School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Improving the Safety of the Pancake Language

by

Halogen Truong

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Software Engineering

Submitted: November 2025

Supervisor: Thomas Sewell

Student ID: z5362371

Abstract

Pancake is a research programming language with the goal of achieving low-level programming that is formal verification friendly. However, the compiler's lack of any kind of type safety, though originally intentional, has revealed to be a major usability issue as the language sees more rigorous use. The syntax for its simple type system compounds the issue, making code brittle to changes and prone to subtle mistakes. This project addresses this problem by extending the language and the compiler to shift burdens of identifying type errors away from the programmer. The solution focuses on reporting type errors when they occur, minimising the syntax burden associated with type checking and preventing type errors through new syntax that preserves programmer intent. Evaluating this solution against Pancake's key design goals reveals strong improvements in the usability of the language, and a number of further avenues of improvement.

Acknowledgements

I would like to thank Thomas Sewell for his guidance and patience as my supervisor throughout the year amidst my unending distractions.

I am also grateful to Craig McLaughlin, Liam O'Connor and Johannes Åman Pohjola for their literature recommendations, with extra thanks to Johannes who was always awake for advice and feedback when Australians were asleep.

Special thanks goes to the current Pancake user-base, especially Junming Zhao and Richard Shen, for providing thoughts and opinions on my changes.

Finally, I would like to thank my thesis peers for support during the project, with particular thanks to Thomas Liang as author of the original Pancake scope checker, which established the foundations for the conception of this project.

Contents

Chapter 1

Introduction

Pancake is an in-development programming language that is designed as an alternative to C for verifiable low-level programming. Its key design goals are ease of formal verification, a minimal trusted computing base and usability by C-familiar systems programmers. As part of this, it features a simplified, flexible memory model and a bare-bones, machine-word-based type system, with no safety enforcements for either.

However, unlike with memory safety, the lack of type safety compromises these core design goals, appearing to stem from neglect of the language's features relating to types (referred to as *shapes*). It most significantly threatens usability, as the lack of compiler checks for shapes forces the programmer to manually find and fix any type errors, and such errors may be subtle and unpredictable during run-time. This issue is exacerbated by inconvenient, error-prone and non-descriptive syntax for its tuple-like structs, compounding the programmer burden of manually inspecting their program's shapes. As such, the addition of shape safety to Pancake is an obvious target for improving system programmer usability.

The approach for adding shape safety is informed by other low-level languages and approaches to type safety in literature. Examining languages such as C and Rust reveals that it possible for safety to interfere with usability, especially with the language's usefulness and practicality, so the chosen approach must take care to minimise the introduction of new usability issues. Between the standard type safety enforcement strategies of type checking and type inference, checking offers understandability and practicality benefits in its familiarity to C programmers, and avoids the need for complicated inference around struct shapes. However, it also introduces a greater shape declaration burden than inference, which also compromises backwards compatibility. A bidirectional typing approach aims to provide a middle-ground with the advantages of both, but has limited precedent in imperative languages and should not be relied on as the primary approach.

Based on this motivation and examination of the literature, the project aims to improve the shape safety of Pancake as an avenue of improving language usability, especially by progressing and conversing the key design goals. In particular, the proposed solution consists of three parts: reacting to bad input using static shape checking; preserving existing syntax in the presence of

the shape information required for shape checking by adding a default shape; and preventing trivial shape errors by introducing C-like named structs to improve readability, robustness and practicality. This solution designates most verification work that is consequently prompted as out of the project scope.

These tasks are grouped into two phases, and this report details the design decisions and methodology that comprise their implementation. The first phase includes shape checking and the default shape feature, producing a pen-and-paper description of the checking algorithm using inference rules, extensions to the language to support the user providing additional shape information, implementations and tests of both features, and a successful pull request into the main Pancake compiler. The second phase consists of only the named structs feature, producing extensions to the language as well as its formal semantics, extensions to the description, implementation and testing of the shape checking to include named structs, a simple compilation pass that leverages the existing struct compilation strategy and a locally built, unverified compiler binary for testing and evaluation.

The solution is evaluated with respect to the Pancake design goals, comparing this evaluation with feedback from the target user-base, and limitations and future work are identified accordingly. These evaluations show that the solution strongly addresses the design goals, and usability in particular, but further improvements still remain to achieve the ideal levels of usability for Pancake's shape features.

Chapter 2

Motivation

2.1 The Pancake Language and Its Design

Pancake is a research programming language in development as a verification-friendly alternative to C for low-level systems programming [Pohjola et al., 2023]. It is being developed by the Trustworthy Systems (TS) research group at UNSW, in partnership with the University of Gothenburg, The Australian National University, and Chalmers University of Technology.

Within systems programming, Pancake is aimed at device driver implementation, particularly for TS projects such as Microkit [seL4 Foundation, 2023] and sDDF [Heiser et al., 2024], with more rigorous use beginning over the past year. To this end, its key design focuses are *ease of formal verification*, a *minimal trusted computing base*, and *usability by systems programmers*.

Ease of Verification. Formal verification of a given program, particularly the task of proving functional correctness, is closely reliant on the semantics of the language the program is written in — the specification for how each aspect of the language behaves. A key motivation behind the Pancake language is the cost and complexity of verifying C code, as its semantics feature pain points such as a complex memory model, ambiguous evaluation order and copious undefined behaviour. Instead, Pancake's formal semantics are designed to be simple, sidestepping these pitfalls so the language is easier to reason about and functional correctness proofs are simpler.

Minimal Trusted Computing Base. The *trusted computing base* is what is assumed correct for a program's proof to apply to its run-time behaviour. For proofs at the source code level, this connection is typically assumed correct in its entirety; that is, it is *trusted*. Pancake addresses this by verifying its compiler's *end-to-end correctness*, such that the machine code outputted is proven to preserve the semantic behaviour of the input program, *provided said behaviour is error free*. This correctness allows proofs at the Pancake level to apply to the compiler output, reducing the trusted gap to between the machine code and hardware behaviour. It consists of proofs for the individual compiler passes composed in sequence, and leverages the existing verified compiler of the functional CakeML language [Kumar et al., 2014] as shown in Figure 2.1, implemented in the HOL4 interactive theorem prover to directly integrate its proofs.

Systems Programmer Usability. In order to reap the verification benefits of Pancake, systems programmers must be willing and able to adopt and use it. As such, Pancake aims to be understandable, useful and practical to systems programmers, especially those well-versed in C. As part of this, the language puts great emphasis on programmer flexibility and freedom, since a key feature of C that makes it favoured by systems programmers is the easy low-level control that higher-level languages abstract away or prevent.

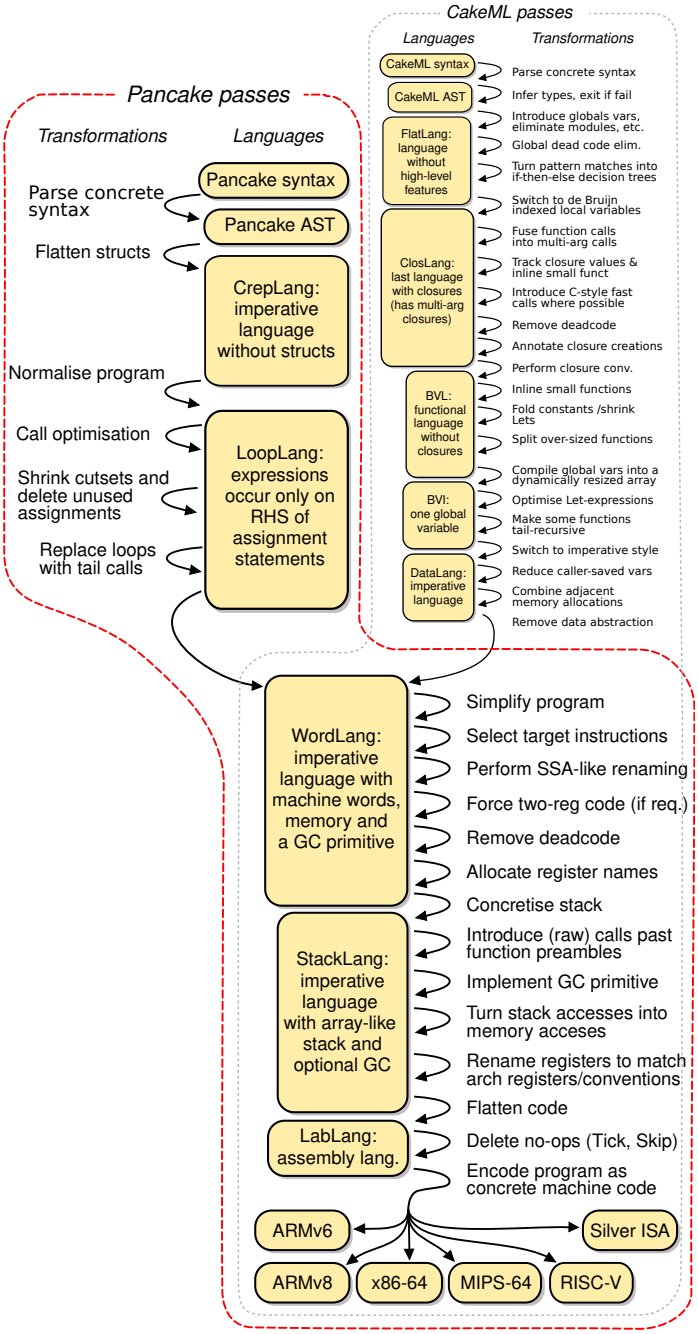

Figure 2.1: The shared compiler stack of Pancake and CakeML.

2.2 Pancake and Safety

The *safety* of a programming language is the extent to which it prevents certain classes of bugs and issues through compile-time and run-time features. Let us examine Pancake's treatment of safety through two popular language safety features: memory safety and type safety.

2.2.1 Memory Safety

A language that is *memory safe* prevents operations that may cause issues associated with memory, such as null pointer dereferences or out-of-bounds array accesses. Systems programmers often rely on C's lack of such safety; for example, being able to access seemingly arbitrary memory addresses is required when those addresses map to device memory.

Pancake does not enforce memory safety, and instead provides a simple and flexible memory model wherein the programmer may freely access statically-allocated user memory, and an interface for accessing memory shared with other programs. It takes the stance that memory safety unnecessarily limits programmer freedom, complicates the language semantics and does not provide any helpful information for creating functional correctness proofs. In other words, it compromises the language's design goals of ease of verification and systems programmer usability, while providing no benefit to the language's core purpose of facilitating formal verification.

To demonstrate Pancake's lack of memory safety, consider the following Pancake program:

```
1 fun main() {
2     var x = 0;
3     st 0, x;
4     return 1;
5 }
```

Listing 2.1: Minimal example of invalid memory access

This program attempts to store the value of variable `x` to the local memory address `0`, which is equivalent to accessing `NULL` in C. Since Pancake has no memory safety features, there are no checks to detect or prevent this during compilation nor run-time. When the program is run, it attempts to access the `0` address and, depending on the specific platform being run on, triggers a segmentation fault which crashes the program. Since this is considered an error in the Pancake semantics, the program cannot be proven correct and the compiler correctness is not applicable.

2.2.2 Type Safety

A language that is *type safe* describes its data according to a type system and prevents operations that violate this system, such as using a boolean where a character is expected or invoking subtraction on a string. While C does have and enforce a type system, it also provides the ability to *cast* data to other types. Systems programmers often use type casting to `void *` as a way to circumvent C's type system when moving and storing data that does not fit neatly within it.

In place of a traditional type system, Pancake has what it calls its *shape* system, which describes data only in terms of its memory representation. Namely, the possible shapes of data are machine words and structs, which are possibly-nested collections of machine words akin to tuples. Similarly to the memory model, this simplified type system allows for simpler semantics and greater programmer freedom, and Pancake has no safety features to enforce this type system. However, unlike with memory safety, it is not clear that this lack of safety is necessary or helpful; in fact, it appears to be an undesirable artefact from shapes' lack of development attention, which is further explored in Section 2.3.1.

To demonstrate Pancake's lack of shape safety, consider the following Pancake program:

```
1 fun main() {
2     var x = <0, 1, 3>;
3     return x.5;
4 }
```

Listing 2.2: Minimal example of invalid struct access

This program first declares a variable `x`, whose shape is a struct of 3 machine words. It then attempts to access the element at index 5—structs are zero-indexed—which does not exist. Like memory safety, there are no checks at compile- nor run-time to prevent this, and the compiler produces machine code as if the element existed. When the program runs, the value obtained will likely be 0, but in more complex programs this may result in garbage values that remain as artefacts of other structs in the program. The program then continues with this value, which can result in very subtle and possibly unpredictable bugs when the value is used elsewhere. Like `NULL` accesses, invalid field access is considered an error in the Pancake semantics, so functional correctness again cannot be proven nor does the compiler correctness guarantee apply.

2.3 Issues with Shapes in Pancake

2.3.1 The Causes

As alluded to in Section 2.2.2, though the initial decision against *shape safety* support in Pancake stems from the design stance on safety, Pancake's development being largely request-driven has been a notable factor in the continued neglect of more complex shape features. The bulk of Pancake programs thus far rely solely on machine words, revealing a self-perpetuating loop — Pancake users choose to avoid struct features since they are unpolished, and Pancake developers prioritise tasks other than struct features since they hear few requests to improve it.

Another contributing reason is the relationship between safety and compiler correctness. As mentioned in Section 2.2, safety issues such as memory or type errors lead to error states in the semantics. Since the compiler correctness proofs hinge on the input program being free from such errors, inputting programs containing such errors cause the correctness to be *vacuously true*. That is, since the condition that guards the proof guarantee is false, the compiler is free to output whatever it pleases without violating the proof, but the behaviour of the output machine code is not guaranteed. So, in the eyes of verification, the compiler is working correctly.

