Submitted in part fulfilment for the degree of MEng.

Modelling of control systems: case studies using C

Matthew Windsor
6 May 2014

Supervisor: Prof. Ana Cavalcanti

Number of words = 33724, as counted by detex main | wc -w (with appendices and listings removed). This includes the body of the report only.
Abstract

Verification of implementations against their specifications is important in the context of safety-critical systems. This is because it provides a formal, rigorous assurance of the correctness of an implementation in the context of specified requirements, precluding the potential of implementation errors that risk lives.

One verification technique involves the production, from both the specification and the implementation, of models using a formal modelling language. We may then use refinement to prove the implementation model satisfies all the properties defined in the specification model. This requires a translation from the implementation to a model that satisfies the need for formality and correctness.

A prototypal approach to developing models of control-system implementations in Haskell is developed and discussed; we focus on implementations in a working, safe subset of the programming language C. We use the Circus notation, which combines Z, CSP and refinement calculus in an integrated manner. We focus on C for its popularity and Circus due to the existence of a formal process for verification by refinement of models in that notation given models of implementations.

We conclude by discussing potential future work in automating and perfecting the approach, including the relaxation of assumptions made in the prototype.
In memory of Craig Richard Windsor (1969–2012), a wonderful uncle, friend, and inspiration, without whom our family is incomplete.

Acknowledgements
I would first like to thank Ana Cavalcanti and Pedro Ribeiro, for offering insights, assistance and advice throughout this project, and without whom it would not have been possible.

I would also like to thank Ana for being a wonderful supervisor this academic year, for pushing and motivating me to achieve my potential with great patience and kindness, and for introducing me to the elegance and beauty of formal methods in my third year, which has been a great influence on my work this year and, hopefully, for years to come.

I also thank Lilian Blot, for being an excellent supervisor through much of my undergraduate studies; Donal, Danny and Brian, for being true friends through the years we shared our course; and to everyone who supported me throughout my education and helped me to become who I am today.

Statement of Ethics
To our knowledge, our is free from any direct ethical concerns. It is a theoretical exercise and has not yet been used in practice. We expect that any application of our techniques will only be performed after a thorough analysis of their correctness and aptness, in order to prevent the possibility of injury or death resulting from any use of our work. No experimentation on humans or animals was performed during the course of our work.
Contents

1 Introduction 8
  1.1 Motivation ........................................ 8
  1.2 Objectives ......................................... 9
  1.3 Structure of this document ...................... 9

2 Programming Languages in the Safety-Critical Domain and their Formal Models 10
  2.1 Ada .................................................. 10
  2.2 Java ................................................ 10
  2.3 C .................................................... 11
  2.4 C-Like Languages .................................. 12
  2.5 Formal Models of Safety-Critical Languages ... 13
  2.6 Modelling programs in Circus .................. 15
  2.7 Final Considerations ................................ 16

3 A Subset of C: ACDC 17
  3.1 Approach ............................................. 17
  3.2 Criteria ............................................. 17
  3.3 ACDC ............................................... 18
  3.4 Final considerations ................................ 20

4 From ACDC to Circus 21
  4.1 Technique .......................................... 21
  4.2 Assumptions ........................................ 21
  4.3 Overview .......................................... 22
  4.4 Syntax and general constructs ................. 24
  4.5 Flattening scopes .................................. 33
  4.6 Capturing declarations ............................ 40
  4.7 Abstracting temporal information ............. 42
  4.8 Capturing functions called by the ACDC main functions ........................................ 45
  4.9 Capturing variable usage ......................... 46
  4.10 Retrieving struct definitions ................... 47
  4.11 Retrieving variable and function types ....... 47
  4.12 Building the Circus model ........................ 48
  4.13 Final considerations ................................ 65

5 Evaluation 66
  5.1 Approaches used .................................. 66
  5.2 Results ............................................. 67
  5.3 Qualitative evaluation ............................ 67
  5.4 Final considerations ................................ 67

6 Conclusions 68
  6.1 Haskell as a language for model generation . 68
  6.2 Differences from the Ada process ................ 68
  6.3 Revisiting the context ............................. 69
  6.4 Future work ........................................ 69

Appendix A Additional Definitions 71
  A.1 Syntax .............................................. 71
  A.2 Flattening scopes .................................. 72
  A.3 Capturing declarations ............................ 74
  A.4 Abstracting temporal information ............. 74
  A.5 Capturing functions called by the ACDC main functions ........................................ 75
  A.6 Capturing variable usage ......................... 75
  A.7 Building the Circus model ....................... 75
  A.8 Traversal functions ................................ 81
  A.9 Syntax utility functions ......................... 85

Appendix B Supplemental Files and Running the Process 90
  B.1 Supplemental files .................................. 90
  B.2 Evaluating the process ............................ 90

Appendix C Automated test log 91

Appendix D Circus model of the case study 95

Appendix E Glossary 98
  E.1 Haskell syntax used in the project .............. 98
1 Introduction

In this chapter, we introduce our work and its context. In section 1.1 we discuss our motivation, including a case study. We then provide a summary of our objectives in section 1.2, before outlining the report structure in section 1.3.

1.1 Motivation

The problem. Lack of rigour in developing safety-critical systems—failure of which may cause injury or death—is a significant problem. Poor design and quality control in software development for the Therac-25 radiation treatment system, for example, caused fatal doses of radiation to be administered [1]. In summarising the Therac-25 case, Leveson notes that the focus on removing bugs alone is insufficient; care in the design of safety-critical systems, and attention towards the specification and documentation of such systems, is key.

Towards a solution. One approach to improving this situation is formal methods, defined by Woodcock et al. [2] as 'mathematical techniques, often supported by tools, for developing software and hardware systems'. Formal methods bring the rigour of mathematical reasoning to software development, allowing the proof of properties of systems and a high degree of confidence in the correctness of software.

However, formal methods require time, effort and mathematical knowledge to follow manually. Although Hall [3] argues that the training required to practice formal methods by hand is simple, training nonetheless is required, and this can be seen as a distraction from the task of sending an implementation to market as cheaply and quickly as possible. Hall also notes that, while this may be the case for formal specifications, performing any form of proof-based methods requires far greater mathematical skill, and is out of the reach of the average programmer.

An ideal situation: automation. Therefore, the automation of part or all of a formal technique, lessening or removing the need for manual effort, greatly improves the suitability of that technique in the safety-critical domain. Cliff, O'Hearn and Woodcock argue that, given the appropriate automating tool support, formal methods 'could become widespread and transform software engineering', and that formal methods could eventually be considered the cheapest method of producing software, even outside the safety-critical domain [4].

The formal specification of safety-critical systems is of particular interest. In the case of the Therac-25, formally specifying its software system for automated verification would have captured design flaws even using the techniques available in 1993 [5]. We focus on specification using formal modelling techniques, an area of formal methods in active research at time of writing. Specifically, we wish to be able to use these specifications in proofs of the correctness of implementations.

The ideal situation is that in which a specification and implementation can be handed to a fully automated process that mechanically derives models of specification and implementation, proving correctness of the latter with regards to the former via refinement, with no expensive manual effort required. This process could be performed by designers and programmers without the need to consult formal methods experts. Ideally, no deliberate formal methods effort need be planned for—unlike in techniques such as correctness by construction [4], where the implementation must be derived formally from the specification—, allowing flexibility as to when and how formalism should enter the software engineering process. Such post facto techniques could be applied to existing systems, including legacy systems already in production.

The present situation. Progress is being made towards this ideal. A high-level strategy for proof via refinement of implementations of Simulink control-law diagrams has been devised by Cavalcanti et al. [6]. This process uses the ClawZ [7] tool for mechanically translating Simulink diagrams to Z specifications, and targets implementations in a subset of the Ada programming language, adhering to a particular architecture. A technique for producing Circus models of such implementations has been formalised by Ribeiro [8], providing the missing part of the proof strategy.

For a subset of implementations of Simulink specifications, there is thus a procedure for proving the correctness of the former with respect to the properties of the latter. However, to achieve the ideal, there are questions we must answer:

- **Can we extend the process to languages other than Ada?** Ada is a well-known language in the safety-critical sector, but is not the only language used. Many embedded safety-critical systems use the more low-level, but less safety-focused, C programming language. We cannot simply apply the Ada strategy to C programs: their semantics are different. We must explore the possibility of extending the process beyond its focus on Ada.

- **Can we automate the process?** The work by Ribeiro is a fully specified process, but is not automated. It is provided in a mathematical meta-notation that cannot be directly executed over an Ada program to provide a Circus model. To make formal methods more applicable, we need to reduce the amount of work that must be done manually by leveraging the ability of computers to follow the rigorous translation processes we define.

