased. The one exception in these reduction rules are type function applications: they are *not* to be substituted in the body, as any conditional types or union types they carry will not be properly represented in the final type.

$$
\begin{aligned}
&(\Pi_{()}.\tau) = \tau \\
&([\Pi_{(\alpha_1,\ldots,\alpha_n)}.\tau]\ \tau_1 \ldots \tau_{i-1}\ \boxed{\tau_i}\ \tau_{i+1} \ldots \tau_n) = \\
&\quad \begin{cases}
\text{if } \tau_i = \tau_a \vee \tau_b & \Rightarrow \bigvee \begin{array}{l} ([\Pi_{(\alpha_1,\ldots,\alpha_n)}.\tau]\ \tau_1 \ldots \tau_{i-1}\ \boxed{\tau_a}\ \tau_{i+1} \ldots \tau_n) \\ ([\Pi_{(\alpha_1,\ldots,\alpha_n)}.\tau]\ \tau_1 \ldots \tau_{i-1}\ \boxed{\tau_b}\ \tau_{i+1} \ldots \tau_n) \end{array} \\
\text{if } \tau_i = (\tau_t\ ?{=}\ \tau_a) \cdot \tau_b & \Rightarrow (\tau_t\ ?{=}\ \tau_a) \cdot ([\Pi_{(\alpha_1,\ldots,\alpha_n)}.\tau]\ \tau_1 \ldots \tau_{i-1}\ \boxed{\tau_b}\ \tau_{i+1} \ldots \tau_n) \\
\text{if } \tau_i = \alpha & \Rightarrow ([\Pi_{(\alpha_1,\ldots,\alpha_{i-1},\alpha_{i+1},\ldots,\alpha_n)}.\tau[\alpha/\tau_i]\ \tau_1 \ldots \tau_{i-1}\ \tau_{i+1} \ldots \tau_n) \\
\text{if } \tau_i = \gamma & \Rightarrow ([\Pi_{(\alpha_1,\ldots,\alpha_{i-1},\alpha_{i+1},\ldots,\alpha_n)}.\tau[\gamma/\tau_i]\ \tau_1 \ldots \tau_{i-1}\ \tau_{i+1} \ldots \tau_n) \\
\text{if } \tau_i = \Pi_{(\beta_1,\ldots,\beta_m)}.\tau_b & \Rightarrow ([\Pi_{(\alpha_1,\ldots,\alpha_{i-1},\alpha_{i+1},\ldots,\alpha_n)}.\tau[\Pi_{(\beta_1,\ldots,\beta_m)}.\tau_b/\tau_i]\ \tau_1 \ldots \tau_{i-1}\ \tau_{i+1} \ldots \tau_n)
\end{cases}
\end{aligned}
$$

Figure 2.5: Reduction rules for type fucntion application

As an example, we show how the following type function application is reduced:

$$
\begin{aligned}
&(\Pi_{(\alpha_x)}.\overbrace{[(\mathsf{int}\ ?{=}\ \alpha_x) \cdot \mathsf{int}]}^{\text{type function body}}\ \overbrace{[(\mathsf{string}\ ?{=}\ \alpha_a) \cdot \alpha_a] \vee (\alpha_c\ \mathsf{int}\ \mathsf{string})}^{\text{argument}}) \\
&= (\Pi_{(\alpha_x)}.[(\mathsf{int}\ ?{=}\ \alpha_x) \cdot \mathsf{int}]\ [(\mathsf{string}\ ?{=}\ \alpha_a) \cdot \alpha_a]) \vee (\Pi_{(\alpha_x)}.[(\mathsf{int}\ ?{=}\ \alpha_x) \cdot \mathsf{int}]\ (\alpha_c\ \mathsf{int}\ \mathsf{string})) \\
&= [(\mathsf{string}\ ?{=}\ \alpha_a) \cdot (\Pi_{(\alpha_x)}.[(\mathsf{int}\ ?{=}\ \alpha_x) \cdot \mathsf{int}]\ \alpha_a)] \vee (\Pi_{(\alpha_x)}.[(\mathsf{int}\ ?{=}\ \alpha_x) \cdot \mathsf{int}]\ (\alpha_c\ \mathsf{int}\ \mathsf{string})) \\
&= [(\mathsf{string}\ ?{=}\ \alpha_a) \cdot (\mathsf{int}\ ?{=}\ \alpha_a) \cdot \mathsf{int}] \vee (\Pi_{(\alpha_x)}.[(\mathsf{int}\ ?{=}\ \alpha_x) \cdot \mathsf{int}]\ (\alpha_c\ \mathsf{int}\ \mathsf{string}))
\end{aligned}
$$

Note that the $(\alpha_c\ \mathsf{int}\ \mathsf{string})$ argument is not substituted into the type function application. Further reduction of this type can only happen once α_c is known.

Inference The rules for inferring types from a program are given in figure 2.6. Each rule is of the form $\Gamma \vdash \mathsf{e} : \tau$, meaning that the type environment Γ assigns a certain type τ to expression e. Rules T-LET and T-APPLY use helper functions, these are defined in figure 2.7. We go over these rules one by one:

T-const Constants and primitive operations are assigned a type using the conventional Typeof function.

T-var Types of variables are looked up in the environment.

T-if The type of a conditional expression is the union of the types of the two branches. If types of the branches happen to be equal, only one type is returned.

T-let The inference rule for let expressions is at its core the same as for other type systems: the type of the expression e_x is inferred first and x is bound to this type when the body is inferred. However, the type of e_x might contain a conditional type, so the Leaves function is used to strip away the conditional types, leaving only the concrete types at the leaves of the returned type tree. Intuitively, this

$\boxed{\Gamma \vdash \mathsf{e} : \tau}$

$$\frac{}{\Gamma \vdash \mathsf{c} : \mathsf{Typeof}(\mathsf{c})} \quad (\text{T-CONST}) \qquad \frac{\Gamma(\mathsf{x}) = \tau}{\Gamma \vdash \mathsf{x} : \tau} \quad (\text{T-VAR})$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : \tau_1 \qquad \Gamma \vdash \mathsf{e}_2 : \tau_2}{\Gamma \vdash (\mathsf{if\ s\ e}_1\ \mathsf{e}_2) : \tau_1 \vee \tau_2} \quad (\text{T-IF})$$

$$\frac{\Gamma \vdash \mathsf{e}_\mathsf{x} : \tau_x \qquad \Gamma, \mathsf{x} : \tau_L \vdash \mathsf{e} : \tau \qquad \tau_L = \mathsf{Leaves}(\tau_x)}{\Gamma \vdash (\mathsf{let}\ ([\mathsf{x}\ \mathsf{e}_\mathsf{x}])\ \mathsf{e}) : (\Pi_{(\alpha_x)}.[\tau]\ \tau_x)} \quad (\text{T-LET})$$

$$\frac{\Gamma, \mathsf{x}_1 : \alpha_1, \ldots, \mathsf{x}_n : \alpha_n \vdash \mathsf{e} : \tau \qquad \alpha_1, \ldots, \alpha_n \text{ fresh}}{\Gamma \vdash (\mathsf{lambda}\ (\mathsf{x}_1 \ldots \mathsf{x}_n)\ \mathsf{e}) : \Pi_{(\alpha_1 \ldots \alpha_n)}.\tau} \quad (\text{T-LAMBDA})$$

$$\frac{\Gamma \vdash \mathsf{s}_i : \tau_i \quad \forall i \in [0 \ldots n] \qquad \ell_f = \mathsf{Label}((\mathsf{s}_0\ \mathsf{s}_1 \ldots \mathsf{s}_n)) \qquad \alpha_1, \ldots, \alpha_n \text{ fresh}}{\Gamma \vdash (\mathsf{s}_0\ \mathsf{s}_1 \ldots \mathsf{s}_n) : (\Pi_{(\alpha_1 \ldots \alpha_n)}.[\mathsf{Apply}(\tau_0, \ell_f, \alpha_1 \ldots \alpha_n)]\ \tau_1 \ldots \tau_n)} \quad (\text{T-APPLY})$$

Figure 2.6: Type inference rules

$$\begin{aligned}
\mathsf{Apply}(\Pi_{(\alpha_1 \ldots \alpha_m)}.\tau_f, \ell_f, \tau_1 \ldots \tau_n) &= \tau_f[\tau_1 \ldots \tau_n / \alpha_1 \ldots \alpha_m][\ell_{\tau_1} \ldots \ell_{\tau_m} / \ell_{\alpha_1} \ldots \ell_{\alpha_m}] \quad \text{if } m = n \\
\mathsf{Apply}(\tau_\alpha \vee \tau_\beta, \ell_f, \tau_1 \ldots \tau_n) &= \mathsf{NoError}(\mathsf{Apply}(\tau_\alpha, \ell_f, \tau_1 \ldots \tau_n), \mathsf{Apply}(\tau_\beta, \ell_f, \tau_1 \ldots \tau_n)) \\
\mathsf{Apply}(\Pi_{(\alpha_1 \ldots \alpha_m)}.\tau_f, \ell_f, \tau_1 \ldots \tau_n) &= (\text{function-arity-n ?= } \Pi_{(\alpha_1 \ldots \alpha_m)}.\tau_f)^{\ell_f}_{\ell_f} \cdot \mathsf{error}(\ell_f) \quad \text{if } m \neq n \\
\mathsf{Apply}(\alpha, \ell_f, \tau_1 \ldots \tau_n) &= (\text{function-arity-n ?= } \alpha)^{\ell_f}_{\ell_\alpha} \cdot (\alpha\ \tau_1 \ldots \tau_n) \\
\mathsf{Apply}(\gamma, \ell_f, \tau_1 \ldots \tau_n) &= (\text{function-arity-n ?= } \gamma)^{\ell_f}_{\ell_f} \cdot \mathsf{error}(\ell_f)
\end{aligned}$$

$$\begin{aligned}
\mathsf{Leaves}(\tau_1 \vee \tau_2) &= \mathsf{Leaves}(\tau_1) \vee \mathsf{Leaves}(\tau_2) \\
\mathsf{Leaves}((\tau_t \text{ ?= } \tau) \cdot \tau_1) &= \mathsf{Leaves}(\tau_1) \\
\mathsf{Leaves}(\tau) &= \tau
\end{aligned}$$

$$\begin{aligned}
\mathsf{NoError}(\mathsf{error}(\ell_f), \mathsf{error}(\ell_g)) &= \mathsf{error}(\{\ell_f, \ell_g\}) \\
\mathsf{NoError}(\mathsf{error}(\ell_f), \tau_\beta) &= \tau_\beta \\
\mathsf{NoError}(\tau_\alpha, \mathsf{error}(\ell_f)) &= \tau_\alpha \\
\mathsf{NoError}(\tau_\alpha, \tau_\beta) &= \tau_\alpha \vee \tau_\beta
\end{aligned}$$

Figure 2.7: Auxiliary definitions

reflects the fact that any type tests have already been done by the time the let body is entered. The type of the body is then inferred with a type union of the concrete types bound to the variable x.

When constructing the type for the *whole* let expression, care must be taken to ensure that any type tests performed by the expression are not discarded (remember listing 2.1). This is done using an immediately-applied type function: according to the reduction rules for applied type functions (figure 2.5), conditional types are pushed outwards and the body is duplicated for union types. Type variables and concrete types are substituted as normal, only type function applications are not reduced.

Leaves is defined in figure 2.7.

T-lambda The body of a lambda is first inferred with fresh type variables bound to its parameters. These type variables are then bound by a type function constructor.

T-apply This last rule is the most complex. First it infers the types of the function and its arguments, then it uses the Label function to find the label for the application expression. These are passed to the Apply function, which traverses the type tree τ_f. The cases in the Apply function (figure 2.7) are as follows:

1. The first case covers the application of a type function to the correct number of arguments. Apply then substitutes the arguments and their labels for the corresponding variables and labels in the type function body.
2. For the application of a union type, Apply tries to apply the types on both sides to the given arguments. The NoError helper function (defined at the bottom of figure 2.7) removes branches that return error. If all branches are error, the entire type is error.
3. The next case is concerned with application of a type function with the wrong number of arguments.
4. Applying a type variable yields a conditional type with a type function application in its body, as explained above.
5. Finally, trying to apply a ground type as a function is an error, but a conditional type is still generated.

Check inference at work We work through a small example (shown in listing 2.3) to demonstrate check inference. The centerpoint of this example is the `partial` function which partially applies the given function to a given argument.

First, the type of `partial` is inferred. Following T-LAMBDA, the parameters `f` and `x` are bound to type variables α_f and α_x. The body of `partial` is again a function, so `y` is bound to the type variable α_y. The next applicable rule is T-APPLY, which

```
1 (let ([partial (lambda (f x)
2        (lambda (y)
3         (f x y)))])
4   (let ([g (partial + "hello")])
5     (g 9)))
```

Listing 2.3: Example program for check inference

invokes $\mathsf{Apply}(\alpha_f, \ell, \alpha_x, \alpha_y)$, where ℓ references the source location of the application expression. The fourth case of the Apply function is applicable, which returns a type function application guarded by a conditional type. We can thus construct the type of `partial` as

$$\tau_{\mathsf{partial}} = \Pi_{(\alpha_f, \alpha_x)}.[\Pi_{(\alpha_y)}.[(\mathsf{function\text{-}arity\text{-}2} \mathbin{?=} \alpha_f) \cdot (\alpha_f\ \alpha_x\ \alpha_y)]]$$

Next, the type of `g` is constructed by filling in the type variables with the types of the arguments. This time, a known function is applied, so we enter the first case of Apply, which replaces the type variables α_f and α_x with the types of `+` and `"hello"`, respectively τ_+ and string, with

$$\tau_+ = \Pi_{(\alpha_w, \alpha_z)}.(\mathsf{int} \mathbin{?=} \alpha_w) \cdot (\mathsf{int} \mathbin{?=} \alpha_z) \cdot \mathsf{int}$$

Note that we have renamed the type variables in the type of `+` to avoid variable capture. The type of `g` is

$$\begin{aligned}\tau_{\mathsf{g}} &= \Pi_{(\alpha_y)}.(\mathsf{function\text{-}arity\text{-}2} \mathbin{?=} \tau_+) \cdot (\tau_+\ \mathsf{string}\ \alpha_y) \\ &= \Pi_{(\alpha_y)}.(\tau_+\ \mathsf{string}\ \alpha_y) \\ &= \Pi_{(\alpha_y)}.(\mathsf{int} \mathbin{?=} \mathsf{string}) \cdot (\mathsf{int} \mathbin{?=} \alpha_y) \cdot \mathsf{int}\end{aligned}$$

Finally, the type of the application `(g 9)` becomes

$$\begin{aligned}\tau_{(\mathtt{g\ 9})} &= (\mathsf{int} \mathbin{?=} \mathsf{string}) \cdot (\mathsf{int} \mathbin{?=} \mathsf{int}) \cdot \mathsf{int} \\ &= (\mathsf{int} \mathbin{?=} \mathsf{string}) \cdot \mathsf{int}\end{aligned}$$

Unlike traditional type systems, the program is not rejected even though it clearly contains a type error. The expression might be part of a function that is never executed, or it might reside in a branch of an if expression that is never taken. Instead, the faulty expression will be guarded by a run-time type test, which raises an error only if the containing function or branch are entered. Run-time type test are introduced by the check introduction stage in the next section.

2.3 Check Introduction

The result of check inference is a final type environment Γ which associates each expression in the program with a type. This stage will insert check expressions around

all function application expressions. The preconditions of these check expressions are derived from the conditional types (if any) of the function applications.

Conversion to preconditions The algorithm for converting conditional types to preconditions is defined by the function ToPrec, shown in figure 2.8.

$$
\begin{aligned}
\mathsf{ToPrec}((\tau_t \mathbin{?=} \tau_c)^{\ell_b}_{\ell_c} \cdot \tau) &= \mathsf{ToPrec}(\tau) && \text{if } \tau_t \cong \tau_c \\
\mathsf{ToPrec}((\tau_t \mathbin{?=} \tau_c)^{\ell_b}_{\ell_c} \cdot \tau) &= (\tau_t?\ \mathsf{s})^{\{\ell_b\}} \wedge \mathsf{ToPrec}(\tau) && \text{where } \mathsf{s} = \mathrm{Expr}(\ell_c) \\
\mathsf{ToPrec}(\tau_1 \vee \tau_2) &= \mathsf{ToPrec}(\tau_1) \vee \mathsf{ToPrec}(\tau_2) \\
\mathsf{ToPrec}(\tau) &= \#\mathsf{t}
\end{aligned}
$$

Figure 2.8: Conversion from types to check preconditions

The first two rules here are the most important: the first prevents the generation of preconditions where types are statically known and equal, such as $(\mathsf{int} \mathbin{?=} \mathsf{int})$. We use the $\cong$ operator instead of strict equality, such that function-arity-2 $\cong \Pi_{(\alpha_1,\alpha_2)}.\tau$ also holds. The second rule actually generates preconditions: it first uses the Expr function to look up the variable or constant associated with cause label ℓ_c . The condition part of the conditional type is then converted into a predicate $\tau_t?$ on that variable or constant. This predicate assigns blame to the labels in the singleton set $\{\ell_b\}$. The resulting precondition is then the conjunction of this predicate and the precondition for the rest of the conditional type. Note that conditional types are associated with *one* label, while every predicate is associated with a *set* of labels. In the case of ToPrec, these are singleton sets. For a union type, the precondition is simply the disjunction of preconditions. Other types such as ground types, type variables and type functions are converted to the no-op precondition #t, as they do not result in a run-time check.

For a conditional type, the type test is derived from τ_t: there exists a corresponding type test function for every ground type. For example, $(\mathsf{int} \mathbin{?=} \tau)$ becomes $(\mathsf{int}?\ \tau)$. If τ_t is of the form function-arity-n, the run time test checks whether the argument is a function of arity n using the function-arity-n predicate. Remember that such types occur when a type variable is applied to arguments.

Check introduction Having defined how types are converted to preconditions, we now explain how and where check expressions are inserted into the program. The Insert function for check introduction can be seen in figure 2.9. It is applied to every expression in the program, with the type environment Γ from the previous stage as input. We introduce checks only around function applications; other expressions are left alone. For each function application, we look up its type in the environment Γ and convert it into a precondition using the ToPrec function. The function application is then wrapped with a check expression. Even if function application does not give rise

to a conditional type, a (check #t e) is still generated. This simplifies the rules in the next stage.

$$\mathsf{Insert}(\Gamma, \mathsf{e}) = \begin{cases} (\mathsf{check\ p\ e}) & \text{with } \mathsf{p} = \mathsf{ToPrec}(\Gamma(\mathsf{e})) \quad \text{if e is of the form} (\mathsf{s\ s}_1 \ldots \mathsf{s}_n) \\ \mathsf{e} & \text{otherwise} \end{cases}$$

Figure 2.9: Introduction of check expressions into the program

At the end of this stage, every function application in the input program is guarded by a check expression.

2.4 Check Mobility

In this stage of the transformation, check expressions in the program are moved (or "floated") upwards in the program tree. The rules for doing so are described in fig-

$$\frac{\mathsf{e} \to \mathsf{e}' \uparrow \mathsf{p}}{\mathsf{e} \to_P (\mathsf{check\ p\ e}')} \quad (\text{F-PROGRAM}) \qquad \mathsf{c} \to \mathsf{c} \uparrow \#\mathsf{t} \quad (\text{F-CONST})$$

$$\mathsf{x} \to \mathsf{x} \uparrow \#\mathsf{t} \quad (\text{F-VAR}) \qquad \frac{\mathsf{s}_i \to \mathsf{s}_i' \uparrow \#\mathsf{t} \qquad \forall i \in 1 \ldots n}{(\mathsf{check\ p}\ (\mathsf{s}_1 \ldots \mathsf{s}_n)) \to (\mathsf{s}_1' \ldots \mathsf{s}_n') \uparrow \mathsf{p}} \quad (\text{F-APPLY})$$

$$\frac{\mathsf{s} \to \mathsf{s}' \uparrow \#\mathsf{t} \qquad \mathsf{e}_1 \to \mathsf{e}_1' \uparrow \mathsf{p}_1 \qquad \mathsf{e}_2 \to \mathsf{e}_2' \uparrow \mathsf{p}_2}{(\mathsf{if\ s\ e}_1\ \mathsf{e}_2) \to (\mathsf{if\ s\ (check\ p}_1\ \mathsf{e}_1')\ (\mathsf{check\ p}_2\ \mathsf{e}_2')) \uparrow \mathsf{p}_1 \vee \mathsf{p}_2} \quad (\text{F-IF})$$

$$\frac{\mathsf{e}_\mathsf{x} \to \mathsf{e}_\mathsf{x}' \uparrow \mathsf{p}_\mathsf{x} \qquad \mathsf{e} \to \mathsf{e}' \uparrow \mathsf{p} \qquad \mathsf{p}' = \mathsf{mask}(\mathsf{x}, \mathsf{p})}{(\mathsf{let}\ ([\mathsf{x}\ \mathsf{e}_\mathsf{x}])\ \mathsf{e}) \to (\mathsf{let}\ ([\mathsf{x}\ \mathsf{e}_\mathsf{x}'])\ (\mathsf{check\ p\ e}')) \uparrow \mathsf{p}_\mathsf{x} \wedge \mathsf{p}'} \quad (\text{F-LET})$$

$$\frac{\mathsf{e} \to \mathsf{e}' \uparrow \mathsf{p}}{(\mathsf{lambda}\ (\mathsf{x}_1 \ldots \mathsf{x}_n)\ \mathsf{e}) \to (\mathsf{lambda}\ (\mathsf{x}_1 \ldots \mathsf{x}_n)\ (\mathsf{check\ p\ e}')) \uparrow \#\mathsf{t}} \quad (\text{F-LAMBDA})$$

$$\begin{aligned} \mathsf{mask}(\mathsf{x}, (\tau?\ \mathsf{x}')^{\ell_b}) &= (\tau?\ \mathsf{x}')^{\ell_b} && \text{if } \mathsf{x} \neq \mathsf{x}' \\ \mathsf{mask}(\mathsf{x}, (\tau?\ \mathsf{x}')^{\ell_b}) &= \#\mathsf{t} && \text{if } \mathsf{x} = \mathsf{x}' \\ \mathsf{mask}(\mathsf{x}, \mathsf{p}_1 \vee \mathsf{p}_2) &= \mathsf{mask}(\mathsf{x}, \mathsf{p}_1) \vee \mathsf{mask}(\mathsf{x}, \mathsf{p}_2) \\ \mathsf{mask}(\mathsf{x}, \mathsf{p}_1 \wedge \mathsf{p}_2) &= \mathsf{mask}(\mathsf{x}, \mathsf{p}_1) \wedge \mathsf{mask}(\mathsf{x}, \mathsf{p}_2) \\ \mathsf{mask}(\mathsf{x}, \#\mathsf{t}) &= \#\mathsf{t} \end{aligned}$$

Figure 2.10: Rules for moving checks up the evaluation tree

ure 2.10. Check mobility rules are of the form $e \to e' \uparrow p$, with a reading of "expression e is rewritten to expression e', propagating precondition p upwards". The only difference between e and e' is the position of check expressions and their preconditions.
The rules of figure 2.10 are as follows:

- Check mobility is initiated by the F-PROGRAM rule[2], which invokes the check mobility rules on the top-level expression. Any preconditions propagated upwards from the root expression are captured and inserted at the top of the program.

- Rules F-CONST and F-VAR are for completeness: constants and variables will never have preconditions, as check expressions are only introduced at function application sites.

- The F-APPLY rule rewrites a function application to itself while propagating the precondition of its associated check upwards. As we converted the program to ANF in the first stage of the blame prediction transformation, the arguments are all simple expressions. Therefore, they are either constants, variables or lambda terms, which cannot contribute preconditions.

- Conditional expressions (the F-IF rule) combine two paths through the program, which can have different expectations from their environment. For example, one branch might expect variable `foo` to be a string, while the other might use `foo` as an int or not use it at all. Rule F-IF reconciles the differing expectations of the two branches by propagating the disjunction $p_1 \vee p_2$ of the two preconditions upwards. Each branch of the if is still guarded by its own check (respectively p_1 and p_2).

- In the rule F-LET, there are two sources of preconditions: the expression e_x bound to the variable x and the body e. Since e_x will be evaluated before the body, its preconditions p_x are propagated upwards without changes. By contrast, the preconditions propagated upwards from the body may contain references to the bound variable x, which is unbound outside of the let expression. Therefore, we take care not to propagate these checks beyond the binding for x.

  The $\mathsf{mask}(x, p)$ function traverses the structure of a precondition and replaces all type tests on the variable x with the no-op precondition #t. The resulting precondition p' is propagated upwards in conjunction with p_x. Additionally, the returned expression checks the precondition p again at the top of the let body, such that any preconditions on x are checked there. This introduces some duplication of checks, but the next stage will remove some of them again.

[2]This rule is a separate rule for programs, indicated by the use of $\to_p$.

- Rule F-LAMBDA captures the preconditions propagated from its body and inserts them again at the top of the body. This is important to preserving the program behavior: preconditions must be verified when anonymous functions are *executed*, not when they are defined.

Example of check mobility We clarify the F-LET rule by applying check mobility to the small example in listing 2.4. Note that the number? type test for `tmp1` was never generated, as the type system asserts that `string-length` returns a number.

```
1 (let ([x (generate-x)])
2   (let ([tmp1 (check [string? x] (string-length x))])
3     (check [number? y] (+ y tmp1))))
```

Listing 2.4: Example of check mobility: input program

Applying F-APPLY to the calls to `string-length` and `+` moves their respective checks upwards. Applying rule F-LET for the inner let yields listing 2.5. The preconditions on x and y are simply moved up and out of the inner let, as they do not mention `tmp1`.

```
1 (let ([x (generate-x)])
2   (check [string? x ∧ number? y]
3     (let ([tmp1 (string-length x)])
4       (check [number? y] (+ y tmp1)))))
```

Listing 2.5: Example of check mobility: after floating the inner let

Applying check mobility to the outer let yields the code in listing 2.6. This time, the preconditions to be moved mention the bound variable x, so in the outer check it has been replaced by #t. The original precondition is left at the top of the let body; it will be eliminated by the simplification stage. The duplicate number? checks on lines 3 and 5 will also be eliminated, as number? y has already been checked on line 1.

```
1 (check [#t ∧ number? y]
2   (let ([x (generate-x)])
3     (check [string? x ∧ number? y]
4       (let ([tmp1 (string-length x)])
5         (check [number? y] (+ y tmp1))))))
```

Listing 2.6: Example of check mobility: After floating the outer let

2.5 Check Simplification

The transformation presented above implements the core idea of blame prediction: it makes preconditions of primitive operations (whether applied directly or indirectly) explicit and subsequently moves them upwards in the program. In the resulting program a number of preconditions are checked multiple times, however. Obviously, reducing the number of check expressions has a big impact on the run-time performance

of a blame predicted program. The goal of this section is therefore to present a number of simplifications which reduce the number of checks in the program without altering its semantics.

Each subsection will present a different simplification, but they can be applied in any order. For example, removing a precondition from a check expression (section 2.5.3) can enable an or–true simplification (section 2.5.1). For convenience we omit the braces around single labels (i. e. $(\texttt{number? n})^{<}$ instead of $(\texttt{number? n})^{\{<\}}$).

The running example for this section can be found in listing 2.7. It is the result of applying the previous stages of the blame prediction transformation to the well-known `factorial` function. The arithmetic operations in the alternative of the **`if`** expression give rise to several number? preconditions, labeled with the operation that generated them.

```
1 (define (factorial n)
2   (check [number? n^<
3           ∧ (#t ∨ number? n^− ∧ number? n^*)]
4     (if (< n 1)
5       1
6       (check [number? n^− ∧ number? n^*]
7           (let ([tmp-fac (factorial (- n 1))])
8             (* n tmp-fac))))))
```

Listing 2.7: Running example: factorial

2.5.1 Or–true simplification

The first simplification removes preconditions that have #t in one branch of a disjunction. These arise as a result of conditional expressions that do not perform any type tests in one of the branches, such as the one on line 4. Since a precondition with $\texttt{\#t} \vee \texttt{p}$ will always be true regardless of p, the precondition can simply be replaced by #t. This does not alter the program semantics, as the branch that *does* perform a type test still has a check expression at the head, as shown by check mobility rule F-IF. The result can be seen in listing 2.8.

```
1 (define (factorial n)
2   (check [number? n^<]
3     (if (< n 1)
4       1
5       (check [number? n^− ∧ number? n^*]
6           (let ([tmp-fac (factorial (- n 1))])
7             (* n tmp-fac))))))
```

Listing 2.8: Example after applying or–true simplification

2.5.2 And–check simplification

The second simplification that can be performed removes duplicate predicates in the same precondition. In the running example (listing 2.8), the second check expression tests (number? n) two times. Duplicate predicates in a conjunction can be removed, taking care to preserve the blame labels in case the check expression fails. If a duplicate predicate is removed, its blame labels are added to the first remaining predicate. The result is shown in listing 2.9.

```
1 (define (factorial n)
2   (check [number? n^<]
3     (if (< n 1)
4       1
5       (check [number? n^{-,*}]
6           (let ([tmp-fac (factorial (- n 1))])
7             (* n tmp-fac))))))
```

Listing 2.9: Example after applying and–check simplification

2.5.3 Check–check simplification

The final simplification opportunity lies in eliminating repeated preconditions across check expressions. Continuing with the example in listing 2.9, there are two checks that verify whether the variable n contains a number. Moreover, the second check (line 5) is in the scope of the first check (line 2), meaning that the same property is verified twice. The inner check can thus safely be removed. The result of this simplification can be seen in listing 2.10. This does not change the semantics of the program, as evaluation

```
1 (define (factorial n)
2   (check [number? n^{<,-,*}]
3     (if (< n 1)
4       1
5       (let ([tmp-fac (factorial (- n 1))])
6         (* n tmp-fac)))))))
```

Listing 2.10: Example after check–check simplification

will already predict blame at the first check if n is not a number. Again, care must be taken to preserve blame information: we must also merge the sets of blame labels in nested checks into outer checks.

2.5.4 Note on simplification of failing tests

It is tempting to simply substitute #f for failing tests on literal expressions such as (string? 9), as logically speaking these behave the same. However, this would have

two immediate consequences: first, this #f would not identify the actual primitive operation that failed, and multiple blame labels would be assigned to the precondition "#f", confusing the programmer. The second consequence is worse: the semantics of programs would change! Consider the expression

```
(if decision (+ 3 "hello") (string-length x))
```

Applying blame prediction results in the precondition $(\mathsf{number?}\ \texttt{"hello"}) \vee (\mathsf{string?}\ \mathsf{x})$. Substituting #f for the number? test yields $\#\mathsf{f} \vee (\mathsf{string?}\ \mathsf{x})$, which could then be simplified to $(\mathsf{string?}\ \mathsf{x})$. This simplification disregards the role of `decision`: it predicts blame if x is not a string.

2.6 Correctness

In order to prove that the blame prediction transformation does not change program semantics, we will prove two equivalences[3] between an original program P and its transformation P′. Recall from section 2.1 that $\mathsf{P} \rightsquigarrow \mathsf{v}$ means that program P evaluates to value v, and similarly $\mathsf{P} \rightsquigarrow \text{err-}\omega$ means that evaluation of the program P results in an error err-ω.

Value Preservation
Iff P runs to completion and produces a value v, P′ produces the same value v.
Formally: $\mathsf{P} \rightsquigarrow \mathsf{v} \Leftrightarrow \mathsf{P}' \rightsquigarrow \mathsf{v}$.
Use–Blame Entailment
If P raises an error, the blame predicted program P′ must predict blame.
Formally: $\mathsf{P} \rightsquigarrow \text{err-}\omega \Rightarrow \mathsf{P}' \rightsquigarrow \text{err-blame}(\mathsf{p})$.
Blame–Use Entailment
If P′ predicts blame, the original program P must raise an error.
Formally: $\mathsf{P}' \rightsquigarrow \text{err-blame}(\mathsf{p}) \Rightarrow \mathsf{P} \rightsquigarrow \text{err-}\omega$.

We prove these properties through *trace equivalence*, where the traces consist of the expressions encountered along the path through the input program P and the blame predicted program P′. The preconditions of primitive operations which *might fail* are represented by "use(p)" expressions in a trace.
For example, the expression `(+ 3 "hello")` with precondition $(\mathsf{number?}\ \texttt{"hello"})$ manifests in a trace as

$$[\ldots, \mathsf{use}(\mathsf{number?}\ \texttt{"hello"}), \ldots]$$

Definition 1 (Traces)**.** We define a trace T for a program P as a sequence of check and use expressions, as they are encountered while evaluating a program. Scheme$_\beta$ programs that return a value or raise an error must have finite traces.

[3]The second equivalence is constructed from two entailments.

Definition 2 (Equivalence modulo check)**.** Two Scheme$_\beta$ programs are **equivalent modulo** check iff the syntax trees are identical after replacing check expressions with their body, i. e. $(\mathsf{check}\ \mathsf{p}\ \mathsf{e}) \mapsto \mathsf{e}$. This equivalence extends to traces: if two programs are equivalent modulo check, removing the check nodes from their traces results in identical traces.

Next, we define four lemmas that will be used in the proofs of the equivalences above. Proofs for these lemmas will follow in section 2.7.

Lemma 1. Verifying the preconditions p of a $(\mathsf{check}\ \mathsf{p}\ \mathsf{e})$ expression in a well-formed program never gives rise to an error.

Lemma 2. A blame-predicted program P′ is equivalent modulo check to its input program P.

Lemma 3. In a blame-predicted program P′, every primitive operation which relies on a precondition p is in the scope of a $(\mathsf{check}\ \mathsf{p} \wedge \mathsf{p}'\ \ldots)$ expression.

Lemma 4. A blame-predicted program P′ only contains check expressions that reference preconditions from the primitive operations in their scope, or conjunctions and disjunctions of such preconditions.

Proof of Value Preservation

Proof. We prove each direction separately.
$\Rightarrow$: Given that $\mathsf{P} \rightsquigarrow \mathsf{v}$, prove that $\mathsf{P}' \rightsquigarrow \mathsf{v}$.
The successful evaluation of P yields a trace

$$\mathsf{T} = [\mathsf{use}(\mathsf{p}_1), \mathsf{use}(\mathsf{p}_2), \ldots, \mathsf{use}(\mathsf{p}_n)]$$

where every p_i must be true. Therefore, the trace describes properties on variables in the program, such as $(\mathsf{number?}\ \mathsf{x})$ or $(\mathsf{string?}\ \mathsf{y})$. Moreover, it must be internally consistent, so a variable x cannot be both a number? and a string?. Because Scheme$_\beta$ does not allow mutation of variables, a precondition tested in a use expression must be true for the entire lifetime of the variable.

We generate a hypothetical trace T′ for P′ by walking the same path as program P and recording the checks encountered. In order for P′ to result in a value, every use and check in T′ must succeed. T′ is equivalent modulo check to T, so only the checks make T′ differ from T. For every element of T′, there are two possibilities:

1. $\mathsf{use}(\mathsf{p}_i)$: These are also present in T, where all uses succeeded.
2. $\mathsf{check}(\mathsf{p}_i)$: From lemma 4, we know that the checks in P′ are conjunctions and disjunctions of the uses in P. Since every use in T succeeded, this check must also succeed. Moreover, the evaluation of the precondition will not raise an error because of lemma 1.

Since all checks and uses in T' succeed, the evaluation of P' must also succeed and produce the same value. □

$\Leftarrow$: Given that $\mathsf{P}' \rightsquigarrow v$, prove that $\mathsf{P} \rightsquigarrow v$.
The trace T' of program P' is equivalent modulo check to the trace T of program P. In other words, removing the checks from T' yields T. T' succeeds, so T must also succeed, and thus program P must produce the same value as program P'. □

Proof of Use-Blame Entailment

Proof. As described by the evaluation rules, the only expressions in the program P that can raise errors are primitive operations. Let

$$\mathsf{T} = [\mathsf{use}(\mathsf{p}_1), \mathsf{use}(\mathsf{p}_2), \ldots, \mathsf{use}(\mathsf{p}_k)]$$

be the trace of the program P up to and including the $\mathsf{use}(\mathsf{p}_k)$ that raised the error.

Since the trace T' of P' is equivalent modulo check to T, $\mathsf{use}(\mathsf{p}_k)$ must also be a member of T'. Lemma 3 states that $\mathsf{use}(\mathsf{p}_k)$ must be in the scope of a $\mathsf{check}(\mathsf{p}_j)$ which critically depends on p_k ($\neg\mathsf{p}_k \implies \neg\mathsf{p}_j$). Upon reaching $\mathsf{check}(\mathsf{p}_j)$, evaluation of program P' will definitely stop, as p_k does not hold. However, it is possible that another check in the trace fails even earlier. Regardless of which check actually raises the error, the program stops and predicts blame. □

Proof of Blame-Use Entailment

Proof. P' evaluates to an $\mathsf{err\text{-}blame}(\mathsf{p})$ error, which must be raised by a $\mathsf{check}(\mathsf{p}_k)$ expression in P'. The traces up to that point for P' and P are, respectively

$$\mathsf{T}' = [\ldots, \mathsf{use}(\mathsf{p}_1), \ldots, \mathsf{use}(\mathsf{p}_j), \ldots, \mathsf{check}(\mathsf{p}_k)]$$
$$\mathsf{T} = [\mathsf{use}(\mathsf{p}_1), \ldots, \mathsf{use}(\mathsf{p}_j)]$$

There can be many paths that continue with T as prefix. Lemma 4 states that p_k is a combination of preconditions further in the program. Moreover, this lemma holds for *all* traces with T as prefix, so every such trace must contain at least one use for a precondition in p_k within its scope. In program P, these traces will fail as soon as they reach $\mathsf{use}(\mathsf{p}_k)$. □

Check mobility and ordering

For the two entailments we do *not* claim that a transformed program P' predicts the same error as raised by program P. The reason for this is that the blame prediction transformation does not take ordering of checks into account. For example, consider the blame predicted program in listing 2.11. Because of the structure of the program, the property `(number? x)` is verified before `(number? z)`.

```
1 (define (f x y)
2   (check (number? x)
3     (let ([z y])
4       (check (number? z)
5         (let ([y (+ z 1)])
6           (let ([w (/ x 2)])
7             ...))))))
```

Listing 2.11: Example of ordering inconsistencies after blame prediction

Assume `f` is called with two strings `(f "hello" "world")`: under blame prediction, the very first check expression on line 2 will fail and predict blame (err-blame(number? x)) for the / expression on line 6. The non-blame predicted program will raise a err-not-int(z) error when evaluating the call to + on line 5.

2.7 Proofs of lemmas

In this section we prove lemmas 1 to 4.

Lemma 1: Verifying preconditions of a check never raises an error in a well-formed program

Proof. Evaluation of the precondition of a check expression can only go wrong if the referenced variables are unbound. We consider the check introduction and mobility stages as potential sources of references to unbound variables:

1. The definition of well-formed programs in section 2.1 forbids free variables and therefore unbound variables. Therefore, variable references in the program can only appear in the scope of a let or lambda expression higher up in the program. The check introduction stage inserts check expressions directly around function application expressions. Any variable in a precondition must come from the application expression it wraps, so the precondition cannot contain unbound variable references.

2. The check mobility stage can potentially lift a precondition out of the scope in which it is defined. We perform case analysis on the check mobility rules (figure 2.10), especially for the let and lambda forms:

   a) In rule F-LET, the preconditions p from the body e can reference the bound variable x. The preconditions floated up from a let are filtered through the mask function, which prevents unbound references to x by replacing them with #t.

   b) Rule F-LAMBDA simply prevents preconditions from floating out of the lambda body, so unbound references cannot occur.

c) The other check mobility rules only move existing preconditions of check expressions upwards, so they cannot cause references to be unbound.

□

Lemma 2: Equivalence modulo check **after transformation**

Proof. For every stage of the transformation, we consider the input program P and the program P' under transformation: At the start of the transformation, the programs are identical and thus trivially equivalent modulo check.

1. Check introduction: As per figure 2.9, check expressions are inserted around function application expressions. Removing these again yields the input program.
2. Check mobility: This stage moves check expressions around, as per figure 2.10. Removing all the check expressions from the flotation rules results in the identity transformation.
3. Check simplification: This stage again only merges or removes check expressions, other expressions are not touched. Therefore, after this stage the input program and the blame-predicted program are equivalent modulo check.

□

Lemma 3: every primitive operation is guarded by a check

Proof. The proof for this lemma is split in three parts: first we prove that the check introduction stage establishes the property to be proven. We then prove that this property is unaffected by check mobility or simplification.

Proof for check introduction The type system relates check expressions and primitive operations as follows: the type of a primitive operation corresponds to its internal preconditions, and check introduction generates checks based on that type. In this part of the proof we intend to show that the type inference stage correctly assigns ground types where types can be assigned at compile time, and that conditional or union types are assigned where types depend on the path taken by execution. We show this by induction on the inference rules from figure 2.6.

- Rules T-CONST and T-VAR are for constants and variables, evaluating these will not give rise to errors.
- Rule T-IF combines the types of both paths in a union type.
- The T-LET rule ensures that any checks made by the bound expression e_x are prefixed to those in the body.

- Rule T-LAMBDA wraps the type of the body in a type function. When this is later applied, this type is made accessible to the caller, along with any conditional types it contains.
- Rule T-APPLY is the base case: applications of primitive operations give rise to checks on the arguments. As stated above, these checks correspond with the use expressions inside the wrappers. If s_f is not a primitive operation however, the type of the function being applied may also contain conditional types, but their preconditions will be checked both at the call site and inside the function body. At run-time, the preconditions of the function will be checked at the top of the function body as well as call sites. This means that there will be more than one check for the primitive operations in the function body.

Proof for check mobility The previous stage has inserted check expressions around application expressions, so we need to prove that check mobility reinserts check expressions with these preconditions higher up in the tree. We will thus not consider F-CONST, F-VAR or F-APPLY, as they are leaf expressions in the expression tree.

- Rule F-IF captures the preconditions floated up by each branch and inserts them at the top of each branch.
- Rule F-LET has two preconditions to consider: p_x from e_x, and the p from e. p_x is simply floated upwards and captured higher up in the program, whereas p is inserted at the top of the let body.
- Rule F-LAMBDA captures the preconditions floated upwards from its body and inserts a check at the top.

Finally, Rule F-PROGRAM makes sure preconditions are reinserted at the top of the program. Every precondition floated up from a check is thus inserted again higher up in the program, therefore every use is still covered by at least one check.

Proof for check simplification Check simplification does not affect the property to be proven. We discuss every simplification strategy in turn:

- And–check simplification: duplicate predicates are removed from the precondition, but a single copy always remains.
- Or–true simplification: this simplification removes check expressions from the program. However, disjunctions of preconditions can only be generated from an if expression, which keeps copies of preconditions at the top of its branches.
- Check–check simplification: this stage removes duplicate preconditions in nested checks. Since the inner check is merged into the outer check, the precondition is still preserved and the property still holds.

We have shown that the three major stages of the blame prediction transformation introduce and maintain the property to be proven, so the property holds for the output of the transformation. □

Lemma 4: check preconditions only reference (combinations of) use **preconditions**

Proof. In the previous proof we already showed that the check introduction stage inserts check expressions that exactly match the preconditions of the primitive operations they guard. We therefore only need to prove that the check mobility stage generates checks from combinations of existing preconditions. Like the previous proof, we perform case analysis on the flotation rules. Almost all rules insert preconditions floated up from subexpressions, either directly (F-APPLY, F-LAMBDA, F-PROGRAM) or a combination of preconditions (F-IF).

This leaves the F-LET rule: the precondition p_x is floated upwards, in conjunction with the result of $\mathsf{mask}(x, p)$. The latter constructs a new precondition by removing parts of the precondition p floated up from the let body. The precondition $p_x \wedge \mathsf{mask}(x, p)$ thus only consists of preconditions floated up from subexpressions.

In conclusion, every rule satisfies the property we set out to prove. □

2.8 Discussion

In this section we discuss some extensions to our analysis.

2.8.1 Variable-arity functions

Scheme allows programmers to define functions of variable arity and call them just like regular functions. Such functions can have a number of required arguments, and any arguments after those are bound to a list. Listing 2.12 illustrates this: it defines a function `flexible` with fixed parameters `a` and `b`, with the remaining arguments bound to the variable `rest`. Some of the built-in functions (such as `+` and **`for-each`**) can also vary in their arity.

```
1 (define (flexible a b . rest)
2   (display (+ a b)) (newline)
3   (display rest) (newline))
4
5 ; First displays "6", then "(5 8 13)"
6 (flexible 2 4 5 8 13)
```

Listing 2.12: Example of a variable-arity function and its use

Adding support for variable-arity functions requires a few changes to our analysis. First, support for list types must be added such that types can be inferred for function bodies with variable arguments. Next, the type system must make a distinction

between fixed-arity and variable-arity functions: in case of a variable-arity function, the Apply and ToPrec helper functions must check if the number of arguments passed to the function is greater than or equal to the amount of fixed arguments. Finally, the built-in functions must receive a special type that performs type tests on all their arguments.

2.8.2 Supporting a full numeric tower in blame prediction

In section 2.2 we remarked that R5RS Scheme, on which Scheme_β is based on, only has a single numeric type number. Many other languages make a distinction between various kinds of numbers, typically arranged in a "numeric tower" where higher tiers are subsets of lower tiers. These towers resemble the numeric tower in math, where $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$.

In the Racket language, for example, the `vector-ref` primitive operation requires a `non-negative-integer?` for its index argument, which is a specific subset of the values which satisfy the `integer?` predicate. As another example, the `numerator` primitive operation requires a `rational?` argument and returns an `integer?`. Contrast this to our approach, which only checks for type *equality*, not type subsumption.

In order to introduce support for a numeric tower in the blame prediction transformation as described in this chapter, two changes have to be made. Firstly, in the check inference stage, conditional types such as $(\mathsf{rational}\ ?{=}\ \mathsf{int}) \cdot \tau$ must be allowed to simplify to just τ, as $\mathsf{int} \subset \mathsf{rational}$. Secondly, a similar change is needed for the check simplification stage: type subsumption must be used as criterium in the and–check simplification as well as the check–check simplification steps.

2.9 Conclusion

In this chapter we introduced the blame prediction transformation on Scheme_β, a functional Scheme-like language without recursion. This transformation annotates the input program with check expressions that make the preconditions of primitive operations explicit. Every primitive operation is now guarded by one or more check expressions which check its preconditions ahead of time. The check expressions are inserted into the program such that they report errors as early as possible, but only if an error is *guaranteed* to occur. This makes it a "must-fail" analysis

The blame prediction transformation consists of four stages:

Check inference uses a type system to infer types for all expressions in the program, in addition to the type tests which must be made "along the way".

Check introduction converts these type tests to the preconditions of check expressions inserted around function application expressions.

Check mobility moves these check expressions upwards, making sure no wrong errors are predicted;

Check simplification finally eliminates duplicate or ineffective checks.

