DEPARTMENT OF COMPUTER SCIENCE

CompaSS, an Automated Proof of Incorrectness System For JSS

Ill-typed Programs Don't Evaluate Or Do Use Undesired Behaviour

Elliot Buckingham

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

Sunday 14th July, 2024

Abstract

Squeezing a dynamic language into a static type system for which a programming paradigm is established is often an insurmountable feat. In this way, those inference tools that have succeeded in extending an existing dynamic language's type safety before runtime (e.g. TypeScript [5]) are rare and highly valuable. Such static type systems are generally for proving correctness and what they fail to prove correct is assumed to not evaluate, thus introducing false positives with respect to finding bugs. However, *CompaSS* (Complement and Success System, our type system and implementation) flips this on its head. We present several related type systems and use them in an implementation for automatic incorrectness reasoning for *JSS* programs (JavaScript Success, a subset of JavaScript). We present the major algorithms involved, and examples of the application of CompaSS on JSS programs. We depart from typical error-checking solutions in seeking only to prove incorrectness. Our method maintains a guarantee that we produce no false positives in bug-finding; we argue that we get Ramsay and Walpole's Theorem of One-Sided Syntactic Soundness [16] providing us with the guarantee that all programs we prove incorrect (*ill-typed* programs) don't evaluate. We extend ill-typed to include the use of undesired behaviour (see Section 3.1.4) for JSS. We argue we keep this theorem due to the similarity of our type system to theirs and the nearly-identical reasoning behind the rules, alongside proofs of soundness and completeness (appendix A) for the constraint rules used in the implementation. We don't seek to change the philosophy of JS, and though we focus on proving the incorrectness of JSS, a *functional* subset of JS, the shift in paradigm is a result of our choice of subset of the language. Note that, given we use over-approximate reasoning, if a program cannot be ill-typed it may yet still fail to evaluate. That is to say we do not maintain the true negatives theorem as Ramsay and Walpole present for their two-sided system.

Dedication and Acknowledgements

The help and support I've had from my supervisor Dr. Steven Ramsay, Senior Lecturer and Researcher under the Programming Languages Research Group at the University of Bristol, has been unwavering and inspiring. The dedication and passion that he shows for those he supervises is as much as any student could ask for, and this project wouldn't have gotten anywhere close to what we see in this work without his unrelenting helpfulness. There was no obligation for him to spend an hour each week with me and yet we managed to overrun almost every time, so in him I also now see great patience perhaps bolstered by his unshakable enthusiasm for his field.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

Elliot Buckingham, Sunday 14th July, 2024

Contents

List of Figures

Ethics Statement

This project did not require ethical review, as determined by my supervisor, Dr. Steven Ramsay.

Supporting Technologies

The implementation of CompaSS was written mainly in JavaScript with some Python. Libraries and modules used to accomplish this are mentioned below:

- For access to the JavaScript abstract syntax tree, we used the *Acorn AST Walker*[1] library.
- Node Package Manager was used for installation of Acorn.
- NodeJS and Bun run-time environments were used for observing the behaviour of the implementation.
- Z3's Python API Z3Py was used for syntactically solving boolean formulae as part of solving constraints.

Chapter 1

Introduction

In this work, we present CompaSS (*Complement and Success System*), a constraint-based inference algorithm & implementation for showing the incorrectness of JSS (*JavaScript Success*), a functional subset of JavaScript. The system departs from typical type checkers for JavaScript (e.g. TypeScript) and indeed most static type systems imposed upon dynamic languages. Specifically, it seeks solely to prove the incorrectness of programs, instead of correctness. For example, CompaSS will automatically report that the following JSS program is guaranteed to crash or diverge at runtime:

```
1 const pair = s => t => p => p(s)(t);
2 const fst = s => t => s;
3 const snd = s => t => t;
4 const quotInner = n => d => q => {
5     return n + 1 <= 0
6         ? pair(q - 1)(n + d)
7         : quotInner(n - d)(d)(q + 1);
8 }
9 const quot = n => d => quotInner(n)(d)(0);
10 const result = quot(x => x)(12);
```

Figure 1.1: A program defining quotient division, then misusing it.

The rules for this system constitute a *success* type system. Success type systems represent an important method of tackling the undecidable nature of complete program verification. For complete verification, we're required to prove the correctness of programs which do not crash (*go wrong*) while also proving the incorrectness of those which do go wrong. The system and implementation presented here seek to achieve the latter for JSS.

Conversely, most static type systems rely on ordinary judgement semantics and do not try to provide this, so they can produce false positives when failing to prove correctness. That is, programs which may be correct are flagged as falsely erroneous to show they *may* go wrong. As such they cannot guarantee incorrectness.

A success type system will provide us with the subtle semantic change that if a term is given a type, it does not just mean that it will evaluate to a value or diverge, but also includes the possibility of going wrong and use of undesired behaviour. For example if we have a typing `(x => x)(0)` : Num it means that from its type we can say that `(x => x)(0)` either crashes, diverges, or evaluates to a Num.

With this in mind, we look at what CompaSS tells us when given the program in Figure 1.1. We can

read the output it gives us with no insight or deliberation as follows:

> Let `pair` be typed as `Comp(Ok) -> Num`.
> Let `fst` be typed as `Num -> Ok`.
> Let `snd` be typed as `Comp(Ok) -> Num`.
> Let `quotInner` be typed as `(Num -> Ok) -> (Ok -> (Num -> Comp(Ok)))`.
> Let `quot` be typed as `(Num -> Ok) -> (Num -> Comp(Ok))`.
> It follows `result` is typable as `Comp(Ok)`.
> We conclude the program is `Ill-typed`. We show this for `result`.

Note that `Comp(Ok)` is the complement of type Ok: the type of all terms which reduce to closed values. Complementation which comes from Ramsay and Walpole's paper [16] but is revisited in Chapter 2.3 tells us type inhabitants of the complemented type are exactly those terms which are not in the original type. Therefore intuit `Comp(Ok)` is a type which includes exactly no inhabitants and by the success semantics means any term of this type will diverge or crash. The output the above proof comes from is therefore telling us `result` will fail, showing the program's ill-typedness.

We argue that CompaSS automatically infers the incorrectness of JSS programs without ever ruling out any which are correct. In achieving this if a program is ill-typed it will definitely exhibit crashing, divergent or undesired behaviour. This eliminates all false positives.

The value we see in proving the incorrectness of programs relates to where automated proofs of correctness become difficult and thus in some cases we arrive at false positives; bugs are flagged in programs which do not in fact fail. Some argue that bug-finding is a significant portion of the value program verification tools offer, this value is diminished when such tools 'cry wolf' too often. An example of one such false-positive, a valid program which the type checker presents as ill-typed, is the following JSS program:

```
1 const toolId = 0;
2 const toolZero = 1;
3 const toolTwice = 2;
4 const multitool = which => {
5     const id = x => x;
6     const zero = 0;
7     const twice = f => x => f(f(x));
8     return which <= 0 ? id : which - 1 <= 0 ? zero : twice;
9 }
10 const useId = multitool(toolId)(0);
11 const useZero = 0 + multitool(toolZero);
12 const useTwice = (multitool(toolTwice)(x => x - 1))(2);
```

Perhaps this example does not represent excellent programming practice, but it allows us to demonstrate that some correctness type systems cannot capture the full flexibility of a dynamic language.
When this program is run with NodeJS, it terminates and `useId`, `useZero` and `useTwice` evaluate to zero without using undesired behaviour. Therefore it does not fail, and yet TypeScript highlights errors where the program can fail. These errors are all related to the union return-type `0 | ((x: any) => any)` it assigns the `multitool` function. Of course, TypeScript is not wrong about this, the function can indeed return the number zero or a function (the type of which it does not discern), but because TypeScript cannot narrow the type when the applications of `multitool` occur on lines 10, 11 and 12 it cannot absolutely show correctness. Thus we get our errors in a program which in fact does not fail.
When we type-check this program with CompaSS, we get back that the program is `Untypable`. CompaSS is telling us it cannot prove incorrectness in our program, and therefore we do not know anything; perhaps it will evaluate, or perhaps it will fail[1]. Importantly this means we do **not** return a false positive, in this case or indeed for any possible JSS program through maintenance of Ramsay and Walpole's theorem of one-sided syntactic soundness [16]. However, even with this guarantee we get no guarantees about what is not ill-typed, e.g. there are certainly programs which fail which come out as `Untypable`. Therefore false negatives do exist in CompaSS because otherwise we would have achieved complete program verification.

One may then realise it's trivial to uphold the guarantee that ill-typed programs don't evaluate if we just say every JSS program is `Untypable` and call it a day. However, this is not at all what CompaSS is doing. It is actually trying its hardest to prove the incorrectness of everything it is given, using *constraint-based* rules covered in Section 3.3.1. We start with a second example wherein TypeScript and CompaSS conclude similar things, and in this case, CompaSS gives a stronger[1] typing than TypeScript[2].

```
1 const pair = s => t => p => p(s)(t);
2 const confusedPair = pair(0)(0)(0);
```

First of all, this program does in fact go wrong. When run with NodeJS we get a crash and a TypeError telling us `p is not a function`. This is because on evaluation zero is passed into the partially applied pair instead of a binary function like `s => t => s` which would get the first item in such a pair. Yet, for this program TypeScript does not actually detect anything is wrong. We get no statically-detected errors for this program which goes wrong, which we deem a *false negative*. This is because TypeScript's automatic type inference provides us with no type safety in the type signature for `pair` it creates: `(s: any) => (t: any) => (p: any) => any`. TypeScript does not know what the argument type ought to be, cannot rule out `0`, and thus it can proceed no further.

CompaSS, however, presents us with the following information:

```
                Solution
          pair: Num -> (Num -> (Num -> Comp(Ok))
confusedPair: Comp(Ok)
                Ill-typed and fails at: confusedPair
```

Which is interpreted by the programmer as:

> Let `pair` be typed as `Num -> (Num -> (Num -> Comp(Ok)))`.
>
> It follows `confusedPair` is of type `Comp(Ok)`.
>
> We conclude the program is `Ill-typed`. We show this for `confusedPair`.

This is the type environment it produces (the raw output of CompaSS), and then framed as a proof. Given that when CompaSS tells us a program is ill-typed it means it will fail (crash, diverge or use undesired behaviour), we can see it successfully shows the program is incorrect. We also get to see the statements which make the program go wrong. One-sided syntactic soundness is the crux of what makes CompaSS and our rules so enticing: that we can make this guarantee of incorrectness of exactly what we can prove is ill-typed.

In the following examples, wherever the usage of CompaSS is discussed "the programmer" refers to someone giving it their JSS program then receiving a response indicating it is incorrect plus the type environment of the program, or that proof of the former failed and thus nothing can be said for its incorrectness.

1.1 Examples

We show small JSS programs on which we run CompaSS and discuss its output. Details on the exact process by which these programs are proven to fail will be formally presented in Chapter 3: Implementation. Here we offer some intuition on what the system is doing. The programmer would be able to find out which one of their programs would certainly need to be fixed given they had written the following examples thanks to CompaSS.

1.1.1 Use with Care

To demonstrate more clearly the effect of our method of reasoning about functions, we investigate whether the following program can be ill-typed:

[1] By stronger we just mean in terms of showing incorrectness. It just so happens that CompaSS returns more type information and avoids a false negative in this example.

[2] For this unannotated JS; JSS does not contain type annotations.

```
1 const id = x => x;
2 const explode = x => id - 1;
3 const zero = id(0);
```

We get the following output:

```
        id:  Untypable
   explode:  Untypable
      zero:  Untypable
        Inconclusive
```

To understand this output, we note first that all the top-level variables in the program appear. This is because we're presented with the program type Δ, a type environment. A type environment is suitable for a program type as it allows us to map the top-level statements to types to discern when the program fails, and by which variable. We weren't able to prove its incorrectness since none of the top-level statements had type Ok^c (represented by CompaSS as `Comp(Ok)`, the type of ill-typedness). In actuality, the program succeeds; we don't ever run `explode`, so it is accurate that CompaSS cannot show it fails.
Note that even though we get the guarantee that all of what is ill-typed will go wrong [16], we do not get that all of what is not ill-typed evaluates. However, to uphold our guarantee we design a system which never types a program that will possibly terminate in a value.

1.1.2 Earliest Proof of Failure

Programs that CompaSS can show are ill-typed may fail anywhere, not just at the end of the program. This is what we see here in this example:

```
1 const earlyFail = (x => 0)(0)(0);
2 const two = 1 + 1;
3 two + 1;
```

```
          Solution
earlyFail:  Comp(Ok)
       two:  Ok
     eval#0:  Ok
          Ill-typed and fails at:  earlyFail
```

The program is shown to fail on the first line. If the program were run we would get an error suggesting `0(0)` caused our failure (since this is what the assignment on line one reduces to at runtime). When run such an application raises a TypeError in NodeJS; the remaining lines do not run. Yet they are typed as Ok since they themselves were not dependent on the failure of earlier lines. Note that the only guarantee that CompaSS' type system gives us is determining with absolute certainty that all ill-typed programs will crash, diverge or exhibit undesired behaviour. We do *not* get a guarantee that it will find all the definitions or evaluations on which the program fails, nor that all such definitions it proves incorrect are in fact definitely wrong. But if a program is ill-typed we do get the guarantee that at least one of the variables in the program type which are typed Ok^c cause the program to fail. In practise, as we discuss in Chapter 4, it usually discerns the failing lines of the program correctly; in fact we have not found a case where it does not just via testing. An example in support of this follows.

1.1.3 Where is my Failure

CompaSS is able to discern multiple solutions if the programmer makes multiple mistakes the type system can prove to fail. We still get the guarantee that at least one of these is always a true positive, and as such still argue that we maintain one-sided syntactic soundness just for whole programs. This example shows the common case where all lines that are shown to fail are true positives:

```
1 const id = x => x;
2 const crash = id - id;
```

```
3 const pred = n => n - 1;
4 const bang = pred(id);
5 const succ = m => m + 1;
6 const wallop = id(crash);
```

```
                    Solution
                id:  Comp(Ok) -> Num
             crash:  Comp(Ok)
              pred:  Comp(Ok) -> Num
              bang:  Ok
              succ:  (Num -> Num) -> Num
            wallop:  Ok
                    Solution
                id:  Comp(Ok) -> Num
             crash:  Comp(Ok)
              pred:  Comp(Num) -> (Num -> Num)
              bang:  Ok
              succ:  Comp(Ok) -> (Num -> Num)
            wallop:  Comp(Ok)
                    Solution
                id:  Comp(Ok) -> Num
             crash:  Comp(Ok)
              pred:  (Comp(Ok) -> Num) -> Comp(Ok)
              bang:  Comp(Ok)
              succ:  Comp(Ok) -> (Num -> Num)
            wallop:  Comp(Ok)
                    Ill-typed and fails at:  crash,bang,wallop
```

This is a large output so we should pay special attention to anything in the program type that appears as `Comp(Ok)`. But note if it appears in the antecedent of a function it does not indicate failure, which we explain in Section 3.2.3 in deriving (App2). Note that CompaSS only generates a new solution if it can deduce a new variable has the type `Comp(Ok)`. This means we limit the number of iterations for which we solve the constraints by the number of top-level definitions in the program, as we can never prove more failures in the program than there are variables in the program type.

1.1.4 Wrong Function Defintions

Assume the programmer writes a function to sum two pairs element-wise and return the resulting pair. Their implementation is as follows, but before giving to CompaSS they do not spot the issue on line eight:

```
1 const fst = s => t => s;
2 const snd = s => t => t;
3 const pair = s => t => p => p(s)(t);
4 const p1 = pair(0)(1);
5 const p2 = pair(2)(3);
6 const zipSumPairs = pair1 => pair2 => {
7     const e1 = pair1(fst) + pair2(fst);
8     const e2 = pair1(snd) + snd(pair2);
9     return pair(e1)(e2);
10 }
11 const sumPair = zipSumPairs(p1)(p2);
```

```
                    Solution
                fst:  Ok -> Ok
                snd:  Ok -> (Comp((Num -> (Num -> Num))) -> Ok)
               pair:  ((Num -> Num) -> (Num -> Num)) -> Ok
                 p1:  Ok
                 p2:  Ok
        zipSumPairs:  Ok -> (Ok -> Comp(Ok))
            sumPair:  Comp(Ok)
                    Ill-typed
```

The mistake causes a number to be added to a partially applied function (the astute reader may notice `snd(pair2)` reduces to id), which comes under our definition of undesired behaviour (see Section 3.1.4) and thus is provably wrong. CompaSS tells us the program is ill-typed thus it is proven to fail, and our first ill-typed variable is actually `sumPair`, the variable below `zipSumPairs`. This might seem odd at first because we intuit the programmer made an error in the function, not the line below it. But recall CompaSS is only able to prove incorrectness if it certainly happens in all cases (no false positives). What this means is a function with a body that fails must be *applied* before it can be proven to be incorrect, even if it has an application inside its body. This mirrors the behaviour of a JS interpreter; the body of any function, given it is syntactically correct, cannot cause a failure until we execute it because it isn't interpreted until then. Therefore in general we find if the programmer does not use the function, it has no bearing on the program or whether it fails. Another perspective is that (arrow) functions are values and so cannot fail.

1.1.5 Eager yet Delayed

JS does not support lazy evaluation like for example Haskell, but it does support ways of delaying execution e.g. *thunks* as Alves et al. outline in their section on lazy evaluation [7]. Although thunks (nullary arrow functions in JS) are not possible to write in JSS since functions must be unary, we must note that all arrow functions we create are only evaluated when run, thusly they do not cause failures if they are never run. As such this constitutes delayed delayed execution. Given that the following program doesn't fail, we see CompaSS is not able to type it as incorrect.

```
1 const delayed = go => {
2     (f => x => f(x)(f(x)))(x => 0)(0);
3     return go;
4 }
5 const normalRunThis = x => 0;
```

```
                              delayed:  Untypable
                        normalRunThis:  Untypable
                                      Inconclusive
```

Yet when we run our delayed routine, we see that CompaSS shows it's ill-typed. This means the program fails. In this case, we can run the program and it crashes.

```
1 const delayed = go => {
2     (f => x => f(x)(f(x)))(x => 0)(0);
3     return go;
4 }
5 const normalRunThis = x => 0;
6 const shouldFail = delayed(0);
```

```
Solution
delayed:  Ok -> Comp(Ok)
normalRunThis:  Comp(Ok) -> Num
shouldFail:  Comp(Ok)
Ill-typed
```

This is what we expect given what we know about delayed execution. CompaSS of course respects this given we see a failure is only provable when such incorrect code is delayed and then run.

1.1.6 Recursive Typability

CompaSS can prove the incorrectness of some programs with recursive functions thanks to its program rules that introduce a free function with the name of the definition in the assumptions as we type the body.

```
1 const pair = m => n => p => p(m)(n);
2 const divide = n => d => q => {
3     const r = n - d;
4     return r + 1 <= 0 ? pair(q)(n) : divide(r)(d)(q + 1);
5 }

7 const badResult = divide(x => x)(10)(0);
```

```
Solution
pair:  Comp(Ok) -> (Num -> (Num -> Num))
divide:  (Comp(Ok) -> Num) -> (Ok -> (Ok -> Comp(Ok)))
badResult:  Comp(Ok)
Ill-typed and fails at:  badResult
```

CompaSS uses similar reasoning to the fixpoint rules in Ramsay and Walpole type systems [16]. This simple feature gives us type assignment with recursion and thus gives us enough expressibility to extract type information from programs which make use of this feature of JS.

1.1.7 Inconclusivity

Here we discuss what it means for a definition or statement to be untypable, which is just if we cannot ill-type the whole program.

```
1 const id = x => x;
```

```
id:  Untypable
Inconclusive
```

A program is untypable just if CompaSS cannot prove it fails. But importantly because we argue we maintain soundness and completeness from Ramsay and Walpole's [16] One-Sided Success System with Complements, and so as in their system, we *only* get that ill-typed programs are guaranteed to go wrong, diverge or use undesired behaviour. We get no guarantees about programs which we cannot show are ill-typed, which are just those that are untypable. This is to say untypable programs may still yet go wrong or diverge.

Further interesting examples of CompaSS are discussed in the evaluation Chapter 4.

1.2 Challenges, Constraint-based Rules

The major hurdles of this project include adapting Ramsay and Walpole's One-Sided Type Success Type System for their PCF-like functional language, to the multi-paradigm language of JS. JSS can however take advantage of the fact that functions are first-class objects [3] in JS, so JSS may implement the functional paradigm. Encompassing elements of JS that live outside of PCF-like languages will present a challenge since their effect on the type of a term or program needs to be reasoned about carefully.
Additionally, we must produce a new set of inference rules for constraint generation and type inference. These rules will include ways to generate *constraints*, which are for our purposes systems of syntactic type equalities combined with logical conjunction and disjunction. Constraint rules lend themselves to automated and algorithmic derivations as syntax-direction using a constraint does not come at the price of expressiveness. Specifically, where one may have to otherwise cut the number of rules down in order to make at most one for every grammar shape in an 'ordinary' non-constraint system, with constraint rules we merge multiple non-syntax directed rules together through disjunction on the constraints they ask of the type variables. The resulting type derivations under such a system do not fail, and they do not seek to type a term but instead construct a system of constraints all of which are with respect to syntactic equality. We then check if a satisfying assignment to these constraints exists under the restriction that at least one term in the program fails. This process will be elucidated in Chapter 3.3 where we cover how we derive our constraint rules and why they appear as they do.

1.3 Goals

The goals of this project are to demonstrate the possibility for a success type system to apply to JavaScript by defining a one-sided incorrectness type system for a subset of the language, and given success semantics extended from Ramsay and Walpole's system [16] argue it maintains their theorem of one-sided syntactic soundness. This means the type system never produces false positives in finding bugs, and thus provides more value than many major static non-success over-approximate type systems with respect to showing incorrectness for JSS. Additionally, we set the goal of producing an implementation (CompaSS) of these rules in a JavaScript program through construction of a constraint-based system which we shall prove to be sound and complete with respect to the one-sided success system. We also aim to show its utility when applied to non-trivial programs which fail. Finally, we produce two main extensions to these goals. Firstly, we add composition to the grammar, the type systems and the implementation letting us prove multi-line programs ill-typed. Secondly, we augment the constraint solver with the ability to produce many ill-typings at once delivering to the programmer much more information from the constraints generated in CompaSS. The latter is an extension of simply deciding whether the program fails at all or is untypable.

1.4 Contributions

This project contributes the following to the fields of programming languages research and type systems:

- A formal specification for a functional language subset of JavaScript (JS) named JSS. Importantly this means JSS is real runnable JS code.
- A success type system to prove the incorrectness of JSS programs.
- A set of constraint-based type inference rules which mirror those above, syntax-directed on JSS.
- A proof of the equivalence of the syntax-directed system to the non-syntax-directed system of rules via soundness and completeness. This may be viewed in Appendix A.
- An algorithmic implementation (CompaSS) of the success type system for whole-program incorrectness reasoning, using constraint-based rules.
- Analysis of results of the algorithm applied to various interesting programs, demonstrating and reasoning about its strengths and limitations.
- Presentation of ways our work may be extended: discussion of adding to the grammar for JSS to ill-type more of JS, and increasing CompaSS' performance via pruning of superfluous ordinary rules.

1.5 Outline

A discussion of related works is given in Chapter 2 and relevant background for the best understanding of subsequent chapters. Chapter 3 presents JSS and the rationale behind what we choose to include in it. We discuss what we define as a failure by extending typical success semantics with undesirable behaviour in Section 3.1.4. We then depict the typing rules which apply to JSS and how they're derived from our two-sided type system for JSS, as well as their similarity to Ramsay and Walpole's[16] One-sided Success System. In Section 3.2 we derive our one-sided rules from Ramsay and Walpole's two-sided system and then proceed to convert these into constraint rules in Section 3.3, with clear reasoning for changes to the type system we incorporate.

Implementation of CompaSS is explored in Section 3.5 including algorithmic representations of the type system, efficient constraint resolution and generating multiple solutions. Our evaluation (Chapter 4) finds the limits of CompaSS and critically discusses what it is best at and what it cannot express. We then seek to prove soundness and completeness between the one-sided non-constraint system and the syntax-directed system, aiming to build confidence in the reader that our final system is very close that of Ramsay and Walpole's system and as such arguably inherits most of the vital properties of their system. Chief among these properties is that of one-sided syntactic soundness. To conclude we explore possible future work in Chapter 5.2 which involves postulating possible extensions to the grammar for JSS, and improving the runtime of CompaSS.

Chapter 2

Background

In this chapter, we provide relevant context for the different aspects contributing to our rules and CompaSS, and bring up some related works. Pre-requisites covered include Ramsay and Walpole's two-sided type system, their one-sided version, and converting human-usable rules into syntax-direct rules which are sound and complete wrt. to the original rules (see Appendix A for corresponding proof). Note when explaining rules, for brevity we say that for some term M and type A that "M evaluates to an A" or "M is an A" to mean "M evaluates to a value V which is of type A".

2.1 PCF

PCF, or Programming Computable Functions, is one of the core programming languages used to investigate theory questions central to programming languages and type systems. Here we introduce the understanding of Ramsay and Walpole's version of PCF[16] so that we may reference it in designing JSS.

Definition 2.1.1 (Language) *The denumerable set of term variables x, y, z are assumed. This is a simple functional programming language whos values, usually V, W, and terms, M, N, P, Q we define with the following grammar[16]:*

$$\begin{array}{rrll} \text{(Values)} & V, W & ::= & x \mid \mathsf{succ}^n(\mathsf{zero}) \mid (V, W) \mid \mathsf{fix} f(x).M \\ \text{(Terms)} & M, N, P, Q & ::= & \mathsf{zero} \mid \mathsf{succ}(M) \mid \mathsf{pred}(M) \mid \mathsf{ifz}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P \mid \\ & & & \quad x \mid M\ N \mid \mathsf{fix} f(x).M \mid (M, N) \mid \mathsf{let}\ (x, y) = M\ \mathsf{in}\ N \end{array}$$

Terms are considered up to renaming bound variables and usual conventions are adopted in their treatment (including, for example, capture-avoiding substitution). A closed term, a member of Λ_0, is one without free variables. Therefore all of the variables x, y, z that appear in the term are bound. Otherwise, a term is open. The language that CompaSS and its rules operate on (JSS) is fundamentally similar to this PCF-like language. It's defined to incorporate changes needed to represent valid JS, and allow it to type more programs e.g. multi-line programs as seen in the introduction. The rationale for its grammar is presented in Chapter 3.1.

For Ramsay and Walpole's PCF-like language, we define abbreviations for numerals so that we can write $\mathsf{succ}^n(\mathsf{zero})$ as $\underline{n}$. $\lambda x.M$ is our abbreviation for $\mathsf{fix} f(x).M$ when f does not occur free in M. Thereby we essentially have an applied CBV λ-Calculus with fixpoints.

Definition 2.1.2 (Evaluation contexts) *Evaluation contexts for this version of PCF are defined by the following grammar*

$$\begin{array}{rcl} \mathcal{E}, \mathcal{F} & ::= & \Box \mid (\mathsf{fix} f(x).M)\mathcal{E} \mid \mathcal{E}N \mid \mathsf{succ}(\mathcal{E}) \mid \mathsf{pred}(\mathcal{E}) \mid \\ & & \quad (\mathcal{E}, M) \mid (V, \mathcal{E}) \mid let(x, y) = \mathcal{E}\ \mathsf{in}\ N \mid \mathsf{ifz}\ \mathcal{E}\ \mathsf{then}\ N\ \mathsf{else}\ P \end{array}$$

Provided a context $\mathcal{E}$, we write $\mathcal{E}[M]$ for the term with the hole $\Box$ replaced with M. Single-step reduction is written $M \rhd N$ is the binary relation on terms which may be open. It is defined by the following

schema under evaluation contexts:

$$
\begin{array}{lrcl}
(\mathsf{IfZ}) & \mathsf{ifz}\,\underline{0}\,\mathsf{then}\,N\,\mathsf{else}\,P & \rhd & N \\
(\mathsf{IfS}) & \mathsf{ifz}\,\mathsf{succ}(\underline{n})\,\mathsf{then}\,N\,\mathsf{else}\,P & \rhd & P \\
(\mathsf{Let}) & \mathsf{let}\,(x,y) = (V,W)\,\mathsf{in}\,M & \rhd & M[V/x, W/y] \\
(\mathsf{PredZ}) & \mathsf{pred}(\underline{0}) & \rhd & \underline{0} \\
(\mathsf{PredS}) & \mathsf{pred}(\underline{n+1}) & \rhd & \underline{n} \\
(\mathsf{Fix}\beta) & (\mathsf{fix}\,f(x).\,M)\,V & \rhd & M[V/x, \mathsf{fix}\,f(x).\,M/f]
\end{array}
$$

Redexes appear on the left-hand side, and we write $M \rhd^* N$ for M reaching form N in some number of steps which can include zero. A term M that cannot step is said to be in *normal form*, and one with no normal form is said to *diverge*. We write $M \Downarrow V$ iff $M \rhd^* N$ where $M \in \Lambda_0$, and we say M *evaluates*. A term is *stuck* just if it is normal form but the normal form is not a value. Thus all terms that *go wrong* or *get stuck* reduce to a stuck term. The rules we devise for CompaSS are for proving the incorrectness of such terms and programs which diverge, get stuck when executed or use undesired behaviour which we discuss in Section 2.5 then define concretely by extending the success semantics in Section 3.1.4.