- **Can we reproduce the previous success of the process?** The work by Ribeiro and Cavalcanti et al. demonstrated, via case study, that the process of proving correctness by model refinement is feasible for Ada. More evaluation of the strategy would yield greater confidence in the strategy, especially when the strategy has been generalised and automated to address the preceding questions.
Answering these questions, and contributing towards the goal of providing flexible, automated, and cheap techniques for proof-by-refinement, motivates our work. By making it easier, faster and cheaper to gain confidence about the correctness of programs, we can reduce the potential for quality control in safety-critical systems to be overlooked and sidelined, and the fatal consequences that follow. In the next section, we define the objectives of our work.

1.2 Objectives

We thus aim to contribute to the ability to reason about the correctness of safety-critical software—especially embedded systems such as the Therac-25—by investigating the generation of formal models of implementations of control systems that implement formal specifications. Specifically, this work sets out to answer the questions we posed in the motivation.

We focus on implementations in the C programming language. C is a popular language used in safety-critical systems, and particularly embedded and resource-constrained software. Although the Therac-25 was not implemented in C, it is not unusual to expect a similar system programmed today to have software components programmed in that language.

As a language for which there is no formal technique for performing the form of proof-by-refinement discussed earlier, our work will make progress towards providing this technique and, thus, a highly automated means of reasoning about the correctness of C implementations of control systems.

In this work, we aim to:

- Identify a practical, safe subset of C—by examining previous work on subsets of C and other languages—, such that implementations therein can be formally modelled, and the subset is representative of usage in the safety-critical sector;

- Identify a technique suitable for the modelling of safety-critical, state-rich, reactive systems—such as the Therac-25—with the ability to prove refinement of implementation models against specification models;

- Devise a translation process for creating models of implementations in said subset that may be used in the wider process of verification of implementations of specifications via model refinement;

- Automate the translation process to a degree such that we can input an implementation of a control-law diagram and receive as output a full Circus model of that implementation. (We leave the automation of the rest of Cavalcanti et al.’s strategy, for C implementations, as future work.)

Our main deliverable is the translation process. We also discuss relevant literature, aiming to justify decisions made up to the point of elaborating the translation process by reference and comparison to said literature. In the next section we discuss the structure for the remainder of this report.

1.3 Structure of this document

In the next chapter, we review a non-exhaustive set of languages commonly used in the safety-critical domain, as well as techniques for formally modelling implementations in those languages. We choose our target language and modelling technique—C and Circus respectively—, and defend our choice with respect to the alternatives explored.

In chapter 3, we consider the need to reduce the complexity and strengthen the safety profile of our implementation language. We present ACDC, a simple and safe language subset our process accepts as its input. We justify our subset by referring to practices used in the industry as well as arguing for the safety benefits of our decisions.

In chapter 4, we define the process of formulating Circus models of implementations using the Haskell language, Circus being defined in chapter 2. This section, a pre-processed and abridged Literate Haskell script, is the focus of the report.

We conclude in chapter 6 by discussing our contributions to the existing body of work on verification of safety-critical implementations. We also consider the potential for future work in our area.
2 Programming Languages in the Safety-Critical Domain and their Formal Models

There is no one language in use for the programming of safety-critical systems. We consider a small set of well-used languages in the field, and justify the choice of one language from this set for further study. Specifically, we discuss Ada, Java, C, and profiles and subsets thereof used in the safety-critical domain. Following this, we discuss techniques for the formal modelling of implementations in these languages, seeking to justify the choice of one modelling notation. We begin by discussing Ada.

2.1 Ada

Ada, originally Green [9], was designed to meet the STEELMAN requirements produced as part of the United States Department of Defense’s Common Language Effort [10]. This effort was intended to reduce the large number of incompatible programming languages in use by the DoD, allowing a focus on one language.

The focus of STEELMAN was towards the development of embedded computer systems: software contained inside larger systems not immediately tasked with computation, such as those used in vehicles. The DoD’s focus on military systems required attention to software safety; errors in military-grade systems could have effects from failing to accomplish a mission to endangering the lives of civilians and friendly troops.

Ada was the winning entry in a design competition initiated by the DoD after finding that no language sufficiently met its requirements of reliability, concurrency, real-time access, machine-independence and other concerns. Ada is, thus, highly suitable for developing safety-critical systems, especially embedded-level systems.

Ada has more verbose syntax than C and Java. For example, in the latter languages, an if statement is closed by the end of its body statement, which is normally either a semicolon or a closing curly brace denoting the end of a compound statement; Ada expects an explicit end if to follow its if. In Ada, programming errors related to accidentally leaving bodies open (and inadvertently closing them when trying to close other statements) are thus easier to trap at compile time. However, the need to write more ‘boilerplate’, as opposed to useful code, may cause a decrease in attention, and increase in errors.

Ada is, then, a language with differences in intent, syntax and history to other languages used in the safety-critical field. It is designed for safety and reliability, rather than having been repurposed for those concerns.

Despite being created for use in a safety-conscious sector, Ada contains aspects unsafe or unhelpful for safety-critical design. An Ada subset, SPARK [11], enhances its safety and ease of verification. SPARK removes some aspects of Ada, while mandating the use of annotations on the code, to meet criteria including logical soundness, verifiability, and space and time bounds.

SPARK’s annotations, or formal comments, facilitate the static analysis and verification of Ada code. They capture information about the intended use of code fragments by their environment, such as the explicit set of global variables imported and exported by procedures. Tools such as the SPARK Examiner use SPARK’s rules and a program’s annotations to ensure that the program is consistent with both.

SPARK’s restrictions to Ada include the removal of go-to statements and access types (similar to C’s pointers). Tasks, exceptions and generic units are also removed, and some areas of the language are simplified.

SPARK is one example of a safety-critical subset of a language. We will see this concept applied to Java and C later on.

We now consider Java.

2.2 Java

Java was created in 1995 as part of an intended ‘small, reliable, portable, distributed, real-time operating platform’ ready for the Internet-driven boom in distributed network programming [12]. It shares much of its syntax, types, and concepts with C.

Java is object-oriented: unlike C, it includes as a type the concept of an object. This is a trait it shares with recent versions of Ada, and with extensions to C such as C++ and Objective-C. Another interesting feature of Java is its use of automatic garbage collection—its run-time detects objects no longer referenced and removes them from memory without cues from the programmer—, coupled with automatic allocation of memory for objects upon initialisation. This allows dynamic memory access without enabling the class of errors related to memory mis-management, but makes it harder to reason about the timings and safety of code. This motivates the design of a variant of Java, described below.

Real-Time Specification for Java (RTSJ). The RTSJ [13] is a Java profile for real-time systems. Developed starting from 1999 by the Real-Time for Java Experts Group, the RTSJ intended to address issues with Java’s scheduling, memory management and synchronisation to allow reasoning about temporal behaviour. This reasoning is important in the analysis of real-time systems and is also useful for safety-critical systems. The RTSJ was designed to a set of requirements [14], including requirements on the temporal behaviour of garbage collection and the means of concurrency and synchronisation.

The RTSJ was conceived before real-time garbage collectors with sufficiently predictable latencies were available. Thus, the RTSJ provides for memory areas: ‘[regions] of memory outside the garbage-collected heap that [can be used] to allocate objects’. These include the concept of immortal memory, which persists all objects created in it until the program terminates, and scoped memory, in which memory-area objects govern a syntactic block in which objects may be created; allocated scoped memory
is deallocated at the end of the syntactic block. Memory areas remove the need for garbage collection except in the cases of references from memory area objects to heap objects, thus allowing for a more predictable temporal behaviour.

The RTSJ was intended not to introduce new syntax to Java, to allow the existing set of Java tools to work as-is. Later, we consider safety-critical language subsets that follow suit, and those which, for various reasons, do not.

The RTSJ was defined to meet the predictability and analysability needs of real-time systems. Reliability concerns, prevalent in safety-critical systems, are not as rigorously addressed.

**Safety-Critical Java (SCJ).** SCJ is more recent: the draft standard JSR-302 was first released in 2010 [15]. SCJ builds on RTSJ, with the compatibility aim of SCJ-compliant programs being executable on RTSJ-compliant platforms.

SCJ is designed for use in safety-critical systems where, in addition to the demands on predictability that shaped RTSJ, reliability concerns are of vital importance. To improve Java’s performance and the ability to reason about programs, parts of the Java environment are scaled back and optimised for efficiency.

SCJ structures programs using the concept of **missions**: bounded sets of limited, schedulable objects given access to a pool of memory in which objects can be created, but will remain until the termination of the mission. Missions have two phases, initialisation and execution, in which objects are allocated into both mission and immortal memory.

The tightening of memory access into the concept of pools in which reclamation only occurs at mission or program termination is an example of the restrictions made to RTSJ to improve the verifiability and reliability of SCJ programs.

SCJ-hardened Java is a highly appropriate language for the specification of real-time, safety-critical software due to its modern, object-oriented paradigm and basis in a popular, high-level language. One disadvantage of SCJ is that it is recent and thus not fully explored. Its draft standard notes that ‘the safety-critical software community is conservative in adopting new technologies, approaches and architectures.’

We now discuss the language, C, that forms the focus of our work.