We have proven that the blame prediction transformation does not alter the semantics of a program: if a program finishes successfully, its transformed version will also finish. The same cannot be said for programs that fail: we have proven that programs where blame is predicted *will* fail, although not necessarily with the same error. Vice versa, a normal program that fails will also fail after blame prediction, though again not with the same error.

In the next chapter of this thesis we expand the coverage of the blame prediction transformation with support for recursion. The addition of recursion introduces "infinite types", which need to be reduced to finite types before they can be used.

3

RECURSION

In the previous chapter we defined blame prediction for a functional core language Scheme$_\beta$. In this chapter we first extend the syntax and semantics of the language to add an explicit heap in order to support both recursion and mutable variables (section 3.1). Next, we augment the stages of the blame prediction transformation in order to support recursion (section 3.2).

3.1 Syntax and Semantics

Before we extend the blame prediction transformation with support for recursion and mutation (in the next chapter), we need to add syntax and semantics for these features to Scheme$_\beta$. Unfortunately, the semantics presented in the previous chapter eagerly substitute values for variables, for example the evaluation of `let` and `lambda` expressions. This makes it impossible to describe programs which perform assignments, as variables need to be changed after they have been substituted. Likewise, definition of recursive functions is impossible as functions cannot refer to themselves except by substituting their definition, which gives rise to infinitely large programs. Therefore we need new semantics for Scheme$_\beta$ that incorporates support for variable mutation and recursion. This semantics adds an explicit heap, which maps addresses to values, and an environment, which maps variables to addresses.

Syntax The extended syntax is presented in figure 3.1. Most of the syntax is the same as in figure 2.2, but the `letrec` and `set!` special forms are new. For recursion we define the `letrec` special form: the binding groups in the `letrec` keyword define variables $x_1, \ldots, x_n$ which are bound to expressions $e_1, \ldots, e_n$. The variables in the binding group $x_1, \ldots, x_n$ are in scope in every e_i as well as in the body. In order to support mutation, we add the `set!` keyword. `(set! x s)` assigns the value of the simple expression `s` to a previously defined variable `x`.

$\mathsf{e} \in \text{Exp}$	$::=$	s	Simple expressions
	$\mid$	$(\mathsf{s}\ \mathsf{s}_1 \ldots \mathsf{s}_n)$	Application
	$\mid$	$(\mathsf{if}\ \mathsf{s}\ \mathsf{e}\ \mathsf{e})$	Conditional
	$\mid$	$(\mathsf{let}\ ([\mathsf{x}\ \mathsf{e}])\ \mathsf{e})$	Let
	$\mid$	$(\mathsf{check}\ \mathsf{p}\ \mathsf{e})$	Check
	$\mid$	$(\mathsf{letrec}\ ([\mathsf{x}_1\ \mathsf{e}_1] \ldots [\mathsf{x}_n\ \mathsf{e}_n])\ \mathsf{e})$	New Recursive definition
	$\mid$	$(\mathsf{set!}\ \mathsf{x}\ \mathsf{s})$	New Variable assignment
	$\mid$	$(\mathsf{begin}\ \mathsf{e}_1 \ldots \mathsf{e}_n)$	New Expression sequencing
$\mathsf{s} \in \text{Simp}$	$::=$	x	Variables
	$\mid$	c	Constants and literals
	$\mid$	$(\mathsf{lambda}\ (\mathsf{x}_1 \ldots \mathsf{x}_n)\ \mathsf{e})$	Lambda expressions
$\mathsf{v} \in \text{Val}$	$::=$	$\mathsf{void} \mid \#\mathsf{f} \mid \#\mathsf{t} \mid \mathsf{n} \mid \langle \mathcal{E}, \mathsf{x}_1 \ldots \mathsf{x}_n, \mathsf{e} \rangle$	Run-time values
$\ell \in \text{Loc}$	$\subset$	$\mathbb{N}$	Memory locations
$\mathcal{E} \in \text{Env}$	$::=$	$\text{Var} \rightharpoonup \text{Loc}$	Environments
$\mathcal{H} \in \text{Heap}$	$::=$	$\text{Loc} \rightharpoonup \text{Val}$	Heaps

Figure 3.1: Extended syntax and auxiliary definitions

The second part of figure 3.1 defines environments and heaps in terms of *partial* functions from variables to memory locations (ℓ) and memory locations to values, respectively. These values can be one of the primitive boolean or integer values, or they can contain a closure. Both environments and heaps are partial functions, as they only contain mappings for bound variables resp. allocated heap cells. A closure is represented as a triple $\langle \mathcal{E}, \mathsf{x}_1 \ldots \mathsf{x}_n, \mathsf{e} \rangle$, where $\mathcal{E}$ is the captured environment, $\mathsf{x}_1 \ldots \mathsf{x}_n$ are the names of its formal parameters, and e is its body.

In order to simplify the presentation of the check inference rules, we require that letrec expressions only consist of groups of mutually recursive functions. However, the input program might include letrec expressions containing either non-function expressions or non-recursive function definitions. To enforce this invariant, we require a transformation such as defined in [Peyton Jones, 1987, Section 6.2.8] and Waddell et al. [2005]. After the transformation, expressions and non-recursive functions are bound in let expressions, and every recursive function in the original letrec is contained in the smallest possible binding group of mutually recursive functions. A worked out example, taken from Peyton Jones [1987], is shown in figure 3.3. Going further, we assume this transformation has taken place, and that the binding part of a letrec only contains one or more mutually recursive functions.

For the sake of simplicity (and presentation) we will also allow begin expressions into the language. A begin expression evaluates its expressions from left to right and returns the value of the last expression. Any begin expression can be translated to a set of nested let expressions, as shown in figure 3.4. In the last rule, the `ignore` variable

$$\mathcal{H}, \mathcal{E}, \mathsf{x} \rightsquigarrow \mathcal{H}, \mathcal{H}[\mathcal{E}[\mathsf{x}]] \quad \text{(E-VAR)} \qquad \mathcal{H}, \mathcal{E}, \mathsf{c} \rightsquigarrow \mathcal{H}, \mathsf{c} \quad \text{(E-CONST)}$$

$$\mathcal{H}, \mathcal{E}, (\mathsf{lambda}\ (\mathsf{x}_1 \ldots \mathsf{x}_n)\ \mathsf{e}) \rightsquigarrow \mathcal{H}, \langle \mathcal{E}, \mathsf{x}_1 \ldots \mathsf{x}_n, \mathsf{e} \rangle \quad \text{(E-LAMBDA)}$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{E}, \mathsf{s} \rightsquigarrow \mathcal{H}, \mathsf{v}_c \\ i = 2 \quad \text{if } \mathsf{v}_c = \#\mathsf{f}, \quad 1 \text{ otherwise} \\ \mathcal{H}, \mathcal{E}, \mathsf{e}_i \rightsquigarrow \mathcal{H}_1, \mathsf{v}\end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{if}\ \mathsf{s}\ \mathsf{e}_1\ \mathsf{e}_2) \rightsquigarrow \mathcal{H}_1, \mathsf{v}} \quad \text{(E-IF)} \qquad \frac{\begin{array}{c}\mathcal{H}, \mathcal{E}, \mathsf{s}_i \rightsquigarrow \mathcal{H}, \mathsf{v}_i \qquad \forall i \in [0 \ldots n] \\ \mathcal{H}_1, \mathcal{E}_1, \mathsf{e} = \delta(\mathcal{H}, \mathcal{E}, \mathsf{v}_0, \mathsf{v}_1 \ldots \mathsf{v}_n) \\ \mathcal{H}_1, \mathcal{E}_1, \mathsf{e} \rightsquigarrow \mathcal{H}_2, \mathsf{v}\end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{s}_0\ \mathsf{s}_1 \ldots \mathsf{s}_n) \rightsquigarrow \mathcal{H}_2, \mathsf{v}} \quad \text{(E-APPLY)}$$

$$\frac{\begin{array}{c}\ell \text{ fresh} \\ \mathcal{H}, \mathcal{E}, \mathsf{e}_\mathsf{x} \rightsquigarrow \mathcal{H}_1, \mathsf{v}_\mathsf{x} \\ \mathcal{E}_2 = \mathcal{E}[\mathsf{x} \to \ell] \qquad \mathcal{H}_2 = \mathcal{H}_1[\ell \to \mathsf{v}_\mathsf{x}] \\ \mathcal{H}_2, \mathcal{E}_2, \mathsf{e} \rightsquigarrow \mathcal{H}_3, \mathsf{v}\end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{let}\ ([\mathsf{x}\ \mathsf{e}_\mathsf{x}])\ \mathsf{e}) \rightsquigarrow \mathcal{H}_3, \mathsf{v}} \quad \text{(E-LET)} \qquad \frac{\begin{array}{c}\mathcal{H}, \mathcal{E}, \mathsf{p} \rightsquigarrow \mathcal{H}, \mathsf{v}_\mathsf{p} \\ \text{If } \mathsf{v}_\mathsf{p} = \#\mathsf{f}, \text{raise a err-blame(p) error.} \\ \mathcal{H}, \mathcal{E}, \mathsf{e} \rightsquigarrow \mathcal{H}_1, \mathsf{v}\end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{check}\ \mathsf{p}\ \mathsf{e}) \rightsquigarrow \mathcal{H}_1, \mathsf{v}} \quad \text{(E-CHECK)}$$

$$\frac{\begin{array}{c}\ell_1, \ldots, \ell_n \text{ fresh} \\ \mathcal{E}_1 = \mathcal{E}[\mathsf{x}_1 \to \ell_1, \ldots, \mathsf{x}_n \to \ell_n] \\ \mathcal{H}, \mathcal{E}_1, \mathsf{e}_i \rightsquigarrow \mathcal{H}, \mathsf{v}_i \qquad \forall i \in [1 \ldots n] \\ \mathcal{H}_1 = \mathcal{H}[\ell_1 \to \mathsf{v}_1, \ldots, \ell_n \to \mathsf{v}_n] \\ \mathcal{H}_1, \mathcal{E}_1, \mathsf{e} \rightsquigarrow \mathcal{H}_2, \mathsf{v}\end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{letrec}\ ([\mathsf{x}_1\ \mathsf{e}_1] \ldots [\mathsf{x}_n\ \mathsf{e}_n])\ \mathsf{e}) \rightsquigarrow \mathcal{H}_2, \mathsf{v}} \quad \text{(E-LETREC)}$$

$$\frac{\mathcal{H}, \mathcal{E}, \mathsf{s} \rightsquigarrow \mathcal{H}, \mathsf{v} \qquad \mathcal{H}_1 = \mathcal{H}[\mathcal{E}[\mathsf{x}] \to \mathsf{v}]}{\mathcal{H}, \mathcal{E}, (\mathsf{set!}\ \mathsf{x}\ \mathsf{s}) \rightsquigarrow \mathcal{H}_1, \mathsf{void}} \quad \text{(E-SET)}$$

$$\begin{array}{rcll}
\delta(\mathcal{H}, \mathcal{E}, \mathsf{o}_\#, \mathsf{v}_1, \ldots, \mathsf{v}_m) & = & \mathsf{err\text{-}not\text{-}int}(\mathsf{v}_i) & \text{if } \exists i : \neg\mathsf{number?}\ \mathsf{v}_i \\
\delta(\mathcal{H}, \mathcal{E}, \mathsf{o}_\#, \mathsf{v}_1, \ldots, \mathsf{v}_m) & = & \mathcal{H}, \mathcal{E}, \mathsf{o}_\#(\mathsf{v}_1, \ldots, \mathsf{v}_m) & \text{otherwise} \\
\delta(\mathcal{H}, \mathcal{E}, \langle \mathcal{E}_\mathsf{f}, \mathsf{x}_1 \ldots \mathsf{x}_n, \mathsf{e} \rangle, \mathsf{v}_1, \ldots, \mathsf{v}_m) & = & \mathsf{err\text{-}args\text{-}}\lambda(\lambda \mathsf{x}_1 \ldots \mathsf{x}_n.\mathsf{e}) & \text{if } m \neq n \\
\delta(\mathcal{H}, \mathcal{E}, \langle \mathcal{E}_\mathsf{f}, \mathsf{x}_1 \ldots \mathsf{x}_n, \mathsf{e} \rangle, \mathsf{v}_1, \ldots, \mathsf{v}_m) & = & \mathcal{H}[\ell_1 \to \mathsf{v}_1, \ldots, \ell_n \to \mathsf{v}_n], \mathcal{E}_\mathsf{f}[\mathsf{x}_1 \to \ell_1, \ldots, \mathsf{x}_n \to \ell_n], \mathsf{e} & \\
\delta(\mathcal{H}, \mathcal{E}, \mathsf{v}, \ldots) & = & \mathsf{err\text{-}not\text{-}}\lambda(\mathsf{v}) & \text{if } \neg\mathsf{function?}(\mathsf{v})
\end{array}$$

Figure 3.2: Semantics for Scheme$_\beta$ with support for recursion and mutable variables

```
(letrec ([x (fac z)]
         [fac (lambda (n) (if (= n 1) 1 (* n (fac (- n 1)))))]
         [z 4]
         [sum (lambda (x y) (if (= x 0) y (sum (- x 1) (+ y 1))))])
  (sum x z))
```

(a) Before separation

```
(let ([z 4])
  (letrec ([sum (lambda (x y) (if (= x 0) y (sum (- x 1) (+ y 1))))])
    (letrec ([fac (lambda (n) (if (= n 1) 1 (* n (fac (- n 1)))))])
      (let ([x (fac z)])
        (sum x z)))))
```

(b) After separation

Figure 3.3: Enforcing invariants: Splitting a letrec into nested let and letrec expressions

$$
\begin{aligned}
[\![\texttt{(begin)}]\!] &= \texttt{(void)} \\
[\![\texttt{(begin e)}]\!] &= \texttt{e} \\
[\![\texttt{(begin e1 e2 ...)}]\!] &= \texttt{(let ([ignore e1])} \; [\![\texttt{(begin e2 ...)}]\!]\texttt{)}
\end{aligned}
$$

Figure 3.4: Translation of **begin** to **let**

may not occur free in e2 or later expressions. The bodies of functions, let and letrec-expressions are implicitly wrapped in a begin expression.

Semantics The new semantics, shown in figure 3.2, add an extra indirection between variables and values. Rather than having variables directly contain values (and substituting value for variables when evaluating let expressions), variables now point to memory locations in a heap. Assignment expressions designated by the set! keyword can update the value in the heap at the location indicated by the variable. Evaluation of lambda expressions also becomes more complex, as free variables are no longer substituted into their body; Instead, the evaluation of lambda expressions now need to capture the environment in which they are defined, which may differ from the environment in which they are called. Evaluating a lambda expression returns a *closure*, which is represented as a triple $\langle \mathcal{E}, \mathsf{x}_1 \ldots \mathsf{x}_n, \mathsf{e} \rangle$, where $\mathcal{E}$ is the captured environment, $\mathsf{x}_1 \ldots \mathsf{x}_n$ are the names of its formal parameters, and e is the body. A function with no parameters (also called a thunk) is represented as $\langle \mathcal{E}, (), \mathsf{e} \rangle$.

Evaluation rules in the new semantics are of the form $\mathcal{H}, \mathcal{E}, \mathsf{e} \rightsquigarrow \mathcal{H}', \mathsf{v}$: given a starting heap $\mathcal{H}$ and environment $\mathcal{E}$, the expression e evaluates to v, returning a potentially modified heap $\mathcal{H}'$. We use the syntax $\mathcal{E}[\mathsf{x}]$ and $\mathcal{H}[\ell]$ to denote lookups in environments

and heaps, respectively. $\mathcal{E}[x \to \ell]$ or $\mathcal{H}[\ell \to v]$ denote a new environment or heap with an updated binding, respectively for x and ℓ.
The new evaluation rules are as follows:

- E-CONST: evaluating a constant simply returns the constant.
- E-VAR: to evaluate a variable, its memory location is first looked up in the environment, which is then used to look up its value in the heap.
- E-LAMBDA: a lambda expression captures the environment in which it is defined.
- E-IF: the correct subexpression is evaluated depending on s being #f or not.
- E-APPLY: after evaluating the function v_f and arguments $v_1 \ldots v_n$, the δ function returns an expression, along with an environment and heap to evaluate it in.
- E-LET: after evaluating the expression e_x, a new memory location ℓ is initialized with the value v_x. The variable x is bound to this memory location, and the body is evaluated in this new heap and environment.
- E-CHECK: the check expression raises an error if its precondition evaluates to #f. All pending evaluation steps are aborted and the evaluation returns an err-blame(p) error.
- E-LETREC: every expression e_i is evaluated in the environment $\mathcal{E}_1$, which associates every variable in the binding group with a memory location. Note that attempting to read from or write to any of the bound variables during evaluation of these expressions will result in a stuck evaluation. Because of the restriction that letrec expressions only bind functions, this will never happen. After all functions have been evaluated to closures over the new environment $\mathcal{E}_1$, the heap is extended to associate the function names with these closures and finally the body is evaluated in the extended environment and heap.
- E-SET: finally, evaluating a set! expression looks up a variable's memory location in the environment and updates the heap at that location with the new value. Evaluating set! expressions always results in a void return value.

In this semantics we allow a rule to evaluate subexpressions directly, where the previously defined semantics (presented in figure 2.3) would rely on evaluation contexts and substitution. If any of these subexpressions raises an error — or in the case of E-VAR, the variable is not bound in the environment — evaluation immediately stops with the raised error.

The extended syntax and semantics are sufficient to describe and evaluate programs that use recursion or mutate variables. In the next section we discuss how recursion can be integrated into the three steps of the blame prediction transformation.

3.2 Recursion

The blame prediction transformation as introduced in the previous chapter does not support recursion, as the combination of recursion and the type system presented there gives rise to infinite types. In order to explain what infinite types are and how they might arise from recursive functions, consider the recursive function in listing 3.1, which counts down from `n` and returns the symbol `done` when the counter reaches zero:

```
1 (define (count n)
2   (if (= n 0)
3     'done
4     (count (- n 1))))
```

Listing 3.1: Recursive function which leads to an infinite type

Inferring the type of this function yields the following type:

$$\tau_{\alpha_{count}} = \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{symbol} \vee (\tau_{\alpha_{count}}\ \mathsf{int})]$$

This is an *infinite* type, as $\tau_{\alpha_{count}}$ contains itself. We can attempt to expand the function application inside the body of the type function and reduce it again:

$$\begin{aligned}
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{symbol} \vee (\Pi_{(\alpha_n)}.[(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{symbol} \vee (\tau_{\alpha_{count}}\ \mathsf{int})]]\ \mathsf{int})] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{symbol} \vee [(\mathsf{int} \mathbin{?=} \mathsf{int}) \cdot [\mathsf{symbol} \vee (\tau_{\alpha_{count}}\ \mathsf{int})]]] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{symbol} \vee [\mathsf{symbol} \vee (\tau_{\alpha_{count}}\ \mathsf{int})]] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{symbol} \vee (\tau_{\alpha_{count}}\ \mathsf{int})]
\end{aligned}$$

which produces the original type again. Clearly this process can be repeated an infinite number of times, hence the name *infinite type*.

In the previous chapter we used a type system for the basis of check inference, where conditional types such as the int? test in the above type result in check expressions in the final program. When functions with infinite types such as $\tau_{\alpha_{count}}$ are used, the type will be expanded an infinite number of times and the type inferencer will diverge. Instead of working with infinite types, we define a Solve function in section 3.2.2 that reduces infinite types to finite types, which do not contain self-references. The Solve function is based on well-known fixpoint techniques from abstract interpretation [Cousot and Cousot, 1977b].

In the rest of this chapter we develop support for recursion in the blame prediction transformation. The development is structured as follows: first, section 3.2.1 shows the check introduction and mobility rules for recursion. The check inference rules create infinite types, which must be reduced to finite types in order to be used in the check introduction stage. Section 3.2.2 defines the Solve function which performs this reduction. Some recursion patterns produce ever-growing types, which cause the Solve function

to diverge. We show these patterns and introduce a widening operation to remove these patterns in section 3.2.3. After that, we present a proof that the Solve function always terminates in section 3.2.4. Finally, section 3.2.5 shows worked-out examples of check inference on common recursion patterns.

3.2.1 Check Inference for Recursion

Inferring types for a letrec expression is different from a let expression for two reasons. First, due to the transformation described in section 3.1, all expressions in the letrec binding group must be function definitions. This means that evaluation of the values in the binding group can never result in an error. Secondly, the functions in the binding group can refer to both themselves and other functions in the group. The check inference and mobility rules for letrec expressions are shown in figure 3.5:

$$\frac{\begin{array}{c}\alpha_1, \ldots, \alpha_n \text{ fresh} \qquad \Gamma, x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash e_i : \overline{\tau_i} \qquad \forall i \in 1 \ldots n \\ \tau_1 \ldots \tau_n = \mathsf{Solve}(\overline{\tau_1} \ldots \overline{\tau_n};\ \alpha_1 \ldots \alpha_n) \\ \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau\end{array}}{\Gamma \vdash (\mathsf{letrec}\ ([x_1\ e_1] \ldots [x_n\ e_n])\ e) : \tau} \quad (\textsc{T-letrec})$$

$$\frac{\begin{array}{c}e_i \to e_i' \uparrow \#t \qquad \forall i \in 1 \ldots n \\ e \to e' \uparrow p \qquad p' = \mathsf{mask}(x_1 \ldots x_n, p)\end{array}}{(\mathsf{letrec}\ ([x_1\ e_1] \ldots [x_n\ e_n])\ e) \to (\mathsf{letrec}\ ([x_1\ e_1'] \ldots [x_n\ e_n'])\ (\mathsf{check}\ p\ e')) \uparrow p'} \quad (\textsc{F-letrec})$$

Figure 3.5: Check inference and mobility for letrec

To infer checks for a letrec expression, the various expressions in the binding part of the letrec are check-inferred separately, with the variables $x_1, \ldots, x_n$ bound to fresh type variables $\alpha_1, \ldots, \alpha_n$. This allows the functions to refer to themselves and other functions in the same binding group. The resulting types $\overline{\tau_1}, \ldots, \overline{\tau_n}$ are all type functions which contain *unexpanded* applications of the type variables $\alpha_1, \ldots, \alpha_n$. As stated earlier, the Solve function (section 3.2.2) is responsible for reducing these types to finite types $\tau_1, \ldots, \tau_n$. As the finite types no longer contain recursive type function applications, they can be used in the rest of the check inference process. Finally, the type of the body is inferred with the new types bound to the variables $x_1, \ldots, x_n$. The type of the body is returned as the type inferred for the entire letrec expression.

Next to the check inference rule, figure 3.5 also defines a check mobility rule. This rule is functionally identical to check mobility for let expressions, with two differences. Since all the expressions in the binding parts are function expressions, evaluating these expressions does not contribute any preconditions. Any preconditions propagated by the body are masked against all of the bound variables, to ensure they do not escape the letrec expression.

```
(letrec ([sum (lambda (x y)
                (if (= x 0)
                  y
                  (sum (- x 1) (+ y 1))))])
  ...)
```

Listing 3.2: Definition of the sum function

Having described the necessary rules, we will now demonstrate how check inference for letrec expressions works by applying it to the sum function from figure 3.3. For reference, a letrec expression defining the sum function is replicated in listing 3.2.

First, a fresh type variable α_{sum} is generated and bound to the sum variable; check inference is then performed on the function body. This yields the following type:

$$\overline{\tau_{sum}} = \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot (\alpha_{sum}\ \mathsf{int}\ \mathsf{int})]$$

To verify that this type is infinite, we can substitute $\overline{\tau_{sum}}$ for α_{sum} and reduce the type function application:

$$\begin{aligned}
&\Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot \boxed{(\overline{\tau_{sum}}\ \mathsf{int}\ \mathsf{int})}\ ] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot \boxed{(\mathsf{int} \mathrel{?=} \mathsf{int}) \cdot [\mathsf{int} \vee (\mathsf{int} \mathrel{?=} \mathsf{int}) \cdot (\alpha_{sum}\ \mathsf{int}\ \mathsf{int})]}\ ]
\end{aligned}$$

We can now simplify the conditional types:

$$= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot [\mathsf{int} \vee (\alpha_{sum}\ \mathsf{int}\ \mathsf{int})]]$$

If we substitute $\overline{\tau_{sum}}$ for α_{sum} again, we get the following:

$$\begin{aligned}
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot [\mathsf{int} \vee \boxed{(\overline{\tau_{sum}}\ \mathsf{int}\ \mathsf{int})}\ ]] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot [\mathsf{int} \vee \\
&\qquad \hookrightarrow \boxed{(\mathsf{int} \mathrel{?=} \mathsf{int}) \cdot [\mathsf{int} \vee (\mathsf{int} \mathrel{?=} \mathsf{int}) \cdot [\mathsf{int} \vee (\alpha_{sum}\ \mathsf{int}\ \mathsf{int})]]}\ ]] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot [\boxed{\mathsf{int} \vee \mathsf{int} \vee (\alpha_{sum}\ \mathsf{int}\ \mathsf{int})}\ ]] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot [\mathsf{int} \vee (\alpha_{sum}\ \mathsf{int}\ \mathsf{int})]]
\end{aligned}$$

which is identical to the first expansion. Therefore, $\overline{\tau_{sum}}$ is an infinite type.

In order to reduce the recursive type function to a finite type function, we must apply the Solve function. We will show the intermediate steps in section 3.2.2, but the end product is a finite type τ_{sum}:

$$\tau_{sum} = \mathsf{Solve}(\overline{\tau_{sum}}) = \Pi_{(\alpha_x,\alpha_y)}.[(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot \mathsf{int}]]$$

Looking back at the definition of sum, we see that the function first performs a = test on x, followed by either returning y or performing a recursive call with (+ x 1) and

`(- y 1)` as arguments. The type we have inferred and then made finite therefore correctly describes the actual behavior of the function.

An equivalent function in a conventional type system such as the one of Haskell would receive type $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$. Unlike Haskell, however, our function can also be invoked as `(sum 0 "hi")`, which does not perform a type test on the second argument.

We can now infer the body of the letrec with the `sum` variable bound to this type. It is only applied once, to two variables of type int. The type of this expression (and the entire letrec expression) is thus:

$$(\tau_{sum}\ \mathsf{int}\ \mathsf{int}) = \mathsf{int} \vee (\mathsf{int}\ ?{=}\ \mathsf{int}) \cdot (\mathsf{int}\ ?{=}\ \mathsf{int}) \cdot \mathsf{int} = \mathsf{int}$$

and check inference is complete.
In the next section, we describe how the Solve function reduces (groups of) recursive type functions to finite type functions.

3.2.2 Solving types for recursive functions

As the previous section showed, naïve expansion of infinite types can get stuck in an infinite loop. In this section we define the Solve function and show how it reduces infinite types to finite types. Instead of expanding the infinite types, the Solve function is based on a well-known technique from abstract interpretation and program analysis called the "frontier method" [Cousot and Cousot, 1977b].

Definition 3. The Solve function receives a list of infinite types $\overline{\tau_1} \ldots \overline{\tau_n}$ and a list of corresponding type variables $\alpha_1 \ldots \alpha_n$. It calculates a list of finite types $\tau_1 \ldots \tau_n$ using the following three steps:

1. Solve starts by parameterizing the infinite type functions over the type variables $\alpha_1 \ldots \alpha_n$ bound by the letrec expression. The resulting types are called *functionals* F^i. The functional for every infinite type is thus constructed as follows:

$$\begin{cases} \mathrm{F}^1 = \Pi_{(f_1 \ldots f_n)}.\overline{\tau_{f_1}}[f_1 \ldots f_n / \alpha_1 \ldots \alpha_n] \\ \mathrm{F}^2 = \Pi_{(f_1 \ldots f_n)}.\overline{\tau_{f_2}}[f_1 \ldots f_n / \alpha_1 \ldots \alpha_n] \\ \qquad\qquad \vdots \\ \mathrm{F}^n = \Pi_{(f_1 \ldots f_n)}.\overline{\tau_{f_n}}[f_1 \ldots f_n / \alpha_1 \ldots \alpha_n] \end{cases}$$

   The original infinite type functions $\overline{\tau_{f_i}}$ can be reconstructed from the functionals by applying the functionals to all type variables $\alpha_1 \ldots \alpha_n$ bound by the binding group:

$$\overline{\tau_{f_i}} = (\mathrm{F}^i\ \alpha_1\ \ldots\ \alpha_n)$$

2. Next, the Solve function constructs better approximations of the desired finite types by applying the functionals to earlier approximations. We write F_j^i for the j^{th} approximation of the finite type for the i^{th} function. The first approximation F_0^i for every finite type is $\bot$, the type which only contains diverging computations. Formally, these approximations form a (possibly infinite) ascending chain $\mathrm{F}_0^i \sqsubseteq \mathrm{F}_1^i \sqsubseteq \mathrm{F}_2^i \sqsubseteq \cdots \sqsubseteq \mathrm{F}_j^i \sqsubseteq \cdots$ according to a partial order $\sqsubseteq$ defined over types. According to the fixpoint theorem this series converges to a fixpoint, which is the desired finite type. We will discuss this further in section 3.2.4, as part of the discussion on termination.

   The progression of the approximations can be seen below from left to right, with the general case shown at the far right.

$$\begin{cases} \mathrm{F}_0^1 = \bot & \mathrm{F}_1^1 = (\mathrm{F}^1\ \mathrm{F}_0^1\ \cdots\ \mathrm{F}_0^n) & \dots & \mathrm{F}_{j+1}^1 = (\mathrm{F}^1\ \mathrm{F}_j^1\ \cdots\ \mathrm{F}_j^n) \\ \mathrm{F}_0^2 = \bot & \mathrm{F}_1^2 = (\mathrm{F}^2\ \mathrm{F}_0^1\ \cdots\ \mathrm{F}_0^n) & \dots & \mathrm{F}_{j+1}^2 = (\mathrm{F}^2\ \mathrm{F}_j^1\ \cdots\ \mathrm{F}_j^n) \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{F}_0^n = \bot & \mathrm{F}_1^n = (\mathrm{F}^n\ \mathrm{F}_0^1\ \cdots\ \mathrm{F}_0^n) & \dots & \mathrm{F}_{j+1}^n = (\mathrm{F}^n\ \mathrm{F}_j^1\ \cdots\ \mathrm{F}_j^n) \end{cases}$$

3. Finally, after some k iterations this process stabilizes, such that:

$$\begin{cases} \mathrm{F}_{k+1}^1 = (\mathrm{F}^1\ \mathrm{F}_k^1\ \cdots\ \mathrm{F}_k^n) = \mathrm{F}_k^1 \\ \mathrm{F}_{k+1}^2 = (\mathrm{F}^2\ \mathrm{F}_k^1\ \cdots\ \mathrm{F}_k^n) = \mathrm{F}_k^2 \\ \vdots \\ \mathrm{F}_{k+1}^n = (\mathrm{F}^n\ \mathrm{F}_k^1\ \cdots\ \mathrm{F}_k^n) = \mathrm{F}_k^n \end{cases}$$

   When this point is reached, Solve returns a list of finite types $\mathrm{F}_k^1 \dots \mathrm{F}_k^n$. We prove that stabilization (and thus termination) is guaranteed in section 3.2.4.

After every iteration in the second step, the types must be reduced as much as possible. Especially applications of type functions must be reduced, according to the rules we defined earlier in section 2.2 (figure 2.5). Additionally, the reductions in Figure 3.6 must be applied. The intuition behind these reductions is as follows:

1. Rules (3.1) through (3.5) define $\bot$ as the absorbing element with respect to the given kinds of types. For example, rule (3.2) states that a type test with $\bot$ can be reduced to $\bot$ directly. $\bot$ here is a computation which diverges instead of producing a type. Rule (3.4) short-circuits the reduction of applied type functions when the body of the function is $\bot$, and (3.5) reduces applications of $\bot$ to arguments to just $\bot$.

2. Type unions, however, can discard a $\bot$ branch (rule (3.6)), which corresponds to gathering all choices from the type union which do not diverge.

3. Rule (3.7) states that the union of identical types is the same as just taking one branch.

4. Rule (3.8) reduces a type by moving identical type tests out of a type union.

5. Finally, rule (3.9) enables us to remove conditional types which are "covered" by identical conditional types higher up in the type. The construct $\mathrm{T}\langle\tau\rangle$ used here is a "type context" with a type τ in the focus, similar to the evaluation context $\mathrm{E}\langle\mathsf{e}\rangle$ used in chapter 2. In this case, the reduction rule searches for a conditional type which contains a type context where another conditional type (with the same type test) is in focus.

$$(\gamma \mathbin{?=} \bot) \cdot \tau = \bot \tag{3.1}$$

$$(\gamma \mathbin{?=} \tau) \cdot \bot = \bot \tag{3.2}$$

$$(\Pi_{(\alpha)}.[\tau]\ \bot) = \bot \tag{3.3}$$

$$(\Pi_{(\alpha)}.[\bot]\ \tau) = \bot \tag{3.4}$$

$$(\bot\ \tau_1\ \ldots\ \tau_n) = \bot \tag{3.5}$$

$$\tau \vee \bot = \bot \vee \tau = \tau \tag{3.6}$$

$$\tau_1 \vee \tau_2 = \tau_1 \qquad \text{if } \tau_1 = \tau_2 \tag{3.7}$$

$$((\gamma_1 \mathbin{?=} \alpha_1) \cdot \tau_1) \vee ((\gamma_2 \mathbin{?=} \alpha_2) \cdot \tau_2) = (\gamma_1 \mathbin{?=} \alpha_1) \cdot (\tau_1 \vee \tau_2) \qquad \text{if } \gamma_1 = \gamma_2 \wedge \alpha_1 = \alpha_2 \tag{3.8}$$

$$(\gamma_1 \mathbin{?=} \alpha_1) \cdot \mathrm{T}\langle(\gamma_2 \mathbin{?=} \alpha_2) \cdot \tau\rangle = (\gamma_1 \mathbin{?=} \alpha_1) \cdot \mathrm{T}\langle\tau\rangle \qquad \text{if } \gamma_1 = \gamma_2 \wedge \alpha_1 = \alpha_2 \tag{3.9}$$

Figure 3.6: Reduction rules used during the reduction stage of Solve

In the rest of this section we show a concrete example of using Solve. We will use the following typographical conventions:

1. This and later examples will make a number of successive simplification steps; the parts under consideration will be highlighted using a darker background like this. In the first few examples the reduction rule applied is shown in [brackets];

2. In order to conserve space, parts of the type can be elided with dots (...);

3. To conserve even more space, subsequent identical type tests are merged. For example, $(\mathsf{int} \mathbin{?=} \alpha_x) \cdot (\mathsf{int} \mathbin{?=} \alpha_y) \cdot \tau$ is abbreviated as $(\mathsf{int} \mathbin{?=} \alpha_x, \alpha_y) \cdot \tau$;

4. Repeated applications of F to $\bot$ are named F_n, so:
$\mathrm{F}_1 = (\mathrm{F}\ \bot)$, $\mathrm{F}_2 = (\mathrm{F}\ (\mathrm{F}\ \bot))$, $\mathrm{F}_3 = (\mathrm{F}\ (\mathrm{F}\ (\mathrm{F}\ \bot)))$, ...

5. Finally, a type that does not fit on one line will be wrapped to the next line. Wrapped lines are indented and start with a gray arrow like so: ↪.

```
1 (define (fac n)
2   (if (< n 2)
3     1
4     (* n (fac (- n 1)))))
```

Listing 3.3: Factorial function

In order to illustrate the Solve function, we show how it can produce a finite type for the well-known factorial function. The factorial function is defined as shown in listing 3.3 and it gives rise to the following recursive type:

$$\alpha_{fac} = \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_x)}.[(\mathsf{int} \mathbin{?=} \alpha_x) \cdot \mathsf{int}]\ (\alpha_{fac}\ \alpha_n))]$$

We can rewrite this type as a functional, where α_{fac} is an argument:

$$\mathrm{F} = \Pi_{(\alpha_{fac})}.\Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_x)}.[(\mathsf{int} \mathbin{?=} \alpha_x) \cdot \mathsf{int}]\ (\alpha_{fac}\ \mathsf{int}))]$$

We apply F to $\bot$, which yields a first approximation F_1:

$$\begin{aligned}
\mathrm{F}_1 &= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_x)}.[(\mathsf{int} \mathbin{?=} \alpha_x) \cdot \mathsf{int}]\ (\bot\ \alpha_n))] && \text{[Rule (3.5)]} \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_x)}.[(\mathsf{int} \mathbin{?=} \alpha_x) \cdot \mathsf{int}]\ \bot)] && \text{[Rule (3.3)]} \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{int} \vee \bot] && \text{[Rule (3.6)]} \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot \mathsf{int}
\end{aligned}$$

Note that, because of the elimination rules for $\bot$, the recursive path has effectively been removed from the type. We can now apply F to F_1:

$$\begin{aligned}
\mathrm{F}_2 &= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_x)}.[(\mathsf{int} \mathbin{?=} \alpha_x) \cdot \mathsf{int}]\ (\mathrm{F}_1\ \alpha_n))] && \text{[Expand } \mathrm{F}_1\text{]} \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_x)}.[(\mathsf{int} \mathbin{?=} \alpha_x) \cdot \mathsf{int}]\ (\Pi_{(\alpha_m)}.[(\mathsf{int} \mathbin{?=} \alpha_m) \cdot \mathsf{int}]\ \alpha_n))] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_x)}.[(\mathsf{int} \mathbin{?=} \alpha_x) \cdot \mathsf{int}]\ (\mathsf{int} \mathbin{?=} \alpha_n) \cdot \mathsf{int})] && \text{[Rule (3.9)]} \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_x)}.[(\mathsf{int} \mathbin{?=} \alpha_x) \cdot \mathsf{int}]\ \mathsf{int})] && \text{[Apply function]} \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{int} \vee (\mathsf{int} \mathbin{?=} \mathsf{int}) \cdot \mathsf{int}] && \text{[Trivial test]} \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{int} \vee \mathsf{int}] && \text{[Rule (3.7)]} \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot \mathsf{int}
\end{aligned}$$

We have now reached a fixpoint, as $\mathrm{F}_2 = (\mathrm{F}\ \mathrm{F}_1) = \mathrm{F}_1$.
The final type of the factorial function is therefore

$$\alpha_{fac} = \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot \mathsf{int}$$

which matches the conventional type $\mathsf{int} \to \mathsf{int}$.

Finally, we can apply Solve to find the finite type of the sum function as well. Its functional is:

$$F = \Pi_{(\alpha_{sum})}.\Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot (\alpha_{sum}\ \mathsf{int}\ \mathsf{int})]$$

For sum, approximation stabilizes after three iterations:

$$\begin{aligned}
F_1 &= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot (\ F_0\ \mathsf{int}\ \mathsf{int})] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot \ (\bot\ \mathsf{int}\ \mathsf{int})\ ] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee \ (\mathsf{int} \mathrel{?=} \alpha_y) \cdot \bot\ ] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\ \alpha_y \vee \bot\ ] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot \alpha_y \\
F_2 &= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot (\ F_1\ \mathsf{int}\ \mathsf{int})] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot \ (\mathsf{int} \mathrel{?=} \mathsf{int}) \cdot \mathsf{int}\ ] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot \mathsf{int}] \\
F_3 &= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot (\ F_2\ \mathsf{int}\ \mathsf{int})] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot (\mathsf{int} \mathrel{?=} \mathsf{int}) \cdot [\mathsf{int} \vee \ (\mathsf{int} \mathrel{?=} \mathsf{int}) \cdot \mathsf{int}\ ]] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot (\mathsf{int} \mathrel{?=} \mathsf{int}) \cdot [\ \mathsf{int} \vee \mathsf{int}\ ]] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot \ (\mathsf{int} \mathrel{?=} \mathsf{int}) \cdot \mathsf{int}\ ] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot \mathsf{int}]
\end{aligned}$$

The final type is

$$\alpha_{sum} = \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\alpha_y \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot \mathsf{int}]$$

as stated in the previous section.
In the next section we discuss some recursion patterns where the Solve function is not able to terminate, and we introduce a *widening* operation which ensures termination.

3.2.3 Non-terminating recursion patterns

In this section we discuss three recursion patterns, for which every iteration of the Solve function yields a different, larger type. For types which contain these patterns, iteration will thus never stabilize and Solve will never terminate.

In order to still guarantee termination, we will show how type terms can be transformed such that these problematic patterns no longer appear. These transformations are lossy: they will remove problematic parts of the type in exchange for termination. They are an instance of a technique from abstract interpretation called "widening" [Cousot and Cousot, 1977a, 1992a]. In the original formulation, widening was introduced as a means of making the frontier method (which is the basis for our Solve function) converge faster, with a separate "narrowing" step afterwards.

Pattern 1: Accumulation of type function applications

In chapter 2 we showed how the type inference rules for let expressions give rise to immediately-applied type functions. The resulting type terms can often be simplified immediately, except when the expression being bound to the value is a type variable application. Consider the function `rep` in listing 3.4, which applies the function `f n` times to the starting value `x`. For example, the result of `(rep double 2 3)` is 12.

```
1 (define (rep f n x)
2   (if (= n 0)
3     x
4     (let ([tmp1 (f x)])
5       (rep f (- n 1) tmp1)))
```

Listing 3.4: Example of accumulation of type function applications

Its type and functional are as follows:

$$\alpha_{rep} = \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_x \vee (\Pi_{(\alpha_y)}.[(\alpha_{rep}\ \alpha_f\ \mathsf{int}\ \alpha_y)]\ (\alpha_f\ \alpha_x))]$$
$$\mathrm{F} = \Pi_{(\alpha_{rep})}.\Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_x \vee (\Pi_{(\alpha_y)}.[(\alpha_{rep}\ \alpha_f\ \mathsf{int}\ \alpha_y)]\ (\alpha_f\ \alpha_x))]$$

Applying fixpoint iteration yields the following:

$$\begin{aligned}
\mathrm{F}_1 &= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_x \vee (\Pi_{(\alpha_y)}.[\ (\bot\ \alpha_f\ \mathsf{int}\ \alpha_y)\ ]\ (\alpha_f\ \alpha_x))] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_x \vee\ (\Pi_{(\alpha_y)}.[\bot]\ (\alpha_f\ \alpha_x))\ ] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\ \alpha_x \vee \bot\ ] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot \alpha_x \\
\mathrm{F}_2 &= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_x \vee (\Pi_{(\alpha_y)}.[\ (\mathrm{F}_1\ \alpha_f\ \mathsf{int}\ \alpha_y)\ ]\ (\alpha_f\ \alpha_x))] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_x \vee (\Pi_{(\alpha_y)}.[\ (\mathsf{int} \mathbin{?=} \mathsf{int}) \cdot \alpha_y\ ]\ (\alpha_f\ \alpha_x))] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_x \vee\ (\Pi_{(\alpha_y)}.[\alpha_y]\ (\alpha_f\ \alpha_x))\ ] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_x \vee (\alpha_f\ \alpha_x)] \\
\mathrm{F}_3 &= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_x \vee (\Pi_{(\alpha_y)}.[\ (\mathrm{F}_2\ \alpha_f\ \mathsf{int}\ \alpha_y)\ ]\ (\alpha_f\ \alpha_x))] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_x \vee (\Pi_{(\alpha_y)}.[\ (\mathsf{int} \mathbin{?=} \mathsf{int}) \cdot [\alpha_y \vee (\alpha_f\ \alpha_y)]\ ]\ (\alpha_f\ \alpha_x))] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_x \vee (\Pi_{(\alpha_y)}.[\alpha_y \vee (\alpha_f\ \alpha_y)]\ (\alpha_f\ \alpha_x))]
\end{aligned}$$

A pattern emerges: every expansion adds an extra application of the type function α_f, which cannot be reduced further until `f` is known, which will not happen until `rep` is applied to its arguments.

As the size of the type never stops growing, a fixpoint will never be reached and the iteration will never stop. The widening technique we will apply approximates the type by replacing the type function α_f by a type function which returns the "any type" $\star$, of which nothing is known. We introduced this type in chapter 2, as the type of expressions "of which nothing is known". Inserting a type function which returns $\star$ ensures termination, as the type function being applied is now known and can be reduced immediately. There are two direct results from applying this widening: first, the loss of any record of that function being invoked. Any conditional types which would have been introduced by the higher order function being passed in will never be exposed when the recursive function is called. Second, the introduction of the "any type" $\star$ will reduce the precision of the final finite type. For example, the set of values denoted by the type $\alpha_x \vee \star$ is the same as that by $\star$, so a type test will always be generated.

Applying widening to the `rep` function above results in the following modified functional F^*, where α_f has been replaced with $\Pi_{(\alpha_z)}.[\star]$:

$$\begin{aligned}
F^* &= \Pi_{(\alpha_{rep})}.\Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_x \vee (\Pi_{(\alpha_y)}.[(\alpha_{rep}\ \alpha_f\ \mathsf{int}\ \alpha_y)]\ \ (\Pi_{(\alpha_z)}.[\star]\ \alpha_x)\ )] \\
&= \Pi_{(\alpha_{rep})}.\Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_x \vee\ (\Pi_{(\alpha_y)}.[(\alpha_{rep}\ \alpha_f\ \mathsf{int}\ \alpha_y)]\ \star)\ ] \\
&= \Pi_{(\alpha_{rep})}.\Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_x \vee (\alpha_{rep}\ \alpha_f\ \mathsf{int}\ \star)]
\end{aligned}$$

If we now apply fixpoint iteration, we get a finite type after three iterations.

$$\begin{aligned}
F_1^* &= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_x \vee\ (\bot\ \alpha_f\ \mathsf{int}\ \star)\ ] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\ \alpha_x \vee \bot\ ] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot \alpha_x \\
F_2^* &= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_x \vee\ (F_1^*\ \alpha_f\ \mathsf{int}\ \star)\ ] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_x \vee\ (\mathsf{int}\ ?{=}\ \mathsf{int}) \cdot \star\ ] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_x \vee \star] \\
F_3^* &= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_x \vee\ (F_2^*\ \alpha_f\ \mathsf{int}\ \star)\ ] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_x \vee\ (\mathsf{int}\ ?{=}\ \mathsf{int}) \cdot [\star \vee \star]\ ] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_x \vee [\ \star \vee \star\ ]] \\
&= \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_x \vee \star]
\end{aligned}$$

The final type for the `rep` function is thus:

$$\alpha_{rep} = \Pi_{(\alpha_f,\alpha_n,\alpha_x)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_x \vee \star]$$

which only tests whether its `n` argument is a number and produces $\alpha_x \vee \star$, which is the same as $\star$.

Let us briefly consider two alternative widening strategies:

- The first alternative strategy is to keep the first type function application in each path of the type and replace subsequent type function applications with $\star$. This is similar to the simplification rule for nested conditional types. On the upside, this widening strategy exposes the type function application to the caller, which means that its conditional type tests can be exposed through calls in the body of the letrec. As a downside, it is unclear whether one type function application suffices, e. g. for a function which performs polymorphic recursion.
- The second alternative is to replace all type function applications of an unknown type variable α_f with a new type variable α_r which represents the result type of such an application. However, this technique ignores dependencies on the argument types (e. g. $\Pi_{(\alpha_a)}.\alpha_a$), any type unions returned by the function (e. g. $\Pi_{(\alpha_a)}.[\alpha_a \vee \mathsf{int}]$) and any type tests made along the way (e. g. $\Pi_{(\alpha_a)}.[(\mathsf{int} \mathbin{?=} \alpha_a) \cdot \mathsf{int}]$). In effect, this type variable α_r would yield no more precise types than a type function returning $\star$ would.