2.2 The Two-Sided System

Ramsay and Walpole introduce a two-sided type system[16] which is concerned with typing and refutation (ill-typing). Refutation that a term inhabits a type in this system is not merely failing to create a type derivation with the given rules, but in fact it is creating a type derivation which shows that a term cannot possibly have a certain type. In this case, the assignment appears on the left of the turnstile with no consequents on the right. What allows us to do this is the introduction of $A \succ\!\!- B$ which we read as "*only-to*"; the *necesseity arrow*. The meaning of this is different from $\to$, or the typical *sufficiency arrow* since, if a term has type $A \to B$ and we give it a term of type A we can guarantee it reduces to a term of type B. Conversely, if a term has type $A \succ\!\!- B$, and it reduces to a B we can guarantee it was given an A. This is a subtle notion, but it gives us something that sufficiency does not represent; it allows us to infer the type of term that was provided to a function if we know the type of its output.

Definition 2.2.1 (Two-sided types) *The two-sided types are represented by type variables A, B, and are defined by the following grammar:*

$$A, B \;::=\; \mathsf{Nat} \mid A \times B \mid A \to B \mid A \succ\!\!- B$$

Note it is also possible for a function to be typed as necessary and typed as sufficient; this represents the tighter restriction that a B is given just if an A is provided.

Pure Calculus Rules

$$(\mathsf{Id})\ \frac{}{\Gamma,\, x : A \vdash x : A, \Delta}$$

$$(\mathsf{AppL})\ \frac{\Gamma \vdash M : B \succ\!\!- A,\ \Delta \quad \Gamma,\, N : B \vdash \Delta}{\Gamma,\, MN : A \vdash \Delta} \qquad (\mathsf{AppR})\ \frac{\Gamma \vdash M : B \to A,\ \Delta \quad \Gamma \vdash N : B,\ \Delta}{\Gamma \vdash MN : A,\ \Delta}$$

$$(\mathsf{FixsR})\ \frac{\Gamma,\, f : A \to B,\, x : A \vdash M : B,\ \Delta}{\Gamma \vdash \mathsf{fix}\,f(x).\,M : A \to B,\ \Delta} \qquad (\mathsf{FixnR})\ \frac{\Gamma,\, f : A \succ\!\!- B,\, M : B \vdash x : A,\ \Delta}{\Gamma \vdash \mathsf{fix}\,f(x).\,M : A \succ\!\!- B, \Delta}$$

Figure 2.1: Pure calculus rules for the two-sided system.

.

CompaSS and its rules rely on a version of these types with a handful of changes, and the success semantics which we discuss in Section 2.4.

A typical one-sided system like simply typed λ-Calculus requires typing judgements take the form of $\Gamma \vdash M : A$, with exclusively variable typings in Γ and exactly one term typing $M : A$ on the right. Ramsay and Walpole propose their type system which allows for term typings to appear on the left, and for multiple typings to appear on the right. This gives typing judgements the shape $\Gamma \vdash \Delta$ where Δ is a set of typings understood disjunctively. Their paper however primarily focuses on judgements where there are just one or zero typings on the right; still allowing us refutation by the empty conclusion.

To demonstrate, consider a judgement $\Gamma \vdash M : B,\ N : C$. What this says in Ramsay and Walpole's system is that 'if Γ, then $M : B$ or $N : C$, or both'. A more interesting case and one used throughout their rules is $M : B,\ N : C \vdash$ which is understood as 'if $M : B$ and $N : C$ then $\bigvee \emptyset$ (false)', where disjunction over nothing is false. This means it cannot be the case that $M : B$ and $N : C$, so either $M \not: B$ or $N \not: C$, or both. Sensibly, this is utilised to express refutation. An example of such a rule that uses a combination of affirmation and refutation in the pure calculus is the rule (AppL).

$$(\mathsf{AppL})\ \frac{\Gamma \vdash M : A \rightarrowtail B, \Delta \quad \Gamma, N : A \vdash \Delta}{\Gamma, (MN) : B \vdash \Delta}$$

Unpacking this rule we read that M applied to N is refuted if we show that we can affirm that M only gives us a B when provided an A, while refuting that N is an A at all. Thereby we show that M does not get what it needs to evaluate to a B when applied to N.

Another example from the pure calculus which highlights the semantics of the necessity arrow is (AbnR).

$$(\mathsf{AbnR})\ \frac{\Gamma, M : B \vdash x : A, \Delta}{\Gamma \vdash (\lambda x.M) : A \rightarrowtail B, \Delta}$$

This rule is the necessity version of (AbsR) in the full two-sided rule system (fig. 2.2); it allows us to conclude that certain functions are in fact necessity functions. Contrary to a typical abstraction typing rule, one does not have to show that given the variable x is of type A that M is a B, but we must show the converse: assume that M is a B and proceed to show we can conclude that x was an A. This rule exists so that we may use the new semantics in derivations.

Furthermore, the rules they present allow for formal derivation of affirmation with judgements such as:

$$\mathsf{succ}(\mathsf{succ}(z)) : \mathsf{Nat},\ \ \mathsf{ifz}\ y\ \mathsf{then}\ \underline{0}\ \mathsf{else}\ \underline{1} : \mathsf{Nat} \vdash \underline{\mathsf{add}}(y, z) : \mathsf{Nat}$$

Note first $\vdash \underline{\mathsf{add}} : (\mathsf{Nat} \times \mathsf{Nat}) \rightarrow \mathsf{Nat}$. We suppose the assumptions, and intuitively we can see that given the constant $\vdash \mathsf{succ} : \mathsf{Nat} \rightarrowtail \mathsf{Nat}$, that $\vdash z : \mathsf{Nat}$. Furthermore, as the conditional of the language is typed $\vdash \mathsf{ifz} : \mathsf{Nat} \rightarrowtail A \rightarrow A \rightarrow A$, it requires that its condition is a Nat. We then have $\vdash y : \mathsf{Nat}$ as well. Thus this judgement holds because y and z are sufficient for $\underline{add}$ to evaluate; $\vdash (y, z) : \mathsf{Nat} \times \mathsf{Nat}$.

It is however critical to note that since their PCF-like language, and indeed our JSS, is call-by-value there is a specific meaning to term typings in the type environment. On the left, a term typing $M : A$ tells us that M reaches a value with type A. However, there is a difference on the right where terms are allowed to diverge or evaluate (i.e. M evaluates to value $V : A$, or $M \Uparrow$).

The rules for the system of two-sided type inference are presented in Figure 2.2, in which there are most notably *left* and *right* flavours of rule. We will explain their purpose here, and give formal definitions of the typing formulae.

Definition 2.2.2 (Type Assignment) *A typing formula, or just typing, appears as $M : A$ for some term M and type A. The subject of such a formula is M and a typing judgement is a pair of finite sets of typings; $\Gamma \vdash \Delta$. Such a judgement is provable just if we can construct a typing derivation (or proof) for it using the rules in Figure 2.2. Additional requirements are placed on the variables in Γ and Δ when applying the rules (AbsR), (AbnR), (LetR), (LetL1), (LetL2) and (FixR). These rules require that $x \notin \mathsf{dom}\Gamma$ and $x \notin \mathsf{dom}\Delta$.*

Definition 2.2.3 (Disjointness) *For the two-sided systems, it is the case that $A \,||\, B$ just if either i) one is a Nat and the other is not, or ii) one is of shape $A \rightarrow B$ and the other is not, or finally iii) one is of shape $A \times B$ and the other is not.*

Left rules we can loosely think of as 'refutation rules' in that they help us refute whether a term has a certain type. The rule for disjointness (Dis), we may read as: "to refute that M is an A, it is sufficient to

Structural Rules

$$(\mathsf{Id})\ \frac{}{\Gamma,\ x : A \vdash x : A,\ \Delta}$$

Right Typing Rules

$$(\mathsf{ZeroR})\ \frac{}{\Gamma \vdash \mathtt{zero} : \mathsf{Nat},\ \Delta} \qquad (\mathsf{cR})\ \frac{\Gamma \vdash M : \mathsf{Nat},\ \Delta}{\Gamma \vdash c(M) : \mathsf{Nat},\ \Delta}\ c \in \left\{ \begin{array}{c} \mathtt{succ} \\ \mathsf{pred} \end{array} \right\}$$

$$(\mathsf{FixsR})\ \frac{\Gamma,\ f : A \to B,\ x : A \vdash M : B,\ \Delta}{\Gamma \vdash \mathsf{fix}\, f(x).\, M : A \to B,\ \Delta}\ (\mathsf{FixnR})\ \frac{\Gamma,\ f : A \rightarrowtail B,\ M : B \vdash x : A,\ \Delta}{\Gamma \vdash \mathsf{fix}\, f(x).\, M : A \rightarrowtail B,\ \Delta}$$

$$(\mathsf{AppR})\ \frac{\Gamma \vdash M : B \to A,\ \Delta \quad \Gamma \vdash N : B,\ \Delta}{\Gamma \vdash M\, N : A,\ \Delta}\ (\mathsf{PairR})\ \frac{\Gamma \vdash M : A,\ \Delta \quad \Gamma \vdash N : B,\ \Delta}{\Gamma \vdash (M,\ N) : A \times B,\ \Delta}$$

$$(\mathsf{LetR})\ \frac{\begin{array}{c}\Gamma \vdash M : B \times C,\ \Delta \\ \Gamma,\ x : B,\ y : C \vdash N : A,\ \Delta\end{array}}{\Gamma \vdash \mathsf{let}\,(x, y) = M\,\mathsf{in}\,N : A,\ \Delta} \qquad (\mathsf{IfzR})\ \frac{\begin{array}{c}\Gamma \vdash P : A,\ \Delta \\ \Gamma \vdash M : \mathsf{Nat},\ \Delta \quad \Gamma \vdash N : A,\ \Delta\end{array}}{\Gamma \vdash \mathsf{ifz}\, M\, \mathsf{then}\, N\, \mathsf{else}\, P : A,\ \Delta}$$

Left Typing Rules

$$(\mathsf{Dis})\ \frac{\Gamma \vdash M : B,\ \Delta}{\Gamma,\ M : A \vdash \Delta}\ A \parallel B \qquad (\mathsf{PairL})\ \frac{\Gamma,\ M_i : A_i \vdash \Delta}{\Gamma,\ (M_1,\ M_2) : A_1 \times A_2 \vdash \Delta}\ i \in \{1, 2\}$$

$$(\mathsf{AppL})\ \frac{\Gamma \vdash M : B \rightarrowtail A,\ \Delta \quad \Gamma,\ N : B \vdash \Delta}{\Gamma,\ M\, N : A \vdash \Delta}\ (\mathsf{cL})\ \frac{\Gamma,\ M : \mathsf{Nat} \vdash \Delta}{\Gamma,\ c(M) : \mathsf{Nat} \vdash \Delta}\ c \in \left\{ \begin{array}{c} \mathtt{succ} \\ \mathsf{pred} \end{array} \right\}$$

$$(\mathsf{IfzL1})\ \frac{\Gamma,\ M : \mathsf{Nat} \vdash \Delta}{\Gamma,\ \mathsf{ifz}\, M\, \mathsf{then}\, N\, \mathsf{else}\, P : A \vdash \Delta} \qquad (\mathsf{IfzL2})\ \frac{\Gamma,\ N : A \vdash \Delta \quad \Gamma,\ P : A \vdash \Delta}{\Gamma,\ \mathsf{ifz}\, M\, \mathsf{then}\, N\, \mathsf{else}\, P : A \vdash \Delta}$$

$$(\mathsf{LetL1})\ \frac{\Gamma,\ N : A \vdash \Delta}{\Gamma,\ \mathsf{let}\,(x, y) = M\,\mathsf{in}\,N : A \vdash \Delta} \qquad (\mathsf{LetL2})\ \frac{\begin{array}{c}\Gamma,\ M : B_1 \times B_2 \vdash \Delta \\ \Gamma,\ N : A \vdash x_i : B_i,\ \Delta\ (\forall i)\end{array}}{\Gamma,\ \mathsf{let}\,(x_1, x_2) = M\,\mathsf{in}\,N : A \vdash \Delta}$$

Figure 2.2: Two-sided type assignment for Ramsay and Walpole's [16] PCF-like language.

affirm it is a B or that it diverges", where B is disjoint from A. We will discuss this rule further once the others are introduced. (PairL) tells us we may refute (M_1, M_2) is a pair of type A_1 with A_2 just if we can evidence either M_1 or M_2 is not an A_1 or A_2 respectively. (AppL) lets us refute an application evaluates to an A just if we can show the applied M only takes values of type B to values of type A, and additionally refute the applicant N evaluates to a B value. Note this unique combination of affirmation combined with refutation in a pair of premises was discussed with the pure calculus rule for (AppL), too. (cL) allows us to refute that taking succ or pred of M will give us a Nat just if we can refute that M is a Nat. For conditional terms, we get a pair of rules which allow us to reason separately about the guard or the branches. They sensibly do this since these are the two ways we can refute a conditional will be of type A; (IfzL1) does this by requiring we refute the guard is a Nat, and (IfzL2) does this when we refute both branches evaluate to an A. Let statements have two options for refutation as well, namely (LetL1) says that to refute $\mathsf{let}\,(x, y) = M\,\mathsf{in}\,N$ we refute that $N : A$ regardless of any substitution. (LetL2) requires us to refute that M is a pair of type $B_1 \times B_2$ and then show that for N to be of type A it must be that x was a term of type B_1 and y was a B_2. Given we have already shown the first premise it cannot be the case that x and y are B_1 and B_2 respectively; therefore we refute $N : A$ since the right side of the judgement is false.

Right rules are mostly morally similar to those found in a typical affirmation-only one-sided type system, except that they allow for the more general diverging kind of judgement. In this, we will include the *Structural Rule* (Id) which ought to look familiar if you've seen simply typed λ-Calculus before. We may conclude that, given x is a variable of type A (evaluates to a value of type A) it trivially evaluates to a value of type A or diverges. (ZeroR) lets us trivially conclude the language constant introductory form zero is indeed a value of type Nat. (cR) is the right dual of (cL), and it allows us to affirm that *succ*/*pred* of some term M is a Nat just if we can affirm M is a Nat. Fix terms/abstractions have a pair of rules like in the pure calculus that allow us to reason using $\rightarrowtail$ as well as the typical sufficiency. Note that they do not have left-rule duals, which we are able to make up for partly with disjointness reasoning; the specifics are discussed in Section 2.2.1. (FixsR) lets us affirm that $\mathsf{fix}\,f(x).\,M$ is a value of type $A \to B$ just if we can affirm M is a term of type B after we include that x is a variable of type A and f takes on the function type in the conclusion. (FixnR) tells us we may affirm that $\mathsf{fix}\,f(x).\,M$ is a value of type $A \rightarrowtail B$ if we can affirm that, given M is a term which evaluates to a B and f is a variable of type $A \rightarrowtail B$, we can show x is a variable of type A. (AppR) lets us affirm that an application evaluates to an A or diverges just if we can show M is a term of type $B \to A$ as well as N evaluates to a value of type B or diverges. The remaining rules are also fairly standard, noting that for (IfzR) we must affirm the guard, and both the conditional branches are all the right type to affirm evaluation to an A or divergence.

2.2.1 How powerful is Disjointness

Initially, it may appear that we are missing some rules left which would otherwise mirror those in the right rules like refutations for function types etc. Ramsay and Walpole present a set of demonstrative alternative rules which would be used to prove disjointness for introductory forms:

$$\text{(ZeroL)}\ \frac{}{\Gamma, \mathtt{zero} : A \vdash \Delta}\ \mathrm{A} \neq \mathsf{Nat} \qquad \text{(cL2)}\ \frac{}{\Gamma, c(M) : A \vdash \Delta}\ \begin{array}{c} A \neq \mathsf{Nat} \\ c \in \mathtt{succ}, \mathsf{pred} \end{array}$$

$$\text{(PairL2)}\ \frac{}{\Gamma, (M, N) : A \vdash \Delta}\ \begin{array}{c} \forall B, C \\ A \neq (B \times C) \end{array} \qquad \text{(FixL)}\ \frac{}{\Gamma, \mathsf{fix}\,f(x).\,M : A \vdash \Delta}\ \begin{array}{c} \forall B, C \\ A \neq B \to C \end{array}$$

Figure 2.3: Alternative disjointness rules.

They conclude that which set we decide to use, either these or the (Dis) as we end up using, represent a difference in the kinds of things we can show. Specifically, (Dis) expresses the general notion of disjointness and thus is not reliant on any special case or shape of a typing judgement. In choosing this one rule we sacrifice being able to type non-well-typed terms like $\mathsf{pred}(\mathsf{fix}\,f(x).\,\mathtt{succ}(x)) : \mathsf{Nat} \times \mathsf{Nat}$. However, we get that we can prove disjointness for types in of term and it doesn't need to be in introductory form as the rules in Figure 2.3 would have it. To demonstrate, one may refute that $(\underline{0}, \lambda x.x) : \mathsf{Nat} \to \mathsf{Nat}$ with (Dis), but not with any of the rules in Figure 2.3.

Typing Rules for Ok

$$(\mathsf{OkVarR})\ \frac{}{\Gamma \vdash x : \mathsf{Ok},\ \Delta} \qquad (\mathsf{OkL})\ \frac{\Gamma,\ M : \mathsf{Ok} \vdash \Delta}{\Gamma,\ M : A \vdash \Delta} \qquad (\mathsf{OkR})\ \frac{\Gamma \vdash M : A,\ \Delta}{\Gamma \vdash M : \mathsf{Ok},\ \Delta}$$

$$(\mathsf{OkApL1})\ \frac{\Gamma,\ M : \mathsf{Ok} \rightarrowtail A \vdash \Delta}{\Gamma,\ M\ N : \mathsf{Ok} \vdash \Delta} \qquad (\mathsf{OkApL2})\ \frac{\Gamma,\ N : \mathsf{Ok} \vdash \Delta}{\Gamma,\ M\ N : \mathsf{Ok} \vdash \Delta}$$

$$(\mathsf{OkSL})\ \frac{\Gamma,\ M : \mathsf{Nat} \vdash \Delta}{\Gamma,\ \mathsf{succ}(M) : \mathsf{Ok} \vdash \Delta} \qquad (\mathsf{OkPL})\ \frac{\Gamma,\ M : \mathsf{Nat} \vdash \Delta}{\Gamma,\ \mathsf{pred}(M) : \mathsf{Ok} \vdash \Delta} \qquad (\mathsf{OkPrL})\ \frac{\Gamma,\ M_i : \mathsf{Ok} \vdash \Delta}{\Gamma,\ (M_1,\ M_2) : \mathsf{Ok} \vdash \Delta}$$

Figure 2.4: Type Assignment with Ok [16].

2.3 Ill Typedness and Ok

To be able to reason directly about crashing or divergence, Ramsay and Walpole introduce their new type Ok to represent 'evaluates'. Semantically every term that evaluates to a value inhabits this type. We provide the formal definition below.

Definition 2.3.1 (Semantics of Types) *The semantic 'meaning' of a type is expressed by $[\![A]\!]$ for some type variable A. $[\![A]\!]$ is the set of all value inhabitants or 'values of a type' A. For completeness, we will mention the semantics of all the types in the two-sided system as well as those which we will use in Section 2.4 to define the success semantics. $\mathcal{T}[\![A]\!]$ are those terms which evaluate to a value of type A.*

$$\begin{aligned}
\mathsf{Vals}_0 &= \{V | V \in \Lambda_0\} \\
[\![\mathsf{Ok}]\!] &= \mathsf{Vals}_0 \\
[\![\mathsf{Nat}]\!] &= \{\underline{0}, \underline{1}, \underline{2}, \dots\} \\
[\![A \times B]\!] &= \{(V, W) | V \in [\![A]\!], W \in [\![B]\!]\} \\
[\![A \to B]\!] &= \{\mathsf{fix}\, f(x).\, M | \forall V \in \mathsf{Vals}_0. V \in [\![A]\!] \Rightarrow M[V/x][\mathsf{fix}\, f(x).\, M] \in \mathcal{T}_\perp[\![B]\!]\} \\
[\![A \rightarrowtail B]\!] &= \{\mathsf{fix}\, f(x).\, M | \forall V \in \mathsf{Vals}_0. M[V/x][\mathsf{fix}\, f(x).\, M/f] \in \mathcal{T}[\![B]\!] \Rightarrow V \in [\![A]\!]\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}[\![A]\!] &= \{M | M \Downarrow V, V \in [\![A]\!]\} \\
\mathcal{T}_\perp[\![A]\!] &= \{M | M \in \mathcal{T}[\![A]\!] \vee M \Uparrow\} \\
\mathcal{T}_{\perp\!\!\!/}[\![A]\!] &= \{M | M \in \mathcal{T}_\perp[\![A]\!] \vee M \not\Downarrow V\}
\end{aligned}$$

By this definition, we see that $\mathcal{T}[\![\mathsf{Ok}]\!]$ are all terms which evaluate, $\mathcal{T}_\perp[\![\mathsf{Ok}]\!]$ are all the terms that do not get stuck. As discussed earlier, functions may diverge in their application when they are typed $A \to B$, which is what we see in $\mathcal{T}[\![A \to B]\!]$ which is all the terms that when given a value from $[\![A]\!]$ evaluate to a term in $\mathcal{T}[\![B]\!]$ or diverge in the process. Note that there is a fundamental asymmetry between the arrow types as presented in this definition. This arises since CBV is based towards values, but in Section 2.4 we discuss how the success semantics create a desirable symmetry for the one-sided system.

The necessity arrow has semantics which allow for trivial satisfaction when the function does not ever produce an inhabitant of $\mathcal{T}[\![B]\!]$, e.g. $\lambda x.(x, x) \in [\![\mathsf{Nat} \rightarrowtail \mathsf{Nat}]\!]$. We see $\lambda x.(x, x)$ never produces a Nat simply because it only produces pairs. In fact, in the two-sided system we may use (Dis) to show $\lambda x.(x, x) : \mathsf{Nat} \rightarrowtail \mathsf{Nat}$, and note this behaviour is exploited in the proof system via the disjointness rule.

Ok is appended to the two-sided type system with the rules in Figure 2.4.

2.4 Complements Success Semantics

By the refutational form of two-sided judgements there is an intrinsic notion of exclusion from the semantic of a type; $M : A \vdash \iff M \notin \mathcal{T}[\![A]\!]$. Ramsay and Walpole opt to introduce a complement operator to incorporate such negation into the one-sided type system, as has been done before in a number of

traditional type systems [16]. Can we fully simulate two-sided reasoning with such a complement operator? That is to say can we simulate judgements like $M : A \vdash$ in writing $\vdash M : A^c$; we explore their findings here.

Definition 2.4.1 *We extend the grammar of types in Definition 2.2.1 by a complement operator* $A ::= \ldots \mid A^c$ *and extend their two-sided system of rules with the addition of the following two below.*

Complement Rules

$$(\mathsf{CompL})\ \frac{\Gamma \vdash M : A, \Delta}{\Gamma, M : A^c \vdash \Delta} \qquad (\mathsf{CompR})\ \frac{\Gamma, M : A \vdash \Delta}{\Gamma \vdash M : A^c, \Delta}$$

We find immediately that these rules are unsound, in a proof by counterexample[16] we see adding these rules to the two-sided system *without* success semantics lets us prove that $\vdash \mathsf{pred}(\underline{\mathsf{id}})$ will diverge which is a contradiction because we can see it goes wrong. In this case, since we prove $\vdash \mathsf{pred}(\underline{\mathsf{id}}) : \mathsf{Ok}^c$ we show $\mathsf{pred}(\underline{\mathsf{id}}) \in \mathcal{T}[\![\mathsf{Ok}^c]\!]$ i.e. it is in just those terms that diverge. We next introduce the subtle change they make to the semantics of typings which restores soundness to the system after this addition.

In Ramsay and Walpole's two-sided type system, we specify the semantics of a judgement in a way that is similar to that of a typical one-sided system. However, after the addition of complementation, we end up with unsoundness and an asymmetry in judgements. Suppose we have the judgements $M : A \vdash$ and $\vdash M : A^c$; we would expect such a pair of judgements to have the same meaning, intuitively refuting that M evaluates to an A ought to be the same as saying M evaluates to not an A, right? This is actually not the case before we assert success semantics, since the first judgement specifically says $M \notin \mathcal{T}[\![A]\!]$, meaning importantly it's permitted that $M \in \mathcal{T}_{\perp\ell}[\![A^c]\!]$. The second says more strictly that $M \in \mathcal{T}_{\perp}[\![A^c]\!]$, i.e. it says M must not crash, whereas the first holds even if M does. That can't be fair and indeed is the root of unsoundness here. The subtle but vital change that we must introduce to balance this is the introduction of success semantics; $\vdash M : A$ holds of *everything* in $\mathcal{T}_{\perp\ell}[\![A]\!]$. In simpler terms, M may evaluate to a value of type A, diverge or **go wrong**. To formalise this we give the definition below.

Definition 2.4.2 *The set of closed terms that may go wrong, diverge or evaluate to a value in* $[\![A]\!]$ *we define for each type* A*.*

$$\mathcal{T}_{\perp\ell}[\![A]\!] = \{M | M \in \mathcal{T}[\![A]\!] \text{ or } M \text{ diverges or } M \text{ goes wrong}\}$$

Furthermore, the sufficiency arrow is adjusted to allow the body to go wrong as well as diverge when executing on an argument:

$$[\![A \to B]\!] = \{\mathsf{fix} f(x)M \in \mathsf{Vals}_0 | \forall V \in [\![A]\!].M[V/x][\mathsf{fix}(x)M/f] \in \mathcal{T}_{\perp\ell}[\![B]\!]\}$$

Finally satisfaction now takes the form $M : A$ *is satisfied by* θ on the right iff $M\theta \in \mathcal{T}_{\perp\ell}[\![A]\!]$. Ramsay and Walpole prove the syntactic soundness of (CompL) and (CompR) under these semantics.

Even though this gives us symmetrical judgements, it means we lose Ramsay and Walpole's true-positives theorem they present for the two-sided system; specifically we cannot now say for certain that $\vdash M : \mathsf{Ok}$ will not crash. For this case in particular where something is typed Ok we can conclude exactly nothing; a term always must evaluate, diverge or go wrong and that is exactly what this tells us under these semantics!

It is not of concern to us in this work that we lose true negatives; CompaSS is built to only prove incorrectness and as such we require that we must retain the theorem of one-sided syntactic soundness *ill-typed programs don't evaluate*. As such that which we type as Ok^c with this system must go wrong or diverge and that is the heart of their [16] one-sided system and the heart of ours and our inference algorithm, too. Specifically, we call this ill-typed and mirror their theorem on which this work depends:

Theorem 2.4.1 (One-Sided Syntactic Soundness) *If* M *is ill-typed, then* M *does not evaluate.*

This theorem relates specifically to their system of one-sided type assignment given here in Figure 2.5 and discussed as we reason about CompaSS' type system.