### 2.3 C

C [16] is an imperative, low-level yet machine-portable language that compiles to native machine code. It supports direct memory access via pointers (values representing memory addresses), primitive types such as integers (signed and unsigned, in a variety of fixed bit-sizes) and floating-point numbers, as well as basic composite types such as structs (conjunctions of types) and unions (untagged disjunctions of types). Arrays are supported, but are for most purposes equivalent to pointers to the start of contiguous allocated memory blocks. Strings are implemented as arrays of integers representing character points.

As we mainly consider C in this work, we discuss it in more detail than Java and Ada. We first consider its type system.

**Types.** According to Cardelli [17], a language’s type system shall ‘prevent the occurrence of execution errors during the running of a program.’ It is important, then, to consider it when judging a language’s appropriateness for safety-critical systems; a type system forbidding as many execution errors as possible is of higher value in an environment that cannot tolerate any type-based execution errors.

C’s ancestors, the BCPL [18] and B [19] languages, were *untyped*: programs could change the interpretation of their values from one expression to the next. This provides a degree of flexibility that may lead to simple errors, such as utilising an operator intended for one conceptual type on a value of the other, being allowed to occur at run-time. C introduced a type system that is *static*: types are known at compile time and type errors can be checked before the program may run—, and *manifest*: each variable and function is explicitly given a signature that identifies its type with no need for inference. However, C is *weakly* typed: it permits the implicit conversion of arithmetic types. C’s type system, in general, betrays the fact that it derives from languages that had no such system at all.

**Critique and rationale.** We wish to justify our focus on C in the context of safety-critical systems, especially as it is not the sole language of use in the field. There are advantages to C from a safety-critical perspective [20]. Notably, C is:

- **Portable** C compilers exist for a wide variety of architectures, processors, and operating systems;
- **Efficient** Programs in C compile to direct machine code, as opposed to virtual-machine or interpreted bytecode;
- **Standard** International standards—ISO/IEC 9899:1990 (C90), 9899:1999 (C99), and 9899:2011 (C11)—define C;
- **Low-level** C allows direct, low-level access to hardware capabilities;
- **Statically analysable** Many tools exist to do so, for example the Clang static analyser [21];
- **Popular** There is thus there is a sizeable corpus of experience in C, including in the fields in which we are interested.

The last observation is the most important: the popularity of C in the safety-critical sector means that work furthering the formality of development in C will have a wide, strong impact.

C benefits from what Ritchie et al. describe in 1978 as ‘economy of expression’: it allows great expressive power in a simple, terse, easily learned syntax. We thus need not consider the transformation of large amounts of syntactic forms and cases.

There are, however, some problems with C in the context of our work. Kernighan and Ritchie [22] state that ‘C, like any other language, has its blemishes.’ For our purposes, these include the following:
• Items left unspecified, undefined or left to the implementation by the standards, including evaluation order of expressions and side-effects;
• The low-level nature of C binds it to assumptions about its target hardware, such as the availability of memory as an array of directly addressable locations, that need not hold for other languages;
• C’s dynamic memory access allows for a dangerous class of errors, such as segmentation violations, double-frees and memory leaks. Neither Java nor Ada allow such low-level access and seldom suffer from these problems.

C has benefits in the context of safety-critical systems, notably its popularity and efficiency. Primarily, we consider C for its widespread industrial use. We must, however, deal with ‘blemishes’ that do not exist for other languages used in the field.

We now consider the field of C-like languages and their usage in the safety-critical domain.

2.4 C-Like Languages

C’s popularity and maturity has seen the creation of many derivative languages. Some are designed for safety-critical concerns; this is similar to the relationship between Ada and SPARK, as well as between Java and SCJ. We identify a cross-section of C-like languages and how they build on, or restrict, C to improve safety, facilitate reasoning, and other benefits.

2.4.1 MISRA C

A key effort in subsetting C in the industry is that of the Motor Industry Software Reliability Association (MISRA). MISRA is a collaboration between vehicle manufacturers, component suppliers and engineering consultancies which seeks to promote best practice in developing safety-related electronic systems in road vehicles and other embedded systems.

MISRA C is defined in ‘Guidelines for the use of the C language in Vehicle Based Software’ (1998 [24]) and ‘Guidelines for the use of the C language in critical systems’ (2004 [25] and 2012 [26]), each new document superseding the previous. These form a negative subset of C: the guidelines impose restrictions on standard C to form a C subset.

The 2012 version of MISRA C is a series of guidelines ranging in level of enforcement (from advisory guidelines, which may be ignored, to required guidelines, which must only be deviated from with a formal exception declaration, and mandatory guidelines, which cannot be deviated from at all), level of decidability (whether or not the guidelines can be checked statically, and level of specification (whether or not the guidelines are complete enough to be checked statically without further information). It is based on the C99 standard of C; previous versions used the then-more common C90 standard.

Many of the guidelines are decidable and fully specified. Therefore, static code analysers exist that are claimed to be able to check MISRA C compliance or, more correctly, compliance to those guidelines that can indeed be analysed statically [26].

In summary, MISRA is an example of the style of subsets of C used in the industry, and is one of the most popular of what we call C-like languages in use in safety-critical systems.

2.4.2 Clight (Blazy et al.)

Clight [27], a ‘large subset’ of C, is the language targeted by the CompCert formally verified compiler. Its main differences from Standard C are that:

• The integral types (char, short, long etc.) are given a specific and well-defined size;
• Clight’s semantics imposes a left-to-right evaluation order and is deterministic;
• There are no block-scoped and static variables, and therefore no block statement;
• Expressions cannot contain side-effects, including assignments and function calls.

The specification of language areas left ill-defined by Standard C (here, the integral type sizes, and expression evaluation order) is an important step when subsetting a language for safety-critical use: it is important to be able to reason about a program’s behaviour in a deterministic manner.

The absence of compound statements (blocks) from the syntax allowed by Clight is problematic. Without the ability to declare variables at block level, we lose the possibility of shadowing variables at higher scope, but also lose the advantage of being able to limit the lifespan of variables to sub-scopes within a wider function (thus reducing the likelihood of those variables being used outside of their area of intended use). Another effect of the removal of compound statements is that this limits if, for and while statements to possessing bodies of strictly one statement, restricting their usefulness.

One mitigation is to replace blocks with externalised functions, and pass the environment of automatic local variables used by those functions as parameters. This would formalise the contract between the outer and inner scopes defined by the existence of the block, but potentially generate function-call overhead, and certainly increase the verbosity of the programs (and, thus, the potential for typographical errors).

Removing blocks from the language eliminates the need to discern which variables are in scope at each statement, and thus removes a significant step in the modelling and reasoning process for C programs. However, we feel that the existence of blocks is justified, as the advantages of limiting scope without the overhead of functions outweigh this technical gain.

Clight, despite being a permissive subset, makes progress towards being a C-like language usable for safety-critical development. Some of its restrictions, however, are contentious.
2.4.3 C-Light (Nepomniashchyy et al.)

We also investigate another, earlier subset of C, named C-Light (not to be confused with the above Clight). This subset was proposed by Nepomniashchyy et al. in 2002 [28].

C-Light is a subset of C devised for the purpose of formal verification, albeit via the avenue of providing a language for which formal semantics is defined. At the time of publication, the authors assert, no formal verification-orientated semantics of Standard C existed, and thus the benefit provided by C-Light was that it introduced a formal semantics for a verification-orientated, dynamic memory access capable, representational subset of C.

A summary of key differences between Standard C and C-Light follows:

- As in MISRA C, variadic functions are forbidden;
- Function pointers are also disallowed;
- Abstract declarations of arguments (such as char[], and as opposed to char named[]), are also forbidden.

A key disadvantage of C-Light is that it is not a strict subset. It introduces two keywords new and delete from the C++ language (itself a non-strict superset of C), to replace the standard library functions malloc() and free(), for dynamic memory allocation and de-allocation, respectively. The justification for this change is that by incorporating these keywords, the full dynamic memory model can be reasoned about during deduction of the formal semantics of the language: the semantics given does make use of these keywords.

C-Light is, then, a C-like language that is not a strict C subset. Its changes to C mainly allow for the specification of a formal semantics for the language. Some of its changes assist in a safety-critical context.

2.4.4 Cyclone

Cyclone [29] is a dialect of C designed to remove from C many of the issues permeating the language design, including buffer overflows, unstructured switches, and freeform pointer arithmetic.

Cyclone significantly extends C in order to recoup the generality lost from its restriction of unsafe actions: for example, Cyclone adds 'never-NUL pointers'; 'fat' pointers with run-time bounds checking on pointer arithmetic; exceptions; tagged unions and more. Unlike with C-Light, the extensions to the language are not simply minor, reversible additions to facilitate reasoning about the semantics. Cyclone’s extensions are significant, conscious design decisions to improve safety whilst preserving many of the benefits of C (efficiency, manual memory management) that could be lost in a strict subset.