Pattern 2: Construction of true infinite types

The second pattern involves the construction of infinite types by functionals which add layers of type functions. Consider the functions `wrap`, `wrap2` and `unwrap` in listing 3.5. `wrap` and `wrap2` both wrap the function `f` in `n` layers of dummy functions, except one is tail-recursive and the other is not. The `unwrap` function undoes this wrapping by passing `#t` to all added functions. For all non-negative `n`, the identity `(unwrap n (wrap n f) x)` $\equiv$ `(f x)` holds, similarly for `unwrap` and `wrap2`.

```
1  (define (wrap n f)
2    (if (= n 0)
3      f
4      (wrap (- n 1) (lambda (x) f))))
5
6  (define (wrap2 n f)
7    (if (= n 0)
8      f
9      (lambda (x) (wrap2 (- n 1) f))))
10
11 (define (unwrap n f x)
12   (if (= n 0)
13     (f x)
14     (unwrap (- n 1) (f #t) x)))
```

Listing 3.5: Construction of infinite types through recursion

We first construct the type and functional for `wrap`:

$$\alpha_{wrap} = \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_f \vee (\alpha_{wrap}\ \mathsf{int}\ \Pi_{(\alpha_x)}.\alpha_f)]$$

$$\mathrm{F} = \Pi_{(\alpha_{wrap})}.\Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee (\alpha_{wrap}\ \mathsf{int}\ \Pi_{(\alpha_x)}.\alpha_f)]$$

Let us now attempt to calculate its fixpoint:

$$\begin{aligned}
\mathrm{F}_1 &= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee (\bot\ \mathsf{int}\ \Pi_{(\alpha_x)}.\alpha_f)] \\
&= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee \bot] \\
&= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot \alpha_f \\
\mathrm{F}_2 &= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee (\mathrm{F}_1\ \mathsf{int}\ \Pi_{(\alpha_x)}.[\alpha_f])] \\
&= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee \Pi_{(\alpha_x)}.[\alpha_f]] \\
\mathrm{F}_3 &= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee (\mathrm{F}_2\ \mathsf{int}\ \Pi_{(\alpha_x)}.[\alpha_f])] \\
&= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee [\Pi_{(\alpha_x)}.[\alpha_f] \vee \Pi_{(\alpha_y)}.[\Pi_{(\alpha_x)}.[\alpha_f]]]] \\
\mathrm{F}_4 &= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee (\mathrm{F}_3\ \mathsf{int}\ \Pi_{(\alpha_x)}.[\alpha_f])] \\
&= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee [\Pi_{(\alpha_x)}.[\alpha_f] \vee [\Pi_{(\alpha_y)}.[\Pi_{(\alpha_x)}.[\alpha_f]] \vee \Pi_{(\alpha_z)}.[\Pi_{(\alpha_y)}.[\Pi_{(\alpha_x)}.[\alpha_f]]]]]]
\end{aligned}$$

With every iteration, the argument to `wrap` is added to the type, and the argument to the recursive type invocation is wrapped in another type function. The approximations for `wrap` will thus always grow larger and a fixpoint will never be reached.
For `wrap2`, we get the following type and functional:

$$\begin{aligned}
\alpha_{wrap2} &= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee \Pi_{(\alpha_x)}.(\alpha_{wrap2}\ \mathsf{int}\ \alpha_f)] \\
\mathrm{G} &= \Pi_{(\alpha_{wrap2})}.\Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee \Pi_{(\alpha_x)}.(\alpha_{wrap2}\ \mathsf{int}\ \alpha_f)]
\end{aligned}$$

We attempt fixpoint iteration again:

$$\begin{aligned}
\mathrm{G}_1 &= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee \Pi_{(\alpha_x)}.(\bot\ \mathsf{int}\ \alpha_f)] \\
&= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee \Pi_{(\alpha_x)}.\bot] \\
\mathrm{G}_2 &= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee \Pi_{(\alpha_x)}.(\mathrm{G}_1\ \mathsf{int}\ \alpha_f)] \\
&= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee \Pi_{(\alpha_x)}.[\alpha_f \vee \Pi_{(\alpha_y)}.\bot]] \\
\mathrm{G}_3 &= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee \Pi_{(\alpha_x)}.(\mathrm{G}_2\ \mathsf{int}\ \alpha_f)] \\
&= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee \Pi_{(\alpha_x)}.[\alpha_f \vee \Pi_{(\alpha_y)}.[\alpha_f \vee \Pi_{(\alpha_z)}.\bot]]] \\
\mathrm{G}_4 &= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee \Pi_{(\alpha_x)}.(\mathrm{G}_3\ \mathsf{int}\ \alpha_f)] \\
&= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\alpha_f \vee \Pi_{(\alpha_x)}.[\alpha_f \vee \Pi_{(\alpha_y)}.[\alpha_f \vee \Pi_{(\alpha_z)}.[\alpha_f \vee \Pi_{(\alpha_w)}.\bot]]]]
\end{aligned}$$

Similarly to `wrap`, `wrap2` produces a type that keeps on growing.

In statically typed programming languages, this kind of functions cannot be processed. For example, attempting to enter a literal translation of the `wrap` function above

into the Glasgow Haskell Compiler prints the following error, with a nearly identical error for `wrap2`:

```
wrap.hs:4:31:
    Occurs check: cannot construct the infinite type: t1 ~ t -> t1
    Relevant bindings include
      x :: t (bound at wrap.hs:4:26)
      f :: t -> t1 (bound at wrap.hs:1:8)
      wrap :: a -> (t -> t1) -> t -> t1 (bound at wrap.hs:1:1)
    In the expression: f
    In the second argument of 'wrap', namely '(\ x -> f)'
```

Listing 3.6: Occurs check error in Haskell

The famous occurs check [Robinson, 1965] mentioned in listing 3.6 is a part of the constraint resolution stage of the Hindley-Milner type system [Hindley, 1969; Milner, 1978]. It prevents a type variable from unifying with a type that includes itself but is not just the same type variable. Attempting to do so would introduce an infinite type and thus a diverging substitution.

In some systems, such infinite types are avoided by introducing an explicit recursive binder. However, this is exactly the kind of type our Solve function attempts to reduce to a finite type! Instead, we propose the following widening strategy: replace all nested type functions which contain references to the bound type functions or their bound argument type variables with a type function that returns $\star$. Type functions which only contain ground types or references to external type variables (i. e. those bound outside the letrec expression) are left intact. By leaving out such "varying" type variables, the body of a type function can only reference ground types, its own argument type variables and type variables bound by the environment.

Let us apply this widening to F (call it F^*) and attempt to calculate the fixpoint again:

$$
\begin{aligned}
F^* &= \Pi_{(\alpha_{wrap})}.\Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_f \vee (\alpha_{wrap}\ \mathsf{int}\ [\Pi_{(\alpha_x)}.[\star]])] \\
F_1^* &= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_f \vee \ (\bot\ \mathsf{int}\ [\Pi_{(\alpha_x)}.[\star]])\ ] \\
&= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\ \alpha_f \vee \bot\ ] \\
&= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot \alpha_f \\
F_2^* &= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_f \vee \ (F_1^*\ \mathsf{int}\ [\Pi_{(\alpha_x)}.[\star]])\ ] \\
&= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_f \vee [\Pi_{(\alpha_x)}.[\star]]] \\
F_3^* &= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_f \vee \ (F_2^*\ \mathsf{int}\ [\Pi_{(\alpha_x)}.[\star]])\ ] \\
&= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_f \vee [\ [\Pi_{(\alpha_x)}.[\star]] \vee [\Pi_{(\alpha_x)}.[\star]]\ ]] \\
&= \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\alpha_f \vee \Pi_{(\alpha_x)}.[\star]]
\end{aligned}
$$

After three steps, fixpoint iteration terminates, with resulting type

$$\alpha_{wrap} = \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} ?= \alpha_n) \cdot [\alpha_f \vee [\Pi_{(\alpha_x)}.[\star]]]$$

We now calculate the fixpoint for G after widening, which we call G^*:

$$G^* = \Pi_{(\alpha_{wrap2})}.\Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} ?= \alpha_n) \cdot [\alpha_f \vee [\Pi_{(\alpha_x)}.[\star]]]$$
$$G^*_1 = \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} ?= \alpha_n) \cdot [\alpha_f \vee [\Pi_{(\alpha_x)}.[\star]]]$$

Since G^* no longer contains any calls to α_{wrap2}, fixpoint iteration immediately terminates. The type for `wrap2` is then

$$\alpha_{wrap2} = \Pi_{(\alpha_n,\alpha_f)}.(\mathsf{int} ?= \alpha_n) \cdot [\alpha_f \vee [\Pi_{(\alpha_x)}.[\star]]]$$

Pattern 3: Fractal types

The final recursion pattern that will cause non-termination is a pattern we call "fractal types": functions which contain un-applied references to themselves or other functions in the letrec binding group. The name stems from the self-referential nature of fractals, as every approximation of the fractal type contains another copy of itself. For example, listing 3.7 is a function that calls itself recursively until its input is odd or smaller than two. However, the programmer wrote `f` instead of `n` by mistake in the last line, so the type of `f` effectively contains itself.

```
1 (define (f n)
2   (if (< n 2)
3     n
4     (if (even? n)
5       (f (div n 2))
6       f)))
```

Listing 3.7: Example of a fractal type

We can see this in the type and functional of `f`:

$$\alpha_f = \Pi_{(\alpha_n)}.(\mathsf{int} ?= \alpha_n) \cdot [\mathsf{int} \vee (\alpha_f\ \mathsf{int}) \vee \alpha_f]$$
$$F = \Pi_{(\alpha_f)}.\Pi_{(\alpha_n)}.(\mathsf{int} ?= \alpha_n) \cdot [\mathsf{int} \vee (\alpha_f\ \mathsf{int}) \vee \alpha_f]$$

Fixpoint iteration yields the following steps:

$$\begin{aligned}
F_1 &= \Pi_{(\alpha_n)}.(\mathsf{int} ?= \alpha_n) \cdot [\mathsf{int} \vee (\bot\ \mathsf{int}) \vee \bot] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} ?= \alpha_n) \cdot [\mathsf{int} \vee [\bot \vee \bot]] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} ?= \alpha_n) \cdot [\mathsf{int} \vee \bot] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} ?= \alpha_n) \cdot \mathsf{int} \\
F_2 &= \Pi_{(\alpha_n)}.(\mathsf{int} ?= \alpha_n) \cdot [\mathsf{int} \vee (F_1\ \mathsf{int}) \vee F_1]
\end{aligned}$$

$$
\begin{aligned}
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\mathsf{int} \vee \mathsf{int} \vee \boxed{\mathrm{F}_1}] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\mathsf{int} \vee [\Pi_{(\alpha_m)}.(\mathsf{int} \mathrel{?=} \alpha_m) \cdot \mathsf{int}]] \\
\mathrm{F}_3 &= \Pi_{(\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\mathsf{int} \vee \boxed{(\mathrm{F}_2\ \mathsf{int})} \vee \mathrm{F}_2] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\mathsf{int} \vee \mathsf{int} \vee [\Pi_{(\alpha_m)}.(\mathsf{int} \mathrel{?=} \alpha_m) \cdot \mathsf{int}] \vee \boxed{\mathrm{F}_2}] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\mathsf{int} \vee [\Pi_{(\alpha_m)}.(\mathsf{int} \mathrel{?=} \alpha_m) \cdot \mathsf{int}] \vee \\
&\qquad \hookrightarrow [\Pi_{(\alpha_o)}.(\mathsf{int} \mathrel{?=} \alpha_o) \cdot [\mathsf{int} \vee [\Pi_{(\alpha_p)}.(\mathsf{int} \mathrel{?=} \alpha_p) \cdot \mathsf{int}]]]]
\end{aligned}
$$

We can see that with every iteration the size of the approximation increases without end, leading to non-termination. The widening strategy here is the same as the strategy in the previous pattern, namely to replace the "fractal" reference to the function α_f by a type function of equal arity which returns $\star$.

The widened functional and fixpoint iteration steps now are as follows:

$$
\begin{aligned}
\mathrm{F}^* &= \Pi_{(\alpha_f)}.\Pi_{(\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\mathsf{int} \vee (\alpha_f\ \mathsf{int}) \vee [\Pi_{(\alpha_x)}.\star]] \\
\mathrm{F}^*_1 &= \Pi_{(\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\mathsf{int} \vee \boxed{(\bot\ \mathsf{int})} \vee [\Pi_{(\alpha_x)}.\star]] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\mathsf{int} \vee [\boxed{\bot \vee [\Pi_{(\alpha_x)}.\star]}]] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\mathsf{int} \vee [\Pi_{(\alpha_x)}.\star]] \\
\mathrm{F}^*_2 &= \Pi_{(\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\mathsf{int} \vee \boxed{(\mathrm{F}^*_1\ \mathsf{int})} \vee [\Pi_{(\alpha_x)}.\star]] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\mathsf{int} \vee [\boxed{\mathsf{int} \vee [\Pi_{(\alpha_x)}.\star]}] \vee [\Pi_{(\alpha_x)}.\star]] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\mathsf{int} \vee [\Pi_{(\alpha_x)}.\star]]
\end{aligned}
$$

Fixpoint iteration on the widened functional now stops after two steps and yields the type

$$\alpha_f = \Pi_{(\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_n) \cdot [\mathsf{int} \vee [\Pi_{(\alpha_x)}.\star]]$$

Final widening strategy

We have discussed three problematic cases which prohibit termination and presented widening strategies to ensure termination. We now combine these strategies into a final widening strategy.

Definition 4 (Widening strategy)**.** Given a set of functionals $\mathrm{F}^1, \mathrm{F}^2, \ldots$, the following widening operations must be applied to ensure termination:

1. Type function applications where the function is a type variable not bound by the letrec binding group must be replaced by $\star$;
2. If any nested type functions exist which contain references to type variables *not* bound by the environment, replace their bodies with $\star$;

3. References to functionals $\alpha_{f_1}, \alpha_{f_2}, \ldots$ outside of type function applications must be replaced with type functions of equal arity, with $\star$ as body.

We will prove that widening guarantees termination in the next section.

Summary
To recap the contribution of this section: in the beginning of this chapter we remarked that applying type inference to recursive functions yields types which reference themselves. In order to infer types for applications of these functions the infinite types must be expanded first, which results in an infinite loop. In this section we described how the Solve function reduces these infinite types to finite types, which do not contain self references. Solve uses the well-known "frontier method" [Cousot and Cousot, 1977b] to reduce a group of infinite type functions into a group of finite type functions. With these finite type functions, the type inference part of the blame prediction transformation is able to produce types for all expressions in the program in a finite amount of time.

However, because our type system features conditional types, there are three recursion patterns which induce infinitely large types and therefore cause the Solve function to diverge. For each pattern we showed how to apply appropriate widening strategies — first introduced by Cousot and Cousot [1977b, 1992a] — to sacrifice precision in order to guarantee termination.

3.2.4 Termination properties of Solve

As stated in the previous section, the Solve function only produces a finite type if the repeated applications of the functional reach a fixpoint. Every step of the Solve function expands its argument by substituting function applications and subsequently reduces it again through simplification. However, not all parts of types can be simplified, or at least not immediately.

In this section we will prove that the Solve function always terminates for a set of functional equations, given that the widening operations from the previous section have been applied.

Before we begin, we define a normal form and partial ordering for types.

Definition 5 (Normal form for types)**.** The general shape of a type in normal form is shown below: a type union where every branch consists of zero or more type tests followed by either an atomic type term, a type function, an immediately-applied type

function, or a type function application of a type variable. A type can reference type variables bound by the environment, named α_1 through α_n.

$$\tau = \bigvee (\gamma_1 \mathbin{?=} \alpha_1) \cdot \ldots \cdot (\gamma_n \mathbin{?=} \alpha_n) \cdot \begin{cases} \mathsf{atom} & \mathsf{atom} \in \{\bot, \star, \gamma, \alpha_1, \ldots, \alpha_n\} \\ \Pi_{(\alpha_{n+1}, \ldots, \alpha_{n+m})}.[\tau] & \\ (\Pi_{(\alpha)}.[\tau_b]\ (\tau\ \tau_1\ \ldots\ \tau_n)) & \tau_b \text{ is in normal form} \\ (\tau\ \tau_1\ \ldots\ \tau_n) & \end{cases}$$

Additionally, the following restrictions apply:

1. Satisfied conditional types of the form $(\gamma \mathbin{?=} \gamma)$ are not present.
2. None of the reduction rules from figure 3.6 are applicable.
3. The arguments to a type function application are reduced as far as possible, per figure 2.5. For example, for arbitrary types τ_a, τ_b, τ_c, and τ_d:

   $$(\Pi_{(\alpha_2, \alpha_3, \alpha_4)}.[(\alpha_1\ \alpha_2\ \alpha_3\ \alpha_4)]\ [(\mathsf{int} \mathbin{?=} \alpha_m) \cdot \tau_a]\ [\tau_b \vee \tau_c]\ \tau_d)$$

   is not in normal form, but it can be reduced to the following, which is:

   $$(\mathsf{int} \mathbin{?=} \alpha_m) \cdot [(\Pi_{(\alpha_2, \alpha_3, \alpha_4)}.[(\alpha_1\ \alpha_2\ \alpha_3\ \alpha_4)]\ \tau_a\ \tau_b\ \tau_d) \vee (\Pi_{(\alpha_2, \alpha_3, \alpha_4)}.[(\alpha_1\ \alpha_2\ \alpha_3\ \alpha_4)]\ \tau_a\ \tau_c\ \tau_d)]$$

4. Type tests from conditional types must be grouped by target variable, then sorted by cause label. There can only be one conditional type test per type–variable combination, as figure 3.6 eq. (3.9) compresses multiple conditional types on the same variable for the same type into one conditional type. Note that a type in normal form may contain several groupings of conditional types, for example inside a type function.

Definition 6 (Partial ordering for types)**.** We define a partial order $\tau_1 \sqsubseteq \tau_2$ (read: "τ_2 is equal to or more defined than τ_1") over the domain of types in normal form, shown in figure 3.7. This partial order has seven properties:

1. $\bot$ makes up the bottom end of the partial order;
2. Directly above $\bot$ in the hierarchy is $\star$, with $\bot \sqsubset \star \sqsubseteq \tau$ for any non-$\bot$ τ. Recall that $\star$ types are introduced by mutable variables or as a result of applying the widening operator. Therefore, $\star$ is not at the top of the hierarchy, as it only represents a single type for values of which nothing is known, and which cannot be refined later on; item Types are equal to themselves;
3. A type union is more defined than its constituent parts. Type unions must be compared point-wise.

4. type functions with the same amount of arguments are compared with respect to their bodies;
5. If two types start with the same conditional type, their bodies are compared. A type with more conditional types (i. e. a longer one) is considered more defined;
6. Immediately-applied type functions are also compared with respect to their bodies.

$$\frac{\tau \neq \bot}{\bot \sqsubset \star \sqsubseteq \tau} \qquad \tau \sqsubseteq \tau \qquad \tau_1 \sqsubseteq \tau_1 \vee \tau_2 \qquad \tau_2 \sqsubseteq \tau_1 \vee \tau_2 \qquad \frac{\alpha_1 \sqsubseteq \alpha_3 \qquad \alpha_2 \sqsubseteq \alpha_4}{\alpha_1 \vee \alpha_2 \sqsubseteq \alpha_3 \vee \alpha_4}$$

$$\frac{\tau_1 \sqsubseteq \tau_2}{\Pi_{(\alpha_1 \ldots \alpha_n)}.\tau_1 \sqsubseteq \Pi_{(\alpha_1 \ldots \alpha_n)}.\tau_2} \qquad \frac{\tau_1 \sqsubseteq \tau_2}{(\tau_t \mathbin{?=} \alpha) \cdot \tau_1 \sqsubseteq (\tau_t \mathbin{?=} \alpha) \cdot \tau_2} \qquad \frac{\tau_1 \neq (\tau_t \mathbin{?=} \alpha) \cdot \tau_1}{\tau_1 \sqsubseteq (\tau_t \mathbin{?=} \alpha) \cdot \tau_2}$$

$$\frac{\tau_1 \sqsubseteq \tau_2}{(\Pi_{(\alpha_z)}.[\tau_1]\ (\alpha_f\ \alpha_{x_1}\ \ldots\ \alpha_{x_n})) \sqsubseteq (\Pi_{(\alpha_z)}.[\tau_2]\ (\alpha_f\ \alpha_{x_1}\ \ldots\ \alpha_{x_n}))}$$

Figure 3.7: Definition of the partial order $\sqsubseteq$ over types in normal form

Having defined the normal form for types, we turn to the criteria necessary for termination of the Solve function. We focus on the case where there is only a single function in the letrec binding group, but the results generalize to multiple functions.

The fixpoint iteration algorithm attempts to find the fixpoint of the functional of this function by constructing the Kleene sequence $\{\mathrm{F}_i \mid i \geq 0 \wedge \mathrm{F}_i \sqsubseteq \mathrm{F}_{i+1}\}$. The algorithm terminates when a $j \geq 0$ is found such that $\mathrm{F}_j = \mathrm{F}_{j+1}$. We will show that there are a finite number of variations of types in normal form, given a fixed functional F and a restricted set of type variables $\alpha_1, \ldots, \alpha_n$ bound by the environment.

We define names for the following sets of type variables:

Ext the set of "external" type variables $\alpha_1, \ldots, \alpha_n$, i. e. type variables bound by the environment;

Arg the set of type variables $\alpha_{n+1}, \ldots, \alpha_{n+m}$ bound by the function;

In addition, we define Gnd as the set of ground types (e. g. int, string, ...).

Definition 7 (Type universe)**.** For a given step j of the fixpoint iteration algorithm, F_j is of the form:

$$\mathrm{F}_j = \Pi_{(\alpha_{n+1},\ldots,\alpha_{n+m})}.\bigvee \tau_g \qquad \text{where } \tau_g \in \mathsf{Gen}(\mathsf{Ext} \cup \{\alpha_{n+1}, \ldots, \alpha_{n+m}\})$$

The helper function $\mathsf{Gen}(\mathsf{Vars})$ constructs the set of all types in normal form which can reference the type variables in Vars. Every type has the following shape:

$$\tau_g = \underbrace{(\gamma_1 \mathbin{?=} \alpha_1) \cdot \dots \cdot (\gamma_n \mathbin{?=} \alpha_n)\cdot}_{①} \begin{cases} \mathsf{atom} & ② \ \mathsf{atom} \in \{\bot, \star\} \cup \mathsf{Gnd} \cup \mathsf{Ext} \cup \mathsf{Arg} \\ \Pi_{(\alpha_{n+m+1},\dots,\alpha_{n+m+o})}.[\tau] & ③ \end{cases}$$

Note that the bottom two cases of definition 5 are not present in the approximations F_j. This omission is explained as follows: the widening applied to the functional prevents type function applications of anything but the type variables bound by letrec, which are filled in by successive steps of the Solve function. We therefore only need to be concerned with atoms and nested type functions.

Theorem 1. The set of all types in the set $\mathsf{G} = \mathsf{Gen}(\{\alpha_1, \dots, \alpha_n\} \cup \{\alpha_{n+1}, \dots, \alpha_{n+m}\})$ is finite.

Proof. We prove this by calculating the number of possible variations for every numbered component above. By construction, $|\mathsf{G}| = \left|①\right| \times (\left|②\right| + \left|③\right|)$. For every component, we calculate the number of variations:

① The set of ground types Gnd is finite, and each of the $n + m$ type variables can be tested for at most once for every ground type. The number of variations is thus the number of subsets of Gnd(the powerset $\mathcal{P}$) times the number of variables which can be tested:

$$\left|①\right| = |\mathcal{P}(\mathsf{Gnd})| \times |\mathsf{Ext} \cup \mathsf{Arg}| = 2^{|\mathsf{Gnd}|} \times (n + m)$$

② The number of atom type terms is limited:

$$\left|②\right| = |\{\bot, \star\}| + |\mathsf{Gnd}| + |\mathsf{Ext}| + |\mathsf{Arg}|$$

③ Any type functions present in a step of the Solve function may only contain references to type variables in the Ext set. Moreover, they must already be part of the functional (i. e. there is a finite amount of such functions), as the widening we applied ensures new type functions cannot be created. Successive steps may insert these type functions elsewhere in the type, but their depth never increases.

There are a finite number of variations for every component, therefore the set

$$\mathsf{G} = \mathsf{Gen}(\{\alpha_1, \dots, \alpha_n\} \cup \{\alpha_{n+1}, \dots, \alpha_{n+m}\})$$

is finite. □

Finally, we can prove our original goal:

Theorem 2 (Guaranteed termination for fixed type universes)**.** Fixpoint iteration by the Solve function always terminates, provided that widening has been applied to the functionals.

Proof. As we showed in definition 7, every step of the fixpoint iteration algorithm contains a type union of one or more elements of the finite set $\mathsf{G} = \mathsf{Gen}(\{\alpha_1, \ldots, \alpha_n\} \cup \{\alpha_{n+1}, \ldots, \alpha_{n+m}\})$ for that step. Recall that every step in the fixpoint iteration process forms an ascending chain $\mathrm{F}_0 \sqsubseteq \mathrm{F}_1 \sqsubseteq \cdots \sqsubseteq \mathrm{F}_j \sqsubseteq \mathrm{F}_{j+1} \sqsubseteq \cdots$. In terms of the partial order we have defined, this means either adding a branch to the type union at the top level, or replacing a branch with a better-defined branch. As the set G is finite, every extended branch must eventually reach a supremum. Likewise, only a finite number of distinct branches can be added. Therefore, fixpoint iteration *must* terminate after a finite number of steps j, for which $\mathrm{F}_{j+1} = \mathrm{F}_j$. □

Conclusion With this section proving the termination of the Solve function, our treatment of recursive functions in the blame prediction transformation is complete. We started by extending Scheme$_\beta$ to support the letrec keyword and defining how type inference could be extended to letrec expressions. The inferred types were *infinite*, however, which needed to be converted to finite functions by the Solve function. After defining this function, we discussed some recursion patterns which induced ever-growing types and thus an infinite loop in the Solve function. With the widening operation defined in section 3.2.3 and the proof contained in this section, we are guaranteed that Solve — and thus check inference for recursive functions — always terminates.
In the next section we show a number of common recursive functions and apply the Solve function to calculate finite types for them.

3.2.5 Worked out examples

In the absence of a formal reference work on recursion schemes, we have opted to present some recursion patterns commonly found in programs. For each example we show a function definition, the inferred recursive type, and the functional F derived from this type. This is followed by the fixpoint iteration steps necessary to reach a fixpoint.

3.2.5.1 Fibonacci

The first function we examine is the Fibonacci function, which invokes itself twice in its own body, see listing 3.8. The type and functional that correspond with this code also contain two invocations:

$$\alpha_{fib} = \Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_{x1},\alpha_{x2})}.[(\mathsf{int} \mathbin{?=} \alpha_{x1}) \cdot (\mathsf{int} \mathbin{?=} \alpha_{x2}) \cdot \mathsf{int}]\ (\alpha_{fib}\ \mathsf{int})\ (\alpha_{fib}\ \mathsf{int}))]$$
$$\mathrm{F} = \Pi_{(\alpha_{fib})}.\Pi_{(\alpha_n)}.(\mathsf{int} \mathbin{?=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_{x1},\alpha_{x2})}.[(\mathsf{int} \mathbin{?=} \alpha_{x1}) \cdot (\mathsf{int} \mathbin{?=} \alpha_{x2}) \cdot \mathsf{int}]\ (\alpha_{fib}\ \mathsf{int})\ (\alpha_{fib}\ \mathsf{int}))]$$

```
1 (define (fib n)
2   (if (< n 1)
3     0
4     (+ (fib (- n 1)) (fib (- n 2)))))
```

Listing 3.8: Fibonacci function

We approximate until a fixpoint is reached:

$$
\begin{aligned}
F_1 &= \Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_{x1},\alpha_{x2})}.[(\mathsf{int} ?{=} \alpha_{x1}) \cdot (\mathsf{int} ?{=} \alpha_{x2}) \cdot \mathsf{int}]\ (\bot\ \mathsf{int})\ (\bot\ \mathsf{int}))] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_{x1},\alpha_{x2})}.[(\mathsf{int} ?{=} \alpha_{x1}) \cdot (\mathsf{int} ?{=} \alpha_{x2}) \cdot \mathsf{int}] \bot\ (\bot\ \mathsf{int}))] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_{x1},\alpha_{x2})}.[(\mathsf{int} ?{=} \alpha_{x1}) \cdot (\mathsf{int} ?{=} \alpha_{x2}) \cdot \mathsf{int}] \bot \bot)] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot [\mathsf{int} \vee \bot] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot \mathsf{int} \\
F_2 &= \Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_{x1},\alpha_{x2})}.[(\mathsf{int} ?{=} \alpha_{x1}) \cdot (\mathsf{int} ?{=} \alpha_{x2}) \cdot \mathsf{int}]\ (F_1\ \mathsf{int})\ (F_1\ \mathsf{int}))] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_{x1},\alpha_{x2})}.[\cdots]\ (\Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot \mathsf{int}\ \mathsf{int})\ (F_1\ \mathsf{int}))] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_{x1},\alpha_{x2})}.[\cdots]\ (\mathsf{int} ?{=} \mathsf{int}) \cdot \mathsf{int}\ (F_1\ \mathsf{int}))] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_{x1},\alpha_{x2})}.[(\mathsf{int} ?{=} \alpha_{x1}) \cdot (\mathsf{int} ?{=} \alpha_{x2}) \cdot \mathsf{int}]\ \mathsf{int}\ (F_1\ \mathsf{int}))] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot [\mathsf{int} \vee (\Pi_{(\alpha_{x1},\alpha_{x2})}.[(\mathsf{int} ?{=} \alpha_{x1}) \cdot (\mathsf{int} ?{=} \alpha_{x2}) \cdot \mathsf{int}]\ \mathsf{int}\ \mathsf{int})] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot [\mathsf{int} \vee (\mathsf{int} ?{=} \mathsf{int}) \cdot (\mathsf{int} ?{=} \mathsf{int}) \cdot \mathsf{int}] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot [\mathsf{int} \vee \mathsf{int}] \\
&= \Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot \mathsf{int}
\end{aligned}
$$

After two steps, the fixpoint $F_2 = (F\ F_1) = F_1$ is reached.
The final type of the Fibonacci function is

$$\alpha_{fib} = \Pi_{(\alpha_n)}.(\mathsf{int} ?{=} \alpha_n) \cdot \mathsf{int}$$

3.2.5.2 Ackermann

The Ackermann function is known for growing very quickly given small inputs; its definition is shown in listing 3.9. The main reason for this phenomenon is the recursive invocation on line 6 below, where the output of a recursive call is used as parameter for another recursive call. We can infer the following type and functional:

$$
\begin{aligned}
\alpha_{ack} &= \Pi_{(\alpha_m,\alpha_n)}.(\mathsf{int} ?{=} \alpha_m, \alpha_n) \cdot [\mathsf{int} \vee (\alpha_{ack}\ \mathsf{int}\ \mathsf{int}) \vee (\Pi_{(\alpha_d)}.[(\alpha_{ack}\ \mathsf{int}\ \alpha_d)]\ (\alpha_{ack}\ \mathsf{int}\ \mathsf{int}))] \\
F &= \Pi_{(\alpha_{ack})}.\Pi_{(\alpha_m,\alpha_n)}.(\mathsf{int} ?{=} \alpha_m, \alpha_n) \cdot [\mathsf{int} \vee (\alpha_{ack}\ \mathsf{int}\ \mathsf{int}) \vee (\Pi_{(\alpha_d)}.[(\alpha_{ack}\ \mathsf{int}\ \alpha_d)]\ (\alpha_{ack}\ \mathsf{int}\ \mathsf{int}))]
\end{aligned}
$$

```
1 (define (ack m n)
2   (if (= m 0)
3     (+ n 1)
4     (if (and (> m 0) (= n 0))
5       (ack (- m 1) 1)
6       (ack (- m 1) (ack m (- n 1))))))
```

Listing 3.9: Ackermann function

```
1 (define (tak x y z)
2   (if (< y x)
3     (tak (tak (- x 1) y z)
4          (tak (- y 1) z x)
5          (tak (- z 1) x y))
6     z))
```

Listing 3.10: Tak function

We approximate until a fixpoint is reached:

$$
\begin{aligned}
F_1 &= \Pi_{(\alpha_m,\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_m, \alpha_n) \cdot [\mathsf{int} \vee \ (\bot\ \mathsf{int}\ \mathsf{int})\ \vee (\Pi_{(\alpha_d)}.[\ (\bot\ \mathsf{int}\ \alpha_d)\ ]\ \ (\bot\ \mathsf{int}\ \mathsf{int})\ )] \\
&= \Pi_{(\alpha_m,\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_m, \alpha_n) \cdot [\mathsf{int} \vee \bot \vee \ (\Pi_{(\alpha_d)}.[\bot]\ \bot)\ ] \\
&= \Pi_{(\alpha_m,\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_m, \alpha_n) \cdot [\ \mathsf{int} \vee \bot \vee \bot\ ] \\
&= \Pi_{(\alpha_m,\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_m, \alpha_n) \cdot \mathsf{int} \\
F_2 &= \Pi_{(\alpha_m,\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_m, \alpha_n) \cdot [\mathsf{int} \vee \ (F_1\ \mathsf{int}\ \mathsf{int})\ \vee (\Pi_{(\alpha_d)}.[(F_1\ \mathsf{int}\ \alpha_d)]\ \ (F_1\ \mathsf{int}\ \mathsf{int})\ )] \\
&= \Pi_{(\alpha_m,\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_m, \alpha_n) \cdot [\mathsf{int} \vee \mathsf{int} \vee \ (\Pi_{(\alpha_d)}.[(F_1\ \mathsf{int}\ \alpha_d)]\ \mathsf{int})\ ] \\
&= \Pi_{(\alpha_m,\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_m, \alpha_n) \cdot [\mathsf{int} \vee \mathsf{int} \vee \ (F_1\ \mathsf{int}\ \mathsf{int})\ ] \\
&= \Pi_{(\alpha_m,\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_m, \alpha_n) \cdot [\ \mathsf{int} \vee \mathsf{int} \vee \mathsf{int}\ ] \\
&= \Pi_{(\alpha_m,\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_m, \alpha_n) \cdot \mathsf{int}
\end{aligned}
$$

After two steps, the fixpoint $F_2 = (F\ F_1) = F_1$ is reached.
The final type of the Ackermann function is therefore

$$\alpha_{ack} = \Pi_{(\alpha_m,\alpha_n)}.(\mathsf{int} \mathrel{?=} \alpha_m) \cdot (\mathsf{int} \mathrel{?=} \alpha_n) \cdot \mathsf{int}$$

3.2.5.3 Tak

The Tak function is another well-known recursive function, defined in listing 3.10. It features a recursive call that requires three other recursive calls. Its type and functional are as follows:

$$\alpha_{tak} = \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int} \mathrel{?=} \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int} \mathrel{?=} \alpha_z) \cdot$$

$$\hookrightarrow(\Pi_{(\alpha_d,\alpha_e,\alpha_f)}.[(\alpha_{tak}\ \alpha_d\ \alpha_e\ \alpha_f)]\ (\alpha_{tak}\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ (\alpha_{tak}\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ (\alpha_{tak}\ \mathsf{int}\ \mathsf{int}\ \mathsf{int}))]$$

$$\mathrm{F} = \Pi_{(\alpha_{tak})}.\Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot$$

$$\hookrightarrow(\Pi_{(\alpha_d,\alpha_e,\alpha_f)}.[(\alpha_{tak}\ \alpha_d\ \alpha_e\ \alpha_f)]\ (\alpha_{tak}\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ (\alpha_{tak}\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ (\alpha_{tak}\ \mathsf{int}\ \mathsf{int}\ \mathsf{int}))]$$

We approximate until a fixpoint is reached.

$$\begin{aligned}
\mathrm{F}_1 &= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot (\Pi_{(\alpha_d)}.[\cdots]\ (\bot\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ )] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot (\Pi_{(\alpha_d)}.[\cdots]\ \bot)\ ] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\ \alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot \bot\ ] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\ \alpha_z \vee \bot\ ] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot \alpha_z \\
\mathrm{F}_2 &= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot (\Pi_{(\alpha_d)}.[\cdots]\ (\mathrm{F}_1\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ )] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot (\Pi_{(\alpha_d,\alpha_e)}.[\cdots]\ \mathsf{int}\ (\mathrm{F}_1\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ )] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot \\
&\qquad \hookrightarrow(\Pi_{(\alpha_d,\alpha_e,\alpha_f)}.[(\mathrm{F}_1\ \alpha_d\ \alpha_e\ \alpha_f)]\ \mathsf{int}\ \mathsf{int}\ (\mathrm{F}_1\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ )] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot \\
&\qquad \hookrightarrow (\Pi_{(\alpha_d,\alpha_e,\alpha_f)}.[(\mathrm{F}_1\ \alpha_d\ \alpha_e\ \alpha_f)]\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ ] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot (\mathrm{F}_1\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ ] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot \mathsf{int}] \\
\mathrm{F}_3 &= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot \\
&\qquad \hookrightarrow(\Pi_{(\alpha_d,\alpha_e,\alpha_f)}.[(\mathrm{F}_2\ \alpha_d\ \alpha_e\ \alpha_f)]\ (\mathrm{F}_2\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ (\mathrm{F}_2\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ (\mathrm{F}_2\ \mathsf{int}\ \mathsf{int}\ \mathsf{int}))] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot \\
&\qquad \hookrightarrow(\Pi_{(\alpha_d,\alpha_e,\alpha_f)}.[(\mathrm{F}_2\ \alpha_d\ \alpha_e\ \alpha_f)]\ \mathsf{int}\ (\mathrm{F}_2\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ (\mathrm{F}_2\ \mathsf{int}\ \mathsf{int}\ \mathsf{int}))] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot \\
&\qquad \hookrightarrow(\Pi_{(\alpha_d,\alpha_e,\alpha_f)}.[(\mathrm{F}_2\ \alpha_d\ \alpha_e\ \alpha_f)]\ \mathsf{int}\ \mathsf{int}\ (\mathrm{F}_2\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ )] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot \\
&\qquad \hookrightarrow (\Pi_{(\alpha_d,\alpha_e,\alpha_f)}.[(\mathrm{F}_2\ \alpha_d\ \alpha_e\ \alpha_f)]\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ ] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot (\mathrm{F}_2\ \mathsf{int}\ \mathsf{int}\ \mathsf{int})\ ] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot [\ \mathsf{int} \vee \mathsf{int}\ ]] \\
&= \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int}\ ?{=}\ \alpha_x, \alpha_y) \cdot [\alpha_z \vee (\mathsf{int}\ ?{=}\ \alpha_z) \cdot \mathsf{int}]
\end{aligned}$$

After three steps, the fixpoint $F_3 = (F\ F_2) = F_2$ is reached.
The final type is

$$\alpha_{tak} = \Pi_{(\alpha_x,\alpha_y,\alpha_z)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot (\mathsf{int} \mathrel{?=} \alpha_y) \cdot [((\mathsf{int} \mathrel{?=} \alpha_z) \cdot \mathsf{int}) \vee \alpha_z]$$

In other type systems this function simply has type $\mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}$, but note that `z` is returned immediately if `(>= y x)`, so `(tak 1 1 "hello")` will produce the string `"hello"`. Therefore, the type produced by approximation is correct.

3.2.5.4 Infinite recursion

The examples above were well-behaved with respect to termination. However, programmers will sometimes accidentally write functions that do not terminate. For example, consider the following function:

```
1 (define (wrong x y)
2   (if (= x 0)
3     (wrong y x)
4     (wrong (- x 1) (+ y 1))))
```

Listing 3.11: Example of infinite recursion

Its type and functional are:

$$\alpha_{wrong} = \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [(\alpha_{wrong}\ \alpha_y\ \mathsf{int}) \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot (\alpha_{wrong}\ \mathsf{int}\ \mathsf{int})]$$
$$F = \Pi_{(\alpha_{wrong})}.\Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [(\alpha_{wrong}\ \alpha_y\ \mathsf{int}) \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot (\alpha_{wrong}\ \mathsf{int}\ \mathsf{int})]$$

Note that all paths through the function go through a recursive call, therefore the function can never end. We approximate until a fixpoint is reached:

$$\begin{aligned}
F_1 &= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\ (\bot\ \alpha_y\ \mathsf{int})\ \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot\ (\bot\ \mathsf{int}\ \mathsf{int})\ ] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\bot \vee\ (\mathsf{int} \mathrel{?=} \alpha_y) \cdot \bot\ ] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\ \bot \vee \bot\ ] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot \bot \\
F_2 &= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\ (F_1\ \alpha_y\ \mathsf{int})\ \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot (F_1\ \mathsf{int}\ \mathsf{int})] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [(\mathsf{int} \mathrel{?=} \alpha_y) \cdot \bot \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot\ (F_1\ \mathsf{int}\ \mathsf{int})\ ] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\ (\mathsf{int} \mathrel{?=} \alpha_y) \cdot \bot\ \vee (\mathsf{int} \mathrel{?=} \alpha_y) \cdot\ (\mathsf{int} \mathrel{?=} \mathsf{int}) \cdot \bot\ ] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\bot \vee\ (\mathsf{int} \mathrel{?=} \alpha_y) \cdot \bot\ ] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\ \bot \vee \bot\ ] \\
&= \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot \bot
\end{aligned}$$

After two steps, the fixpoint $F_2 = (F\ F_1) = F_1$ is reached.
The final type is

$$\alpha_{wrong} = \Pi_{(\alpha_x,\alpha_y)}.(\mathsf{int}\ ?{=}\ \alpha_x) \cdot \bot$$

Note that the $\bot$ remains in the type, which indicates that the function will never return. During the check insertion stage, expressions that come after a call to this function could be marked as dead code.

3.2.5.5 Cyclic recursion

We have remarked before that dynamically typed programming languages allow a function to have multiple return types. The `rotate` function below returns one of its three last arguments when its first becomes zero:

```
1 (define (rotate n a b c)
2   (if (= n 0)
3     a
4     (rotate (- n 1) c a b)))
```

Listing 3.12: Cyclic recursion

Its type and functional are as follows:

$$\alpha_{rot} = \Pi_{(\alpha_n,\alpha_a,\alpha_b,\alpha_c)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_a \vee (\alpha_{rot}\ \mathsf{int}\ \alpha_c\ \alpha_a\ \alpha_b)]$$
$$F = \Pi_{(\alpha_{rot})}.\Pi_{(\alpha_n,\alpha_a,\alpha_b,\alpha_c)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_a \vee (\alpha_{rot}\ \mathsf{int}\ \alpha_c\ \alpha_a\ \alpha_b)]$$

We approximate until a fixpoint is reached:

$$\begin{aligned}
F_1 &= \Pi_{(\alpha_n,\alpha_a,\alpha_b,\alpha_c)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_a \vee (\bot\ \mathsf{int}\ \alpha_c\ \alpha_a\ \alpha_b)] \\
&= \Pi_{(\alpha_n,\alpha_a,\alpha_b,\alpha_c)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_a \vee \bot] \\
&= \Pi_{(\alpha_n,\alpha_a,\alpha_b,\alpha_c)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot \alpha_a \\
(F_2\ \bot) &= \Pi_{(\alpha_n,\alpha_a,\alpha_b,\alpha_c)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_a \vee (F_1\ \mathsf{int}\ \alpha_c\ \alpha_a\ \alpha_b)] \\
&= \Pi_{(\alpha_n,\alpha_a,\alpha_b,\alpha_c)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_a \vee \alpha_c] \\
(F_3\ \bot) &= \Pi_{(\alpha_n,\alpha_a,\alpha_b,\alpha_c)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_a \vee (F_2\ \mathsf{int}\ \alpha_c\ \alpha_a\ \alpha_b)] \\
&= \Pi_{(\alpha_n,\alpha_a,\alpha_b,\alpha_c)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_a \vee \alpha_c \vee \alpha_b] \\
(F_4\ \bot) &= \Pi_{(\alpha_n,\alpha_a,\alpha_b,\alpha_c)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_a \vee (F_3\ \mathsf{int}\ \alpha_c\ \alpha_a\ \alpha_b)] \\
&= \Pi_{(\alpha_n,\alpha_a,\alpha_b,\alpha_c)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_a \vee \alpha_c \vee \alpha_b \vee \alpha_a] \\
&= \Pi_{(\alpha_n,\alpha_a,\alpha_b,\alpha_c)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_a \vee \alpha_c \vee \alpha_b]
\end{aligned}$$

After four iterations we have $F_4 = (F\ F_3) = F_3$, with a resulting type of

$$\mathsf{rot} = \Pi_{(\alpha_n,\alpha_a,\alpha_b,\alpha_c)}.(\mathsf{int}\ ?{=}\ \alpha_n) \cdot [\alpha_a \vee \alpha_c \vee \alpha_b]$$

3.2.5.6 Mutual Recursion

even and odd are a textbook example of two mutually recursive functions, they are shown in listing 3.13. The types of even and odd are:

$$\alpha_{even} = \Pi_{(\alpha)}.(\mathsf{int} \mathrel{?=} \alpha) \cdot [\mathsf{boolean} \vee (\alpha_{odd}\ \mathsf{int})]$$
$$\alpha_{odd} = \Pi_{(\alpha)}.(\mathsf{int} \mathrel{?=} \alpha) \cdot [\mathsf{boolean} \vee (\alpha_{even}\ \mathsf{int})]$$

The corresponding functionals E and O:

$$\mathrm{E} = \Pi_{(\alpha_{even},\alpha_{odd})}.\Pi_{(\alpha_x)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\mathsf{boolean} \vee (\alpha_{odd}\ \mathsf{int})]$$
$$\mathrm{O} = \Pi_{(\alpha_{even},\alpha_{odd})}.\Pi_{(\alpha_x)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\mathsf{boolean} \vee (\alpha_{even}\ \mathsf{int})]$$

```
1 (letrec ([even (lambda (n)
2                  (if (= n 0)
3                    #t
4                    (odd (- n 1))))]
5          [odd  (lambda (n)
6                  (if (= n 0)
7                    #f
8                    (even (- n 1))))])
9   (even 13))
```

Listing 3.13: Mutually recursive functions: even and odd

We first expand with $\bot$ for both E and O:

$$\begin{aligned}
\mathrm{E}_1 &= \Pi_{(\alpha_x)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\mathsf{boolean} \vee (\bot\ \mathsf{int})] \\
&= \Pi_{(\alpha_x)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\mathsf{boolean} \vee \bot] \\
&= \Pi_{(\alpha_x)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot \mathsf{boolean} \\
\mathrm{O}_1 &= \Pi_{(\alpha_x)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\mathsf{boolean} \vee (\bot\ \mathsf{int})] \\
&= \Pi_{(\alpha_x)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\mathsf{boolean} \vee \bot] \\
&= \Pi_{(\alpha_x)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot \mathsf{boolean} \\
\mathrm{E}_2 &= \Pi_{(\alpha_x)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\mathsf{boolean} \vee (\mathrm{O}_1\ \mathsf{int})] \\
&= \Pi_{(\alpha_x)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\mathsf{boolean} \vee \mathsf{boolean}] \\
&= \Pi_{(\alpha_x)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot \mathsf{boolean} \\
\mathrm{O}_2 &= \Pi_{(\alpha_x)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\mathsf{boolean} \vee (\mathrm{E}_1\ \mathsf{int})] \\
&= \Pi_{(\alpha_x)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot [\mathsf{boolean} \vee \mathsf{boolean}] \\
&= \Pi_{(\alpha_x)}.(\mathsf{int} \mathrel{?=} \alpha_x) \cdot \mathsf{boolean}
\end{aligned}$$

After two iterations, we have $O_1 = O_2$ and $E_1 = E_2$. The resulting types for even and odd are thus:

$$\alpha_{even} = \Pi_{(\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_x) \cdot \mathsf{boolean}$$
$$\alpha_{odd} = \Pi_{(\alpha_x)}.(\mathsf{int} \mathbin{?=} \alpha_x) \cdot \mathsf{boolean}$$

3.3 Conclusion

In chapter 2 we developed blame prediction for a subset of Scheme, reminiscent of the λ-calculus, which did not support for recursion or mutable variables. In this chapter we added support for recursion, such that blame prediction can be applied to programs which make use of these features.