2.3.2 The Effects

Let us explore the effects of this neglect through the following Pancake program, containing a simple linked list representation and partial interface:

```
1  fun main() {
2      // {1,1}: head ptr, size
3      var llist = <@base, 0>;
4      // ...
5      llist = inc_sz(llist);
6      // ...
7      return 1;
8  }
9
10 // increase linked list size by 1
11 fun inc_sz({1,1} ll) {
12     var size = ll.1 + 1;
13     return <ll.0, size>;
14 }
```

Listing 2.3: Sample linked list representation

Here, we represent the linked list as a struct containing a pointer to the list's head node and its size. They are initialised on line 3 with the `@base` keyword, which holds the address of the bottom or *base* of the allocated user memory, and 0, respectively. Being two machine words, we write the resulting shape as `{1,1}`. We also define a function, `inc_sz`, which takes in such a linked list and returns an updated value for the struct with the size field incremented. This is done by accessing the size field at index 1 at line 12 and placing the new value of size in that position when returning (leaving the head pointer unchanged). We then, at some point, call this function on our linked list, as in line 5.

We then decide to extend our linked list representation by including a pointer to the list's tail node, slotting it between the existing fields to keep the pointers grouped. This gives the linked list the new shape `{1,1,1}`, but we do not update `inc_sz`:

```
1  fun main() {
2      // {1,1,1}: head ptr, tail ptr, size
3      /* previously var llist = <@base, 0>; */
4      var llist = <@base, @base, 0>;
5      // ...
6      llist = inc_sz(llist);
7      // ...
8      return 1;
9  }
10
11 // increase linked list size by 1
12 /* unchanged */
13 fun inc_sz({1,1} ll) {
14     var size = ll.1 + 1;
15     return <ll.0, size>;
16 }
```

Listing 2.4: Extended linked list representation

Since the `inc_sz` function has not been updated, it no longer works as intended, still expecting the old representation and using the index 1 for the size field. Calling this function with the new representation does not make sense since the argument and return value shapes do not mismatch, and the intention behind its field accesses has become misaligned from the code. Despite this, the compiler accepts it without complaint, producing output that increments the tail pointer and overrides the size field with a default value.

Although this is permitted for compiler correctness, the fact that the compiler does not identify and communicate these issues even though static type analysis is possible is an issue for usability and debugging. These errors are not being caught at compile-time and, in the worst case scenario, may not be reliably replicable during run-time. This puts extra burden on the programmer to detect, identify and fix such issues on their own, even though the compiler should be able to detect the shape mismatches present if given enough information. The lack of debugging support also becomes increasingly concerning as programs get larger and more realistic.

These usability issues are compounded by the struct syntax, which makes the code harder to read and debug when issues *are* present. For instance, it takes a careful eye and a good memory to notice that the size index used by `inc_sz` should be 2 instead of 1, and since the field shapes are identical, even static analysis cannot be of help since it has no awareness of the intention behind each field. It also makes the code quite brittle, as every function using the linked list representation must now be updated to account for the tail pointer, which becomes an even bigger issue as structs get larger, requiring every mention of it to be replicated exactly. While there is shorthand available for shapes—the new representation's shape can be simplified as `3`, for instance—the issue still stands that the syntax for defining struct shapes scales poorly. Additionally, there is no syntax available to update individual struct fields; to return the updated list value from `inc_sz`, we must reconstruct the entire struct, which must also be updated in tandem with the representation and has similar scalability issues.

From this example, we can see that the usability of Pancake suffers from a compiler that does not detect and report shape issues in incoming programs, and from error-prone struct syntax that poorly supports programmer intent and scalability, if at all. As such, unlike the treatment of memory which is simple and unsafe by design, Pancake's shapes are simple by design but unsafe by neglect. Shapes' lack of safety, along with the other side effects of their neglect, is a usability issue that threatens the core design goal of the Pancake language of being accessible to the systems programmers who will use it. Extending the compiler to address these issues, then, is a key target for improvement.

Chapter 3

Literature Review

3.1 Safety and Usability in Other Languages

Establishing how other low-level languages approach safety and how this interacts with usability at a high level can inform the concerns and goals in determining how to approach improving shape safety in Pancake.

C. C is considered the default language for low-level and systems programming, with origins tied to the implementation of the Unix operating system. The key reason for this is its relatively thin abstractions and high control over machine instructions and behaviours. Among these, as described in Section 2.2, is that it lacks memory safety and provides a method of subverting its type system (referred to as *type punning*) in the type casting feature. Systems programmers regularly rely on this divergence from safety to enable the seemingly unsafe operations that low-level code often demands.

Cyclone. Cyclone is a research language that extends C to address elements of type and memory safety [Grossman et al., 2005]. This dialect of C introduces a number of features to avoid errors such as out-of-bounds array accesses and `NULL` access without compromising the performance and memory control of C by, for example, introducing a full run-time environment as higher level languages do. It does so by adding information such as array bounds and union tags which come with compile- and run-time checks on top of C, with the latter implemented as safety checks inserted into the equivalent C code. This approach alleviates the programmer burden of writing those checks manually, making it easier to engage in safer memory and type usage in C-reliant situations.

Rust. Rust is a low-level language that applies stricter compile-time safety to systems programming to provide safety without the performance overhead of a run-time environment. This most notably includes tracking and limiting the access permissions of data, to address a variety of memory errors. While this is intended to be the primary state of Rust code, known as *safe Rust*, they also provide the `unsafe` keyword, allowing the programmer to deliberately mark code as exempt from these safety checks, known as *unsafe Rust*. Unsafe Rust is intended to be used

in situations where the safety features prevent the intended functionality under the following conditions: sparingly, with clear intention and behind safe abstractions. Notably, this includes supporting lower-level operations for which the compiler cannot guarantee safety, such as interfacing with devices or other programming languages. In practice, unsafe Rust finds usage both in and out of this intention [Astrauskas et al., 2020], and demonstrates the need for low-level code to step outside of safety features in order to be useful.

These examples support the stance that Pancake takes with memory safety, namely that the presence of safety can also be a threat to usability, particularly in terms of practicality and usefulness. As such, considered approaches must take care not to introduce new usability issues in the pursuit of addressing existing ones, such as by compromising control or degrading performance.

3.2 Type Safety Techniques

The natural starting point to determine an implementation for shape safety is examining the standard approaches for enforcing type safety at compile-time: type checking and type inference.

Type Checking. Type checking compares the pre-defined types of variables, functions and values with how they are used, and throws errors when they do not match. It requires types to be declared when creating new variables and functions—known as *manifest typing*—using these declarations as the sole source of type information, and is recognisable in languages such as C. As the algorithm walks through the input program's internal representation, it collects the types at each declaration and uses this with the types of literals and constants to determine whether the arguments to function calls and operations, including initialisation and assignment, have the correct types. As such, when type errors are reported, they often refer to *mismatched types*, quoting the expected and actual types found, and occur where the mismatch between usage and declared type was found.

Type Inference. Type inference looks only at the usage of variables and functions to determine their type, and throws errors from uses that contradict prior inferences [Hindley, 1969; Milner, 1978]. It requires code to be annotated with types only where usage alone is insufficient to determine the type, referred to as *implicit typing*, so type information comes from both usage and annotations. In this case, the algorithm initially regards variables and functions as having unknown types and incrementally builds up a collection of facts as it encounters usages of them. As it does so, it attempts to *unify* the new information with the existing facts, including to *instantiate* them with a type that is completely known. Hence, type errors reported from inference typically refer to *failed unification* (conflicting information) and *undecidability* (insufficient information), and may occur either at the point of misuse or when a later intended usage contradicts with that misuse.

3.2.1 Checking vs Inference

From inspection, a checking-based approach comes with the usability advantage of familiarity, since C programmers will be used to the manifest typing and error reporting style associated with it. Debugging is also clearer, as the compiler has all the information it needs to report issues, and similarly, the programmer can clearly track the compiler's understanding of the program's types in the syntax. This contrasts with type inference, where scouring the file for the missing or contradictory information, most of which is implicit, is necessary for fixing instances of undecidability or failed unification.

Checking also has the added benefit of avoiding inference about structs, also known as *record* or *row* types. Such inference would introduce the need to consider *row-polymorphism* since, for instance, accessing the first field of a struct is possible for any struct with more than one element. This is a deceptively complex endeavour [Simon, 2014], which risks unnecessary complexity and exceeding the scope of the project.

There is also some support in the literature for the usability benefits of checking. Though empirical studies in programming language design are scarce, one experiment found that static type checking was favourable over *dynamic typing*—where types are determined and checked at runtime—for debugging type errors [Hanenberg et al., 2014]. This result was attributed to error information being more informative and having better locality—that is, being reported closer to the source of the issue—which is also a benefit that checking, anecdotally, holds over inference.

However, a checking-based approach introduces a shape declaration burden that did not exist in the language previously, namely for variables and function return values. This impacts the practicality of the language, where inference would reduce the annotations required. It also introduces a backwards compatibility issue for existing Pancake code.

3.2.2 Bidirectional Typing

One less common technique that exists is *bidirectional typing*, which combines both checking and inference [Pierce and Turner, 2000]. This technique aims to avoid both the increased annotation burden of checking and the risk of undecidability from inference. As such, it is a promising alternative to purely checking or inference approaches.

Work has been done to develop a recipe and criteria for creating bidirectional typing rules from a non-bidirectional system [Dunfield and Krishnaswami, 2021]. This recipe describes a process of identifying and handling the following components:

- Variable lookups
- Explicit annotations
- Directional changes
- Syntax that introduces or eliminates a *type connective* (such as a function arrow)

The categorisation of introductions and eliminations, which makes up the bulk of the recipe, is closely related to functional language features. A likely candidate for such a type connective in

Pancake is structs, so one avenue of investigating bidirectional shapes is to ensure expressions with a struct shape are checked and not inferred, then apply the recipe around that. That being said, the authors note that there is difficulty in applying the recipe to imperative systems that do not align well with this introduction-elimination categorisation, and limited precedent was found for other work applying this recipe to imperative languages. This implies that, while a promising technique, it may not turn out to be a viable one for Pancake after looking at the entire type system, and so the chosen shape safety approach should not rely heavily on this possibility.

Chapter 4

Aim

The high-level aim of the project is to improve shape safety in Pancake from the perspective of a usability feature. It should do so without compromising the language's key design goals of verification ease, a minimal trusted computing base and usability for systems programmers.

To this end, the solution should progress the language's usability goals of understandability, usefulness and practicality. It should address known issues such as insufficient compiler support for diagnosing shape issues; and a syntax that increases the likelihood of errors and is blind to programmer intention. It should also conserve existing strengths, avoiding major degradations in, for instance, complexity, backwards compatibility or performance.

Following the focus on usability and the Pancake project's claim that static safety does not aid in program correctness, achieving formal type safety is not a priority for the project nor are any verification aspects that may arise from the solution. The solution should focus on the language design decisions and implementation, so substantial proof obligations may be delegated to future work but should still be considered against the Pancake design goals.

The project hypothesises that the aim is achievable with the following three step strategy.

React to bad input. The compiler should be extended with a shape safety enforcement step in the style of traditional *static type checking*. This step should detect input programs containing shape issues, prevent the program from reaching the main compilation sequence and report the issue to the programmer appropriately. The Pancake syntax should be extended such that users are able to declare sufficient shape information for the compiler to carry out the analysis.

Preserve existing good input. The additional declaration burden required for shape checking threatens both language practicality and backwards compatibility. To address this, noticing that the bulk of existing Pancake code features largely machine words, the syntax for shape declarations should allow omitted shapes, with a machine word shape assumed in its place for the purposes of shape checking. This simplified *default shape* approach is chosen over a bidirectional typing approach since the latter has limited precedent in being applied to imperative languages, while the former has more predictable benefits and feasible required effort.

Prevent future bad input. The current struct syntax is inconvenient, brittle, easy to misread and is not well-representative of programmer intent. Before even reaching shape checking, there is a high likelihood of shape errors in a program using structs, prompting additional debugging overhead. The language should be extended with an additional struct shape akin to C structs, where names may be declared for the overall shape and its fields. Referring to the existing struct shape as *raw structs* for disambiguation, this *named struct* feature can then encode programmer intent within the declared names and incorporate this information into shape checking, while also abstracting away the underlying shape to be more convenient and less brittle.

Other features, such as bidirectional typing, can then be addressed as stretch goals. Another stretch goal is addressing the lack of support for direct struct field assignment, a notable point of friction in struct usability but not directly related to the usability of shape safety.

The solution should be developed with the expectation of being integrated into the mainline compiler repository, bar any tasks delegated to future work.

The evaluation of the solution, and its composite design decisions, should consist of a number of qualitative judgements:

- Does the solution advance the usability design goal?
- Does the solution avoid degrading the language design goals?
- Do the target users agree with the previous two assessments?

These questions should then inform the limitations and future work of the project.

Chapter 5

Methodology

Implementation for the solution had two phases: the shape checking and default shape declarations as one phase, and the named struct feature as the second. This immediately offsets the declaration burden of checking through the default shape feature. Each phase was intended to be merged into the main repository upon completion. The stretch goals were not reached during the duration of the project.

5.1 Shape Checking and Default Shape Declaration

5.1.1 Checking Inference Rules

Work on the shape checker began with laying out the inference rules that dictate the conditions that make up a *shape correct* program, and therefore the component steps of the checking algorithm to derive this status. This largely served as a planning phase for the checking implementation — the rules exist only in pen-and-paper form, without a formal HOL4 definition, and were not used for any manual derivations.

The checks introduced at this phase of the project can be summarised as follows:

1. Values provided to variable initialisation and assignments must match their declared shape
2. Arguments provided to operators must match their expected shape
3. Arguments provided to functions must match their declared shape
4. Return values within functions must match their declared shape
5. Field accesses for structs must use an index that exists within the struct's shape
6. The declared return shape for functions must not exceed 32 words total
7. Functions declared for exporting using the multiple entry points feature must use single words for their arguments and return value

The first five of these conditions are self-explanatory, aligning with the established goals of type checking. The size limit on returned shapes comes from the language semantics and is tied to the

current implementation of function returns. The word limit on exported functions—functions that are exposed for calling from outside Pancake—is due to the limits of the platform calling conventions upon which the export feature is based.

The shape checking rules use the judgements summarised in Table 5.1, with the full rules for this phase listed in Appendix A. The syntax components used in these judgements—sh, exp, $stmt$, $decl$ and $program$—use the *abstract syntax* representation of Pancake used within the compiler. For instance, the shapes `1` and `{1,1}` are written as **One** and **Comb** [**One**, **One**]. Since some of these components may form part of another, judgements may reference other judgements in the rule set and/or be recursive.

Table 5.1: The shape checking judgements and their readings.