Significantly, Cyclone forbids the free() function for de-allocating memory, and replaces its use with a choice between garbage collection and the concept of ‘growable regions’, which automatically de-allocate any memory allocated into them. These growable regions remove the possibility of double frees and attempts to free an unallocated pointer by forcing dynamic memory into a clearly defined scope, while still allowing manual dynamic memory management and the flexibility thereof.

Cyclone was initially implemented in terms of a parser, static analyser, and translator that outputted C code. Thus, one can produce C implementations from those written in Cyclone.

Cyclone proves to be an interesting effort in providing the efficiency, flexibility and familiarity benefits of C without the safety issues. One concern is its popularity and usage in the industry.

In conclusion, we have discussed a selection of languages notable for their usage in the safety-critical domain. In particular, we have introduced the C language and its basic principles. In the next section, we introduce formal modelling techniques commonly used in the domain and prior work towards specifying modelling processes for Ada, Java, and C.

2.5 Formal Models of Safety-Critical Languages

We intend to support the proof that implementations refine specifications via the construction of formal models, thus we consider formal modelling techniques used in conjunction with the languages used to implement safety-critical systems. We first investigate modelling languages used in the field. We then explore the body of work undertaken to apply one such language, Circus, to implementations in the Ada and Safety-Critical Java languages.

We now discuss a cross-section of the range of modelling techniques available to us for the purpose of describing the state and behaviour of programs implemented in the above languages.

2.5.1 Models of state: the Z notation

One approach to modelling is to specify a program by models of its state changes, inputs and outputs. Specifically, for each operation in the program, we consider the preconditions that must hold before the operation begins and the postconditions that must hold for the state after the operation completes.

For example, consider the operation of in-place real division as a process over a dividend $k$ given a divisor $d$. The division results in a new state, where the final value of the dividend is represented by $k'$ and the new divisor is $d'$. To model this process, we consider the preconditions and postconditions. The divisor and dividend must be reals before and after the process: this yields preconditions on $k$ and $d$, and postconditions on $k'$ and $d'$. 

The divisor is unchanged: a postcondition \( d = d' \) is thus appropriate. Division by zero, a valid real, is undefined; we can support this by using a precondition \( d \neq 0 \) (division by zero is not supported) or a postcondition specifying the behaviour given division by zero (perhaps an error flag is provided, or a specific value is returned).

Finally, in order to specify that division is indeed the result of our process, we could specify as a postcondition that \( k' = \frac{k}{d} \).

We have now modelled the division process in terms of the expected state before and after that process.

One notation supporting this modelling style is \( Z \) \cite{30}. \( Z \) is based on the concept of schema: a sub-specification representing a data structure or operation thereon, which may be composed by inclusion into further schemas. A schema contains a binding of variables and a series of predicates over those variables defining the invariants, preconditions and postconditions that must hold when the schema is invoked.

We now model the real divider in \( Z \). First, we specify the data structure, \textit{Divider}, as being a schema containing the dividend \( k \) and divisor \( d \). Then, we specify \textit{Divide}, the change of state corresponding to division.

\[
\Delta \text{Divider} = \{ r, d : \mathbb{R} | d \neq 0 \}
\]
\[
\text{Divide} = \{ \Delta \text{Divider} | d' = d \land d \cdot r' = r \}
\]

The \( \Delta \text{Divider} \) notation is a shorthand for instantiating two copies of the \textit{Divider} schema: the pre-state, denoted by the \textit{undecorated} use of the schema variables, and the post-state, denoted by the addition of primes.

\( Z \) has been extended to cover paradigms and programming concepts outside of its original remit. For example, the Object-Z notation \cite{31} extends \( Z \) with notions of inheritable and composable classes, implemented as schemas for class state and operations encapsulated within a general class schema.

The \textit{ClawZ} tool we discussed earlier provides for automated translation of control-law diagrams in Simulink to \( Z \) models \cite{7}. \( Z \) is thus usable as the endpoint of an automatic formal specification process, and \textit{ClawZ} is part of the strategy by Cavalcanti et al. for proof-via-refinement of Ada implementations of Simulink models.

\( Z \) is not the only notation for modelling state-rich systems. \textit{Alloy} \cite{32} is a notation based on \( Z \), but focused on object modelling. Alloy’s ASCII-based concrete syntax is more reminiscent of programming languages than \( Z \)’s mathematical notation. It is thus, more approachable to programmers, but less concise and understandable by those used to mathematical language. In introducing Alloy, Jackson \cite{32} criticises the perceived awkwardness of composing, formatting and teaching \( Z \).

### 2.5.2 Models of behaviour: CSP

When observing the initial and final state of a program is an acceptable level of modelling, the techniques described earlier suffice. For \textit{reactive systems}—the behaviour of which must be observed and reasoned about at intermediate stages—, models of state alone are insufficient. We seek a notation for modelling, and reasoning about, this intermediate behaviour.

Hoare’s Communicating Sequential Processes (CSP) is a theory of the behaviour of reactive systems. CSP is a process algebra, specifying the communication and interaction patterns of processes in an algebraic form \cite{33}.

CSP is based on the observation of communications of processes with their environment. This is modelled using \textit{events}, whose communication must be agreed upon by both environment and process. The most basic process in CSP, \textit{Stop}, accepts the communication of strictly no events: it thus represents \textit{deadlock}.

From \textit{Stop}, one adds events via \textit{prefixing}: the process \( P = a \rightarrow \text{Stop} \) is willing to communicate the event \( a \), then deadlocks. As \( P \) is a process and the prefixing operator takes at its right-hand side a process, one can specify recursive processes easily: \( P = a \rightarrow P \) will communicate \( a \) forever.

CSP has operators that capture interactions between processes. The \textit{external choice} operator \( P \parallel Q \) will communicate any of the initial events of \( P \) and \( Q \) then behave as the remainder of the process from which that event originated. The \textit{internal choice} operator \( P \otimes Q \) defines a non-deterministic decision by the process between \( P \) and \( Q \), which may deadlock if the environment communicates the initial event available only for the process not chosen.

As a process algebra for reasoning about concurrent systems, CSP has operators for specifying forms of parallel communication and synchronisation between processes. Each form of parallelism may be reduced, via step laws, into an equivalent structure of internal and external choices; CSP does not inherently distinguish between the sequential and the parallel. As a process is defined by its communication patterns, a rigid distinction is not relevant or desired.

CSP is given meaning through three types of semantics. As a process algebra, its laws form the basis of an algebraic semantics that may be used to show equality of processes, by transforming processes to normal forms, amongst other goals. Models of denotational semantics of CSP include models based on traces, stable failures and failures-divergences. An operational semantics based partially on Plotkin’s structural operational semantics (SOS) is given by Roscoe \cite{33}.

The failures-divergences model of CSP is used within the technique of \textit{failures-divergences refinement} (FDR). A tool by this name, by Formal Systems (Europe), Ltd., allows proof of refinement of one CSP model from another. QinetiQ’s \textit{Clasp} \cite{6} tool provides tooling for the generation of CSP specifications capturing the concurrency inherent in Simulink models.

CSP forms the theoretical foundation of several programming languages, including \textit{occam} \cite{34} and, to a lesser extent, \textit{Erlang} \cite[p. 74]{55}. The \textit{Go} language is also based on the theory of CSP: the \textit{Go} Frequently Asked Questions notes that experience shows ‘the CSP model fits well into a procedural language framework’ \cite{36}.

CSP is not the only process algebra. Milner’s Calculus for Communicating Systems (CCS) \cite{37} is one such other process algebra, but lacks a suitable concept of refinement.
2.5.3 Integrated models: Circus

We have covered techniques for modelling the state of programs as well as those capturing the behaviour of reactive programs. A logical extension of these approaches is to combine both into a unified modelling technique, allowing one model to specify both aspects of a program. Also, to be able to prove refinement of one model from the other, we need a modelling notation with a formally defined refinement calculus.

Circus is such a modelling technique. It combines Z, CSP, Morgan’s refinement calculus and Dijkstra’s guarded commands. Circus’s semantics is based on Hoare and He’s Unifying Theories of Programming (UTP) [38]: a mathematical framework for achieving in computer science the unification of theories seen in the physical sciences. In UTP, programs, according to a particular theory, are predicates over an alphabet of symbols characteristic of that theory. Theories are defined by their sets of operators and constants (their signatures), and their healthiness conditions which all predicates in, or reducible to, a normal form must satisfy. UTP unifies these theories by the use of relations as a basis.

The basis of Circus on the UTP allows for the ease of extension of its semantics by combining it with other UTP-based theories. For example, CircusTime, an extension of Circus for the modelling of real-time systems, has been formalised by embedding a UTP-based time theory into Circus [39].

The choice of Circus. We use Circus in our translation process, for its seamless coverage of both the state and behaviour of untimed reactive systems, which can be extended to the cases of timed systems and other specialisations via the integration of further UTP theories.