Extended syntax and semantics First we extended the syntax and semantics of Scheme$_\beta$ in order to accommodate the new set! and letrec special forms. The simple substitution-based semantics in chapter 2 were replaced by semantics where variables reference memory locations on a heap, which contains the actual values. With these new semantics we are able to evaluate programs that define recursive functions or perform variable mutation.

Recursion In the remainder of this chapter we developed support for analysing and transforming recursive functions in the context of blame prediction. Recall that the first stage of the blame prediction transformation is check inference, which is based on type inference. We started by remarking that letrec expressions give rise to types which reference themselves. Upon encountering a function call expression involving such an *infinite type*, the type inferencer starts expanding the infinite type and goes into an infinite loop, hence the name. In order to perform blame prediction for expressions which make use of recursive functions, we require *finite types*, which do not contain self references.

In order to transform infinite types into finite types, we developed the Solve function (based on the "frontier method" [Cousot and Cousot, 1977b]), which iteratively produces better approximations of the sought-after finite types. This method relies on the existence of an upper bound for these types, however, and we discovered three recursion patterns which would produce ever-increasing terms, inducing non-termination. In order to ensure termination, we again applied a technique from abstract interpretation, called widening [Cousot and Cousot, 1977b, 1992a], which sacrifices precision to ensure termination. In our case, this means that some preconditions of recursive functions are not tested as early as they could be.

Finally, after check inference the blame prediction transformation can then introduce check expressions around function call expressions. These check expressions can then be moved upwards and simplified as described in chapter 2. In conclusion, this

chapter has enabled blame prediction for functional programs which contain recursive functions.

In the next chapter we extend the transformation again to also support mutable variables (with the changed semantics from section 3.1). The next chapter also contains the proof that the extended blame prediction transformation is still semantics-preserving, similar to the proof in section 2.6. Finally, at the end of the chapter we speculate on how blame prediction could support additional language features commonly found in programming languages.

4

MUTATION

In the previous two chapters we only considered functional Scheme$_\beta$ programs, i. e. programs which do not use mutation. In this chapter we add support for mutation to the blame prediction transformation (section 4.1). Section 4.2 proves that the blame prediction transformation, extended with support for recursion and mutation, still preserves program semantics. Finally, in the discussion section (section 4.3) we speculate on how to support other language features such as user-defined data structures, objects, non-local control flow, and more.

4.1 Mutation

In the context of program analysis and type systems, mutation is considered to be a prominent member of the "awkward squad" [Peyton Jones, 2001], as most analyses only study functional cores of languages. Mutation severely limits program optimizations, as expressions cannot be reordered freely for example. In the context of type systems, mutation is always limited to be *type-preserving*: an assignment may not change the static type of a variable.

In this section we introduce support for mutation in blame prediction in the form of variable assignment. Unlike mutation in statically typed languages, the blame prediction transformation allows variable assignments to change types of variables. Recall that the goal of blame prediction is to accept *any* program and produce a new program which performs its type tests as early as possible, even in the face of arbitrary mutation.

Consider the program in listing 4.1. Running it in an interpreter will print

```
43 hi hi world
```

Unlike the programs we studied in the previous chapters, the value (and its type) in `tmp` changes over time: on line 2 it is `#f` (boolean), on line 3 it becomes `42` (int), and on line 7 it becomes `"hi"` (string). Moreover, the call to `+` on line 5 triggers a int? check,

while the `string-append` on line 8 triggers a `string?` check. It is an error to move the second check "over" the **`set!`** on line 7, as that is where its type changes.

The result of check insertion for listing 4.1 can be seen in listing 4.2. For brevity we assume there is an implicit `begin` in the body of a `check` expression, such that it can contain multiple expressions. Note how the `check` expression on line 4 has jumped over the **`set!`** on line 5, while the `check` on line 9 has not moved above the set! on line 8.

```
1  (let* ([hold #f]
2         [tmp hold])
3    (set! tmp (or hold 42))
4    (set! hold "hi")
5    (display (+ tmp 1))(display " ")
6    (display hold)(display " ")
7    (set! tmp (or hold 42))
8    (display (string-append tmp " world"))))
```

Listing 4.1: Example of side effects using `set!`

```
1  (let* ([hold #f]
2         [tmp hold])
3    (set! tmp (or hold 42))
4    (check [int? tmp]
5      (set! hold "hi")
6      (display (+ tmp 1))(display " ")
7      (display hold)(display " ")
8      (set! tmp (or hold 42))
9      (check [string? tmp]
10       (display (string-append tmp " world")))))
```

Listing 4.2: Listing 4.1 after `check` insertion

To cope with side effects such as variable mutation, the research community developed techniques to analyse not only the types of expressions, but also the *side effects* they may have. The original *type and effect system* put forward in [Gifford and Lucassen, 1986] simply identified whether an expression is pure, or whether it reads or writes some memory external to the expression. Talpin and Jouvelot [1992] extended this system to take distinct memory regions into account. Jouvelot and Gifford [1991] first proposed an inference system for such systems, such that programmers do not need to add effect annotations. Finally, a "pluggable" generic type and effect system was proposed by Marino and Millstein [2009].

Note that the meaning of "side effects" can be defined in multiple ways. For example, in the work by Gifford and Lucassen [1986] a side effect meant "allocation of, or access to a memory location outside of the expression". More recent work [Marino and Millstein, 2009] has shown that throwing exceptions can also be considered to be a side effect.

For the rest of this chapter, our definition of "side effects" (or just effects) is as follows:

Definition 8 (Side effects)**.** An expression in Scheme$_\beta$ has side effect $\overline{x}$ (which is a set of variables) if, out of all traces[1] which can be generated from the expression, at least one trace records the evaluation of a (set! x s) expression, with $x \in \overline{x}$.

We will consider the side effects of expressions as barriers during the check mobility stage of the blame prediction transformation. Therefore, the smaller the inferred side effect, the more check expressions can be moved upwards.

At run-time only one of the possible traces will effectively be chosen, making $\overline{x}$ a superset of the actual set of variables mutated by the expression. We can always over-approximate the side effect of expressions as $\overline{\star!}$, the set of all variables mutated in the program. However, we can typically derive a much smaller side effect for a given expression, with one exception: the application of a function received as a parameter or returned from a function call. In that case, we must be conservative and assign side effect $\overline{\star!}$.
The rest of this section is subdivided as follows:

- First, we augment the blame prediction transformation with (side) effect inference, in order to determine which variables can be mutated by a given expression (section 4.1.1);
- The results of this effect inference is used in the check mobility stage (section 4.1.2);
- check simplification must also be slightly altered to account for mutable variables (section 4.1.3);
- Finally, we show how the augmented blame prediction transformation handles some common idioms in dynamically typed programming languages (section 4.1.4);

4.1.1 Effect Inference for Scheme$_\beta$

In this section, we define effect inference, a process which determines the side effects of every expression in a Scheme$_\beta$ program. The inferred effects will enable check mobility to move checks upwards where possible, without changing its meaning. To see how side effects interact with check mobility, consider the function in listing 4.3, right after applying check introduction.

We would like to move the check on line 6 upwards as much as possible, but this depends on the side effects the expression `some-expression` has with respect to the global variable `name`. Intuitively, there are four possibilities:

[1] The traces from chapter 2 can be trivially extended to record set! expressions.

```
1 (define name "Dries")
2 ...
3 (define (some-function f)
4   (display (check [string? name] (string-length name)))
5   some-expression
6   (check [string? name] (string-append "Hello, " name)))
```

Listing 4.3: Example of why effect inference is needed

1. `some-expression` definitely mutates the variable `name`:
   In this case, the `check` may not be moved over `some-expression`. The type of `name` might have changed, therefore the `check` on `name` *must* be performed after it has been mutated.

2. `some-expression` definitely mutates the variable `name`, but the assignment preserves the type (`string`): In that case the `check` could be elided.

3. `some-expression` invokes the argument `f`:
   We have no information about the side effects of `f`, therefore we must conservatively assume it can mutate `name` as well. The `check` may not be moved over `some-expression`.

4. `some-expression` definitely does not mutate `name`:
   In this case, the `check` can safely be moved over `some-expression` and merged with the first `check`.

In order to determine which case applies, the blame prediction transformation performs effect inference in addition to the type inference in the previous chapters. With effect inference we can determine which of the cases above is applicable and use this information to correctly move (or not move) `check` expressions upwards. Effect inference must sometimes assign the $\overline{\star!}$ effect, which denotes that the side effects of the expression contain all the mutated variables in the program. The set $\overline{\star!}$ is constructed by traversing the AST at the start of the transformation and collecting all variables which are the target of a `set!` expression.

Figure 4.1 describes our type and effect inference rules for Scheme$_\beta$. The typing rules are almost unmodified from those in section 2.2, so we will focus on the effect part of the rules. Judgments are now of the form $\Gamma \vdash_E \mathtt{e} : \tau \,!\, \overline{x}$, meaning that, according to environment Γ, expression e has type τ and side effect $\overline{x}$. There is one small change in the types: type-level functions are now denoted $\overset{\overline{x}}{\prod}_{(\alpha_1 \ldots \alpha_n)} . \tau$, where $\overline{x}$ is the side effect of applying the function.

Mutable variables receive type $\star$, the "any" type, as introduced in figure 2.4. In this work types are only used to statically eliminate type tests, so functions that are applied on variables of type $\star$ will always result in a run-time type test. The idea is that after the `check` mobility stage, most of the type tests on variables of type $\star$ are grouped right

below expressions which (potentially) mutates such variables. The check simplification stage can then eliminate all duplicate type tests.

In the inference rules, we restrict the scope of side effects to the parts of the program where the affected variables are actually bound. This complicates the rules for expressions which bind variables somewhat. However, restricting the scope of effects like this has the benefit that the effects inferred are only those which are *externally observable*. This significantly aids in the implementation of the check mobility stage, as expressions will have smaller effect sets.

The type- and effect inference rules are as follows:

- TE-VAR looks up the types of non-mutable variables in the environment.
- TE-VAR-MUT always assigns mutable variables type $\star$.
- TE-CONST is unchanged.
- TE-IF states that a conditional expression receives a union type consisting of the types of the two branches, and its effect is the union of the effects of both branches.
- TE-LET uses an immediately-applied type function to combine the types of the expression and body. The effect is the union of the effects of both subexpressions, but without the bound variable x.
- The type returned by TE-LAMBDA is a type-level function with given parameters and body type. The lambda expression itself has no effects, but any side effects of its body (minus those made against the arguments) are stored as an annotation on the returned type function.
- TE-APPLY constructs its return type using the same Apply function as in section 2.2. The effects of a function application are determined by the FunEffect function, shown at the bottom of figure 4.1. This function analyzes the type τ_0 being applied and returns its side effect. Applying a known type function results in the effect stored in the type function, while applying a union type results in the union of the effects. Applying a function whose type is a type variable or the "any type" $\star$ results in the $\overline{\star!}$ effect.
- TE-LETREC is largely the same as in section 3.2. Recall that the transformation described in the beginning of section 3.1 ensures that only functions reside in a letrec binding group, and that all functions are called by at least one of the other functions in the binding group. Therefore, the effect of every function in the letrec binding group is the union of the effects of all functions. The Solve function assigns this effect E to all returned types $\tau_1 \ldots \tau_n$.
- Finally, TE-SET has type void and the intended effect.

$$\boxed{\Gamma \vdash_E \mathsf{e} : \tau \,!\, \overline{\mathsf{x}}}$$

$$\frac{\Gamma(\mathsf{x}) = \tau}{\Gamma \vdash_E \mathsf{x} : \tau \,!\, \emptyset} \quad \text{(TE-VAR)} \qquad \frac{\mathsf{x} \in \overline{\star!}}{\Gamma \vdash_E \mathsf{x} : \star \,!\, \emptyset} \quad \text{(TE-VAR-MUT)} \qquad \frac{}{\vdash_E \mathsf{c} : \mathsf{Typeof}(\mathsf{c}) \,!\, \emptyset} \quad \text{(TE-CONST)}$$

$$\frac{\Gamma \vdash_E \mathsf{e}_1 : \tau_1 \,!\, \overline{\mathsf{x}}_1 \qquad \Gamma \vdash_E \mathsf{e}_2 : \tau_2 \,!\, \overline{\mathsf{x}}_2}{\Gamma \vdash_E (\mathsf{if}\ \mathsf{s}\ \mathsf{e}_1\ \mathsf{e}_2) : \tau_1 \vee \tau_2 \,!\, \overline{\mathsf{x}}_1 \cup \overline{\mathsf{x}}_2} \quad \text{(TE-IF)}$$

$$\frac{\Gamma \vdash_E \mathsf{e}_\mathsf{x} : \tau_\mathsf{x} \,!\, \overline{\mathsf{x}}_\mathsf{x} \qquad \tau_L = \mathsf{Leaves}(\tau_\mathsf{x}) \qquad \Gamma, \mathsf{x} : \tau_L \vdash_E \mathsf{e} : \tau \,!\, \overline{\mathsf{x}}}{\Gamma \vdash_E (\mathsf{let}\ ([\mathsf{x}\ \mathsf{e}_\mathsf{x}])\ \mathsf{e}) : (\Pi_{(\alpha)}.[\tau]\ \tau_\mathsf{x}) \,!\, \overline{\mathsf{x}}_\mathsf{x} \cup \overline{\mathsf{x}}} \quad \text{(TE-LET)}$$

$$\frac{\Gamma, \mathsf{x}_1 : \alpha_1, \ldots, \mathsf{x}_n : \alpha_n \vdash_E \mathsf{e} : \tau \,!\, \overline{\mathsf{x}} \qquad \alpha_1, \ldots, \alpha_n \text{ fresh}}{\Gamma \vdash_E (\mathsf{lambda}\ (\mathsf{x}_1 \ldots \mathsf{x}_n)\ \mathsf{e}) : \overset{\overline{\mathsf{x}}}{\Pi}_{(\alpha_1 \ldots \alpha_n)}.\tau \,!\, \emptyset} \quad \text{(TE-LAMBDA)}$$

$$\frac{\Gamma \vdash_E \mathsf{s}_i : \tau_i \,!\, \emptyset \quad \forall i \in [0 \ldots n] \qquad \ell_f = \mathsf{Label}((\mathsf{s}_0\ \mathsf{s}_1 \ldots \mathsf{s}_n)) \qquad \alpha_1, \ldots, \alpha_n \text{ fresh}}{\Gamma \vdash_E (\mathsf{s}_0\ \mathsf{s}_1 \ldots \mathsf{s}_n) : (\Pi_{(\alpha_1 \ldots \alpha_n)}.[\mathsf{Apply}(\tau_0, \ell_f, \alpha_1 \ldots \alpha_n)]\ \tau_1 \ldots \tau_n) \,!\, \mathsf{FunEffect}(\tau_0)} \quad \text{(TE-APPLY)}$$

$$\frac{\begin{array}{c}\alpha_1, \ldots, \alpha_n \text{ fresh} \qquad \Gamma, \mathsf{x}_1 : \alpha_1, \ldots, \mathsf{x}_n : \alpha_n \vdash_E \mathsf{e}_i : \overline{\tau_i} \,!\, \emptyset \qquad \forall i \in 1 \ldots n \\ \mathrm{E} = \mathsf{FunEffect}(\overline{\tau_1}) \cup \cdots \cup \mathsf{FunEffect}(\overline{\tau_n}) \qquad \tau_1 \ldots \tau_n = \mathsf{Solve}(\overline{\tau_1} \ldots \overline{\tau_n};\ \alpha_1 \ldots \alpha_n;\ \mathrm{E}) \\ \Gamma, \mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_n : \tau_n \vdash_E \mathsf{e} : \tau \,!\, \overline{\mathsf{x}}\end{array}}{\Gamma \vdash_E (\mathsf{letrec}\ ([\mathsf{x}_1\ \mathsf{e}_1] \ldots [\mathsf{x}_n\ \mathsf{e}_n])\ \mathsf{e}) : \tau \,!\, \overline{\mathsf{x}}} \quad \text{(TE-LETREC)}$$

$$\frac{}{\Gamma \vdash_E (\mathsf{set!}\ \mathsf{x}\ \mathsf{s}) : \mathsf{void} \,!\, \{\mathsf{x}\}} \quad \text{(TE-SET)}$$

$$\begin{array}{ll} \mathsf{FunEffect}(\overset{\overline{\mathsf{x}}}{\Pi}_{(\alpha_1 \ldots \alpha_n)}.\tau) & = \overline{\mathsf{x}} \\ \mathsf{FunEffect}(\tau_1 \vee \tau_2) & = \mathsf{FunEffect}(\tau_1) \cup \mathsf{FunEffect}(\tau_2) \\ \mathsf{FunEffect}(\alpha) & = \overline{\star!} \\ \mathsf{FunEffect}(\star) & = \overline{\star!} \\ \mathsf{FunEffect}(\tau) & = \emptyset \end{array}$$

Figure 4.1: Inference of side effects and definition of the FunEffect helper function

One important property of our type and effect system is that we always assign type $\star$ to mutable variables, even if these variables are initialized and mutated with the same types. Listing 4.4 shows such a program: it defines a simple `counter` function which references a global variable `count`. When called, the function increments this number and returns the new number. Note that `count` is initialized with the value `1` (a number) and updated with the results of `(+ count 1)` (also a number). Unfortunately, our type and effect system assigns type $\star$ to `count`. All applications of the function will then result in a run-time type test (because `n` is of type $\star$) even though `counter` always returns a number.

```
1 (define count 1)
2
3 (define (counter)
4   (set! count (+ count 1))
5   count)
```

Listing 4.4: Example of type-preserving mutation

In the absence of type annotations and with the impossibility of raising type errors at compile time, this is the best our system can do. Assume that we optimistically use the initial value for variables to infer their type, and we encounter a set! with a radically different type. At this point, the variable may have been used in other expressions, which only saw the initial type. In order to remain correct, these intermediate expressions thus need to be inferred again, this time with the variable bound to a union type of the old type and the new type. Alternatively, we could raise a type error, but this goes against our goal of accepting *all* programs. In section 4.1.3 we will introduce a new simplification that aims to simplify check expressions which appear immediately after a set!.

To conclude the section on type and effect inference, we will apply it to some small example functions and expressions. For each function in listing 4.5, we will show its type and the set of variables it mutates. Both `x` and `y` are assigned to in this program, so they belong to $\overline{\star!}$ and receive type $\star$. The `set-x` function sets the `x` variable to the given value *only* if it is set to `#f`, but still receives effect { `x` }. `twice` applies the function `f` twice on the argument `w`. Since `f` could mutate potentially *any* variable in the program, it has effect $\overline{\star!}$, even if `f` is a pure function as shown on line 15. Finally, the begin block on line 18 calls `set-x!` (thus mutating `x`, then assigns to `y` and then displays `y`. This expression thus potentially mutates both `x` and `y`.

Now that we have defined effect inference for Scheme$_\beta$ with mutation, we can use the type and effect inference as the basis of check introduction. The check introduction stage only uses the inferred types: check expressions are still inserted around function application expressions according to the inferred types. In the next section we discuss check mobility with mutable variables, which makes use of the inferred side effects.

```
1  (define x #f) ; type = ⋆
2  (define y 9) ; type = ⋆
3
4  ; type = Π_(α_nv).[void ∨ boolean], effect = { x }
5  (define (set-x! new-value)
6    (if (not x)
7      (set! x new-value)
8      #f))
9
10 ; type = Π_(α_f,α_w).[(Π_(α_a).[(Π_(α_b).[α_b] (α_f α_a))] (α_f α_w))], effect = ⋆!
11 (define (twice f w)
12   (f (f w)))
13
14 ; type = (int ?= α_y) · int, effect = ⋆!
15 (twice (lambda (x) (+ x 1)) y)
16
17 ; type = void, effect = { x, y }
18 (begin (set-x! 5)
19        (set! y 32)
20        (display y))
```

Listing 4.5: Examples of type and effect inference

4.1.2 Check mobility

We now turn to the check mobility stage of the blame prediction transformation. Recall from chapter 2 that it is very important that the blame prediction transformation does not change the semantics of the program. In the original formulation of check mobility, we enforced two invariants:

1. checks on variables may not escape the let expressions that bind them. Otherwise, check expressions might perform run-time type tests against variables that are not bound;

2. checks may not escape lambda expressions. Otherwise, run-time type tests might be performed for code that is not executed, e. g. if the lambda is never applied. Additionally, these type tests may reference the arguments of the lambda expression, which are only bound in the function body.

We now add a third invariant, namely

3. checks on mutable variables may not freely move over expressions with side effects which mutate them. If such an expression happens to change the type of the value stored in the variable, moving the run-time type test changes the semantics of the program.

To illustrate this new invariant, consider the code in listing 4.6. As x is a mutable variable, it receives type $\star$ in every expression where it is used, meaning that every

```
1 (define (foo x)                                    ; type of x is unknown
2   (check [number? x] (set! x (+ x 1)))             ; type of x becomes number
3   (check [number? x] (set! x (integer->string x))) ; type of x becomes string
4   (check [string? x] (string-append "Hello, " x)))
```

Listing 4.6: Restrictions on check mobility

use of x in a primitive function will generate a check. The comments on each line indicate the *run-time type* of the value of x. If the `number?` test on x on line 2 succeeds, a number is assigned to x. The next line replaces this value with a string, using the `integer->string` function. Finally, the last line applies the `string-append` function to x, which requires it to be a string.

If mutability is not taken into account, check mobility would move the `string?` test to the top of the function, together with the `number?` test. A variable cannot contain a value that is both a `number?` and a `string?`, so the resulting program would predict blame for a program that otherwise succeeds.

By enforcing the third invariant, however, the `string?` test is not allowed to move over the set! expression on line 3. Likewise, the `number?` test on line 3 is not allowed to move over the set! expression on line 2. The function thus remains as shown and the transformed program only reports an error if `foo` is called with a value other than a number. While the checks on lines 3 and 4 have not been moved, they will be eliminated in the check simplification stage, in section 4.1.3.

In figure 4.2 we present a set of rules (updated from those in section 2.4) for check mobility. Every rule is of the form $e \to e' \uparrow p$, meaning that check mobility transforms expression e to e′, propagating the predicate p upwards. The new and updated check mobility rules are as follows:

- The FE-SET rule handles mobility for set! expressions. Since its argument expression is always a simple expression like a constant value, a variable or a lambda expression, set! expressions never propagate preconditions upwards. However, the argument might be a lambda expression, so check mobility still needs to be applied to the argument.
- FE-LET is similar to the rule in section 2.4, where only the checks in the body that do not involve x are propagated. However, any predicate on variables mutated by e_x cannot be moved up beyond the let expression. This is done by a generalized version of the mask function from section 2.4, which masks the predicate p against a *set* of variables instead of one variable. In order to enforce the third invariant, preconditions are masked against variable x and all variables $\overline{x}$ mutated by e_x.

As before, any predicates which are propagated out of the program are captured and inserted into the program again.

$$\boxed{e \to e' \uparrow p}$$

$$\frac{e \to e' \uparrow p}{e \to_p (\mathsf{check}\ p\ e')}\ (\textsc{FE-program}) \qquad c \to c \uparrow \#t\ (\textsc{FE-const})$$

$$\frac{s \to s' \uparrow \#t}{(\mathsf{set!}\ x\ s) \to (\mathsf{set!}\ x\ s') \uparrow \#t}\ (\textsc{FE-set}) \qquad x \to x \uparrow \#t\ (\textsc{FE-var})$$

$$\frac{s_i \to s_i' \uparrow \#t \qquad \forall i \in 0 \ldots n}{(\mathsf{check}\ p\ (s_0\ s_1 \ldots s_n)) \to (s_0'\ s_1' \ldots s_n') \uparrow p}\ (\textsc{FE-apply})$$

$$\frac{s \to s' \uparrow \#t \qquad e_1 \to e_1' \uparrow p_1 \qquad e_2 \to e_2' \uparrow p_2}{(\mathsf{if}\ s\ e_1\ e_2) \to (\mathsf{if}\ s'\ (\mathsf{check}\ p_1\ e_1')\ (\mathsf{check}\ p_2\ e_2')) \uparrow p_1 \vee p_2}\ (\textsc{FE-if})$$

$$\frac{e_x \to e_x' \uparrow p_x \qquad e \to e' \uparrow p \qquad \overline{x} = \mathsf{Effects}(e_x) \qquad p^{\uparrow} = \mathsf{mask}(x \cup \overline{x},\ p)}{(\mathsf{let}\ ([x\ e_x])\ e) \to (\mathsf{let}\ ([x\ e_x'])\ (\mathsf{check}\ p\ e')) \uparrow p_x \wedge p^{\uparrow}}\ (\textsc{FE-let})$$

$$\frac{\begin{array}{c} e_i \to e_i' \uparrow \#t \qquad \forall i \in 1 \ldots n \\ e \to e' \uparrow p \qquad p' = \mathsf{mask}(x_1 \ldots x_n,\ p) \end{array}}{(\mathsf{letrec}\ ([x_1\ e_1] \ldots [x_n\ e_n])\ e) \to (\mathsf{letrec}\ ([x_1\ e_1'] \ldots [x_n\ e_n'])\ (\mathsf{check}\ p\ e')) \uparrow p'}\ (\textsc{FE-letrec})$$

$$\frac{e \to e' \uparrow p}{(\mathsf{lambda}\ (x_1 \ldots x_n)\ e) \to (\mathsf{lambda}\ (x_1 \ldots x_n)\ (\mathsf{check}\ p\ e')) \uparrow \#t}\ (\textsc{FE-lambda})$$

Figure 4.2: Check mobility in the presence of side effects

4.1.3 Check Simplification

After check introduction and mobility, the resulting program still contains some extraneous check expressions and -predicates. In section 2.5, we defined three check simplification rules, which either remove check expressions or simplify their predicates. The simplification rules were as follows:

Or–true simplification check expressions where the predicate is of the form $p \vee \#t$ or $\#t \vee p$ can be elided, as the type tests in the non-#t branch will be immediately repeated in the corresponding conditional branch.

And–check simplification Sometimes a predicate contains multiple type tests on the same variable, for example $p_1 \wedge p_1 \wedge p_2 \wedge p_1 \wedge p_3 \wedge p_2$ can be simplified to $p_1 \wedge p_2 \wedge p_3$. All but the first of the duplicate tests on each variable can be removed, merging the blame labels into the remaining test.

Check–check simplification Similar to and–check simplification, sometimes a check expression tests a predicate in the scope of a check expression higher up in the program, where the same predicate has already been tested. Again, the predicate in the inner check expression can be removed and its labels merged with the predicate in the outer check expression.

These simplification rules still apply, with one important caveat: check–check simplification on a mutable variable may not be applied if there is an expression between the two check expressions which mutates the variable. The function in listing 4.7 shows why this restriction is necessary. It first uses the variable x in an addition (line 3), then sets it to the result of (f x 5) (line 4), and finally tries to use it as a number again (line 6). Check–check simplification clearly applies to the check expressions on lines 2 and 5. However, the type of the value in x is potentially changed on line 4, depending on the function f which is unknown. Eliminating the inner check on line 5 could thus result in the application of + raising a run-time type error.

```
1 (define (simple f x)
2   (check (number? x)
3     (let ([y (+ x 1)])
4       (set! x (f x 5))
5       (check (number? x)
6         (+ x 1)))))
```

Listing 4.7: Example where check–check simplification is not appropriate

As we remarked in section 4.1.1, the choice of always using type $\star$ for mutable variables results in extraneous check expressions which accumulate after a set!. We now concretize this remark as an extra simplification:

Set–check simplification Typically the expression used as argument to a set! expression is one with a ground type, rather than a union type or type variable. At the same time, the first expression after the set! is often a check expression with a precondition on the mutated variable. In such cases, we allow *set–check simplification* to simplify the precondition of the check with regards to the mutated variable.

Figure 4.3 shows the general template for set–check simplification: if the pattern on the left-hand side appears with a type judgment $\Gamma \vdash \mathsf{s} : \gamma \,!\, \varnothing$ then any predicates of the form $\gamma?\ \mathsf{x}$ may be replaced by #t.

As in the check simplification stage for functional programs, simplification continues until no rules are applicable.

With the addition of set–check simplification, however, the rules can no longer be applied in an arbitrary order. The snippet in listing 4.8 demonstrates this: if the check on line 3 is removed first by set–check simplification, the check on line 5 remains. If check–check simplification is performed first, however, the check on line 5 is removed, followed by the check on line 3. In order to remove a maximal number of checks from the program, set–check simplification must be performed last.

```
(begin ...
       (set! x s) ; s has type γ
       (check (... (γ? x) ...) e)
       ...)
```

(a) Template to be matched, for some type γ and simple expression s

```
(begin ...
       (set! x s)
       (check (... #t ...) e)
       ...)
```

(b) Resulting simplified expression

Figure 4.3: Template for set–check simplification

```
1 (let ([x 5])
2   (set! x 42)
3   (check (int? x)
4     ...
5     (check (int? x)
6       ...)))
```

Listing 4.8: Example of when check–check simplification cannot be applied first

Having shown how the blame prediction transformation needs to be adapted to deal with mutation, we now show how it applies to some examples.

4.1.4 Examples

Example: Memoizing Consider the code in listing 4.9: a simple example of applying memoization to an expensive computation. The variable `computed?` is used to remember whether the expensive computation has taken place, the result of this computation is stored in `computed-value`.

```
1 (define computed-value #f)
2 (define computed?      #f)
3
4 (define (get-or-compute)
5   (if computed?
6     computed-value
7     (begin (set! computed? #t)
8            (set! computed-value (expensive-computation))
9            computed-value)))
```

Listing 4.9: Caching expensive computations

The type and effect inference as defined in section 4.1.1 assigns type $\star$ to both `computed?` and `computed-value`, as they are mutated in the program. The `get-or-compute` function, meanwhile, has type $\star \vee \star$, or just $\star$. At the start of the program, both `computed?` and `computed-value` contain a value of type `boolean`. After invoking `get-or-compute` once, `computed-value` will contain the value computed by `expensive-computation`.

Example 2: Read–process–write Listing 4.10 defines a common helper function to perform an arbitrary operation on a file and write back the results. The operation to be performed is defined in the `process` function, which is passed in by the programmer. This function should return a string, as the `write-file` function requires a string for the file's body. The blame prediction transformation will generate a check for (string? `input`) around `write-file`. The check mobility stage will attempt to move this test upwards, past the set!. Because the set! modifies the variable `input`, the check is not allowed to go past the set!. At run-time, this check correctly verifies whether `process` returned a value of type string.

```
1 (define (with-file-do filename process)
2   (let ([input (read-file filename)])
3     (set! input (process input))
4     (write-file filename input)))
```

Listing 4.10: Example of a common read–process–write helper function

4.2 Proof of Safety

In this section we show that the extensions made to the blame prediction transformation do not affect program correctness. We first prove two lemmas which establish basic properties of the blame prediction transformation (sections 4.2.1 and 4.2.2). Next, we extend the trace semantics first presented in section 2.6 with support for mutation (section 4.2.3). Finally, we prove the safety properties first defined in section 2.6 using these trace semantics (section 4.2.4).

4.2.1 Correctness of Effect Inference

Central to our proof of correctness is the effect inference from section 4.1.1: for any given expression e, it must correctly infer its side effect (as per definition 8 in section 4.1), i. e. all variables $\overline{x}$ that *might* be mutated by the evaluation of that expression. Formally, a variable x *might* be mutated by an expression e if, out of all traces which can be generated from the expression, at least one trace mutates the variable. Unfortunately, the number of traces becomes infinite if the expression invokes a function passed by argument or extracted from a data structure. In that case our analysis must be conservative and assign the $\overline{\star!}$ effect, which is trivially correct for any expression. We start by showing that the type and effect inference correctly *over-approximates* the set of variables which are mutated.
As a simple example, consider the snippet below:

```
1 (if (even? x)
2   (set! even (+ even 1))
3   (set! odd (+ odd 1)))
```

It modifies either even or odd, but which variable exactly depends on the run-time value of x. Effect inference must therefore be conservative and infer the side effect of this expression as $\{\texttt{even}, \texttt{odd}\}$.

Lemma 5. Let e be a Scheme$_\beta$ expression; then the judgment $\Gamma \vdash_E \texttt{e} : \tau \,!\, \overline{\texttt{x}}$ infers the side effect of the expression e as $\overline{\texttt{x}}$.

Proof. By induction on the evaluation rules from section 3.1:

- E-VAR, E-CONST: evaluation simply returns a value, no side effects.
- E-LAMBDA: this evaluation rule constructs a closure from current environment and the function body. This evaluation rule also has no side effects, as it simply returns a value. Applying the induction hypothesis on the body e yields that it has side effect $\overline{\texttt{x}}$. In the effect inference rule TE-LAMBDA, this side effect is stored as an annotation on the returned type, we will use this fact in the E-APPLY rule.
- E-IF: the condition is a simple expression, so it cannot have side effects. The side effects of this expression are those of the chosen branch, which is unpredictable. It is guaranteed that the union of the side effects of both branches covers the side effects of whichever branch is chosen, which is what TE-IF does.
- E-APPLY: all sub-expressions of a function application are simple expressions, so they cannot produce side effects. The side effects of a function application entirely depend on the function being applied. There are two possibilities, given the inferred type τ_0 of the function $\texttt{s}_0$:
  1. τ_0 is a union type where one or more branches is a type variable or $\star$:
     By inversion on the type, the value $\texttt{v}_0$ can be any function. In the absence of precise information, we must assume it can affect *any variable*, therefore such a function application has side effect $\overline{\star!}$. This is exactly the definition of FunEffect.
  2. Otherwise, τ_0 is fully known or a union type where all branches are concrete type functions:
     In this case the function being applied must fit one of these type functions. As with the conditional expression, by taking the union of side effects we are certain to cover all mutated variables.
- E-LET: we apply the induction hypothesis to $\texttt{e}_\texttt{x}$, which yields a side effect $\overline{\texttt{x}}_\texttt{x}$. Applying the induction hypothesis to the body yields side effect $\overline{\texttt{x}}$. The side effect of the let expression is thus just $\overline{\texttt{x}}_\texttt{x} \cup \overline{\texttt{x}}$.
- E-CHECK: the predicate of a check expression cannot contain set! expressions by construction. The side effect of a check expression is thus that of its body.

- E-LETREC: As we remarked in section 3.1, binding groups of letrec expressions only contain functions which (in)directly call every other function. When the body applies one of these functions, it can either follow a path where it calls one or more of the other functions, or it can follow a path where it does not. In the first case, that function again has paths where it calls other functions, and paths where it does not. We can cover the side effects of all possible paths through the functions by taking the union of the side effects of the individual functions. The TE-LETREC rule annotates each type function with this over-approximating side effect, which will be ascribed to every function call which invokes one of the bound functions. Otherwise, side effects in the letrec body will occur as normal.
- E-SET: Finally, the side effect of a set expression is exactly the variable it modifies, as the argument is again a simple expression. □

Having proved that effect inference calculates side effects correctly, we can focus on the correctness of the blame prediction transformation in the presence of mutable variables.

4.2.2 Program Properties after Blame Prediction

Lemma 6. The check mobility and simplification stages preserve the following properties:

1. check expressions never refer to unbound variables;
2. check expressions verify the same values as the preconditions they guard.

The proof of lemma 6 is split across the check mobility and check simplification stages. We begin the proof with check mobility.

Proof. For check mobility, we prove the lemma by induction over the mobility rules (figure 4.2).

- FE-PROGRAM simply captures any preconditions that escape the program and reinserts them.
- FE-CONST, FE-SET and FE-VAR do not propagate any preconditions upwards.
- FE-APPLY simply propagates any preconditions attached to it. This is the base case, as any preconditions start from the function application expressions.
- FE-IF lifts the preconditions p_1 and p_2 out of its branches. Property 1 is preserved because a conditional expression does not introduce new bindings, and property 2 is preserved because the condition s cannot contain any set! expressions.

- FE-LET is the most important rule in regards to the two properties. Regarding property 1, it must ensure that the preconditions to be propagated do not mention x, as that would introduce a reference to an unbound variable. Regarding property 2, preconditions on variables mutated by the expression e_x may not be lifted out of the body. Both are satisfied by the generalized mask function: it masks p against x and any of the variables $\bar{x}$ mutated by e_x. $p^{\uparrow}$ is thus guaranteed not to contain any references to any of these variables. Moreover, lemma 5 states that e_x mutates *at most* the variables returned by the Effects function.
- FE-LAMBDA, as before, prevents any preconditions from floating out of the lambda expression. □

Proof. For the check simplification stage, we simply enumerate the simplifications (section 4.1.3):

- **Or–true simplification** removes predicates of the form $\#t \vee p$, where p would never be evaluated. checks are removed, not added, so both properties still hold.
- **And–check simplification** merges multiple identical predicates on the same variable in an aggregate predicate. From the previous stage we know that the individual preconditions would not reference unbound variables or check values different from the blame label, therefore the new predicate maintains these properties.
- **Check–check simplification** maintains the first property: we remove an innermost check expression which must already satisfy the first property, and we merge it with the outermost check expression, which must also already satisfy it.

  There is potential to violate the second property, however. Duplicate predicates in nested check expressions may only be merged into outer check expressions if the expressions between the check expressions are guaranteed not to mutate the variable being tested. As we remarked in the description of check–check simplification, check mobility depends on the output of effect inference, and in lemma 5 we showed that expressions do not mutate any variables outside their inferred effect set. Therefore, the mutation-aware check–check simplification maintains the second property.
- **Set–check simplification** removes check expressions which appear directly after a set!. Both properties must hold before *and* after this simplification. □

4.2.3 Traces and Trace Semantics

In the next section we prove that the blame prediction transformation, augmented with support for recursion and mutation, still preserves the program semantics. Before we can prove the three properties described in section 2.6, however, we need to extend

traces to record the effects of mutating variables. We defined traces in section 2.6 as a means of matching up check expressions (check elements) with the primitive operations whose preconditions they verify (use elements). With the addition of an explicit heap and mutable variables to Scheme$_\beta$, we need to keep track of the type of their values when they are used in a check or primitive operation. In order to do that we record the run-time type of a mutable variable whenever one is initialized or mutated. Subsequent check(p) elements in the trace can then be verified against the last known type.

The extended definition of traces is shown in figure 4.4. Appending a step s to a trace T is denoted as $\mathsf{T} \cdot \mathsf{s}$. Traces can consist of a use(p) or check(p) step like in section 2.6, or a "set type" step $\ell \leftarrow \tau$. A "set type" step indicates that a memory location belonging to a mutable variable has been set, changing the run-time type of its value to τ.

To obtain traces, we execute the program in a modified version of the semantics presented in section 3.1. The modified semantics, shown in Figure 4.4, build up a trace T while evaluating the program. We will only discuss the most important tracing rules, the others simply thread the trace through the evaluation.
In these tracing semantics, rules take the form $\mathcal{H}, \mathcal{E}, \mathsf{e}, \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}', \mathsf{v}, \mathsf{T}'$, meaning that evaluating expression e with heap $\mathcal{H}$ and environment $\mathcal{E}$ results in a value v and new heap $\mathcal{H}'$, extending the trace T to T′.

In contrast to the tracing semantics of section 2.6, traces track memory locations rather than variable names. The reasoning behind this is that recursive function invocations can cause multiple copies of the same variable name to appear in a trace, while the scoping enforced by the environment ensures that their memory locations are distinct. Aliasing multiple variables to the same memory location is ruled out by definition, as every variable in the program uniquely corresponds to one memory location.

The modified rules are as follows:

- ET-LET evaluates the expression e_x and updates the heap to store its value to a new memory location ℓ. If the variable being bound is not in the "mutated variables" set (i. e. $x \notin \overline{\star!}$), the memory location is not recorded in the trace. If the variable being bound is mutated in the program however, the trace is amended to note that the newly allocated memory location now contains a value of type τ. The Typeof function determines the run-time type of a value, so either a ground type like int, string or a type function.

- ET-CHECK records the fact that the precondition p has been tested. We use the syntax $\mathsf{p}[\mathcal{E}[\mathsf{x}]/\mathsf{x}]$ to denote "replace all variable references in p with their location in the heap".

- ET-SET updates the trace to note that the variable x has been updated.

To illustrate the tracing semantics, let us examine the trace produced by the program listing 4.11. This program only has one possible trace, but in general there are many

$$
\begin{array}{lcll}
\mathsf{T} & ::= & \mathsf{T} \cdot \mathsf{s} \mid \epsilon \mid \mathsf{err\text{-}blame(p)} \mid \mathsf{err\text{-}use(p)} & \\
\mathsf{s} & ::= & \mathsf{use(p)} & \text{use step, as before} \\
& \mid & \mathsf{check(p)} & \text{check step, as before} \\
& \mid & \ell \leftarrow \tau & \text{Update memory at } \ell \text{ to type } \tau
\end{array}
$$

$$\boxed{\mathcal{H}, \mathcal{E}, \mathsf{e}, \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}', \mathsf{v}, \mathsf{T}'}$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{E}, \mathsf{e_x}, \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}_1, \mathsf{v_x}, \mathsf{T}_1 \\ \ell \text{ fresh} \qquad \mathcal{E}' = \mathcal{E}[\mathsf{x} \to \ell] \qquad \mathcal{H}_2 = \mathcal{H}_1[\ell \to \mathsf{v_x}] \\ \tau_{\mathsf{x}} = \mathsf{Typeof}(\mathsf{v_x}) \qquad \begin{cases} \mathsf{T}_2 = \mathsf{T}_1 \cdot (\ell \leftarrow \tau_{\mathsf{x}}) & \text{if } \mathsf{x} \in \overline{\star!} \\ \mathsf{T}_2 = \mathsf{T}_1 & \text{otherwise} \end{cases} \\ \mathcal{H}_2, \mathcal{E}', \mathsf{e}, \mathsf{T}_2 \rightsquigarrow_{\mathsf{T}} \mathcal{H}_3, \mathsf{v}, \mathsf{T}_3 \end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{let}\ ([\mathsf{x}\ \mathsf{e_x}])\ \mathsf{e}), \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}_3, \mathsf{v}, \mathsf{T}_3} \quad (\text{ET-LET})$$

$$\frac{\mathcal{H}, \mathcal{E}, \mathsf{p}, \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}, \mathsf{v_p}, \mathsf{T} \qquad \begin{cases} \mathsf{T}_1 = \mathsf{err\text{-}blame(p)} & \text{if } \mathsf{v_p} = \#\mathsf{f} \\ \mathcal{H}, \mathcal{E}, \mathsf{e}, \mathsf{T} \cdot \mathsf{check}(\mathsf{p}[\mathcal{E}[\mathsf{x}]/\mathsf{x}]) \rightsquigarrow_{\mathsf{T}} \mathcal{H}_1, \mathsf{v}, \mathsf{T}_1 & \text{otherwise} \end{cases}}{\mathcal{H}, \mathcal{E}, (\mathsf{check\ p\ e}), \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}_1, \mathsf{v}, \mathsf{T}_1} \quad (\text{ET-CHECK})$$

$$\frac{\mathcal{H}, \mathcal{E}, \mathsf{s}, \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}, \mathsf{v}, \mathsf{T} \qquad \mathcal{H}_1 = \mathcal{H}[\mathcal{E}[\mathsf{x}] \to \mathsf{v}] \qquad \tau_{\mathsf{x}} = \mathsf{Typeof}(\mathsf{v}) \qquad \mathsf{T}_1 = \mathsf{T} \cdot (\mathcal{E}[\mathsf{x}] \leftarrow \tau_{\mathsf{x}})}{\mathcal{H}, \mathcal{E}, (\mathsf{set!\ x\ s}), \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}_1, \mathsf{void}, \mathsf{T}_1} \quad (\text{ET-SET})$$

$$\mathcal{H}, \mathcal{E}, \mathsf{x}, \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}, \mathcal{H}[\mathcal{E}[\mathsf{x}]], \mathsf{T} \quad (\text{ET-VAR}) \qquad \mathcal{H}, \mathcal{E}, \mathsf{c}, \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}, \mathsf{c}, \mathsf{T} \quad (\text{ET-CONST})$$

$$\mathcal{H}, \mathcal{E}, (\mathsf{lambda}\ (\mathsf{x}_1 \ldots \mathsf{x}_n)\ \mathsf{e}), \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}, \langle \mathcal{E}, \mathsf{x}_1 \ldots \mathsf{x}_n, \mathsf{e} \rangle, \mathsf{T} \quad (\text{ET-LAMBDA})$$

$$\frac{\mathcal{H}, \mathcal{E}, \mathsf{s}, \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}, \mathsf{v}_c, \mathsf{T} \qquad \begin{cases} \mathcal{H}, \mathcal{E}, \mathsf{e}_2, \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}_1, \mathsf{v}, \mathsf{T}_1 & \text{if } \mathsf{v}_c = \#\mathsf{f} \\ \mathcal{H}, \mathcal{E}, \mathsf{e}_1, \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}_1, \mathsf{v}, \mathsf{T}_1 & \text{otherwise} \end{cases}}{\mathcal{H}, \mathcal{E}, (\mathsf{if\ s\ e}_1\ \mathsf{e}_2), \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}_1, \mathsf{v}, \mathsf{T}_1} \quad (\text{ET-IF})$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{E}, \mathsf{s}_i, \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}, \mathsf{v}_i, \mathsf{T} \qquad \forall i \in [0 \ldots n] \\ \mathcal{H}_1, \mathcal{E}_1, \mathsf{e} = \delta(\mathcal{H}, \mathcal{E}, \mathsf{v}_0, \mathsf{v}_1 \ldots \mathsf{v}_n) \qquad \mathcal{H}_1, \mathcal{E}_1, \mathsf{e}, \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}_2, \mathsf{v}, \mathsf{T}_1 \end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{s}_0\ \mathsf{s}_1 \ldots \mathsf{s}_n), \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}_2, \mathsf{v}, \mathsf{T}_1} \quad (\text{ET-APPLY})$$

$$\frac{\begin{array}{c}\ell_1, \ldots, \ell_n \text{ fresh} \qquad \mathcal{E}_1 = \mathcal{E}[\mathsf{x}_1 \to \ell_1, \ldots, \mathsf{x}_n \to \ell_n] \qquad \mathsf{T}_0 = \mathsf{T} \\ \begin{cases} \mathcal{H}, \mathcal{E}_1, \mathsf{e}_i, \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}, \mathsf{v}_i, \mathsf{T} \quad \mathsf{T}_i = \mathsf{T}_{i-1} \cdot (\ell_i \leftarrow \mathsf{Typeof}(\mathsf{v}_i)) & \text{if } \mathsf{x}_i \in \overline{\star!} \\ \mathcal{H}, \mathcal{E}_1, \mathsf{e}_i, \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}, \mathsf{v}_i, \mathsf{T} \quad \mathsf{T}_i = \mathsf{T}_{i-1} & \text{otherwise} \end{cases} \quad \forall i \in [1 \ldots n] \\ \mathcal{H}_1 = \mathcal{H}[\ell_1 \to \mathsf{v}_1, \ldots, \ell_n \to \mathsf{v}_n] \qquad \mathcal{H}_1, \mathcal{E}_1, \mathsf{e}, \mathsf{T}_n \rightsquigarrow_{\mathsf{T}} \mathcal{H}_2, \mathsf{v}, \mathsf{T}_{n+1} \end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{letrec}\ ([\mathsf{x}_1\ \mathsf{e}_1] \ldots [\mathsf{x}_n\ \mathsf{e}_n])\ \mathsf{e}), \mathsf{T} \rightsquigarrow_{\mathsf{T}} \mathcal{H}_2, \mathsf{v}, \mathsf{T}_{n+1}} \quad (\text{ET-LETREC})$$

Figure 4.4: Modified semantics for generating traces

traces a program might produce, depending on external inputs such as user input, the random seed, contents of the file system, network communication, etc. We will not consider these factors: we will be comparing *every* path through the same program, before and after the blame prediction transformation.

```
1 (let* ([counter 0]
2        [add! (lambda (x)
3          (let ([new-counter (+ counter x)])
4            (set! counter new-counter)))])
5   (display counter)
6   (add! 5)
7   (display counter)
8   (add! 2)
9   (display counter))
```

Listing 4.11: Example program for tracing semantics

The trace is then:

$$\mathsf{T} = [\underbrace{\ell_c \leftarrow \mathsf{int}}_{\text{line 1}}, \underbrace{\mathsf{check}(\mathsf{number?}\ \ell_c)}_{\text{line 3}}, \underbrace{\ell_c \leftarrow \mathsf{int}}_{\text{line 4}}, \underbrace{\mathsf{check}(\mathsf{number?}\ \ell_c)}_{\text{line 3}}, \underbrace{\ell_c \leftarrow \mathsf{int}}_{\text{line 4}}]$$

The first part of the trace shows how the let expression assigns type int to the memory location ℓ_c, which it allocated for the variable `counter`. The memory location of `add!` is not recorded, as it is not a member of $\overline{\star!}$. The next two parts of the trace are the invocation of `add!` on line 6, which adds 1 to `counter` (the check), then stores the result back (the $\leftarrow$ part). The final two parts are identical to the second part.
This trace exhibits two important properties:

1. First, mutable variables (or rather the memory locations they point to) are always initialized with a type. It is an error to refer to a variable before it is bound, which corresponds to the first property of section 4.2.2.