Judgement	**Reading**
$\Gamma_V; \Gamma_G \vdash exp : sh$	exp has shape sh in variable context $\Gamma_V; \Gamma_G$
$scope; \Gamma_V; \Gamma_G; \Gamma_F \vdash stmt$ **ok**	$stmt$ shape checks in $scope$, variable and function context $\Gamma_V; \Gamma_G; \Gamma_F$
$\Gamma_G; \Gamma_F \vdash decl \rightsquigarrow \Gamma'_G; \Gamma'_F$	$decl$ updates global variable and function context $\Gamma_G; \Gamma_F$ to produce $\Gamma'_G; \Gamma'_F$
$\Gamma_G; \Gamma_F \vdash decl$ **ok**	$decl$ shape checks in global variable and function context $\Gamma_G; \Gamma_F$
$program$ **ok**	$program$ shape checks

As an example, consider the following $stmt$ rule, DEC:

$$\frac{\Gamma_V; \Gamma_G \vdash exp : sh \quad scope; \Gamma_V, v : sh; \Gamma_G; \Gamma_F \vdash stmt \textbf{ ok}}{scope; \Gamma_V; \Gamma_G; \Gamma_F \vdash \textbf{Dec } v\ sh\ exp\ stmt \textbf{ ok}}\ \textsc{Dec}$$

This rule can be read as:

> The local variable declaration **Dec** v sh exp $stmt$ is shape correct in a given context if, first, the initialising expression exp matches the declared shape sh in that context and, second, the statement with this declaration in scope $stmt$ is shape correct when the variable v is added to the context with the shape sh.

While most of the rules follow naturally from the aforementioned checks, the following details reflect specific existing features of Pancake:

- Expressions used as addresses in load and store operations are simply checked to have the shape **One**, as opposed to C pointers which identify the type of the data being pointed to.

$$\frac{\Gamma_V; \Gamma_G \vdash addr : \textbf{One}}{\Gamma_V; \Gamma_G \vdash \textbf{Load } sh\ addr : sh}\ \textsc{Lds}$$

$$\frac{\Gamma_V; \Gamma_G \vdash addr : \textbf{One} \quad \Gamma_V; \Gamma_G \vdash exp : sh}{scope; \Gamma_V; \Gamma_G; \Gamma_F \vdash \textbf{Store } addr\ exp \textbf{ ok}}\ \textsc{St}$$

- All variables must be initialised at declaration. This is to prevent both the possibility of use before initialisation and the need to check for this possibility.

$$\frac{\begin{array}{c}\Gamma_V;\Gamma_G \vdash exp : sh \\ scope;\Gamma_V, v : sh;\Gamma_G;\Gamma_F \vdash stmt\ \mathbf{ok}\end{array}}{scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{Dec}\ v\ sh\ exp\ stmt\ \mathbf{ok}}\ \textsc{Dec}$$

$$\frac{\mathbf{empty};\Gamma_G \vdash exp : sh}{\Gamma_G;\Gamma_F \vdash \mathbf{Decl}\ sh\ v\ exp \rightsquigarrow \Gamma_G, v : sh;\Gamma_F}\ \textsc{Glob}$$

- Function calls are specific statements and may not appear in expressions. This is to ensure Pancake's expressions are *effect-free* for verification simplicity, such that evaluating them does not have side effects such as changing variable values.

$$\frac{\begin{array}{c}f : sh, [sh_1 .. sh_n] \in \Gamma_F \\ \Gamma_V;\Gamma_G \vdash exp_1 : sh_1 \quad .. \quad \Gamma_V;\Gamma_G \vdash exp_n : sh_n \\ scope;\Gamma_V, v : sh;\Gamma_G;\Gamma_F \vdash stmt\ \mathbf{ok}\end{array}}{scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{DecCall}\ v\ sh\ f\ [exp_1 .. exp_n]\ stmt\ \mathbf{ok}}\ \textsc{DecCall}$$

$$\frac{\begin{array}{c}f : sh_0, [sh_1 .. sh_n] \in \Gamma_F \\ v : sh_0 \in \Gamma_V \\ \Gamma_V;\Gamma_G \vdash exp_1 : sh_1 \quad .. \quad \Gamma_V;\Gamma_G \vdash exp_n : sh_n\end{array}}{scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{AssignCall}\ v\ handle\ f\ [exp_1 .. exp_n]\ \mathbf{ok}}\ \textsc{AssignCall}$$

- All global variables and functions are added to the context before all function bodies are checked, so all global variables and functions are in scope for all function bodies regardless of declaration order, but the relative scope of global variables between each other does depend on declaration order.

$$\frac{\begin{array}{c}\mathbf{empty};\mathbf{empty} \vdash decl_1 \rightsquigarrow \Gamma_{G2};\Gamma_{F2} \quad .. \quad \Gamma_{Gn};\Gamma_{Fn} \vdash decl_n \rightsquigarrow \Gamma_G;\Gamma_F \\ \Gamma_G;\Gamma_F \vdash decl_1\ \mathbf{ok} \quad .. \quad \Gamma_G;\Gamma_F \vdash decl_n\ \mathbf{ok}\end{array}}{[decl_1 .. decl_n]\ \mathbf{ok}}\ \textsc{Prog}$$

On the other hand, these details betray design decisions of the checking algorithm:

- Shape declarations are treated as the true source of programmer intent. For instance, the size limit for function return shapes is checked at the function declaration and the shapes of return expressions are expected to match the declared return shape.

$$\frac{\begin{array}{c}f \neq \text{“}\mathbf{main}\text{”} \\ \mathbf{size}\,(sh_0) \leq 32\end{array}}{\Gamma \vdash \mathbf{Func}\ sh_0\ f\ \mathbf{false}\ [(v_1, sh_1) .. (v_n, sh_n)]\ stmt \rightsquigarrow \Gamma, \mathbf{Fn}\ f : sh_0, [sh_1 .. sh_n]}\ \textsc{FuncStatic}$$

$$\frac{\Gamma_V;\Gamma_G \vdash exp : sh}{\mathbf{Body}\ sh;\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{Return}\ exp\ \mathbf{ok}}\ \textsc{Return}$$

- The return shape of the current function is stored within the scope while checking function bodies. This allows the checks for return statements and tail calls—function calls whose return value are immediately returned by the function calling them—to reference the current scope and match the two return shapes.

$$\frac{\Gamma_V; \Gamma_G \vdash exp : sh}{\textbf{Body}\ sh; \Gamma_V; \Gamma_G; \Gamma_F \vdash \textbf{Return}\ exp\ \textbf{ok}}\ \textsc{Return}$$

$$\frac{\begin{array}{l} f : sh, [sh_1 .. sh_n] \in \Gamma_F \\ \Gamma_V; \Gamma_G \vdash exp_1 : sh_1 \quad .. \quad \Gamma_V; \Gamma_G \vdash exp_n : sh_n \end{array}}{\textbf{Body}\ sh; \Gamma_V; \Gamma_G; \Gamma_F \vdash \textbf{TailCall}\ f\ [exp_1 .. exp_n]\ \textbf{ok}}\ \textsc{TailCall}$$

5.1.2 Syntax Extension

Shape checking requires that users explicitly declare the shapes of all variables, function arguments and returns. While such shape declarations were already present for global variables, function arguments and certain local variables prior to the project, the declarations of function returns and the general case for local variables had to be extended to take in a shape at the concrete syntax level and store it at the abstract syntax level. Since this phase also included the default shape feature, each shape declaration site had to be made optional, with an omitted shape being filled in with a machine word.

The changes to the concrete syntax were isolated to the parsing stage, which consists of the following steps:

1. *Lexing* the input program into tokens
2. *Parsing* the tokens into an initial tree structure, with tokens grouped into nodes according to the expected sequence for each language component
3. *Converting* the parse tree into the actual abstract syntax structure, which is then passed into the rest of the compiler

The lexer was extended with a token with no concrete syntax equivalent that represents the default shape. The parser was then extended to add shape declarations to local variables and function returns, and make all shape declarations save the new default shape token to the parse tree when omitted. This inserted-token approach sees its benefit in the conversion step, where it can be detected as any other shape would be and mapped to **One**. The alternative approach, to save nothing to the parse tree when the shape declaration is omitted, clashes with existing structures in the parser that do the same — if two elements in a tree node may be omitted but only one of them is, the conversion requires extra steps to distinguish which one was provided.

The abstract syntax updates, consisting solely of additional shape arguments in the structures for local variable and function declarations, were comparatively very simple. However, due to the prevalence of their use throughout the compiler, many definitions referencing the abstract syntax had to be updated. Since the change itself was small and had no impact on the compilation steps or verification, it was fairly simple to update all usage sites, including in proofs.

5.1.3 Checking Implementation

The shape checking and its associated tests were implemented as a direct extension of the prior static checker for Pancake, which was in turn built off the scope checker previously implemented by Thomas Liang. It is invoked after parsing and before the main compilation passes. Any errors that are found at this point halt compilation after being reported, while warnings are reported but do not prevent compilation. There are currently no warnings invoked in the shape checking portion of the checker.

> ***Static Checker.*** The static checker was implemented as prior work to the project, and integrated into the mainline compiler as preliminary work[1]. It added a number of miscellaneous checks to the scope checking of functions and variables, targeting general cases where the compiler may produce output that does not make sense. A notable example is rejecting functions with missing return statements, which can cause execution to run off the end of a function and into the one following if allowed to compile. The full list of cases covered is listed in Appendix B. The work also included streamlining the error and warning reporting machinery, as well as setting up a simple testing environment for the checking step.

The checking algorithm[2] closely follows the steps laid out in the inference rules. The existing context in the static checker was extended to track the shapes of variables and function signatures, which are populated at each declaration point. One notable decision was to combine the tracking of shapes with the existing mechanism for estimating local or shared addresses into one datatype, as this added struct field granularity to the estimation mechanism, which was previously brushed over. This combined datatype then received a number of helper functions to compare them against other shapes, both in this representation and the abstract syntax representation. Several helper functions were also written to standardise error messages associated with shape issues, as well as to handle error reporting within a sequence of checks to report the specific element prompting the error, such as the arguments of a function.

The testing for the checking algorithm was similarly an extension of the existing test suite for the static checker[3]. These tests have a *unit-test* style; for instance, the shape mismatch checks are tested with different passing or failing combinations of shapes in each location where they may occur. The testing is necessary since the static checker is orthogonal to the compiler correctness—it merely bars certain programs from entering the main compiler sequence and makes no transformations—and contributes to the overall regression testing suite.

---

[1] The pull request for the static checker work can be found at `https://github.com/CakeML/cakeml/pull/1138`.

[2] The implementation is located in the `pancake/panStaticScript.sml` file.

[3] These tests are located in the `pancake/static_checker/panStaticExamplesScript.sml` file.

5.1.4 Compatibility and Pull Request

With the checking, default feature and requisite language extensions implemented, only miscellaneous compatibility steps remained before being able to open a pull request for the first phase of work.

First, various pieces of documentation in the repository were to be updated, such as the `NEWS.md` file that records user-facing changes to the language, and the list of checks recorded at the top of the static checker file.

The second requirement deals with how the compiler goes through a translation step into CakeML while being built. This allows the compiler to compile *itself* and aso benefit from the correctness proofs on the CakeML side. This means that the translation steps must be made aware of any new functions added to the implementation of the compiler, and only supports features of HOL4 that can be expressed in CakeML. Unfortunately, because these updates are typically only addressed at the end of any given body of work updating the compiler, implementations that rely on HOL4 library functions that aren't supported by the translator typically are not identified until this stage, and with some obfuscation by error messages that are specific to the translator's inner workings. This was true for the shape checker, which used the `EVERY2` function to walk through the constituent list of shapes for two structs to check if every pair matches. After the issue was finally identified, the implementation was updated to use mutually recursive functions to do this check instead, which the translator could easily handle.

The pull request itself was unfortunately delayed due to having the global variables work as a dependency, which overlapped with this first stage. The work existed as part of the private version of the repository, which meant that the shape checking work also could not be made public until the global variables work was approved for public release into the main repository. This dependency also created logistical issues with collaboration during the implementation of this stage. Thankfully, the work was approved not long after the phase was completed, allowing the pull request to be made, checked against the regression tests and successfully merged in[4].

5.2 Named Structs and Fields

5.2.1 Syntax Extension

Where the syntax updates required for the first stage were largely minimal, the updates for the named struct feature were more invasive. Both the concrete and abstract syntax required an additional top-level declaration structure for declaring the struct name and fields, a new kind of shape distinguished from the existing raw structs, and new kinds of expressions for the named versions of struct constants and field accesses.

[4]The pull request for the first phase of the project can be found at https://github.com/CakeML/cakeml/pull/1221.

Concrete Syntax

The chosen concrete syntax for these constructs is as follows:

```
1  struct my_struct {
2      1 single,
3      2 double,
4      3 triple
5  }
6  var my_struct s = my_struct <
7      single = 4,
8      double = <5, 6>,
9      triple = <1, 2, 3>
10 >;
11 var 2 d = s.double;
```

Listing 5.1: Minimal example of named struct syntax.

Here, a struct with fields of varying size is given the name `my_struct`, with field names `single`, `double` and `triple`. The shapes greater than `1` are existing syntax shorthand for raw structs containing that many words. The global variable `s` is initialised with a struct constant, while the other, `d`, is initialised with only the `double` field of this struct, which is accessed by name.

Notice that the struct constant syntax includes the names of both the struct and the fields. This was done so that the eventual shape checking extension for named structs can easily tell what the intended struct name is, and what fields are expected for that name. Otherwise, it has to determine the struct name based on the fields present before it can be checked to match the declared shape. In this scenario, the checker would have to guess at the intended named struct, since the shape isn't determined by the length of the struct as it is with raw structs — the process becomes akin to inference! Taking inspiration from Rust's syntax for structs, we instead bake the name of the struct into the expression. This also avoids the need for field names to be unique across the entire program.

Parsing Stage

Parsing this new syntax came with a couple of puzzles.

Distinguishing named shapes and defaulted identifiers. When faced with an identifier at a shape declaration location, the existing parsing style could not distinguish between the case where that identifier referred to a struct name and the case where it referred to the function or variable being declared where the shape has been omitted. Mixing up these cases could result in parsing failure despite a perfectly acceptable input program.