2.5 Undesired Behaviour in JavaScript

The set of terms we want to show go wrong in JSS when viewed as a PCF-like language is larger than just those which cause a runtime crash in JS. In exploring what terms to add to those which JS sees as wrong,

$$\text{(Ok)}\ \frac{}{\Gamma \vdash M : \mathsf{Ok}} \qquad \text{(OkC1)}\ \frac{}{\Gamma,\, x : \mathsf{Ok}^c \vdash M : A} \qquad \text{(OkC2)}\ \frac{\Gamma \vdash M : \mathsf{Ok}^c}{\Gamma \vdash M : A}$$

$$\text{(Contra)}\ \frac{}{\Gamma,\, x : A,\, x : A^c \vdash M : A} \qquad \text{(Var)}\ \frac{}{\Gamma,\, x : A \vdash x : A} \qquad \text{(Disj)}\ \frac{\Gamma \vdash M : B}{\Gamma \vdash M : A^c}\, B \mid\mid A$$

$$\text{(Zero)}\ \frac{}{\Gamma \vdash \mathtt{zero} : \mathsf{Nat}} \qquad \text{(Succ1)}\ \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathtt{succ}(M) : \mathsf{Nat}} \qquad \text{(Succ2)}\ \frac{\Gamma \vdash M : \mathsf{Nat}^c}{\Gamma \vdash \mathtt{succ}(M) : A}$$

$$\text{(Pred1)}\ \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{pred}(M) : \mathsf{Nat}} \qquad \text{(Pred2)}\ \frac{\Gamma \vdash M : \mathsf{Nat}^c}{\Gamma \vdash \mathsf{pred}(M) : A}$$

$$\text{(Fix)}\ \frac{\Gamma,\, f : A \to B,\, x : A \vdash M : B}{\Gamma \vdash \mathsf{fix}\, f(x).\, M : A \to B} \qquad \text{(Let3)}\ \frac{\Gamma \vdash N : A}{\Gamma \vdash \mathsf{let}\,(x, y) = M\,\mathsf{in}\,N : A}$$

$$\text{(Let2)}\ \frac{\Gamma \vdash M : (B_1 \times B_2)^c \quad \Gamma,\, x_i : B_i^c \vdash N : A\ (\forall i)}{\Gamma \vdash \mathsf{let}\,(x_1, x_2) = M\,\mathsf{in}\,N : A} \qquad \text{(Let1)}\ \frac{\Gamma \vdash M : B \times C \quad \Gamma,\, x_1 : B,\, x_2 : C \vdash N : A}{\Gamma \vdash \mathsf{let}\,(x_1, x_2) = M\,\mathsf{in}\,N : A}$$

$$\text{(App1)}\ \frac{\Gamma \vdash M : B \to A \quad \Gamma \vdash N : B}{\Gamma \vdash M\,N : A} \qquad \text{(App2)}\ \frac{\Gamma \vdash M : (\mathsf{Ok}^c \to A)^c}{\Gamma \vdash M\,N : A} \qquad \text{(App3)}\ \frac{\Gamma \vdash N : \mathsf{Ok}^c}{\Gamma \vdash M\,N : A}$$

$$\text{(Pair1)}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M,\, N) : A \times B} \qquad \text{(Pair2)}\ \frac{\Gamma \vdash M_i : \mathsf{Ok}^c}{\Gamma \vdash (M_1, M_2) : A} \qquad \text{(Pair3)}\ \frac{\Gamma \vdash M_i : A_i^c}{\Gamma \vdash (M_1, M_2) : (A_1 \times A_2)^c}$$

$$\text{(IfZ1)}\ \frac{\Gamma \vdash M : \mathsf{Nat}^c}{\Gamma \vdash \mathsf{ifz}\,M\,\mathsf{then}\,N\,\mathsf{else}\,P : A} \qquad \text{(IfZ2)}\ \frac{\Gamma \vdash N : A \quad \Gamma \vdash P : A}{\Gamma \vdash \mathsf{ifz}\,M\,\mathsf{then}\,N\,\mathsf{else}\,P : A}$$

Figure 2.5: One-Sided Type Assignment.

'undesired behaviour' is a sensible choice. One aspect motivating this decision was the baffling result that, in JSS (and thus JavaScript) the statement `(x => x) + (x => x)` adding two arrow functions does *not* get stuck. In fact, JS says this is `"x => xx => x"`, a string formed by the function bodies being coerced into strings and appended. We argue that this represents a strong example of undesired behaviour, and as such rules around plus and minus in CompaSS try to show a term goes wrong when either is used with anything but two numbers. It is acceptable for JSS as presented to ill-type where plus is, in our view, acceptable to use with two strings. In this case, since JSS does not have strings in its grammar, nor a string type, strings are not considered values and as such would not constitute evaluation. Another similar example is adding a number to an arrow function, `0 + (x => x)` produces the result `"0x => x"` which again we argue is generally unwanted. Note incorrectness remains when we swap our `+` with a `-`, and still the program does not crash. Instead of a string, `(x => x) - (x => x)` and `0 - (x => x)` both quietly result in `NaN`, which we deem undesired and thus something we want to prove incorrect. This symmetry of failures for `+`/`-` we exploit by generalising them both into a set of (NumOp) rules operating on one grammar shape.

One other example of undesired behaviour is the permitted comparison between terms of different types in JSS, specifically relating to the if-zero grammar shape. In our grammar we allow conditionals guarded on whether their first argument is less than or equal to zero. JS is actually happy to find out whether a function is less than or equal to zero, too! We write `(x => x) <= 0` and JS says this is `false`. Note it does not crash or go wrong, but we say it ought to represent undesired behaviour. Thus we make our type system try to restrict the first term of our if-zero term to being a number by allowing it to type programs which violate this to be ill-typed.

If we were to remove these cases where we want to show undesired behaviour, we would only end up with a less expressive system which is not able to prove as many things wrong (ignores more true positives). This naturally impacts any potential utility drastically, and so we choose to keep this extra utility and detect undesired behaviour as well as actual JS crashes; JS' philosophy of trying its hardest not to crash during runtime is what has put us in this position.

2.6 Related Work

Lindahl and Sagonas[14] apply a success type system to Erlang, a functional programming language widely used in concurrent communications systems. Like JS, Erlang philosophies and programming practises are already well established. Trying to uproot this structure by putting in place a restrictive static type system which may miss a function's intended use is not productive; few such efforts have seen wide adoption. The chief goal of this work was to show their success type system was practical, flexible and scalable, which departs from our work. We do not put as much focus on practical applications of CompaSS, which is instead chiefly to show that a success type system *can* be applied to some fragment of JavaScript, and its implementation can show definite type clashes in functional programs. It is our contribution as, to our knowledge, this has not been done prior. JSS was not chosen to be similar to Erlang, but JSS is also of the functional paradigm, which allows us to lean on existing reasoning e.g. type reasoning for PCF-like languages[16] to create our rules.

Sagonas[17] takes it upon himself to strengthen the static-analysis tool Dialyzer[2] for Erlang. For this language, analyses which produce success typings may be *considerably more powerful t*han purely dataflow-based analysis. Their rewrite for Dialyzer sought to over-approximate the sets of input terms to functions allowing them to evaluate to a value. When these more flexible functions are used in a provably wrong way, we know for certain this will fail with their original smaller types, too.
The user can choose to include false alarms/false positives in the system, which is a notable departure from our work. The main reason is data races are subtle and difficult to locate, and this option allows the tool to treat higher-order calls with greater care. This means correctness is emphasised over defect detection which is *much stronger* than what we present when failure is not an option. Extensions to this work may plausibly include such a setting for dealing with JavaScript's concurrency; for typing Promises.

Before the inception of TypeScript in 2012 [5](see copyright), efforts towards unintrusive static inference systems for JavaScript include Anderson et al.[8] and their property-focussed type inference system. They define a subset of JavaScript on which they focus their efforts, JS_0, which still includes important features like defining object constructors through setting properties and methods of functions with `this`. We also seek to preserve important features of the language, albeit functional features, in JSS. For example, we exclusively use arrow-function (or lambda function) definitions. This also allows for elegant typing rules since functions are first-class objects we can treat them as values much like in PCF and Lambda Calculus. Their work defines a type-annotated version of JS_0 called JS_0^{T} where a grammar for types is laid out. This

grammar implicitly gives every method a type for `this` along with any argument types, which means where methods are called we can perform inference on the object type they belong to. The shape of an object depends on a set of 'potential' properties which become 'definite' when assigned. They conclude early on if a program is well-typed then it never accesses 'potential' properties. Fields and methods are distinct in that the former is either an object or an Int, and the latter a function type. Most notably $\text{JS}_0^\top$ includes subtyping, which our system does not. We made this decision to keep type constraints simple by restricting them to syntactic equality; for example $\mathsf{Ok} \neq \mathsf{Num}$ despite $\mathsf{Num} \sqsubseteq \mathsf{Ok}$. This has the added benefit of allowing us to use abstract data types for constraint resolution.
Their system provides rules for type congruence used throughout their subtyping rules. This is useful as it ultimately lets us infer whether an object is a subtype of another, which is used throughout their inference rules. For example, to type method calls we use subtyping to express that the parameter must have at least as many defined properties of the same name and type as the formal parameter ($\mathsf{parameter} \sqsubseteq \mathsf{formal}$), and that the receiver object is also stronger than that which the parameter belongs to.
A type inference algorithm for this system is defined, but unlike our work, it is not implemented nor is the implementation evaluated in the paper.

Learner et al. discuss their type system framework for JavaScript that arose from creating multiple type systems for the language [13]. In this work, they use the framework to implement essential features of TypeScript by overriding the framework's arrow type to introduce new semantics. Having control over the semantics suggests it may be customisable enough to implement a success type system, but this is not discussed. What is considered to be well typed in their base system excludes implicit object-to-string conversions, which is similar to our undesired behaviour extension of the success semantics which includes implicit conversion of functions to strings by Definition 3.1.4. Perhaps more importantly is they find this conservative approximation to be sufficient to type-check most well-behaved programs [13], which indicates it is a sensible approximation for us to make as well. Undesired behaviour is thereby something else that we can show as ill-typed, as we'll see for the first example in Chapter 4.

To avoid false positives in bug finding, Gao et al. [12] implement a gradual type system. What this means is they minimally add type annotations until the type-checker finds an error or they decide that other gradual type systems would not see this as an error (e.g. TypeScript [5]). Such type systems as TypeScript avoid finding false positives in bug catching by being *permissive* in that on some occasions when its not certain that we can derive a type, the `any` type is used as a wildcard [12]. This method, while reducing false positives, does not eliminate them in the case of TypeScript as we see in the introduction examples. Therefore, unlike our system, we get no guarantees about what is ill-typed whereas CompaSS can guarantee that all which is ill-typed diverges, goes wrong or uses undesired behaviour.

Chapter 3

Implementation

3.1 JSS

In this chapter we detail the process of choosing the subset of JS to apply the one-sided system to. Requirements were for the syntax to be described by a context-free grammar much like simply typed λ-Calculus or PCF. The grammar shapes also had to be similar enough to reuse most of the reasoning Ramsay and Walpole employ for their version of PCF in all rule systems they introduce (but most importantly the one-sided system). This requirement means, provided a good argument for the similarity, we can lean on some theorems proven in their paper[16]. Specifically the theorem of one-sided syntactic soundness, which guarantees that if M is ill-typed, then M does not evaluate. We will argue for the soundness with respect to our definition of not evaluating which includes the cases where clearly undesired results are produced. The specifics of an 'undesired result' are covered in Section 3.1.4 and this notion was motivated by our opinion on JS' overly flexible `+` and - operators.

3.1.1 From PCF

We began by producing a grammar for just the terms in the language, i.e. this language describes some expressions that would reduce to values normally stored to variables in JS programs. This approach was taken as we could start with a simple version of JSS on which to verify that CompaSS worked, then work outwards. In doing so we avoided pushing the whole language through while developing CompaSS, and this helped us reach an MVP before extending the language to empower CompaSS to handle multi-line programs. The grammar for terms and values is outlined below.

Definition 3.1.1 (Terms of JSS) *M, N are term variables, x is a bound variable name, $\underline{n}$ is a `number` (in the JavaScript sense; a double-precision float), $\circ \in \{$`+`, `-`$\}$ and V, W are values in the following grammar:*

$$
\begin{array}{llll}
\text{(Values)} & V, W & ::= & x \mid \underline{n} \mid x \texttt{ => } M \\
\text{(Terms)} & M, N & ::= & x \mid \underline{n} \mid M \circ N \mid x \texttt{ => } M \mid M\texttt{(}N\texttt{)} \mid M \texttt{ <= 0 ? } N \texttt{ : } P \\
 & & & \qquad \text{where } \circ \in \{\texttt{+}, \texttt{-}\}
\end{array}
$$

Any `fixed-width` text in a grammar or rule should be understood as literal syntactically valid JavaScript. We began by finding JS analogues of Ramsay and Walpole's[16] PCF-like language's terms. Note M`(`N`)` is an application which in JS requires brackets on the argument N. We have excluded pairs because they are not a special structure in JS, and instead are just arrays of length two. This would otherwise mean creating typing for a fictional construct which would be likely to introduce false positives, as far as JS incorrectness is concerned. We reuse some of the reasoning around pairs when considering plus or minus (hereby "*number operations*"), evident when comparing typing rules (NumOp1) versus (Pair1), (NumOp2) versus (Pair3) for the JSS one-sided system and PCF one-sided system, respectively. Both of these systems are introduced in Section 3.2.
We may also define evaluation contexts for these terms.

Definition 3.1.2 (JSS term evaluation) *These evaluation contexts describe where one may replace term variables with terms as they do in Definition 2.1.2.*

$$\mathcal{E}, \mathcal{F} ::= \square \mid (x \texttt{ => } M)(\mathcal{E}) \mid \mathcal{E}(N) \mid \mathcal{E} \circ M \mid V \circ \mathcal{E} \mid \mathcal{E} \texttt{ <= 0 ? } N \texttt{ : } P$$
$$\text{where } \circ \in \{+, -\}$$

As with our PCF-like language, given a context $\mathcal{E}$ we can replace the hole with M in writing $\mathcal{E}[M]$. As such we end up with the following single-step reduction relations under these evaluation contexts. Consequently, this attempts to represent how the term would be evaluated in JS:

$$\begin{array}{lrcll}
\text{(IfLtZ)} & \underline{n} \texttt{ <= 0 ? } N \texttt{ : } P & \triangleright & N & \iff \underline{n} \leq 0 \\
\text{(IfGtZ)} & \underline{m} \texttt{ <= 0 ? } N \texttt{ : } P & \triangleright & P & \iff \underline{m} > 0 \\
\text{(NumOp)} & \underline{n} \circ \underline{m} & \triangleright & \underline{r} & \text{where } \underline{r} = \underline{n} \circ \underline{m} \text{ and } \circ \in \{+, -\} \\
\text{(App)} & (x \texttt{ => } M)V & \triangleright & M[V/x] &
\end{array}$$

Apparent is the simplicity of the terms; it is effectively simply-typed λ-Calculus with the addition of built-in numbers, a conditional, and two number operations. We will just get the ability to type simple programs but do not yet get recursion for the absence of fix. Of course, this does not stop us from defining the Z-combinator as in Figure 3.1 (since JS eagerly evaluates, the Y combinator always diverges), but the Z-combinator is difficult to type and did not end up being typable by CompaSS. In the subsequent section, we introduce the grammar for writing sequential programs which allows variable definition and recursion, and this represents the full version of JSS. It also means we can write significantly more realistic programs in JSS than with just its terms.

```
f => (x => f(v => x(x)(v)))(x => f(v => x(x)(v)))
```

Figure 3.1: The Z-combinator as a JSS term.

3.1.2 Sequential Extension

We extend the simple grammar for terms by creating a grammar for variable definitions and blocks terminated with return statements. Top-level definitions cannot have a return as a global return in JS is not syntactically valid; we express this with separate grammars for blocks and for programs. We call the top-level statements/definitions the grammar for *Programs* denoted with a $\mathcal{P}$, $\mathcal{Q}$. *Blocks* are denoted with $\mathcal{B}$, $\mathcal{C}$. When discussing JSS programs *statement* specifically means a term of shape `const` f `=` M''`;` $\mathcal{P}$ i.e. a term which is not assigned to a variable. Additionally, *definition* is a term of shape `const` g `=` x `=>` M`;` $\mathcal{P}$ i.e. a term that defines a variable. This language is generally just used when referring to program (top-level) terms.

Definition 3.1.3 (Full JSS) *A JSS program is a recursively defined list of top-level definitions or statements. The former kind always adds a new variable to the global type environment, whereas the latter are typed and then ignored, unless they are ill-typed in which case the program is ill-typed.*

$$\begin{array}{rrcl}
\text{(Programs)} & \mathcal{P}, \mathcal{Q} & ::= & \texttt{const } g \texttt{ = } x \texttt{ => } M\texttt{;}\ \mathcal{P} \mid \texttt{const } f \texttt{ = } M''\texttt{;}\ \mathcal{P} \mid M'\texttt{;}\ \mathcal{P} \mid \varepsilon \\
\text{(Blocks)} & \mathcal{B}, \mathcal{C} & ::= & \texttt{const } y \texttt{ = } x \texttt{ => } M\texttt{;}\ \mathcal{B} \mid \texttt{const } x \texttt{ = } M''\texttt{;}\ \mathcal{B} \mid M'\texttt{;}\ \mathcal{B} \mid \\
 & & & \texttt{return } M'\texttt{;} \\
\text{(Terms)} & M, N, P, Q & ::= & x \mid \underline{n} \mid \{\mathcal{B}\} \mid M \circ N \mid x \texttt{ => } M \mid M(N) \mid M \texttt{ <= 0 ? } N \texttt{ : } P \\
 & & & \text{where } M' = M \backslash \{\mathcal{B}\},\ M'' = M' \backslash x \texttt{ => } N
\end{array}$$

We define some aliases for *terms excluding blocks* (M') and *terms excluding blocks and functions* (M''). This is to respect the syntax of JS and also allows us more expressive power when defining constants.

Specifically, in JS we are forbidden from assigning a term to a block, hence the need for the grammar terms `const` f `=` M''`;` $\mathcal{P}$ and `const` x `=` M''`;` $\mathcal{B}$. From these we also exclude functions to allow us to distinguish the first and second grammar shapes from $\mathcal{P}, \mathcal{B}$.

While we are allowed to write a block that is not assigned to a variable, we are not then allowed to terminate it with a return, so it must fall outside our grammar for blocks and we forbid it (it is excluded in shape $M'; \mathcal{P}$). Where it gives us more expressive power is we may now conclude whether the type of the defined variable in our global or block-scoped assumptions is a function. This then lets us re-introduce recursion as we had in the PCF-like language with fixpoints, where on typing a named function we insert the function name into the environment with a function type. Specifics of this will be presented in Chapter 3.2. We will discuss this also when designing constraint rules in Section 3.3.
Important to note is that programs may terminate in the empty program ε. The same is not true for blocks, which must always terminate in a return statement as this is the only non-recursive term in the block grammar. While this means we lose the ability to define void functions, it means we gain the guarantee that functions always have a return type and that type is not void. We thusly do not need a void or undefined type in our grammar for types, which is beneficial because we can then treat all such cases where one would arise as wrong in JSS. While this is not strictly the case in JS as you can also run void functions, it is at least trivially consistent with the goal of disallowing false positives within JSS itself as it is not within JSS.
Evaluation contexts for this grammar we present as follows, which again aim to show how eager evaluation would apply to JSS programs, blocks and terms.

Definition 3.1.4 (JSS program evaluation) *These evaluation contexts subsume the evaluation contexts for terms and include where we evaluate the new parts of the grammar.*

$$\begin{aligned}\mathcal{E}, \mathcal{F} \ ::= \ & \square \mid \texttt{(}x \texttt{ => } M\texttt{)(}\mathcal{E}\texttt{)} \mid \mathcal{E}\texttt{(}N\texttt{)} \mid \mathcal{E} \circ M \mid V \circ \mathcal{E} \mid \mathcal{E} \texttt{ <= 0 ? } N \texttt{ : } P \mid \texttt{const } f \texttt{ = } x \texttt{ => } M\texttt{; } \ldots \texttt{; } f(\mathcal{E}) \mid \\ & \texttt{const } x \texttt{ = } \mathcal{E}\texttt{; } \mathcal{P} \mid \texttt{const } x_1 \texttt{ = } V_1\texttt{; const } x_2 \texttt{ = } V_2\texttt{; } \ldots \texttt{; } \mathcal{E}\texttt{; } \mid \mathcal{E}\texttt{; } \mathcal{P} \mid V_1\texttt{; } V_2\texttt{; } \ldots \texttt{; } \mathcal{E}\texttt{; } \mid \{\mathcal{E}\} \mid \\ & \texttt{return } \mathcal{E}\texttt{;} \\ & \qquad \text{where } \circ \in \{\texttt{+}, \texttt{-}\}\end{aligned}$$

Also note that at this point we allow multi-character identifiers purely for realism and ease-of-understanding. This decision has no effect on the language semantics but we do restrict that variable shadowing does not occur.

3.1.3 Type system

We introduce a simple one-sided type system for JSS. Importantly this grammar includes types which feature in the fully-fledged one-sided type system with complements presented in Ramsay and Walpole's work [16] and whose origin and necessity are introduced in chapters 2.2 and 2.4.

Definition 3.1.5 (Two-Sided JSS Types) *Below are the types that terms and statements take on, plus a necessity arrow for use in the two-sided rules:*

$$\text{(Types)} \quad A, B \quad ::= \mathsf{Num} \mid A \to B \mid \mathsf{Ok} \mid A^c \mid A \rightarrowtail B$$

The main difference here to the PCF-like types presented we base our system off (definition 2.2.1) is the use of Num instead of Nat, as well as the removal of pairs. Using Num allows us to represent the built-in double-precision float type that JS uses for all numbers which are not of type `BigInt`. We take advantage of this by using the built-in mathematical operations `+`, `-` which allow us to define some rules for writing simple mathematical expressions and functions in JSS.

We discard pairs as they are not inherently related to JS' type system. There is no specific data structure which mirrors this, and while we could have defined a sort of pseudo-pair e.g. a list of length exactly two, we felt it was not worth the added overhead and would rather implement lists or not include built-in sequential data structures at all (we chose the latter option for elegance). However pairs may still be defined in the typical λ-Calculus way using a function of arity three, which JS is happy with due to functions being first-class objects in the language.

This type system is simple; from the perspective of JS it may seem too trivial. This is a fair point to make because we make no attempt to type objects/JSONs and their properties as e.g. Anderson et al. [8] do in their work. This is largely thanks to the reduced scope of this project; the final system is very close

to a PCF-like system rather than a JS like system, despite the language of JSS being a subset of JS. But we are more focused on implementing the success type system presented by Ramsay and Walpole [16] and showing that it can be practically applied to a language automatically in an inference program. A very exciting extension of this work would be able to express objects and their methods/properties. The fact we are not trying to do this benefits us; it becomes relatively straightforward to argue that the rationale in our rules is parallel to the system on which it is based. The expressive full JSS grammar does allow us to at least type interesting programs despite their simply constructed types.

The types we choose to give to programs are not of this grammar but instead constitute a type environment Δ as we see in the 'Program Rules' in Figure 3.3.

Definition 3.1.6 (JSS Program Types) *Below we define the types that programs inhabit.*

$$\Delta ::= \{x : A \mid x \text{ is a top-level variable name or statement label}\}$$

CompaSS one-sided type system and the block and program rules are reasoned about and discussed in Section 3.2.4, and the term rules are reasoned about in Section 3.2.

3.1.4 Crashing Extension

As introduced in Section 2.5 we concretely define what CompaSS' type system sees as definitely wrong with which we extend the set of programs which diverge or crash. The need for this stems from JS' apparent unending desire to make almost anything the programmer writes runnable or silently go wrong, which often end in failure all the same except a more expensive, nuanced or hard-to-detect kind. It's for this reason we want to make an effort to prove at least some of it as wrong with CompaSS.

Definition 3.1.7 (Undesired Behaviour) *Undesired or unpredictable behaviour is that which does not crash the program but instead silently fails and produces values which we deem unwanted. We write the specific terms in JSS below where* a: A, b : B *s.t.* $A, B \parallel \mathsf{Num}$, n: Num *and* $branchA$: Ok, $branchB$: Ok. *Note* A, B *are one-sided types as defined in 3.2.1.*

```
1    n + a; // string "n<a.function_and_body>"
2    n - a; // NaN
3    a + n; // string "<a.function_and_body>n"
4    a - n; // NaN
5    a + b; // string "<a.function_and_body><b.function_and_body>"
6    a - b; // NaN
7    a <= 0 ? branchA : branchB; // branchB
```

These sorts of failure is represented by (NumOp2) and (IfZ1) in the system through the flexibility of disjointness. We write `function_and_body` because the only concrete disjoint term from numbers in JSS are indeed functions. If we try to add or subtract any other grammar shapes in the language CompaSS will exit early for grammar violation or syntactic invalidity will arise, and as such it does not bare mentioning here.

We extend the success semantics to include this possibility for terms by the following definition.

Definition 3.1.8 (Success Semantics with Undesired Behaviour) *The set of terms that may go wrong, diverge, use undesired behaviour or evaluate to a value in* $[\![A]\!]$ *we define for each type* A*. We clarify, for the implementation and subsequent chapters, "fail" refers specifically to any one of these three things.*

$$\mathcal{T}_{\perp\lightning?}[\![A]\!] = \{M | M \in \mathcal{T}[\![A]\!] \text{ or } M \Uparrow \text{ or } M \text{ crashes or } M \text{ uses undesired behaviour}\}$$

Furthermore, the sufficiency arrow is adjusted to allow the body to use undesired behaviour as well as diverge or go wrong when executing on an argument:

$$[\![A \to B]\!] = \{M\,(x) \in \mathsf{Vals}_0 | \forall V \in [\![A]\!].M[V/x] \in \mathcal{T}_{\perp\lightning?}[\![B]\!]\}$$

We make that addition so that when the result of a function is ill-typed we allow for that function to have used undesired behaviour as well as having gone wrong or diverged.

We decide that we would not like there to be a type for `null`, `undefined`, `NaN` or `void` in JSS as they often make reasoning about incorrectness much more difficult. It also avoids the 'billion-dollar mistake' (Tony Hoare on Null References) of having `null`s become part of JSS' programming paradigm, whose exclusion some would see as an advantage. Though we do acknowledge such types exist in JS, and perhaps this constitutes yet another possible valid extension to CompaSS.

3.2 One-Sided Rules

We present all the formal rules used for deriving JSS and for CompaSS' rule system itself. There are three main sets of rules for JSS: the two-sided rules (figure 3.2), the one-sided rules (figure 3.3), and the constraint rules (figures 3.5 and 3.6). The two-sided rules are included to show the derivation from a system almost indistinguishable from Ramsay and Walpole's to our one-sided system, and our constraint rules are the underlying type system actually used by CompaSS.

3.2.1 One-Sided Types

Definition 3.2.1 (One-Sided JSS Types) *From Definition 3.1.3 we remove the necessity arrow and preserve all other types which constitute the types used in the one-sided system for JSS.*

$$\text{(Types)} \quad A, B \quad ::= \mathsf{Num} \mid A \to B \mid \mathsf{Ok} \mid A^c$$

The necessity arrow does not appear in the one-sided system for JSS for the same reason it does not appear in Ramsay and Walpole's one-sided system; it is equivalent under the success semantics to sufficiency with complements; $[\![A \rightarrowtail B]\!] = [\![A^c \to B^c]\!]$ [16]. Intuition for this is given in the derivations in Section 3.2.3 when discussing (App2).

3.2.2 From Two-Sided Rules for JSS

We did not start with a two-sided system when creating CompaSS' one-sided rules, rather they came after creating the one-sided rules and stemmed from the need to show similarity to the original two-sided system. Most of these rules are taken directly from Ramsay and Walpole's [16] two-sided type system where relevant to our language. Certain cases are transformed while their names are kept similar to show their change in use in CompaSS' system. Chief among our goals for the derivation is to remove the necessity arrow as it does not appear in the one-sided system, whilst maintaining some of the refutational expression that the two-sided system affords us.

Identical Rules (OkApL1) is unchanged from Ramsay and Walpole's system and says we refute that an application $M(N)$ evaluates if we can refute that M evaluates to a function. 'Evaluates to a function' is as general as we can write it; for M to evaluate to a specific type A it's just necessary that we give it anything and thus we write $\mathsf{Ok} \rightarrowtail A$. OkL is also exactly as Ramsay and Walpole present it, which is also the case for (OkApL2), (Disj), (Id), CompL and CompR.

Left Rules We make the following changes to (IfZL1): we replace Nat with Num, and replace the ifz then else with JS syntax. The semantics of the rule remain unchanged from Section 2.2. The exact same is true for (IfZL2) as well. (NumPrL) was created with the same strategy as (PairL) in Ramsay's two-sided system and can almost be thought of as a simpler special case for pairs of numbers. We may refute that M_1 plus or minus M_2 evaluates to a Num if we can show this is the case for either M_i, i.e. this rule has disjoint premises.

Right Rules (NumR) hails from (ZeroR) which we say is morally acceptable since we can treat all the numbers in JS like built-in constants (literals) and thus a proof of their type follows immediately. (AbsR) is identical to (FixsR) except with JS syntax and no fixterm capabilities (recursion is something we get once we define block rules and program rules). (AppR) is another such rule that comes straight from Ramsay's system except we trade in JS application syntax with brackets. Like the left rule, (NumPrR) we deduce using the same reasoning as with (PairR) in Ramsay's system; like for a pair to be of type $A \times B$ we must affirm that both terms are their respective types, for a number operation to be a Num we must affirm that both operands are numbers as well.