2. Second, every check refers to the type of the value that was *last assigned* to a particular memory location. We could therefore rearrange checks in a trace, as long as they do not skip over an assignment of the variable(s) they inspect. This corresponds to the second property of section 4.2.2.

4.2.4 Proof of equivalences

Having defined traces for Scheme$_\beta$ with mutation, we now turn to proving the two equivalences [2]from section 2.6. They are repeated below, where P is the input program and P′ is the transformed program. Note that the equivalences are formulated slightly differently: with the introduction of recursion it is possible to construct programs

[2]As in chapter 2, the second equivalence is constructed from two entailments.

that loop indefinitely. This requires a small modification to the definition of value preservation and blame–use entailment:

Value Preservation
Iff P runs to completion and produces a value v, P' produces the same value v.
Formally: $P \rightsquigarrow v \Leftrightarrow P' \rightsquigarrow v$.
Use–Blame Entailment
If P raises an error, the transformed program P' must predict blame.
Formally: $P \rightsquigarrow \mathsf{err}\text{-}\omega \Rightarrow P' \rightsquigarrow \mathsf{err}\text{-}\mathsf{blame}(p)$.
Blame–Use Entailment
If P' predicts blame, the original program P must raise an error or loop indefinitely.
Formally: $P' \rightsquigarrow \mathsf{err}\text{-}\mathsf{blame}(p) \Rightarrow P \rightsquigarrow \mathsf{err}\text{-}\omega \vee P \not\rightsquigarrow$.

Before we prove the equivalences, we prove lemma 7, which is analogous to lemma 3.

Lemma 7. Check introduction guards all type tests that could fail in primitive operations with a check expression.

Proof. The proof for this is largely analogous to the proof in section 2.6. In the chapter on recursion we defined a widening operator which eliminated conditional types from infinite types in order to guarantee termination. These conditional types give rise to type tests inside the function definitions, however, so this property is still holds for recursive functions.

In this chapter we defined check inference to always assign type $\star$ to mutable variables, as per inference rule TE-VAR-MUT. Therefore, any use of these variables will generate a conditional type and thus a check expression, trivially satisfying the property. □

Proof of Value Preservation

Proof. We prove each direction separately.
$\Rightarrow$: Given that $P \rightsquigarrow v$, prove that $P' \rightsquigarrow v$.
Analogously to the proof in section 2.6, we start by comparing the trace T of P with T' from the transformed program P'. P returned a value, so it finished and every $\mathsf{use}(p)$ in the trace T succeeded. check introduction will insert a check expression to guard every primitive operation that might fail (see lemma 7). We now use lemma 6, which states that the mobility and simplification stages will preserve the following properties of check expressions: 1) no unbound variables are referenced, and 2) the preconditions verify the same values as the primitive operations they guard

For every check in T', there is a use in T' (and thus T) that requires the same predicate to be true. Every check and use expression in T succeeds, thus T' cannot contain any errors; it must produce a value.

$\Leftarrow$: Given that $\mathsf{P}' \rightsquigarrow v$, prove that $\mathsf{P} \rightsquigarrow v$.
The trace T' of program P' is equivalent modulo check to the trace T of program P. In other words, removing the checks from T' yields T. T' succeeds, so T must also succeed, and thus program P must produce the same value as program P'. □

Proof of Use-Blame Entailment

Proof. Evaluation of the program P raises an error on a use(p). We discriminate on the form of p:

1. $\mathsf{p} = (\tau?\ \mathsf{c})$: a check expression would be generated and propagated upwards as high as possible, the proof is analogous to that in section 2.6.

2. $\mathsf{p} = (\tau?\ \mathsf{x})$, with $\mathsf{x} \notin \overline{\star!}$: the check expression for this precondition is propagated as high as possible, up to the binding of x or the nearest enclosing function. The proof is also analogous to that in section 2.6.

3. $\mathsf{p} = (\tau?\ \mathsf{x})$, with $\mathsf{x} \in \overline{\star!}$: in this case, the check expression must be propagated up to the binding of x, the nearest enclosing function, or the most recent assignment of x. Lemma 6 ensures that both the check and the use expressions verify the type of the same value. Therefore, at the very latest the check expression will predict blame for the use expression. □

Proof of Blame-Use Entailment

Proof. P' evaluates to an err-blame(p) error, which must be raised by a $\mathsf{check}(\mathsf{p}_k)$ expression in T'. The proof is analogous to that in section 2.6, the only difference is the presence of $\ell \leftarrow \tau$ steps in the trace, but lemma 6 guarantees that these do not affect the values tested by check expressions. Alternatively, it is possible that P loops indefinitely instead of producing an error; we present one such example below. □

Note that blame–use entailment does not hold if the original program loops indefinitely. In listing 4.12 for example, the `collatz` function is invoked on a large number. If this function returns, `sqrt` is applied to the value of x, which is mistakenly defined as a string. It is an open mathematical problem whether the `collatz` function stops for all inputs, so it is possible that the `sqrt` is never reached.

Blame prediction will however always propagate the (number? x) precondition of the `sqrt` function to right below the definition of x, on line 1. In this case it will point out an error that might never manifest itself, which in turn hinders programmers attempting to develop and debug their program. Unfortunately, we do not foresee a general solution to this problem, bar specific situations such as "`(while true ...)`" loops. This and other patterns could be treated specially by the blame prediction transformation in the future. More involved techniques such as abstract interpretation could discover more infinite loops, at the cost of more processing time.

```
(define x "5")

(define (collatz n)
  (if (= n 1)
    #t
    (if (even? n)
      (collatz (/ n 2))
      (collatz (+ 1 (* 3 n))))))

(collatz 109889544521617720)

(display (sqrt x))
```

Listing 4.12: Pathological case for programs that contain loops

4.2.5 Conclusion

In the first part of this chapter we developed support for mutation in Scheme$_\beta$. We remarked that other type systems and program analysis techniques have varying support for mutation: either they support a subset of the language without mutation, they only support type-preserving mutation, or they only support mutation of local variables. As our goal is to help programmers find errors in *unmodified* code, we have opted to support full-blown type-changing mutation of all variables in the program.

We first defined how type- and effect inference is performed for Scheme$_\beta$. This analysis complements the type inference of chapter 2 by inferring the side effects of every expression in the program. During type inference, all mutable variables (i. e. variables mutated anywhere in the program) received type $\star$, the "any" type. This means that passing mutable variables to primitive operations always results in a check expression and thus a run-time type test.

In the check mobility stage, type tests are moved upwards as before. The expressions with side effects now act as a barrier for type tests: a check predicate involving a variable x may not be moved over an expression which can potentially mutate it. The most extreme barriers are applications of variables whose type is a type variable (such as function arguments): they block *all* predicates involving mutable variables.

Finally, in the check simplification stage some check expressions can be eliminated again. From the simplifications in section 2.5, two ("or–true" and "and–check" simplification) can be applied without modifications. The third, check–check simplification — which removes check predicates covered by an earlier check expression — can be applied as well, with the caveat that the variable in the predicate may not have been mutated inbetween the check expressions. We defined a fourth simplification which attempts to remove check expressions which were moved to just under a set! expression: if the type of the value being assigned matches the type being tested in a predicate of the check expression immediately afterward, the check can be simplified.

This concludes our discussion on mutable variables in blame prediction. In the second part of this chapter we will discuss how the blame prediction transformation could be expanded to over the remaining features of a modern dynamically typed language.

4.3 Discussion and Future Work

The goal of this chapter was to expand blame prediction to take mutation into account. In this section we discuss some remaining features of dynamically typed languages and how they could be implemented in the blame prediction transformation.

4.3.1 Blame Prediction for Compound Data Structures

In this chapter we have developed support for blame prediction with mutable variables. However, any non-trivial program also uses compound data structures such as lists, vectors, dictionaries, ... In this section we speculate on how basic support for these types could be added and discuss what would be necessary to fully support (im)mutable compound data structures.

Looking at the documentation of a modern dynamically typed programming Scheme-like language such as Racket, we can find the following compound data structures:

Pairs and Lists The most basic compound data structure available is the immutable pair which can store two values, historically accessed using `car` and `cdr`. Lists can be created by nesting pairs such that the second value is either the empty list, or again a pair which defines a list. The elements of the list are then all those in `car`s of the pairs. A more convenient way of constructing lists is using the `(list ...)` or `'(...)` constructs. Accessing individual elements of the list (`list-ref`) requires time linear in the index, while the entire list can be operated on using the **`map`** and **`for-each`** functions. There are also mutable pairs and lists, which can be manipulated using the `set-mcar!` and `set-mcdr!` functions.

Vectors Vectors represent a dense array of values, indexed using natural numbers. These are typically stored contiguously, but this is not required. Immutable vectors can be defined from their parts using the `vector-immutable` function or the `#(...)` construct. The elements of a vector can be accessed using the `vector-ref` function. Mutable vectors are created by the `vector` function, and vectors can be updated at a certain index with the `vector-set!` function.

Structures Structures define a record of named fields, which can be mutated independently. A single `(struct foo ...)` definition defines the `make-foo` constructor function, as well as a `foo?` type test. Each value can be accessed using a dedicated accessor function `foo-fieldname`. In Racket, fields are immutable by default unless they are created with the `#:mutable` option. For every mutable field `fieldname`, a setter function `set-foo-fieldname!` is generated.

Objects Objects are much like structures, except that every object is an instance of a class which defines a set of fields and methods. Moreover, classes can inherit fields and methods from other classes. While objects are not strictly immutable, they can be considered immutable if their fields are not altered after construction.

4.3.1.1 Supporting Immutable Data Structures

The biggest change necessary to support immutable data structures in blame prediction is in the check inference stage. Every compound data structure in the program must now be associated with one or more type parameters for the types contained within. For example, the type of `(vector-immutable 3 5 7)` is "vector int", while `(cons 1 "hi")` has type "pair int string". Structures and objects can be tagged similarly with their struct or class name and a mapping of field names to types (also known as *structural typing* [Madsen and Møller-Pedersen, 1989]). These type parameters can contain union types as well, for example a vector with elements of different types can be of type "vector $(\mathsf{int} \vee \mathsf{string})$". vector $(\mathsf{int} \vee \mathsf{string})$ Inferring types for applications of the various constructor functions is then a matter of inferring types for the components, and constructing a union type where necessary. For example, `(vector-immutable 1 "hello" #t)` has type "vector $(\mathsf{int} \vee \mathsf{string} \vee \mathsf{boolean})$".

In order to extract these type parameters from compound data types, we could introduce a new kind of type construct called *binders*, which bind fresh type variables to the type parameters associated with the compound data types. The various accessor functions can then use these binders to make the type parameters available to the rest of the function. For example, the type of `vector-ref`, which accesses a vector at a given index, becomes:

$$\mathsf{vector\text{-}ref} :: \Pi_{(\alpha_v,\alpha_i)}.(\mathsf{vector}\ \alpha_e \sim \alpha_v) \cdot (\mathsf{int}\ ?{=}\ \alpha_i) \cdot \alpha_e$$

The $(\mathsf{vector}\ \alpha_e \sim \alpha_v)$ binder first asserts that α_v must be a vector and then binds the element type to a fresh type variable α_e. We propose the following reduction rules for types involving binders, in this case as applied to `vector-ref`, where α_v is bound to the type of its first argument. There are three possibilities:

- The argument α_v is a "vector τ":
  τ is substituted for α_e in the type of `vector-ref`, so the return type becomes τ. The binder has bound a type variable, so it can be eliminated from the type. The remaining type is then $(\mathsf{int}\ ?{=}\ \alpha_i) \cdot \tau$. If the index parameter to `vector-ref` also has type int, the type of the whole application is further reduced to τ.

- The argument α_v is a type variable α:
  α is substituted for α_v, but nothing else happens.

- The argument α_v is a ground type γ, but not a vector:
  The binder is replaced by a conditional type $(\mathsf{vector}\ ?{=}\ \gamma)$, α_e is replaced by $\star$ in the rest of the type.

Listing 4.13 shows a function that returns the sum of the list found at a given index `i` in the vector `v`. `sum` has type $\Pi_{(\alpha_l)}.(\mathsf{list}\ \alpha_e \sim \alpha_l) \cdot (\mathsf{int}\ ?{=}\ \alpha_e) \cdot \mathsf{int}$. If we apply type

```
1 (define (sum-at v i)
2   (let ([l (vector-ref v i)])
3     (sum l)))
```

Listing 4.13: Example: taking the sum of elements in one part of a vector

inference to `sum-at` and apply the reduction rules for binders like above, the type for `sum-at` becomes:

$$\mathsf{sum\text{-}at} :: \Pi_{(\alpha_v,\alpha_i)}.(\mathsf{vector}\ \alpha_e \sim \alpha_v) \cdot (\mathsf{int}\ ?{=}\ \alpha_i) \cdot (\mathsf{list}\ \alpha_f \sim \alpha_e) \cdot (\mathsf{int}\ ?{=}\ \alpha_f) \cdot \mathsf{int}$$

while this type is quite bulky, it has the advantage of linking each test made to the primitive function which requires it.

We think that the above description is a workable approach on supporting immutable data structures in the blame prediction transformation, however further research is necessary. An alternative approach is to assign type $\star$ to all values extracted from such compound data structures, which is is far easier to implement but sacrifices precision. The type for `sum-at` is then simply

$$\mathsf{sum\text{-}at} :: \Pi_{(\alpha_v,\alpha_i)}.(\mathsf{vector}\ ?{=}\ \alpha_v) \cdot (\mathsf{int}\ ?{=}\ \alpha_i) \cdot \star$$

In either case, more research on supporting immutable data structures in blame prediction is needed. The next section will speculate on *mutable* data structures, which pose additional challenges.

4.3.1.2 Mutable Data Structures

Compound data structures like cons cells and vectors typically are first class values, so they can be passed around *by reference* and these references can be stored in other memory locations as well. In our setting, this introduces the problem of *aliasing*, where two expressions can evaluate to references to the same (location in a) data structure. In listing 4.14, we create an alias to the mutable vector `numbers` in the variable `numbers2`. Then, we mutate an element of `numbers2` (and `numbers`), and fetch the same element from `numbers`. Evaluating this example will raise a type error, as `(+ "hello" 1)` fails.

This example demonstrates two difficulties in accommodating mutable compound data structures. First, mutation can change the types of elements in a data structure, and therefore the type of the data structure as well. For example, the assignment on line 5 changes the type of `numbers2` to $\mathsf{vector}\ (\mathsf{int} \vee \mathsf{string})$. However, because `numbers2` is an alias of `numbers`, its type should also be changed, such that the `vector-ref` on line 6 returns a value of type $\mathsf{int} \vee \mathsf{string}$. Secondly, aliasing introduces an "action at a distance" effect, where mutation through one reference also affects other references

```
1 (define numbers (vector 4 8 15 16 23 42))
2 (define numbers2 numbers)
3 (define number-pair (cons numbers numbers2))
4
5 (vector-set! numbers2 3 "hello")
6 (vector-set! numbers  3 (+ (vector-ref numbers 3) 1))
```

Listing 4.14: Example of aliasing

pointing to the same data structure. For example, after the calls to vector-set!, the type of `number-pair` changes to pair $(\text{vector}\ (\text{int} \vee \text{string}))\ (\text{vector}\ (\text{int} \vee \text{string}))$

Instead, we suggest a simpler approach for blame prediction involving mutable compound data structures: mutable compound data structures have their proper types, but with the element types set to $\star$. For example, `numbers2` will have type vector $\star$. This approach is similar to the decision to have type $\star$ for mutable variables, except now type tests are generated for values *extracted* from mutable compound data structures. For example, the `vector-ref` call in listing 4.14 (line 5) would be guarded by a number? type test.

Most compound data structures share a common interface for the mutable and immutable variant. For example, `vector-ref` works on both kinds of vectors, while `vector-set!` only works on mutable vectors. Passing a mutable vector to a function which expects an immutable vector will work fine, as its type vector $\star$ matches with the binder $(\text{vector}\ \alpha_e \sim \alpha)$. In order to prevent the opposite, the type for `vector-set!` could be as follows:

$$\text{vector-set!} :: \Pi_{(\alpha_v,\alpha_i,\alpha_e)}.(\text{vector ?}= \alpha_v) \cdot (\text{mutable? } \alpha_v) \cdot (\text{int ?}= \alpha_i) \cdot \text{void}$$

where the mutable? test verifies whether α_v is a mutable vector.

Summary In this section we briefly sketched how the blame prediction transformation could be adapted to support compound data structures. For immutable data structures it suffices to encode the element types as type parameters. For mutable data structures we suggest a similar approach as to mutable variables: their element types are always $\star$, which results in type tests whenever they are accessed. However, more research is needed to evaluate whether this approach is viable or whether better approaches exist.

4.3.2 Blame Prediction and Modules

We presented blame prediction for isolated programs where *all* the program text is known in advance. However, a key property of modern programming languages is the ability to split up programs into several modules. There are several reasons for doing so; in this section we consider two chief reasons and discuss how blame prediction might accommodate them.

The first use of modules is to organize code according to its responsibilities: one module might provide functionality for logging, another for networking, and so on. The program as a whole then becomes a graph of modules which import and export functionality from and to each other. If the graph does not contain cycles, blame prediction can take a similar approach to Typed Racket [Tobin-Hochstadt and Felleisen, 2008], where the results of applying blame prediction to a module can be attached as metadata. Subsequently, blame prediction can perform a topological sort on the module graph and use metadata from already-analyzed modules. If the module import graphs *does* contain cycles, the modules which form strongly-connected components need to be analyzed together.

The second use of modules is to import common functionality from external repositories, a la Perl's `CPAN` or Ruby's `RubyGems`. These modules should be analyzed once, for example when they are uploaded to the repository; their blame prediction metadata can then again be distributed along with the code. Note that these analysis results can strongly vary as different versions of dependent modules are used. Complementary to inferred types, the blame prediction transformation could also use types written by module authors as part of their documentation.

Finally, if a module is imported whose program text is not known, blame prediction cannot be applied. The imported variables and functions are then treated as having or returning type $\star$. Unfortunately, values flowing *into* the unknown module cannot be tested.

4.3.3 Blame Prediction and Non-local Control Flow

In Scheme$_\beta$, function execution always goes to the leaves of the syntax tree and return values are the result of evaluating the expressions at these leaves. There are no mechanisms or syntactic constructs for changing the control flow. In this section we discuss three techniques that alter the control flow of a program, and how blame prediction might be extended to support them.

Return expressions In almost all programming languages, evaluating a `return` expression immediately ends the current function with its argument as return value. Code after a `return` expression is never executed, so it could be removed before — or as part of — the ANF transformation. Therefore, the blame prediction transformation we described can be applied to languages with support for explicit returns.

Exceptions A second means of altering control flow is to use exceptions. In most languages, errors such as dividing by zero or opening a nonexistent file result in exceptions, which can be caught and handled without aborting the program. When an exception is raised, execution of the current function is aborted and resumed at the nearest enclosing `catch` expression which applies. Recall that the aim of blame prediction is to prevent run-time type errors by checking the preconditions of primitive

operations ahead of time. However, this does not cover other run-time errors such as failing to open a file or dividing by zero. The difficulty stems from the interaction with checks produced from blame prediction: which source of errors gets precedence in reporting?

Listing 4.15 is a simple example which opens the `high-scores.txt` file, reads the contents, and compares it against the given `current-score` variable. It contains two sources of run-time errors: either the call to `open-input-file` (line 2) raises a "file not found" exception because the file does not exist, or the call to `>=` (line 8) raises an error because one of its operands is not a number. What happens now if the file does not exist *and* `current-score` is not a number? Should the program report the "`current-score` is not a number" error before or after the call to `read-file`?

```
1 (define (read-file filename)
2   (let ([port (open-input-file filename)])
3     (let ([contents (read port)])
4       (close-input-port port)
5       contents)))
6
7 (define (is-high-score? current-score)
8   (>= current-score (read-file "high-scores.txt")))
```

Listing 4.15: Example demonstrating the interplay between checks and exceptions

One approach is to ignore all exceptions when doing blame prediction: this means that transformed programs might report errors that would be masked by an exception happening. With this approach, the example above would abort stating that `current-score` is not a number regardless of whether the file exists.

Another approach is to make the existing type system sensitive to exceptions — as we did with side effects in this chapter — by inferring thrown exceptions for expressions and recording them in type functions. The check mobility stage could then treat exception-raising primitives as a barrier. This approach yields prediction accuracy at the cost of not propagating check expressions upwards as much as possible. Another big drawback is that this analysis must be conservative: a function passed as argument to another function could raise *any* kind of exception, similar to the side effects of higher-order functions. With this approach the example would abort with a "file not found" error without predicting blame for the `current-score` variable.

Continuations Finally, continuations are the most extreme form of exerting control over the flow of the program. Continuations record the execution state at a certain point in time into a value and later restore the execution to that point again, aborting the current execution. There are several variations of continuations (escape, one-shot, and delimited, to name a few), but we will focus here on "full-blown" continuations.

```
1 (define (f g x)
2   (check (number? x)
3     (+ x (g 2))))
```

Listing 4.16: Example of a function affected by continuations

In Scheme, invoking continuations is syntactically identical to invoking functions. This makes it impossible to impossible to reason about programs that use continuations. For example, consider the function in listing 4.16: if our language does not support continuations, the type of a call to `f` only depends on the type of `g` and `x`. If `g` is a continuation, however, the expression `(g 2)` aborts the computation of `f`. This means that not only does `(g 2)` not produce a value, but the `+` computation will never be executed. In that case, blame prediction will perform the `(number? x)` test for a computation that never happens!

One way of supporting continuations in the blame prediction transformation requires a bit of programmer cooperation: instead of allowing continuations to be invoked directly as `(k x)`, we could introduce a primitive operation `(invoke-continuation k x)` with the same semantics. Duba et al. [1991] suggest a similar approach: continuation invocations are made distinct from function applications with the `throw` keyword. With this information, the blame prediction transformation can assume all function applications are just function applications, and applications of continuations can be treated as function exit points.

Without programmer cooperation, the blame prediction transformation must conservatively assume that *every function* passed as argument is a continuation as soon as **`call-with-current-continuation`** appears anywhere in the program. Type tests are then not allowed to move beyond applications of functions passed as arguments. This severely limits the ability of the blame prediction transformation to move type tests upwards, and is thus not desirable.

4.3.4 Blame Prediction and Debug Prints

When encountering a problem in their code, many programmers insert print statements to visually determine whether the code behavior matches their expectations. For example, in listing 4.17, `generate-vector` is some kind of computation that is supposed to return a vector. This vector is then indexed using `vector-ref` at index `(calculate-index i)`. We have annotated it with the `check` expressions generated by blame prediction. The programmer's intention was to print the values of `vec`, `i` and `idx` before they are used in the call to `vector-ref`. However, if `(calculate-index i)` does not return a number, the `check` expression on line 4 will immediately (correctly!) predict blame and stop the program. Unfortunately, the program will then not execute the three `display` expressions that follow, which were supposed to help the programmer diagnose the error.

```
1 (let ([vec (generate-vector)])
2   (check (vector? vec)
3     (let ([idx (calculate-index i)])
4       (check (number? idx)
5         (display vec)
6         (display i)
7         (display idx)
8         (vector-ref vec idx)))))
```

Listing 4.17: Example of how blame prediction can interfere with debug prints

In order to prevent this scenario from happening, we could introduce a primitive operation barrier which acts as a barrier for check mobility, much like let, set! and lambda expressions. Adding such an operation would alter the above code as shown in listing 4.18. Note how the check expressions generated by the `vector-ref` call are not

```
1 (let ([vec (generate-vector)])
2   (let ([idx (calculate-index i)])
3     (display vec)
4     (display i)
5     (display idx)
6     (barrier)
7     (check (and (vector? vec) (number? idx))
8       (vector-ref vec idx))))
```

Listing 4.18: Example of how explicit barriers can prevent check mobility

propagated over the `barrier` expression on line 6. This allows the program to execute the debug prints on lines 3–5 before aborting because of the check expression on line 7.

4.4 Conclusion

In this chapter we extended the blame prediction transformation for Scheme$_\beta$ to support programs which use mutable variables.

Mutation Section 4.1 discussed how programs using mutation can be supported by the blame prediction transformation. Typically, program analyses only support mutation in dynamically typed languages if it is type-preserving, or scoped to local variables. Our analysis allows unrestricted mutation of variables. This introduces the complication that, in the worst case, every expression could modify (the type of) the value inside a variable. This conflicts with the goal of testing preconditions on variables in advance: preconditions on mutable variables can only be tested in advance if there is a guarantee that their value will not change between the test and the primitive operation which requires it. Moreover, we cannot assume anything about the type of a mutable variable, so we must assign the "any type" $\star$ to mutable variables.

We developed a conservative analysis which infers the side effect of every expression, i. e. the variables which evaluation of that expression might mutate. These expressions then form "barriers" for the check mobility and simplification stages: preconditions cannot be moved past barriers to which they apply, and nested checks may not be removed if there is a barrier inbetween. Type tests on mutable variables tend to accumulate right after expressions which potentially modify them, in the case of a set! we can eliminate such checks depending on the type of the value being assigned.

Safety proof In section 4.2 we demonstrated that the blame prediction transformation does not change the behavior of a terminating program with respect to type tests. Central in this proof are extended traces (from chapter 2). These traces not only register type tests, but now also track the types assigned to mutable variables to prove that type tests on these variables are applied to the correct value. The "terminating" qualifier is important, as a non-terminating program might raise blame prediction errors for expressions which are never reached.

Other features In Section 4.3, we discussed a number of other language features common to (dynamically typed) programming languages and how they might be supported in the blame prediction transformation as explained in this thesis. Supporting these features well would enable one to apply blame prediction to say, JavaScript, Ruby or Python, which are popular among beginning programmers. Concretely, we discussed our research roadmaps for 1) the addition of mutable and immutable compound data structures; 2) modular programs; 3) non-local control flow; 4) and the impact of blame prediction on debug print expressions.

Conclusion With the additions in this chapter, we have fully developed the blame prediction transformation and proved it correct.

In the next chapter we describe our prototype implementation in broad strokes, i. e. its architecture and techniques used for program representation and transformation. In addition, we also discuss how we found and solved two bottlenecks.

5

PROTOTYPE IMPLEMENTATION

In this chapter we describe our prototype implementation[1] of the blame prediction transformation as presented in the preceding chapters. Section 5.1 contains some general remarks about our program representation and some of the libraries used to manipulate it. Section 5.2 describes the architecture and design decisions in our implementation. In section 5.3 we discuss how we solved two important bottlenecks in our prototype implementation. Some of the code snippets presented in this chapter have been adapted for presentation and do not exactly match the code as published.

To give a small taste of the effects of the blame prediction transformation, we have transformed the `tak` program from the Gabriel benchmarks. The input program is shown in listing 5.1, the transformed program in listing 5.2. Most notable are the addition of `check` expressions and `@` expressions. Every precondition in a `check` expression is followed by a series of numbers; these are labels identified by the `@` expressions further down in the program.

5.1 General remarks

Before we explain the components of our prototype implementation, we must first make some general remarks. Our prototype implementation is written in Haskell [Peyton Jones, 2003], a statically typed purely functional programming language. In the following sections we will present small snippets of Haskell code to exemplify our architecture. We therefore assume the reader is familiar with basic Haskell syntax such as type declarations, expressions and pattern matching.

Intermediate representation Our implementation keeps an intermediate representation of the input program as an Abstract Syntax Tree (AST). Every expression carries

[1] This code is available via DOI `doi:10.5281/zenodo.30614` [Harnie, 2015], which is generated from a Git repository hosted on GitHub (`https://github.com/Botje/crystal`). The `eval-phd-final` release was used to perform our experiments.

```
(define (tak x y z)
  (if (not (< y x))
      z
      (tak (tak (- x 1) y z)
           (tak (- y 1) z x)
           (tak (- z 1) x y))))

(let ([input (with-input-from-file "input.txt" read)])
  (time
   (let loop ([n 500] [v 0])
     (if (zero? n)
         v
         (loop (- n 1) (tak 18 12 (if input 6 0)))))))
```

Listing 5.1: Input program: `tak`

```
(letrec ([tak (lambda (x y z)
                (check (and (number? y 1004 1020) (number? x 1004 1012))
                       (if (not (@ 1004 < y x))
                           z
                           (check (number? z 1028)
                            (tak (tak (@ 1012 - x 1) y z)
                                 (tak (@ 1020 - y 1) z x)
                                 (tak (@ 1028 - z 1) x y))))))])
        (let ([input (with-input-from-file "input.txt" read)])
         (time (letrec ([loop (lambda (n v)
                                (if (zero? n)
                                    v
                                    (check (number? n 1050)
                                      (loop (@ 1050 - n 1)
                                            (tak 18 12 (if input 6 0))))))])
                           (loop 500 0)))))
```

Listing 5.2: Output from the blame prediction transformation

an annotation such as its source position or type. This representation was inspired by that of the Glasgow Haskell Compiler (GHC), as described in [Brown and Wilson, 2012, Chapter 5]. There are two benefits to this representation: first, every node in the tree must have an annotation of the same type, forcing the code which modifies it to consider every case of the AST when changing the type of the annotation. This enables the compiler to raise a type error if a portion of the AST is not modified. The second benefit is that the annotations are readily available while traversing the tree, as opposed to looking them up in an external data structure and dealing with missing annotations. We will describe the various annotations attached to the AST as they are introduced.

```
1  type Ident = String
2  data Expr a = Expr a (InExpr (Expr a))
3  data InExpr e = Lit    LitVal
4                | Ref    Ident
5                | Appl   e [e]
6                | If     e e e
7                | Let    [(Ident, e)] e
8                | LetRec [(Ident, e)] e
9                | Lambda [Ident] (Maybe Ident) e
10               | Begin  [e]
```

Listing 5.3: Abstract Syntax Tree representation

The Haskell definition of our AST is shown in listing 5.3. Concretely, the AST is of type `Expr a`, where `a` represents the type of the annotation(s) attached to every expression. The constructors of the `InExpr e` type represent (in order): literal expressions, variable references, function applications, if-expressions, let-expressions, letrec-expressions, lambda-expressions, and begin-expressions. All expressions correspond directly with the definitions in figure 2.2 (and letrec in figure 3.1), except for lambda expressions.

For our prototype implementation we implement a superset of Scheme$_\beta$ (but still a subset of full R5RS Scheme). Most notably, the implementation *does* support all of the data types defined in R5RS, almost all of the library functions, variadic functions (see below), and special forms for control such as `cond` and **`do`**. Our implementation does *not* support syntax macros, nor does it support continuations or `dynamic-wind`.

Variadic functions In the previous chapters we only considered lambda expressions with a fixed number of parameters. As we discussed in section 2.8, the Scheme programming language also supports functions with a variable number of parameters. We opted to add support for this feature in our implementation, as a number of the benchmarks we evaluate in chapter 6 use variadic functions.

The **`Maybe`** argument to the `Lambda` constructor represents the name of the optional variable to capture extra arguments. For example, the expression

```
(lambda (a b . rest) body)
```

is represented by the term (a1 and a2 are annotations)

```
Expr a1 (Lambda ["a", "b"] (Just "rest") (Expr a2 body))
```

Mutation The AST as presented above does not have an explicit data constructor for assignment. Instead, assignment is modeled as an application of the `set!` "function". This enables our implementation to share a large portion of the code for variable mutation with the code for function application. In addition, it simplifies the code for check mobility significantly.

Uniplate: Generic AST traversals Our implementation performs many different transformations on the AST, but typically only on specific parts of the program. In regular Haskell, we would have to write lots of boilerplate code to perform the transformation everywhere in the program and manually recombine the results. Instead, we have implemented almost all of the transformations in this chapter using Uniplate [Mitchell and Runciman, 2007a]. This library contains functions such as `universe`, which automatically traverses a data structure and returns all values of the specified type. For example, it enables us to find the free variables of an expression with a one-liner:

```
freeVariables :: Expr a → Set Ident
freeVariables expr = fromList [ r | Ref r ← universe expr ]
```

In addition, the Uniplate library contains `transform`, which performs a bottom-up traversal of a data structure and applies a function to every value of a certain type. We use this function to selectively perform modifications by using pattern matching. For example, the snippet in listing 5.4 identifies let expressions which bind one variable and inlines their definition into the let body if they are only used once. The interesting case here is on line 5, when both the pattern match on line 3 *and* the extra condition is satisfied. If the expression is a let expression which either binds more than one variable or the bound variable is used more than once, the case on line 6 returns the original expression If the expression is not a let expression, it is immediately returned without modifications.

```
1 inlineSingleUse :: Expr a → Expr a
2 inlineSingleUse expr = transform go expr
3   where go orig@(Expr a (Let bnds bod)) =
4           case bnds of
5             [(id, e)] | 1 == numberOfUses id bod → inline id e bod
6             otherwise → orig
7         go orig = orig
```

Listing 5.4: Example of `transform`: inlining variables which are used only once

```
getEffect :: Expr (Label, Type, Effect) → Effect
getEffect (Expr (_, _, ef) _) = ef

setEffect :: Effect → Expr (Label, Type, Effect)
                    → Expr (Label, Type, Effect)
setEffect ef (Expr (l, t, _) ie) = Expr (l, t, ef) ie

modifyEffect :: (Effect → Effect) → Expr (Label, Type, Effect)
                                  → Expr (Label, Type, Effect)
modifyEffect f (Expr (l, t, ef) ie) = Expr (l, t, f ef) ie
```

Listing 5.5: Boilerplate code for getting, setting and modifying the effect of an annotation of an expression

Lenses: Composable structure introspection and modification Finally, our implementation often needs to inspect and modify values which are deeply nested in another data structure. For example, say that we have an expression of type `Expr (Label, Type, Effect)`. In order to get, set and modify the `Effect` part of this value, we must write the three functions in listing 5.5. These functions cannot be reused for expressions with different annotations such as `Expr (Label, Effect)`.

In order to avoid writing several such functions, our implementation makes use of the lens library by Edward Kmett. This library consists of a large variety of small combinators called *lenses* which focus on sub-parts of a compound value. Several lenses can be composed to focus even further. The focused value can then be read or mutated easily. We can reimplement `getEffect` and `setEffect` as follows:

```
-- Given ann :: Simple Lens (Expr a) a
annEffect = ann . _3
getEffect      expr = expr ^. annEffect
setEffect   ef expr = expr & annEffect .~ ef
modifyEffect f expr = expr & annEffect %~ f
```

Listing 5.6: Lens code for getting, setting and modifying the effect of an annotation of an expression

As this example demonstrates, lenses can be composed using the `.` (dot) operator. Once focused, the value can be accessed using the `^.` operator, mutated using the `.~` operator, and transformed using the `%~` operator. In our implementation we do not write the helper functions, but just use the lenses and their combinators directly.

5.2 Architecture

The overall architecture of our implementation is a pipeline (shown in listing 5.7), where every stage performs a specific part of the blame prediction transformation. These stages are monadic actions in the `Step` monad, which combines two different monads. The first is `ReaderT Config`, which makes the command-line options available

```
1 type Step a = WriterT [StepResult] (ReaderT Config IO) a
2 type StepResult = (String, Text)
3
4 pipeline = transformC >=> infer >=> addChecks >=> postprocess
```

Listing 5.7: Overview of the blame prediction pipeline

to the stages, and the `WriterT [StepResult]` enables steps to report arbitrary information which is printed either during program execution, or when it exits. The four parts of `pipeline` are as follows:

1. `transformC` (section 5.2.2), which establishes some invariants and transforms the code to Administrative Normal Form (ANF);
2. `infer` (section 5.2.3), which infers types for all expressions in the program;
3. `addChecks` (sections 5.2.4 and 5.2.5), which converts the conditional types into `check` annotations, performs `check` mobility, and renders the `check` annotations into the code.
4. `postprocess` (section 5.2.6), which largely undoes the ANF transformation. This part is not integral to the transformation, but it aids significantly in manual inspection of the resulting program.

We start by explaining how our implementation parses programs into their in-memory representation.

5.2.1 Parsing

We use the well-known Parsec [Leijen and Meijer, 2002] library for parsing Scheme$_\beta$ programs, which enables a declarative, top-down style of writing parsers. For example, listing 5.8 shows a vertical slice of the parser code involved in processing a `lambda` expression. The `parameters` function is responsible for processing the parameter list of the `lambda` expression. The subset of Scheme supported by our prototype implementation allows three forms of parameter bindings:

1. `(lambda (a b) ...)`, which expects two parameters `a` and `b`;
2. `(lambda (c . d) ...)`, which expects *at least* one parameter `c` and other arguments are bound to a list in `d`; and
3. `(lambda e ...)`, which simply binds its arguments to a list in `e`.

The `lambda` parser is invoked by the `sexp` parser after it encounters a `lambda` keyword in parentheses. This parser uses the `parameters` and `letBody` parsers to parse the `lambda`'s parameters and body, respectively. Finally, the `makeExpr` function generates a label for the `lambda` expression.

The `parameters` parser works as follows: if it can consume an open parenthesis character `"("`, the code reads one or more `identifiers` (line 1), optionally followed by a dot and another identifier (line 2). If such an identifier is not found (case 1), `rest` is **`Nothing`**. If an identifier is found (case 2), `rest` is bound to **`Just`** the identifier. Line 3 returns a tuple of a list of static parameters and the optional extra identifier. If there is no open parenthesis character at the start of `parameters`, line 6 attempts to parse an identifier (case 3).

```
1  parameters =     parens (do vars ← many1 ident
2                            rest ← optionMaybe (symbol "." >> ident)
3                            return (vars, rest))
4               <|> do i ← ident
5                      return ([], Just i)
6
7  lambda = do (fixed, rest) ← parameters
8              body ← letBody
9              makeExpr $ Lambda fixed rest body
10
11 sexp = parens ((reserved "lambda" >> lambda) <|> ...)
12
13 makeExpr ie = do s ← getState
14                  putState (succ s)
15                  return $ Expr s ie
```

Listing 5.8: Parsec code for a lambda expression

The result of the parsing stage is an AST where every expression is identified by a unique label, i. e. `Expr Label`. Our implementation keeps a counter for these labels as the "user state type". If there was a parse- or logic error, this is reported and our implementation stops further processing.

5.2.2 Preprocessing and ANF Transformation

After parsing, the program consists of a number of declarations and a number of expressions. These are combined into one letrec expression, such that the rest of the transformation pipeline can work on a single expression. This expression undergoes a number of transformations, as described below.

First, alpha renaming is performed. This is a common technique used in program transformations, which ensures that variable bindings are not shadowed by others. This makes identification of free variables and function inlining trivial.

Next, our implementation splits the binding groups of letrec expressions in strongly-connected components (SCCs). This technique is described in [Peyton Jones, 1987, Section 6.2.8] and Waddell et al. [2005]. The main result of this technique is that every letrec expression binds exactly one (group of) mutually recursive function(s). This makes the inference of side effects of any function as simple as taking the union of

the side effects of all functions in the binding group. In addition, the reduction of infinite types to finite types (by Solve) becomes faster. Any variable or non-recursive function definitions in a letrec binding group become ordinary let expressions.

Next, our implementation eliminates let expressions with an empty binding group. Also, let expressions which bind multiple variables are split into nested let expressions. This has the effect of making the order of evaluation of expressions in a let expression explicit, and ensuring every let expression binds exactly one variable, as in the type rules.

Next, the **and**, **or**, **do**, **cond**, **when**, and **unless** Scheme special forms are transformed into regular Scheme expressions according to the expansions in R5RS [Abelson et al., 1998]. R5RS also defines some functions which compose successive `car` and `cdr` calls. Our implementation transforms such functions into their constituent parts, so `(cadr x)` becomes `(car (cdr x))`.

Finally, the program is transformed to administrative normal form (ANF; [Sabry and Felleisen, 1993]). This transformation, as explained in chapter 2, makes the order of evaluation explicit by enforcing one simple rule: arguments to function applications or conditional expressions may only be simple expressions such as variable references or constants. The ANF transformation then ensures that nested function applications are bound to temporary variables before they are used.

Listing 5.9 describes a part of the ANF transformation responsible for generating **let** expressions for these temporary variables.. The entire transformation runs in a `State` monad which keeps track of allocated labels. Key to the transformation is a continuation function `k`, which represents a "hole" that must be filled in with an expression. In order to transform a function application, first the function and its arguments must be transformed. If any of these is again a function application, the `float` function is applied, which generates code to assign the nested function application to a variable. It does this by generating a variable name (line 4), transforming the nested function application (line 7), calling the continuation `k` with a reference to this new variable (line 8), and finally constructing the let expression (line 9).

```
1 float :: Expr Label -> (Expr Label -> State Int (Expr Label))
2                     -> State Int (Expr Label)
3 float expr k =
4   do var <- next "tmp-"
5      labRef <- nextSeq
6      labLet <- nextSeq
7      expr_ <- anf expr return
8      rest <- k (Expr labRef $ Ref var)
9      return $ Expr labLet $ Let [(var, expr_)] rest
```

Listing 5.9: ANF transformation code

At the end of this stage, the program is still an expression of type `Expr Label`, but with a number of invariants:

1. Every bound variable has a unique name.

2. Every letrec expression binds one or more mutually recursive functions.
3. Every let expression binds exactly one variable.
4. Function applications and conditional expressions have only simple expressions (such as variable references and constants) as arguments.

5.2.3 Check Inference

After parsing the program into an expression and establishing the aforementioned invariants above, we turn to check inference. Recall from chapter 2 that check inference consists of type inference and check introduction. In order to perform type inference, we must walk the program tree and, using the rules outlined in chapters 2 to 4, assign a type and an effect (the variables it can mutate) to every expression. The process itself is structured around the `Infer` monad, defined as follows:

```
type Infer a = ReaderT (Map Ident Type) (State TVar) a
```

This monad combines a reader monad with a state monad: the first is used to propagate variable bindings downwards into the program, while the second is used to allocate fresh type variables.

The type rules themselves are implemented by a function with the following type:

```
Expr Label -> Infer (Expr (TLabel, Type, Effect))
```

There is a one-to-one correspondence between the type rules and the various clauses of this function. The output is a program annotated with labels (real or synthetic), types, and effects.