Much prior work exists for creating Circus models of control-law diagrams and implementations in languages other than C, and verifying the latter with respect to the former via the refinement calculus. This simplifies the creation of a technique for proving refinement of specifications by implementations in C to the creation of a technique for yielding Circus models of those implementations. This process has already been specified for other languages.

Circus integrates Z and CSP, two techniques in wide industrial use. Circus is thus familiar to users with knowledge of those notations, and allows the usage of existing tools for Z and CSP in working with aspects of Circus models.

Thus, we focus on Circus due to its unified coverage of state and behaviour, its extensibility via UTP, the existence of useful related work, and its flexible combination of two popular modelling techniques.

Alternatives. Circus is not the only technique to integrate models of state and behaviour. Another is CSP-OZ [49], a combination of a restricted version of Object-Z and CSP. In addition to being based on Object-Z as opposed to plain Z like Circus, CSP-OZ takes a different approach to semantics. Whereas Circus uses the UTP as a semantic basis for integrating both Z and CSP, the meaning of a CSP-OZ specification is defined via the denotational semantics of CSP.

CSP-OZ semantically embeds Object-Z within CSP, but syntactically allows the embedding of CSP fragments into Object-Z classes as well as the combination of Object-Z classes using certain CSP operators such as choice, hiding and parallel composition. CSP-OZ has the advantage that, as its semantics is based on the CSP failures-divergences model, it reuses much of the same laws and theoretical basis as CSP. One approach to proving refinement of CSP-OZ models involves translating them to pure CSP and using the FDR tool [41].

CSP || B [42] combines CSP and the B Method, another state-based modelling technique. CSP is used to specify a ‘control executive’ driving operations specified in B. This rigid split lacks the flexibility of Circus’s more free-form combination of CSP and Z. This flexibility is important to model the low level designs of programs, which are seldom cleanly separated into controller and operation set.

Oliveira [43] notes that a full semantic combination of techniques such as Circus allows for a highly integrated concept of refinement that treats both state and behaviour equally throughout. This is an advantage of Circus over other techniques, and allows the free-form combination discussed above. However, one cannot use the existing tools for Circus’s integrated techniques directly on Circus models in the same way that the CSP || B modeller can use FDR and the set of tools for B.

Having discussed techniques for the modelling of the state and behaviour of programs, we now discuss existing work for applying those techniques to the programming languages discussed earlier.

2.6 Modelling programs in Circus

Work has been undertaken to allow the production of formal models of programs in the Ada and SCJ languages, with the case of C being our focus. The modelling processes below use Circus.

2.6.1 Ada

Circus is used by Cavalcanti et al. [6] to formalise a process for proving correctness of Ada implementations of control law diagrams. The approach of generating models of both the specification (in this case, Simulink diagrams) and the implementation (in this case, Ada) is broadly the same as that which motivates our work.

This process targets Ada programs in a subset ‘similar to SPARK Ada and with a particular architecture’. The need to subset the C language in our own work will be discussed later on.

Building on this work, Ribeiro [8] outlines in detail a process for generating Circus models of implementations of control law diagrams in Ada. This is effectively the counterpart in Ada to our work’s contribution in C; indeed, the general process
<table>
<thead>
<tr>
<th>Language</th>
<th>Advantages</th>
<th>Disadvantages</th>
<th>Modelling Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCJ</td>
<td>Modern; safety focus; automated Circus derivation.</td>
<td>Not as mature as Ada or C.</td>
<td>Zeyda et al. [44]</td>
</tr>
<tr>
<td>Ada</td>
<td>Safety focus; standardised.</td>
<td>Not as popular, in general, as C or Java.</td>
<td>Ribeiro [8] and Cavalcanti et al. [6].</td>
</tr>
<tr>
<td>C</td>
<td>Popular; efficient; simple.</td>
<td>Allows unsafe practices; low-level.</td>
<td>Focus of this work.</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of programming languages used in the safety-critical domain.

defined therein is a major influence on the process defined in the next chapter. The differences between Ada and C, however, preclude a simple translation of Ribeiro’s process.

2.6.2 SCJ

Progress has been made towards the generation of Circus models of SCJ programs [44]. The process for SCJ makes use of two variants to the core Circus notation: an object-oriented extension of Circus, OhCircus [45], and the CircusTime [39] variant, which supports timed behaviours. The combination of extended Circus variants is possible due to Circus’s Unified Theories of Programming-based semantics. Zeyda et al.’s process does not consider all aspects of SCJ: notably, capturing of memory management is left for future work.

Each SCJ program is built atop a general framework of classes and concepts: thus, a large part of this initiative involves modelling this framework in Circus. Ada and C do not use such a framework, but the process defined by Ribeiro and our work assumes a specific architecture.

An automated tool for generating Circus models of SCJ implementations—hiJaC (High-Integrity Java Applications using Circus)—was created as part of this effort. Similar work for C would logically extend our project.

2.7 Final Considerations

Table 2.1 summarises, in table format, the advantages, disadvantages and formal modelling support for Java (specifically SCJ), Ada, and C.

We have now concluded our review of the languages used for programming and formal modelling in the safety-critical domain. We have placed C, the language we target in our work, in the context of its usage in, and suitability for, safety-critical programming. We find that C has the advantages of popularity, efficiency, and relative minimalism. However, its use in safety-focused contexts is an afterthought, as opposed to being a design driver for SCJ and Ada. This makes C an interesting, challenging target for a process such as the one we formalise.

In the next section, we take inspiration from the set of C-like languages we have seen in answering the challenges C poses.
3 A Subset of C: ACDC

Having discussed safety-critical languages and made our case for specialising in the C language, we now develop and specify the process that is the primary goal of our work. We must first define exactly the language the process will accept as its input. In our exploration of languages in the safety-critical domain, we noted the existence of safety-critical subsets of each language investigated (including the MISRA guidelines for C), in addition to the existence of safety issues in standard C. It is thus a logical step to use a subset of C as our target language.

We begin this section by discussing the general approach we take in devising this subset. We then describe the criteria our subset shall fulfil, based mainly on industrial practice. Finally, we present the subset itself.

3.1 Approach

In order to subset the C language we begin with a base definition of the language and proceed to remove and refine elements of that language until a series of criteria are met.

This is similar to the approach the MISRA guidelines take, but with one key exception: our goal is to define not a series of restrictions on C but a fully specified subset language, complete with its own fully elaborated syntax definition and model semantics as defined by our translation to Circus, which form a complete and standalone language specification.

This process has various advantages. We can explain the rationale for each diversion from standard C in terms of its necessity and impact on various desired aspects of the subset. Also, by only removing aspects from standard C, we can be sure that our subset is a strict restriction of C and can be interpreted as C by a standard tool-chain. Finally, despite the ‘negative’ approach to forming the language subset, by defining it in terms of its own syntax and translation into an existing modelling language we can define our subset without reference to other standards or documents.

3.2 Criteria

Hatton [46] states that

‘Software safety, like safety in other engineering disciplines, is about the enforced avoidance of known problems, the avoidance of needless complexity and the adherence to simple and well-established engineering principles which are observed to behave safely.’

Our subset shall enforce the avoidance of known safety issues in standard C by syntactically removing the source of such problems where possible. Complexity is reduced by removing constructs unused or unnecessary in the realm of safety-critical systems programming. To inform our subset choices, we look at existing principles and subsetting efforts in the area, including the C-like languages seen previously. We now specify the criteria and their justifications.

3.2.1 Basic properties

There are certain basic, abstract properties that our subset must uphold.

Lack of unspecified and undefined behaviour. Carré and Garnsworthy [11] state that ‘to produce safety critical-software in any language, the first requirement is predictability of a program’s behaviour.’ Standard C leaves several points unspecified. Many are left to the implementation, architecture-dependent, or otherwise not reliably consistent throughout the C programming landscape. These include such concerns as: the actual domains of integral types; the effects of casting integers in a non-widening manner; and, worryingly, the evaluation order of expressions and side-effects therein. This causes unpredictability and non-determinism in C programs, where it is impossible to reason about all facets of a program (with respect to differing compiler implementations, with respect to different machine architectures, or at all).

Hatton [46] provides a good summary of the state of C as of 1995 with respect to unspecified, undefined and implementation-defined behaviour, as well as the perceived impact and possible mitigation for each. We wish to remove or restrict as much of this behaviour as possible, especially those items for which Hatton suggests full avoidance.

Support of typical usage. The subset must be broad enough to support programs that constitute examples of the type of system produced in the safety-critical domain. We evaluate this by attempting to implement a case study representative of systems found in the avionics domain: our subset should allow this. We also justify any removal of widely used features, and ensure that our changes do not make the subset too small and unrepresentative.

Minimal complexity. The more complex the subset is, the more we need to model and check for potential unsafe usage. A smaller, simpler subset will improve our ability to model the language, reduce the possibilities of unsafe behaviour, and simplify the parsing and automated transformation of subset programs.