3.2.3 Term Rules

CompaSS type system contains three major sorts of typing rule; *term*, *block* and *program*. Program rules allow us to construct the type of a JSS program. They are always given conclusions of the form $\Gamma \vdash \mathcal{P} : \Delta$ i.e. given some type *environment*, this program "is of type" or "produces" this type environment Δ. In this way Δ is used to accumulate the types of all the top level defintions registered by the program rules. Block rules we just use to discern the type of a function body. They completely describe type assignment to multi-line functions which is always the type of the return statement given the statements and definitions in the block above. Term rules for JSS arose first as a nearly one-to-one mapping from

Right Rules

$$\text{(OkR)}\ \frac{\Gamma \vdash M : A,\ \Delta}{\Gamma \vdash M : \mathsf{Ok},\ \Delta} \qquad \text{(Id)}\ \frac{}{\Gamma,\ x : A \vdash x : A,\ \Delta} \qquad \text{(Disj)}\ \frac{\Gamma \vdash M : B,\ \Delta}{\Gamma,\ M : A \vdash \Delta}\ \mathrm{A}\ ||\ B$$

$$\text{(NumR)}\ \frac{}{\Gamma \vdash n : \mathsf{Num},\ \Delta} \qquad \text{(AbsR)}\ \frac{\Gamma,\ x : A \vdash M : B,\ \Delta}{\Gamma \vdash x \texttt{ => } M : A \to B,\ \Delta}\ \mathrm{x} \notin \mathsf{dom}(\Gamma)$$

$$\text{(AppR)}\ \frac{\Gamma \vdash M : B \to A,\ \Delta \quad \Gamma \vdash N : B,\ \Delta}{\Gamma \vdash M\texttt{(}N\texttt{)} : A,\ \Delta}$$

$$\text{(NumPrR)}\ \frac{\Gamma \vdash M : \mathsf{Num},\ \Delta \quad \Gamma \vdash N : \mathsf{Num},\ \Delta}{\Gamma,\ M \circ N : \mathsf{Num} \vdash \Delta}\ \circ \in \{\texttt{+}, \texttt{-}\} \qquad \text{(CompR)}\ \frac{\Gamma,\ M : A \vdash \Delta}{\Gamma \vdash M : A^c,\ \Delta}$$

Left Rules

$$\text{(OkApL1)}\ \frac{\Gamma,\ M : \mathsf{Ok} \rightarrowtail A \vdash \Delta}{\Gamma,\ M\texttt{(}N\texttt{)} : \mathsf{Ok} \vdash \Delta} \qquad \text{(OkL)}\ \frac{\Gamma,\ M : \mathsf{Ok} \vdash \Delta}{\Gamma,\ M : A \vdash \Delta} \qquad \text{(OkApL2)}\ \frac{\Gamma,\ N : \mathsf{Ok} \vdash \Delta}{\Gamma,\ M\texttt{(}N\texttt{)} : \mathsf{Ok} \vdash \Delta}$$

$$\text{(IfZL1)}\ \frac{\Gamma,\ M : \mathsf{Num} \vdash \Delta}{\Gamma,\ M \texttt{ <= 0 ? } N \texttt{ : } P : A \vdash \Delta} \qquad \text{(IfZL2)}\ \frac{\Gamma,\ N : A \vdash \Delta \quad \Gamma,\ P : A \vdash \Delta}{\Gamma,\ M \texttt{ <= 0 ? } N \texttt{ : } P : A \vdash \Delta}$$

$$\text{(NumPrL)}\ \frac{\Gamma,\ M_i : \mathsf{Num} \vdash \Delta}{\Gamma,\ M_1 \circ M_2 : \mathsf{Num} \vdash \Delta}\ \circ \in \{\texttt{+}, \texttt{-}\}\ i \in \{1, 2\} \qquad \text{(CompL)}\ \frac{\Gamma \vdash M : A,\ \Delta}{\Gamma,\ M : A^c \vdash \Delta}$$

Figure 3.2: Two-sided for rules with which we derive CompaSS' one-sided rules for terms.

Structural Rules

$$\text{(Var)}\ \frac{}{\Gamma,\, x : A \vdash x : A}$$

Program Rules

$$\text{(Defn)}\ \frac{\Gamma \vdash M : A \quad \Gamma,\, f : A \vdash \mathcal{P} : \Delta}{\Gamma \vdash \texttt{const}\ f\ \texttt{=}\ M\texttt{;}\ \mathcal{P} : \{f : A\} \cup \Delta}\ \mathrm{f} \notin \mathsf{dom}(\Gamma) \qquad \text{(TopStmt)}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash \mathcal{P} : \Delta}{\Gamma \vdash M\texttt{;}\ \mathcal{P} : \{M : A\} \cup \Delta}$$

$$\text{(DefnAbs)}\ \frac{\begin{array}{c}\Gamma,\, g : B \to A \vdash x\ \texttt{=>}\ N : B \to A \\ \Gamma,\, g : B \to A \vdash \mathcal{P} : \Delta\end{array}}{\Gamma \vdash \texttt{const}\ g\ \texttt{=}\ x\ \texttt{=>}\ N\texttt{;}\ \mathcal{P} : \{g : B \to A\} \cup \Delta}\ \mathrm{g} \notin \mathsf{dom}(\Gamma) \qquad \text{(Stop)}\ \frac{}{\Gamma \vdash \varepsilon : \Gamma}$$

Block Rules

$$\text{(Compo1)}\ \frac{\begin{array}{c}\Gamma \vdash M : B \\ \Gamma,\, x : B \vdash \mathcal{B} : A\end{array}}{\Gamma \vdash \texttt{const}\ x\ \texttt{=}\ M\texttt{;}\ \mathcal{B} : A}\ \mathrm{x} \notin \mathsf{dom}(\Gamma) \qquad \text{(Compo2)}\ \frac{\Gamma \vdash \mathcal{B} : A}{\Gamma \vdash \texttt{const}\ x\ \texttt{=}\ M\texttt{;}\ \mathcal{B} : A}$$

$$\text{(CompoAbs1)}\ \frac{\begin{array}{c}\Gamma,\, y : B \to C \vdash x\ \texttt{=>}\ N : B \to C \\ \Gamma,\, y : B \to C \vdash \mathcal{B} : A\end{array}}{\Gamma \vdash \texttt{const}\ y\ \texttt{=}\ x\ \texttt{=>}\ N\texttt{;}\ \mathcal{B} : A}\ \mathrm{y} \notin \mathsf{dom}(\Gamma)$$

$$\text{(CompoAbs2)}\ \frac{\Gamma \vdash \mathcal{B} : A}{\Gamma \vdash \texttt{const}\ y\ \texttt{=}\ x\ \texttt{=>}\ N\texttt{;}\ \mathcal{B} : A}$$

$$\text{(Ret)}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \texttt{return}\ M\texttt{;} : A} \qquad \text{(Stmt1)}\ \frac{\Gamma \vdash M : B \quad \Gamma \vdash \mathcal{B} : A}{\Gamma \vdash M\texttt{;}\ \mathcal{B} : A} \qquad \text{(Stmt2)}\ \frac{\Gamma \vdash M : \mathsf{Ok}^c}{\Gamma \vdash M\texttt{;}\ \mathcal{B} : A}$$

Term Rules

$$\text{(Num)}\ \frac{}{\Gamma \vdash n : \mathsf{Num}} \qquad \text{(Abs)}\ \frac{\Gamma,\, x : A \vdash M : B}{\Gamma \vdash x\ \texttt{=>}\ M : A \to B}\ \mathrm{x} \notin \mathsf{dom}(\Gamma)$$

$$\text{(App1)}\ \frac{\Gamma \vdash M : B \to A \quad \Gamma \vdash N : B}{\Gamma \vdash M\texttt{(}N\texttt{)} : A} \qquad \text{(App2)}\ \frac{\Gamma \vdash M : B}{\Gamma \vdash M\texttt{(}N\texttt{)} : A}\ \mathrm{Ok}^c \to A \mid\mid B$$

$$\text{(App3)}\ \frac{\Gamma \vdash N : \mathsf{Ok}^c}{\Gamma \vdash M\texttt{(}N\texttt{)} : A} \qquad \text{(IfZ1)}\ \frac{\Gamma \vdash M : B}{\Gamma \vdash M\ \texttt{<= 0 ?}\ N\ \texttt{:}\ P : A}\ \mathrm{Num} \mid\mid B$$

$$\text{(IfZ2)}\ \frac{\Gamma \vdash N : A \quad \Gamma \vdash P : A}{\Gamma \vdash M\ \texttt{<= 0 ?}\ N\ \texttt{:}\ P : A} \qquad \text{(NumOp1)}\ \frac{\Gamma \vdash M : \mathsf{Num} \quad \Gamma \vdash N : \mathsf{Num}}{\Gamma \vdash M \circ N : \mathsf{Num}} \circ \in \left\{ \begin{array}{c} + \\ - \end{array} \right\}$$

$$\text{(NumOp2)}\ \frac{\exists i\ .\ \Gamma \vdash M_i : B}{\Gamma \vdash M_1 \circ M_2 : A} \circ \in \left\{ \begin{array}{c} + \\ - \end{array} \right\}\ \mathsf{Num} \mid\mid B \qquad \text{(Blk)}\ \frac{\Gamma \vdash \mathcal{B} : A}{\Gamma \vdash \{\mathcal{B}\} : A}$$

$$\text{(Ok)}\ \frac{}{\Gamma \vdash M : \mathsf{Ok}} \qquad \text{(OkC1)}\ \frac{}{\Gamma,\, x : \mathsf{Ok}^c \vdash M : A}$$

Figure 3.3: One-sided typing rules; the foundation of CompaSS' type system.

Ramsay and Walpole's one-sided rules for their PCF-like language (fig. 2.5). This approach was chosen in the interest of simplicity, and correctness since JSS terms are almost identical to their language. To explore the soundness of these rules, and whether any should be removed, we then created a two-sided ruleset for JSS (figure 3.2). What follows are the derivations that link the two- and one-sided rules for JSS together and mention those rules we were able to remove, too. Note the two-sided rules for JSS are only sound under the success semantics since they include (CompL) and (CompR), which rely on the symmetry of success judgements.

The derivations given below do not all represent actual proof trees. The aim of this figure is to convince the reader of their soundness or at least sound reasoning with respect to the two-sided rules. Certain informal transformations in the trees e.g. *Contrapositive* are not concrete rules but reason about the semantics of the types included, instead of just the syntax.

(Id) $\longrightarrow$ (Var) (NumR) $\longrightarrow$ (Num) (AppR) $\longrightarrow$ (App1) (AbsR) $\longrightarrow$ (Abs) (NumPairR) $\longrightarrow$ (NumOp1)

The mappings above simply indicate removing the Δ terms in the premises and conclusions. This is only possible with the right rules, and if done with the left rules we would end up with zero terms on the right side, and as such all these rules had exactly one term on the right ignoring Δ. One should notice the similarity between the rules in these pairs when looking between 3.2 and 3.3.

$$(\mathsf{OkR})\ \frac{\Gamma \vdash M : A,\ \Delta}{\Gamma \vdash M : \mathsf{Ok},\ \Delta} \longrightarrow (\mathsf{Ok})\ \frac{}{\Gamma \vdash M : \mathsf{Ok}}$$

(OkR) came from Ramsay and Walpole's system plus Ok, before success types were added. As such, one had to show M evaluated to something $(V : A)$, which was necessary to show $M : \mathsf{Ok}$. However, under the success semantics, all terms trivially inhabit Ok whether they evaluate or not, so the derivation here represents an update to the rule, as well as removing Δ to create a one-sided rule for CompaSS' system.

$$\begin{array}{r} (\mathsf{Disj})\ \dfrac{\Gamma \vdash M : B,\ \Delta}{\Gamma,\ M : A \vdash \Delta}\ \mathrm{A} \parallel B \\ (\mathsf{CompR})\ \dfrac{}{\Gamma \vdash M : A^c,\ \Delta} \end{array} \longrightarrow (\mathsf{PureDisj})\ \frac{\Gamma \vdash M : B}{\Gamma \vdash M : A^c}\ \mathrm{A} \parallel B$$

We derive a rule for pure disjunction in the one-sided system, even though it does not exist on its own in the rules in Figure 3.3. The rules in the figure arose after a few iterations of removing and changing rules, and one such change was instead of having a disjunction rule, we spread disjunction through the rules anywhere a rule introduced a complement in its premise (except Ok^c). But of course in order to do this disjunction needed to be part of the one-sided system already for us to use it, and this derivation is how one may arrive at such a rule. We apply (CompR) to allow us to feed in a judgement with a term on the right, aligning the conclusion and premise judgements to the target right rule.

The spreading of disjunction is sound to us because before spreading one would always be able to apply (PureDisj) to the premise of such a complement-introducing rule. We do not believe it leaves us with less power either i.e. it is complete because the only place we can use (PureDisj) is exactly where a type of some term is complemented. The motivation for this spreading is covered in the subsequent Section 3.3 on constraint rules.

$$\begin{array}{r} (\mathsf{Disj})\ \dfrac{\Gamma \vdash M : B,\ \Delta}{\Gamma \vdash M : (\mathsf{Ok}^c \to A)^c,\ \Delta}\ \mathrm{Ok}^c \to A \parallel B \\ (\mathsf{CompL})\ \dfrac{}{\Gamma,\ M : \mathsf{Ok}^c \to A \vdash \Delta} \\ \textit{Simplify}\ \dfrac{}{\Gamma,\ M : \mathsf{Ok}^c \to C^c \vdash \Delta} \\ \textit{Contrapositive}\ \dfrac{}{\Gamma,\ M : \mathsf{Ok} \rightarrowtail C \vdash \Delta}\ \text{let } C = A^c \\ (\mathsf{OkApL1})\ \dfrac{}{\Gamma, M(N) : \mathsf{Ok} \vdash \Delta} \\ (\mathsf{OkL})\ \dfrac{}{\Gamma,\ M(N) : A^c \vdash \Delta} \\ (\mathsf{CompR})\ \dfrac{}{\Gamma \vdash M(N) : A,\ \Delta} \end{array} \longrightarrow (\mathsf{App2})\ \frac{\Gamma \vdash M : B}{\Gamma \vdash M(N) : A}\ \mathrm{Ok}^c \to C \parallel B$$

(App2) has the longest derivation, fitting for such a powerful rule. At the heart of the derivation is left-rule (OkApL1) which in Ramsay's system lets us choose a type for the result of the function in the premise. We say the result type C is A^c where A came from the bottom judgement, which when we take the

contrapositive allows us to get back A as the result type of the complemented function type in the disjunction (thereby we say M is not any function at all that could give us back an A).

For intuition on the *Constrapositive* step, all functions which evaluate to an A inhabit the type $\mathsf{Ok} \rightarrowtail A$; intuitively to get back a term of specific type A, diverge, go wrong or use undesired behaviour, all functions necessarily require *anything at all* (incl. divergence and crashing). Taking the contrapositive of this as in the derivation, we intuit equivalently[16] the type of all functions (written with the sufficiency arrow) are those which it is sufficient to give *nothing at all* to get a term of a specific type A, diverge, go wrong or use undesired behaviour. From the perspective of the Curry-Howard correspondence, this idea is parallel to a vacuously true implication: the premise is 'false' and A follows trivially. The necessity and sufficiency versions are equivalent because neither place any requirement on their input, and both either evaluate to a term of type A or crash or diverge in the process.

The final step of this derivation uses (Disj) since we chose to spread disjointness through the rules instead of having it as a separate rule. Before we did this (App2) had a derivation which finished at (CompL), and we would use (PureDisj) on top to produce the rule we see here.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash N : \mathsf{Ok}^c,\ \Delta}{\Gamma, N : \mathsf{Ok} \vdash \Delta}\,(\mathsf{CompL})}{\Gamma,\ M\,(N) : \mathsf{Ok} \vdash \Delta}\,(\mathsf{OkApL2})}{\Gamma,\ M\,(N) : A^c \vdash \Delta}\,(\mathsf{OkL})}{\Gamma \vdash M\,(N) : A, \Delta}\,(\mathsf{CompR}) \longrightarrow \dfrac{\Gamma \vdash N : \mathsf{Ok}^c}{\Gamma \vdash M\,(N) : A}\,(\mathsf{App3})$$

(App3) is the only rule with a complement in its premise judgement because we do not spread disjunction here. To see why we can apply (PureDisj) on top of (App3). The resulting rule says we can affirm $M\,(N) : A$ by affirming that $N : B$ s.t. $B \parallel \mathsf{Ok}^c$ i.e. we must show that $N : \mathsf{Ok}$ which is trivially concluded; this is unsound. This unsoundness stems from placing

$$\dfrac{\dfrac{\Gamma \vdash M : B,\ \Delta}{\Gamma,\ M : \mathsf{Num} \vdash \Delta}\,(\mathsf{Disj})\ \mathrm{Num} \parallel B}{\Gamma \vdash M \texttt{ <= 0 ? } N \texttt{ : } P : A,\ \Delta}\,(\mathsf{IfZL1}) \longrightarrow \dfrac{\Gamma \vdash M : B}{\Gamma \vdash M \texttt{ <= 0 ? } N \texttt{ : } P : A}\,(\mathsf{IfZ1})\ \mathrm{Num} \parallel B$$

(IfZ1) we spread disjunction into, and it's the mechanism we use to pull the typing to the right when deriving our one-sided rule. Otherwise, this rule is similar to the two-sided counterpart, except it is more flexible in a system without a separate disjointness rule. The goal of (IfZ1) is to allow any type assignment A to the conclusion if the guard is anything other than a number.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash N : A,\ \Delta}{\Gamma,\ N : A^c \vdash \Delta}\,(\mathsf{CompL}) \quad \dfrac{\Gamma \vdash P : A,\ \Delta}{\Gamma,\ P : A^c \vdash \Delta}\,(\mathsf{CompL})}{\Gamma,\ M \texttt{ <= 0 ? } N \texttt{ : } P : A^c \vdash \Delta}\,(\mathsf{IfZL2})}{\Gamma \vdash M \texttt{ <= 0 ? } N \texttt{ : } P : A,\ \Delta}\,(\mathsf{CompR}) \longrightarrow \dfrac{\Gamma \vdash N : A \quad \Gamma \vdash P : A}{\Gamma \vdash M \texttt{ <= 0 ? } N \texttt{ : } P : A}\,(\mathsf{IfZ2})$$

We use the complement rules to turn the refutation version of the branch type-assignment into the affirmation version. In fact this is the only affirmation rule we derive here as all others are simply identical to their two-sided right-rules. We use the refutation version because it has the correct number of premises to reflect the conditional term rules in Ramsay and Walpole's system for their (IfZ2) (fig. 2.5). Having two rules for conditionals means we can type(IfZ2) them and ill-type(IfZ1) them, naturally giving us more expressive power than if we were just able to refute them or affirm them.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash M_i : B,\ \Delta}{\Gamma \vdash M_i : \mathsf{Num}^c,\ \Delta}\,(\mathsf{Disj})\ \mathrm{B} \parallel \mathsf{Num}}{\Gamma,\ M_i : \mathsf{Num} \vdash \Delta}\,(\mathsf{CompL})}{\Gamma,\ M_1 \circ M_2 : \mathsf{Num} \vdash \Delta}\,(\mathsf{NumPairL})\ \mathrm{i} \in \{1,2\}}{\Gamma \vdash M_1 \circ M_2 : \mathsf{Num}^c,\ \Delta}\,(\mathsf{CompR})\ \circ \in \{+, -\} \longrightarrow \dfrac{\exists i\ .\ \Gamma \vdash M_i : B}{\Gamma \vdash M_1 \circ M_2 : A}\,(\mathsf{NumOp2})\ \circ \in \left\{ \begin{array}{c} + \\ - \end{array} \right\}\ \mathsf{Num} \parallel B$$

The two-sided left-rule (NumPairL) in the middle of this derivation uses the same reasoning as (PairL) in Ramsay's two-sided system (fig. 2.2). Specifically, in the case of pairs, we can refute a pair is of type

$A \times B$ if we can refute the first is an A or the second is a B. If we imagine $A = \mathsf{Num}$ and $B = \mathsf{Num}$ we end up very close to (NumPairL), except we are really typing an addition or subtraction. Therefore the result must be Num for the correct input types Num and Num, and conversely, we can show an addition or subtraction is anything if we can show either input is disjoint to a number. This rule is yet another that disjointness has been built into.

In summary, the changes we see in the one-sided term rules for JSS can be described as changes we make to Ramsay and Walpole's one-sided rules for PCF:

- Remove (OkC2) (see constraint rules Section 3.3), (Contra)
- Remove (Pair) rules (and mimic their reasoning for (NumOp)s)
- Replace fixed-terms with non-recursive abstractions (recursion comes back with block and program rules once added)
- Move disjointness among those rules it would otherwise apply to the premises of, then remove the dedicated rule (Disj) (see constraint rules Section 3.3)
- Remove (Let) rules
- Remove (Succ)/(Pred) rules
- Remove (Zero)
- Add new (Num) and (NumOp) rules
- Modify the syntax of all terms in judgements such that they are vanilla JavaScript

What we are left with is a relatively simple and elegant set of rules with which we type terms in this system, and while we are still able to write and ill-type complicated terms e.g. CompaSS can tell us

```
(y => x => y(1) + y(x))(x => x + 1)((z => z <= 0 ?  (x => x) :  (y => y))(0))
```
: Ok^c

such programs do not look much like any JavaScript I've ever seen. We have already introduced the grammar for programs and blocks in JSS to write more typical programs, and so we now explain the rules that we use to type them in the one-sided system.

3.2.4 Block and Program Rules

Any rules we reference here are included in Figure 3.3 in the previous subsection. The other two key sorts of rules in CompaSS' one-sided system are program and block rules; both of which aim to assign types to *compositions*. These are terms variable assignments joined with semicolons, the likes of which build ordinary JavaScript programs. To disambiguate, we say *terms* are anything in the grammar for terms, *programs* are anything in the whole grammar for JSS, *definitions* are anything beginning with `const`, *blocks* are a composition program that constitutes a function body and has a terminal `return` M`;`, and *statements* are terms like `(x => x);` that evaluate and do not contribute to the type environment at top-level or at block-level. *Top-level* is strictly not in a function body, and *block-level* is in some function body. One reason we make a distinction between program and block composition for the key reason that JavaScript forbids programs from having a top-level return statement outside of a function. However, given that we require all functions to have a return type (there is no void, null or undefined type in our system), this would prove challenging to incorporate into one or two rules for general statement composition. Instead, we split our efforts and create two sets of rules with similar forms but different reasoning.

Block rules are all of the following: (Compo1), (Compo2), (CompoAbs1), (CompoAbs2), (Ret), (Stmt1) and (Stmt2). To start unpacking a block, we begin with the term rule (Blk) which simply says to show block $(\mathcal{B})$: A we must show $\mathcal{B} : A$. To achieve this we assign $\mathcal{B}$ a type with the block rules. We further uncover why there are two shapes for definitions in the grammar, one for variables and one for abstractions. (Compo1) and (Compo2) apply only to blocks starting with a definition which is not a function definition, whereas (CompoAbs1) and (CompoAbs2) apply only to blocks starting with a function definition. It matters more that this is the way things are for the constraint rules (see Section 3.3), but for the one-sided rules it matters for recursion. Given that we want recursion to type interesting programs e.g.:

```
1    const mul = x => y => {
2        return x <= 0 ? y : y + mul(x - 1)(y);
3    }
```

we must allow named function definitions to call themselves, but we don't want a variable to have access to itself during typing as this is not supported by JS. The simplest way to do this is to have a rule for both sorts of definitions.

(Compo1) uses similar reasoning to (LetR) in Ramsay's two-sided type system, except instead of introducing two variables at once via a pair we just define one variable in the current scope. Much like (LetR) we must show that $M : B$ and then the remaining block is of type A given the variable we just defined is a term of type B. (Compo2) does the same thing as (LetL1) except it was complemented for affirmation, i.e. to show the block is of type A we don't need to use the first term. (CompoAbs1) borrows the reasoning from (Compo1) except we do as we promised earlier and insert the type of the function into the environment, the same way Ramsay does with the (FixsR) rule for recursion in their PCF. However, unlike (FixsR) this is a composition so it's required that the rest of the block is typable as an A. (CompoAbs2) is happy to follow (Compo2)'s lead and says we may ignore the function definition if it is not used in the block. (Ret) is what gives all blocks their type; all JSS multi-line functions must have just one final return and this is what allows us to assign blocks their type like this. M must be shown to be of type A given all the things we typed before it in the block. (Stmt1) lets us type blocks that include statements. To show a block starting with M is of type A it's sufficient to assign some type to M then ignore it and continue with typing $\mathcal{B}$. In contrast, (Stmt2) says to show same block beginning with M is of any type we need only show that it is ill-typed; if it goes wrong, diverges or expresses undesired behaviour then we consider the block body it's part of to do the same.

Program rules are just the following: (Defn), (DefnAbs), (TopStmt) and (Stop). We type programs with a type environment, Δ. A complete type derivation for the program $\mathcal{P} : \Delta$ results in a type environment Δ s.t. $\mathsf{dom}(\Delta) =$ (every top-level name in the program $\cup$ all statement labels), as defined in 3.1.3. Starting with the simplest rule, (Stop) allows us to finish a program and places no requirements on the final term. All we do here is mirror the type environment on the left. (Defn) is similar to one of its block-rule counterparts (Compo1), except the variable typing is added to the program type. (DefnAbs) is similar to (CompoAbs1) except again we add this new function typing to the program type. (TopStmt) is our final program rule and says that even though program statements cannot be referenced by the rest of the program (are ignored) they are still able to make the program fail, which is why their typing is added to the program type. CompaSS refers to these unnamed statements in Δ with labels of the form `eval#`n where a fresh n is used each time a new statement is added.

3.3 Constraint and Syntax Directed Rules

Constraint rules are used in a similar way to ordinary rules to build a type derivation. Instead of assigning a type to a term, however, they only ever assign a type variable. These variables are then *constrained* in *type equations* which come with every rule in the system. Type variables are carried up the derivation, after which type constraints accumulate down. They aren't solved in the derivation and instead just recorded for later consideration [15]. Once we show the conclusion we end up with a conclusion type variable X and a set of type equations C in some quantifier-free first-order logical formulation with just conjunction and disjunction. To show that X is of the desired type, we must find a solution to C such that the assignment is consistent with the equations. Explaining by way of example, we give a derivation using the rules for pure unannotated λ-Calculus with Zero as they appear in Chiang's class material [9] in Figure 3.4. We make some minor notational changes to better explain our rules for CompaSS, and remove fixpoints as they are not needed for our example. What we mean when we say *fresh* X in 3.4 is the conclusion type variable X is introduced fresh at this application of the rule, unused in any other rule applications. It is a slight informality that is perfectly correct algorithmically [9].

Definition 3.3.1 (Constraint Type Variables) *Types variables introduced in the conclusions for constraint rules are strings X followed by some base-26 string of uppercase letters e.g. XAB. Type variables introduced by abstraction or definition rules will be an uppercase base-26 string preceded by a Y e.g. YUK. Type variables introduced by top-level rules are preceded by a T e.g. TN. Lastly, type variables introduced by disjointness constraints (see Section 3.3.1) are preceded by a Z or W e.g. ZBN, WBN.*

Definition 3.3.2 (Constraint Judgement) *$\Gamma \Vdash M : X \mid C$ is a constraint judgement which is read "term M has type X under assumptions Γ whenever constraints C are satisfied" [15].*

Calculus Rules

$$\text{(CTVar)}\ \frac{}{\Gamma,\, x : Y \Vdash x : X \mid (X == Y)}\ \text{fresh X} \qquad \text{(CTZero)}\ \frac{}{\Gamma \Vdash 0 : X \mid (X == \mathsf{Nat})}\ \text{fresh X}$$

$$\text{(CTAbsInf)}\ \frac{\Gamma,\, x : Y \Vdash M : T_2 \mid C}{\Gamma \Vdash \lambda x.M : X \mid (X == Y \to T_2) \wedge C}\ \begin{array}{l}\text{fresh } X \\ x \notin \mathsf{dom}(\Gamma)\end{array}$$

$$\text{(CTApp)}\ \frac{\Gamma \Vdash M : T_1 \mid C_1 \quad \Gamma \Vdash N : T_2 \mid C_2}{\Gamma \Vdash MN : X \mid (T_1 == T_2 \to X) \wedge C_1 \wedge C_2}\ \text{fresh } X$$

Figure 3.4: Pure calculus rules as constraint rules from [9] with minor adjustments.

We derive the type constraints for $(\lambda x.x)0$ and solve them to conclude its type Nat. Since these constraint rules will generate many type variables, a fresh for every rule used, we note that type variables generated by CompaSS respect Definition 3.3.

$$\text{(CTApp)}\ \frac{\text{(CTAbsInf)}\ \frac{\text{(CTVar)}\ \frac{}{x : YA \Vdash x : XC \mid XC == YA}}{\Vdash \lambda x.x : XB \mid (XB == YA \to XC) \wedge (XC == YA)} \qquad \text{(CTZero)}\ \frac{}{\Vdash 0 : XD \mid XD == \mathsf{Nat}}}{\Vdash (\lambda x.x)0 : XA \mid (XB == XD \to XA) \wedge (XB == YA \to XC) \wedge (XC == YA) \wedge (XD == \mathsf{Nat})}$$

Which, almost like we were doing a type derivation in a very general way, gives us the bottom set of equations to solve. Such a solution we term θ, where we aim to find a θ s.t. $\theta \models C$ and furthermore that the conclusion type variable $XA \mapsto \mathsf{Nat}$. Constraint solving is at a syntactic level in CompaSS and so we reason about it in the same way here.

$$\begin{aligned} \text{Let } & XD = \mathsf{Nat}, XB = XD \to XA, XB = YA \to XC \\ \text{Therefore } & YA = XD, XA = XC \\ \text{Let } & XC = YA \\ \text{Therefore } & XA = XC = YA = XD = Nat_{\square} \end{aligned}$$

This may seem like just a long-winded way to achieve what was already possible with ordinary rules, but this method of type derivation lends itself perfectly to an algorithm. Specifically in this case where we only have conjunction in our type constraints, *unification* may be used. Type unification is a syntactic replacement algorithm for determining the single type that the conclusion must be. We will not go over it here because we actually cannot use unification to solve the constraints CompaSS' gives us, since such constraints always include disjunction as well as conjunction. There is no disjunction in Figure 3.4 because there is a one-to-one mapping between ordinary rules and constraint rules even with the requirement of syntax direction. This is not the case when converting CompaSS' one-sided rules.