For example, listing 5.10 shows how the type is inferred for a lambda expression. For brevity, we only show the code for fixed-length functions. Line 2 generates fresh type variables for all of the function arguments, and lines 3–4 construct a new typing environment `env'` which maps the argument names to these type variables. Line 5 performs check inference of the body in this new environment. Finally, lines 6–9 construct a new `Lambda` expression with the inferred type `t_lambda` and an empty effect. Note that any effects produced by the body are recorded in the `TFun` constructor. For comparison, figure 4.1 defined the type rule for lambda expressions as follows:

$$\frac{\Gamma, x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash_E e : \tau \,!\, \overline{x} \qquad \alpha_1, \ldots, \alpha_n \text{ fresh}}{\Gamma \vdash_E (\text{lambda } (x_1 \ldots x_n)\ e) : \overline{\prod}^{\overline{x}}_{(\alpha_1 \ldots \alpha_n)} . \tau \,!\, \emptyset} \quad (\text{TE-LAMBDA})$$

The support for recursion in our prototype implementation is implemented by a call to `solveLetrec` in the check inference clause for letrec expressions. It receives the labels and (infinite) types of the functions defined in the letrec binding group. After performing the widening as described in section 3.2.3, it repeatedly applies the functionals to approximations and reduces them to normal form until the approximations

stop changing. The resulting finite types are returned to the inference process, which uses them in the inference of the letrec body.

```
infer (Expr l (Lambda ids Nothing bod)) =
  do a_ids <- mapM (const freshTVar) ids
     let t_ids = map (\v -> (LVar v, TVar v, emptyEffect) a_ids
     env'  <- asks (extendMany ids t_ids)
     e_bod <- local (const env') (infer bod)
     let (t_bod, ef_bod) = (e_bod ^. ann._2, e_bod ^. ann._3)
     let t_lambda = TFun a_ids ef_bod $ simplify t_bod
     return $ Expr (LSource l, t_lambda, emptyEffect)
                   (Lambda ids Nothing e_bod)
```

Listing 5.10: Check inference for lambda expressions

```
data Type = TAny -- ⋆ in this dissertation
          | TError
          | TVar TVar
          | TInt | TString | TBool | TSymbol | TVoid | TVec | TPair | TList
          | TNull | TChar | TPort | THash
          | Tor [Type]
          | TFun [TVar] Effect Type
          | TVarFun VarFun
          | TIf (TLabel, TLabel) Type Type Type -- (blame label, cause label)
          | TAppl Type [(TLabel, Type)]
          | TChain Type TVar TLabel Type

type Effect = Set Ident
data VarFun = VarFun Ident TLabel ([(TLabel, Type)] -> TLabel -> Type)

data Check = Cnone
           | Cand [Check]
           | Cor [Check]
           | Check [TLabel] Type (Either LitVal Ident)
```

Listing 5.11: Definition of types

For completeness, the definition of the `Type` and `Check` data types is shown in listing 5.11. Most of the data constructors of `Type` correspond directly to the types declared earlier[2]. One exception is the `TVarFun` constructor, which represents the type of variadic built-in Scheme$_\beta$ (and Scheme) functions. For example, the + function can take any number of arguments and returns an int, but it also requires that each argument is an int. The corresponding `VarFun` value contains a Haskell function which generates the correct type given the types and labels for its arguments (the labels are required

[2]As in chapter 2, we use `TInt` instead of `TNumber`. The run-time type test generated for `TInt` is number?.

to correctly ascribe blame). Another non-standard data constructor is `TChain`, which represents immediately-applied type functions.

The `Check` data type is used to represent preconditions of `check` expressions in annotations. The `Cnone` data constructor is the "no-op" precondition `#t`. `Cand` and `Cor` represent conjunctions and disjunctions of preconditions, respectively. Finally, the fields of the `Check` data constructor are, in order: a list of labels to blame if the precondition is not met, a concrete type τ_t to test against, and either a literal value or a variable which is to be tested.

Finally, after inferring types and effects for every expression in the program, `check` introduction transforms the types into predicates for `check` expressions. Converting types into `check` predicates is done exactly as specified in section 2.3, i. e. it only inserts `check`s in the annotations for `Appl` expressions and performs `check` simplification afterward. The final type after `check` introduction becomes `Expr (TLabel, Check, Type, Effect)`.

5.2.4 Check Mobility

The `check` mobility stage is implemented as a bottom-up transformation which runs over the program. For every expression, it analyses the `Check` annotations of its subexpressions and alters them according to the rules laid out in sections 2.4 and 4.1.2. Listing 5.12 shows an extract of this functionality, with clauses for function applications, variable references and conditional expressions.

```
1  move :: Expr CheckedLabel -> Expr CheckedLabel
2  move simple@(Expr lctef@(l, c, t, ef) e) =
3    case e of
4         Appl op args     -> Expr lctef $ Appl (move op) (map move args)
5         Ref r            -> simple
6         If cond cons alt ->
7           let (cond', cons', alt') = (move cond, move cons, move alt)
8               (cons_c, alt_c)      = (cons' ^. annCheck, alt' ^. annCheck)
9               c'                   = simplifyC $ Cor [cons_c, alt_c]
10              lctef'               = lctef & annCheck .~ c'
11          in Expr lctef' $ If cond' cons' alt'
12        {- other cases -}
```

Listing 5.12: Example of check mobility

For comparison, figure 4.2 defined `check` mobility for these constructs as follows:

$$\frac{\mathsf{s}_i \to \mathsf{s}_i' \uparrow \#\mathsf{t} \qquad \forall i \in 0 \ldots n}{(\mathsf{check}\ \mathsf{p}\ (\mathsf{s}_0\ \mathsf{s}_1 \ldots \mathsf{s}_n)) \to (\mathsf{s}_0'\ \mathsf{s}_1' \ldots \mathsf{s}_n') \uparrow \mathsf{p}} \quad (\text{FE-APPLY}) \qquad \mathsf{x} \to \mathsf{x} \uparrow \#\mathsf{t} \quad (\text{FE-VAR})$$

$$\frac{\mathsf{s} \to \mathsf{s}' \uparrow \#\mathsf{t} \qquad \mathsf{e}_1 \to \mathsf{e}_1' \uparrow \mathsf{p}_1 \qquad \mathsf{e}_2 \to \mathsf{e}_2' \uparrow \mathsf{p}_2}{(\mathsf{if}\ \mathsf{s}\ \mathsf{e}_1\ \mathsf{e}_2) \to (\mathsf{if}\ \mathsf{s}'\ (\mathsf{check}\ \mathsf{p}_1\ \mathsf{e}_1')\ (\mathsf{check}\ \mathsf{p}_2\ \mathsf{e}_2')) \uparrow \mathsf{p}_1 \vee \mathsf{p}_2} \quad (\text{FE-IF})$$

The clause for function applications simply performs mobility on its simple sub-expressions, which might still contain lambda expressions. The clause for variable references returns itself, as such a reference is never annotated with checks.

The clause for conditional expressions is more elaborate. It performs a number of steps: first, check mobility is applied to all three sub-expressions (line 7). Next, the check annotations on the branches of the conditional expressions are extracted using the `^.` lens operator (line 8). The checks are combined with a `Cor` constructor and simplified on line 9. Finally, the `.~` lens operator replaces (only) the `Check` part of the annotation with the composed checks (lines 10–11).

Although we treat begin expressions as syntactic sugar for nested let expressions, for performance reasons we treat them in a separate clause. In short, the predicates are "bubbled up" through the begin expression, where every expression can both contribute new predicates and block some predicates from propagating. The check predicates which arrive at the top of a begin expression are moved into the annotation of the *entire* begin expression. For ease of implementation, predicates are not reduced to their minimal form at this stage, meaning that the check annotation for the n^{th} expression in a begin expression consists of the union of all propagated predicates below it. These will be cleaned up in the check simplification step, described next.

5.2.5 Check Simplification

In this stage, the remaining check expressions are analysed and any redundant predicates or expressions are removed. It consists of two separate transformations: one implements and–check and check–check simplifications as described in section 2.5, and the other implements set–check simplification as specified in section 4.1.3.

The first transformation is very difficult to implement in a purely functional setting due to the nature of check–check elimination. This specific simplification not only eliminates predicates in nested check expressions where the outer check already verified the predicates, but it also merges the blame labels of the inner check into the outer check. In essence, the simplification needs to proceed from the top down, while accumulating labels into the check expressions at the top. To further complicate things, this simplification may only be performed *if* there is no intermediate mutation of the variables being tested, as mutation can invalidate the results of an earlier type test (recall section 4.1.3).

We have opted to implement this transformation as follows: first, our implementation walks the AST and constructs a `checksMap :: Map TLabel (Check, Type, Effect)`, which maps each expression to its annotation. In addition to this, a second map `varMap :: Map Ident (TLabel, Type)` keeps track of the label *where* a variable was last successfully checked, and which type it was tested against. When a nested check expression is encountered, the variables involved in the predicates are looked up in `varMap`. For every variable — where the type being tested for matches that in `varMap` — the predicate is removed from the check and added to the one in `checksMap` where the

```
1  eliminateRedundantChecks ast = do
2    let exprs = universe ast
3    let startChecks = M.fromList [ (l,(c,t,ef)) | Expr (l,c,t,ef) _ <- exprs ]
4    let (augChecks, _) = execState (walk expr) (startChecks, M.empty)
5    let finalChecks = M.map (_1 %~ (simplifyC . mergeSameChecks)) augChecks
6    return $ updateChecks finalChecks ast
7
8  walk :: Expr (TLabel, Check, Type, Effect)
9            -> State (ChecksMap, VarMap) (Expr (TLabel, Check, Type, Effect))
10 walk (Expr (l,c,t,ef) ie) = do
11   c' <- removeAlreadyChecked l c
12   updateVarMap l c'
13   case ie of
14     (Appl f args) -> do
15       mapM_ walk (f:args)
16       forM_ (effectToList ef) $ \id -> modify (varMap %~ M.delete id)
17     (Lambda ids r bod) -> do
18       (cm, vm) <- get
19       forM_ allSet $ \id -> modify (varMap %~ M.delete id)
20       case r of
21         Nothing -> return ()
22         Just x  -> modify (varMap %~ M.insert x (l, TList))
23       walk bod
24       modify (varMap .~ vm) -- restore varMap
25       -- other cases
```

Listing 5.13: Simplified overview of check simplification

predicate was last checked. However, if an expression is encountered which mutates variables, the variables are removed from `varMap`, as their type could have changed. Finally, the altered checks in `checksMap` are applied to the AST again.

Because the full code is too complex to present (it is almost one hundred lines of Haskell code), we have chosen to show a very simplified overview of this algorithm in listing 5.13. The `walk` function implements check–check simplification as a monadic action in the `State (ChecksMap, VarMap)` monad. We have defined associated lenses `checksMap` and `varMap` which access the correct part of the state tuple. The `removeAlreadyChecked` function on line 11 performs three tasks at once: it removes predicates from `c` if they are already satisfied according to `varMap`, it re-adds those predicates to the outer check expressions in `checksMap`, and finally returns a simplified check predicate without the duplicate predicates. `updateVarMap` on the next line examines the check predicate and adds or updates the types of any predicates which must have succeeded into `varMap`. Finally, the remainder of the `walk` function walks the actual expression, updating `varMap` where necessary.

The clause for `Lambda` expressions (lines 17–24) processes the body without assuming any knowledge about mutable variables (line 19) and restores `varMap` after processing

the function. Merging redundant predicates in the same check predicate (and–check simplification) is done by the call to `mergeSameChecks` on line 5.

The second transformation is the set–check simplification, which is *much* simpler. It walks the sub-expressions of begin expressions, searching for a set! expression followed by an expression with check predicates which test the mutated variable. If the predicates are satisfied by the type of the value passed to set!, they are replaced with `Cnone` (#t) and the check predicate is simplified.

After check simplification has finished, the check expressions are reified in the program. The expressions that are referred to by the check expressions are identified with an `@` marker. At the same time, the type of the AST is reduced to `Expr TLabel`, dropping all annotations.

5.2.6 Postprocessing

In the beginning of this chapter we described the preprocessing stage, which enforces certain invariants in the program. This stage undoes most of the structural changes made to the program, in order to produce a program which is as close as possible to the input program. The aim of this postprocessing is to make it easier to manually inspect the resulting program.

The first transformation undoes the ANF transformation by inlining any temporary variables it generated. Recall that the ANF transformation makes the order of evaluation explicit by binding the results of nested function applications to temporary variables. This transformation looks for let expressions which bind such a temporary variable and replaces them with their body, with the variable replaced by the bound expression. There is one important restriction however: the variable may not be used in a check precondition, as otherwise this will cause an unbound variable to be referenced. This step can eliminate almost all temporary variables introduced by the ANF transformation.

The very final transformation step in our implementation undoes the conversion made after parsing: the nested letrec expressions are turned into define statements for (groups of) functions. Similarly, any let expressions become define statements too. The expression at the bottom of the program is the first non-let, non-letrec expression encountered while traversing the program.

In order to actually produce output, our implementation uses a pretty-printer in the style of Hughes [1995] and Wadler [2003]. It could just emit the S-expressions that make up the program on a few lines without any significant whitespace, but the addition of a pretty printer makes it easier to manually inspect the program for debugging.

```
1 instance Uniplate Check where
2   uniplate (Cnone)             = plate Cnone
3   uniplate (Cand cs)           = plate Cand ||* cs
4   uniplate (Cor cs)            = plate Cor ||* cs
5   uniplate (Check labs typ lv) = plate Check |- labs |- typ |- lv
```

Listing 5.14: Uniplate instance for the `Check` datatype

5.3 Optimizations

During the development of our prototype implementation, we paid much attention to correctness, but only little to the memory and CPU usage of our code. When some of the benchmark programs (from chapter 6) took over a minute to process, we started profiling our prototype application and eliminating the biggest bottlenecks. In this section, we discuss these bottlenecks and show how we removed them.

Uniplate The Uniplate library, as mentioned earlier in this chapter, was key in rapidly implementing transformation passes over our AST. However, as the number of passes grew, our run-time profile showed more and more time was being spent inside Uniplate. Rewriting some of these passes naïvely by manually traversing the program AST resulted in a program speedup, but this would result in significant code bloat if done for all passes.

Explaining the solution requires some background knowledge about the Uniplate library and generic programming in Haskell first. In order to function correctly, Uniplate needs to know — given a data constructor for a type — which of its fields can contain values of that type. For example, when using the `universe` function to look for values of type `Check`, the `Cnone` constructor contains none, while the `Cand` constructor contains a list of `Check` values. In the absence of user annotations, Uniplate can make use of the `Typeable` and `Data` subclasses to locate these fields, explained in [Lämmel and Peyton Jones, 2003, 2005]. However, the decision of which fields to examine must be made at run-time, which leads to a lot of overhead. We implemented the necessary Uniplate instances by hand and saw an order of magnitude speedup.

For example, listing 5.14 shows the required instance for the `Check` data type (defined in listing 5.11). In essence, each clause of the `uniplate` function must return a tuple containing 1) a list of all the `Check` values contained in its argument, and 2) a function which reconstructs the value given a new identically-sized list of `Check` values. The `plate` function constructs such a tuple and the `||*` and `|-` operators extend this tuple for every field of the various data constructors. The `||*` operator signals that the field to the right contains a list of the desired `Check` values. The `|-` operator used in the `Check` constructor indicates there will `never` be a value of the right type in that field. In case of the `Check` constructor, those fields are a list of labels, a ground type, and a variable name, neither of which can contain a value of type `Check`.

Simplification and sharing Another source of excessive memory and CPU usage was our `simplify` function for types. When types are generated or constructed from other types, we use the `simplify` function to both reduce the size of the type and canonicalize it. For example, our first implementation simplified union types as shown in listing 5.15. Aside from the use of **`nub`** — which is $\mathcal{O}(n^2)$ but n is always very small — there

```
1 simplify (Tor ts) = let l = nub $ map simplify ts
2                     in case l of
3                          [t] -> t
4                          ts' -> Tor ts'
```

Listing 5.15: Naïve simplification for union types

is another problem with this implementation: every invocation of this `simplify` function will return a completely reconstructed value. This destroys sharing (i. e. where multiple references to the same variable reference the same address), and forces our implementation to spend a lot of time and memory in constructing identical types.

Our solution was to make the `simplify` function more complex in order to keep track of which invocations of `simplify` *actually* performed a simplification. If no simplification was performed, it returns the original input, preserving sharing and avoiding unnecessary allocations. Listing 5.16 presents the modification-aware `simplify` function. The `simplify` function is now a wrapper around the `simp` function, which returns **`Nothing`** if no simplification was made and **`Just`** the simplified type otherwise.

Lines 5–12 show how simplification of union types is done in a modification-aware way. The value of `ts'` (line 5) is a list of **`Maybe`** `Type` values, where a **`Just`** constructor indicates simplification happened, and **`Nothing`** indicates no simplification happened. This list is **`zip`**ped together (line 7) with the input types, producing a list of either the modified type if simplification happened, or the original type if not. Finally, if any simplification happened in the first step (`anySimplified`) or the new list of types is shorter than the input list (`anyDuplicate`) **`Just`** a new type is returned, otherwise `simp` returns **`Nothing`**.

The addition of this modification-aware simplification trades complication and a bit of extra CPU usage for reduced memory use and no more useless value reconstructions.

5.4 Conclusion

In this chapter we described our prototype implementation of the blame prediction transformation. The architecture was built around the three stages outlined in chapter 2, namely:

Parsing This stage reads in a file with S-expressions and constructs an Abstract Syntax Tree (AST) with labels.

```
simplify :: Type -> Type
simplify t = maybe t id $ simp t

simp :: Type -> Maybe Type
simp (Tor ts) = let ts'           = map simp ts
                    anySimplified = any isJust ts'
                    ts''          = zipWith (\t m -> maybe t id m) ts ts'
                    ts'''         = nub ts''
                    anyDuplicate  = length ts''' < length ts''
                in if anySimplified || anyDuplicate
                     then Just (Tor ts''')
                     else Nothing
```

Listing 5.16: Modification-aware simplification for union types

Preprocessing and ANF Transformation This stage does macro expansion and applies the ANF transformation [Sabry and Felleisen, 1993] to the program to make control flow explicit and ensure that there are no nested function applications.

Check Inference This stage performs check inference as outlined in sections 2.2, 3.2.1 and 4.1.1: at the end, every expression in the program has a type and effect associated with it.

Check Mobility This stage transforms the conditional types in the program to explicit check annotations and performs check mobility as defined in sections 2.4, 3.2.1 and 4.1.2.

Check Simplification This stage attempts to eliminate duplicate check expressions. It uses the rules in section 2.5 and the extra rule from section 4.1.3.

Postprocessing This stage undoes the ANF transformation where possible, resulting in output close to the input program. Additionally, it inserts labels for the check expressions.

Finally, we discussed two key optimizations we performed to make our prototype fast enough for interactive development, namely hand-written annotations for the Uniplate library and modification-aware simplification in order to preserve sharing and prevent unnecessary type reconstruction.

In the next chapter we evaluate our prototype implementation of the blame prediction transformation against programs from the well-known Gabriel benchmarks, as well as from the Computer Language Benchmark Game (CLBG). We evaluate these programs according to two criteria: the first is a measure of how well the blame prediction transformation can move type tests upwards. The second criterium is the reduction in type tests, which is a by-product of our type system. Finally, we apply techniques from mutation testing to generate programs which contain type errors and

subsequently show how the blame prediction transformation is able to identify these errors significantly earlier.

6

EVALUATION

In the previous chapters we defined the blame prediction transformation, extended it to support recursion and mutation, and described our prototype implementation. The goal of the blame prediction transformation is to extract type tests from primitive functions and move them upwards as far as possible, in order to reduce the amount of code the programmer needs to inspect and the time spent waiting for the program to report a type error. In this chapter we will evaluate the blame prediction transformation according to two metrics: the distance between `check` expressions and the primitive operations they guard, and the reduction in type tests. Both metrics can be measured both in terms of the program text and run-time operations.

6.1 Metrics

In this section we define two metrics. The first metric measures the distance between a `check` expression and its corresponding primitive operation. The second metric measures the number of type tests the blame prediction transformation is able to eliminate, which attacks on the "long time to crash" problem. We will evaluate each metric both in terms of the program text and the run-time behavior.

To give an intuition for the effect on run-time behavior, consider the code in listing 6.1: a wrapper around a `copy-file` function which prints the number of bytes copied. Unfortunately, the programmer made a mistake and applied the `length` function instead of the `file-size` function. Upon running the program, it will spend some time copying the file (maybe a long time if the file is big), only to report an error on the call to `length`, as `source` is not a list. By contrast, the blame prediction transformation will insert a `check` expression *before* the call to `copy-file`. In terms of the program text, moving the test on the precondition of `length` over one statement is not a huge win. In terms of run-time behavior however, the program immediately predicts blame, so the programmer does not need to wait for the program to copy the file first.

```
1 (define (copy-file-and-report source destination)
2   (copy-file source destination)
3   (display "Copied " (length source) " bytes"))
```

Listing 6.1: Example where the static prediction is small, but run-time prediction can be large

Metric 1: check–use **distance**
At the start of this dissertation we postulated the hypothesis that time spent debugging a type error in a dynamically typed language is correlated with the distance between the place where the value which causes the error is *computed* and where it is *used* in a primitive operation. Consider the program in listing 6.2: when the programmer runs it as-is, the program produces an error such as `line 4: +: expected number, given "0"`. The cause of this error is the string `"0"` passed as the second argument to the `triangle` function. To discover this error, the programmer must search for the argument which is faulty (`res`), and trace back through the program to find out where `res` received its value. In listing 6.2, the point where the variable `res` is passed as argument to `+` is marked with `use`, and the place where it is defined is marked with `def`. The `use` coincides with `check`, the place where the preconditions of the function are checked.

```
1 (define (triangle n res)  def
2   (if (= n 0)
3       res
4       (triangle (- n 1) (+ n res))))  check ····· use
5
6 (triangle 10 "0")
```

Listing 6.2: Example of debugging errors in dynamically typed languages

As described in the previous chapters, the goal of the blame prediction transformation is to insert check expressions which perform type tests on variables before they are actually used in primitive operations, reducing the distance the programmer needs to cover. Listing 6.3 shows the program from listing 6.2 after the blame prediction transformation has inserted check expressions. Because of the check expression on line 6, the program can already predict blame for the `res` variable, which is only used on line 8. When blame is predicted at `check`, the programmer receives a report stating which precondition was violated (`number? res`), the `def` and `use` points for the variable(s) involved, along with bindings for other local variables.

As can be seen, applying the blame prediction transformation makes the distance between the definition `def` and the check expression `check` smaller than the distance between the definition `def` and the application of `+` (`use`).

```
1 (define (triangle n res)  def
2   (check (number? n)
3     (if (= n 0)
4         res
5         (let ([tmp1 (- n 1)])
6           (check (number? res)  check···
7             (let ([tmp2 (+ n res)])  use
8               (triangle tmp1 tmp2)))))))
9 (triangle 10 "0")
```

Listing 6.3: Example of debugging errors with the help of blame prediction

To summarize the above, we define the check–use metric as the difference of the following distances:

define–check the distance between the definition of a variable and the first check expression involving it.

define–use the distance between the definition of a variable and the first usage of this variable in a primitive operation.

Note that the define–check distance must *always* be smaller than or equal to the define–use distance. The check–use distance must therefore be a non-negative number. In dynamically typed programs such as listing 6.2, the distance is zero, as primitive operations check their preconditions themselves. The goal of the blame prediction transformation is to maximize the check–use distance by inserting check expressions into the program as high as possible.

We can express this metric in terms of the program text by counting "levels"; we call this the "static prediction". We count one level for every if or let expression between a variable definition and its use in a primitive operation. In the case of listing 6.3 for example, the define–check distance for the `res` variable is two, as there are an if-expression (line 3) and a let-expression (line 5) between the definition and the first check expression. The define–use distance for the same variable is three, as the let-expression on line 7 comes after the check expression. Having computed both distances, the value for the check–use metric is the difference: one level.

We can also measure the check–use metric in terms of the program run-time; we call this the "run-time prediction". We take the evaluation of one primitive operation (`+`, `sqrt`, `string-append`, ...) as one unit of work or "tick". The ticks are a measure for the amount of computation that happens, invariant with regard to computer speed or the particular evaluation technique used. In particular, the use of recursive functions will cause a significant difference between static and run-time prediction. The check–use metric is measured as the difference in the number of ticks recorded by an instrumented interpreter at the `check` and `use` points.

One issue with this metric is that it depends on the input to the program and the (dynamic) path taken through it. For example, in a program that reads a list of num-

bers from a file, sorts the list and performs a computation over the ten largest values, the size of the list will determine the number of ticks between the start of the program and a failing type test. A variable that is defined before the list is sorted the list should ideally be checked before the work is done. In the benchmarks we perform in section 6.3, only a small minority of variables are affected by the size of the input.

Finally, this metric is related to the "root cause analysis" and "long time to crash" problems we defined in the introductory chapter. The first problem is related to finding the root cause of the type error, when one is reported. The larger the check–use distance in static prediction, the less code the programmer needs to inspect to find the real root cause. The second problem is centered around the time the program takes to report a type error and halt. This is the domain of run-time prediction, which measures the number of ticks saved between the check and use points.

Metric 2: Reduction in type tests

The second metric we define in this chapter is the number of type tests that the blame prediction transformation can statically verify and eliminate. In order to give an intuition for this metric, consider the `tak` program shown in listing 6.4. This code is annotated with black dots (•) at each location where evaluation must perform a type test before continuing. For example, before performing the `<` primitive operation on line 2, the evaluator must not only verify that the arguments `x` and `y` are numbers, but also that `<` is a function. Similarly, the recursive invocations of `tak` must verify that `tak` is a function before it can be invoked.

```
1 (define (tak x y z)
2   (if (not• (<• y• x•))
3       z
4       (tak• (tak• (-• x• 1•) y z)
5             (tak• (-• y• 1•) z x)
6             (tak• (-• z• 1•) x y))))
```

Listing 6.4: The original `tak` program

Compare this to the blame predicted `tak` program in listing 6.5, where the majority of type tests have been removed by check inference, and check simplification has merged duplicate type tests such as `(number? x)`. As the figure shows, blame prediction is able to remove almost all type tests[1], with only three explicit check expressions remaining. As with the check–use metric above, we can also measure this reduction in terms of run-time steps. For example, evaluating `(tak 10 5 2)` with the definition in listing 6.4 performs 5,427 type tests, while the definition in listing 6.5 only performs 1,685. This is a 69% reduction in the number of type tests performed!

[1] Of course there are more advanced and more suitable type systems [Tobin-Hochstadt et al., 2011; Wright and Cartwright, 1994] which are focused on eliminating the number of type tests. However,

There are two ways of looking at this reduction: if we apply blame prediction to a program and run it using a normal interpreter, the remaining check expressions represent *overhead*. Alternatively, if we run a blame predicted program using a trusting interpreter — one which does not test any preconditions in its primitive operations — only the check expressions explicitly introduced by the blame prediction transformation prevent the program from crashing. This metric thus relates to the "long time to crash" problem as well. By reducing the number of type tests in the program, the program spends less time verifying preconditions, reducing the time the programmer needs to wait for the program.

```
1  (define (tak x y z)
2    (check (and (number?• y)^{<,-}
3                (number?• x)^{<,-})
4      (if (not (< y x))
5        z
6        (check (number?• z)^{-}
7          (let ([t1 (tak (- x 1) y z)]
8                [t2 (tak (- y 1) z x)]
9                [t3 (tak (- z 1) x y)])
10           (tak t1 t2 t3))))))
```

Listing 6.5: The `tak` program after blame prediction

We calculate the reduction in type tests by comparing against a "dumb" variant of the type system. This variant does not infer types for variables and constants, it does not propagate types obtained by function calls, and it does not perform check mobility or simplification. In essence, this dumb type system produces check expressions at every primitive operation, like a dynamically typed programming language would. We can then count the reduction statically (the number of check expressions in the program), and dynamically by counting the number of check expressions evaluated. There can be a significant reduction in the number of dynamic type tests if blame prediction can hoist checks out of loops, or remove them altogether.
To recap, we have defined two metrics in this section: check–use distance, and reduction in the number of type tests. Additionally, both metrics can be measured both in terms of the program expressions and in terms of the run-time behavior of the program. In the next section we describe the program corpora on which we will apply these metrics in order to evaluate blame prediction.

these type systems are traditional, i. e. they refuse to run the program if they find a type error. Converting them to predict blame will require some work.

6.2 Program Corpora

In order to evaluate the effectiveness of blame prediction, we have applied it to programs from two well-known benchmark suites. These programs were originally written to evaluate the performance of programming language implementations, but have also served as program corpora for other program analysis techniques, such as type reconstruction [Tobin-Hochstadt et al., 2011; Wright and Cartwright, 1994]. We chose these two sets of programs in particular for two reasons: first, because they cover a variety of program purposes and programming styles. Secondly, these sets of programs are available in the Scheme programming language [Abelson et al., 1998], which is a superset of the Scheme$_\beta$ language we studied in this dissertation. As Scheme is a superset of Scheme$_\beta$, there were some programs which our implementation could not handle, which we briefly discuss in section 6.2.3. It is important to keep in mind that the aim of this section is not the performance of the programs themselves, but rather whether blame prediction can help with early detection of type errors.

6.2.1 Gabriel Benchmarks

The first corpus we use is a set of benchmarks collected by Gabriel [1985] to gauge the performance of LISP implementations. This book builds on earlier work [Gabriel and Masinter, 1982], where the authors discuss how to best benchmark programming language implementations.

The programs in the Gabriel benchmarks have been used to measure the performance of proposed interpretation or compilation techniques, such as concurrent garbage collection [Appel et al., 1988; Boehm et al., 1991], optimizing compilers [Adams et al., 1986], automatic parallelization [Harrison III, 1989], and more fundamental interpreter techniques [Clinger et al., 1988; Steenkiste and Hennessy, 1987].

6.2.2 Computer Language Benchmarks Game

The second corpus we use to evaluate the effectiveness of blame prediction is that in the Computer Language Benchmarks Game [Fulgham, 2008]. The purpose of the Computer Language Benchmarks Game, or CLBG, is to compare a set of standard programs between programming language *implementations* in terms of memory usage and execution speed. Anyone in the community can submit an improved program, as long as it adheres to a number of requirements. For example, the `fibo` program *must* calculate Fibonacci numbers in a tree-recursive fashion rather than iteratively.

Programs from this benchmark set have been used recently to evaluate the performance of new programming languages or implementations [Grossman et al., 2005; Schäfer and Poetzsch-Heffter, 2010; Tobin-Hochstadt et al., 2011], supercompilers [Mitchell and Runciman, 2007b], and interpretation techniques [Brunthaler, 2010].

The CLBG contains programs for a number of implementations of the Scheme programming language (Bigloo Scheme, Chicken Scheme, Gambit Scheme, MzScheme),

with minor variations with respect to input and output. We have chosen to analyze programs for Chicken Scheme [CHI] because this implementation is close to the R5RS [Abelson et al., 1998] specification of Scheme, and still supported on our development system. Some minor alterations have been made, such as removing optional argument declarations from the program and hard-coding command-line arguments where necessary.

6.2.3 Excluded programs

Both the Gabriel benchmark set and the CLBG set contain programs which our implementation could not handle. In this section we briefly discuss some of the language features which were the reason for discarding these programs.

Use of define-record By default, Scheme has only two data types: vectors and pairs (which can be used to construct lists, trees and other data structures). One popular language extension — formalized in Scheme Request For Implementation 9 [Kelsey, 1999] — enables programmers to define custom compound data types, like C structures. For example, in the `nbody` program from the CLBG program set we find the following:

```
(define-record body x y z vx vy vz mass)
```

This declares a new *record type* `body` with the named fields, accessors and mutators for these fields (such as `body-x` and `body-x-set!`), a constructor `make-body`, and a predicate `body?`. Custom record types are not supported in either our implementation or our formalism.

Use of syntax macros Scheme features hygienic macros [Kohlbecker et al., 1986], which enable programmers to write custom syntactic constructs without fear of variable capture. The `nucleic2` and `kanren` programs of the Gabriel benchmark set make extensive use of macros. Syntax macros can be expanded before passing the program to the blame prediction transformation. Note that the use of macros might hide extensive amounts of code, in which blame prediction might insert `check` expressions. Predicting blame for expressions that originate inside such syntax macros might be confusing to the programmer, therefore our implementation only handles simple macros such as **`and`**, **`or`**, **`unless`** and **`when`**.

Use of call/cc The **`call-with-current-continuation`** primitive, or **`call/cc`** as it is commonly called, is a Scheme primitive for capturing and reifying the continuation that is waiting for the result of the expression in the body. In the Gabriel benchmarks, the `ctak`, `maze` and `puzzle` programs use **`call/cc`** as an early exit mechanism. In the CLBG, the `heapsort` and `except` programs also use **`call/cc`** like this. We previously discussed **`call/cc`** as part of the section on non-local control

flow (section 4.3.3). We concluded that continuations can be used to express arbitrary control flow, and so all but the simplest uses of **`call/cc`** cannot be analyzed statically. Therefore, we exclude programs which make use of **`call/cc`**.

Other extensions to Scheme Some programs in the CLBG use Chicken-specific extensions to Scheme. For example, the `objinst` and `methcall` programs use a CLOS-like object system. The `message`, `prodcons` and `process` programs require support for threading, as defined in SRFI 18 [Feeley, 2001]. Finally the `regexdna`, `regexmatch` and `wordfreq` programs require support for regular expressions.

In the end, we retain 30 programs of the 39 originally from the Gabriel set, and 20 of the 43 programs originally from the CLBG set.

6.3 Results

In this section we will evaluate the blame prediction transformation according to the metrics defined in the beginning of this chapter, using the selected programs from the Gabriel benchmarks and the CLBG set. First, we describe how we present each metric.

Presentation of static prediction
We have instrumented our implementation of the blame prediction transformation to produce the define–use and define–check distances for all variables which appear in a check expression. We present our results (figs. 6.1 and 6.4) as a bar graph, with variables on the X axis. The Y axis shows the define–use distance in dark grey, with the define–check distance overlaid in light grey. The check–use distance is thus the light grey part at the top of each dark grey bar. Intuitively, the smaller this part is, the higher up the check expression is in the variable's lifetime. Additionally, every bar is scaled with respect to the size of the function, and variables are sorted according to their define–use distance relative to the function size. This scaling accounts for the cases where the blame prediction transformation inserts a check expression in small functions, which would otherwise be counted as a misleading 50% decrease in the define–check distance over normal dynamically typed programs. Finally, the bar chart only shows variables where the blame prediction transformation was able to effectively move a check expression upwards. For every benchmark set we describe how many type tests were moved.

Presentation of run-time prediction
As described earlier, the transformed program is run under an instrumented Scheme$_\beta$ interpreter. This interpreter keeps a tick count, which is incremented for every invocation of a primitive operation. Whenever a variable is introduced, checked, or used in a primitive operation, the current tick count is recorded. The number we report for

a given variable is then the number of ticks between the first check expression involving that variable and the first call to a primitive operation which uses it. Unlike the distance in the program text, the results for this distance can span multiple orders of magnitude. We therefore group the recorded distances in bins, where every bin is the logarithm base 2 of the number of ticks between check and use, rounded upwards. We present the bins on the X axis, and the number of variables for each bin on the Y axis. As with the distance in the program text, we do not include results for variables where check–use distance is zero. We report this metric in figs. 6.2 and 6.5.

Presentation of reduction in the number of type tests
Finally, we present the reduction in type tests (figs. 6.3 and 6.6) as a simple percentage: tests remaining in the blame predicted program versus tests in the input program. We measure this percentage both statically and dynamically, with programs on the X axis and the percentage of remaining type tests on the Y axis. The reduction in terms of program text (static) is presented in dark grey, the reduction in terms of dynamic type tests is presented in light grey. Additionally, the raw tables containing the remaining type tests in both the static and dynamic case can be found in tables 6.1 and 6.2 at the end of this chapter.

6.3.1 Gabriel Benchmarks

Static prediction Figure 6.1 shows the aggregated static prediction for all variables in the Gabriel benchmark. Out of 786 type tests, 48 type tests could be moved upwards over a distance of half their containing function. 71 type tests more (for a total of 119) could be moved up by 40%, and 63 more (for a total of 182) could be moved up by 30%. 604 type tests could be moved upwards by 20% or less, of which 95 could not be moved up at all.

Run-time prediction Figure 6.2 shows the logarithm base 2 of the run-time prediction on the X axis, and the number of type tests for each distance. We can see that the overwhelming majority of dynamic type tests happen up to 2^8 (= 256) ticks before they are used. While this is not a huge amount, it could still be a win for the programmer if the run-time prediction spans across the invocation of a user-defined function.

Some type tests are performed significantly earlier than they normally would be: respectively 2^{11}, 2^{12}, 2^{15}, 2^{16} and 2^{22} steps. These represent a significant number of primitive function invocations and thus a significant amount of time savings.

Reduction in type tests Figure 6.3 shows a bar chart of the type tests remaining after applying the blame prediction transformation. Statically only 29% of the type tests remain, while dynamically 26% remain. For most programs, there are roughly as many remaining type tests for both static and dynamic. `deriv` and `lattice2` are exceptions, where only 10% of dynamic tests remain, versus 20% of the static type tests. Vice

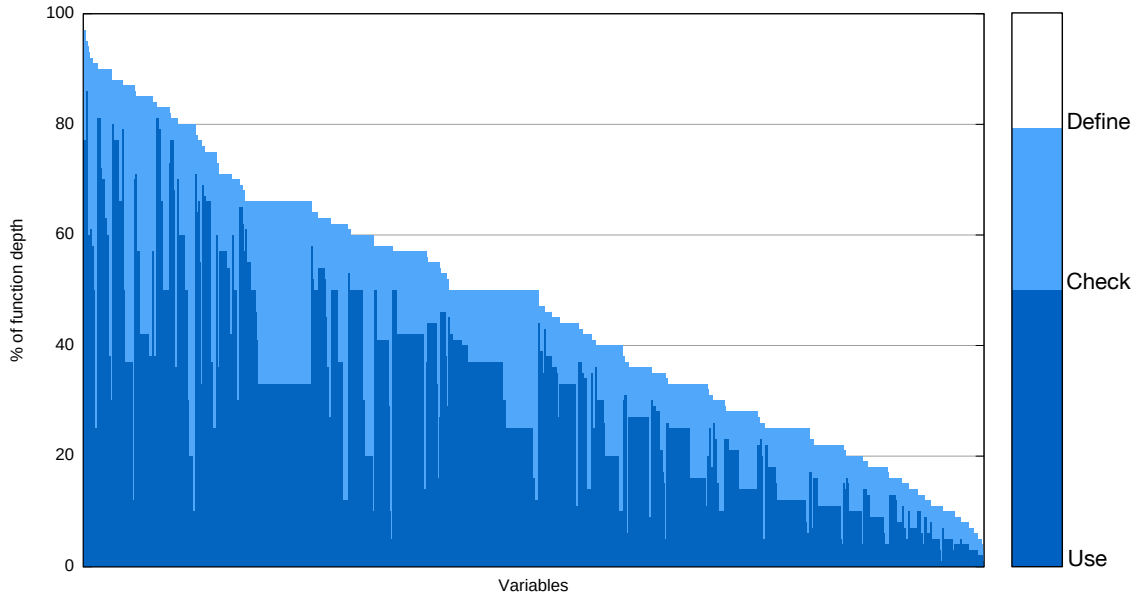

Figure 6.1: Static prediction across all variables in the Gabriel benchmarks

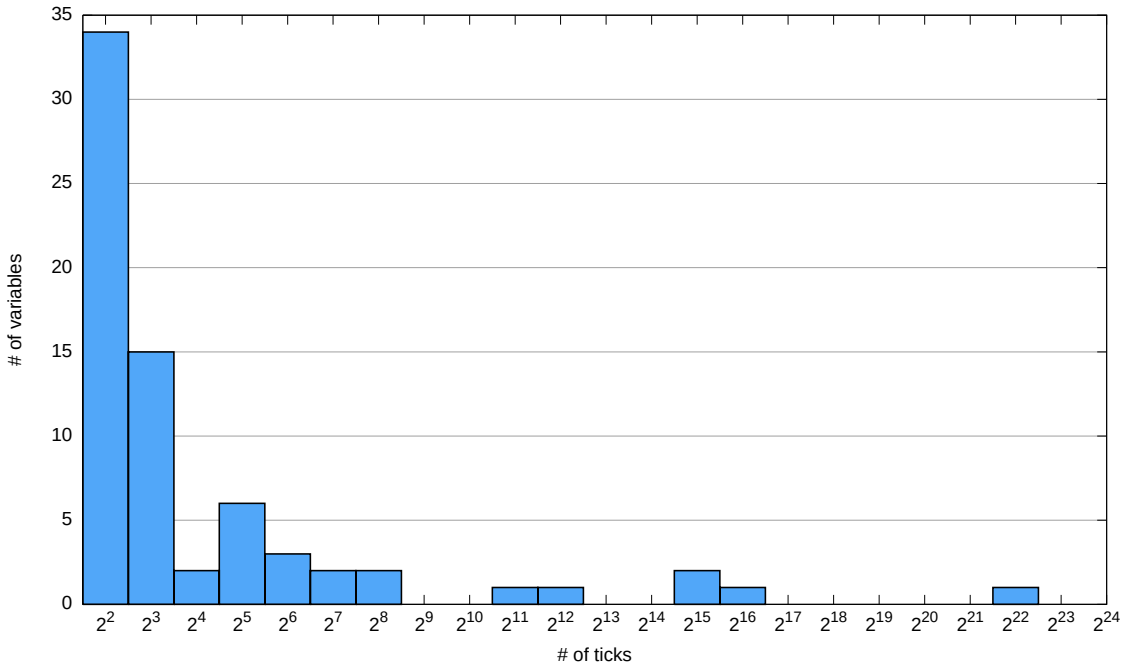

Figure 6.2: Run-time prediction across all variables in the Gabriel benchmarks

versa, we find that `mazefun`, `nestedloop` and `triangle` retained a considerable number of dynamic type tests. The raw numbers can be found in tables 6.1a and 6.2a at the end of this chapter.

6.3.2 Computer Language Benchmark Game

Static prediction Figure 6.4 shows an overview of all variables across all test programs in the CLBG benchmarks. Out of 141 type tests, only 8 could be moved upwards over a distance of half their containing function. 5 type tests more (for a total of 13) could be moved up 40%, and 10 more (for a total of 23) could be moved up by 30%. 118 type tests could be moved upwards by 20% or less, of which 9 could not be moved up at all.

Run-time prediction Figure 6.5 shows the run-time prediction for the CLBG benchmark. There are fewer type tests overall, and the programs tend to run for a shorter amount of time than the Gabriel benchmarks. We see a similar pattern as with the Gabriel benchmarks: the majority of type tests are only up to 2^4 (=16) ticks before they are used. As with the CLBG benchmarks only two type tests — at 2^9 and 2^{14} ticks respectively — represent early error detection opportunities.

Reduction in type tests Figure 6.6 shows a bar chart of the type tests remaining after applying the blame prediction transformation. Statically only 33% of the type tests remain, while dynamically 34% remain. For some programs (`ary`, `binarytrees`, `matrix`, `nestedloop`) the number of dynamic type tests is substantially higher. This can be attributed to type tests in loops which cannot be eliminated. The raw numbers can be found in tables 6.1b and 6.2b at the end of this chapter.

6.4 Evaluating Blame Prediction by Random Mutation

In the previous section we evaluated the blame prediction transformation by applying it to programs from both the Gabriel and the CLBG benchmarks. The programs in these suites were intended for performance benchmarking, so they do not contain any errors. In this section, we will techniques inspired by mutation testing [Howden, 1982; Jia and Harman, 2011] to test how well blame prediction can detect the inserted type errors and report them early.