This is a trade-off with the previous criterion: the least complex language is the empty one—not very supportive of industrial usage. Also, stripping down the language to its core removes features that simplify certain tasks (for example, eliminating switch statements would necessitate redundant if code), making the source code larger and harder to audit and reason about.
CHAPTER 3. A SUBSET OF C: ACDC

{Table 3.1: The concrete criteria for our subset, the abstract criteria to which they relate, and references to any precedents for the criteria in previous work.}

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Ill-defined</th>
<th>Complexity</th>
<th>Modelling</th>
<th>Precedents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard C</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>No dyn. memory</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>None</td>
</tr>
<tr>
<td>No union</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>[20, Rule 19.2]</td>
</tr>
<tr>
<td>Full type model</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>[27]</td>
</tr>
</tbody>
</table>

Suitability for modelling. Finally, the subset should be suited to modelling with Circus. This is assisted by reduced complexity: a simpler language is easier to translate into models. Our subset should also depend as little as possible on the behaviour of the architecture on which it runs: otherwise, we would need to model such low-level concerns.

3.2.2 Specific requirements

Following from the basic criteria, we make concrete decisions shaping the subset’s feature set.

Use Standard C. The subset must be a restriction of an ISO standard form of C. This allows it to be understood by existing tools and processes, without imposing a specific set (for example, by allowing GNU extensions we force usage of the GNU C Compiler and compatibles). This also allows us to use a formally defined and rigorous basis for our language; extensions to C may be defined in overly informal terms. Disallowing extensions also defends our subset from complexities that make the reasoning about the safety of the language more difficult.

No dynamic memory facilities. The subset must not allow the allocation, de-allocation or access via pointer of dynamic memory. C’s dynamic memory facilities have known problems we would like to avoid, which lead to the possibility of:

- Reading un-allocated memory, causing protection faults or invalid data to be read;
- Attempting to de-allocate un-allocated memory (a double free error);
- Failing to de-allocate memory after use, leading to memory leaks that may exhaust available memory.

We have seen two approaches for resolving this safety issue: garbage collection, as in Java, and language extensions, as in Cyclone. However, these have issues. Garbage collection is possible for C, but adding it would pose issues with regards to the verifiability, analysability and ability for the program to meet real-time requirements. We could extend the language, but this would break the compatibility of our subset with standard C compilers, analysers, tool-chains and methodologies; it would be difficult to argue that our process is a true verification strategy for C programs.

Both of these approaches violate the principle of safety through complexity mitigation. Also, dynamic memory access requires the modelling of dynamic memory, which would make the modelling process complex and depend on the creation of a memory model. Therefore, we remove dynamic memory management completely in our subset.

No unions. These are discouraged by MISRA-C:2012 Advisory Rule 19.2, due to the unspecified and implementation-defined behaviour arising when one union member is written and a different member is read back. In order to eliminate the risk of ill-specified behaviour, we remove unions without exception as a further restriction on that mandated by MISRA.

Fully-specified type model. The primitive types of C are defined in terms of their minimum size. This allows the range of these types to adapt to changes in computer architectures. The int type is at least 16 bits long; on 16-bit architectures, it is usually that size. However, on 32-bit and 64-bit IBM PC-compatible architectures, it is conventionally extended to 32 bits.

While this allows the same set of types to adapt to expanding data word sizes, it poses the disadvantage that the range of types can change when porting a C program across architectures. A program designed for a system whose int is 64 bits long may crash when ported to a 32-bit architecture, as a large proportion of the previous range of int is now disallowed.

In order to specify the model transformation of the subset, we need to specify the exact ranges of these types. However, since the ranges constitute an ill-specified part of the C specification, we can make this necessity part of the criteria.

Summary of criteria. We summarise the concrete criteria, and their relationship to the abstract criteria, in Table 3.1. We now define a small subset of C, which we call ACDC (A Cut Down C).

3.3 ACDC

ACDC is a C subset that removes unsafe syntax and semantics, while specifying items left undefined by the standard. The resulting language will be predictable, easy to model, and meet our subset criteria. An overview of ACDC follows:
• Dynamic memory access is forbidden, as per MISRA Directive 4.12. We remove pointers and any operators for working with them (the dereferencing operator, \* addressing operator \&, and struct member dereferencing operator ->);
• Arrays are allowed, but may only be handled via array syntax and not pointer operations; unlike Standard C, array subscript notation is not ‘syntactic sugar’ over pointer arithmetic;
• Unions are not available, as per MISRA Rule 19.2;
• Other features removed for safety reasons include variadic arguments (MISRA Rule 17.1), goto statements, and abstract declarations in parameter lists (as in C-Light);
• Bitwise operators have been removed, due to the difficulty of modelling bitwise operations;
• Some syntactic forms have been removed or tightened to improve safety, including mandatory else branches and compound statements in if statements. Assignments, including increments and decrements, are only permitted at the statement level and not in expressions.

In addition, ACDC specifies aspects of the language usually left to the C compiler to define. These include evaluation order and the size of primitive types.

3.3.1 Syntax

ACDC is a strict subset of C90 syntactically; the reason we choose C90 here is that its syntactic separation of declarations and code simplifies the construction of the procedure for generating C models. This effectively makes it a restriction of the syntax offered in Kernighan and Ritchie \[22, App. A\]: our syntax specification is adapted from that source. Much of ACDC’s restrictiveness comes from the changes made to the syntax. There are non-syntactic restrictions that we specify later on.

For space reasons, we omit the full grammar of ACDC; it is, effectively, that of C90 with the changes mentioned here. Later, we provide an abstract syntax encoding in Haskell that may also be used as a guide to ACDC syntax. Here, we shall summarise the changes made to said syntax and provide a sketch of the language.

Pointers. ACDC does not, syntactically, support pointers. The removal of one of the characteristic concepts of the C programming language is justified by the necessity of removing completely the unsafe notion of dynamic memory allocation, destruction and arbitrary access.

One vestige of Standard C’s pointers exists in the form of arrays and array subscript notation. In C90, the following three expressions are equivalent \[22, A8.6.2\]:

\[
\begin{align*}
((E_1 + E_2)) &\quad /*\text{ dereference pointer } E_2 + E_1 */ \\
E_1[E_2] &\quad /*\text{ access } E_2\text{th element of } E_1 \text{ (zero indexed) } */ \\
E_2[E_1] &\quad /*\text{ since addition is commutative, we can exchange } E_2 \text{ and } E_1 */ 
\end{align*}
\]

Since the removal of pointers has changed the nature of array sub-scripting from being syntactic sugar over pointers, we instead treat the subscript syntax as its own special case where the left hand side must be an array and the right an integer. This makes the third form above invalid.

Bodies of selections and iterations. In standard C, each selection-statement and iteration-statement has as its body a statement, which is responsible for allowing code such as

\[
\begin{align*}
\text{if } (a == 3) \\
a & = 4;
\end{align*}
\]

The single statement \(a = 4\) is the body of the selection statement if \(a == 3\). This is a potential hazard, as programmers (especially those without experience in C) may assume that extending the selection to

\[
\begin{align*}
\text{if } (a == 3) \\
a & = 4; \\
b & = 5;
\end{align*}
\]

will cause the statement \(b = 5\) to be part of the selection body (it will not, as it is syntactically the statement following the selection). Thus, ACDC prohibits this at the syntactic level, for reasons of code safety as well as simplifying the syntax by eliminating unnecessary productions. The case of a single statement body can easily be transformed into that of a compound statement body holding said statement.

Assignments. We restrict assignments so that they may not appear inside arbitrary expressions. This reduces the number of cases of side-effects inside expressions, which prove difficult for modelling and reasoning. We also restrict the potential assignment targets (left-hand values in assignments) to identifiers, array subscripts of other targets, and struct members of other targets, as these are the only expressions that make sense as a target of an assignment given ACDC’s lack of pointers.

Operators. As well as pointer operators, we remove bitwise operators (bit-shifts \(<<\text{ and }>>\), bitwise negation \(\sim\), exclusive OR \(^\oplus\), and their equivalents in assignments) due to the difficulty of providing semantics to them in terms of the Z notation. As we do not allow incrementing and decrementing assignments (\(++\text{ and }--\)) inside expressions, there is no difference in semantics between their prefix and postfix forms, so we omit the prefix forms.
corresponding Circus

Therefore, in until

has no dynamic memory assignment, there is no pointer type or

ACDC

int

the body of tools and techniques built up for standard C.

We have introduced and justified ACDC, a conservative C subset. ACDC provides the essence of C, while greatly reducing the safety concerns and complexity of C. It provides us with a manageable and safe target for our work, while still working with the body of tools and techniques built up for standard C.

In the next chapter, we begin formalising the process that will take as input a program in ACDC and yield as output its corresponding Circus notation model.

### 3.3.2 Non-syntactic concerns

ACDC is defined by both its restrictions to the syntax of Standard C, as well as its strengthening of the specification of how programs are to be interpreted.