To combat this, the parser was refactored to pair together shape declarations with the identifier it describes. It then attempts to parse the *entire* pair during appropriate declaration points and, if tokens for both are not found, reattempts parsing just the identifier and saves the default shape token. This clearly establishes the order in which the cases are considered and preserves the existing method for converting parsed shapes.

Nested field access by index and name. The parser included field index accesses among the expressions with highest precedence — the "order of operations" prioritisation of a given structure within a language, where only higher precedence operations may be considered sub-expressions of lower-precedence ones. This set of expressions, which included things such as identifier names and numeric literals, was already rather unwieldy in the number of options it included. As such, it was desirable for the named field access to not be added to these expressions. However, doing so would result in the parser not allowing name access as a sub-expression of index access; the expression `x.a.0` would not be accepted without additional parentheses as `(x.a).0` while `x.0.a` would be parsed without issue.

So, this set of highest precedence expressions was refactored to remove index accesses, which was given its own precedence level below the highest that it shares only with named field access. This refactoring was somewhat complex to propagate into the conversion step of the parsing, since the parsed groupings are much more complicated to reason about than the abstract syntax. Once it was done, however, the refactor also addressed the premature rejection of indexing into numeric literals — `1.0` was previously considered a parsing error as opposed to a shape error.

Abstract Syntax

On the abstract syntax side, the new declarations and expressions were straightforwardly modelled from existing structures. However, the question remained of how to represent the named struct shape.

The two possible options were for the field information to be stored within every instance of the shape, or instead to be stored in a central context and looked up when needed, with the shape only carrying the name. The former allows named struct shapes to be more self-contained, but may result storing in redundant information in multiple places. It also requires the shapes to be populated during the conversion step in the parsing stage, which, as mentioned, is rather difficult to work with and would involve creating a context of all the declared structs anyway. This made the latter option a tentatively more viable choice, and ended up being carried through to the final implementation version.

There were also number of candidate HOL4 data structures to use for this context, from high level mappings akin to dictionaries to less abstracted but simpler structures such as lists of pairs. This list of pairs representation, known as an *association list*, was chosen for the flexibility in being able to use the simple abstractions available from the HOL4 library and the underlying list-of-pairs structure without abstractions. The order of struct declarations is then baked into the ordering of the list.

That being said, the choice to use this central context did not come without difficulties. One of the requirements for functions in HOL4 is *proven termination* for recursive functions — that is, a recursive function should be guaranteed to eventually terminate when called with arbitrary arguments. This typically involves identifying an argument that maps to some number such that the number decreases with every recursive call, standing in for the number of calls remaining. In most cases, HOL4 is able to identify this argument on its own and prove termination

automatically.

However, with the introduction of named struct shapes, functions that recurse over a shape structure can no longer use the remaining sub-shapes as this measurement, as the field shapes are pulled from the struct context and not the named struct shape argument itself. In fact, such functions are no longer guaranteed to terminate — if the field of a named struct has the struct itself as its shape, the function will recurse infinitely. This meant there were two tasks at hand: ensure such functions do terminate, and manually prove their termination.

In order to combat this issue, a new requirement is established for shape safety under named structs: named structs can only use other structs for their fields if those structs have already been declared. This mirrors the scoping treatment of global variables, and all global variables and functions should still be allowed to see all named structs. This requirement meant that recursive calls could truncate the current context, such that it only contained the structs that existed at the time of declarations.

With this, the measurement for termination could consider both the size of remaining shape structure and the length of the struct context — eventually we will run out of shapes to recurse on, whether from the input shape structure *or* the context. Since the issue of termination proofs lay squarely within the realm of verification, plenty of assistance was solicited from supervisors and similar in order to complete the actual manual proof.

5.2.2 Semantics Extension

As part of propagating the abstract syntax changes, the HOL4 formalisation of Pancake's language semantics is also updated to include the named structs feature[5].

The semantics describes the evaluation of expressions down to either word or raw struct values, while the top-level declarations and function body statements are evaluated as changes in the program state. Each of the abstract syntax additions was fully integrated into this description, with a new kind of value for named structs. It now also describes the process of building up a struct context as struct name declarations are encountered, feeding this context to the global variable and function declarations for use, and finally to the function bodies, which pull from this context as needed to ensure struct name constants and field accesses adhere to their declared shapes and names.

This integration reflects the decision to have named structs be unrelated to their raw counterparts from the perspective of the user. Namely, struct names are not an *alias* like C's `typedef` feature, as this does not enforce the programmer intent that has been provided. Preserving this intent into the abstract syntax and semantics allows the shape checking to do more powerful analysis. It also means there are no *subtyping* relationships between the two shapes, or indeed any *polymorphism* of any kind. In this way, whenever a raw struct is expected in a program, a named struct with the same underlying structure will never be accepted by the shape checking, and vice versa.

[5]The semantics is located in the `pancake/semantics/panSemScript.sml` file.

The Case Against Type Aliasing and Polymorphism

The importance of this decision is clearer not in the comparison between a named struct and its raw counterpart, but in comparing two name structs that share the same raw equivalent. Consider the following named structs, which both have `{1,1,1}` or `3` as their raw equivalent, and an example operation on `3`:

```
1  struct my_point_3d {
2      1 x,
3      1 y,
4      1 z
5  }
6  struct my_vector_3d {
7      1 x,
8      1 y,
9      1 z
10 }
11 fun 3 incr_3(3 a) {
12     return <a.0 + 1, a.1 + 1, a.2 + 1>;
13 }
```

Listing 5.2: Example of equivalently structured but distinct named structs.

We now examine different relationships these three shapes can have other than being entirely unrelated.

Type Aliasing. The simplest approach is to declare both `my_point_3d` and `my_vector_3d` as being equivalent shapes to `3`, just under different names. This means that these two named structs are also equal *to each other*, so the `incr_3` function can receive and be used with any of the three, as can any function or variable expecting one of these shapes.

This approach leaves the feature as no more than a convenient but thin abstraction over the raw struct. It completely abandons the goal of imbuing more programmer intention in the struct syntax since different names may be used interchangeably, and was deemed not suitable for the project.

Subtype Polymorphism. A natural approach that distinguishes between the two named structs is to think of them as specific subsets of the raw struct; that is, `my_point_3d` and `my_vector_3d` are *subtypes* of `3`, so they may be used in place of it. However, a programmer is likely to look at `incr_3`, expect to be able to pass it, say, a `my_point_3d`, and then expect to be able to *assign the result* to a `my_point_3d` variable. In subtype polymorphism, this is not permitted, as the assignment is attempting to coerce the more general `3` result into the more specific `my_point_3d`, even though it originated as one.

Ad-hoc Polymorphism. Considering this, the next approach may be to treat `my_point_3d` and `my_vector_3d` as *instances* of `3`, similar to a *type class*. This means the `3` shape acts like an interface, which may be instantiated with any named struct that fulfils this interface; note that the raw `3` shape is then also an instance. In other words, if a `my_point_3d` is given to `incr_3`, the function is now about `my_point_3d` rather than `3` for that usage, allowing the expected return value assignment behaviour. However, it also raises the question of whether the two

instantiations of `3` are required to be the same, and if not, how the programmer may signal that they intend them to be the same.

Explicit Coercion. For either of the above forms of polymorphism, there is the possibility of supporting explicit coercion from raw structs to named ones, such that a programmer can specify that a `3` should be converted to, for example, a `my_point_3d`. This would mirror the implicit coercion from `my_point_3d` to `3` provided by the polymorphism and likely use the struct name as an operator.

These polymorphism approaches add complexity for both the programmer and the compiler implementation. Not only do they go against Pancake's simple semantics design, they are not obviously useful in the systems-level code Pancake is intended for. As such, the named struct feature did not incorporate any of these features.

Propagating Semantics Extensions

Pancake's language semantics is only used in the compiler proofs and related verification tooling, *not* the implementation. Although it was updated with the named structs feature, the propagation of these changes in existing proofs was declared out of the project scope.

5.2.3 Checking Extension

Each of the components of the shape checking feature were updated to account for the named struct feature. This included the inference rules, implementation and testing.

The additional shape checks introduced are:

- Shape names must be defined and in-scope
- Field accesses for named structs must use a name that exists within the struct's shape
- Concrete struct values must include every field exactly once
- Values provided to struct fields within concrete struct values must match their declared shape

The first condition stems from the scoping requirement from Section 5.2.2. Note that the check for accessing named fields and that for raw index fields in Section 5.1.1 prevent using one access method for the other, maintaining their lack of relationship established in Section 5.2.2.

In other to fulfil these, the judgements are updated as in Table 5.2. The separated contexts have been combined for conciseness. Note the additional context entries for struct names and the new judgement for well-scoped shapes. The full rules for the updated checking algorithm in the second phase are listed in Appendix C.

Table 5.2: The updated shape checking judgements for named structs.

Judgement	**Reading**
$\Gamma \vdash sh$ **ok**	sh is wellformed in context Γ
$\Gamma \vdash exp : sh$	exp has shape sh in context Γ
$scope; \Gamma \vdash stmt$ **ok**	$stmt$ shape checks in $scope$, context Γ
$\Gamma \vdash decl \overset{\text{name}}{\rightsquigarrow} \Gamma'$	$decl$ adds a struct name to context Γ to produce the updated context Γ'
$\Gamma \vdash decl \rightsquigarrow \Gamma'$	$decl$ adds a global variable or function to context Γ produce the updated context Γ'
$\Gamma \vdash decl$ **ok**	$decl$ shape checks in context Γ
$program$ **ok**	$program$ shape checks

The checking implementation is extended accordingly. The struct context for this implementation also uses association lists to take advantage of the flexibility described in Section 5.2.2, as well as existing functions for the abstract syntax, such as the word-size calculation for shapes. Using the higher level mapping structures present in the rest of the context would require unnecessarily converting between the two representations. Error messages are also improved.

Upon updating the test suite, the limits of the unit-testing style began to show. The testing file reached approximately 5500 lines by the end of these updates, with the tests being largely simple but very tedious. For instance, looking at shape mismatches alone, there were 7 sites targeted for possible shape mismatches and 8 permutations of incorrect pairings at each site. This already elides some varieties of function call and, with success cases, produces upwards of 45 test cases for mismatches alone. There is currently no tooling support for test maintenance.

5.2.4 Compilation Pass

The named struct feature introduces new syntax constructs with new behaviour, and these need to be compiled. Recall from Figure 2.1 that the main compilation sequence consists of a number of passes which transform an input program along the compiler's *intermediate representation* languages. The passes that apply to the Pancake abstract syntax are a simplifier that performs some syntax-shuffling, a pass that compiles global variables into user memory accesses and one that transitions the program to `CrepLang`, a flattened version of Pancake without any structs.

In order to leverage the existing handling of raw structs in the translation to `CrepLang`, a separate compilation pass was added to compile named structs into their raw equivalents. This was placed between the simplification and global variable passes to minimise the changes propagated into the global variable pass. This means that all named struct features are not expected to be present within either of the passes that follow it. Noticing that nonsense cases, such as variables being out of scope or features that should have already been compiled away, are handled silently in these later passes, their treatment of named struct features are implemented similarly similarly. The soundness of these silent failures is guarded in verification by the relevant components in the compiler correctness verification, and in practice by the static checking.

Note that the shape checking extension for named structs does not impose an order on the fields in a named struct constant. This means the compilation should appropriately handle any field order within constants without causing the fields to become misaligned from other usages of that same named struct. In order to achieve this, the compilation pass references the declaration order of the fields, which is preserved in the context via a nested association list between the fields and their shapes. It then reconstructs the struct using the field order, with the expressions safely rearranged thanks to Pancake's effect-free expressions mentioned in Section 5.1.1.

Since this new pass extends the main compiler sequence, the compiler correctness proofs must be updated, which was also declared out of the project scope. Unfortunately, this limited confidence in the correctness of the pass implementation to manually tested sample input. To this end, a local version of the compiler with named struct support that side-steps the verification was built. This binary was used in manual testing and demonstrations.

5.2.5 Compatibility

As with the previous project phase, the documentation and translation steps needed to be updated with the new changes. The more invasive changes and now-broken compiler correctness meant that several proofs contained within the translation had to be side-stepped and delegated to future work. There was also some difficulty with the 32-bit specific translation updates, where slight differences from the 64-bit version required explicitly making the translator aware of a HOL4 library function, `OPT_MMAP`. This takes in a function that takes a single argument and optionally returns a value, and applies it on a provided list of arguments, optionally returning a list of outputs. Since this is a fairly common operation in functional programming and the compiler, it was expected to have already been translated elsewhere.

Due to the outstanding proof obligations, no pull request has yet been made for this phase[6]. However, the simplicity of the passes and data structures involved suggests it is feasible.

[6]The progress is visible at `https://github.com/halogentlepersuasion/cakeml/tree/pan_shape_structs`.

Chapter 6

Results

6.1 Solution Output

The three steps of the proposed solution strategy were implemented, summarised in Table 6.1.

Table 6.1: The outputs of the implementation phases.

Checking/Default Shape Phase	**Named Struct Phase**
Inference rules	Syntax extensions
Syntax extensions	Semantics extensions
Implementation	Inference rule extensions
Test suite	Checking & test suite extensions
Translation extension	Compilation pass
Merged pull request	Translation extension

To demonstrate the user-facing effect of the solution, we revisit the partially updated linked list program from Section 2.3.2:

```
1  fun main() {
2      // {1,1,1}: head ptr, tail ptr, size
3      var llist = <@base, @base, 0>;
4      // ...
5      llist = inc_sz(llist);
6      // ...
7      return 1;
8  }
9
10 // increase linked list size by 1
11 fun inc_sz({1,1} ll) {
12     var size = ll.1 + 1;
13     return <ll.0, size>;
14 }
```

Listing 6.1: Extended linked list representation with errors.

6.1.1 Checking/Default Demonstration

We begin with the code as it appears in Listing 6.1. Recall that the `inc_sz` function is currently being called with an incorrect argument shape and its return value is being used as an incorrect assignment, on line 5. It also accesses the wrong field on line 12 and returns a new list value with one field missing in line 13.