Constraint rules were chosen as a mechanism to convert the system to a syntax-directed system, and so make constraint generation and constraint resolution possible algorithmically (the latter using Z3 [10], see Section 3.5). This means we must combine a number of rules that apply to the same grammar shape, and the way we must do that is by disjunctively combining the constraints from the otherwise separate constraint-rule versions of the ordinary system. Intuitively, assuming we can make any ordinary rule into a constraint-version of itself, if we just did that with CompaSS' one-sided system we end up with having multiple rules apply to some grammar shapes; this rule *or* this rule apply. Thus disjunction in the type constraints allows us to represent all the possible constraint derivation trees that could exist for a term, without our type derivation exploding in size every time a shape which had many rules in the ordinary system is encountered. Instead, the number of constraints explodes. But we handle them using a separate more optimised system to solve them, in this case, Microsoft Research's Satisfiability Modulo Theorems Solver Z3 [10]. A requirement for combining the rules is we need to be able to write the premise and conclusion of all the rules we want to combine in a general enough way to allow us to do so. The differences between the rules are then offloaded to the constraints. This generalisation manifests by all

Structural Constraints

$$\mathsf{Disj}_{\text{fresh } Z,W}(A,B) = \begin{array}{c}(A == \mathsf{Num} \wedge B == Z \to W) \vee (A == Z \to W \wedge B == \mathsf{Num}) \vee \\ (A == B^c) \vee (A^c == B)\end{array}$$

$$\mathsf{Struct}(\Gamma, X) = (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$$

Term Rules

$$(\mathsf{CTVar})\ \frac{}{\Gamma,\ x : Y \Vdash x : X \mid (X == Y) \vee \mathsf{Struct}(\Gamma, X)}\ \text{fresh } X$$

$$(\mathsf{CTNum})\ \frac{}{\Gamma \Vdash n : X \mid (X == \mathsf{Num}) \vee \mathsf{Struct}(\Gamma, X)}\ \text{fresh } X$$

$$(\mathsf{CTAbsInf})\ \frac{\Gamma,\ x : Y \Vdash M : T_1 \mid C_1}{\Gamma \Vdash x \texttt{ => } M : X \mid \ (X == Y \to T_1 \wedge C_1) \vee \mathsf{Struct}(\Gamma, X)}\ \begin{array}{l} x \notin \mathsf{dom}(\Gamma) \\ \text{fresh } X, Y\end{array}$$

$$(\mathsf{CTApp})\ \frac{\Gamma \Vdash M : T_1 \mid C_1 \quad \Gamma \Vdash N : T_2 \mid C_2}{\Gamma \Vdash M\texttt{(}N\texttt{)} : X \mid \begin{array}{c}(T_1 == T_2 \to X \wedge C_1 \wedge C_2) \vee (\mathsf{Disj}(T_1, \mathsf{Ok}^c \to X) \wedge C_1) \vee \\ (T_2 == \mathsf{Ok}^c \wedge C_2) \vee \mathsf{Struct}(\Gamma, X)\end{array}}\ \text{fresh } X$$

$$(\mathsf{CTIfZ})\ \frac{\Gamma \Vdash M : T_1 \mid C_1 \quad \Gamma \Vdash N : T_2 \mid C_2 \quad \Gamma \Vdash P : T_3 \mid C_3}{\Gamma \Vdash M \texttt{ <= 0 ? } N \texttt{ : } P : X \mid \begin{array}{c}(T_2 == T_3 \wedge T_2 == X \wedge C_2 \wedge C_3) \vee \\ (\mathsf{Disj}(T_1, \mathsf{Num}) \wedge C_1) \vee \mathsf{Struct}(\Gamma, X)\end{array}}\ \text{fresh } X$$

$$(\mathsf{CTNumOp})\ \frac{\Gamma \Vdash M : T_1 \mid C_1 \quad \Gamma \Vdash N : T_2 \mid C_2}{\Gamma \Vdash M \circ N : X \mid \begin{array}{c}(X == \mathsf{Num} \wedge T_1 == \mathsf{Num} \wedge T_2 == \mathsf{Num} \wedge C_1 \wedge C_2) \vee \\ (\mathsf{Disj}(T_1, \mathsf{Num}) \wedge C_1) \vee (\mathsf{Disj}(T_2, \mathsf{Num}) \wedge C_2) \vee \\ \mathsf{Struct}(\Gamma, X)\end{array}}\ \begin{array}{l}\circ \in \{\texttt{+}, \texttt{-}\} \\ \text{fresh } X\end{array}$$

$$(\mathsf{CTBlk})\ \frac{\Gamma \Vdash \mathcal{B} : T_1 \mid C_1}{\Gamma \Vdash \texttt{\{}\mathcal{B}\texttt{\}} : X \mid (X == T_1 \wedge C_1) \vee \mathsf{Struct}(\Gamma, X)}\ \text{fresh } X$$

Figure 3.5: Constraint rules for the terms of JSS.

conclusions having type X and that some disjunctive constraint sets don't use all the conclusions of the rule; looking at figures 3.5 and 3.6 we see the constraint rules that CompaSS uses in constraint generation. These rules maintain the same *term*, *block* and *program* rule groups from the ordinary rules. We also add constraint functions in their own *Structural Constraints* division as they are reused in many rules.

3.3.1 Constraint Rules from Ordinary Rules

We go through the three groups of rules once more for this final transformation of the rules, elucidating the structure and reasoning behind the constraint rules for JSS, the likes of which CompaSS actually implements.

Constraint Term Rules (CTVar) combines (Var), (Ok), and (OkC1). (Var) is encoded by the constraint $(X == Y)$, and $\mathsf{Struct}(\Gamma, X)$ encodes the latter two. Note Struct is included in the constraints of every rule because (Ok) and (OkC1) have the most general term shape in the conclusion: M. Since these rules are syntax-directed they must capture the behaviour of all the original one-sided rules that could have applied to the term in the conclusion. (Ok) and (OkC1) make no assumptions about the term, apply to all terms and thus appear in all rules. In a similar vein, the original one-sided system used to have a

Structural Constraints

$$\mathsf{Struct}(\Gamma, X) = (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$$

Block Rules

$$\text{(CTCompo)}\ \frac{\Gamma \Vdash M : T_1 \mid C_1 \quad \Gamma, x : Y_2 \Vdash \mathcal{B} : T_2 \mid C_2 \quad \Gamma \Vdash \mathcal{B} : T_3 \mid C_3}{\Gamma \Vdash \texttt{const}\ x\ \texttt{=}\ M\texttt{;}\ \mathcal{B} : X \mid \begin{array}{c}(C_1 \wedge C_2 \wedge T_1 == Y_2 \wedge X == T_2) \vee \\ (X == T_3 \wedge C_3) \vee \mathsf{Struct}(\Gamma, X)\end{array}} \quad \begin{array}{l} x \notin \mathsf{dom}(\Gamma) \\ \text{fresh } X, Y_2 \end{array}$$

$$\text{(CTCompoAbs)}\ \frac{\Gamma, y : Y_1 \to Y_2 \Vdash x\ \texttt{=>}\ N : T_1 \mid C_1 \quad \Gamma, y : Y_3 \Vdash \mathcal{B} : T_2 \mid C_2 \quad \Gamma \Vdash \mathcal{B} : T_3 \mid C_3}{\Gamma \Vdash \texttt{const}\ y\ \texttt{=}\ x\ \texttt{=>}\ N\texttt{;}\ \mathcal{B} : X \mid \begin{array}{c}(X == T_2 \wedge Y_1 \to Y_2 == Y_3 \wedge \\ T_1 == Y_3 \wedge C_1 \wedge C_2) \vee \\ (X == T_3 \wedge C_3) \vee \mathsf{Struct}(\Gamma, X)\end{array}} \quad \begin{array}{l} y \notin \mathsf{dom}(\Gamma) \\ \text{fresh } X, Y_1 \\ Y_2, Y_3 \end{array}$$

$$\text{(CTRet)}\ \frac{\Gamma \Vdash M : T_1 \mid C_1}{\Gamma \Vdash \texttt{return}\ M\texttt{;} : X \mid (X == T_1 \wedge C_1) \vee \mathsf{Struct}(\Gamma, X)} \quad \text{fresh } X$$

$$\text{(CTStmt)}\ \frac{\Gamma \Vdash M : T_1 \mid C_1 \quad \Gamma \Vdash \mathcal{B} : T_2 \mid C_2}{\Gamma \Vdash M\texttt{;}\ \mathcal{B} : X \mid (X == T_2 \wedge C_1 \wedge C_2) \vee (T_1 == \mathsf{Ok}^c \wedge C_1) \vee \mathsf{Struct}(\Gamma, X)} \quad \text{fresh } X$$

Program Rules

$$\text{(CTDefn)}\ \frac{\Gamma \Vdash M : X \mid C_1 \quad \Gamma, g : T_1 \Vdash \mathcal{P} : \Delta \mid C_2}{\Gamma \Vdash \texttt{const}\ g\ \texttt{=}\ M\texttt{;}\ \mathcal{P} : \{g : T_1\} \cup \Delta \mid (X == T_1 \wedge C_1 \wedge C_2)} \quad \begin{array}{l} g \notin \mathsf{dom}(\Gamma) \\ \text{fresh } X, T_1 \end{array}$$

$$\text{(CTDefnAbs)}\ \frac{\Gamma, f : T_1 \to T_2 \Vdash x\ \texttt{=>}\ N : X \mid C_1 \quad \Gamma, f : T_1 \to T_2 \Vdash \mathcal{P} : \Delta \mid C_2}{\Gamma \Vdash \texttt{const}\ f\ \texttt{=}\ x\ \texttt{=>}\ N\texttt{;}\ \mathcal{P} : \{f : T_1 \to T_2\} \cup \Delta \mid (X == T_1 \to T_2 \wedge C_1 \wedge C_2)} \quad \begin{array}{l} f \notin \mathsf{dom}(\Gamma) \\ \text{fresh } X, T_1 \\ T_2 \end{array}$$

$$\text{(CTTopStmt)}\ \frac{\Gamma \Vdash M : X \mid C_1 \quad \Gamma \Vdash \mathcal{P} : \Delta \mid C_2}{\Gamma \Vdash M\texttt{;}\ \mathcal{P} : \{\texttt{eval\#}n : T_1\} \cup \Delta \mid (X == T_1 \wedge C_1 \wedge C_2)} \quad \begin{array}{l} \text{next } n \\ \text{fresh } X, T_1 \end{array}$$

$$\text{(CTStop)}\ \frac{}{\Gamma \Vdash \varepsilon : \Gamma \mid (\top)}$$

Figure 3.6: *Program* rules for the system for program type assignment, and *Block* rules for function bodies terminating in a `return`.

separate rule for disjunction we called (PureDisj), but this rule had an issue which prevented us from using it as a constraint rule: its premise was the same shape as its conclusion. Noting this reveals we cannot perform constraint-rule conversion on rules whose premises are not strictly smaller than their conclusion. Otherwise, the constraint generation process would definitely never terminate. For example, if we kept (PureDisj) it would apply to every term since its conclusion was of shape M. We would then add the constraints for (PureDisj) to all the rules and also require they all have an additional premise which is equal to the conclusion. We could then apply all the rules which were just applied to the conclusion to this new premise over again ad infinitum. We also introduce a fresh X as in all constraint rules to avoid type-variable capture between rule applications. This is not a problem for the small programs we look at here as we have a denumerable set of types as described in Definition 3.3; type variable names never exceed three/four characters during testing.

(CTNum) combines (Num), (Ok), and (OkC1). Noting it would not have been possible to use Num here instead of X for the conclusion type as Chiang does with his (CTZero) rule [9] because (Ok) requires the type is not interpreted at type-reconstruction-time, otherwise their constraints would always be false (Num $\neq$ Ok). We would not make a significant saving either way, since we only add one constraint per branch and per number-literal, in many hundreds of other constraints.

(CTAbsInf) combines (Abs), (Ok), and (OkC1). We introduce an additional fresh Y type variable for a free type to give our new variable in the assumptions. Type variables in assumptions can be constrained in the same way as type variables in the conclusion, and (OkC1) relies on this too. Turning our attention to Struct again we read the second clause as saying if we can type anything in the environment as Ok then we get that the constraints for that rule "vacuously" hold, by the fact it's disjunctive. This idea of making constraints hold without constraining their conclusion type X is one method CompaSS uses to assign Ok^c to terms; further information is given in Section 3.5.

(CTApp) combines (App1), (App2), (App3), (Ok), and (OkC1). The first disjunctive clause corresponds to (App1) and the second to (App2) and so on. Thereby we should appreciate that (CTApp) is a powerful rule since it captures the power of affirmation of evaluation, and two modes of refuting evaluation. The first refutation strategy adds the constraints that represent disjointness. These are a structural constraint that is used just in the constraint rules which have subsumed those with a disjoint side condition like (App2) and (IfZ1). Investigating the constraints for disjointness, we get what we might expect represents syntactic disjointness: A and B are disjoint just if A is a number and B is a function, A is a function and B is a number, or that one is the complement of the other.

(CTIfZ) combines (IfZ1), (IfZ2), (Ok) and (OkC1). We again see disjointness apply to mirror (IfZ2)'s side condition that the guard is of some type disjoint to Num, which if true satisfies the constraints without restricting X, allowing CompaSS to assign the term to Ok^c if it needs to in proving incorrectness.

(CTNumOp) combines (NumOp1), (NumOp2), (Ok) and (OkC1). In the original one-sided system (NumOp2) is already disjunctive in a sense because we can show $M_1 : B$ s.t. $B \parallel \mathsf{Num}$ *or* show the same thing about M_2. This means we end up with three disjunctive constraint clauses (plus (Ok) and (OkC1)), one from (NumOp1) affirming the conclusion type is Num just if both the premises are, and the other two from (NumOp2). This is really quite elegant since we don't need any special mechanisms or additional rules to represent this behaviour inherited from (NumOp2); constraints combined with even just a subset of quantifier-free first-order logic ($\wedge, \vee$) can be quite flexible.

(CTBlk) just combines (Blk) with the same (Ok) and (OkC1) structural constraints as all the others discussed so far. It is probably the simplest rule with a premise in the constraint rules for terms.

See Appendix A for the soundness and completeness of the term constraint rules with respect to the original one-sided system.

Constraint Block Rules (CTCompo) combines (Compo1), (Compo2), (Ok), and (OkC1). In the conclusion we are sure to combine the constraints from the leftmost branch which are generated from typing the body of the assignment so that it is consistent with the rest of the constraints we make from the other branches. Except in the case of the rightmost premise, which to encode (Compo2) disregards the current definition and carries on typing the block, and this is in a separate disjunctive clause of course. We require an additional free Y type variable because we add a typing of what we define to the second premise's assumptions.

(CTCompoAbs) requires a lot of free variables, which is primarily to do with needing to infer that the type of y in the leftmost branch is a function. When developing the block and composition rules at one point there was the issue of (OkC1) being able to type all our named function definitions as ill-typed in the leftmost premise, because there were no constraints put into this branch that told us it must be a function (and thus does not fail without being fully applied). Putting y into the assumptions as a variable of type

$Y_1 \to Y_2$ means the constraints generated by (OkC1), which try to ill-type y, are always unsatisfiable (since syntactically $A \to B \neq \mathsf{Ok}^c$ for any instantiation of A, B). This rule combines (CompoAbs1), (CompoAbs2), (Ok) and (OkC1).

(CTRet) combines (Ret), (Ok), and (OkC1). Much like (Ret) it primarily serves to constrain the type of the block and thus its assignee's return type, which must be equal to that of the term preceded by the return, M. Struct allows for an unconstrained return type just if we may set any types in the environment to Ok^c.

(CTStmt) combines (Stmt1), (Stmt2), (Ok), (OkC1). Thereby it is capturing the power to affirm or refute that a statement will cause a function body to crash, or whether it will leave it unaffected and can be solved consistently.

Constraint Program Rules (CTDefn) does not combine any rules and, like all program rules, represents a direct conversion from the one-sided system. Its primary purpose apart from building the program type Δ is accumulating the constraints for all the type variables that appear in Δ, conjoining them throughout constraint generation. We make a fresh type variable Y_1 for the remaining program to use the current statement, then finally equate it to the left-premise type once we have completed constraint generation for the right premise. We do this in the term rules and block rules out of necessity because they must represent multiple rules. This is not the case here, so we could have perhaps used X in the places that we see Y_1 in this rule, and (CTTopStmt), but we maintain this philosophy of maximal genericness for consistency with the other sorts of rule.

(CTDefnAbs) is similar to (CTDefn) except that we take a similar approach to (CTCompoAbs) and introduce this fresh arrow type into the environment to allow us to type recursive functions. This same fresh type is again what we add to Δ, and then we constrain it to equal the left premise's conclusion type.

(CTTopStmt) uses *next* n to mean we increment n to create a denumerable set of fresh labels each time we encounter a statement that evaluates and doesn't form a definition. An additional constraint is added besides just C_1 for the constraints of the left premise and C_2 for the right, with which we require the type in the program type Δ of the evaluation to match what we infer from the left premise.

(CTStop) provides a true conjoined to the end of whatever program we finish (of course not affecting the constraint satisfiability) and returns the program type unmodified. Note this is the only rule in the whole constraint system which does not introduce a fresh type variable.

3.3.2 Example Syntax Directed Rule Derivation

Once CompaSS has generated the constraints for a program using the system in figures 3.6 and 3.5 it will then try solving these constraints. Before doing so it conjoins the constraints with another disjunctive set of type equations that collectively mandates at least one of the top-level definition or statement type variables in Δ be equal to Ok^c.

Definition 3.3.3 (Forced Ill-Typeness) *We force ill-typing specifically by mandating that at least one of the definitions or statements in the program's type is ill-typed:*

$$\mathsf{IllTy}(\Delta, C) ::= C \wedge (\bigvee \{T == \mathsf{Ok}^c \mid T \in \mathsf{range}(\Delta)\})$$

where C is the result of finished constraint generation

When we say range of Δ we mean specifically extracting the set of type variables to which the defined variables and evaluation labels map. Applying IllTy to the program type and the constraints returns a new set of constraints which we then try to solve. One may start to see why in the examples in subsections 1.1.1, 1.1.5 and 1.1.7 given in the introduction CompaSS do not give any type information and instead give us that the program type is `Inconclusive`. This is the case just if there was no solution to the constraints, which we now see is just if CompaSS could not assign at least one top-level type variable to Ok^c. This ought not to be conflated with the true-negatives theorem in Ramsay and Walpole's two-sided system, since we gave up that power in introducing success semantics. Rather, forcing ill-typedness guarantees that we are only able to prove incorrectness and thus we only return assignments to typings in Δ exactly when we are sure of them (just when the program definitely fails because of them).

We give an example constraint generation derivation of a small correct program using the rules given in figures 3.5 and 3.6. Even small programs such as this generate a large number of constraints where we apply rules like (CTNumOp) and (CTApp) which represent many rules in the one-sided system and so introduce multiple disjunctive clauses each time. Thus we are restricted to small derivations on paper, but by giving an example we hope to give better intuition for what constraint generation looks like.

We look first at this program which we can see definitely does not fail:

$$\textsf{Gen}_1\quad \textsf{(CTAbsInf)}\ \frac{\textsf{(CTVar)}\ \dfrac{}{x : YE \Vdash x : XF \mid (XF == YE) \vee (XF == \textsf{Ok}) \vee (YE == \textsf{Ok}^c) \text{ hereafter } C_1}}{\Vdash \texttt{x => x} : XD \mid ((C_1 \wedge XD == YE \to XF) \vee XD == \textsf{Ok}) \text{ hereafter } C_2}$$

$$\textsf{Gen}_2\quad \textsf{(CTVar)}\ \frac{}{\texttt{id} : TB \to TC \Vdash \texttt{id} : XK \mid (XK == TB \to TC) \wedge (XK == \textsf{Ok}) \text{ hereafter } C_3}$$

$$\textsf{Gen}_3\quad \textsf{(CTNum)}\ \frac{}{\texttt{id} : TB \to TC \Vdash \texttt{0} : XL \mid (XL == \textsf{Num}) \wedge (XL == \textsf{Ok}) \quad \text{hereafter } C_4}$$

$$\textsf{Gen}_4\quad \textsf{(CTApp)}\ \frac{\textsf{Gen}_3 \quad \textsf{Gen}_4}{\texttt{id} : TB \to TC \Vdash \texttt{id(0)} : XJ \mid \begin{array}{c}(C_3 \wedge C_4 \wedge XK == XL \to XJ) \vee (XJ == \textsf{Ok}) \vee \\ (\textsf{Disj}(XK, \textsf{Ok}^c \to XG) \wedge C_3) \vee (XK == \textsf{Ok} \wedge C_4)) \\ \text{hereafter } C_5\end{array}}$$

$$\textsf{Gen}_5\quad \textsf{(CTTopStmt)}\ \frac{\textsf{Gen}_4 \quad \textsf{(CTStop)}\ \dfrac{}{\texttt{id} : TB \to TC \Vdash \varepsilon : \{\texttt{id} : TB \to TC\} \mid \top}}{\begin{array}{l}\texttt{id} : TB \to TC \Vdash \texttt{id(0);} : \{\texttt{id} : TB \to TC, \texttt{eval\#0} : TI\} \mid \\ (XJ == TI) \wedge C_5 \wedge \top \quad \text{hereafter } C_6\end{array}}$$

$$\textsf{Gen}_6\quad \textsf{(CTDefnAbs)}\ \frac{\textsf{Gen}_1 \quad \textsf{Gen}_5}{\begin{array}{l}\Vdash \texttt{const id = (x => x); id(0);} : \{\texttt{id} : TB \to TC, \texttt{eval\#0} : TI\} \mid \\ (XD == TB \to TC) \wedge C_2 \wedge C_6\end{array}}$$

Figure 3.7: An example derivation of `const id = (x => x); id(0);` using CompaSS' constraint system.

```
1 const id = (x => x);
2 id(0);
```

CompaSS internally produces the following constraint generation tree we see in Figure 3.7 which we construct in parts in the interest of compactness. After the constraint generation, we get $(XD = TB \to TC) \wedge C_2 \wedge C_6$ which we will call C. We then apply $\mathsf{IllTy}(\{\texttt{id} : TB \to TC, \texttt{eval\#0} : TI\}, C) = C \wedge (TB \to TC == \mathsf{Ok}^c \vee TI == \mathsf{Ok}^c)$, which mandates that at least one definition or statement in the program fails to evaluate correctly. Even though it might be able to solve the constraints, it cannot do so and thus cannot show ill-typedness in this example. Recall we must satisfy the new constraints IllTy adds as well as those generated in the tree and so we end up with no solutions with the following output from CompaSS:

```
    id:  Untypable
 eval0:  Untypable
Inconclusive
```

This is what we expect for this program since it does not diverge, go wrong or use undesired behaviour which we can determine by inspection for this program.

3.4 From JS to Constraints

The development methodology of CompaSS was primarily outward rather than forward. What we mean by this is we chose to implement the whole program for the minimal grammar (JSS terms i.e. without composition), and then extend it width-ways once we had the whole pipeline built. This was a good decision in hindsight since it allowed us to experiment with methods of constraint solving early on, rather than just at the end of development. Initially, we had unification as our method of type resolution, but this process only acts by syntactic replacement so introducing disjunction in our constraints meant we could not continue using it. We instead opted for a solver which is able to handle quantifier-free first-order logic. Section 3.4.1 explores how we use the abstract syntax tree of JS to develop a way to explore, restructure and describe JSS within JS. We discuss our constraint generation algorithm in CompaSS, and elucidate our constraint resolution algorithm in the following Section 3.5.

3.4.1 AST

Acorn AST (Abstract Syntax Tree) Walker [1] was used to turn strings of JavaScript programs into AST objects. Such ASTs naturally have nodes of different kinds, the specific ones of interest to us for full JSS are terms found in the following node types, which we formalise as a grammar:

Definition 3.4.1 (AST Node Types) *Below are the types of nodes that are allowed to be in a JSS abstract syntax tree:*

(Node Types) $K ::=$ `Literal` | `BinaryExpression` | `ArrowFunctionExpression` | `CallExpression` | `ConditionalExpression` | `Identifier` | `VariableDeclaration` | `BlockStatement` | `ReturnStatement` | `ExpressionStatement`

Below we then give the correspondence between node types and JSS grammar forms. Note unfortunately these don't map one-to-one, which we will handle with term shape resolution (inspecting node fields) in the case of `VariableDeclaration` and `ExpressionStatement`:

JSS Grammar Type	JSS AST Node Type
$\underline{n}$	`Literal`
$M \circ N$	`BinaryExpression`
x `=>` M	`ArrowFunctionExpression`
M`(`N`)`	`CallExpression`
M `<= 0 ?` N `:` P	`ConditionalExpression`
x	`Identifier`
`const` g `=` x `=>` M`;`$\mathcal{P}$ `const` f `=` M''`;`$\mathcal{P}$ `const` y `=` x `=>` M`;`$\mathcal{B}$ `const` x `=` M''`;`$\mathcal{B}$	`VariableDeclaration`
`{`$\mathcal{B}$`}`	`BlockStatement`
`return` M`;`	`ReturnStatement`
M'`;`$\mathcal{P}$ M'`;`$\mathcal{B}$	`ExpressionStatement`

During the entire derivation, we keep terms in their raw AST form. This decision was made as its recursive structure was close to the desired shape with respect to traversal; nodes include their subterms in all cases except two: top-level statements and blocks. These are stored in the AST as arrays, which is a departure from our recursively-defined grammar. To recover this recursive structure for blocks, we simply work backwards from the return statement along the array placing the node before into a "next" field we add to each node in the block. The return statement does not get a next field because like in the grammar it terminates the block.

We choose to keep programs as arrays of definition and statement ASTs, even though this is different to how the grammar is defined. This was done because it was more elegant algorithmically to exactly replicate the recursive behaviour of the program rules with a loop instead, thanks to the one-to-one correspondence between top-level terms and typings in the program type. This is the only time we derive constraint trees non-recursively, and for all other rules i.e. term and block rules we maintain the recursive behaviour through recursive calls and data transfer. Recursion benefits us in many ways e.g. we create a class with all the rules giving us a nearly one-to-one mapping from theory to code, reducing the chance of making implementation mistakes. Furthermore, state management for the scoping of rule variables arises naturally from the recursive call structure.

3.4.2 Grammar Specification as JSON

There are three JSON files that describe what JSS is and how we should use it with respect to the AST. `AST_grmmr` lets us map any AST node to a term shape according to the table above. `AST_subtm` tells us what field accesses to make inside AST nodes to fetch child nodes corresponding to the subterms in the grammar shape. `AST_require` restricts what AST nodes we treat as invalid under JSS by allowing us to exclude values and types of fields with a primitive "meta-language" which is covered below.

We restrict the AST to check that inputs are valid JSS and don't violate the grammar. This works by loading the JSON with the language specification, whereby one can specify restrictions on the fields in the AST node types K. There are just two checks we can do on fields of AST nodes with this meta-language: equality and JavaScript type. These checks are done at the end of a chain of field accesses, and these checks are grouped under several logical reductions: *ANY*, *ALL*, and *NONE*. This lets us specify things we want to be true and false about each AST node object which is expressive enough to express JSS almost entirely. There is the caveat that once we added the block and the program extensions it was no longer expressive enough to verify the full language; e.g. it was not able to describe we wanted all blocks to terminate with a return. But its merit comes from its ease of configuration; we were able to edit the grammar early on without changing any code and instead just modifying these AST restrictions. Being able to make easy changes to the grammar lends itself to future extensions of JSS, too.

An example of such a restriction on fields of an AST node is for `ConditionalExpression`s, where we want to limit the kinds of such nodes that we accept. We show a small part of the meta-language for JSS and then explain its components.

```
"ConditionalExpression": {
    "test": { // top-level field immediately accessible in the node
        "satisfies": "ALL", // logical reducer
        "rules": [ // checks
            {"P": {"type": {"F": {"equals": "BinaryExpression"}}}},
            {"P": {"operator": {"F": {"equals": "<="}}}},
            {"P": {"right": {"P": {"raw": {"F": {"equals": "0"}}}}}}
        ]
    }
}
```

Recall that `ConditionalExpression`s correspond to the grammar shape M `<= 0 ?` N `:` P i.e. our conditional. In this snippet we place three restrictions on the `test` field in ternary expressions (M `<= 0`), that it must be a `BinaryExpression`, that the comparison operator must be `<=`, and that the right argument must be a `0`. Thus we have the requirements for our JSS conditional if-zero statement. The preceding P and F letters dictate whether the following string must be a property accessor or a function to apply to the current property, respectively.