**Types.** To satisfy the type width criterion, ACDC defines the primitive type widths as follows: char is 8 bits, short is 16 bits, int is 32 bits, long is 64 bits, float is IEEE 754 single-precision (32 bits), and double is IEEE 754 double-precision (64 bits). As ACDC has no dynamic memory assignment, there is no pointer type or size_t; there is also no long long as this is not part of C99. These widths are chosen because:

- The integer widths correspond to the LP64 model used in 64-bit architectures on Unix operating systems [48]; this model is representative of typical type width assignments used today;
- Every common byte-width of integer is represented once without redundancy;
- char is assigned its conventional width of eight bits.

One solution to the issue of C type ranges changing across platforms is to use the C99 <stdint.h> header file, which defines typedefs offering explicit ranges retained across architectures. For example, in ACDC, the typedefs uint8_t and int64_t would map to unsigned char and signed long. Since ACDC allows both typedefs and the C preprocessor, we can allow this convention to be used in ACDC programs such that they can be ported to models other than LP64 without modification.

**Time.** Neither the C language nor the standard library provide primitives or functions for delaying computation. This contrasts with Ada, which provides a delay statement, and Java, which provides Thread.sleep() in the standard library. This is a problem: our case study must include delay statements in order to allow the frames to execute at the correct time. We thus allow the use of the POSIX function unsigned int sleep(unsigned int seconds); within the ACDC code permitted as input to the process. (This is, properly speaking, an extension to ACDC.)

The case study needs the ability to delay until a given time, as opposed to a fixed delay period. While Ada 95 has a delay until variant of the delay statement, for ACDC we can use the C standard function time_t time(time_t *tloc). This uses a pointer, which we forbid, so we always take the argument to be 0. We also assume that this function always returns seconds: thus, sleep(next_time - time(0)) delays a program until next_time. This approach can fail; when the return value of time crosses the upper limit of time_t, the value will overflow and cause the above time calculation to yield an unwanted result.

Because ACDC has no casting, the type of time_t must be the same as that of the parameter to sleep, which is unsigned int. Therefore, in ACDC, time_t is taken to be an unsigned 32-bit integer, which has a wrap-around time in 2106. A disadvantage is that time-stamps may not represent times prior to midnight on the first of January 1970 using time_t.

### 3.4 Final considerations

We have introduced and justified ACDC, a conservative C subset. ACDC provides the essence of C, while greatly reducing the safety concerns and complexity of C. It provides us with a manageable and safe target for our work, while still working with the body of tools and techniques built up for standard C.

In Table 3.2 we summarise the main advantages and disadvantages of each C-like language we have seen, including Standard C and ACDC.

In the next chapter, we begin formalising the process that will take as input a program in ACDC and yield as output its corresponding Circus notation model.
4 From *ACDC* to *Circus*

We now formalise the process of generating *Circus* models of valid *ACDC* programs. We first discuss the technique we use for the formalisation, in section 4.1. In section 4.2, we detail the assumptions that must hold of input programs. We then provide an overview of the process in section 4.3, before formalising it starting from section 4.5.

4.1 Technique

We define our process as a prototype literate program in the functional language *Haskell* \[\text{HS}\]. We take this approach over that of using an abstract notation, such as Z, for several reasons:

**Type-checking.** Using a strongly typed language such as *Haskell* allows us to use its automated type checking facilities to defend against a class of potential errors in the process.

**Rapid evaluation.** We can evaluate the process by executing it directly on the case studies, type-checking, and conventional software testing strategies. This evaluation can occur at the same time as the code is written, with minimal manual effort.

**Automation.** The existence of the process in code form makes further automation of the process easy: the process can already automatically output a *Circus* model in certain cases.

**Algebraic data types.** *Haskell*’s data type syntax allows us to capture the syntax and mathematical constructs of the process in a natural manner, and operate on those types in concisely and effectively.

The main body of this chapter is a pre-processed form of the source code of the process: test cases, module definitions and comments have been stripped, and the code has been typeset using the *Hs2TeX* processor. More information on the techniques used is given in the appendices. We use several additional notational conventions:

- We denote the fully qualified name of a variable in the output by joining each element of the name with a double colon. The local variable `var` of the function `func` thus has the full name `var::func`.

- We substitute mathematical notation for some common operators in *Haskell*, to improve readability. For example, function composition is represented in *Haskell* code as `.` but appears in our listings as `@`; the `elem` function is likewise represented with `∈`.

- When building the *Circus* model, we define many custom operators in *Haskell* that are similar or identical in appearance to the *Circus* operators they imitate. For example, relational overrides are performed by the `@` operator, and assignment guarded commands appear as `=:`. This is to achieve a balance of conciseness and familiarity.

To achieve a concise translation process, we make use of extensions to the standard *Haskell* language, as well as idioms and syntactic forms peculiar to *Haskell* itself. Where appropriate, we discuss these as we introduce and use them. Appendix E contains a review of some of those syntactic forms, in ‘cheat sheet’ form.

4.2 Assumptions

We make assumptions about the input to the process: necessary assumptions that must hold for any such process to work, and simplifying assumptions that we make to make the process easier to define. We expect the input to conform to a particular architecture, which is discussed in this section.

**Necessary assumptions.** These are:

1. The input is valid, well-formed *ACDC* (effectively MISRA-C:2012 rule 1.1);

2. Any preprocessing needed has already happened, and there are no preprocessor directives in the input. Thus, the input can be processed as a unit without reference to header files and other included code;

3. There are no name conflicts (that is, non-static top-level declarations have unique names across all translation units), as in MISRA-C:2012 rules 5.1 and 5.2.
Simplifying assumptions. These are:

1. Our abstract syntax for ACDC has simplifications, such as the merging of some productions, which have no safety rationale and no loss of generality from the concrete syntax. We assume that there is a translation process that correctly maps from the concrete syntax to the abstract syntax in these cases;

2. The translation units are presented to the process in a concatenated, preprocessed format with topological ordering: each external reference reached by a unit has already been defined in a previously seen unit;

3. Static unit-level variables must not conflict in name with any other unit-level variables. This allows us to consider static variables as if they were normal global variables;

4. There are no possible overflows or assignments of out-of-range values in the numeric operations of the program. This allows us to discard the ACDC integral types and model only using the Z counterparts;

5. There are no possible floating-point precision errors in the numeric operations of programs. This allows us to avoid modelling the floating-point calculations in detail and model only using real types as modelled in Z. Although Standard Z does not include the real numbers as a type, a model of real numbers in Z is available, for instance, as part of the ProofPower Z encoding [50, sec. 9.1.10].

6. There is no use of ACDC’s short-circuit semantics: for every logical AND and OR expression, the effect on the program if both sides are evaluated must be the same as if only the left side was evaluated. This allows us to avoid implementing the short-circuit semantics.

7. As in Ribeiro [8], the set of statements directly or indirectly accessing time-related variables and the set of all other statements are disjoint. The only effect on the program’s behaviour that can be induced by operations reaching variables used in time() and sleep() calls is precisely the synchronisation between the processes in the cyclic executive; any such variables cannot participate directly or indirectly in any other program behaviour. This facilitates our implementation of the temporal information reduction step of the program.

8. We assume return statements occur only at the end of a function. This allows us to model them as just the setting of the return value of the function, if it has one.

Assumed architecture. We assume the program is a frame-based cyclic executive. This architecture is common in the field of embedded safety-critical systems, and allows for the specification of concurrency and task scheduling in the context of a language such as ACDC that does not inherently support either.

The input program shall have one or more functions designated as ACDC main functions, which are the entry points into concurrent processes. This is distinct from the main() function expected by C compilers. The program may contain main() functions invoking the ACDC main functions, but we do not consider them here.

Each main function invokes a task scheduler which, for each given frame, determines the functions to be executed. The processes defined by the main functions and their executed code communicate through shared, global variables, and synchronisation occurs via the static scheduling of tasks into frames such that no task depends on a result calculated in the same frame.

4.3 Overview

We define the translation process as a composition of functions, representing the following phases:

Flattening scopes ACDC is a lexically scoped language, and each new scope may define variables with names that shadow previous definitions for the scope’s lifetime. To make the modelling process easier, we perform scope-flattening: each variable is assigned a name unique across all scopes, so we need not consider which version of a variable is in scope when we see a reference to that variable.

Capturing declarations We extract declarations of interest in the later stages of the technique from the flattened program, creating a construct that holds all global constants and functions in the program.

Abstracting temporal information Circus does not model the explicit passage of time, so we must remove the effect of concrete temporal activities such as time() and sleep() functions. This phase eliminates, where possible, statements dependent on this temporal information, so that we can replace it with the high-level cyclic executive pattern it represents later in the modelling process.

Capturing functions called by the ACDC main functions We create a reachability graph, capturing dependencies between functions, to deduce which functions are used by which main functions.

Capturing variable usage We create a construct capturing which functions access which variables, and whether the access was a read or a write. This is used when we build the state schemas for each process, and decide which shared variables must be synchronised with their channels during each frame.
Retrieving struct definitions  To allow struct types to be encoded correctly in the Circus model, we need to be aware of each struct type in the program. This process extracts all struct definitions from the program so they can be retrieved more easily in the final stage.