Our methodology is as follows: we have taken benchmarks from both suites which completed in less than one hour. For every benchmark, we generated ten mutants where a variable was randomly replaced by another variable in scope. This simulates a programmer making a typo or copy-paste error. We then applied the blame prediction transformation to the mutants and ran the program under two configurations of our interpreter: one where `checks` were ignored, and one where `checks` halt the program

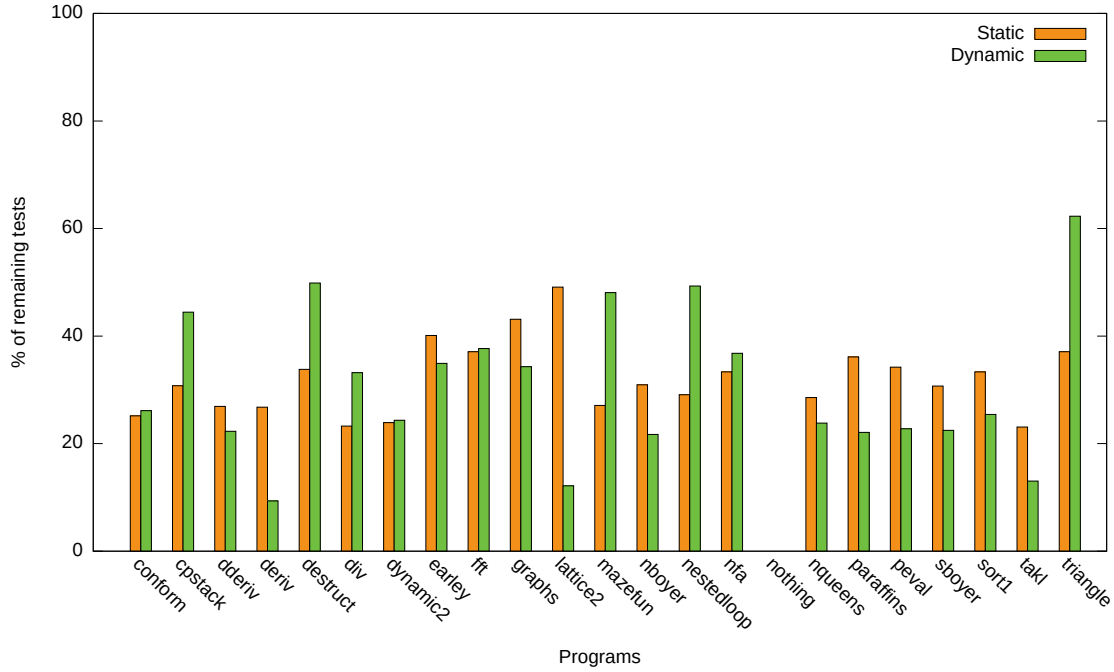

Figure 6.3: Remaining type tests in the Gabriel benchmarks (lower is better)

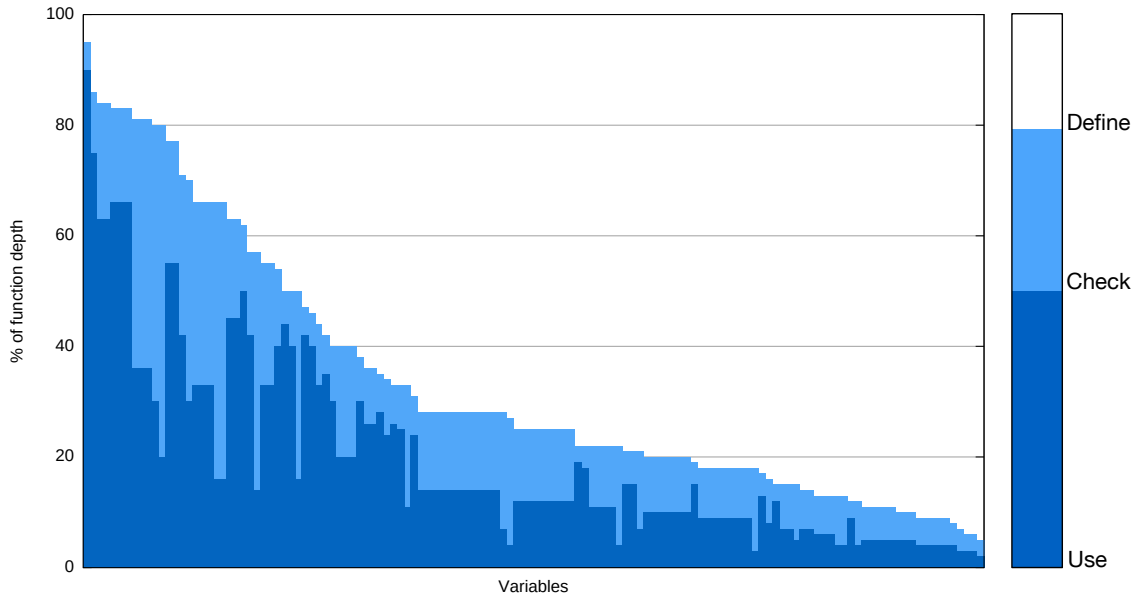

Figure 6.4: Static prediction across all variables in the CLBG benchmarks

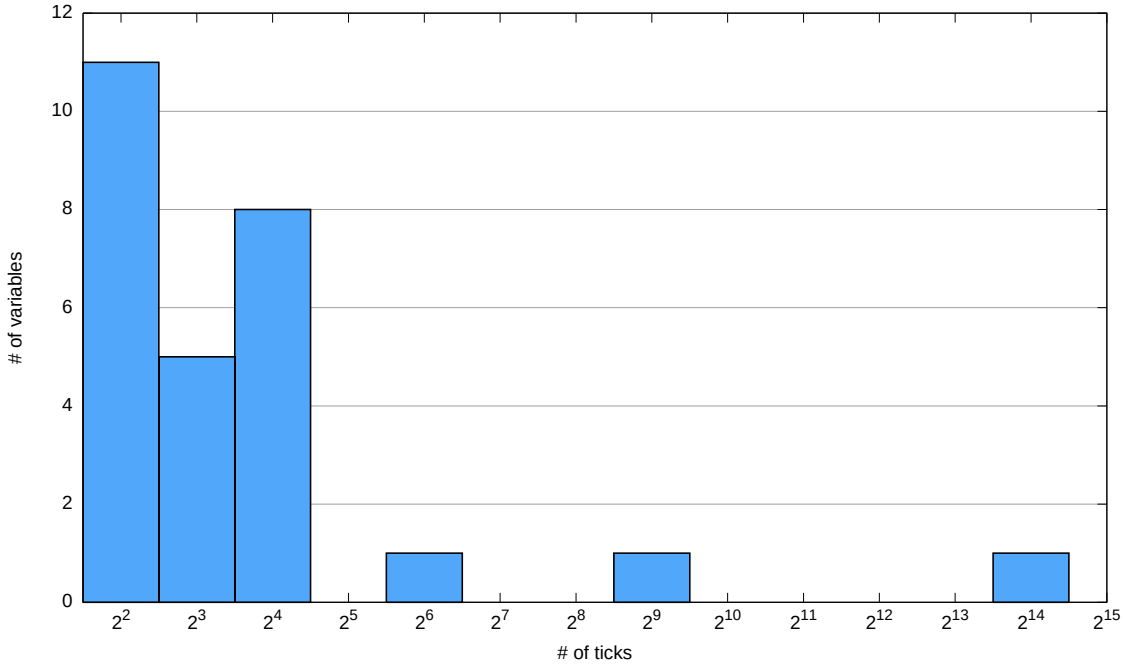

Figure 6.5: Run-time prediction across all variables in the CLBG benchmarks

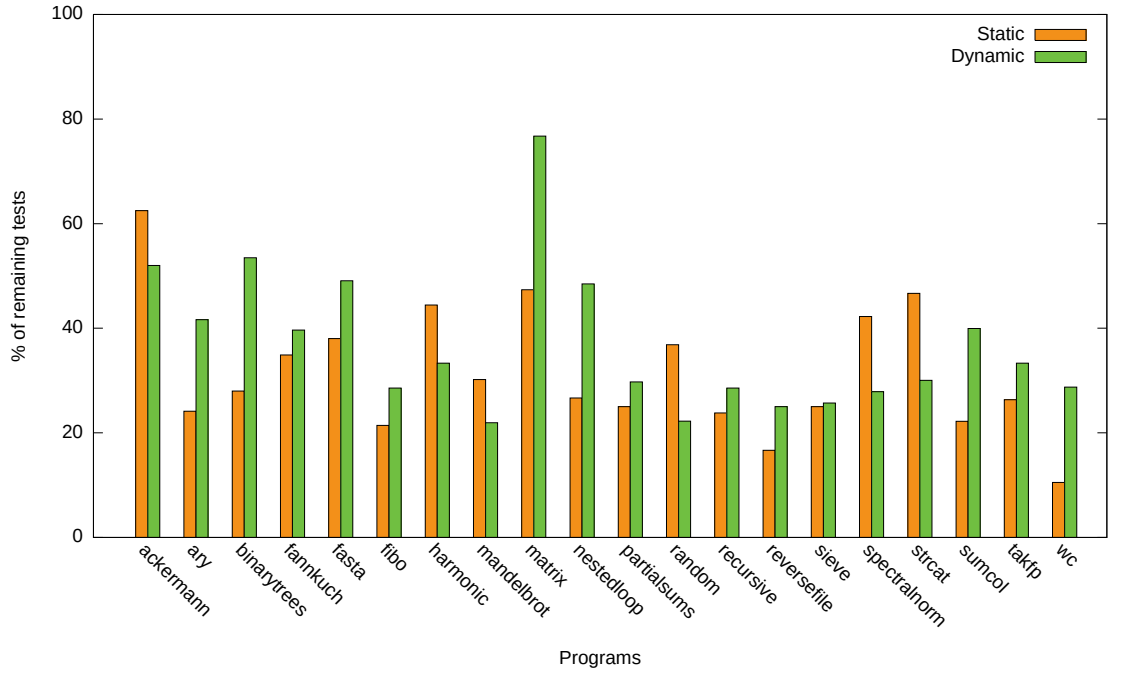

Figure 6.6: Remaining type tests in the CLBG benchmarks (lower is better)

if they detect a type error. In this section we show a few case studies where blame prediction was able to significantly predict type errors in advance.

Case study: `nqueens`

The first example is the one we presented in our introductory chapter, namely the `nqueens` program from the Gabriel benchmarks. In listing 6.6, the `x` in `(car x)` on line 15 has been replaced by `z`. The `try-it` function forms a tree-recursive process, where the first branch (line 13) is able to immediately place a queen at the suggested position and the second branch (line 15) skips a position and tries the next. By applying the mutation however, this process errors out after exploring the entire left side of the tree, because `z` is initially an empty list.

The blame prediction transformation will insert a `check` expression for this case around the `+` expression on line 12. We see that the `check` expression aids the root cause analysis by predicting blame for the `(car z)` expression, which turns out to be the root cause itself. Additionally, with `checks` enabled, running the blame predicted version of this program saves a large amount of time over regular execution. Concretely, the erroneous program performs 384,420 primitive operations before it reports the error, whereas with `checks` enabled it only needs 30 primitive operations. The corrected program performs 222,905 primitive operations in total. In terms of real-world clock times, the erroneous program runs for 2.30 seconds before it reports a type error, whereas the transformed program with `checks` enabled only runs for 0.51 seconds. In terms of the "long time to crash" problem, the numbers above show a clear reduction in time needed to run the program.

Case study: `spectralnorm`

The next example is taken from the `spectralnorm` program from the CLBG, specifically in its `approximate` function. The mutation is situated on line 16, where `vBv` has been replaced by `v`. As can be seen, the mutated program has a type error, as `v` is a vector whereas `vBv` is a number.

Applying blame prediction causes a `check` expression to be inserted right after the definition of `v`. In essence, this `check` stops the program before all of the work on lines 6–14 can be done. When the precondition fails, it blames the `/` operator which is exactly the root cause of the problem. With `checks` disabled it needs 6,421,128 ticks, or 25.34 seconds. Conversely, with `checks` enabled the program only needs 3 ticks or 0.50 seconds. The "long time to crash" problem is thus also solved.

Case study: `reversefile`

Our final example is the short `reversefile` program from the CLBG benchmark set. Listing 6.8 uses a simple recursive function to read lines from a file and print them out in reverse order. However, the mutation on line 5 attempts to apply the `line` variable as a function while it in fact a string.

```
1  (define (nqueens n)
2
3    (define (one-to n)
4      (let loop ((i n) (l '()))
5        (if (= i 0) l (loop (- i 1) (cons i l)))))
6
7    (define (try-it x y z)
8      (if (null? x)
9        (if (null? y)
10         (begin (if trace? (begin (write z) (newline))) 1)
11         0)
12       (+ (if (ok? (car x) 1 z)
13              (try-it (append (cdr x) y) '() (cons (car x) z))
14              0)
15          (try-it (cdr x) (cons (car x → z) y) z))))
16
17   (define (ok? row dist placed)
18     (if (null? placed)
19       #t
20       (and (not (= (car placed) (+ row dist)))
21            (not (= (car placed) (- row dist)))
22            (ok? row (+ dist 1) (cdr placed)))))
23
24   (try-it (one-to n) '() '()))
```

Listing 6.6: Case study: nqueens

The blame prediction transformation inserts a check expression right at the top of the **unless** expression. In effect, the blame predicted program halts the program after reading one line instead of the entire file. When it halts, it predicts blame for the call of line as a function, which is again exactly the root cause. With checks disabled, the program needs 21,468 ticks, or 0.73 seconds. With checks enabled, running the program needs 3 ticks, or 0.58 seconds. For this case, the "long time to crash" problem is only reduced slightly.

These were just a few case studies of where blame prediction can highlight programming mistakes and help the programmer find them. In all three cases, the blame prediction transformation was able to insert a check expression which pinpoints the root cause exactly. In addition, running the transformed programs saved a significant amount of time over the non-transformed programs in two out of three cases, thus solving the "long time to crash" problem.

6.5 Conclusion

In this chapter we have evaluated how well the blame prediction transformation — as described in this dissertation — can move type tests upwards in a program. Recall that the aim of moving type tests upwards is to catch run-time errors earlier. This has two

```
1  (define (approximate n)
2    (let ([u (make-vector n 1.0)]
3          [v (make-vector n 0.0)]
4          [vBv 0.0] [vV 0.0])
5
6      (do ((i 0 (fx+ i 1)))
7          ((fx= 10 i))
8        (mulAtAv n u v)
9        (mulAtAv n v u))
10
11     (do ((i 0 (fx+ i 1)))
12         ((fx= n i))
13       (set! vBv (+ vBv (* (vector-ref u i) (vector-ref v i))))
14       (set! vV (+ vV (* (vector-ref v i) (vector-ref v i)))))
15
16     (sqrt (/ vBv → v vV))))
```

Listing 6.7: Case study: `spectralnorm`

```
1  (define (reverse-input)
2    (let ([line (read-line)])
3      (unless (eof-object? line)
4        (reverse-input)
5        (write-line → line line))))
6
7  (with-input-from-file "reversefile.input" reverse-input)
```

Listing 6.8: Case study: `reversefile`

benefits: the programmer only needs to consider the code between a variable's definition and the first check expression involving it and it reduces the time the programmer has to wait for errors.

We defined two metrics for evaluating blame prediction. The first is the check–use distance, defined as the distance or time between the points where a variable is first checked and where it is used in a primitive operation. The second metric is the reduction in type tests, as the blame prediction transformation statically eliminates a number of type tests. Both metrics can be measured in terms of the program text and in terms of its run-time behavior.

We applied these metrics to two well-known and frequently-used program corpora: the Gabriel benchmarks [Gabriel, 1985] and the Computer Language Benchmarks Game [Fulgham, 2008]. We concluded that the blame prediction transformation was able to increase the check–use distance, both in terms of the program text and the run-time behavior. We highlighted a few cases where blame prediction significantly improved the distance and time the programmer needed to cover in order to find bugs in their program.

In the introductory chapter we introduced two problems related to finding problems in dynamically typed programs. The first problem was that of "root cause analysis", the process of finding the root cause of errors reported by the program. As we showed, the blame prediction transformation introduces type tests close to or at the root cause of the type error. The second problem was that of running the program and waiting for it to report a type error. In the case studies we observed, the time to crash was significantly reduced. In conclusion, we have shown that the blame prediction transformation solves both problems.

In the next chapter we will situate blame prediction among related work in the fields of program analysis, transformation, and debugging. We divide the related work in three broad categories according to the time when the analysis is performed.

Filename	LOC	Dyn	BP	Remaining
conform	437	469	118	25.16%
cpstack	19	13	4	30.77%
dderiv	45	52	14	26.92%
deriv	35	56	15	26.79%
destruct	48	71	24	33.80%
div	34	43	10	23.26%
dynamic2	1518	1978	473	23.91%
dynamic	1518	1984	481	24.24%
earley	448	486	195	40.12%
fft	86	89	33	37.08%
graphs	479	211	91	43.13%
lattice2	174	112	55	49.11%
lattice	186	118	43	36.44%
mazefun	167	166	45	27.11%
nboyer	618	236	73	30.93%
nestedloop	58	55	16	29.09%
nfa	38	57	19	33.33%
nothing	1	1	0	0.00%
nqueens	27	42	12	28.57%
paraffins	165	155	56	36.13%
peval	494	777	266	34.23%
sboyer	626	241	74	30.71%
sort1	99	99	33	33.33%
takl	23	26	6	23.08%
tak	12	16	4	25.00%
triangle	64	62	23	37.10%
Total	7419	7615	2183	28.67%

(a) Gabriel Benchmarks

Filename	LOC	Dyn	BP	Remaining
ackermann	11	16	10	62.50%
ary	18	29	7	24.14%
binarytrees	27	75	21	28.00%
fannkuch	72	86	30	34.88%
fasta	78	71	27	38.03%
fibo	12	14	3	21.43%
harmonic	9	9	4	44.44%
mandelbrot	39	53	16	30.19%
matrix	50	57	27	47.37%
nestedloop	21	30	8	26.67%
partialsums	34	52	13	25.00%
random	18	19	7	36.84%
recursive	31	63	15	23.81%
reversefile	7	6	1	16.67%
sieve	24	28	7	25.00%
spectralnorm	41	71	30	42.25%
strcat	32	30	14	46.67%
sumcol	6	9	2	22.22%
takfp	12	19	5	26.32%
wc	14	19	2	10.53%
Total	556	756	249	32.94%

(b) CLBG Benchmarks

Table 6.1: Tables showing the reduction in *static* type tests, resp. for the Gabriel and the CLBG benchmarks

Filename	Dyn checks	BP checks	Dyn time (s)	BP time (s)	% of checks
conform	52,768,901	13,792,852	863.90	288.51	26.14%
cpstack	57,643,277	25,619,234	879.99	442.78	44.44%
dderiv	22,650,015	5,050,005	350.19	107.74	22.30%
deriv	21,900,003	2,050,001	353.75	88.52	9.36%
destruct	120,582,483	60,126,230	1,788.71	959.94	49.86%
div	144,780,905	48,060,431	2,431.82	919.26	33.20%
dynamic	28,574,151	7,703,107	687.39	200.93	26.96%
dynamic2	27,134,259	6,605,833	698.08	194.98	24.34%
earley	26,997,082	9,428,798	488.69	241.53	34.93%
fft	153,641,006	57,869,001	2,601.11	1,457.61	37.67%
graphs	78,727,812	27,022,212	1,337.54	717.70	34.32%
lattice	696,223,644	75,946,002	11,996.30	3,962.28	10.91%
lattice2	689,661,689	83,897,087	12,309.45	4,237.54	12.16%
mazefun	157,029,503	75,481,500	2,402.88	1,029.74	48.07%
nboyer	553,614,434	120,200,392	9,664.39	2,810.93	21.71%
nestedloop	292,104,980	144,051,770	4,277.72	1,852.91	49.32%
nfa	385,500,005	141,900,001	6,924.74	2,624.34	36.81%
nothing	0	0	0.00	0.00	0.00%
nqueens	124,871,003	29,762,000	1,949.82	820.23	23.83%
paraffins	23,332,703	5,155,900	453.42	394.76	22.10%
peval	79,621,099	18,123,300	1,437.67	482.73	22.76%
sboyer	2,062,783,135	463,245,198	33,426.71	11,722.28	22.46%
sort1	167,343,093	42,560,562	2,858.71	6,312.20	25.43%
tak	119,268,003	39,756,000	1,763.00	882.81	33.33%
takl	265,287,625	34,607,398	4,289.52	1,465.70	13.05%
triangle	214,000,122	133,300,111	3,230.57	2,167.76	62.29%
Total	6,566,040,932	1,671,314,925	109,466.07	46,385.71	25.45%

(a) Table showing the reduction in *dynamic* type tests, Gabriel benchmarks

Table 6.2: Tables showing the reduction in *dynamic* type tests, resp. for the Gabriel and the CLBG benchmarks

Filename	Dyn checks	BP checks	Dyn time (s)	BP time (s)	% of checks
ackermann	427	222	0.01	0.01	51.99%
ary	12,018	5,003	0.22	0.17	41.63%
binarytrees	3,708,480	1,983,586	71.48	39.79	53.49%
fannkuch	923,604	366,195	17.88	10.57	39.65%
fasta	258,445	126,853	5.09	2.88	49.08%
fibo	24,672,048	7,049,157	357.51	162.61	28.57%
harmonic	599,998	199,999	10.84	6.83	33.33%
mandelbrot	73	16	0.01	0.01	21.92%
matrix	2,507,072	1,923,881	46.66	38.68	76.74%
nestedloop	1,235,823	599,186	22.42	11.00	48.48%
partialsums	92,516	27,503	3.71	2.59	29.73%
random	9,007	2,004	0.22	0.09	22.25%
recursive	3,600,075	1,028,683	68.15	28.39	28.57%
reversefile	42,932	10,732	0.76	0.43	25.00%
sieve	144,410	37,131	2.91	1.25	25.71%
spectralnorm	7,225,303	2,012,774	131.32	61.88	27.86%
strcat	3,874,862	1,164,242	71.53	47.29	30.05%
sumcol	5,005	2,000	0.12	0.08	39.96%
takfp	379,511,862	126,503,953	5,943.16	3,735.15	33.33%
wc	42,979	12,355	0.96	0.47	28.75%
Total	428,466,939	143,055,475	6,754.96	4,150.17	33.39%

(b) Table showing the reduction in *dynamic* type tests, CLBG benchmarks

7

RELATED WORK

In this chapter we situate our work in past and ongoing research in preventing and discovering program errors. This related work can be subdivided in three main strategies, which we highlight in section 7.1. In order to compare the related work to blame prediction, we define a number of criteria in section 7.2. Next, we discuss the actual related work, grouped by strategy (sections 7.3 to 7.5). Finally, section 7.6 summarizes and compares all related work.

7.1 The error detection landscape

In this section we discuss the three main strategies for detecting and preventing errors in programs.

Static analysis The first strategy is static analysis, which analyses the program without actually running it. The most basic static analysis, used in nearly every programming language implementation, detects syntax errors and references to non-existant variables.

The most well-known form of static analysis is the *type checking* found in statically typed languages such as Java, C#, Haskell, ML, and others. There has recently been a surge in research on retrofitting type systems onto dynamically typed languages such as Ruby [An et al., 2011; Furr et al., 2009], Erlang [Sagonas and Luna, 2008], and JavaScript [Anderson et al., 2005; Thiemann, 2005]. At the core of this type checking is the *type system*: a set of logical rules that describe prerequisites for correct program execution. The type checker takes the type annotations in a program and uses the rules in the type system to verify these annotations. If a contradiction is found, the type checker reports a type error. The type system determines the sort of errors that can be detected, for example typestate properties [Aldrich et al., 2009; Strom and Yemini, 1986], variable tainting [Ørbæk and Palsberg, 1997; Shankar et al., 2001], and nullable types [Fähndrich and Leino, 2003; Hubert, 2008].

A more powerful form of static analysis is *abstract interpretation*, where the program is executed with *abstractions* of values [Cousot and Cousot, 1977a, 1992b]. The degree of abstraction for values and primitive operations determines the kind of errors an abstract interpretation can detect. For example, abstracting numbers into "negative, zero or positive" already allows the detection of "division by zero" and "square root of negative number" errors. The downside to abstract interpretation is its potentially long running time and high memory usage. We will not consider abstract interpretation in depth for this reason.

Dynamic analysis The second strategy is dynamic analysis, which analyses the program while it is running. The simplest analysis performs run-time type tests when evaluating primitive operations or similarly performs null-pointer tests before accessing objects in object-oriented languages. A more involved method of dynamic analysis is to instrument the run-time system. This can be used to detect errors beyond those in primitive operations, such as printing warnings when variables are used without initializing them, or use of untrusted user input in system commands.

A more advanced strategy is *automatic instrumentation* of programs. In recent years there has been research into type systems that defer some checking to run-time. This enables them to accept more programs while still guaranteeing safety. One example of such a type system is gradual typing [Siek and Taha, 2006], where type casts are automatically inserted and verified at run-time.

Finally, the programmer can manually alter the program to enable dynamic analysis. In dynamically typed languages, programmers can use metaprogramming or annotations to automatically monitor their program for certain conditions. The most prominent example of this is contract technology [Meyer, 1992], where the programmer expresses constraints about certain values that are verified as they are used further on in the program.

Post-mortem analysis The last strategy is post-mortem analysis, which analyses programs that have already encountered an error. The main goal of this strategy is to work back from the error to find its cause. Typically, post-mortem debugging is done using *debuggers*, with the program stopped just after detecting the error or working from a memory dump of a crashed program. Finding the exact cause of unwanted behaviors of crashes is often a time-consuming process, so we will mainly discuss strategies which speed up this process [Bourdoncle, 1993; Lewis, 2003; Marceau et al., 2007].

In the rest of this chapter, we will discuss related work which fits in one of these three strategies.

7.2 Criteria

We define the following criteria:

C1 Time of error detection
While there are *three* strategies for error detection, we distinguish between *four* distinct periods where errors can be detected in the edit-compile-run cycle. Each period represents a trade-off between the ability to detect errors, the time taken to perform the analysis and the time the programmer has to wait for errors:

1. The first period is compile-time error detection: programmers get feedback about errors in their program and potential improvements while their program is being compiled. This is the most desirable, as feedback is instantaneous; the downside is that the class of errors which can be detected is often limited.

2. The second period consists of "ahead of run-time" error detection. By this we specifically mean approaches which report errors at run-time, but some time before they would normally be detected. Approaches in this category could, for example, report errors at the start of functions if some condition is not met or an invariant is violated. These approaches typically perform an analysis on the program up front and instrument the program, so feedback to the programmer comes later than compile-time, but earlier than run-time.

3. The third period are the approaches which only do run-time error detection. Many dynamically typed languages fall under this category, but the approaches we reference attempt to improve on this baseline in some sense.

4. Finally, the fourth period is for post-mortem analysis techniques, i. e. approaches where faults in the program are discovered by analyzing post-crash results.

C2 Amount of programmer intervention needed
Some of the related work described in this chapter requires extra work from the programmer in order to detect errors. For example, in a statically typed language this extra work comes in the form of type annotations. Ideally the programmer does not need to make any changes for errors to be detected, such that an unmodified program can be run with better error detection. Finally, some approaches simply aid in a tedious process, but the brunt of the work must still be done by the programmer.

C3 "Must-fail" analysis
In chapter 1, we outlined the reasons why we are looking for a "must-fail" analysis, i. e. a technique which only report errors which are *guaranteed* to happen. Many techniques form a conservative "might-fail" analysis and report errors even for impossible situations. For example, compilers for statically typed languages will typically reject programs where dead code contains a type error.

C4 Short analysis time
Finally, using an error detection approach should not add much additional time to the edit-compile-run cycle. This is a classical trade-off: the more time taken, the more errors a technique can detect, but the more the programmer's patience is tested. For our purposes, we consider thirty seconds on a modern laptop as the cut-off point. When examining related work, we will attempt to quantify the analysis time, but not many papers give accurate timings.

The ideal approach for error detection while developing dynamically typed programs must adhere to these criteria: C1 errors must be detected as early as possible, ideally at compile time. However, some erroneous situations can only be detected at run-time, sometimes long before the error actually manifests; C2 as little programmer intervention as possible is desired, as not to distract from the program being developed; C3 and C4 are a must, in order not to distract the programmer with errors that will not be triggered in a test run or make the edit-compile-run cycle too long.

7.3 Compile-time error detection: Static analysis

Type systems [Pierce, 2002] are a canonical solution to prevent illegal operations in a program. Illegal operations are those for which the language's semantics have no definition, such as indexing an array using strings or invoking an integer as if it were a function. A type system defines a logical system together with a set of rules for generating logical statements from the program text. If there are no contradictions, every expression in the program has a *principal type*; evaluating this expression will result in a value that has the same run-time type as predicted by the type system. Executing a well-typed program under a certain run-time system can then be proven to never "go wrong" [Milner, 1978].

Statically typed languages traditionally analyse the program at compile time and reject the program if a type error is detected. While this is desirable for the safety purposes outlined above, it prevents the execution of programs as long as there is an expression where the types are wrong, even if this expression is unreachable! For example, the programmer might want to change the type of a variable or method incrementally, testing each changed site separately. A type system will typically prevent the program from running until *all* sites have been changed, as it cannot guarantee safety for the entire program. Research by Hanenberg and Stuchlik [2012] has shown that this has an impact on development speed.

As we would like approaches to be "must-fail", general type systems will not suffice. In this section we describe type systems which are still "might-fail" analyses, but have interesting properties with respect to making type test explicit.

```
(define: (a b) (map [f : (a → b)] [l : (Listof a)]) : (Listof b)
  (if (null? l)
    l
    (cons (f (car l)) (map f (cdr l)))))
```

Listing 7.1: Definition of **map** in Typed Racket

7.3.1 Typed Racket

Typed Racket [Tobin-Hochstadt and Felleisen, 2008] (formerly Typed Scheme) is a language extension to Racket that enables programmers to assign types to their program. Instead of the regular **define** keyword, programmers can use the `define:` keyword to specify types along with expressions. Listing 7.1 shows how to define **map** in Typed Racket. The first line can be read as: "for all types `a` and `b`, **map** takes a function `f` of type `a → b` and a list `l` with elements of type `a`, and produces a list with elements of type `b`". Typed Racket only requires type annotations on top-level definitions and occasionally on recursive bindings such as **letrec**. The types of expressions within a function are determined by local type inference, as presented in Pierce and Turner [2000].

The designers of Typed Racket define modules as the unit of typing, which means that all top-level bindings in a Typed Racket module must have a type signature, and the entire module must be accepted by the type checker. After compilation, every module provides a type dictionary for each of its provided members, which allows for separate compilation [Tobin-Hochstadt et al., 2011, Section 5]. Interaction between Typed Racket modules is transparent: programmers can simply import and use code from other Typed Racket modules, just like regular Racket modules. To import untyped Racket modules into a Typed Racket module, each imported member must be annotated with a type. Vice versa, an untyped Racket module is allowed to import Typed Racket modules, but every imported member is transparently wrapped with a contract that enforces the correct use of that member at run-time.

One of the problems of adding a type system to an expressive dynamically typed language such as Racket is the distinct typing of variables dependent on the code path being followed. For example, the code in listing 7.2 defines a function that takes either a number or a boolean, and always produces a boolean. In the two branches of the if expression, x takes on a different type: it is a number in the true branch, and a boolean in the false branch. Typed Racket supports this common idiom using "occurrence

```
(lambda (x : (⋃ number boolean))
  (if (number? x)
    (= x 1)
    (not x)))
```

Listing 7.2: Typed Racket: Occurrence typing in action

typing", where every occurrence of a variable can have a distinct type. This distinction is made because of the `number?` test in the condition position of the `if` expression. Therefore, `x` must be a number in the true branch, and definitely not a number in the false branch (so a `boolean`). In Typed Racket, every predicate function is annotated with a *latent predicate* that supplies extra information about the arguments it is called with. The type inferencer then uses this latent predicate to modify the type environment when descending into the branches. For example, the `number?` function applies a latent `number?` predicate to its argument, so in the true branch `x` has type `number`, while in the false branch it has type `boolean`. Assigning types in a flow-sensitive manner allows the type system to follow along with the programmer's intentions.

In a follow-up paper, occurrence typing was significantly improved to solve a number of fundamental limitations in the original formulation [Tobin-Hochstadt and Felleisen, 2010]. These limitations involve asymmetry in occurrence typing with the addition of logical connectives (`and`, `or`, `not`). For example, `(and (number? x) (>= x 1000))` guarantees that `x` is a number in the true case, whereas no information is gained in the false case (`x` might not be a number, or a number smaller than 1000). This problem is tackled by introducing two *latent propositions* for each predicate: one for when it succeeds and one for when it fails. The latent predicates of combinations of expressions using `and`, `or`, `not` can then be derived by combining the propositions in a logical manner.

Another limitation solved by this paper involve predicates on values in compound data structures. For example, the test `(number? (car x))` logically implies that `x` is a pair, but also that `(car x)` is a number. The first implementation of occurrence typing only supported a single latent predicate on variables. In the follow-up, this limitation was removed by also recording the path taken to a value under test, which couples the information gained about the nested value with its container. This allows the type checker to make use of *both* facts.

Relevance to blame prediction Typed Racket rejects modules if they contain type errors, before any of the module's code is run. For this reason we classify it as "compile-time" error detection, although "load-time" is a better name. The analysis overhead of Typed Racket is small enough to not slow down the edit-compile-run cycle. The flexible, flow-sensitive type system allows errors to be reported once they are certain to occur, so it also is a "must-fail" analysis. However, Typed Racket requires programmers to annotate all top-level bindings, so its type safety does come at a cost to the programmer.

Nevertheless, the concept of occurrence typing is an inspiration for dealing with run-time type tests. Additionally, Typed Racket operates on entire modules: it demonstrates how typed modules can interface with other typed modules (by means of embedded type dictionaries), and with untyped modules (by means of wrapping code with run-time type tests).

7.3.2 Effect systems

We already explained type systems in section 7.3: they are logical rules which describe how to compute types for given expressions. These types intuitively correspond to the set of values that can be produced by evaluating the expression. However, types do not say anything about the intermediate evaluation performed to reach these values. In some contexts it might be necessary to determine whether a given expression performs side effects such as writing to global variables or performing I/O. The appropriately named effect systems [Gifford and Lucassen, 1986; Nielson and Nielson, 1999] do just that: they not only infer types for expressions, but also any side effects incurred by evaluation. For example, the `print` function has the effect IO, while an assignment to a global variable has effect $\mathsf{write}(\theta)$ where θ is its address. An example of a well-known effect system is the exception system in Java: methods must declare all the exceptions their body can throw or `catch` some of them.

Expressions that contain subexpressions must combine the effects of the subexpressions in a certain way. For example, a sequence of statements might take the union of effects of individual statements (for example, exceptions), or construct an ordered list of effects (for example, ordered memory reads and writes). Other combinations can be more complicated still, for example the exception set of a `try–catch` block does not contain any exceptions caught by `catch` clauses.

One downside to the original formulation of effect systems is that the derivation and calculation of effects was tightly woven into the type system, so a new effect system also requires a new type system. Marino and Millstein [2009] introduced the notion of a *generic* type and effect system. This system concentrates the interaction between type and effect systems in two operators at the type level: **check** and **adjust**. In almost every rule in the type system, the **check** function is invoked to verify whether the effect of a subexpression is allowed according to a privilege environment. The **adjust** function adjusts this privilege environment when the type system descends into subexpressions. Another approach to implement effect systems was shown by Wadler and Thiemann [2003]: they demonstrated how any effect system can be converted into an equivalent system that uses monads [Moggi, 1989]. Finally, Talpin and Jouvelot [1992] showed how to automatically infer types, effects, and the (memory) regions these effects work on.

Relevance to blame prediction Effect systems present the type system with a "side channel" for extra information: expressions have a type for the values they return, and an effect for the side effects they made along the way. In our case, effects could be used to model the required type tests for a primitive operation to succeed, while detecting type errors at compile time. Recall that regular type systems only use the final types of expressions to decide whether a given expression is well-typed. If the whole program is well-typed, the types are typically erased and the program is allowed to run safely (as "well-typed programs cannot go wrong"). An effect system, on the other hand,

could be combined with a very basic type system which always succeeds but generates "type-test" effect whenever a type test is needed.

Just like type systems, effect systems require annotations from the programmer. Effect inference is a part of type inference, which in turn only allows the program to run if no type errors were detected. Effect systems are therefore not a "must-fail" analysis. Finally, the type and effect analysis typically does not have a big impact on the edit-compile-run cycle, unless an error is detected.

7.4 Run-time error detection: Dynamic analysis

In this section we describe techniques which analyse the program as it is running, i. e. dynamic analysis. Interpreters for dynamically typed languages already perform dynamic analysis in their primitive operations and semantics, for example to raise an error when the expression `(+ 3 #t)` is executed, or when a division by zero occurs. This analysis is only rudimentary, however: it only raises errors at the last possible moment, often with a stack trace and nothing else.

In this section we explore different approaches to dynamic analysis which aim to detect and report errors beyond the rudimentary analysis. Note that some approaches already perform some analysis and/or code transformation at load-time, but the actual time of error detection and reporting still occurs occurs run-time.

7.4.1 Contracts

"Design by contract" was proposed by Bertrand Meyer as a novel way of constructing programs [Meyer, 1992]. In the original design philosophy, a "contract" is drawn up between two software components that defines the responsibilities of each party in the contract. When a contract is broken, one of the two parties is at fault; "blame" can be assigned to it. This blame is reported back to the programmer, in order to help discover why the contract is broken and which party is to blame. In this formulation, applying a contract to a value is simply applying a boolean predicate such as `int?` or `natural-number?` to it.

Findler and Felleisen [2002] introduced higher-order contracts, where contracts can be combined into other contracts. This allows programmers to make precise claims about functions and blame the correct party when a function receives the wrong input or when it produces the wrong output. For example, consider the **`map`** higher-order function: its contract states that it receives a function with a specific contract and a list of values. The implementation of **`map`** applies the function to each element of the list, so calling **`map`** with a function with contract $C_1 \to C_2$ requires that each element of the list passes contract C_1, otherwise the module that provided the element is blamed. The output of the function must pass contract C_2, otherwise the module providing the function can be blamed. Finally, Findler and Felleisen [2002] also introduced *dependent*

contracts, a slight variation of function contracts where the output contract of a function can make use of the input values provided.

After publication of that paper, a few variants of higher-order contracts were proposed. Each contract either used slightly different definitions of contract satisfaction or used higher-order contracts in another setting, for example in a lazy setting or during theorem proving. Only in [Dimoulas and Felleisen, 2011] did Dimoulas et al use observational equivalence to investigate contracts.

Contracts are used to document and verify built-in and library functions in the Racket programming language [Flatt and PLT, 2010]. For example, the documentation for the **`map`** function is shown in listing 7.3. Here, `procedure?` and `list?` are flat contracts over values.

```
1 (map proc lst ...+) -> list?
2   proc : procedure?
3   lst : list?
```

Listing 7.3: Documentation for the **`map`** function in Racket

Passing in values of the wrong type results in a contract violation, showing the user exactly which value violated which contract. Racket encourages programmers to define contracts for modules they write as well, which helps in tracking down errors.

Contracts can also be used to impose limits on program behavior; for example, Heidegger et al. [2012] define "monitoring contracts" for JavaScript functions. These contracts allow the programmer to specify sets of objects, and whether they may be read, written, or both. Their specification is in terms of "access paths", which are sequences of member selection operators starting from either a global variable, a function argument, or the receiver object this.

[Tobin-Hochstadt and Van Horn, 2012] combines contracts and symbolic execution, which yields a powerful abstract interpretation technique. The technique replaces concrete values with abstract contracts that flow through the program. This enables the verification of compositions of contracts, even if parts of the composition are only known by their contract (for example, code imported from another module).

Relevance to blame prediction Contracts allow programmers to express specific constraints about expressions in their code. These constraints can range from simple type tests all the way to complex predicates over values. Therefore, they are a powerful tool for ensuring program correctness.

Because of the way contracts wrap values and functions, they can detect errors ahead of normal run-time execution. Deploying contracts only requires the programmer to specify the contract to apply to the value, much like type annotations. More complex contracts require more work to specify. When a contract is broken, it assigns blame to one of the parties; this constitutes a "must-fail" analysis, as the programmer explicitly imposed the contract. Finally, deploying contracts in a program imposes a significant cost at run-time. In [Tobin-Hochstadt and Van Horn, 2012], the authors state that

> "[C]ontract checks take more than half of the running time for large computations such as rendering documentation and type checking large programs [Strickland et al., 2012]."

The remaining parts of this section about run-time error detection consists of approaches which transform programs to make type tests or -conversions explicit.

7.4.2 Soft typing

The goal of soft typing [Cartwright and Fagan, 1991] is to combine the best features of dynamic typing (expressiveness and simplicity) with the best features of static typing (early error detection and efficiency). To this end, the authors propose a modification to the type system used in ML that inserts run-time checks wherever the type system encounters an error. In other words, whenever the logical system reaches a contradiction, the failing rules is allowed to succeed (after inserting a type cast) and type checking continues.

The authors list *heterogeneity* as one of the requirements of their type system, which boils down to allowing expressions to belong to *unions* of types [Barbanera and Dezani-Ciancaglini, 1995; Pierce, 1991] instead of one type. This allows the type system to process programs in which for example the two branches of an if expression are of different types. Additionally, the type system has support for *recursive types*, where types are defined in terms of themselves. These are necessary to support self-application, among others.

The implementation of the soft typing system is done through two modifications of the ML type system: first, *circular unification* is used instead of regular unification. This allows the type inferencer to generate and solve recursive types. Second, types are *encoded* to allow the inference of union types. This encoding makes use of a record type that has a flag field for every possible type in the system. This flag indicates whether the type must be present ($+$) or absent ($-$). After type inference, monotypes are inferred where only one field of the record type is present, and union types where several fields are present. Whenever the set of types for an argument is larger than what a function accepts (for example a value of type $\mathsf{int} \cup \mathsf{string}$ is given to $+$), a *narrower* is inserted that performs a run-time type check.

A follow-up paper expanded soft typing to cope with the realities of Scheme [Wright and Cartwright, 1994]. This "Soft Scheme" supports variable-arity functions and continuations, both frequently used in Scheme. This expansion also supports mutation, but only to local identifiers; assigning to global identifiers or fields in a pair "may cause the accumulation of large, inaccurate types" in the author's words. The authors conclude the paper by remarking that up to 90% of run-time checks can be elided, and that typical programs run up to 3.3 times faster than ordinary dynamically typed programs.

There is still a problem with these approaches, namely that the types inferred are *flow insensitive*: the outputs of a function are completely disconnected from the inputs.

This is a problem especially given the lack of type annotations, as simple visual inspection can often yield a more accurate type. Aiken et al. [1994] tackle this problem using *conditional types*: a type $\tau_1?\tau_2$ (read: "τ_1 if τ_2") links a type with a certain condition. This allows a form of flow sensitivity: the type of an **`if`**-expression is then a union of the consequent *if* the condition is true and the alternative *if* the condition is false.

Relevance to blame prediction Soft typing combines a flexible type system with fallbacks to run-time type tests. When the type system detects a contradiction at a certain expression, it introduces a narrower or an `ERROR` tag instead of aborting. Errors are therefore detected at run-time, when the interpreter would detect it as well. The type system does not require annotations from the programmer, only a restriction on certain features. Soft typing is a "must-fail" analysis: as the narrowers are inserted in places where a conversion is needed, they can only raise errors along paths where the baseline would also throw errors. The analysis does not take a significant amount of time (twenty seconds for a 300-line program on a 1990-era machine), making analysis fast enough on contemporary machines.

7.4.3 Deferred type errors

In section 7.3 we remarked on a study by Hanenberg and Stuchlik [2012], which shows that making programs type-correct has an impact on development speed. *Deferred type errors* temporarily turn off the type checker of a statically typed programming language in order to quickly test changes. Allowing not completely type-safe programs to run *does* discard all safety guarantees normally given by the type system, but at the same time it allows the programmer to experiment and interact with the parts which *are* type safe.

DuctileJ [Bayne et al., 2011] is a plugin to the Java compiler which defers all type checking to run-time. Java's run-time type information and reflection mechanisms then replace the static type checking at a significant performance cost. The downside is that all this extra work at run-time adds a significant slowdown in comparison to normal execution. Apart from the technical contribution, this paper also outlines the software engineering advantages of deferring type errors to run-time. The first advantage is during prototyping: while exploring functionality, unused parts of a program can be left in a wrongly-typed state while the programmer implements a new part. The second advantage is during normal software evolution: performing a refactoring or change of data representation can be done in isolation without maintaining global static type correctness.

Another recent development is *deferred type errors* in the Glasgow Haskell Compiler [Peyton Jones et al., 2012]. While Haskell firmly falls in the statically-typed camp, this extension allows the compiler to emit run-time type errors for expressions which are ill-typed. The authors give a similar argument as the above paragraph for implementing deferred type errors.

Relevance to blame prediction Temporarily turning off the static typing in a statically typed language seems to be a good way to increase productivity when prototyping or evolving software. With the type system out of the way, the programmer can immediately evaluate proposed changes, before committing to making the program globally type-safe again.

These approaches turn the compile-time "might-fail" analysis of statically typed languages into a run-time "must-fail" analysis, with no extra work from the programmer. In both cases the analysis does not hinder interactive development, although no timings are given. In the Haskell work, the authors state that deferred type errors introduce extra boxing and unboxing operations in the program, but that most of them are promptly optimized away by the compiler. In case of DuctileJ the authors state that iterating on the program development was faster, but at run-time their program incurred a significant slowdown because their implementation relies on reflection.

7.4.4 Gradual typing

Dynamic and static typing have a different stance on programs that contain type errors. Dynamic typing allows such programs to run but throw a type error when faulty expressions are reached, whereas static typing rejects such programs at compile time. As such, programs and services are often prototyped in dynamically typed programming languages for development speed, and later rewritten in a statically typed programming language for safety and performance reasons. A direct rewrite has two distinct disadvantages: First, depending on the size of the program, a rewrite might take a significant amount of time, during which the old program must be supported. Second, some expressions might not be directly expressible in the target language, increasing complexity.

One approach to solve this software evolution problem is to *gradually* transform the program to a statically typed one. Gradual typing [Siek and Taha, 2006] enables programmers to gradually add type annotations to their program. A central part of gradual typing is the "unknown type" $\star$ which can represent any type[1]. Unannotated parts of the program automatically receive the type $\star$.

In essence, a gradually typed program is divided into a typed part and an untyped part. If the type system detects an error in the typed part, a type error is signaled just like regular static typing. Errors in the untyped part are reported at run-time, as normal. To make the type system sound, *coercions* are inserted at the boundary between typed and untyped code. These coercions are similar to narrowers in soft typing: they embody a run-time type test that must succeed for the program to continue.

At run-time, a cast verifies that the dynamic type of its argument is *consistent* (written $\tau_1 \sim \tau_2$) with the type it is being cast to. Type consistency is a broader concept than type equality as used by other type systems: two types are consistent if they are equal, or if one of them is $\star$, or if they are consistent in their respective components. It

[1] Early publications represented the unknown type with a question mark (**?**).

is very important to note that type consistency is not transitive, i. e. given that $\alpha \sim \star$ and $\star \sim \beta$, then $\alpha \sim \beta$ does not automatically hold.

In a follow-up paper, the authors extend gradual typing with support for objects [Siek and Taha, 2007]. In this system, objects are records of typed and named slots. They employ width-based subtyping, which means that an object A is a subtype of object B if A contains at least all slots in B. To reconcile gradual typing with object subtyping, the consistency relation from the original paper is augmented with *masking*, which removes the parts that are unknown in one of the two types. The combination of consistency and subtyping relation ($\lesssim$) consists of verifying whether the common parts of both types are subtypes.

$\star$ is considered neutral to subtyping, so only $\star <: \star$ holds. This formulation avoids arbitrary type conversions: if $\star$ is allowed as a top type, the gradual typing mechanics would allow implicit up- and downcasts to and from the type $\star$. This would break type safety for fully annotated parts of the program.

Type inference for gradually typed programs was introduced in [Siek and Vachharajani, 2008], which allows programmers to omit the $\star$ type annotations from most of their program. The type inference algorithm consists of two steps: constraint generation and constraint solution. The first step associates a type variable with every node in the input program and then uses the typing rules to generate constraints between these type variables. The constraint solution step then solves these constraints, always taking the least upper bound among types. Types are ordered by their "informativeness", where types with fewer $\star$ are considered more informative.

That paper realized the base goal of gradual typing: inferring as much information as possible from an untyped program and then performing the remainder of the checks at run-time. A more recent and mature type inferencer is found in [Rastogi et al., 2012], where gradual type inference is applied to ActionScript, a language closely related to JavaScript [Crockford, 2008]. Rastogi et al. report that they are able to recover all the types in 13 out of 17 benchmarks chosen from the SunSpider and V8 benchmarks (V8 Team [2011]; WebKit Team [2010]).

Relevance to blame prediction Similar to soft typing, gradual typing inserts run-time type tests closely around uses of expressions with the "any type". These type tests are inserted automatically, so no programmer intervention is required, and they only trigger errors when they are encountered during execution. Type errors in the fully statically-typed parts of the program still prevent the program from running. There are no reports of the time taken for the analysis itself, although it is considered to be on par with regular type systems.

7.4.5 Preemptive type checking

"Preemptive type checking", a recent development, is described in [Grech, 2013]. Its goal is "to force the termination of the program execution as soon as it can be detected

that a type error is inevitable". At the core of that work is an analysis that determines so-called present and future types for every variable in the program. The present types are a set of types that the variable can contain at any given point in time, while the future types are those types that the variable could be used as. If any future type is not present in the present types, the analysis reports that a type error may manifest itself.

At the core of this approach is a k-CFA analysis [Shivers, 1988, 1991], working on the bytecode of the program. This allows the approach to reason over *call stacks* (of depth k) instead of only the code. In such a call stack, variable reassignments (present types) can be matched against their uses (future types). If there is a mismatch, an `error` statement is inserted into the program. The analysis makes sure that errors are only inserted along critical paths, i. e. places where the erroneous statement *must* be encountered. If this is not done correctly, the semantics of the program are different, which is not desirable when debugging programs.

Relevance to blame prediction This work is *very* relevant to blame prediction, it has the same goal of predicting future type errors. The correctness and usefulness of the results are tied to the underlying static analysis, which requires more and more time as the context size is increased.

Applying preemptive type checking to a program results in (as the name states) preemptive checking of type errors. As presented in Grech [2013], the programmer needs to avoid a lot of features of Python for the analysis to work, but the author states that the analysis can be extended to cover the necessary features. As remarked above, preemptive type checking only inserts `error` statements when it is certain that an error will occur, so it is a "must-fail" analysis. Finally, the underlying analysis imposes a tradeoff between precision and speed: supplying more context enables better and earlier errors, but at the cost of more time spent in the analysis. For medium to large programs, the time spent on analysis can outweigh the time actually running the program. This reducing the usefulness of this approach, as it becomes faster to just run the program rather than wait for analysis to complete.