Let us use the shape checker to guide the repair of this program. Placing this program into a file `llist.pk` and attempting to compile it as-is by specifying the Pancake flags and input and output files, we receive the following error message:

```
$ cake --pancake < llist.pk > llist.S
### ERROR: shape error
AT (UNKNOWN 7:12): expression to initialise local variable llist
has shape {1,1,1} instead of declared shape 1 in function main
```

From this message, we can see that the compiler was not able to shape check the initialisation of the variable `llist` because we did not add a shape annotation for it. This led the compiler to assume our intended shape was `1`, even though we know that we did intend to be a `{1,1,1}`. We can rectify this by adding such an annotation:

```
1  fun main() {
2      // {1,1,1}: head ptr, tail ptr, size
3      /* previously var llist = <@base, @base, 0>; */
4      var {1,1,1} llist = <@base, @base, 0>;
5      // ...
6      llist = inc_sz(llist);
7      // ...
8      return 1;
9  }
10
11 // increase linked list size by 1
12 fun inc_sz({1,1} ll) {
13     var size = ll.1 + 1;
14     return <ll.0, size>;
15 }
```

Listing 6.2: Linked list program with variable shape declaration.

After which the shape checker provides the new error:

```
$ cake --pancake < llist.pk > llist.S
### ERROR: shape error
AT (6:4 6:22): value for argument ll given to function inc_sz has
shape {1,1,1} instead of declared shape {1,1} in function main
```

Now we notice that this mismatch highlights the out-of-date argument shape of the `inc_sz` function, and amend it to the new representation, `{1,1,1}`:

```
1  fun main() {
2      // {1,1,1}: head ptr, tail ptr, size
3      var {1,1,1} llist = <@base, @base, 0>;
4      // ...
5      llist = inc_sz(llist);
6      // ...
7      return 1;
8  }
9
10 // increase linked list size by 1
11 /* previously fun inc_sz({1,1} ll) { */
12 fun inc_sz({1,1,1} ll) {
13     var size = ll.1 + 1;
14     return <ll.0, size>;
15 }
```

Listing 6.3: Linked list program with updated argument shape.

The next error points out the lack of shape declaration for the `inc_sz` return value as it did for `llist`, causing the assumed shape to cause a mismatch when assigning:

```
$ cake --pancake < llist.pk > llist.S
### ERROR: shape error
AT (5:4 5:22): result of function call inc_sz assigned to local
variable llist has shape 1 instead of declared shape {1,1,1} in
function main
```

We can again update the signature for `inc_sz`, this time to include the missing shape:

```
1  fun main() {
2      // {1,1,1}: head ptr, tail ptr, size
3      var {1,1,1} llist = <@base, @base, 0>;
4      // ...
5      llist = inc_sz(llist);
6      // ...
7      return 1;
8  }
9
10 // increase linked list size by 1
11 /* previously fun inc_sz({1,1,1} ll) { */
12 fun {1,1,1} inc_sz({1,1,1} ll) {
13     var size = ll.1 + 1;
14     return <ll.0, size>;
15 }
```

Listing 6.4: Linked list program with updated return shape.

With all mismatches between the signature and usage of `inc_sz` fixed, the shape checker turns our attention to the function body, where the original return value does not match the updated return shape:

```
$ cake --pancake < llist.pk > llist.S
### ERROR: shape error
AT (14:11 14:19): expression to return has shape {1,1} instead
of declared shape {1,1,1} in function inc_sz
```

Here, we recall that the new field in the list representation holds the tail pointer, which the function should not change, at the index 1, and add this to our return value:

```
1  fun main() {
2      // {1,1,1}: head ptr, tail ptr, size
3      var {1,1,1} llist = <@base, @base, 0>;
4      // ...
5      llist = inc_sz(llist);
6      // ...
7      return 1;
8  }
9
10 // increase linked list size by 1
11 fun {1,1,1} inc_sz({1,1,1} ll) {
12     var size = ll.1 + 1;
13     /* previously return <ll.0, size>; */
14     return <ll.0, ll.1, size>;
15 }
```

Listing 6.5: Linked list program with updated return value.

With this, the shape checker is finally happy and the program compiles:

```
$ cake --pancake < llist.pk > llist.S
$
```

Note that we did not need to provide a shape declaration for the `size` local variable in `inc_sz`, since the default shape matched its intended shape. Also note, however, that the value of the new size is still incorrect despite successfully shape checking. It is still calculated via accessing the 1 index, but the shape checker cannot distinguish the intention between different raw struct fields that have the same shape.

6.1.2 Named Struct Demonstration

While we could just fix the `size` variable with a change of its index, this is not a scalable fix, and would require further updates in the future with any changes to the list representation. Instead, we define a named struct representation `llist_t` for the linked list, with the intention of replacing the raw version, such that we can signal our intent for each of the fields:

```
1  /* new declaration */
2  struct llist_t {
3      1 head,
4      1 tail,
5      1 size
6  }
7
8  fun main() {
9      // {1,1,1}: head ptr, tail ptr, size
10     var {1,1,1} llist = <@base, @base, 0>;
11     // ...
12     llist = inc_sz(llist);
13     // ...
14     return 1;
15 }
16
17 // increase linked list size by 1
18 fun {1,1,1} inc_sz({1,1,1} ll) {
19     var size = ll.1 + 1;
20     return <ll.0, ll.1, size>;
21 }
```

Listing 6.6: Linked list program with new named struct declaration.

Since we have only added this new declaration and left the prior lines unchanged, there is no need to run the shape checker immediately, though doing so as a sanity-check indeed raises no errors.

We now want to propagate this new shape in the `main` and `inc_sz` functions. We begin by replacing each instance of the `{1,1,1}` shape declaration with `llist_t`:

```
1  struct llist_t {
2      1 head,
3      1 tail,
4      1 size
5  }
6
7  fun main() {
8      /* previously var {1,1,1} llist = <@base, @base, 0>; */
9      var llist_t llist = <@base, @base, 0>;
10     // ...
11     llist = inc_sz(llist);
12     // ...
13     return 1;
14 }
```

```
15
16 // increase linked list size by 1
17 /* previously fun {1,1,1} inc_sz({1,1,1} ll) { */
18 fun llist_t inc_sz(llist_t ll) {
19     var size = ll.1 + 1;
20     return <ll.0, ll.1, size>;
21 }
```

Listing 6.7: Linked list program with named structs in shape declarations.

In this partial state, the shape checker points out that the `llist` variable now needs to be initialised with an `llist_t`, not the existing `{1,1,1}`:

```
$ cake --pancake < llist.pk > llist.S
### ERROR: shape error
AT (9:8 13:12): expression to initialise local variable llist has
shape {1,1,1} instead of declared shape llist_t in function main
```

We update this constant to specify the named struct we wish to use, and to map each provided value to its field name:

```
1  struct llist_t {
2      1 head,
3      1 tail,
4      1 size
5  }
6
7  fun main() {
8      /* previously var llist_t llist = <@base, @base, 0>; */
9      var llist_t llist = llist_t <
10         head = @base, tail = @base, size = 0
11     >;
12     // ...
13     llist = inc_sz(llist);
14     // ...
15     return 1;
16 }
17
18 // increase linked list size by 1
19 fun llist_t inc_sz(llist_t ll) {
20     var size = ll.1 + 1;
21     return <ll.0, ll.1, size>;
22 }
```

Listing 6.8: Linked list program with named struct constant initialisation.

The next issue is within `inc_sz`, where the attempted size access into the argument still uses index 1. Not only does the 1 index no longer hold the size field, with the named struct representation, it now no longer exists:

```
$ cake --pancake < llist.pk > llist.S
### ERROR: shape error
AT (UNKNOWN 21:23): expression shape llist_t has no field
at index 1 in function inc_sz
```

This error would also apply to the 0 and 1 index accesses in line 21, so we update all field accesses in `inc_sz` to use the matching field name:

```
1  struct llist_t {
2      1 head,
3      1 tail,
4      1 size
5  }
6
7  fun main() {
8      var llist_t llist = llist_t <
9          head = @base, tail = @base, size = 0
10     >;
11     // ...
12     llist = inc_sz(llist);
13     // ...
14     return 1;
15 }
16
17 // increase linked list size by 1
18 fun llist_t inc_sz(llist_t ll) {
19     /* previously var size = ll.1 + 1; */
20     var size = ll.size + 1;
21     /* previously return <ll.0, ll.1, size>; */
22     return <ll.head, ll.tail, size>;
23 }
```

Listing 6.9: Linked list program with struct field access by name.

Our final issue is similar to that in Listing 6.7, namely that our return value in `inv_sz` is of the old {1,1,1} shape and not `llist_t`:

```
$ cake --pancake < llist.pk > llist.S
### ERROR: shape error
AT (22:11 22:29): expression to return has shape {1,1,1}
instead of declared shape llist_t in function inc_sz
```

As such, we update the returned constant in a similar manner:

```
1  struct llist_t {
2      1 head,
3      1 tail,
4      1 size
5  }
6
7  fun main() {
8      var llist_t llist = llist_t <
9          head = @base, tail = @base, size = 0
10     >;
11     // ...
12     llist = inc_sz(llist);
13     // ...
14     return 1;
```

```
15 }
16
17 // increase linked list size by 1
18 fun llist_t inc_sz(llist_t ll) {
19     var size = ll.size + 1;
20     /* previously return <ll.head, ll.tail, size>; */
21     return llist_t <
22         head = ll.head, tail = ll.tail, size = size
23     >;
24 }
```

Listing 6.10: Linked list program with named struct constant return.

With this, the shape checker is happy once again, the `inc_sz` function code matches the intention and the representation is more robust to future updates:

```
$ cake --pancake < llist.pk > llist.S
$
```

6.2 Language Goals

Recall Pancake's language design goals of ease of formal verification, a minimal trusted computing base, and usability by systems programmers; as well as the established usability sub-qualities of understandability, usefulness and practicality. We examine how well the implemented solution progresses and conserves these goals.

The shape checking feature:

- allows the compiler to support the user in detecting and diagnosing shape errors. It does so by notifying the user of the presence and location of these errors at compile time instead of leaving the issue to be discovered at run-time. This *improves practicality*.
- requires increased explicit shape declarations where there was previously none to provide enough information for the checking algorithm. This can serve as implicit documentation for the shapes of identifiers in a program for the programmer and others who read the code. The compiler then reports any shape errors by explaining the details of the mismatched shapes that it found. This *improves understandability*.

The default shape declaration feature:

- reduces the required shape declarations necessary for shape checking by allowing omitted shapes to be assumed as machine words. This was informed by the observation that the majority of existing Pancake program use only words, and any that use raw structs are expected to explore moving to named structs. As such, it also maximises the backwards compatibility of the new features. This *preserves practicality*.

The named struct feature:

- abstracts the underlying shape details of a struct behind custom names. This better communicates programmer intent, provides robustness to changes and avoids replicating the raw struct syntax at every use site. This *improves understandability* and *practicality*.
- integrates the named structs behaviour into the language semantics, treating the feature as more than just type aliases. This strengthens the shape checker's ability to utilise programmer intent to distinguish named structs, which is carried into verification efforts that interface with the language semantics. This reduces the need to remember and reason about raw struct indexes. This *improves practicality* and *ease of formal verification*.
- avoids using polymorphism relationships such as subtyping in the treatment of named structs during shape checking. It instead opts for requiring shapes to be exact matches. This allows the relationship between named and raw structs with equivalent underlying structures to remain simple. This *preserves understandability* and *ease of formal verification*.
- expresses named struct expressions one-to-one with raw structs and compiles them according to this relationship. This means there is no performance overhead at run-time of named structs compared to raw structs. This *preserves usefulness*.
- delegated outstanding proof obligations to after the project. This is particularly notable for integrating the new compiler pass into the end-to-end correctness proofs. Due to the nature of the feature, it is expected that these proof obligations are feasible to fulfil. Once this is complete, this will preserve the minimal trusted computing base.

Overall, we see that, based on the design decisions behind it, the solution exhibits strong improvements and preservations of Pancake's design goals. In situations where preservation has not yet been maintained, doing so has been identified as future work and deemed feasible.

6.3 User Feedback

Evaluation of the solution is not complete without input from the target user-base that will make use of the new features. For this, we examine feedback provided by a volunteer within Trustworthy Systems with substantial Pancake development history in actual device drivers (relative to the language's age). The volunteer was given the local unverified build of the compiler containing named struct support. They chose a few existing device drivers and updated them with named structs as they saw fit. The shape checking and named struct features were then discussed within the context of use in realistic Pancake programs.

Many Pancake device drivers exist as translations of drivers implemented in C. Within those C drivers, interactions with structs exists in two major forms: directly via global variables (to track state or implement data structure) or indirectly via pointers (to provide interfaces). The two historic translation methods for the former usage exist:

1. Direct user memory access, where raw structs are saved at predefined addresses and accessing specific fields involves loading from those addresses with an offset matching the field position. This method made ample use of the C preprocessor to provide macros for operations such as field access or assignment. This method was common before the

introduction of the global variables feature.

2. Separated global variables for struct fields, where the individual fields of a C struct are saved as separate global variables which can then be used by name. These field variables were often single words due to the nature of the C structs. Multiple instances of a struct all require such variables to be defined for each instance. This method came after the introduction of the global variables feature.

The pointer-based usage, by necessity, only uses the first user memory technique.

For example, the following C struct represents the buffer queues used for transporting data between, for instance, device drivers and other components, and has two global instances:

```
1  typedef struct net_queue_handle {
2      /* available buffers */
3      net_queue_t *free;
4      /* filled buffers */
5      net_queue_t *active;
6      /* capacity of the queues */
7      uint64_t capacity;
8  } net_queue_handle_t;
9
10 net_queue_handle_t rx_queue;
11 net_queue_handle_t tx_queue;
```

Listing 6.11: C struct variables for network queues.

The direct memory access representation involves defining many preprocessor macros to provide a more C-like interface to struct interactions and avoid manually transcribing address calculations. Such a translation of `net_queue_handle_t` may look like so (where the `@biw` literal represents the number of bytes in a machine word):

```
1  #define NETFREE_OFFSET      0
2  #define NETACTIVE_OFFSET    @biw
3  #define NETCAPAC_OFFSET     @biw * 2
4
5  #define RXQUEUE_ADDR        @base
6  #define TXQUEUE_ADDR        @base + @biw * 3
7
8  #define get_rxfree(free)                                    \
9      var free = lds 1 (RXQUEUE_ADDR + NETFREE_OFFSET);       \
10
11 #define set_rxfree(free)                                    \
12     st RXQUEUE_ADDR + NETFREE_OFFSET, free;                 \
13
14 #define get_txfree(free)                                    \
15     var free = lds 1 (TXQUEUE_ADDR + NETFREE_OFFSET);       \
16
17 #define set_txfree(free)                                    \
18     st TXQUEUE_ADDR + NETFREE_OFFSET, free;                 \
19
20 // ...
```

Listing 6.12: Pancake memory access for network queues.