With a grammar checker in hand, we move on to the method CompaSS uses to generate constraints.

3.4.3 Constraint generation

We use the Algorithm 1 for processing the constraints generated by the premises of the program rules. This is the last step before passing them on to the Z3 Solver [10]. We introduce this first because it's the routine that calls our recursive constraint generation. While Algorithm 1 is not recursive like the program rules would be when applied manually, the results are the same as if they were.

We pass an "*empty*" judgement of the form $\Gamma \Vdash M$ to the constraint generation procedure. This call takes us to Algorithm 1. We call a judgement "$full$" just if it has a conclusion type and a set of constraints, so of the form $\Gamma \Vdash M : X \mid C$. This is what constraint generation results in when we apply the rules by hand, so it ought to make sense it is what we do in the program as well.

We make a distinction between top-level assumptions and statement assumptions. This is done to make sure we do not add references to statements in the type environment when generating constraints for a new term in the program. Because statements are evaluated without being assigned to a global variable, they cannot be referenced elsewhere, and this is reflected in the rules in Figure 3.6 (CTTopStmt) (the same is true for the block rules, seen in (CTStmt)). To see more clearly why we need this, suppose that we did not make a distinction and add all typings in the program (all top-level assumptions) to the environment of each term in a composition. then suppose we have the program `0; const x = eval#0;` where we deliberately try to capture the statement by its label. Even though it's illegal in JS to name a variable with a hashtag, it's still possible to run CompaSS on this input. With no distinction between assumptions, CompaSS would generate constraints requiring $\Delta($`eval#0`$) == \Delta(x)$ which is nonsense, because `eval#0` is not a definition in the program and thus cannot be referred to.

Next, we look at the aforementioned recursive constraint generation, Algorithm 2 and example rule 3. We pass this rule the same empty judgement GENERATE was given. In CompaSS we also pass in a reference to the current class, which allows the rule MAYBERULE to call GENERATE inside before returning. This allows the recursive behaviour that we wanted, closely representing a real constraint generation tree. We implement all remaining constraint rules one-to-one, which is made less error-prone thanks to the high-level functions for handling the raw AST CompaSS makes available. Algorithm 3 is pseudocode for the rule (CTApp) in the *Rule* class. Not only are the larger constraint rules implemented, but they are built up of individual conjunction-only constraint versions of the original rules in Figure 3.3. These are connected disjunctively inside all the larger rules, which allows us to reuse common rules e.g. (Ok), (OkC1) that appear in Struct. Where rules use $C_1, C_2, C_3 \ldots$ we refer to the constraints from a premise which GENERATE() has been run on prior. Where we use $T_1, T_2, T_3 \ldots$ we refer to the type variable of a premise, where we keep the names consistent between the rules and the implementation.

Once this process is complete we end up with a complete constraint generation tree and a program type whose term types are all type variables. The constraints are returned via $finalJudge$ in Algorithm 1, which then go to our constraint-solving algorithm utilising Z3 SMT solver [10]. We cover this algorithm in the following Section 3.5.

Algorithm 1 Apply Program Rules

procedure RECONSTRUCT($prog$)
  Reset current fresh variable counter
  $terms \leftarrow$ array of abstract syntax trees, one for every term in the program
  if there are no terms in the program **then**
    return
  end if
  Let $labels$ be a map from term $\rightarrow$ variable name or statement label
  Let $stmtLabels$ be a map from term $\rightarrow$ statement label
  $\Delta_{\mathsf{Top}} \leftarrow$ empty type environment to store top-level definition typings
  $\Delta_{\mathsf{Stmt}} \leftarrow$ empty type environment to store top-level statement typings
  $C_{\mathcal{P}} \leftarrow$ empty set of disjunctions for the constraints of each program term
  Let $finalJudge$ be a judgement which the final term is assigned to
  for $term$ in $terms$ **do**
    Let T_1, T_2 be fresh type variables
    if $term$ is an abstraction **then**
      Let T be $T_1 \rightarrow T_2$
      $\Delta_{\mathsf{Top}} \leftarrow \Delta_{\mathsf{Top}} \cup labels[term] : T$
    else
      Let T be T_1
    end if
    $\Delta_{\mathsf{Top}} \Vdash M : X \mid C \leftarrow$ GENERATE($\Delta_{\mathsf{Top}} \Vdash M$) ▷ where M subterm "M'' in $term$ by the grammar
    $C' \leftarrow C \cup (X == T)$
    if $term$ is a variable definition which is not an abstraction **then**
      $\Delta'_{\mathsf{Top}} \leftarrow \Delta_{\mathsf{Top}} \cup labels[term] : T$
    end if
    if $term$ is a statement **then**
      $\Delta'_{\mathsf{Stmt}} \leftarrow \Delta_{\mathsf{Stmt}} \cup stmtLabels[term] : T$
    end if
    $C'' \leftarrow C' \cup \bigwedge C_{\mathcal{P}}$
    $C_{\mathcal{P}} \leftarrow C_{\mathcal{P}} \cup C'$
    $C \leftarrow C''$
    $finalJudge \leftarrow \Delta_{\mathsf{Top}} \Vdash M : X \mid C$
    $\Delta_{\mathsf{Top}} \leftarrow \Delta'_{\mathsf{Top}}$
    $\Delta_{\mathsf{Stmt}} \leftarrow \Delta'_{\mathsf{Stmt}}$
  end for
  $\Delta_{\mathsf{Top}} \leftarrow \Delta_{\mathsf{Top}} \cup \Delta_{\mathsf{Stmt}}$
  return {"$judgement''$: $finalJudge$, "$delta''$: Δ_{Top}} ▷ Multiple return

Algorithm 2 Constraint Generation

procedure GENERATE($\Gamma \Vdash M$)
  MAYBERULE $\leftarrow$ the constraint rule that applies to the shape of M
  if there is a rule that applies to this shape **then**
    $\Gamma \Vdash M : X \mid C \leftarrow$ MAYBERULE($\Gamma \Vdash M$)
    return $\Gamma \Vdash M : X \mid C$
  end if
  THROWERROR($message =$ "Unrecognised grammar shape")

Algorithm 3 Example Rule (CTApp)

procedure CTApp($\Gamma \Vdash M$)
- $C_{\mathsf{OkC1}} \leftarrow \bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\}$ ▷ CTOkC1
- $X \leftarrow$ fresh type variable
- M', $N' \leftarrow$ "M'' in M, "N'' in M
- $\Gamma \Vdash M' : T_1 \mid C_1 \leftarrow$ Generate($\Gamma \Vdash M'$)
- $\Gamma \Vdash N' : T_2 \mid C_2 \leftarrow$ Generate($\Gamma \Vdash N'$)
- Let us begin building a full judgement $\Gamma \Vdash M : X \mid C$ ▷ $C \leftarrow \emptyset$
- $C \leftarrow C_1 \wedge C_2 \wedge (T_1 == T_2 \rightarrow X)$ ▷ CTApp1
- $C \leftarrow C \vee (X \wedge == \mathsf{Ok})$ ▷ CTOk
- **if** there are constraints in C_{OkC1} **then**
  - $C \leftarrow C \vee C_{\mathsf{OkC1}}$
- **end if**
- $C \leftarrow C \vee ((T_1 \mid\mid \mathsf{Ok}^c \rightarrow X) \wedge C_1)$ ▷ CTApp2
- $C \leftarrow C \vee (T_2 == \mathsf{Ok}^c \wedge C_2)$ ▷ CTApp3
- **return** $\Gamma \Vdash M : X \mid C$ ▷ Return the full judgement

3.5 Satisfiability and Solving

Checking for satisfiability then returning the assignments of hundreds or even thousands of type constraints is a task we must approach with the right tools. We elected to use Microsoft Research's Z3 SMT Solver serves us well in terms of efficiency and simplicity. From our main JavaScript program constituting CompaSS we call a Python subprocess to use the z3Py API for Z3 [6]. There exists an API for JS however which we initially tried. However, it was far harder to use and lacked any significant documentation compared with the Python API. Z3Py was thoroughly documented and took advantage of language features.

3.5.1 Z3 and Algebraic Datatypes

In order to solve type constraints, we must first be able to represent them sufficiently well in an SMT solver like Z3. This is where we turn to the theory of algebraic datatypes, ordinarily used to represent common data structures like lists and trees. Naturally, they give us the ability to represent recursive data structures. The grammar for types is represented in CompaSS as a recursive data structure which is converted to a custom datatype *JSTy* that z3Py can understand. We define JSTy then discuss how it works below.

Definition 3.5.1 (JSTy Datatype) *The JSTy Datatype is a z3Py recursive datatype that allows us to express any concrete type we would otherwise be able to write in the grammar for JSS types.*

```
1    JSTy = Datatype('JSTy')
2    JSTy.declare('Num')
3    JSTy.declare('Ok')
4    JSTy.declare('To', ('lft', JSTy), ('rgt', JSTy))
5    JSTy.declare('Comp', ('comp', JSTy))
6    JSTy = JSTy.create()
```

Objects of type JSTy are trees where leaf nodes must either be "`Num`" or "`Ok`" and branches must have degree one or two. Those of degree one represent complementation of their child i.e. for some child type A if it belongs to a "`Comp`" branch node we syntactically get A^c, which as we see in the early examples in Chapter 1 is written as "`Comp(A)`". Similarly, branches with degree exactly two represent function types, whose two children "`lft`" and "`rgt`" are the antecedent and consequent respectively. For example, a node with children $\{A, B\}$ syntactically gives us $A \rightarrow B$ which CompaSS writes as "`A -> B`" in its output.

Z3 allows us to define constants, or nullary functions, which it is able to interpret for a solution. We use this mechanism to create and label one uninterpreted constant for every unique type variable that appears in the constraints. With this Z3 can interpret the constants to solve the constraints, and this interpretation or assignment is returned to the programmer when a solution is found. Note that interpreting constants corresponds to concrete type assignment to the type variable which said constant

represents. There are sometimes multiple possible type assignments, which is an inherent result of there originally being more than one rule applying to each grammar shape. Furthermore, since we are only able to assign concrete types, any such typing is almost always *not* a principal type (or most general type [9]). This is because, given that principal types for simple functions like id are often written with type variables, our constraint solver doesn't leave any type variables unassigned. For example, the function `const id = x => x;` has the principal type $A \to A$. This is the most general type we can give it. However, if CompaSS solves the constraints as part of a larger program which fails, such a function may receive the type $\mathsf{Num} \to \mathsf{Num}$, which we can see is not a principal type. Note though `id` does still inhabit this type and this is what we find generally: given some ill-typed program, the typings in Δ are always inhabited by their terms under the success semantics. This relies on the soundness and completeness of our constraint system, which is proven for the term subsystem in Section A.

3.5.2 Constraint Solving

When our subprocess `wrapper_z3.py` (hereafter "*the solver*") is called, it reads in a file containing `env` : Δ and `constrs` : C in their raw JS object form. The high-level structure of the solver is described in Algorithm 4. While solutions are generated by the delegate call to MakeSolns, we ought to discuss elements of the main routine to explore what the delegate has to work with.

Algorithm 4 Solver Overview

procedure SolverMain(Δ, C)
- $typeVarNames \leftarrow$ the unique type variables mentioned in C
- $topLevelNames \leftarrow \mathsf{dom}(\Delta)$
- $topLevelTypes \leftarrow \mathsf{range}(\Delta)$
- $maxSolns \leftarrow |topLevelNames|$
- Let $JSTy$ be defined as in 3.5.1
- Let $tyNameToConst$ be a map from type variable names $\to$ JSTy Consts
- $allConstrs \leftarrow$ constraints C with type variables and concrete types replaced with $JSTy$ data types
- $allIllTypings \leftarrow$ all type variables set equal to Ok^c
- $atLeastOneIllAssm \leftarrow \bigvee allIllTypings$ as in Definition 3.3.2
- $constrsPlusIlls \leftarrow allConstrs \wedge atLeastOneIllAssm$
- $[\Delta_1, \Delta_2, \ldots] \leftarrow$ MakeSolns($tyNameToConst$, $constrsPlusIlls$, $maxSolns$, $topLevelTypes$, $JSTy$)
- $assigns \leftarrow \{(x, \mathcal{A}) \mid \forall i \,.\, x \in \mathsf{dom}(\Delta_i) \wedge \mathcal{A} = \{A_i \mid x : A_i \in \Delta_i\}\}$
- $fails \leftarrow \{x \mid (x, \mathcal{F}) \in assigns \,.\, \mathsf{Ok}^c \in \mathcal{F}\}$
- **return** $\{$"$deltaTyAssigns$" : $assigns$, "$failsAt$" : $fails\}$

$maxSolns$ is the first non-trivial variable in the main routine, where we tell the solver when to stop iterating through solutions after we meet this number. This is because, as we see in Algorithm 5, the solver can produce a new solution just if it can prove one more term in Δ ill-typed. Thus when considering the maximum number of solutions we need, it becomes naturally bounded at the number of terms in Δ, at which point the solver has proven every term is ill-typed. JSTy we set according to Definition 3.5.1, giving us a *sort* (a Z3 datatype type) used to instantiate JSTy objects using z3Py. Having declared the concrete functions nullary (`Num`, `Ok`), unary (`Comp`) and binary (`To`) we are now free to create any concrete type. $tyNameToConst$ maps the names of the unique type variables that appear in the constraints to constants bearing the same name. These are all uninterpreted nullary functions in Z3 which will be interpreted by the solver. With a map from the type variable names to the actual JSTy type variables, we convert the constraints from their nested object format into a single logical quantifier-free formula. We recursively traverse the conjunctions and disjunctions in the data structure, instantiating these in Z3 as we go, until we meet type constraints. Then we instantiate them using JSTy to create arrows, complements and concrete types e.g. Ok^c, and $tyNameToConst$ to reference any type variables.

A vital part of the main routine is $allIllTypings$. It contains a constraint for every type variable taken from Δ requiring it be ill-typed. If these constraints were then simply just asserted in the solver, the solver would almost always fail to find a solution. This is because it is rare that *all* terms in Δ are ill-typed, which is why instead we assert that *any* are ill-typed. This is exactly equivalent to forced ill-typedness as introduced in Definition 3.3.2. We must therefore find at least one ill-typed top-level term else CompaSS cannot prove that it fails. This is what $atLeastOneIllAssm$ characterises. We then connect the forced ill-typed constraints conjunctively to the original constraints now in z3Py form and pass them on to our delegate which we see in Algorithm 5.

Investigating how we generate solutions, we first of all note that if all CompaSS aimed to do was simply check whether a program goes wrong we would not have to use so much logic nor a while loop in this method. One satisfiability check would determine whether a solution to the type constraints could be found. Ultimately the final behaviour CompaSS subsumes this, whilst also providing the programmer with further details. These include where in the program the failure arose, and the type which the terms involved had that caused the failure while finding as many failures as it can. These benefits are why we opt for a while loop instead of a single satisfiability check. To generate multiple solutions in Z3 we must require that something fundamentally changes between solver iterations, otherwise we may fetch the same solution multiple times. To achieve this we add to the constraints after each iteration requiring that (by If Statement A) at least one non-ill-typed top-level type must be ill-typed next and (by If Statement B) that which is ill-typed remains ill-typed. This simple pair of requirements in practice, once a true-positive or single ill-typing is found, let us find all failures we would expect in a program. We make this claim based on our test suite behaviour and this point is discussed further in Chapter 4.1.

Separately, early on in testing sometimes when finding these 'bonus' ill-typings with the further constraints we add at the end of each iteration, occasionally Z3 began mistakenly interpreting the branch arguments inside JSTy. Solutions with this behaviour not only contained assignments to type variables, but interpretations of `lft`, `rgt`, and `comp`. One such real assignment made by our solver before we fixed this issue was the following:

$$\texttt{rgt}(A) = \begin{cases} \mathsf{Ok}^c & \text{if } A = \mathsf{Ok}^c \to \mathsf{Ok}^c \\ \mathsf{Ok}^c & \text{if } A = \mathsf{Num}^c \to \mathsf{Ok}^c \\ \mathsf{Num}^c & \text{if } A = \mathsf{Ok} \to \mathsf{Num}^c \\ \mathsf{Num} & \text{if } A = \mathsf{Ok}^c \to \mathsf{Num} \\ \mathsf{Num} & \text{if } A = \mathsf{Ok} \to \mathsf{Num} \\ \mathsf{Num} & \text{if } A = \mathsf{Num} \to \mathsf{Num} \\ \mathsf{Ok} & \text{otherwise} \end{cases}$$

We hope you can see this is nonsense given that, because the solver works with syntactic equality (two types are equal when their strings are equal), we end up with a solution that relies on *implicit syntactic transformations* when a datatype is used. Such an interpretation of `rgt` would imply that the type $\mathsf{Num} \to \mathsf{Ok}^c \to \mathsf{Num}$ is actually $\mathsf{Num} \to \mathsf{Num}$ giving us $\mathsf{Num} \to \mathsf{Ok}^c \to \mathsf{Num} = \mathsf{Num} \to \mathsf{Num}$ which is certainly false by syntactic equivalence and represents implicit syntactic transformation (we don't want this). We hereafter call this *assignment to the wrong arity*; constants (which are arity zero) are the only things we ought to assign to. To safeguard against assignment to the wrong arity, we use the *illegalAssign* flag that stops the loop and discards the solution if such a syntactic transformation is required for the solution to be satisfying. When searching stopped in this manner, CompaSS often missed several ill-typings we could find by inspection of test programs. However, one may argue that we were not giving up much value even if we had stopped here from the perspective of a debugging tool. If we can show a given program is ill-typed in one place out of several for one run of CompaSS, we imagine the programmer fixes the issue then runs CompaSS on their program again. Given that finding multiple solutions at once does not increase CompaSS' ability to find any given ill-typing, we will still find every ill-typing if the programmer fixes them one at a time. We decided to continue trying to find a way to produce as many ill-typings as we could whilst assuring no false positives. If we did return some false-positive ill-typings this is arguably morally against our guarantee, even if our guarantee is strictly *only* being able to tell if a program is ill-typed then it must fail. Thereby our guarantee has nothing to do with where or in how many places a program is ill-typed, but as an extension we seek to prove CompaSS' utility nonetheless.

Cancelling the solver on assignment to the wrong arity, we forewent finding all ill-typings at once that one could find with multiple run-fix-run iterations of CompaSS and the programmer. We also avoided the unsoundness of implicit syntactic transformations this way. On further research and testing, we found that adjusting the *relevancy propagation* of the solver allowed us to avoid such assignments. Relevancy propagation in Z3 aims to minimise the number of boolean atoms to keep track of by determining which are relevant to satisfiability [11], in the interest of efficiency. The sorts of constraint that this is important for as noted by Bjørner and Moura [11] are integer and linear arithmetic, bit vectors and quantifiers, since these are the most expensive theorems to solve over. But notice that we do not use these theorems in our constraints, which suggests we might not need relevancy propagation to achieve acceptable solver performance with our simple quantifier-free constraints. This is just what we find when testing CompaSS having lowered relevancy propagation for the solver, as seen in Algorithm 5. *solver*.SetRelevancy(1) *lowers* the relevancy propagation from the default value of two. This, from what we read in Z3's official

parameter documentation [4], reduces the tracking of relevancy thereby requiring more boolean atoms to be evaluated. It is not easy to find further information on why this may be impacting our solver, we conjecture it makes sure all atoms are instantiated and tracked thereby more easily allowing a change in assignments as required at the end of every iteration. Running our test suite and varying relevancy, we see the result of adjusting this parameter from two to one gives us what we want in all cases that we test, instead of stopping before we report all failures. One such example we give below.

```
1 const id = x => x;
2 const mightFail = x => {
3     return x <= 0 ? 1 : 2(3);
4 }
5 const guardFail = x => {
6     return (y => y) + x ? 1 : 2;
7 }
8 const willFail = x => {
9     return x <= 0 ? (0 <= 0 ? 0(0) : 0(0)) : id + id;
10 }
11 const mgw = mightFail(guardFail(willFail));
12 const gmw = guardFail(mightFail(willFail));
13 const wgm = willFail(guardFail(mightFail));
14 const mNum = mightFail(0-1);
15 const gNum = guardFail(0);
16 const wNum = willFail(0);
```

```
...and fails at:  mgw,gmw,wgm,gNum,wNum
```

(a) Relevancy set to one we find all true ill-typings

```
...and fails at:  mgw,gmw
```

(b) Relevancy set to two we find some ill-typings

Figure 3.8: Effect of relevancy on test validity: left output passes the test, the right output does not.

When we say "all true ill-typings" in Figure 3.8, specifically we mean "all true ill-typings *we can expect*". For this program specifically, we have no rule in the system for reasoning about different branch types for conditionals, thus we do not infer anything about the type of `mNum` even in the case of CompaSS performing optimally (relevancy set to one). It cannot prove incorrectness because on evaluation `mightFail(0-1)` does not actually fail and instead evaluates to a Num. This example represents what happens in all tests that fail with relevancy set to two: we inspect where CompaSS tells us the program is ill-typed yet these tests do not pass because it has not found all ill-typings we recognise there are by inspection. Investigating the assignments produced by the solver this is always due to $illegalAssign \leftarrow \top$. Conversely, with relevancy set to one, $illegalAssign$ is never true just because for all tests the constraints are exhausted in finding all true ill-typings we can expect. Note, all solutions are still sound with respect to there being absolutely no implicit syntactic transformations arising from assignment to the wrong arity.

Algorithm 5 Constraint Solving

procedure MAKESOLNS($tyNameToConst$, $constrsPlusIlls$, $maxSolns$, $topLevelTypes$, $JSTy$)
  $solns \leftarrow \{\}$
  Let *solver* be a new solver object
  solver.SETRELEVANCY(1)
  $illegalAssign \leftarrow \bot$
  while $\neg illegalAssign \wedge$ can satisfy *solver* $\wedge$ not exceeded max iterations **do**
    $model \leftarrow solver$.MODEL()
    $modelNegateAny \leftarrow []$
    Let *soln* be an empty map from variable name to concrete type in the solution
    Let $yetToIllType$ keep track of what top-level variables we still need to ill-type
    for $(X : A)_n$ in *model* **do** ▷ n is the arity of the JSTy we are interpreting
      if $n == 0$ **then**
        $soln[X] \leftarrow A$
        if $\neg(model[X] == \mathsf{Ok}^c) \wedge (X$ **in** $yetToIllType)$ **then** ▷ If statement A
          $modelNegateAny \leftarrow modelNegateAny \cup tyNameToConst[X] == \mathsf{Ok}^c$
        end if
        if $model[assign] == okC \wedge (X$ **in** $yetToIllType)$ **then** ▷ If statement B
          solver must satisfy $tyNameToConst[X] == \mathsf{Ok}^c$ next iteration
        end if
        if A **in** $yetToIllType$ **then**
          $yetToIllType \leftarrow yetToIllType \backslash x$
        end if
      else
        $illegalAssign \leftarrow \top$
        return
      end if
    end for
    $solns \leftarrow solns \cup sol$
    $modelNegateAny$ satisfies any $\{tyNameToConst[x] == \mathsf{Ok}^c \mid x \in yetToIllType\}$ next iteration
  end while
  return *solns*

Chapter 4

Evaluation

CompaSS' aims are intrinsic to bug-finding, and offer a tempting perspective that is not often made available to a programmer; ill-typed programs are guaranteed to fail. On the other hand, a programmer using a system that produces false positives for a dynamic language in different situations, e.g. branches with different return types, often needs to use their better judgement to work their way around the issue. That is not the case with CompaSS which may show that the philosophy it implements is a better fit for a dynamically typed language, as is suggested by Lindahl and Sagonas in the case of Erlang, another dynamically typed language which receives a success type system in their paper [14]. Here we explore the test suite with which we evaluated the behaviour of CompaSS and share how the implementation performs highlighting its strengths and limitations.

4.1 Strengths

We give parts of the raw output of CompaSS in all cases and use it to demonstrate where its strengths and limitations arise.

4.1.1 Undesired Behaviour and Subtle Errors

We make sure to note that our definition of ill-typed includes undesired behaviour. This can actually help us find (arguably) bad programming practices alongside crashes and divergence. One such example is when CompaSS tells us this program to compute the dot product of two positive vectors is ill-typed, despite it appearing to work when it is run:

```
1  const nil = 0;
2  const list = s => t => p => p(s)(t);
3  const head = s => t => s;
4  const tail = s => t => t;
5  const vecA = list(0)(list(1)(list(2)(list(3)(list(4)(nil)))));
6  const vecB = list(1)(list(6)(list(7)(list(13)(list(20)(nil)))));
7  //multiline conditionals are valid JSS; no significant whitespace
8  const mulPos = n => m => n - 1 <= 0
9      ? n <= 0
10         ? 0
11         : m
12     : m + mulPos(n - 1)(m);
13 const dotPos = a => b => {
14     return a <= 0
15         ? 0
16         : mulPos(a(head))(b(head)) + dotPos(a(tail))(b(tail));
17 }
18 dotPos(vecA)(vecB);
```

```
Solution
 nil:  Ok
list:  (Num -> (Ok -> ((Ok -> (Ok -> Num)) -> Num)))
head:  (Ok -> Ok)
tail:  (Num -> (Num -> Ok))
vecA:  ((Ok -> (Ok -> Num)) -> Num)
vecB:  Ok
mulPos:  (Comp(Num) -> (Num -> Ok))
dotPos:  (((Ok -> (Ok -> Num)) -> Num) -> (Ok -> Comp(Ok)))
eval#0:  Comp(Ok)
      Ill-typed and fails at:  eval#0
```

Yet when the program is run it produces the correct value for the dot product of $\{0,1,2,3,4\}\cdot\{1,6,7,13,20\}$: 139. Without our success semantic extension as defined rigorously in 3.1.4, we would be violating our guarantee. But we see that what actually happened is that we have just gotten lucky it works while relying on undesired behaviour, and exactly where this happens is hinted at by CompaSS' output. Reading off the type of `dotPos` which CompaSS used to prove this program ill-typed then removing superfluous brackets, we have $dotPos : ((\mathsf{Ok} \to \mathsf{Ok} \to \mathsf{Num}) \to \mathsf{Num}) \to \mathsf{Ok} \to \mathsf{Ok}^c$. That is to say, when we give $dotPos$ a function which takes a function of type $\mathsf{Ok} \to \mathsf{Ok} \to \mathsf{Num}$ and returns a Num (`list` inhabits this type), we can give it anything else at all (Ok) and it will become ill-typed. If we think about our crashing extension we include that `a <= 0 ? branchA : branchB` is undesired behaviour just if $A \mid\mid \mathsf{Num}$, given `a` : A. This is what CompaSS detects here; even though the condition inside `dotPos` makes sense when encountering a `nil` at the end of a list, it does not make sense in any other case. In all other cases where the conditional is interpreted, the guard gets `a` : A where A is a function, therefore comparison to zero does not make sense and we have gotten lucky here that JS inherently determines that this function ought to be larger than zero, and chooses `branchB` until we get to the end of a vector. This is why the program appears to work, but we have been made aware that it uses undesired behaviour that we arguably should not rely on. We can distinguish this from divergence or going wrong in this example when we run the program; it does not diverge or crash at runtime leaving just one option in our extended success semantics given `eval#0` : Ok^c.

4.1.2 Can Find All Ill-Typings

JSS programs can be ill-typed in multiple places, which CompaSS is then able to prove as in the following examples.

```
1  const myWrong = 0(0);
2  const putAWrongIn = x => {
3      const y = z => z;
4      y(x);
5      return 0;
6  }
7  const aRightIn = putAWrongIn(0);
8  const aWrongIn = putAWrongIn(myWrong);
```

```
Ill-typed and fails at:  myWrong,aWrongIn
```

CompaSS finds both solutions which make the program ill-typed, the first ill-typing `myWrong` and the second ill-typing `aWrongIn`. This ability allows the programmer to work on fixing multiple related bugs, as well as show potential links between ill-typed terms. For example here, when we read the type assignments

in the program type we are given a clear idea of what causes the second failure.

```
Solution
    myWrong:  Comp(Ok)
putAWrongIn:  (Comp(Ok) -> (Num -> Num))
   aRightIn:  Ok
   aWrongIn:  Ok
Solution
    myWrong:  Comp(Ok)
putAWrongIn:  (Ok -> Ok)
   aRightIn:  Ok
   aWrongIn:  Comp(Ok)
...
```

In the first solution `myWrong` fails by itself since nothing else in the program is given the type of a function that can produce a `Comp(Ok)`, nor have any others failed. In the second solution, we see that still none of the functions in the program can produce a `Comp(Ok)` which suggests the failure does not depend on them. When the programmer inspects why `aWrongIn` fails after we type `myWrong` as `Comp(Ok)`, we see that it's used as the argument to `putAWrongIn` and therefore the function application fails.

In developing CompaSS, we wrote a suite of over one hundred tests to confirm that the type-checker was behaving as expected. Among those tests, we inspected how many failures it would find and where they were. All such tests pass and so it appears CompaSS finds all failures that we expect it's type system should.