7.4.6 Tagging/Untagging optimizations

In the run-time system of interpreted languages, values are often *tagged* or *boxed*: they are stored together with a tag that identifies their actual type. This allows the system to perform run-time type tests in primitive operations. However, there is a cost associated with this mechanism: for example, to sum two values together they must be untagged, the native numbers must be added together, and the resulting number must be tagged again. In code that uses a lot of primitive operations, values will often be tagged only to immediately be untagged by the next operation.

One effort to avoid such situations is the work done in Henglein [1992]: it presents a "calculus of coercion" that makes tagging and untagging operations explicit in a pro-

gram. Subsequently, type inference is applied and (un)tagging operations are removed where it is safe to do so, or where one value is tagged and immediately untagged. The author reports that a simple analysis can already remove more than half of all tagging and untagging operations from a small set of programs, ranging from 700 bytes to a 232 kilobyte program.

Another instance of a tagging/untagging optimization can be found at the end of Tobin-Hochstadt et al. [2011], as an example of the use of providing language extensions as libraries. Normally, arithmetic operators such as + and - support all numeric types by dispatching on the type tags of the provided arguments and selecting an appropriate type-specialized operator. After inferring types of all expressions in the program, such generic arithmetic operations can be replaced by specialized operators such as `unsafe-fl+` if the types are known statically. This operator bypasses the tag tests and produces a new value tagged as `float`. The authors report that replacing general arithmetic operators with specialized ones yields a modest increase in performance for arithmetic-heavy programs.

Finally, an early report on the development of the Glasgow Haskell Compiler [Peyton Jones, 1996] discusses various optimizations done on the typed intermediate "Core" language. In Haskell, values are only evaluated when they are needed. To this end, every value is boxed: it either contains a computation that produces the value, or a pointer to a concrete value. Similar to the tagging/untagging optimizations above, values need to be evaluated and unboxed before they can be used in a primitive operation. One of the optimizations described in the paper exposes the boxing and unboxing of values to the optimizer, which allows it to remove superfluous boxing/unboxing operations. This in turn allows the compiler to generate efficient low-level code.

Relevance to blame prediction The papers discussed above all serve the same purpose: to make tagging/untagging (or boxing/unboxing) explicit and then remove superfluous pairs of tagging/untagging operations. These approaches do not serve to improve error detection or program debugging, but *do* make the point that making tagging explicit is good for performance reasons. With blame prediction, we argue that making tag tests explicit additionally improves error detection and debuggability, as the tag tests are moved upwards without affecting the program semantics. Removing statically-known-to-be-safe operations yields an improvement in error detection, as the programmer can focus on the remaining tag tests.

As for the criteria, all of the above approaches transform the program at compile-time, they do not require any programmer intervention, and they have a short analysis time. They will not remove tagging/untagging operations that are statically known to fail, so we consider them as "must-fail" analyses, even though their main goal is not error detection.

7.5 Post-mortem error detection: Debuggers

Finally, this section covers error detection "after the fact", i. e. when the program has reported an error and the programmer needs to figure out the mistake(s) which led to the error. The approaches described in this section fall along one of two lines: the first attempts to give the programmer more information, i. e. a full history of the program state instead of a snapshot. The second line of approach automates away some of the labor involved in debugging, namely stepping through and searching for 'interesting" program states.

7.5.1 Omniscient debugging

Traditionally, debuggers are used to run a program under supervision and step through it until either a programmer-defined point in the program is hit, the program crashes, or the program exits normally. In case of a crash, the program state is captured right before it is stopped and removed from memory. The programmer can then walk the stack trace leading up to the crash and inspect some values in order to form a *hypothesis* on why the program crashed. Future debugging sessions then focus on verifying the hypothesis, discarding wrong hypotheses, making new ones, and so on.

One particularly hard hypothesis is centered around variables changing value unexpectedly. To verify this hypothesis, the programmer needs to do a search over various states of the program to see where the variable changes. If the programmer steps too far through the program, the value has been changed by some nested expression or function call and the program must be restarted. This kind of debugging would be easier if the debugger kept a *history* of program states up until a certain point, such that a query "when did this variable change?" suffices. This was exactly the motivation for *omniscient debugging* [Lewis, 2003], where the author calls the above scenario a "lost snake in the grass" problem. While keeping a history of program states is costly both in terms of memory and performance, the overhead imposed by such systems is slowly decreasing [Pothier et al., 2007]. The state of the art, according to [Pothier and Tanter, 2011], is a 10x to 30x slowdown, while earlier systems manage a 100x to 300x slowdown.

Once a history of program states is captured, the programmer can move forwards and backwards in time to step through the execution of the code. For every variable a list of successive states can be shown, which allows the programmer to pick out unexpected changes. Clicking these changes jumps to a different program state, with full stack traces per thread. Unfortunately, such great power comes with a cost: the author reports that programs under omniscient debugging are orders of magnitude slower and consume memory proportionate to the number of events. Omniscient debugging is therefore best left for isolated test cases.

In 2009, the GNU debugger GDB was released with support for "reverse debugging" [Boothe, 2000; Saito, 2005], which allows programmers to run their programs

in reverse. At its core this technique is similar to omniscient debugging, but by reconstructing program states the program can actually *be run in reverse*.

Relevance to blame prediction In contrast to regular debugging, omniscient debugging allows the programmer to freely move forwards and backwards in the program states in order to narrow down error causes. Omniscient debugging is still a manual process: programmers must still trace back through the program to the specific point where an error was made.

Omniscient debugging still requires the programmer to construct and test hypotheses, which is a manual process (whereas little to no intervention is more desirable) and requires a considerable amount of time (which should be minimized). The authors of [Pothier et al., 2007] report a ten-times slowdown when running the program, which results in one experiment taking thirty-eight minutes instead of four. The typical starting point of a debugging session is the point where an error is detected, so we classify omniscient debugging as "post-mortem".

7.5.2 Automatic debugging

An important aspect of debugging is making assertions about properties of the code and verifying whether these assertions hold. Program errors are typically caused by these properties changing in some parts of the code. Automatic (or abstract) debugging [Bourdoncle, 1993] allows properties to be automatically checked by means of abstract interpretation. In the cited paper, the author proposes two classes of assertions: *invariant* assertions and *eventual* assertions. The former are properties that must always hold, whereas the latter must hold in *some* execution paths. In order to automatically debug a program, the programmer specifies these assertions. The program is then evaluated using an abstract interpreter and the resulting tree of program states is inspected with respect to the assertions. If a subtree is found to violate one of the assertions, an error is signaled and the programmer can inspect the states that caused it.

Relevance to blame prediction The paper cited introduced the concept of a "might" versus "must"-analysis, respectively for eventual and invariant assumptions. Blame prediction only predicts blame in the "must" cases, not "might".

Apart from this paper, we could not find other advances in automatic debugging[2], but the remarks from omniscient debugging apply here as well, as the programmer still needs to construct and test hypotheses herself.

[2] Although there is related work for logic programming [Ducassé and Noyé, 1994].

7.5.3 Scriptable debugging

Typically, debugging is a process where the programmer needs to manually halt and resume the execution of the program under inspection. With the release of version 7, the GNU debugger GDB has integrated support for plugins written in the Python programming language. These plugins can also be used to automate repetitive tasks and halt program execution based on complex conditions. An example plugin that ships with GDB automatically pretty-prints the contents of C++ containers.

In order to speed up the debugging process, approaches have been proposed where programmers interact with *streams of events* coming from the program instead of slowly stepping through it. This approach is called "scriptable debugging" [Ducassé, 1999a,b; Marceau et al., 2007]. These events can be method entries or exits, variable updates, and so on. To actually aid in debugging the program, the programmer can combine and consume these streams with FrTime [Cooper and Krishnamurthi, 2006], an implementation of the functional reactive programming (FRP; Elliott and Hudak [1997]) paradigm. When these streams detect an interesting event, they can programmatically halt the program. This enables programmers to describe invariants in their program in a natural way and immediately catch errors before they manifest.

Yit Phang et al. [2013] presents a fusion of scriptable debugging with time-travel debugging. Their approach represents the program execution as a stream of program snapshots and events, which can be manipulated using standard stream-processing primitives. Every point in the stream is materialized on-demand to avoid the exorbitant memory cost associated with time-travel debugging. The authors state that programming in this style (similar to FRP) eliminates problems caused by the callback-oriented interface of other scriptable debuggers, while at the same time enabling reasoning over events at different points in time.

Relevance to blame prediction Scriptable debugging is a powerful technique for reasoning about program execution and closing in on the points where execution starts to go wrong. Unfortunately, using such a debugger requires a high familiarity with both the run-time system and the scripting language used for the debugger. Additionally, the programmer must have previously narrowed the problem down to a specific hypothesis.

Depending on this hypothesis, scriptable debugging is either a post-mortem approach or a run-time approach. The programmer needs to define a script for the debugger to use, which has the potential to be more complex than the code being debugged. Finally, the time taken for analysis depends entirely on the script being used and the original estimate of what the bug is.

This concludes our survey of related work. In the final section we summarize our findings.

7.6 Conclusion

In this chapter we surveyed other research which aims to help programmers with error detection. We first defined criteria for evaluating the related work with respect to our goals. Next, we applied these criteria to the three main analysis strategies: compile-time error detection, run-time error detection, and post-mortem error detection.

These strategies can be summarized as follows:

- Compile-time error detection attempts to identify errors in the program statically, without running it. Type systems were discussed, which aim to eliminate illegal operations at run-time by ensuring variables are defined and used consistently with the same type.
- Run-time error detection runs programs with instrumentation. This instrumentation mainly consists of run-time tests to ensure the program "does not go wrong".
- Finally, post-mortem error detection attempts to identify errors in the program by backtracking after they have been encountered.

Table 7.1 summarizes our enumeration of related work. The three strategies are arranged as groups of rows, while the columns represent the criteria we used. The values for the first two criteria are explained in the legend below the table.

The first group of related work is compile-time error detection. Both strategies report errors at compile time, guided by annotations from the programmer. However, these approaches are not a must-fail analysis.

The next — and largest — group of approaches perform run-time error detection. The first approach (contracts) enables the programmer to specify properties of modules in their program; at run-time these properties are verified where necessary. Both fail when the property is not satisfied, but this is still some time ahead of when the property is actually *used*.

The remaining approaches in the run-time error detection group attempt to insert explicit type tests or remove them altogether. The difference lies in what these approaches do with the remaining explicit type tests: Soft typing inserts explicit type tests only when it cannot determine an exact type in advance. Deferred type errors are run-time type tests inserted in lieu of the compile-time checking of statically typed languages. Gradual typing inserts type conversions when going from and to typed and untyped expressions, but additionally rejects the program when it detects type errors in statically typed expressions. Preemptive type checking will not insert type tests, but rather explicitly abort the program if some type tests are guaranteed to fail. Finally, tagging optimizations simply eliminate redundant (un)tagging operations.

The error detection time for almost all approaches is run-time, except for gradual typing which has additional compile-time checking, and preemptive type checking

and contracts which can abort the program slightly earlier. Apart from preemptive type checking, all approaches have a short enough analysis time to be usable during periods of heavy development.

The last group of related work is post-mortem error detection. These approaches require significant amounts of manual programmer intervention and are therefore not suited for the problems we aim to solve.

	Error detection	Programmer intervention	"Must-fail" analysis	Short analysis time
Typed Racket	C	A	✗	✓
Effect Systems	C	A	✗	✓
Contracts	A	A	✓	✓
Soft typing	R	N	✓	✓
Deferred type errors	R	A	✓	✓
Gradual typing	C + R	N	✗	✓
Preemptive type checking	A	N	✓	✗
Tagging optimizations	R	N	✓	✓
Blame Prediction	A	N	✓	✓
Omniscient debugging	P	M	-	✗
Automatic debugging	P	M	-	✗
Scriptable debugging	P + R	M	-	✗

Legend, least to most desirable

Error detection	(P)ost-mortem; (R)un-time; (A)head of run-time; (C)ompile-time
Programmer intervention	(M)anual; (A)nnotations required; (N)o programmer intervention

Table 7.1: Overview of discussed related work

Finally, we can situate blame prediction among the related work described in this chapter. Blame prediction is a dynamic analysis strategy, as it aims to perform type tests ahead in time compared to the normal interpreter. It runs on unmodified programs and presents a "must-fail" analysis, as it only fails when the original program would fail too. As we will show in the next chapter, it only takes a small amount of time to transform a program. Regarding run-time overhead, blame prediction significantly *reduces* the amount of dynamic checks.

The next and final chapter concludes this dissertation. It reiterates the research goals and the key insights from the earlier chapters. It also sums up a number of known con-

ceptual and practical limitations of blame prediction and outlines future avenues for the blame prediction transformation itself, but also applications of blame prediction.

8

CONCLUSION AND FUTURE WORK

In the final chapter of this dissertation we reiterate on the research goals set out in the first chapter (section 8.1). Next, we restate our contributions in terms of the work discussed in the previous chapters (section 8.2). Finally, we discuss limitations of the current approach in section 8.3, and future avenues for blame prediction that can be explored (section 8.4).

8.1 Summary

This dissertation is formulated in the context of dynamically typed programming languages, which are typically interpreted or run in a virtual machine. The primitive operations of these languages must ensure that their inputs are of the correct type, and report a type error otherwise. For example, the `string-append` function must receive two string values as arguments. While these type tests are never made explicit, programmers use them to reason about the types of expressions that make up their programs.

When a type error *does* occur, the programmer receives a report of the type error, typically with a stack trace of the program execution up to that point. The programmer then needs to trace back through the program in order to find out where the wrongly-typed value was computed or passed through the environment. We called this the *"root cause analysis"* problem, as the programmer needs to find the root cause of the problem. In addition, every change requires the program to run again, which might take a lot of time. We called this the *"long time to crash"* problem.

This stands in sharp contrast to statically typed programming languages, where either types must be declared up front or inferred, and programs are only allowed to run if the type checker accepts the program. Programs written in statically typed programming languages suffer from neither problem, but the downside is that getting programs to be type error-free is not always trivial. Often, only a small slice of the program is actually needed, but the entire program must be type error-free.

In this dissertation we defined a program transformation, called blame prediction, which makes the implicit type tests of dynamically typed programming languages explicit and moves them upwards in the program as far as possible. This tackles the "root cause analysis" problem by reducing the fraction of the program which must be searched for the type error. The "long time to crash" problem is similarly tackled by performing the type tests earlier, ideally before heavy computations.

8.2 Restating the Contributions

This section summarizes each chapter of the dissertation and lists their individual contributions.

- Chapter 2 introduced the blame prediction transformation for a small functional core language similar to the lambda calculus. This transformation consists of four stages: 1) Check inference using a novel type system, which associates every expression in the program with its type and the type tests which are performed by evaluating this expression; 2) Check introduction, where the type tests are inserted as `check` expressions; 3) Check mobility, which moves the type tests upwards in the program in a semantics-preserving manner; 4) Check simplification, which eliminates redundant `check` expressions and simplifies the preconditions of the remaining `checks`. In addition, in this chapter we proved that a transformed program has the same semantics as its input program, except that the transformed program reports type errors earlier.

- Chapter 3 extended the blame prediction transformation with support for recursion. It also introduced new syntax and semantics in order to support recursive functions and mutable variables (used in chapter 4). In this chapter we observed that type inference without unification leads to *infinite types*, which need to be reduced to finite types before they can be used. As part of this process, we observed several patterns where the reduction to finite types diverged. The solution was to introduce a *widening operator* which sacrificed accuracy for guaranteed termination.

- Chapter 4 described how the blame prediction transformation is extended to cope with mutation. The blame prediction transformation as defined in chapters 2 and 3 assumes that the values stored by variables do not change. The key insight here is the addition of effects [Gifford and Lucassen, 1986] to indicate which variables could be mutated by an expression. This enabled us to ensure type tests are not lifted over expressions which mutate variables and simplify type tests for which there were no intermediate variable modifications. In this chapter we proved that the extended blame prediction transformation is again semantics-preserving. Finally, we speculated on blame prediction could support

other features of dynamically typed languages such as compound data structures, objects, non-local control flow, and debug prints.

- Chapter 5 described our prototype implementation in broad strokes. It first discussed how our prototype represents the abstract syntax tree and the various techniques used to manipulate it. Next, it described the implementation of the blame prediction transformation, guided by the stages laid out in chapter 2. Finally, the chapter showed how we found and solved two performance bottlenecks in our implementation.
- Chapter 6 defined two metrics by which we can evaluate our prototype implementation of the blame prediction transformation. The first metric measures how much earlier the blame prediction transformation is able to perform type tests. The second metric measures the reduction in the number of type tests performed. The first metric corresponds to the "root cause analysis" problem, the second metric to the "long time to crash" problem. Both metrics can be measured both in terms of the program text (static) and in terms of run-time behavior (dynamic). We applied these metrics to programs from two well-known benchmark suites, namely the Gabriel benchmarks and the Computer Language Benchmark Game (CLBG). Finally, we applied techniques from mutability testing to generate small defects in our program corpora and made some case studies where blame prediction was really able to help the programmer.
- Chapter 7 situated blame prediction among the fields of static, dynamic and post-mortem program analysis. It defined four criteria according to which the approaches were examined.

  Among the static analysis approaches are type systems, which aim to detect errors statically. These are so-called "might-fail" approaches, which prevent programs from running if there is a chance a type error might occur. In the dynamic analysis approaches, the emphasis is on dynamically verifying properties of values in the program. The various approaches attempt to improve on dynamic typing either by performing type tests earlier and/or removing statically checkable verifiable tests. Unfortunately, most approaches require programmer intervention or, in the case of preemptive type checking, significant analysis. Finally, the post-mortem approaches help the programmer in figuring out the exact point where a program "starts to go wrong", i. e. the point where an invariant is violated or a variable receives the wrong value. While these slightly alleviate a tedious time-consuming process, they still require significant programmer investment.

8.3 Limitations

In this section we discuss a number of limitations in the work presented here.

8.3.1 Explicit type tests

As we remarked in the beginning of this thesis, dynamically typed programs differ from statically typed languages in that *values* have types, not variables. Thus, a value returned from a computation or received as a function parameter may have any type. Programmers often exploit this to provide flexible interfaces in their code. To discriminate a value according to its type, dynamically typed programming languages provide type test predicates such as number? and null?. These predicates are used to form the preconditions we use in the final check expressions.

The type inference presented in chapter 2 does not take into account type tests created by the programmer. Consider the program in listing 8.1, where the function `compute-list-or-num` returns either a number or a list of numbers. The intent is to add one to each number in the result, regardless of whether it is a int or a list. To this end, the programmer has written a `number?` type test which governs the branch taken. If the `number?` test fails, `list-or-num` is not a number and therefore assumed to be a list. Vice versa, if the test succeeds then `list-or-num` is guaranteed to be a number.

```
1 (let ([list-or-num (compute-list-or-num)])
2   (if (number? list-or-num)
3     (+ list-or-num 1)
4     (map (lambda (x) (+ x 1)) list-or-num)))
```

Listing 8.1: Example of explicit type tests

Blame prediction as formulated in this dissertation cannot make use of the results of such type tests. If we apply the blame prediction transformation to this snippet, we receive the snippet in listing 8.2. Note that the `number?` test is repeated in the first branch of the conditional expression.

```
1 (let ([list-or-num (compute-list-or-num)])
2   (check [number? list-or-num ∨ list? list-or-num]
3     (if (number? list-or-num)
4       (check [number? list-or-num] (+ list-or-num 1))
5       (check [list? list-or-num] (map (lambda (x) (+ x 1)) list-or-num))))
```

Listing 8.2: Example of explicit type tests after blame prediction

Other type systems *do* allow type tests to narrow the types of variables in a flow-sensitive manner, notably Typed Racket's occurrence typing [Tobin-Hochstadt and Felleisen, 2010]. Under that type system, expressions are associated with type predicates on variables and "paths" through data structures. Using these expressions in a conditional then allows the type inferencer to adjust the type environment accordingly for each branch.

As future work we propose to incorporate features from occurrence typing, which will enable the blame prediction transformation to make use of explicit type tests.

8.3.2 Cross-module blame prediction

Almost every program written today makes use of code that is external to the program. This code could be part of the standard language distribution (`std::vector` for C++ or `os` in Python), added on to the system (downloaded from CPAN or Rubygems), or previously written by the programmer. Code is organized in several *modules*, which expose some values (functions and variables) to other modules.

Modules are a boon to the software engineering process: they can be developed and tested separately from other parts of a program, and they can subsequently be published for reuse in other programs. For an interpreted language, importing a module typically means locating the file that contains said module, evaluating the contents (which may trigger loading of other modules), and making the module's exports available to the calling program. In a compiled language however, *separate compilation* is desired, meaning that modules can be compiled separately and only combined in the linking phase. For example, a C++ program which calls a function exposed by another module results in a function call instruction whose address is filled in by the linker. This represents a trade-off in performance for modularity: if the function being called could have been inlined, this represents a missed inlining opportunity. Some of these inlining opportunities can be reclaimed by link-time optimization [Lattner and Adve, 2004; Srivastava and Wall, 1993].

For type inference there is a similar problem with modules: a module's source code might not be available. The only way to interact with such a module is to save the result of type inference when building the module somewhere, much like headers which define types and declare values in C++. Users of the module can then reference the saved types when inferring expressions that use the imported functionality.

Blame prediction as presented in this dissertation is a *whole-program analysis*. In order to support modules, the inferred types and checks for every exported function should be stored as part of the definition of a module. Unfortunately, the blame assignment presented in the blame prediction transformation is not capable of representing cross-module blame. Simply referencing the code inside foreign modules requires the programmer to understand the structure and implementation of these modules. One solution is to require contracts or type annotations for exported definitions, which enables the blame prediction transformation to "short-circuit" the blame assignment.

8.4 Avenues for Future Research

8.4.1 Other effect systems

The motivation for this dissertation was the prediction of type errors raised by primitive operations in dynamically typed languages. However, type errors are but one subset of the larger group of errors a modern dynamic programming language can

produce. We believe that blame prediction can be expanded to predict other groups of errors. We highlight a number of likely targets below.

Coercion insertion A number of type systems [Aiken et al., 1994; Peyton Jones et al., 2012; Siek and Taha, 2006], also described in chapter 7, insert run-time coercions in code when they detect a type error. These coercions are situated in the place where the type error was detected, even when the error can be detected much earlier. The check mobility and simplification stages of blame prediction can be applied to the output of these type systems, yielding ahead-of-time errors.

Null pointer detection In the software industry, a sizable amount of developer time is lost on `NullPointerExceptions` [Ayewah et al., 2008]. These exceptions occur when an object is created but one of its instance variables is not set, and these variables are accessed later. The gap between the creation of the bug and when it is encountered can span a large amount of real time. Moreover, it is an exception that is not caught by default, so long-running processes typically catch and report `NullPointerExceptions` (NPEs) at a very high level. The question programmers need to answer when solving an NPE then becomes either "when was this field set to `null`" or "when was this object created". Blame prediction can be used to record the point where objects are created and more importantly, warn in advance when a NPE will occur.

Variable tainting In the modern Internet, there are millions of servers that interact with untrusted user data. The news is full of security breaches or takeovers caused by the input of maliciously crafted data, which was used to subvert these servers and give attackers control. In fact, user data injections are the number one security vulnerability for the past few years, as determined by the Open Web Application Security Project (OWASP) Open Web Application Security Project [2013]. As an example, consider a login form where the user has to supply a username and a password. Ordinary users would supply their actual username and password, for example `"bob"` and `"secret"`.

An attacker can send a cleverly formatted username that subverts the SQL query, however. For example, supplying the following username:

```
Administrator' OR 'x' = '
```

yields a SQL query such as listing 8.3.

```
1 SELECT * FROM users WHERE username = 'Administrator' OR 'x' = ''
2                       AND password = 'aQN78goJ'
```

Listing 8.3: SQL query after injection

The condition of the query is now split in two parts, where one simply looks for a `users` record with username `Administrator`, and the other can never succeed, as `'x' != ''`. In other words, submitting the username above allows an attacker to enter the system as the `Administrator` user, without knowing the password.

There are defensive coding styles that eliminate SQL injections entirely, using so-called *prepared statements* which keep the data and the query separated. However, prepared statements are slow on the uptake among programmers as they have to go out of their way to use these prepared statements, and some online web programming tutorials still only teach the unsafe way of performing SQL queries.

Variable tainting [Clause et al., 2007] is a technique to prevent these kind of injections. It works by "tainting" variables that contain user input, and considering tainted any expression that involves tainted variables. As tainted values are passed through the program, eventually they are passed to sensitive operations, which raise an error if their input is tainted. In the example above, `hash` always produces untainted output, so `hashedPassword` is safe to use, but `username` will still result in tainting errors.

We believe blame prediction can be applied for tainting as well, where "x is not tainted" is a valid precondition.

8.4.2 Tool support for blame prediction

As we saw in chapters 2 to 4, the structure of the program influences the effectiveness of blame prediction. For example, assigning a new value to a variable x will prevent any checks on x from propagating past the assignment. If the programmer can move the assignment expression towards the beginning of the function without affecting its semantics, then blame prediction can also move checks further upwards.

Another important area that can be improved is the type tests performed at the start of functions: currently blame prediction assumes nothing about function parameters, leading to repeated checks that could be elided. For example, consider an implementation of the `vector-ref` function that performs some extra checks on its index argument before actually accessing a vector. Our type system could give it a type like

$$\Pi_{(\alpha_v,\alpha_i)}.(\mathsf{int} \mathbin{?=} \alpha_i) \cdot (\mathsf{error} \vee (\mathsf{vector} \mathbin{?=} \alpha_v) \cdot \star)$$

which always checks whether the i parameter is a number, but only checks whether v is a vector *after* the tests on the index. Contrast this with the programmer's intention, who wants to ensure the v argument is always a vector. In that case, the programmer could annotate the function and override the inferred type.

In order to augment the programmer's interaction with blame prediction, we envision tool support in the form of an Integrated Development Environment (IDE) that automatically applies blame prediction as the program is being edited. This IDE can then annotate the program with the final locations of moved check expressions and identify expressions that block check expressions from moving upwards. To solve the function communication issue, the IDE could show the inferred type for each function definition, inviting the programmer to propose a better type, leading to more checks in other parts of the program. These types could then become part of the documentation as well.

8.5 Concluding Remarks

This section concludes the dissertation. In the beginning we described two problems related to finding and debugging errors in dynamically typed programming languages. The first problem was termed the "root cause analysis" problem, which describes the process of finding the cause of a type error. The second problem was termed the "long time to crash" problem, as the programmer needs to re-run the program after every change.

We set out to develop a technique with the objective of solving these problems. This technique automatically moves type tests in dynamically typed programming languages upwards, such that they are performed as early as possible. This solves both problems effectively, as the blame prediction transformation moves type tests upwards in the program, to the program locations beyond which further evaluation *must* lead to errors. At the same time, the time to crash is reduced: performing type tests earlier also enables the program to stop much earlier.

We showed that the blame prediction transformation is effective at tackling these problems in our evaluation chapter. We defined and applied two metrics to programs from two well-known benchmark sets. The first metric measures the difference between the points where a variable is type tested in advance and where it is used in a primitive operation. This metric can be measured both statically (in terms of the program text) and dynamically (in terms of the program behavior), which corresponds to the "root cause analysis" and the "long time to crash" problems. The second metric was the ability of the blame prediction transformation to reduce the number of type tests, which also contributes to solving the "long time to crash" problem.

Our technique still has room for improvement, however. For example, in this dissertation we used a type inference system as a basis for generating `check` expressions. Other techniques such as abstract interpretation might provide a better basis for `check` introduction, yielding more precision and thus fewer type tests. Another area of improvement is better integration with the debugging process, either by helping the user locate the cause of type errors or by taking advantage of user annotations.

We are convinced that the blame prediction transformation described in this dissertation can be of benefit to developers everywhere.

REFERENCES

CHICKEN Scheme: A practical and Portable Scheme System. URL `http://www.call-cc.org`. 131

Hal Abelson, R Kent Dybvig, Christopher T Haynes, Guillermo Rozas, N I Adams, IV, Daniel Friedman, Eugene Kohlbecker, D H Bartley, R Halstead, D Oxley, Gerald J Sussman, G Brooks, Chris Hanson, Kent Pitman, and Mitchell Wand. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998. 1, 16, 114, 130, 131

Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. Orbit: An optimizing compiler for Scheme. In *SCC '86: Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 219–233, New York, New York, USA, 1986. ACM. 130

Alexander Aiken, Edward L Wimmers, and T K Lakshman. Soft typing with conditional types. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173. ACM Request Permissions, February 1994. 155, 172

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-Oriented Programming. In *OOPSLA '09: Proceedings of the 24th ACM international conference on Object oriented programming systems languages and applications*, pages 1015–1022, New York, New York, USA, 2009. ACM Press. 1, 145

Jong-hoon David An, Avik Chaudhuri, Jeffrey S Foster, and Michael Hicks. Dynamic inference of static types for Ruby. In *POPL '11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 459–472. ACM Request Permissions, January 2011. 2, 145

Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *ECOOP '05: Proceedings of the 19th European conference on Object-oriented programming*, pages 428–452, 2005. 145

Andrew W Appel, John R Ellis, and Kai Li. Real-time Concurrent Collection on Stock Multiprocessors. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on*

Programming Language Design and Implementation, pages 11–20, New York, NY, USA, 1988. ACM. 130

Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, 2008. 172

Franco Barbanera and Mariangiola Dezani-Ciancaglini. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995. 20, 154

Adam Bard. Top Github Languages of 2014, August 2014. URL `http://adambard.com/blog/top-github-languages-2014/`. 3

Michael Bayne, Richard Cook, and Michael D. Ernst. Always-available static and dynamic feedback. In *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering*, pages 521–530. ACM, 2011. 2, 155

Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *ECOOP '14: Proceedings of the 28th European conference on Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. 2

Hans-J Boehm, Alan J Demers, and Scott Shenker. Mostly parallel garbage collection. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 157–164, New York, New York, USA, 1991. ACM, ACM Press. 130

Bob Boothe. Efficient algorithms for bidirectional debugging. In *PLDI '00: Proceedings of the 21st ACM SIGPLAN conference on Programming language design and implementation*, pages 299–310. ACM, 2000. 3, 160

François Bourdoncle. Assertion-based debugging of imperative programs by abstract interpretation. In *ESEC '93: Proceedings of the 4th European Software Engineering Conference*, pages 501–516. Springer Berlin Heidelberg, 1993. 146, 161

Amy Brown and Greg Wilson. *The architecture of open source applications, volume 2*. Structure, Scale, and a Few More Fearless Hacks. Kristian Hermansen, 2012. 109

Stefan Brunthaler. Efficient Interpretation Using Quickening. In *DLS '10: Proceedings of the 6th Symposium on Dynamic Languages*, pages 1–14, New York, NY, USA, 2010. ACM. 130

Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292. ACM Request Permissions, June 1991. 154

Dave Clarke, John M Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98: Proceedings of the 13th ACM international conference on Object*

oriented programming systems languages and applications, pages 48–64. ACM, October 1998. 1

James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206. ACM Request Permissions, July 2007. 173

William D Clinger, Anne Hartheimer, and Eric Ost. Implementation strategies for continuations. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 124–131. ACM, 1988. 130

Gregory H Cooper and Shriram Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In *ESOP '06: Proceedings of the15th European Symposium on Programming*, pages 294–308, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. 162

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, New York, USA, 1977a. ACM Press. 53, 146

Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In *FDPC '77: Proceedings of the IFIP Conference on Formal Description of Programming Concepts*, pages 237–277, 1977b. 46, 49, 61, 72

Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP '92: Proceedings of the fourth international symposium on Programming Language Implementation and Logic Programming*, pages 269–295, 1992a. 53, 61, 72

Patrick Cousot and Radhia Cousot. Abstract Interpretation Frameworks. *Journal of logic and computation*, 2(4):511–547, August 1992b. 146

Douglas Crockford. *JavaScript: the good parts*. O'Reilly Media, Incorporated, 2008. 157

Christos Dimoulas and Matthias Felleisen. On contract satisfaction in a higher-order world. *Transactions on Programming Languages and Systems*, 33(5):27, November 2011. 153

Bruce F Duba, Robert Harper, and David B MacQueen. Typing First-Class Continuations in ML. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, New York, USA, 1991. ACM Press. 103

Mireille Ducassé. Coca: an automated debugger for C. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*. IEEE Computer Society Press, May 1999a. 162

Mireille Ducassé. Abstract views of Prolog executions with Opium. In Paul Brna, Ben du Boulay, and Helen Pain-Lewis, editors, *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*, pages 223–243. Ablex, 1999b. 162

Mireille Ducassé and Jacques Noyé. Logic Programming Environments: Dynamic Program Analysis and Debugging. *J. Log. Program. () 19/*, 19-20:351–384, 1994. 161

Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 263–273. ACM, August 1997. 162

Manuel Fähndrich and K Rustan M Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 302–312. ACM, November 2003. 145

Marc Feeley. SRFI 18: Multithreading support, 2001. URL `http://srfi.schemers.org/srfi-18/srfi-18.html`. 132

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. 17

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the 7th International Conference on Functional Programming*, pages 48–59, 2002. 152

David Flanagan. *JavaScript: the definitive guide*. O'Reilly Media, Inc., 2006. 1

Matthew Flatt and PLT. PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. 153

Brent Fulgham. (The Computer Language Benchmark Game), 2008. URL `http://benchmarksgame.alioth.debian.org`. 130, 140

Michael Furr, Jong-hoon David An, Jeffrey S Foster, and Michael Hicks. Static type inference for Ruby. In *SAC '09: Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 1859–1866, 2009. 145

Richard P Gabriel. *Performance and evaluation of Lisp systems*. MIT press Cambridge, 1985. 4, 130, 140

Richard P Gabriel and Larry M Masinter. Performance of Lisp systems. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and Functional Programming*, pages 123–142, 1982. 130

David Gifford and John Lucassen. Integrating functional and imperative programming. In *LFP '86: Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 28–38, 1986. 76, 151, 168

Jean-Yves Girard. Linear logic: Its syntax and semantics. *London Mathematical Society Lecture Note Series*, pages 1–42, 1995. 1

James Gosling. *The Java language specification*. Addison-Wesley Professional, 2000. 1

Neville Grech. *Preemptive type checking in dynamically typed programs*. PhD thesis, University of Southampton, faculty of physical sciences and engineering, 2013. 157, 158

Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. Cyclone: A type-safe dialect of C. *C/C++ Users Journal*, 23(1):112–139, 2005. 130

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *ECOOP '10: Proceedings of the 24th European conference on Object-oriented programming*, pages 126–150, June 2010. 2

Stefan Hanenberg and Andreas Stuchlik. Static vs. dynamic type systems: an empirical study about the relationship between type casts and development time. In *DLS '11: Proceedings of the 7th symposium on Dynamic languages*, pages 97–106, New York, New York, USA, March 2012. ACM Request Permissions. 2, 148, 155

Dries Harnie. Prototype Implementation of Blame Prediction. Technical report, September 2015. 107

Dries Harnie, Christophe Scholliers, and Wolfgang De Meuter. Ambient Contracts. *Electronic Communications of the EASST*, 28(0), March 2010.

Dries Harnie, Elisa Gonzalez Boix, Theo D'Hondt, and Wolfgang De Meuter. Programming urban-area applications. In *SAC '12: Proceedings of the 27th ACM Symposium on Applied Computing*, pages 1516–1521, 2012.

Dries Harnie, Christophe Scholliers, and Wolfgang De Meuter. Blame Prediction. *TFP '13: Trends in Functional Programming*, 8322:91–106, 2013.

Dries Harnie, Elisa Gonzalez Boix, Theo D'Hondt, and Wolfgang De Meuter. Programming Urban-Area Applications by Exploiting Public Transportation. *Transactions on Autonomous and Adaptive Systems (TAAS)*, 9(2), July 2014.

Dries Harnie, Alexander E Vapirev, Jörg Kurt Wegner, Andrey Gedich, Marvin Steijaert, Roel Wuyts, and Wolfgang De Meuter. Scaling Machine Learning for Target Prediction in Drug Discovery using Apache Spark. In *CCGRID Life '15: Proceedings of the 2015 Workshop on Clusters, Clouds and Grids for Life Sciences*, 2015.

Williams Ludwell Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *LISP and Symbolic Computation*, 2(3-4):179–396, 1989. 130

Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. Access permission contracts for scripting languages. In *POPL '12: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122. ACM, January 2012. 153

Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# programming language*. Adobe Press, 2006. 1

Fritz Henglein. Global tagging optimization by type inference. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and Functional Programming*, pages 205–215. ACM Request Permissions, January 1992. 158

Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, December 1969. 58

Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP '98: Proceedings of the 7th European Symposium on Programming*, pages 122–138, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. 1

William E Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Trans. Software Eng. ()*, SE-8(4):371–379, 1982. 9, 135

Laurent Hubert. A non-null annotation inferencer for Java bytecode. In *PASTE '08: Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 36–42, New York, New York, USA, November 2008. ACM Request Permissions. 145

John Hughes. The Design of a Pretty-printing Library. *Advanced Functional Programming*, 925(Chapter 3):53–96, 1995. 120

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua—An Extensible Extension Language. *Software: Practice and Experience*, 26(6): 635–652, June 1996. 1

Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng. ()*, 37(5):649–678, 2011. 9, 135

Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 303–310. ACM, 1991. 76

Richard Kelsey. SRFI 9: Defining record types. Technical report, 1999. 131

Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. Do static type systems improve the maintainability of software systems? An empirical study. In *ICPC '12: Proceedings of the 18th International Conference on Program Comprehension*, pages 153–162. IEEE, 2012. 2

Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM, August 1986. 131

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 26–37, New York, New York, USA, 2003. ACM Press. 121

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP '05: Proceedings of the 10th ACM SIGPLAN international conference on Functional programming*, pages 204–215, New York, New York, USA, 2005. ACM Press. 121

Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: International Symposium on Code Generation and Optimization*, pages 75–86. IEEE, 2004. 171

Daan Leijen and Erik Meijer. *Parsec: direct style monadic parser combinators for the real world*. 2002. 112

Benjamin S Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. TeJaS: Retrofitting Type Systems for JavaScript. In *DLS '13: Proceedings of the 9th symposium on Dynamic Languages*, pages 1–16, 2013. 2

Bill Lewis. Debugging Backwards in Time. In *AADEBUG '03: Fifth International Workshop on Automated Debugging*, pages 225–235, October 2003. 4, 146, 160

Mark Lutz. *Programming Python*, volume 8. O'Reilly, 1996. 1

Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *OOPSLA '89: Proceedings of the 4th ACM international conference on Object oriented programming systems languages and applications*, pages 397–406. ACM, 1989. 98

Guillaume Marceau, Gregory H Cooper, Jonathan P Spiro, Shriram Krishnamurthi, and Steven P Reiss. The design and implementation of a dataflow language for scriptable debugging. *Automated Software Engineering*, 14(1):59–86, March 2007. 146, 162

Daniel Marino and Todd Millstein. A generic type-and-effect system. In *TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation*, pages 39–50. ACM, 2009. 76, 151

Clemens Mayer, Stefan Hanenberg, Romain Robbes, Andreas Stefik, and Éric Tanter. An empirical study of the influence of static type systems on the usability of undocumented software. In *OOPSLA '12: Proceedings of the 27th ACM international conference on Object oriented programming systems languages and applications*, pages 683–702, New York, New York, USA, November 2012. ACM Request Permissions. 2

Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., January 1992. 146, 152

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978. 58, 148

Robin Milner. *The definition of standard ML: revised*. MIT press, 1997. 1

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 49–60, New York, New York, USA, September 2007a. ACM. 110

Neil Mitchell and Colin Runciman. A Supercompiler for Core Haskell. In *IFL '07: Proceedings of the 19th international workshop on Implementation and Application of Functional Languages*, pages 147–164, Berlin, Heidelberg, 2007b. Springer Berlin Heidelberg. 130

Eugenio Moggi. Computational lambda-calculus and monads. In *LICS '89: Proceedings of the fourth Annual Symposium on Logic in Computer Science*, pages 14–23, 1989. 151

Stijn Mostinckx, Jessie Dedecker, Tom Van Cutsem, and Wolfgang De Meuter. Conversations for Ambient Intelligence. In *EHOOS '05: Proceedings of the 2005 ECOOP Workshop on Exception Handling in Object-Oriented Systems*, pages 1–12, June 2005. 12

Flemming Nielson and Hanne Riis Nielson. Type and effect systems. *Recent Insight and Advances to Correct System Design*, pages 114–136, 1999. 151

Open Web Application Security Project. Top 10 security vulnerabilities of 2013, 2013. URL `https://www.owasp.org/index.php/Top_10_2013`. 172

Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(06):557–591, November 1997. 145

Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., May 1987. 42, 113

Simon Peyton Jones. Compiling Haskell by Program Transformation: A Report from the Trenches. In *ESOP '96: Proceedings of the 5th European Symposium on Programming Languages and Systems*, pages 18–44. Springer-Verlag, April 1996. 159

Simon Peyton Jones. Tackling the Awkward Squad. In CAR Hoare, Manfred Broy, and Ralf Steinbrüggen, editors, *Engineering Theories of Software Construction*, pages 47–96. 2001. 75

Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. 1, 107

Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Equality proofs and deferred type errors: A compiler pearl. In *ICFP '12: Proceedings of the 17th International Conference on Functional Programming*, pages 341–352, March 2012. 2, 155, 172

Benjamin C Pierce. Programming with intersection types, union types, and polymorphism. Technical Report 2019, Carnegie Mellon University, 1991. 20, 154

Benjamin C Pierce. *Types and programming languages*. MIT Press, 2002. 148

Benjamin C Pierce and David N Turner. Local type inference. *Transactions on Programming Languages and Systems*, 22(1):1–44, 2000. 149

Kevin Pinte, Dries Harnie, and Theo D'Hondt. Enabling Cross-Technology Mobile Applications with Network-Aware References. In *COORDINATION '11: Proceedings of the 13th International Conference on Coordination Models and Languages*, pages 142–156. Springer Berlin Heidelberg, 2011a.

Kevin Pinte, Dries Harnie, Elisa Gonzalez Boix, and Wolfgang De Meuter. Network-aware references for pervasive social applications. *PerCol '11: Proceedings of the 2011 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 537–542, 2011b.

Guillaume Pothier and Éric Tanter. Summarized Trace Indexing and Querying for Scalable Back-in-Time Debugging. In *ECOOP '11: Proceedings of the 25th European conference on Object-Oriented Programming*, pages 558–582, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. 4, 160

Guillaume Pothier, Éric Tanter, and José M Piquer. Scalable omniscient debugging. In *OOPSLA '07: Proceedings of the 22nd ACM international conference on Object oriented programming systems languages and applications*, pages 535–552, New York, New York, USA, 2007. ACM Press. 160, 161

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. In *POPL '12: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 481–494, January 2012. 157

John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965. 58

Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3-4):289–360, November 1993. 17, 114, 123

Konstantinos Sagonas and Daniel Luna. Gradual typing of erlang programs: a wrangler experience. In *Erlang '08: Proceedings of the 7th ACM SIGPLAN workshop on Erlang*, pages 73–82, New York, New York, USA, September 2008. ACM. 145

Yasushi Saito. Jockey: A User-space Library for Record-replay Debugging. In *AADEBUG '05: Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, pages 69–76, New York, NY, USA, 2005. ACM. 160

Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *ECOOP '10: Proceedings of the 24th European conference on Object-oriented programming*, pages 275–299. Springer, 2010. 13, 130

Christophe Scholliers, Dries Harnie, Éric Tanter, Wolfgang De Meuter, and Theo D'Hondt. Ambient contracts: verifying and enforcing ambient object compositions à la carte. *Personal and Ubiquitous Computing*, 15(4), April 2011.

Umesh Shankar, Kunal Talwar, Jeffrey S Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *SSYM '01: Proceedings of the 10th conference on USENIX Security Symposium*. USENIX Association, August 2001. 145

Olin Shivers. Control flow analysis in scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 164–174. ACM, July 1988. 158

Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, Carnegie Mellon University, May 1991. 158

Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *SFP '06: Proceedings of the 2006 Workshop on Scheme and Functional Programming*, pages 81–92, 2006. 2, 146, 156, 172

Jeremy G Siek and Walid Taha. Gradual Typing for Objects. In *ECOOP '07: Proceedings of the 21st European conference on Object-oriented programming*, pages 2–27, July 2007. 2, 157

Jeremy G Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *DLS '08: Proceedings of the 4th symposium on Dynamic languages*, New York, New York, USA, July 2008. ACM. 2, 157

Josep Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 44(3):12–41, June 2012. 4

Amitabh Srivastava and David W Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993. 171

Peter Steenkiste and John Hennessy. Tags and type checking in LISP: Hardware and software approaches. *ACM SIGOPS Operating Systems Review*, 21(4):50–59, 1987. 130

T Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *OOPSLA '12: Proceedings of the 27th ACM international conference on Object oriented programming systems languages and applications*, pages 943–962. ACM, November 2012. 154

Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng. ()*, SE-12(1):157–171, January 1986. 145

Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(03):245–271, July 1992. 76, 151

Peter Thiemann. Towards a type system for analyzing javascript programs. In *ESOP '05: Proceedings of the 14th European Symposium on Programming*, pages 408–422. Springer, 2005. 145

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 395–406, 2008. 101, 149

Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 117–128, September 2010. 2, 150, 170

Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In *OOPSLA '12: Proceedings of the 27th ACM international conference on Object oriented programming systems languages and applications*, pages 537–554. ACM Request Permissions, November 2012. 153

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *PLDI '11: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 132–141. ACM Request Permissions, June 2011. 2, 128, 130, 149, 159

V8 Team. V8 Benchmarks, 2011. URL `http://code.google.com/apis/v8/benchmarks.html`. 157

Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures ()*, 40(3-4):112–136, October 2014.

Oscar Waddell, Dipanwita Sarkar, and R Kent Dybvig. Fixing Letrec: A Faithful Yet Efficient Implementation of Scheme's Recursive Binding Construct. *Higher-Order and Symbolic Computation*, 18(3-4), December 2005. 42, 113

Philip Wadler. A prettier printer. *The Fun of Programming*, 2003. 120

Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic*, 4(1):1–32, January 2003. 151

WebKit Team. SunSpider Benchmarks, 2010. URL `http://www.webkit.org/perf/sunspider/sunspider.html`. 157

Andrew K Wright and Robert Cartwright. A practical soft type system for Scheme. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and Functional Programming*, pages 250–262. LFP '94: Proceedings of the 1994 ACM conference on LISP and Functional Programming, July 1994. 128, 130, 154

Tobias Wrigstad. International Workshop on Scripts to Programs, 2009. 2

Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Expositor: Scriptable time-travel debugging with first-class traces. In *ICSE '13: 35th International Conference on Software Engineering (ICSE)*, pages 352–361. IEEE, 2013. 162

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI '12: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, page 2. USENIX Association, April 2012. 14