On the other hand, the more "modern" approach that takes advantage of the global variables feature and separates the fields would look like so:

```
1 var 1 rx_queue_free     = 0;
2 var 1 rx_queue_active   = 0;
3 var 1 rx_queue_capacity = 0;
4
5 var 1 tx_queue_free     = 0;
6 var 1 tx_queue_active   = 0;
7 var 1 tx_queue_capacity = 0;
```

Listing 6.13: Pancake global words for network queues.

The named struct feature enables a third possible translation technique for global C structs: global variables with named struct shapes. The `net_queue_handle` translation would then look like so:

```
1  struct net_queue_handle_t {
2    1 free,
3    1 active,
4    1 capacityt
5  }
6
7  var net_queue_handle_t rx_queue = net_queue_handle_t <
8      free = 0, active = 0, capacity = 0
9  >;
10 var net_queue_handle_t tx_queue = net_queue_handle_t <
11     free = 0, active = 0, capacity = 0
12 >;
```

Listing 6.14: Pancake global named structs for network queues.

Examining it under the Pancake designs goals as in Section 6.2, the technique of combining global variables and named struct shapes:

- is a closer translation of the C source, is more concise and better captures the intent behind the original structs. The link between the different fields of a given instance are maintained outside of variable names, and a C-familiar reader is more likely to understand what the struct is for. This *improves understandability*.
- removes the need for recalling small details, such as addresses, offsets or long yet subtly distinct variable names. For instance, the length of the names `rx_queue_capacity` and `tx_queue_capacity` in Listing 6.13 may obfuscate accidental uses of the opposite instance. Being able to avoid such issues *improves practicality*.
- revives the demand for struct field assignments. The prior techniques have in-built strategies for getting around this by taking advantage of offsets and disconnected variables. To get around this, the global named struct technique can define functions reminiscent of the macros of Listing 6.12 or the `inc_sz` function of Section 6.1. However, this requires work on the part of the programmer to set up for their specific structs. So, this practicality issue cannot be considered improved upon until field assignment is supported.
- revives the interest in field accesses using addreses, akin to the C `->` operator. With an increase in struct syntax usage, the pattern of loading an entire struct by address just to access one field becomes more common. Since this is so common within C code, there is

some demand for syntax that skips this step to avoid loading the entire struct. In practice, the compiler optimises away this load, which means this operator may be implemented as syntax sugar. So, there is still more improvement possible for practicality.
- maps well to verification tools that also support structs. In particular, one avenue of Pancake verification currently undergoing research is automatic translation of Pancake into the Viper *automated theorem prover*. This involves *annotating* a given Pancake program with pre- and post-conditions for each of its functions. Using the global named struct technique, as opposed to the memory access technique previously used in this work, allows those conditions to be expressed in more readable and simpler to verify ways. For example, a function that sets the capacity of the receive queue to 0 may have the post-condition `ensures memory[RXQUEUE_ADDR + NETCAPAC_OFFSET] == 0` before named structs, and `ensures rx_queue.capacity == 0` afterwards. This *progresses ease of formal verification*.

Overall, we see that the named struct feature indeed improves upon and conserves several of the language design goals in the context of real device drivers, but the lack of features such as field assignment holds it back from an ideal usability scenario for structs.

Outside of the judgements on the named struct feature, the volunteer responded positively to the shape checking and its error reporting. In particular, there was some initial trouble with compiler build versions prompted some confusion about the nature of the default shape feature while looking at the shape checker output. However, after moving to the named structs version of the compiler, where shape error messages were updated to be more specific, the confusion was instantly quelled. The volunteer, unprompted, also went through the thought process of considering possible uses of polymorphism or type aliasing, and agreed there were no currently obvious uses for those features in device drivers.

6.4 Limitations and Future Work

Based on the established evaluation, the limitations of the project largely lie in features yet unimplemented that would round out the usability of structs in Pancake in general. In particular, support for struct field assignment and struct field access via addresses. Since these directly reflect user demand, it would be ideal for these features to be added to the language, if not part of this project's body of work, then soon after. The outstanding verification work associated with the named struct feature is similarly a required piece of work before it can be merged into the main compiler. This verification work has already been started by collaborators at the University of Gothenburg and Chalmers University of Technology.

Outside of these identified limitations, a number of possible improvements exist:

- Moving from a checking algorithm to a bidirectional strategy, to replace the default shape feature and possibly allow omission of struct shape declarations where permissible
- Improved location tracking for declarations. This necessitated particularly verbose errors messages in order to provide enough information for the user to find the error within their file.

- Syntactic shorthands that allow fields to be omitted in a named struct constant and initialised with 0 instead of throwing an error about missing fields. This avoids needing to specify a 0 value for every field when initialising named struct variables. However, forgetting to provide a value for a given field is still a possible mistake to make, so such a change would likely be accompanied by a warning that this is occurring.
- Dependency analysis for struct name declarations (and by extension, possibly global variables). The compiler could then use these dependencies to rearrange declarations so that users need not worry about the scoping declaration order. However, this would require cycle detection, and C-familiar programmers are already used to considering declaration order in C programs. As such, whether the benefit of such a feature is worth the effort is questionable.

Chapter 7

Conclusion

This project aimed to address the usability issues associated with the neglect of the Pancake language's shape features, with particular focus on the lack of compiler-checked shape safety and poor error-prone syntax. The implemented solution did so by implementing static shape checking to react to shape errors in incoming programs, a default shape feature to minimise the additional syntax requirements of shape checking and a named struct feature that deviates from the existing raw structs by assigning names to struct shapes and their fields. Even though there is further work to be done, the solution was found to strongly improve the usability issue in reference to the language's key design goals. All in all, the project highlighted that language features should be added because they are useful, not just because they can be added, and features that have been added should be made useful, or else they are useless.

Appendix A: Shape Checking Rules

$\boxed{\Gamma_V; \Gamma_G \vdash exp : sh}$ exp has shape sh in variable context $\Gamma_V; \Gamma_G$

$$\frac{}{\Gamma_V; \Gamma_G \vdash \textbf{Const}\ num : \textbf{One}}\ \textsc{Const}$$

$$\frac{v : sh \in \Gamma_V}{\Gamma_V; \Gamma_G \vdash \textbf{Var Local}\ v : sh}\ \textsc{LocalVar}$$

$$\frac{v : sh \in \Gamma_G}{\Gamma_V; \Gamma_G \vdash \textbf{Var Global}\ v : sh}\ \textsc{GlobalVar}$$

$$\frac{\Gamma_V; \Gamma_G \vdash exp_1 : sh_1 \quad .. \quad \Gamma_V; \Gamma_G \vdash exp_n : sh_n}{\Gamma_V; \Gamma_G \vdash \textbf{Struct}\ [exp_1\ ..\ exp_n] : \textbf{Comb}\ [sh_1\ ..\ sh_n]}\ \textsc{Struct}$$

$$\frac{\Gamma_V; \Gamma_G \vdash exp : \textbf{Comb}\ [sh_1\ ..\ sh_{index}\ ..\ sh_n]}{\Gamma_V; \Gamma_G \vdash \textbf{Field}\ index\ exp : sh_{index}}\ \textsc{Field}$$

$$\frac{\Gamma_V; \Gamma_G \vdash addr : \textbf{One}}{\Gamma_V; \Gamma_G \vdash \textbf{Load}\ sh\ addr : sh}\ \textsc{LdS}$$

$$\frac{\Gamma_V; \Gamma_G \vdash addr : \textbf{One}}{\Gamma_V; \Gamma_G \vdash \textbf{LoadHalf}\ addr : \textbf{One}}\ \textsc{LdH}$$

$$\frac{\Gamma_V; \Gamma_G \vdash addr : \textbf{One}}{\Gamma_V; \Gamma_G \vdash \textbf{LoadByte}\ addr : \textbf{One}}\ \textsc{LdB}$$

$$\frac{\Gamma_V; \Gamma_G \vdash exp_1 : \textbf{One} \quad .. \quad \Gamma_V; \Gamma_G \vdash exp_n : \textbf{One}}{\Gamma_V; \Gamma_G \vdash \textbf{Op}\ binop[exp_1\ ..\ exp_n] : \textbf{One}}\ \textsc{exp}$$

$$\frac{\Gamma_V; \Gamma_G \vdash exp_1 : \textbf{One} \quad .. \quad \Gamma_V; \Gamma_G \vdash exp_n : \textbf{One}}{\Gamma_V; \Gamma_G \vdash \textbf{Panop}\ panop[exp_1\ ..\ exp_n] : \textbf{One}}\ \textsc{Panop}$$

$$\frac{\Gamma_V; \Gamma_G \vdash exp_1 : sh \quad \Gamma_V; \Gamma_G \vdash exp_2 : sh}{\Gamma_V; \Gamma_G \vdash \textbf{Cmp}\ cmp\ exp_1\ exp_2 : \textbf{One}}\ \textsc{Cmp}$$

$$\frac{\Gamma_V; \Gamma_G \vdash exp : \textbf{One}}{\Gamma_V; \Gamma_G \vdash \textbf{Shift}\ shift\ exp\ num : \textbf{One}}\ \textsc{Shift}$$

$$\frac{}{\Gamma_V; \Gamma_G \vdash \textbf{BaseAddr} : \textbf{One}}\ \textsc{Base}$$

$$\frac{}{\Gamma_V;\Gamma_G \vdash \textbf{TopAddr} : \textbf{One}}\ \textsc{Top}$$

$$\frac{}{\Gamma_V;\Gamma_G \vdash \textbf{BytesInWord} : \textbf{One}}\ \textsc{BiW}$$

$\boxed{scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash stmt\ \textbf{ok}}$ $stmt$ shape checks in $scope$, variable and function context $\Gamma_V;\Gamma_G;\Gamma_F$

$$\frac{}{scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash \textbf{Skip}\ \textbf{ok}}\ \textsc{Skip}$$

$$\frac{\Gamma_V;\Gamma_G \vdash exp : sh \quad scope;\Gamma_V, v : sh;\Gamma_G;\Gamma_F \vdash stmt\ \textbf{ok}}{scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash \textbf{Dec}\ v\ sh\ exp\ stmt\ \textbf{ok}}\ \textsc{Dec}$$

$$\frac{f : sh, [sh_1\ ..\ sh_n] \in \Gamma_F \quad \Gamma_V;\Gamma_G \vdash exp_1 : sh_1\ \ ..\ \ \Gamma_V;\Gamma_G \vdash exp_n : sh_n \quad scope;\Gamma_V, v : sh;\Gamma_G;\Gamma_F \vdash stmt\ \textbf{ok}}{scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash \textbf{DecCall}\ v\ sh\ f\,[exp_1\ ..\ exp_n]\ stmt\ \textbf{ok}}\ \textsc{DecCall}$$

$$\frac{v : sh \in \Gamma_V \quad \Gamma_V;\Gamma_G \vdash exp : sh}{scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash \textbf{Assign Local}\ v\ exp\ \textbf{ok}}\ \textsc{AssignLocal}$$

$$\frac{v : sh \in \Gamma_G \quad \Gamma_V;\Gamma_G \vdash exp : sh}{scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash \textbf{Assign Global}\ v\ exp\ \textbf{ok}}\ \textsc{AssignGlobal}$$

$$\frac{f : sh_0, [sh_1\ ..\ sh_n] \in \Gamma_F \quad v : sh_0 \in \Gamma_V \quad \Gamma_V;\Gamma_G \vdash exp_1 : sh_1\ \ ..\ \ \Gamma_V;\Gamma_G \vdash exp_n : sh_n}{scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash \textbf{AssignCall}\ v\ handle\ f\,[exp_1\ ..\ exp_n]\ \textbf{ok}}\ \textsc{AssignCall}$$

$$\frac{\Gamma_V;\Gamma_G \vdash exp : sh}{\textbf{Body}\ sh;\Gamma_V;\Gamma_G;\Gamma_F \vdash \textbf{Return}\ exp\ \textbf{ok}}\ \textsc{Return}$$

$$\frac{f : sh, [sh_1\ ..\ sh_n] \in \Gamma_F \quad \Gamma_V;\Gamma_G \vdash exp_1 : sh_1\ \ ..\ \ \Gamma_V;\Gamma_G \vdash exp_n : sh_n}{\textbf{Body}\ sh;\Gamma_V;\Gamma_G;\Gamma_F \vdash \textbf{TailCall}\ f\,[exp_1\ ..\ exp_n]\ \textbf{ok}}\ \textsc{TailCall}$$

$$\frac{f : sh_0, [sh_1\ ..\ sh_n] \in \Gamma_F \quad \Gamma_V;\Gamma_G \vdash exp_1 : sh_1\ \ ..\ \ \Gamma_V;\Gamma_G \vdash exp_n : sh_n}{scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash \textbf{StandAloneCall}\ handle\ f\,[exp_1\ ..\ exp_n]\ \textbf{ok}}\ \textsc{StandAloneCall}$$

$$\frac{\Gamma_V;\Gamma_G \vdash exp_1 : \textbf{One}\ \ ..\ \ \Gamma_V;\Gamma_G \vdash exp_4 : \textbf{One}}{scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash \textbf{ExtCall}\ f\ exp_1\ exp_2\ exp_3\ exp_4\ \textbf{ok}}\ \textsc{ExtCall}$$

$$\frac{scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash stmt_1\ \textbf{ok} \quad scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash stmt_2\ \textbf{ok}}{scope;\Gamma_V;\Gamma_G;\Gamma_F \vdash \textbf{Seq}\ stmt_1\ stmt_2\ \textbf{ok}}\ \textsc{Seq}$$

$$\frac{\Gamma_V;\Gamma_G \vdash \mathit{exp} : \mathbf{One} \quad \mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathit{stmt}_1\ \mathbf{ok} \quad \mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathit{stmt}_2\ \mathbf{ok}}{\mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{If}\ \mathit{exp}\ \mathit{stmt}_1\ \mathit{stmt}_2\ \mathbf{ok}}\ \textsc{If}$$