4.1.3 Typings Give Detail on How Ill-Typed Terms Arise

We examine specifically how we can use the types of variables in the program to help diagnose the underlying faults. In the following example suppose we wrote a program that defines integer division, and perhaps we came back to it many months later and have forgotten the correct way to use it. We want the quotient part of the division $\frac{120}{10}$ instead of the remainder and pass in `getQuot` to the function too early.

```
1 const pair = m => n => p => p(m)(n);
2 const getQuot = s => t => s;
3 const getRmdr = s => t => t;
4 const intDivide = n => d => q => {
5     const r = n - d;
6     return r + 1 <= 0 ? pair(q)(n) : intDivide(r)(d)(q + 1);
7 }
8 const quot = intDivide(120)(getQuot)(10);
```

```
Solution
    pair:  (Ok -> Ok)
  getQuot:  (Num -> (Num -> Ok))
  getRmdr:  (Comp(Ok) -> Num)
intDivide:  (Ok -> ((Num -> (Num -> Ok)) -> (Num -> Comp(Ok))))
    quot:  Comp(Ok)
        Ill-typed and fails at:  quot
```

When reading the types in the program, our eyes ought to be drawn to all the `Comp(Ok)`s we see. In this case, we see one in `getRmdr` although this is in the antecedent, which constitutes unrestricted sufficiency (vacuous evaluation to a Num as discussed with contrapositive in Section 3.2.3). So this is not the failure we are looking for, instead we should notice the consequent of the function type for `intDivide` is `Comp(Ok)`. We can interpret the type as if we give anything (Ok), then some complicated function type, then a Num

it will fail. Therefore the Ok, the function, and/or the Num we give to `intDivide` cause it to fail. If it were just the Ok, this would imply the function body is wrong as intuitively it would mean "anything you give me, I will fail". For the other two arguments, we can assume that one or both must be wrong if the function body is not wrong. So having understood this we first check the body is correct, suppose we are convinced it is; perhaps this is a library function in our context so we might have strong assurance for this. We rule out Ok (the first argument) and check how the remaining two arguments are used. Immediately we see that `d` is used in a subtraction, except we have passed `getQuot` in for `d`, which is not a number at all. Thus we realise one mistake and can repair the program by making `d`, the second argument, the denominator.

Having scanned the body we also see `intDivide` returns a pair, so deem that `getQuot` must be given to the result of `intDivide`. We make this change and run the program through CompaSS another time.

```
1 const pair = m => n => p => p(m)(n);
2 const getQuot = s => t => s;
3 const getRmdr = s => t => t;
4 const intDivide = n => d => q => {
5     const r = n - d;
6     return r + 1 <= 0 ? pair(q)(n) : intDivide(r)(d)(q + 1);
7 }
8 const quot = intDivide(120)(10)(0)(getQuot);
```

```
pair:  Untypable
getQuot:  Untypable
getRmdr:  Untypable
intDivide:  Untypable
quot:  Untypable
Inconclusive
```

We can see that CompaSS could not prove our program incorrect. We can now be *more* confident that it will work, if only because before we were certain it would not. We can never be certain that it will work just from what CompaSS tells us, but we now have no errors that its type system can show under forced ill-typedness.

4.1.4 Divergence can be Ill-Typed

One property of the success semantics tells us when $M : A$ then $M \in \mathcal{T}_{\perp\not{\iota}?}[\![A]\!]$. Note of course this means $M \in \mathcal{T}_{\perp}[\![A]\!]$ as well. Therefore if we can show a program is ill-typed it may just crash, exhibit undesired behaviour, or diverge. We show an example of the latter with a simple program which diverges and CompaSS determines is ill-typed.

```
1 const f = x => f(f);
2 f(0);
```

```
Solution
f:  (Ok -> Comp(Ok))
eval#0:  Comp(Ok)
Ill-typed and fails at:  eval#0
```

When we run this program with NodeJS we get the following error: "`RangeError: Maximum call stack size exceeded`"; evidence to confirm the program has diverged.

A slightly less trivial example is the following program where we have tried to define a recursive multiply and perhaps forgotten a conditional.

```
1 const mul = n => m => {
2     const nNext = n - 1;
3     return mul(nNext)(m); //forgotten conditional?
4 }
5 const twoInfinityAndBeyond = mul(2)(3);
```

```
                        Solution
                  mul:  (Ok -> (Num -> Comp(Ok)))
    twoInfinityAndBeyond:  Comp(Ok)
                        Ill-typed and fails at:  twoInfinityAndBeyond
```

What is interesting in these cases is that, unlike the examples we show ill-typedness for which rely on misuse of a function, we show even with valid arguments the function is still ill-typed when run. Therefore what we are looking at is a distinct feature of the type system.

4.1.5 JSS is Useful

JSS though small has similar structures to that of an assembly language, where for example we can encode a conditional branch as a JSS conditional with two function calls as branches. This can be used to define useful functions for example greatest common denominator as in Figure 4.1, integer division in Section 4.1.3, and multiplication as in Figure 4.2. The functional nature of JSS lends itself to recursion, to which data structures are no exception as we see in our tree data structure in Figure 4.3.

```
1 const equals = a => b => {
2     return a - b <= 0 ? b - a <= 0 ? 0 : 1 : 1;
3 }

5 const gcd = a => b => { //Euclid's Greatest Common Denominator
6     const diff = a - b;
7     const largest = diff <= 0 ? b : a;
8     const smallest = diff <= 0 ? a : b;
9     const posDiff = largest - smallest;
10    return equals(a)(b) <= 0 ? a : gcd(posDiff)(smallest);
11 }

13 gcd(42)(24); //6
14 gcd(63)(14); //7
15 gcd(4224)(2244); //132
```

Figure 4.1: Greatest Common Denominator in JSS.

```
const treeNode = lft => val => rgt => lftValRgt =>
    lftValRgt(lft)(val)(rgt);
const pft120 = treeNode //prime factor tree of 120
    (treeNode
                (3)
            (12)
    (treeNode       (2)
                (4)
                    (2)))
        (120)
    (treeNode   (5)
            (10)
                (2));
const lft = t => u => v => v;
const rgt = t => u => v => t;
const factorLR = pft120(lft)(rgt); //5
const factorRLR = pft120(rgt)(lft)(rgt); //2
```

Figure 4.3: Recursive Tree Data Structure in JSS.

```
const mulNat = n => m => {
    const nn = n - 1;
    return nn <= 0 ? n <= 0 ? 0 : m : m + mulNat(nn)(m);
}
mulNat(0)(1); //0
mulNat(2)(5); //10
mulNat(239)(458); //109462
```

Figure 4.2: Multiplication on Naturals in JSS.

4.2 Limitations

In this section, we discuss some disadvantages of CompaSS and reason about why they are the case, including some of the weaknesses in expressibility and drawbacks of the language subset here.

4.2.1 Divergence can by Untypable

We do not have to write very complicated diverging functions before CompaSS can no longer prove they are incorrect. The following example starts with the second diverging program in Section 4.1.4 except the return has an addition this time.

```
const mul = n => m => {
    const nn = n - 1;
    return m + mul(nn)(m);
}
const twoInfinityAndBeyond = mul(2)(3);
```

```
                                    mul:  Untypable
            twoInfinityAndBeyond:  Untypable
                                    Inconclusive
```

CompaSS no longer knows what will happen, but in fact this makes no difference to the program and it still diverges. This also highlights that we should not treat `Inconclusive` or `Untypable` as anything other than "I don't know", there is no guarantee either way. The reason this happens here is that in order to prove the return type of `mul` is Ok^c, we *have* to show `mul(nn)(m)` is disjoint from a number in accordance with (NumOp2). Given that the program was not proven ill-typed, we must not be able to express this.

4.2.2 No Utility In Untypability

As stated previously there are no guarantees that come with a program being untypable or inconclusive. CompaSS can never prove anything correct, and this is arguably also a significant portion of the value of traditional debuggers, even with their false positives. Of course, traditional debuggers have false positives *and* false negatives, the former of which we at least do not have. That is to say, traditional debuggers are not always correct in saying programs are right e.g. logic errors may persist or programs may diverge.

CompaSS operating as a success type system means that even a term with the most specific concrete type can still crash, diverge or use undesired behaviour by definition. This is the crux of what makes term typings in CompaSS significantly less powerful than without the success semantics i.e. when $x : A$ is equal to $x \in \mathcal{T}_\perp[\![A]\!]$ as in most traditional systems.

4.2.3 Number of Constraints and Ill-Typing Speed

Each time a constraint generation tree uses another constraint rule, it has at least two constraints in the case of (CTNum), and three for (CTVar) if x in our assumptions is not a top-level variable. Each time (CTAbsInf) is used, it has at least six constraints, specifically three plus the number of constraints in C_1 which is at least three due to the additional variable. Each time (CTApp) is used, it has at least 22 constraints, which is assuming that C_1 and C_2 have at least three constraints. We can imagine with nested applications of the (CTApp) rule, the size of the constraints in the root rule of the constraint derivation tree will have exploded. For example, suppose CompaSS' is given `(x => x)(x => x)(x => x)` which exclusively uses the rules we have discussed here. After generation and before forced ill-typedness it has generated 78 constraints, which is a lot for such a small program. One may naturally ask: how many would a larger program generate? Our recursive tree data structure in Figure 4.3 produces *over 3500* constraints.

A system of over 3500 type equations sounds like a lot, but it matters more how fast CompaSS is able to solve these. It turns out the combined runtime of generation and solving (the latter taking the longest, generation is almost instant) takes approximately 681.3ms averaged over ten runs, on presumably one thread of an AMD Ryzen-7 4700U. It would make sense for those rules that reference the constraints in their premise to increase the number of constraints exponentially, and some testing suggests that this is the case. Modifying our tree example (fig. 4.3) so the tree is defined and accessed in-line, we get many more applications chained together for a much deeper constraint generation tree.

```
1...
2 const pft120 = treeNode
3     (treeNode
4                 (3)
5             (12)
6     (treeNode       (2)
7                 (4)
8                     (2)))
9         (120)
10    (treeNode   (5)
11            (10)
12                (2))(rgt)(lft)(rgt);
```

We now see a runtime of 2971.3ms averaged over ten runs, with *more than 23000* constraints generated. Note we are exploring the worst-case scenario here as we are dealing with long chains of currying i.e. many sequential applications.

Many other rules multiply the number of constraints by a similar amount to (CTApp) which suggests the system would not scale well with longer programs in general. Runtime was never a goal CompaSS intended to meet, so we do not study it in detail here, but it is interesting to note. A system with different goals including sensible performance scaling might have to limit the number of constraints or find a faster

way to solve them than using z3Py. Specifically in the case of (CTApp) modifying the grammar and type system to allow for n-ary functions instead of just unary functions could perhaps be used to exercise greater control over the number of constraints made in situations akin to the tree data structure program.

4.2.4 JS is More Useful

JSS is considerably weaker than JS as a programming language, as we cannot use many things that JS has to offer. Chief among those are objects, non-arrow functions, `class` syntax, promises, multiple files, and many simple types like built-in arrays and strings. While we can still write interesting programs in our subset from a typing perspective, it leaves much to be desired and one may argue it goes against the notion that this type system was not supposed to impose a new programming style on the programmer as static type systems introduced to dynamic languages do [14]. This would be the case should CompaSS have been able to handle more of JS, but we argue that the limitation comes from the grammar and is not intrinsic to the type system. The grammar was kept small to manage the scope of this project, and functional-style type systems have been applied to JS e.g. objects in a dissimilar subset of the language by Anderson et al. successfully as discussed in the background [8].

However, it's clear JSS is disparate from JS even if JSS is still JS; the way that one must write JSS is fundamentally limiting.

Chapter 5

Conclusion

5.1 Summary

We begin our work in Chapter 3, where we first formally introduce JSS with its rationale. We extend JSS with composition to permit JSS programs to contain sequences of terms and for functions to have block-body definitions, which allows us to write much more interesting-looking programs that use more JS syntax. We define the types for the two-sided type system for JSS as well as types for programs. Following that we formalise undesired behaviour to later highlight where it affects our system of rules and why it then makes sense to reuse some PCF-like reasoning in the rule given Ramsay and Walpole in their one-sided system [16]. To that end, we define CompaSS' one-sided system and show how the rules for terms can be derived from a two-sided system for JSS that's nearly identical to that of Ramsay and Walpole's. We also give detailed explanations for the rules that handle our composition extension to JSS: block and program rules. Thereby we discuss similarities to the (Let)-style reasoning from Ramsay and Walpole's rules including (Let1) and (LetR) from their one-sided system. We produce our rules from a system similar to and referencing Ramsay and Walpole's rules to argue that we preserve their theorem of One-Sided Syntactic Soundness where we are guaranteed refutation soundness. Therein we guarantee zero false positives with respect to finding bugs in either programs that don't evaluate or rely on undesired behaviour.

Before introducing our constraint type system, we explain constraint rules in the context of pure λ-Calculus as does Chiang in [9] as well as define the constraint type variables that CompaSS uses. A small example constraint derivation is given to elucidate this process when done by hand. With this background, we can give clearer reasoning for how we arrived at the constraint rules in Figure 3.5 and 3.6. To clarify how these rules are used by CompaSS to generate constraints, we give another example constraint derivation for a simple two-line program and define forced ill-typedness to demonstrate its effect on what CompaSS can tell us about this small program.

We then uncover how CompaSS' implements constraint generation but first, we begin with a short explanation of the JSS AST and grammar specification language we use to check that a program is valid JSS. Following this three pseudocode algorithms are given, one for the program rules, one handling the remaining rules (block, term) that apply recursively, and the implementation of one of the constraint rules (CTApp). Ending our chapter on implementation are important details, findings and methods we used in the algorithms given for constraint solving using the SMT Solver Z3 [10], as well as challenges we had to overcome in the preventing interpretation of Z3 datatype arguments which we proved by example could introduce syntactic unsoundness.

Finally in Chapter 4 we present some major benefits and limitations of CompaSS as an implementation of a success type system for JavaScript. The benefits we highlight show it can prove subtle failures, find multiple failures at once, give further information on the origin of failures, ill-type divergence, and that one can write useful programs in JSS. On the other hand, we discuss that divergence can also be untypable, the lack of utility of untypability, and the supposed exponential runtime of CompaSS for sequential function applications. We also point out that JSS is missing some key parts of JS.

Even so, we have absolutely achieved our goals. We've shown that our implementation of a success type system for a subset of JavaScript is possible and *useful*. We argue it maintains the theorem of one-sided syntactic soundness as desired, given our extended success semantics, and it can show nuanced issues with ill-typed programs. We extended our system from just terms to composition, which let us reason about much more interesting multi-line programs and functions. We met our target and extension of showing

whether a program was incorrect by going on to show where such programs were ill-typed. This even gave the types which cause this behaviour, then allowing the programmer to intuit where the error originates as described in Chapter 4. This culminates in a novel, competent and correct implementation of an adept system for bug-catching with zero false-positive.

5.2 Future Work

This section identifies possible fruitful avenues of research taking CompaSS further with respect to the size of its language, efficiency of the system, and general capabilities.

5.2.1 True Positives Theorem

In this work we only ever *argue* that we maintain the true positives theorem by virtue of similarity to Ramsay and Walpole's one-sided system for which they prove this theorem. This suggests a valuable extension to this work; a proof that the theorem is maintained in CompaSS' one-sided system. This would immediately give us that the constraint system also maintains this theorem since we have proof of soundness and completeness of CompaSS' one-sided system to its constraint system. It would perhaps be easier to produce a two-sided system for CompaSS' block and program rules first, then prove soundness and completeness there instead. Having this as proof instead of just an argument is naturally beneficial for the strength of this work, making it a good candidate for the first extension one might make to this system.

5.2.2 Removal of Compo2 and CompoAbs2 from Constraints

(Compo2) and (CompoAbs2) were originally added to mirror Ramsay and Walpoles one-sided rule (Let3) which is also used to ignore the new variables. Since CompaSS will try both (Compo1) and (Compo2) at once in the constraint rules (same for the (CompoAbs) case) we implicitly require that x is not defined inside $\mathcal{B}$ regardless of whether or not we actually use it, because CompaSS will exit early if a variable is re-defined and note this is actually a syntax error in JS, too. For a concrete example of where we could choose to use (Compo2) with respect to the the one-sided system is given below.

```
1 const newBlock = x => {
2     const unused = 0;
3     return x;
4 }
```

The block body is `const unused = 0; return x;` and we see even in the one-sided system it does not affect the type of the block whether `unused` appears in the assumptions or not, i.e. we get the same result with (Compo1). It is currently not clear whether the constraints that (Compo2) or (CompoAbs2) contribute are ever useful for the ill-typing of JSS programs. We suggest it would not affect the completeness of the constraint system to remove the constraints that (Compo2) and (CompoAbs2) introduce in (CTCompo) and (CTCompoAbs) respectively. Therefore we could reduce the number of constraints produced when deriving blocks, aiding performance for larger programs. This is especially true because each time we use (CTCompo) or (CTCompoAbs) we generate constraints for three different premises, where the final premise is exclusively for (CTCompo/Abs2). Therefore in getting rid of these rules from the constraint system, we would be able to remove an entire third premise from both (CTCompo) and (CTCompoAbs). The second premise already subsumes the third for both these rules in all ways except that $\mathcal{B}$, the rest of the block, is typed with x or y in the assumptions. This affects only whether we can successfully apply (CTVar) if we do reference the variable, and what constraints are added by $\mathsf{Struct}(\Gamma, X)$ at each step.

Viewing the constraint rules as the sequence of ordinary rules we could have used to get there, we can consider the two cases we arrive at when typing a block. Suppose we do reference x or y in the rest of the block, in which case we would need to apply (Var) meaning (Compo/Abs1) would've had to apply and not (Compo/Abs2).

Suppose we don't reference x or y and the block is typed, in which case we did not need to apply (Var) so (Compo/Abs1) would still apply but also (Compo/Abs2).

These cases imply that (Compo1) and (CompoAbs1) subsume their counterparts. This change would also bring the block rules more in line with the program rules, but further research and proof of completeness with respect to the one-sided system would be required to confirm that we can remove the constraints and still maintain equivalence.

5.2.3 Optimised App

All functions in JSS are unary and this requires currying for functions to take multiple arguments. Having to write multiple applications to fully apply a function means overuse of the (CTApp) rule, one of the more expensive rules in the system. As mentioned in the previous section, modifying the grammar and type system to allow for n-ary functions instead of just unary functions could perhaps be used to reduce the number of constraints generated in situations where we otherwise would need long chains of currying. This would potentially avoid an exponential increase in the number of constraints, instead just producing linearly more constraints as the number of arguments increases. Such an outcome would represent a significant optimisation to the current generation algorithm.

5.2.4 Extending JSS

One may seek to introduce new terms to the grammar for JSS to capture more of the enclosing language. More research, rules and extension of the implementation of CompaSS would be required, as well as a refutation soundness proof for any rules that were added to keep the true positives theorem from Ramsay and Walpole's work [16].

Bibliography

[1] "Acorn AST Walker". *GitHub*, Accessed: 05/02/24, [Online] Available:. URL: https://github.com/acornjs/acorn/tree/master/acorn-walk/.

[2] "Erlang". *Dialyzer*, Accessed: 07/05/24, [Online] Available:. URL: https://www.erlang.org/doc/apps/dialyzer/dialyzer_chapter.

[3] "JavaScript Language Overview". *Mozilla Developer Network*, Accessed: 06/05/24, [Online] Available:. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_overview.

[4] "Online Z3 Guide | Parameters". *Z3 Guide*, Accessed: 06/05/24. [Online] Available:. URL: https://microsoft.github.io/z3guide/programming/Parameters/.

[5] "TypeScript"'. *TypeScript homepage*, Accessed: 28/04/24, [Online] Available:. URL: https://www.typescriptlang.org/.

[6] "Z3Py Guide". *Z3Py Guide*, Accessed: 07/05/24, [Online] Available:. URL: https://ericpony.github.io/z3py-tutorial/guide-examples.htm.

[7] Fernando Alves, Delano Oliveira, Fernanda Madeiral, and Fernando Castor. On the bug-proneness of structures inspired by functional programming in javascript projects, 2022. arXiv:2206.08849.

[8] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for JavaScript. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, pages 428–452, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[9] D. Chiang. "Type Reconstruction". *CSE 40431*, Accessed: 07/05/24, [Online] Available:. URL: https://www3.nd.edu/~dchiang/teaching/pl/2019/typerec.html.

[10] L. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[11] Leonardo de Moura and Nikolaj Bjørner. Relevancy propagation. *Technical Report MSR-TR-2007-140, Microsoft Research, Tech. Rep.*, 2007.

[12] Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in javascript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 758–769, 2017. doi:10.1109/ICSE.2017.75.

[13] Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. TeJaS: retrofitting type systems for JavaScript. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, page 1–16, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2508168.2508170.

[14] T. Lindahl and K. Sagonas. "Practical Type Inference Based on Success Typings". *PPDP'06*, vol. 1:page 167–177, Jul 2006. doi:10.1145/1140335.1140356.

[15] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[16] Steven Ramsay and Charlie Walpole. Ill-typed programs don't evaluate, 2023. arXiv:2307.06928.

[17] Konstantinos Sagonas. Using static analysis to detect type errors and concurrency defects in erlang programs. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming*, pages 13–18, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

Appendix A

Proof of Soundness and Completeness

We present two proofs that culminate in equivalence (soundness plus completeness) of the constraint rules 3.5 to the ordinary one-sided rules 3.3. We prove soundness and completeness separately.

A.1 Soundness of Constraint Rules

Theorem A.1.1 (Constraint Soundness) *If* $\Gamma \Vdash M : X \mid C$ *and* $\exists\theta\ .\ \theta \models C$ *then* $\theta\Gamma \vdash M : \theta X$

The following proof is by induction on the rules of Figure

CASE(CTVar). Assume (CTVar) holds giving us $\Gamma, x : Y \Vdash x : X \mid C$ for some fresh type variable X, where $C = (X == Y) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$. Assume also $\theta \models C$. To show $\theta(\Gamma, x : Y) \vdash x : \theta X$ (we call θJ) we case-split the constraints on their disjunctive clauses:

- If $(X == Y)$, and $\theta J = \theta\Gamma, x : \theta Y \vdash x : \theta X$ we get $\theta Y = \theta X$. This lets us conclude with (Var).
- If $(X == \mathsf{Ok})$, then $\theta J = \theta\Gamma, x : Y \vdash x : \mathsf{Ok}$ which we may conclude with (Ok).
- If $(\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$ then $\theta J = \theta\Gamma, x : \mathsf{Ok}^c \vdash x : X$ which we may conclude with (OkC1).

CASE(CTNum). Assume (CTNum) holds giving us $\Gamma \Vdash \underline{n} : \mathsf{Num} \mid C$ for some fresh type variable X, where $C = (X == \mathsf{Num}) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$. Assume also $\theta \models C$. To show $\theta\Gamma \vdash \underline{n} : \mathsf{Num}$ (we call θJ) we case-split the constraints on their disjunctive clauses:

- If $((X == \mathsf{Num}))$ then $\theta J = \theta\Gamma \vdash \underline{n} : \mathsf{Num}$ which we may conclude with (Num).
- If $(X == \mathsf{Ok})$ then $\theta J = \theta\Gamma \vdash \underline{n} : \mathsf{Ok}$ which we may conclude with (Ok).
- If $\theta \models (\bigvee\{Y_i == \mathsf{Ok}^c | Y_i \in 1\})$ then $\exists i$ s.t. $\theta J = \{x_i : \mathsf{Ok}^c | x_i \in \mathsf{dom}(\Gamma)\} \vdash \underline{n} : \theta X$ which we can conclude with (OkC1) since at least one Ok^c variable typing exists.

CASE(CTAbsInf). Assume (CTAbsInf) holds giving us $\Gamma \Vdash x \Rightarrow M : X \mid C$ for fresh type variables X and Y, where $C = (X == Y \to T_1 \wedge C_1) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$. The premise for this rule is of the form $\Gamma, x : Y \Vdash M : T_1 \mid C_1$ which gives us the remaining variables in C. Assume $\exists\theta$ s.t. $\theta \models C$. If $\theta \models C_1$ then **1)** $\theta\Gamma, x : \theta Y \vdash M : \theta T_1$ by the induction hypothesis.
To show $\theta\Gamma \vdash x \Rightarrow M : \theta X$ (we call θJ) we assume $\theta \models C$ by a case-split on disjunction in C:

- If $(X == Y \to T_1 \wedge C_1)$ then we may use the premise as $\theta \models C_1$. With $X \mapsto Y \to T_1$ we get $\theta J = \theta\Gamma \vdash x \Rightarrow M : \theta Y \to \theta T_1$. Using **(1)** and inspection of θJ we can conclude with (Abs).
- If $(X == \mathsf{Ok})$ then $\theta J = \theta\Gamma \vdash x \Rightarrow M : \mathsf{Ok}$ which we may conclude with (Ok).
- If $(\bigvee\{Y_i == \mathsf{Ok}^c | Y_i \in 1\})$ then $\exists i$ s.t. $\theta J = \{x_i : \mathsf{Ok}^c | x_i \in \mathsf{dom}(\Gamma)\} \vdash x \Rightarrow M : \theta X$ which we can conclude with (OkC1) since at least one Ok^c variable typing exists.

CASE(CTApp). Assume (CTApp) holds giving us $\Gamma \Vdash M\,(N) : X \mid C$ for some fresh type variables X, where $C = (T_1 == T_2 \to X \wedge C_1 \wedge C_2) \vee (\mathsf{Disj}(T_1, \mathsf{Ok}^c \to X) \wedge C_1) \vee (T_2 == \mathsf{Ok}^c \wedge C_2) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$. The premises are of shape $\Gamma \Vdash N : T_1 \mid C_1$ and $\Gamma \Vdash P : T_2 \mid C_2$ which give us the remaining variables in C. Assume $\exists\theta$ s.t. $\theta \models C$. If $\theta \models C_1$ then **1)** $\theta\Gamma \vdash N : \theta T_1$ by the induction hypothesis. If $\theta \models C_2$ then **2)** $\theta\Gamma \vdash P : \theta T_2$ by the induction hypothesis. To show $\theta\Gamma \vdash M\,(N) : \theta X$ (we call θJ) we assume $\theta \models C$ by a case-split on disjunction in C:

- If $(\bigvee\{Y_i == \mathsf{Ok}^c | Y_i \in 1\})$ then $\exists i$ s.t. $\theta J = \{x_i : \mathsf{Ok}^c | x_i \in \mathsf{dom}(\Gamma)\} \vdash N\,(P) : \theta X$ which is true under (OkC1).
- If $(X == \mathsf{Ok})$ then $\theta J = \Gamma \vdash N\,(P) : \mathsf{Ok}$ which is true under (Ok).
- If $(T_2 == \mathsf{Ok}^c \wedge C_2)$ then $\theta \models C_2$. Our second premise is concluded and thereby **(2)** looks like $\theta\Gamma \vdash P : \theta T_2 = \theta\Gamma \vdash P : \mathsf{Ok}^c$. Lastly θJ places no constraints on X so we conclude θJ under (App3).
- If $\theta \models (C_1 \wedge \mathsf{Disj}(T_1, \mathsf{Ok}^c \to X))$ then $\theta \models C_1$ and so our first premise **(1)** is satisfied. It looks like $\theta\Gamma \vdash N : \theta T_1$ where $\theta T_1 \,||\, \mathsf{Ok}^c \to X$. Lastly θJ places no constraints on X so we conclude θJ under (App2).
- If $(T_1 = T_2 \to X \wedge C_1 \wedge C_2)$ then $\theta \models C_1$ and $\theta \models C_2$. Both premises are satisfied and for our first **(1)** we have $\theta\Gamma \vdash N : \theta T_1 = \theta\Gamma \vdash N : T_2 \to X$. We conclude θJ with (App1).