$$\frac{\Gamma_V;\Gamma_G \vdash \mathit{exp} : \mathbf{One} \quad \mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathit{stmt}\ \mathbf{ok}}{\mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{While}\ \mathit{exp}\ \mathit{stmt}\ \mathbf{ok}}\ \textsc{While}$$

$$\frac{}{\mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{Break}\ \mathbf{ok}}\ \textsc{Break}$$

$$\frac{}{\mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{Continue}\ \mathbf{ok}}\ \textsc{Continue}$$

$$\frac{\Gamma_V;\Gamma_G \vdash \mathit{exp} : \mathit{sh}}{\mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{Raise}\ \mathit{eid}\ \mathit{exp}\ \mathbf{ok}}\ \textsc{Raise}$$

$$\frac{\Gamma_V;\Gamma_G \vdash \mathit{addr} : \mathbf{One} \quad \Gamma_V;\Gamma_G \vdash \mathit{exp} : \mathit{sh}}{\mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{Store}\ \mathit{addr}\ \mathit{exp}\ \mathbf{ok}}\ \textsc{St}$$

$$\frac{\Gamma_V;\Gamma_G \vdash \mathit{addr} : \mathbf{One} \quad \Gamma_V;\Gamma_G \vdash \mathit{exp} : \mathbf{One}}{\mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{StoreHalf}\ \mathit{addr}\ \mathit{exp}\ \mathbf{ok}}\ \textsc{StH}$$

$$\frac{\Gamma_V;\Gamma_G \vdash \mathit{addr} : \mathbf{One} \quad \Gamma_V;\Gamma_G \vdash \mathit{exp} : \mathbf{One}}{\mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{StoreByte}\ \mathit{addr}\ \mathit{exp}\ \mathbf{ok}}\ \textsc{StB}$$

$$\frac{v : \mathbf{One} \in \Gamma_V \quad \Gamma_V;\Gamma_G \vdash \mathit{addr} : \mathbf{One}}{\mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{ShMemLoad}\ \mathit{opsize}\ \mathbf{Local}\ v\ \mathit{addr}\ \mathbf{ok}}\ \textsc{ShLdLocal}$$

$$\frac{v : \mathbf{One} \in \Gamma_G \quad \Gamma_V;\Gamma_G \vdash \mathit{addr} : \mathbf{One}}{\mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{ShMemLoad}\ \mathit{opsize}\ \mathbf{Global}\ v\ \mathit{addr}\ \mathbf{ok}}\ \textsc{ShLdGlobal}$$

$$\frac{\Gamma_V;\Gamma_G \vdash \mathit{addr} : \mathbf{One} \quad \Gamma_V;\Gamma_G \vdash \mathit{exp} : \mathbf{One}}{\mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{ShMemStore}\ \mathit{opsize}\ \mathit{addr}\ \mathit{exp}\ \mathbf{ok}}\ \textsc{ShSt}$$

$$\frac{}{\mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{Tick}\ \mathbf{ok}}\ \textsc{Tick}$$

$$\frac{}{\mathit{scope};\Gamma_V;\Gamma_G;\Gamma_F \vdash \mathbf{Annot}\ \mathit{string}_1\ \mathit{string}_2\ \mathbf{ok}}\ \textsc{Annot}$$

$\boxed{\Gamma_G; \Gamma_F \vdash \mathit{decl} \rightsquigarrow \Gamma'_G; \Gamma'_F}$ *decl* updates global variable and function context $\Gamma_G; \Gamma_F$ to produce $\Gamma'_G; \Gamma'_F$

$$\frac{\mathit{sh} = \mathbf{One}}{\Gamma \vdash \mathbf{Func}\ \mathit{sh}\ \text{“}\mathbf{main}\text{”}\ \mathbf{false}\ [\,]\ \mathit{stmt} \rightsquigarrow \Gamma,\ \mathbf{Fn}\ \text{“}\mathbf{main}\text{”} : \mathit{sh}, [\,]} \quad \textsc{FuncMain}$$

$$\frac{\mathit{sh}_0 = \mathbf{One} \quad .. \quad \mathit{sh}_n = \mathbf{One}}{\Gamma \vdash \mathbf{Func}\ \mathit{sh}_0\ f\ \mathbf{true}\ [(v_1, \mathit{sh}_1) .. (v_n, \mathit{sh}_n)]\ \mathit{stmt} \rightsquigarrow \Gamma,\ \mathbf{Fn}\, f : \mathit{sh}_0, [\mathit{sh}_1 .. \mathit{sh}_n]} \quad \textsc{FuncExport}$$

$$\frac{\begin{array}{c} f \neq \text{“}\mathbf{main}\text{”} \\ \mathbf{size}\,(\mathit{sh}_0) \leq 32 \end{array}}{\Gamma \vdash \mathbf{Func}\ \mathit{sh}_0\ f\ \mathbf{false}\ [(v_1, \mathit{sh}_1) .. (v_n, \mathit{sh}_n)]\ \mathit{stmt} \rightsquigarrow \Gamma,\ \mathbf{Fn}\, f : \mathit{sh}_0, [\mathit{sh}_1 .. \mathit{sh}_n]} \quad \textsc{FuncStatic}$$

$$\frac{\mathbf{empty}; \Gamma_G \vdash \mathit{exp} : \mathit{sh}}{\Gamma_G; \Gamma_F \vdash \mathbf{Decl}\ \mathit{sh}\ v\ \mathit{exp} \rightsquigarrow \Gamma_G, v : \mathit{sh}; \Gamma_F} \quad \textsc{Glob}$$

$\boxed{\Gamma_G; \Gamma_F \vdash \mathit{decl}\ \mathbf{ok}}$ *decl* shape checks in global variable and function context $\Gamma_G; \Gamma_F$

$$\frac{\mathbf{Body}\ \mathit{sh}; \mathbf{empty}; \Gamma_G; \Gamma_F \vdash \mathit{stmt}\ \mathbf{ok}}{\Gamma_G; \Gamma_F \vdash \mathbf{Func}\ \mathit{sh}\ f\ \mathit{bool}[\mathit{param}_1 .. \mathit{param}_n]\mathit{stmt}\ \mathbf{ok}} \quad \textsc{Function}$$

$$\frac{}{\Gamma_G; \Gamma_F \vdash \mathbf{Decl}\ \mathit{sh}\ v\ \mathit{exp}\ \mathbf{ok}} \quad \textsc{Global}$$

$\boxed{\mathit{program}\ \mathbf{ok}}$ *program* shape checks

$$\frac{\begin{array}{l} \mathbf{empty}; \mathbf{empty} \vdash \mathit{decl}_1 \rightsquigarrow \Gamma_{G2}; \Gamma_{F2} \quad .. \quad \Gamma_{Gn}; \Gamma_{Fn} \vdash \mathit{decl}_n \rightsquigarrow \Gamma_G; \Gamma_F \\ \Gamma_G; \Gamma_F \vdash \mathit{decl}_1\ \mathbf{ok} \quad .. \quad \Gamma_G; \Gamma_F \vdash \mathit{decl}_n\ \mathbf{ok} \end{array}}{[\mathit{decl}_1 .. \mathit{decl}_n]\ \mathbf{ok}} \quad \textsc{Prog}$$

Appendix B: Existing Static Checks

The following is a list of each error and warning covered by the Pancake static checker prior to this project.

Scope checks:

- Errors:
  - Using a function that is undefined or out-of-scope
  - Using a variable that is undefined or out-of-scope
  - Declaring a function with a name that has already been declared in the scope
- Warnings:
  - Declaring a variable with a name that has already been declared in the scope

General checks:

- Errors:
  - Declaring the main function to take in arguments
  - Exporting the main function using the multiple entry points feature
  - Exporting a function with greater than 4 arguments
  - Missing a function exit (return, tail call, etc) in some execution branches of a function
  - Using a loop exit (break, continue) outside of a loop
  - Declaring a function with argument names that are not distinct
  - Giving operator structures an incorrect number of arguments (impossible through parsing, but possible if generated separately)
- Warnings:
  - Placing statements in unreachable locations, namely after a function or loop exit
  - Giving an address to a local memory operation that is not calculated using the user memory base address (since it may not be a local address)
  - Giving an address to a shared memory operation that is calculated using the user memory base address (since it may not be a shared address)

Appendix C: Shape Checking Rules with Named Structs

$\boxed{\Gamma \vdash sh\ \mathbf{ok}}$ shape sh is wellformed in context Γ

$$\frac{}{\Gamma \vdash \mathbf{One}\ \mathbf{ok}}\ \textsc{Word}$$

$$\frac{\Gamma \vdash sh_1\ \mathbf{ok} \quad .. \quad \Gamma \vdash sh_n\ \mathbf{ok}}{\Gamma \vdash \mathbf{Comb}\ [sh_1\ ..\ sh_n]\ \mathbf{ok}}\ \textsc{Comb}$$

$$\frac{\mathbf{Nm}\ sname : [nfldsh_1\ ..\ nfldsh_n] \in \Gamma}{\Gamma \vdash \mathbf{Named}\ sname\ \mathbf{ok}}\ \textsc{Named}$$

$\boxed{\Gamma \vdash exp : sh}$ exp has shape sh in context Γ

$$\frac{}{\Gamma \vdash \mathbf{Const}\ num : \mathbf{One}}\ \textsc{Const}$$

$$\frac{\mathbf{Lc}\ v : sh \in \Gamma}{\Gamma \vdash \mathbf{Var\ Local}\ v : sh}\ \textsc{LocalVar}$$

$$\frac{\mathbf{Gb}\ v : sh \in \Gamma}{\Gamma \vdash \mathbf{Var\ Global}\ v : sh}\ \textsc{GlobalVar}$$

$$\frac{\Gamma \vdash exp_1 : sh_1 \quad .. \quad \Gamma \vdash exp_n : sh_n}{\Gamma \vdash \mathbf{RStruct}\ [exp_1\ ..\ exp_n] : \mathbf{Comb}\ [sh_1\ ..\ sh_n]}\ \textsc{RStruct}$$

$$\frac{\Gamma \vdash exp : \mathbf{Comb}\ [sh_1\ ..\ sh_{index}\ ..\ sh_n]}{\Gamma \vdash \mathbf{RField}\ index\ exp : sh_{index}}\ \textsc{RField}$$

$$\frac{\begin{array}{c}\mathbf{Nm}\ sname : [(fld_1, sh_1)\ ..\ (fld_n, sh_n)] \in \Gamma \\ \Gamma \vdash exp_1 : sh_1 \quad .. \quad \Gamma \vdash exp_n : sh_n\end{array}}{\Gamma \vdash \mathbf{NStruct}\ sname\ [(fld_1, exp_1)\ ..\ (fld_n, exp_n)] : \mathbf{Named}\ sname}\ \textsc{NStruct}$$

$$\frac{\begin{array}{c}\Gamma \vdash exp : \mathbf{Named}\ sname \\ \mathbf{Nm}\ sname : [(fld_1, sh_1)\ ..\ (field, sh)\ ..\ (fld_n, sh_n)] \in \Gamma\end{array}}{\Gamma \vdash \mathbf{NField}\ field\ exp : sh}\ \textsc{NField}$$

$$\frac{\Gamma \vdash sh\ \mathbf{ok} \quad \Gamma \vdash addr : \mathbf{One}}{\Gamma \vdash \mathbf{Load}\ sh\ addr : sh}\ \textsc{LdS}$$

$$\frac{\Gamma \vdash addr : \mathbf{One}}{\Gamma \vdash \mathbf{LoadHalf}\ addr : \mathbf{One}}\ \textsc{LdH}$$

$$\frac{\Gamma \vdash addr : \mathbf{One}}{\Gamma \vdash \mathbf{LoadByte}\ addr : \mathbf{One}}\ \textsc{LdB}$$

$$\frac{\Gamma \vdash exp_1 : \mathbf{One} \quad .. \quad \Gamma \vdash exp_n : \mathbf{One}}{\Gamma \vdash \mathbf{Op}\ binop\ [exp_1\ ..\ exp_n] : \mathbf{One}}\ \textsc{Exp}$$

$$\frac{\Gamma \vdash exp_1 : \mathbf{One} \quad .. \quad \Gamma \vdash exp_n : \mathbf{One}}{\Gamma \vdash \mathbf{Panop}\ panop\ [exp_1\ ..\ exp_n] : \mathbf{One}}\ \textsc{Panop}$$

$$\frac{\Gamma \vdash exp_1 : sh \quad \Gamma \vdash exp_2 : sh}{\Gamma \vdash \mathbf{Cmp}\ cmp\ exp_1\ exp_2 : \mathbf{One}}\ \textsc{Cmp}$$

$$\frac{\Gamma \vdash exp : \mathbf{One}}{\Gamma \vdash \mathbf{Shift}\ shift\ exp\ num : \mathbf{One}}\ \textsc{Shift}$$

$$\frac{}{\Gamma \vdash \mathbf{BaseAddr} : \mathbf{One}}\ \textsc{Base}$$

$$\frac{}{\Gamma \vdash \mathbf{TopAddr} : \mathbf{One}}\ \textsc{Top}$$

$$\frac{}{\Gamma \vdash \mathbf{BytesInWord} : \mathbf{One}}\ \textsc{BiW}$$

$\boxed{scope; \Gamma \vdash stmt\ \mathbf{ok}}$ $stmt$ shape checks in $scope$, context Γ

$$\frac{}{scope; \Gamma \vdash \mathbf{Skip}\ \mathbf{ok}}\ \textsc{Skip}$$

$$\frac{\Gamma \vdash sh\ \mathbf{ok} \quad \Gamma \vdash exp : sh \quad scope; \Gamma, \mathbf{Lc}\ v : sh \vdash stmt\ \mathbf{ok}}{scope; \Gamma \vdash \mathbf{Dec}\ v\ sh\ exp\ stmt\ \mathbf{ok}}\ \textsc{Dec}$$

$$\frac{\Gamma \vdash sh\ \mathbf{ok} \quad \mathbf{Fn}\ f : sh, [sh_1\ ..\ sh_n] \in \Gamma \quad \Gamma \vdash exp_1 : sh_1 \quad .. \quad \Gamma \vdash exp_n : sh_n \quad scope; \Gamma, \mathbf{Lc}\ v : sh \vdash stmt\ \mathbf{ok}}{scope; \Gamma \vdash \mathbf{DecCall}\ v\ sh\ f\ [exp_1\ ..\ exp_n]\ stmt\ \mathbf{ok}}\ \textsc{DecCall}$$

$$\frac{\mathbf{Lc}\ v : sh \in \Gamma \quad \Gamma \vdash exp : sh}{scope; \Gamma \vdash \mathbf{Assign\ Local}\ v\ exp\ \mathbf{ok}}\ \textsc{AssignLocal}$$

$$\frac{\begin{array}{c}\mathbf{Gb}\ v : sh \in \Gamma \\ \Gamma \vdash exp : sh\end{array}}{scope; \Gamma \vdash \mathbf{Assign\ Global}\ v\ exp\ \mathbf{ok}}\ \textsc{AssignGlobal}$$

$$\frac{\begin{array}{c}\mathbf{Fn}\ f : sh_0, [sh_1\ ..\ sh_n] \in \Gamma \\ \mathbf{Lc}\ v : sh_0 \in \Gamma \\ \Gamma \vdash exp_1 : sh_1 \quad .. \quad \Gamma \vdash exp_n : sh_n\end{array}}{scope; \Gamma \vdash \mathbf{AssignCall}\ v\ handle\ f\ [exp_1\ ..\ exp_n]\ \mathbf{ok}}\ \textsc{AssignCall}$$

$$\frac{\Gamma \vdash exp : sh}{\mathbf{Body}\ sh; \Gamma \vdash \mathbf{Return}\ exp\ \mathbf{ok}}\ \textsc{Return}$$

$$\frac{\begin{array}{c}\mathbf{Fn}\ f : sh, [sh_1\ ..\ sh_n] \in \Gamma \\ \Gamma \vdash exp_1 : sh_1 \quad .. \quad \Gamma \vdash exp_n : sh_n\end{array}}{\mathbf{Body}\ sh; \Gamma \vdash \mathbf{TailCall}\ f\ [exp_1\ ..\ exp_n]\ \mathbf{ok}}\ \textsc{TailCall}$$

$$\frac{\begin{array}{c}\mathbf{Fn}\ f : sh_0, [sh_1\ ..\ sh_n] \in \Gamma \\ \Gamma \vdash exp_1 : sh_1 \quad .. \quad \Gamma \vdash exp_n : sh_n\end{array}}{scope; \Gamma \vdash \mathbf{StandAloneCall}\ handle\ f\ [exp_1\ ..\ exp_n]\ \mathbf{ok}}\ \textsc{StandAloneCall}$$

$$\frac{\Gamma \vdash exp_1 : \mathbf{One} \quad .. \quad \Gamma \vdash exp_4 : \mathbf{One}}{scope; \Gamma \vdash \mathbf{ExtCall}\ f\ exp_1\ exp_2\ exp_3\ exp_4\ \mathbf{ok}}\ \textsc{ExtCall}$$

$$\frac{\begin{array}{c}scope; \Gamma \vdash stmt_1\ \mathbf{ok} \\ scope; \Gamma \vdash stmt_2\ \mathbf{ok}\end{array}}{scope; \Gamma \vdash \mathbf{Seq}\ stmt_1\ stmt_2\ \mathbf{ok}}\ \textsc{Seq}$$

$$\frac{\begin{array}{c}\Gamma \vdash exp : \mathbf{One} \\ scope; \Gamma \vdash stmt_1\ \mathbf{ok} \\ scope; \Gamma \vdash stmt_2\ \mathbf{ok}\end{array}}{scope; \Gamma \vdash \mathbf{If}\ exp\ stmt_1\ stmt_2\ \mathbf{ok}}\ \textsc{If}$$

$$\frac{\begin{array}{c}\Gamma \vdash exp : \mathbf{One} \\ scope; \Gamma \vdash stmt\ \mathbf{ok}\end{array}}{scope; \Gamma \vdash \mathbf{While}\ exp\ stmt\ \mathbf{ok}}\ \textsc{While}$$

$$\frac{}{scope; \Gamma \vdash \mathbf{Break}\ \mathbf{ok}}\ \textsc{Break}$$

$$\frac{}{scope; \Gamma \vdash \mathbf{Continue}\ \mathbf{ok}}\ \textsc{Continue}$$

$$\frac{\Gamma \vdash exp : sh}{scope; \Gamma \vdash \mathbf{Raise}\ eid\ exp\ \mathbf{ok}}\ \textsc{Raise}$$

$$\frac{\begin{array}{c}\Gamma \vdash addr : \mathbf{One} \\ \Gamma \vdash exp : sh\end{array}}{scope; \Gamma \vdash \mathbf{Store}\ addr\ exp\ \mathbf{ok}}\ \textsc{St}$$

$$\frac{\begin{array}{c}\Gamma \vdash addr : \mathbf{One} \\ \Gamma \vdash exp : \mathbf{One}\end{array}}{scope; \Gamma \vdash \mathbf{StoreHalf}\ addr\ exp\ \mathbf{ok}}\ \textsc{StH}$$

$$\frac{\Gamma \vdash \mathit{addr} : \mathbf{One} \quad \Gamma \vdash \mathit{exp} : \mathbf{One}}{\mathit{scope}; \Gamma \vdash \mathbf{StoreByte}\ \mathit{addr}\ \mathit{exp}\ \mathbf{ok}}\ \textsc{StB}$$

$$\frac{\mathbf{Lc}\ v : \mathbf{One} \in \Gamma \quad \Gamma \vdash \mathit{addr} : \mathbf{One}}{\mathit{scope}; \Gamma \vdash \mathbf{ShMemLoad}\ \mathit{opsize}\ \mathbf{Local}\ v\ \mathit{addr}\ \mathbf{ok}}\ \textsc{ShLdLocal}$$

$$\frac{\mathbf{Gb}\ v : \mathbf{One} \in \Gamma \quad \Gamma \vdash \mathit{addr} : \mathbf{One}}{\mathit{scope}; \Gamma \vdash \mathbf{ShMemLoad}\ \mathit{opsize}\ \mathbf{Global}\ v\ \mathit{addr}\ \mathbf{ok}}\ \textsc{ShLdGlobal}$$

$$\frac{\Gamma \vdash \mathit{addr} : \mathbf{One} \quad \Gamma \vdash \mathit{exp} : \mathbf{One}}{\mathit{scope}; \Gamma \vdash \mathbf{ShMemStore}\ \mathit{opsize}\ \mathit{addr}\ \mathit{exp}\ \mathbf{ok}}\ \textsc{ShSt}$$

$$\frac{}{\mathit{scope}; \Gamma \vdash \mathbf{Tick}\ \mathbf{ok}}\ \textsc{Tick}$$

$$\frac{}{\mathit{scope}; \Gamma \vdash \mathbf{Annot}\ \mathit{string}_1\ \mathit{string}_2\ \mathbf{ok}}\ \textsc{Annot}$$

$\boxed{\Gamma \vdash \mathit{decl} \overset{\text{name}}{\rightsquigarrow} \Gamma'}$ *decl* adds a struct name to context Γ to produce the updated context Γ'

$$\frac{}{\Gamma \vdash \mathbf{Func}\ \mathit{sh}\ f\ \mathit{bool}\ [\mathit{param}_1\ ..\ \mathit{param}_n]\ \mathit{stmt} \overset{\text{name}}{\rightsquigarrow} \Gamma}\ \textsc{Function}$$

$$\frac{}{\Gamma \vdash \mathbf{Decl}\ \mathit{sh}\ v\ \mathit{exp} \overset{\text{name}}{\rightsquigarrow} \Gamma}\ \textsc{Global}$$

$$\frac{\mathit{fld}_1\ ..\ \mathit{fld}_n\ \mathbf{all\ distinct} \quad \Gamma \vdash \mathit{sh}_1\ \mathbf{ok} \quad ..\quad \Gamma \vdash \mathit{sh}_n\ \mathbf{ok}}{\Gamma \vdash \mathbf{Name}\ \mathit{sname}\ [(\mathit{fld}_1, \mathit{sh}_1)\ ..\ (\mathit{fld}_n, \mathit{sh}_n)] \overset{\text{name}}{\rightsquigarrow} \Gamma, \mathbf{Nm}\ \mathit{sname} : [(\mathit{fld}_1, \mathit{sh}_1)\ ..\ (\mathit{fld}_n, \mathit{sh}_n)]}\ \textsc{StructName}$$

$\boxed{\Gamma \vdash \mathit{decl} \rightsquigarrow \Gamma'}$ *decl* adds a global variable or function to context Γ to produce the updated context Γ'

$$\frac{\mathit{sh} = \mathbf{One}}{\Gamma \vdash \mathbf{Func}\ \mathit{sh}\ \text{“}\mathbf{main}\text{”}\ \mathbf{false}\ [\,]\ \mathit{stmt} \rightsquigarrow \Gamma, \mathbf{Fn}\ \text{“}\mathbf{main}\text{”} : \mathit{sh}, [\,]}\ \textsc{FuncMain}$$

$$\frac{\mathit{sh}_0 = \mathbf{One} \quad ..\quad \mathit{sh}_n = \mathbf{One}}{\Gamma \vdash \mathbf{Func}\ \mathit{sh}_0\ f\ \mathbf{true}\ [(v_1, \mathit{sh}_1)\ ..\ (v_n, \mathit{sh}_n)]\ \mathit{stmt} \rightsquigarrow \Gamma, \mathbf{Fn}\ f : \mathit{sh}_0, [\mathit{sh}_1\ ..\ \mathit{sh}_n]}\ \textsc{FuncExport}$$

$$\frac{f \neq \text{“}\mathbf{main}\text{”} \quad \Gamma \vdash \mathit{sh}_0\ \mathbf{ok} \quad ..\quad \Gamma \vdash \mathit{sh}_n\ \mathbf{ok} \quad \mathbf{size}\,(\mathit{sh}_0) \leq 32}{\Gamma \vdash \mathbf{Func}\ \mathit{sh}_0\ f\ \mathbf{false}\ [(v_1, \mathit{sh}_1)\ ..\ (v_n, \mathit{sh}_n)]\ \mathit{stmt} \rightsquigarrow \Gamma, \mathbf{Fn}\ f : \mathit{sh}_0, [\mathit{sh}_1\ ..\ \mathit{sh}_n]}\ \textsc{FuncStatic}$$

$$\frac{\Gamma \vdash \mathit{sh}\ \mathbf{ok} \quad \Gamma \vdash \mathit{exp} : \mathit{sh}}{\Gamma \vdash \mathbf{Decl}\ \mathit{sh}\ v\ \mathit{exp} \rightsquigarrow \Gamma, \mathbf{Gb}\ v : \mathit{sh}}\ \textsc{Global}$$

$$\frac{}{\Gamma \vdash \textbf{Name}\ sname\ [nfldsh_1\ ..\ nfldsh_n] \rightsquigarrow \Gamma}\ \textsc{StructName}$$

$\boxed{\Gamma \vdash decl\ \textbf{ok}}$ *decl* shape checks in context Γ

$$\frac{\textbf{Body}\ sh; \Gamma \vdash stmt\ \textbf{ok}}{\Gamma \vdash \textbf{Func}\ sh\ f\ bool\ [param_1\ ..\ param_n]\ stmt\ \textbf{ok}}\ \textsc{Function}$$

$$\frac{}{\Gamma \vdash \textbf{Decl}\ sh\ v\ exp\ \textbf{ok}}\ \textsc{Global}$$

$$\frac{}{\Gamma \vdash \textbf{Name}\ sname\ [nfldsh_1\ ..\ nfldsh_n]\ \textbf{ok}}\ \textsc{StructName}$$

$\boxed{program\ \textbf{ok}}$ *program* shape checks

$$\frac{\begin{array}{l}\textbf{empty} \vdash decl_1 \overset{\text{name}}{\rightsquigarrow} \Gamma_2 \quad .. \quad \Gamma_n \vdash decl_n \overset{\text{name}}{\rightsquigarrow} \Gamma_{n+1} \\ \Gamma_{n+1} \vdash decl_1 \rightsquigarrow \Gamma_{n+2} \quad .. \quad \Gamma_{2n} \vdash decl_n \rightsquigarrow \Gamma \\ \Gamma \vdash decl_1\ \textbf{ok} \quad .. \quad \Gamma \vdash decl_n\ \textbf{ok}\end{array}}{[decl_1\ ..\ decl_n]\ \textbf{ok}}\ \textsc{Prog}$$

Bibliography

Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages*, 4 (OOPSLA), 2020. ISSN 24751421. doi: 10.1145/3428204.

Jana Dunfield and Neel Krishnaswami. Bidirectional Typing. *ACM Computing Surveys*, 54(5), 2021. doi: 10.1145/3450952.

Dan Grossman, M. Hicks, T. Jim, and G. Morrisett. Cyclone: A type-safe dialect of C. *C/C++ Users Journal*, 23(1), 2005.

Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5), 2014. doi: 10.1007/s10664-013-9289-1.

Gernot Heiser, Peter Chubb, Alex Brown, Courtney Darville, and Lucy Parker. sDDF Design: Design, Implementation and Evaluation of the seL4 Device Driver Framework, 3 2024. URL `https://trustworthy.systems/publications/papers/Heiser_CBDP_24.pdf`.

R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146, 1969. doi: 10.2307/1995158.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 2014. doi: 10.1145/2535838.2535841.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978. doi: 10.1016/0022-0000(78)90014-4.

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1), 2000. ISSN 01640925. doi: 10.1145/345099.345100.

Johannes Åman Pohjola, Hira Taqdees Syeda, Miki Tanaka, Krishnan Winter, Tsun Wang Sau, Benjamin Nott, Tiana Tsang Ung, Craig McLaughlin, Remy Seassau, Magnus O. Myreen, Michael Norrish, and Gernot Heiser. Pancake: Verified Systems Programming Made Sweeter. In *PLOS 2023 - Proceedings of the 12th Workshop on Programming Languages and Operating Systems, Part of: SOSP 2023*, 2023. doi: 10.1145/3623759.3624544.

seL4 Foundation. seL4 Microkit GitHub, 2023. URL `https://github.com/seL4/microkit`.

Axel Simon. Optimal inference of fields in row-polymorphic records. In *ACM SIGPLAN Notices*, volume 49, 2014. doi: 10.1145/2594291.2594313.