CASE(CTNumOp). Assume (CTNumOP) holds giving us $\Gamma \Vdash N \circ P : X \mid C$ for some fresh type variable X, where $C = (X == \mathsf{Num} \wedge T_1 == \mathsf{Num} \wedge T_2 == \mathsf{Num} \wedge C_1 \wedge C_2) \vee (\mathsf{Disj}(T_1, \mathsf{Num}) \wedge C_1) \vee (\mathsf{Disj}(T_2, \mathsf{Num}) \wedge C_2) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$. We have the following premises $\Gamma \Vdash N : T_1 \mid C_1$ and $\Gamma \Vdash P : T_2 \mid C_2$ which give us the remaining variables in C. Assume $\exists\theta$ s.t. $\theta \models C$. If $\theta \models C_1$ then **1)** $\theta\Gamma \vdash N : \theta T_1$ by the induction hypothesis. If $\theta \models C_2$ then **2)** $\theta\Gamma \vdash P : \theta T_2$ by the induction hypothesis.
To show $\theta\Gamma \vdash N \circ P : \theta X$ (we call θJ) we assume $\theta \models C$ by a case-split on disjunction in C:

- If $(\bigvee\{Y_i == \mathsf{Ok}^c | Y_i \in 1\})$ then $\exists i$ s.t. $\theta J = \{x_i : \mathsf{Ok}^c | x_i \in \mathsf{dom}(\Gamma)\} \vdash N \circ P : \theta X$ which is true under (OkC1).
- If $(X == \mathsf{Ok})$ then $\theta J = \Gamma \vdash N \circ P : \mathsf{Ok}$ which is true under (Ok).
- If $(X == \mathsf{Num} \wedge T_1 == \mathsf{Num} \wedge T_2 == \mathsf{Num} \wedge C_1 \wedge)$ then since $\theta \models C_1$ and $\theta \models C_2$ we get **(1)** and **(2)**. Performing the substitution $\theta J = \theta\Gamma \vdash N \circ P : \mathsf{Num}$. Thus by this shape and its premises we can conclude θJ with (NumOp1).
- If $(\mathsf{Disj}(T_1, \mathsf{Num}) \wedge C_1)$ then $\theta \models C_1$ therefore we get **(1)**. $T_1 \,||\, \mathsf{Num}$ for this θ as required by the side-condition of (NumOp2) when $i = 1$. $\theta J = \theta\Gamma \vdash N \circ P : X$ i.e. X remains free, so we conclude θJ with (NumOp2).
- If $(\mathsf{Disj}(T_2, \mathsf{Num}) \wedge C_2)$ then $\theta \models C_2$ therefore we get **(2)**. $T_2 \,||\, \mathsf{Num}$ for this θ as required by the side-condition of (NumOp2) when $i = 2$. $\theta J = \theta\Gamma \vdash N \circ P : X$ i.e. X remains free, so we conclude θJ under (NumOp2).

CASE(CTIfZ). Assume (CTIfZ) holds giving us $\Gamma \Vdash N \texttt{ <= 0 ? } P \texttt{ : } Q : X \mid C$ for some fresh type variable X, where $C = (T_2 == T_3 \wedge T_2 == X \wedge C_2 \wedge C_3) \vee (\mathsf{Disj}(T_1, \mathsf{Num}) \wedge C_1) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$. We have the premises $N \Vdash T_1 : C_1 \mid$, $P \Vdash T_2 : C_2 \mid$ and $Q \Vdash T_3 : C_3 \mid$. Assume $\exists\theta$ s.t. $\theta \models C$. If $\theta \models C_1$ then **1)** $\theta\Gamma \vdash N : \theta T_1$ by the induction hypothesis. If $\theta \models C_2$ then **2)** $\theta\Gamma \vdash P : \theta T_2$ by the induction hypothesis. Similarly if $\theta \models C_3$ then **3)** $\theta\Gamma \vdash Q : \theta T_3$ by the induction hypothesis. To show $\theta\Gamma \vdash N \texttt{ <= 0 ? } P \texttt{ : } Q : \theta X$ (we call θJ) we assume $\theta \models C$ by a case-split on disjunction in C:

- If $(\mathsf{Disj}(T_1, \mathsf{Num}) \wedge C_1)$ then $\theta \models C_1$ which gives us **(1)**. $\theta T_1 \,||\, \mathsf{Num}$ which satisfies the side condition for (IfZ1). Finally we note X remains free so this allows us to conclude θJ with (IfZ1).
- If $(T_2 == T_3 \wedge T_2 == X \wedge C_2 \wedge C_3)$ then $\theta \models C_2$ and $\theta \models C_3$ are satisfied so we get **(2)** and **(3)**. $\theta T_2 == \theta T_3$ and transitively $\theta T_3 == \theta X$ which lets us conclude θJ with (IfZ2).
- If $(\bigvee\{Y_i == \mathsf{Ok}^c | Y_i \in 1\})$ then $\exists i$ s.t. $\theta J = \{x_i : \mathsf{Ok}^c | x_i \in \mathsf{dom}(\Gamma)\} \vdash N \texttt{ <= 0 ? } P \texttt{ : } Q : \theta X$ which is true under (OkC1).
- If $(X == \mathsf{Ok})$ then $\theta J = \Gamma \vdash N \texttt{ <= 0 ? } P \texttt{ : } Q : \mathsf{Ok}$ which is true under (Ok).

CASE(CTBlk). Assume (CTBlk) holds giving us $\Gamma \Vdash \{\mathcal{B}\} : X \mid C$ for some fresh type variable X, where $C = (C_1 \wedge T_1 == X) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$. We have the premise $\Gamma \Vdash \mathcal{B} : T_1 \mid C_1$. Assume $\exists\theta$ s.t. $\theta \models C$. If $\theta \models C_1$ then **1)** $\theta\Gamma \vdash \mathcal{B} : \theta T_1$ by the induction hypothesis.
To show $\theta\Gamma \vdash \{\mathcal{B}\} : \theta X$ (we call θJ) we assume $\theta \models C$ by a case-split on disjunction in C:

- If $(C_1 \wedge T_1 == X)$ then $\theta \models C_1$ giving us **(1)**. We also get that $\theta T_1 == \theta X$ which is necessary and sufficient given the premise to conclude with (Blk).

- If $(\bigvee\{Y_i == \mathsf{Ok}^c | Y_i \in 1\})$ then $\exists i$ s.t. $\theta J = \{x_i : \mathsf{Ok}^c | x_i \in \mathsf{dom}(\Gamma)\} \vdash \{\mathcal{B}\} : \theta X$ which is true under (OkC1).
- If $(X == \mathsf{Ok})$ then $\theta J = \Gamma \vdash \{\mathcal{B}\} : \mathsf{Ok}$ which is true under (Ok).

A.2 Completeness of Constraint Rules

Theorem A.2.1 (Constraint Completeness) *$\forall \sigma$, if $\Gamma \vdash M : A$ then $\exists C$ s.t. $\sigma\Gamma \Vdash M : X \mid C$ and $\exists \theta$ s.t. $\theta \models C$, $A = \theta X$ and $\sigma\Gamma = \theta\Gamma$.*

We proceed by induction on the ordinary rules to show they can be represented by the constraint system under some type variable substitution σ.

CASE(Num). Assume $\sigma\Gamma \vdash \underline{n} : \mathsf{Num}$. Let C be $(X == \mathsf{Num}) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$. By Definition (CTNum) applies; $\Gamma \Vdash \underline{n} : X \mid C$ where X is fresh. We set $\theta := [X \mapsto \mathsf{Num}] \cup \sigma$, thereby we have $\theta C = (\mathsf{Num} == \mathsf{Num})$ so $\theta \models C$. As required we get $\mathsf{Num} = \theta X$ and conclude that $\sigma\Gamma = \theta\Gamma$ since X is fresh.

CASE(Var). Assume $\sigma\Gamma, x : \sigma A' \vdash x : \sigma A'$. Let C be $(X == A') \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$, By Definition (CTVar) applies; $\Gamma, x : A' \Vdash x : X \mid C$ where X is fresh. We set $\theta := [X \mapsto \sigma A'] \cup \sigma$ thus we have $\theta C = (\sigma A' == \theta A') = (\sigma A' == \sigma A')$ therefore $\theta \models C$. We also get $\sigma A' = \theta X$ by our choice of θ. Finally, since X was fresh we get that $\sigma\Gamma = \theta\Gamma$.

CASE(Abs). Assume $\sigma\Gamma \vdash x \Rightarrow M : A \to B$, where $x \notin \mathsf{dom}(\Gamma)$. We proceed on the premises. Let $\Gamma' = (\Gamma, x : Y)$, $\sigma' = \sigma \cup [Y \mapsto A]$ and so we construct the judgement $\sigma'\Gamma' \vdash M : B$ for the premise. Thus **IH** gives us $\exists C_1$ s.t. $\Gamma, x : Y \Vdash M : T_1 \mid C_1$ and $\exists \theta'$ s.t. $\theta' \models C_1$. Thereby we get $B == \theta' T_1$, and $\sigma'(\Gamma, x : Y) = \theta'(\Gamma, x : A)$ by the induction hypothesis. Returning to our conclusion, we let $C = (X == Y \to T_1 \wedge C_1) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$. By Definition (CTAbsInf) applies: $\Gamma \Vdash x \Rightarrow M : X \mid C$. We set $\theta = [X \mapsto A \to B] \cup \theta' \cup \sigma'$ then we have θC as (where underlined expressions are satisfied):

$$\begin{aligned} \theta C &= (A \to B == \sigma' Y \to \theta' T_1 \wedge \underline{\theta' C_1}) \qquad \textit{by IH} \\ &= (\underline{A \to B == A \to B} \wedge \underline{\theta' C_1}) \end{aligned}$$

So $\theta \models C$. By our θ it follows $A \to B = \theta X$. It remains to show $\sigma\Gamma = \theta\Gamma$. We get that from the **IH** $\sigma'\Gamma = \theta'\Gamma$, where since Y is fresh, $\sigma'\Gamma = \sigma\Gamma$. Since X is fresh, $\theta\Gamma = \theta'\Gamma$ thus $\sigma\Gamma = \sigma'\Gamma = \theta'\Gamma = \theta\Gamma$.

CASE(App1). Assume $\sigma\Gamma \vdash M\,(N) : A$. We proceed on the premises in (App1). Let $\sigma' = \sigma$ and the first premise be $\sigma'\Gamma \vdash M : B \to A$. By the induction hypothesis we get $\exists C_1$ s.t. $\Gamma \Vdash M : T_1 \mid C_1$ and $\exists \theta'$ s.t. $\theta' \models C_1$. Thereby we get $B \to A = \theta' T_1$, and $\sigma'\Gamma = \theta'\Gamma$ by the induction hypothesis. Let $\sigma'' = \sigma$ and the second premise be $\sigma''\Gamma \vdash N : B$. By the induction hypothesis we get $\exists C_2$ s.t. $\Gamma \Vdash N : T_2 \mid C_2$ and $\exists \theta'' \models C_2$. Thereby we get $B = \theta'' T_2$, and $\sigma''\Gamma = \theta''\Gamma$ by the induction hypothesis. Returning to our conclusion, we set $C = (T_1 == T_2 \to X \wedge C_1 \wedge C_2) \vee (\mathsf{Disj}(T_1, \mathsf{Ok}^c \to X) \wedge C_1) \vee (T_2 == \mathsf{Ok}^c \wedge C_2) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$, where X is fresh. By Definition (CTApp) applies: $\Gamma \Vdash M\,(N) : X \mid C$. We set $\theta = [X \mapsto A] \cup \sigma \cup \theta' \cup \theta''$, where $\theta' \cup \theta''$ is a proper substitution because each constraint rule application introduces its X fresh. Then we have θC as:

$$\begin{aligned} \theta C &= \theta(T_1 == T_2 \to X \wedge C_1 \wedge C_2) \\ &= (\theta' T_1 == \theta'' T_2 \to \theta X \wedge \underline{\theta' C_1} \wedge \underline{\theta'' C_2}) \qquad \textit{by IH} \\ &= (\underline{B \to A == B \to A} \wedge \underline{\theta' C_1} \wedge \underline{\theta'' C_2}) \end{aligned}$$

So $\theta \models C$. By our θ it follows $A = \theta X$. It remains to show $\sigma\Gamma = \theta\Gamma$. Note $\sigma'\Gamma = \sigma\Gamma = \sigma''\Gamma$, thus $\sigma\Gamma = \theta'\Gamma = \theta''\Gamma$. Since X is fresh it does not appear in Γ thus $\theta\Gamma = \theta'\Gamma = \theta''\Gamma$, thereby we have $\sigma\Gamma = \theta'\Gamma = \theta''\Gamma = \theta\Gamma$ as required.

CASE(App2). Assume $\sigma\Gamma \vdash M\,(N) : A$. We proceed on the premise of (App2). Let $\sigma' = \sigma$ and the premise $\sigma'\Gamma \vdash M : B$ where $\forall D, B \mid\mid \mathsf{Ok} \to D$. The induction hypothesis gives us that $\exists C_1$ s.t. $\Gamma \Vdash M : T_1 \mid C_1$ and $\exists \theta'$ s.t. $\theta' \models C_1$. Thereby we get $B = \theta' T_1$, and $\sigma'\Gamma = \theta'\Gamma$ by the induction hypothesis. Returning to our conclusion, we set $C = (T_1 == T_2 \to X \wedge C_1 \wedge C_2) \vee (\mathsf{Disj}(T_1, \mathsf{Ok}^c \to X) \wedge C_1) \vee (T_2 == \mathsf{Ok}^c \wedge C_2) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$ where X is fresh. By definition we see (CTApp) applies: $\Gamma \Vdash M\,(N) : X \mid C$. Let $\theta = [X \mapsto A] \cup \theta' \cup \sigma$. Then we have θC as:

$$\begin{aligned} \theta C &= \theta(\mathsf{Disj}(T_1, \mathsf{Ok}^c \to D) \wedge C_1) \\ &= \mathsf{Disj}(\theta' T_1, \mathsf{Ok}^c \to D) \wedge \underline{\theta' C_1} \qquad \textit{by IH} \\ &= \underline{\mathsf{Disj}(B, \mathsf{Ok}^c \to D)} \wedge \underline{\theta' C_1} \qquad \textit{by IH} \end{aligned}$$

So $\theta \models C$. By our θ we get that $A = \theta X$. It remains to show $\sigma\Gamma = \theta\Gamma$. By the induction hypothesis we have $\sigma'\Gamma = \sigma\Gamma$, and since X is fresh $\theta'\Gamma = \theta\Gamma$. Therefore we have $\sigma\Gamma = \sigma'\Gamma = \theta'\Gamma = \theta\Gamma$ as required.

CASE(App3). Assume $\sigma\Gamma \vdash M\,(N) : A$. We proceed on the premise of (App3). Let $\sigma' = \sigma$ and write the premise as $\sigma'\Gamma \vdash N : \mathsf{Ok}^c$. The induction hypothesis then gives us $\exists C_2$ s.t. $\Gamma \Vdash N : T_2 \mid C_2$ and $\exists \theta'$ s.t. $\theta' \models C_2$. Thereby we get $\mathsf{Ok}^c = \theta' T_2$, and $\sigma'\Gamma = \theta'\Gamma$ by the induction hypothesis. Returning to our conclusion, we set $C = (T_1 == T_2 \to X \wedge C_1 \wedge C_2) \vee (\mathsf{Disj}(T_1, \mathsf{Ok}^c \to X) \wedge C_1) \vee (T_2 == \mathsf{Ok}^c \wedge C_2) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$ where X is fresh. By definition we see (CTApp) applies: $\Gamma \Vdash M\,(N) : X \mid C$. Let $\theta = [X \mapsto A] \cup \theta'$. Then we have θC as:

$$\begin{aligned}\theta C &= \theta(T_2 == \mathsf{Ok}^c \wedge C_2) \\ &= \theta' T_2 == \mathsf{Ok}^c \wedge \underline{\theta' C_2} \qquad \textit{by IH} \\ &= \underline{\mathsf{Ok}^c == \mathsf{Ok}^c} \wedge \underline{\theta' C_2}\end{aligned}$$

So θC. By our θ we get that $A = \theta X$. It remains to show $\sigma\Gamma = \theta\Gamma$. Since $\sigma' = \sigma$ we have $\sigma'\Gamma = \sigma\Gamma$, and since X is fresh $\theta'\Gamma = \theta\Gamma$. Therefore we have $\sigma\Gamma = \sigma'\Gamma = \theta'\Gamma = \theta\Gamma$ as required.

CASE(IfZ1). Assume $\sigma\Gamma \vdash M \texttt{ <= 0 ? } N \texttt{ : } P : A$. We proceed on the premise of (IfZ1). Let $\sigma' = \sigma$ and write the premise as $\sigma'\Gamma \vdash M : B$ where $B \,||\, \mathsf{Num}$. By the induction hpyothesis we get that $\exists C_1$ s.t. $\Gamma \Vdash M : T_1 \mid C_1$ and $\exists \theta'$ s.t. $\theta' \models C_1$. We therefore have $B = \theta' T_1$, and $\sigma'\Gamma = \theta'\Gamma$. Returning to the conclusion, we set $C = (T_2 == T_3 \wedge T_2 == X \wedge C_3 \wedge \vee) \vee (\mathsf{Disj}(T_1, \mathsf{Num}) \wedge C_2)(X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$ where X is fresh. By defintion (CTIfZ) applies: $\Gamma \Vdash M \texttt{ <= 0 ? } N \texttt{ : } P : X \mid C$. Let $\theta = [X \mapsto A] \cup \theta$, then we have θC as:

$$\begin{aligned}\theta C &= \theta(\mathsf{Disj}(T_1, \mathsf{Num}) \wedge C_1) \\ &= \mathsf{Disj}(\theta T_1, \mathsf{Num}) \wedge \underline{\theta' C_1} \qquad \textit{by IH} \\ &= \mathsf{Disj}(\theta' T_1, \mathsf{Num}) \wedge \underline{\theta' C_1} \\ &= \underline{\mathsf{Disj}(B, \mathsf{Num})} \wedge \underline{\theta' C_1} \qquad \textit{by IH}\end{aligned}$$

So $\theta \models C$. By our θ we get that $A = \theta X$. It remains to show $\sigma\Gamma = \theta\Gamma$. Since $\sigma = \sigma'$, $\sigma\Gamma = \sigma'\Gamma$, and since X is fresh $\theta'\Gamma = \theta\Gamma$. Thus $\sigma\Gamma = \sigma'\Gamma = \theta'\Gamma = \theta\Gamma$ as required.

CASE(IfZ2). Assume $\sigma\Gamma \vdash M \texttt{ <= 0 ? } N \texttt{ : } P : A$. We proceed on both the premises of (IfZ2). Let $\sigma' = \sigma$ and write the premise as $\sigma' \vdash N : A$. By the induction hypothesis $\exists C_2$ s.t. $\Gamma \Vdash N : T_2 \mid C_2$ and $\exists \theta'$ s.t. $\theta' \models C_2$. Thereby we get $A = \theta' T_2$, and $\sigma'\Gamma = \theta'\Gamma$ by the induction hypothesis. Let $\sigma'' = \sigma$ and write the premise as $\sigma' \vdash P : A$. By the induction hypothesis we get $\exists C_3$ s.t. $\Gamma \Vdash P : T_3 \mid C_3$ and $\exists \theta''$ s.t. $\theta'' \models C_3$. Thereby $A = \theta'' T_2$, and $\sigma''\Gamma = \theta''\Gamma$ by the induction hypothesis. Returining to the conclusion, we set $C = (T_2 == T_3 \wedge T_2 == X \wedge C_2 \wedge C_3) \vee (\mathsf{Disj}(T_1, \mathsf{Num}) \wedge C_1) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$ where X is fresh. By Definition (CTIfZ) applies: $\Gamma \Vdash M \texttt{ <= 0 ? } N \texttt{ : } P : X \mid C$. We let $\theta = [X \mapsto A] \cup \theta' \cup \theta''$, where $\theta' \cup \theta''$ is a valid substitution because all our constraint rules introduce fresh type variable X. We then have θC as:

$$\begin{aligned}\theta C &= \theta T_2 == \theta T_3 \wedge \theta T_2 == \theta X \wedge \theta C_2 \wedge \theta C_3 \\ &= \theta' T_2 == \theta'' T_3 \wedge \theta' T_2 == A \wedge \underline{\theta' C_2} \wedge \underline{\theta'' C_3} \\ &= (\underline{A == A} \wedge \underline{A == A} \wedge \underline{\theta' C_2} \wedge \underline{\theta'' C_3}\end{aligned}$$

So $\theta \models C$. By our θ we get that $A = \theta X$. It remains to show $\sigma\Gamma = \theta\Gamma$. Since $\sigma = \sigma' = \sigma''$ we get $\sigma'\Gamma = \sigma\Gamma = \sigma''\Gamma$, thus $\sigma\Gamma = \theta'\Gamma = \theta''\Gamma$. Since X is fresh we also have $\theta'\Gamma = \theta''\Gamma = \theta\Gamma$. Therefore $\sigma\Gamma = \theta'\Gamma = \theta''\Gamma = \theta\Gamma$ as required.

CASE(NumOp1). Assume $\sigma\Gamma \vdash M \circ N : \mathsf{Num}$ and proceed on the premises for (NumOp1). Let $\sigma' = \sigma$ and write the first premise as $\sigma\Gamma \vdash M : \mathsf{Num}$. By the induction hypothesis $\exists C_1$ s.t. $\Gamma \Vdash M : T_1 \mid C_1$, and $\exists \theta'$ s.t. $\theta' \models C_1$. Thereby we get $\mathsf{Num} = \theta' T_1$, and $\sigma'\Gamma = \theta'\Gamma$ by the induction hypothesis.
Let $\sigma'' = \sigma$ and write the second premise as $\sigma\Gamma \vdash N : \mathsf{Num}$. By the induction hypothesis $\exists C_2$ s.t. $\Gamma \Vdash N : T_2 \mid C_2$ and $\exists \theta''$ s.t. $\theta'' \models C_2$. Thereby we get $\mathsf{Num} = \theta'' T_2$ and $\sigma''\Gamma = \theta''\Gamma$ by the induction hypothesis. Returning to the conclusion, we set $C = (X == \mathsf{Num} \wedge T_1 == \mathsf{Num} \wedge T_2 == \mathsf{Num} \wedge C_1 \wedge C_2) \vee (\mathsf{Disj}(T_1, \mathsf{Num}) \wedge C_1) \vee (\mathsf{Disj}(T_2, \mathsf{Num}) \wedge C_2) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c | (y_i : Y_i) \in \Gamma\})$ where X is fresh. By Definition (CTNumOp) applies: $\Gamma \Vdash M \circ N : X \mid C$. We let $\theta = [X \mapsto \mathsf{Num}] \cup \theta' \cup \theta''$ where $\theta' \cup \theta''$ is assumed a valid substitution due to a fresh X introduced for every constraint rule. Then we have θC as:

$$\begin{aligned}\theta C &= (\theta X == \mathsf{Num} \wedge \theta T_1 == \mathsf{Num} \wedge \theta'' T_2 == \mathsf{Num} \wedge \underline{\theta' C_1} \wedge \underline{\theta'' C_2}) \qquad \textit{by IH} \\ &= (\underline{\mathsf{Num} == \mathsf{Num}} \wedge \underline{\mathsf{Num} == \mathsf{Num}} \wedge \underline{\mathsf{Num} == \mathsf{Num}} \wedge \underline{\theta' C_1} \wedge \underline{\theta'' C_2})\end{aligned}$$

So $\theta \models C$. By our θ we get that $\mathsf{Num} = \theta X$. It remains to show $\sigma\Gamma = \theta\Gamma$. Since $\sigma = \sigma' = \sigma''$ we get $\sigma'\Gamma = \sigma\Gamma = \sigma''\Gamma$, thus $\sigma\Gamma = \theta'\Gamma = \theta''\Gamma$. Since X is fresh we also have $\theta'\Gamma = \theta''\Gamma = \theta\Gamma$. Therefore $\sigma\Gamma = \theta'\Gamma = \theta''\Gamma = \theta\Gamma$ as required.

CASE(NumOp2). Assume $\sigma\Gamma \vdash M_1 \circ M_2 : A$ and we proceed on the premise of (NumOp2). Let $i \in \{1, 2\}$ and let $\sigma_i = \sigma$ and write the premise as $\sigma_i\Gamma \vdash M_i : B_i$. By the induction hypothesis we get $\exists C_i$ s.t. $\Gamma \Vdash M_i : T_i \mid C_i$, and $\exists \theta_i$ s.t. $\theta_i \models C_i$. Thereby we get $B_i = \theta_i T_i$ and $\sigma_i\Gamma = \theta_i\Gamma$ by the induction hypothesis. Returning to the conclusion, for (NumOp2) to either **1)** $B_1 \mid\mid \mathsf{Num}$ or **2)** $B_2 \mid\mid \mathsf{Num}$. We set $C = (X == \mathsf{Num} \wedge T_1 == \mathsf{Num} \wedge T_2 == \mathsf{Num} \wedge C_1 \wedge C_2) \vee (\mathsf{Disj}(T_1, \mathsf{Num}) \wedge C_1) \vee (\mathsf{Disj}(T_2, \mathsf{Num}) \wedge C_2) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c \mid (y_i : Y_i) \in \Gamma\})$ where X is fresh. By Definition (CTNumOp) applies: $\Gamma \Vdash M \circ N : X \mid C$. We proceed by case split on i:

- **CASE(1).** Suppose $B_1 \mid\mid \mathsf{Num}$, let $\theta' = [X \mapsto A] \cup \theta_1$. Then we have $\theta' C$ as follows:

$$\begin{aligned}
\theta' C &= (\mathsf{Disj}(\theta' T_1, \mathsf{Num}) \wedge \theta' C_1) \vee (\mathsf{Disj}(\theta' T_2, \mathsf{Num}) \wedge \theta' C_2) \\
&= (\mathsf{Disj}(\theta_1 T_1, \mathsf{Num}) \wedge \underline{\theta_1 C_1}) \vee (\mathsf{Disj}(\theta' T_2, \mathsf{Num}) \wedge \theta' C_2) && \textit{by IH} \\
&= (\underline{\mathsf{Disj}(B_1, \mathsf{Num})} \wedge \underline{\theta_1 C_1}) \vee (\mathsf{Disj}(\theta' T_2, \mathsf{Num}) \wedge \theta' C_2) && \textit{by supposition}
\end{aligned}$$

  So $\theta' \models C$. By our choice for θ' we get $A = \theta' X$. It remains to show $\sigma\Gamma = \theta'\Gamma$; note $\sigma = \sigma_1 = \sigma_2$ and thus $\sigma\Gamma = \sigma_1\Gamma = \sigma_2\Gamma$, which gives us $\sigma\Gamma = \theta_1\Gamma = \theta_2\Gamma$. Since X is fresh we also have $\theta'\Gamma = \theta_1\Gamma = \theta_2\Gamma$ which lets us conclude $\sigma\Gamma = \sigma_1\Gamma = \sigma_2\Gamma = \theta'\Gamma$ as required.

- **CASE(2).** Suppose $B_2 \mid\mid \mathsf{Num}$, let $\theta'' = [X \mapsto A] \cup \theta_2$. Then we have $\theta'' C$ as follows:

$$\begin{aligned}
\theta'' C &= (\mathsf{Disj}(\theta'' T_1, \mathsf{Num}) \wedge \theta'' C_1) \vee (\mathsf{Disj}(\theta'' T_2, \mathsf{Num}) \wedge \theta'' C_2) \\
&= (\mathsf{Disj}(\theta'' T_1, \mathsf{Num}) \wedge \theta'' C_1) \vee (\mathsf{Disj}(\theta_2 T_2, \mathsf{Num}) \wedge \underline{\theta_2 C_2}) && \textit{by IH} \\
&= (\mathsf{Disj}(\theta'' T_1, \mathsf{Num}) \wedge \theta'' C_1) \vee (\underline{\mathsf{Disj}(B_2, \mathsf{Num})} \wedge \underline{\theta_2 C_2}) && \textit{by supposition}
\end{aligned}$$

  So $\theta'' \models C$. By our choice for θ'' we get $A = \theta'' X$. It remains to show $\sigma\Gamma = \theta''\Gamma$; note $\sigma = \sigma_1 = \sigma_2$ and thus $\sigma\Gamma = \sigma_1\Gamma = \sigma_2\Gamma$, which gives us $\sigma\Gamma = \theta_1\Gamma = \theta_2\Gamma$. Since X is fresh we also have $\theta''\Gamma = \theta_1\Gamma = \theta_2\Gamma$ which lets us conclude $\sigma\Gamma = \sigma_1\Gamma = \sigma_2\Gamma = \theta''\Gamma$ as required.

CASE(Blk). Assume $\Gamma \vdash \{\mathcal{B}\} : A$ and proceed on the premise of (Blk). Let $\sigma' = \sigma$ and so rewrite the premise as $\sigma'\Gamma \vdash \{\mathcal{B}\} : A$. The induction hypothesis gives us $\exists C_1$ s.t. $\Gamma \Vdash \mathcal{B} : T_1 \mid C_1$ and $\exists \theta'$ s.t. $\theta \models C_1$. Thereby $A = \theta' T_1$ and $\sigma'\Gamma = \theta'\Gamma$. Set $C = (X == T_1 \wedge C_1) \vee (X == \mathsf{Ok}) \vee (\bigvee\{Y_i == \mathsf{Ok}^c \mid (y_i : Y_i) \in \Gamma\})$. By defn (CTBlk) applies: $\Gamma \Vdash \{\mathcal{B}\} : X \mid C$. Let $\theta = [X \mapsto A] \cup \theta'$ therefore θC is:

$$\begin{aligned}
C\theta &= (\theta X == \theta T_1 \wedge \theta C_1) \\
&= (A == \theta' T_1 \wedge \underline{\theta' C_1}) && \textit{by IH} \\
&= (\underline{A == A} \wedge \underline{\theta' C_1})
\end{aligned}$$

So $\theta \models C$ and we see $A = \theta X$. We need $\sigma\Gamma = \theta\Gamma$, notice $\sigma = \sigma'$ so $\sigma\Gamma = \sigma'\Gamma$ then $\sigma\Gamma = \theta'\Gamma$. Since X is fresh $\theta'\Gamma = \theta\Gamma$ so $\sigma\Gamma = \theta'\Gamma = \theta\Gamma$ as required.