Retrieving variable and function types  To ease the resolving of the types of variables and functions, we extract from the flattened program a mapping from variable and function names to types.

Building the Circus model  We finally use the constructs produced in the previous phases to construct the Circus model.

In addition to the phases that constitute the process as formalised, we also use a basic ACDC parser, based on existing Haskell C language libraries, and a Circus emitter. There is also a degree of pre-processing required on the input to the process. We require the source code to be collated into one file and preprocessed before admission, and the creation of a context file containing a partially defined input context (a CircusifyInput, the format of which is described later) containing the frame count, frame counter variable, main function names, external variables, and frame mapping.

We now describe the above phases in more detail. Figure 4.1 gives a pictorial overview of these functions.

Flattening scopes.  In ACDC, the mapping from variable names to variables may be overridden wherever a new compound statement is introduced, for the lifetime of that compound statement.

Example.  Consider the following fragment of ACDC.

```c
int i = 0;
{
    int i = 5;
    f(i);
}
g(i);
```
The brace-delimited compound statement introduces a scope where \( i \) is remapped from the variable with value 0 to the variable with value 5. Thus, \( f(i) \) is equivalent to \( f(5) \), and \( g(i) \) expands to \( g(\theta) \). (End of example.)

To remove the need to consider the implicit mapping from variable names to actual variables created by the nesting of scopes in ACDC, we make the role of scopes explicit by renaming parts of the ACDC code. We also collapse a program of zero or more translation units into a single list of external declarations, removing the distinction between units that is not useful for the modelling process. This phase is performed by the function `flatten`, of type `Program LocalName \rightarrow [External FullName]` and defined in section 4.5.

Capturing declarations. We next capture the function and constant declarations in the program. The output of this phase is a construct, `Decl`, that contains profiles for every function and constant in the ACDC program: these profiles contain the declaration’s name and the constant or function declaration itself. The `Decl` construct is used in the later phases, and is one of several views into the program constructed to simplify the translation process. This phase is performed by the function `declCapture`, of type `[External FullName] \rightarrow Decl` and defined in section 4.6.

Abstracting temporal information. Circus does not model the passing of real time. As our case studies include `sleep` statements, in order to achieve a standard Circus model we need to abstract over this temporal information.

One method of systematically removing the temporal information involves `decomposition slices` [51], a technique for decomposing a program into multiple components. The result of this step is the complement with respect to all variables in the program, with the criteria that the global variables `start_time` and `end_time` are to be excluded. This phase is performed by the function `abstractTime`, of type `Decl` `\rightarrow Decl` and defined in section 4.7.

Capturing functions called by the ACDC main functions. We next capture the dependencies between ACDC functions: for each function, we are concerned with the functions called inside it. This phase computes a reachability graph, a relation from function identifiers to function identifiers, represented by the type `RGraph`. This phase is performed by the function `reachability`, of type `Decl` `\rightarrow RGraph` and defined in section 4.8.

Capturing variable usage. The final Circus model must take into account the usage of variables. To allow this, we need to capture the set of variables used in each function. We map each function, by means of its identifier, to a set of variable records containing pairs of variables used in that function, and whether they were written or read. This phase is performed by the function `variableUsage`, of type `Decl` `\rightarrow VUsage` and defined in section 4.9.

Retrieving struct definitions. The definitions of each struct type in the program need to be collected, so we can construct the necessary Circus definitions to allow them to be used in the program. This phase yields a list of pairs mapping struct type names to lists of their member definitions. This phase is performed by the function `getStructs`, of type `[External FullName] \rightarrow StructMap` and defined in section 4.10.

Retrieving variable and function types. We take the types of every variable in the program, as well as the return types of functions, to allow the Circus model generator to look up the types of variables and return values when constructing their declarations. This phase produces a list of pairs mapping variable full-names to their types, and function names to their return types. This phase is performed by the function `getTypes`, of type `[External FullName] \rightarrow TypeMap` and defined in section 4.11.

Building the Circus model. Finally, we build the Circus model corresponding to the ACDC input. The final phase takes a construct of type `CircusifyInput` containing the results of the previous phases and contextual information such as the frame count and set of main programs, and returns the syntax tree of the corresponding Circus model. This phase is performed by the function `circusify`, of type `CircusifyInput \rightarrow Circus` and defined in section 4.9.

Before we formalise the translation process, we discuss the syntax trees of ACDC and Circus used therein.

4.4 Syntax and general constructs

Our translation process is primarily based on the abstract syntax tree representations of both ACDC programs and Circus models, which are given in this section. In subsection 4.4.1, we provide an abstract syntax tree in Haskell for ACDC. We then, in subsection 4.4.2, provide one for Circus, including a subset of the Z notation. Finally, in subsection 4.4.3 we provide types for general constructs used in both syntax trees.

4.4.1 ACDC

Our abstract syntax is too ambiguous to be suitable for direct parsing: it does not encode precedence levels for expressions. When parsing, we use an existing Haskell C parser, and map from its syntax tree to ours.

Most syntactic constructs are qualified by a type parameter encoding the naming style used in that construct and its children. For example, the type `Program a` is parameterised over the type `a`: the complete type `Program LocalName` denotes an ACDC program whose declarator identifiers are local names, whereas `Program FullName` is a program whose identifiers are full names.
Programs. An ACDC program, represented by the type `Program`, contains a list of translation units. The record notation used below automatically constructs a function, `programUnits`, that yields a program’s units.

```haskell
newtype Program a = Program { programUnits :: [TranslationUnit a] } deriving Show
```

A `newtype` definition introduces a new type, with the name on the left hand side and the definition on the right. The term `Program` on the right hand side is a type constructor, and may be used to produce a value of the new type given a value of the type as given in the record notation. In this instance, given a list of translation units `xs`, the expression `Program xs` yields the corresponding program.

The `deriving` term instructs the compiler to generate code to make `Program` an instance of the type class `Show`. This allows the function `show` to be applied to a `Program`, returning a string describing the program.

Other type classes we derive later on include `Read`, which allows its instances to be read from strings; `Eq`, which allows instances to be compared for equality; `Enum`, which is used for types that may be mapped to the natural numbers, and `Bounded`, which is for types that have lower and upper bounds.

Translation units. A translation unit is a non-empty sequence of external declarations. We use the `Seq1` type, defined later, to enforce the non-emptiness constraint.

```haskell
type TranslationUnit a = Seq1 (External a)
```

The `type` definition constructs a type synonym, which effectively allows the type on the left hand side of the definition to be used as an alias for the type on the right hand side. Thus, a variable of type `TranslationUnit FullName` is actually a variable of type `Seq1 (External FullName)`.

External declarations. These are function definitions and prototypes, declarations, and enum and struct specifiers.

```haskell
data External a = EDFunction (Function a) 
  | EDP prototype (Signature a) 
  | EDD eclaration (Declaration a) 
  | EDDEnum EnumSpecifier 
  | EDStruct StructSpecifier deriving (Read, Show, Eq)
```

We use `data`, the final form of type definition in Haskell. This generalises `newtype` to allow product and sum types, which are similar in concept to the struct and tagged union types of C. Here, we use a sum type to allow an external declaration to be constructed from one of five constructors, each capturing a different external declaration type.

Function signatures. These are declaration specifiers, a name, and zero or more parameters, and form a product type. We use record notation to allow the specifiers, name and parameters to be retrieved and updated easily.

```haskell
data Signature a = Signature { signatureSpecs :: DeclarationSpecifiers, 
                              signatureName :: Identifier, 
                              signatureParameters :: [Parameter a] } deriving (Read, Show, Eq)
```

We use `data`, the final form of type definition in Haskell. This generalises `newtype` to allow product and sum types, which are similar in concept to the struct and tagged union types of C. Here, we use a sum type to allow an external declaration to be constructed from one of five constructors, each capturing a different external declaration type.

Function signatures. These are declaration specifiers, a name, and zero or more parameters, and form a product type. We use record notation to allow the specifiers, name and parameters to be retrieved and updated easily.

```haskell
data Declaration a = Declaration { declarationSpecs :: DeclarationSpecifiers, 
                                    declarationName :: a, 
                                    declarationInitialiser :: Maybe (Initialiser a) } deriving (Read, Show, Eq)
```

Declarations. Declarations contain declaration specifiers, a name, and an optional initialiser. We represent an optional value by using the `Maybe a` type, where `a` is the type of the value should it exist. This can either be `Nothing`, representing an absence of a value, or `Just a`, representing a presence.

Aside (Maybe). The `Maybe` convention for representing values which may or may not exist is used throughout our translation process. It is an instance of the type class `Functor`: this means that, for a type `Maybe a`, functions `f` of type `a -> b` may be lifted to a function `g` of type `Maybe a -> Maybe b`. For `Maybe`, `g` maps `Nothing` to `Nothing` and `Just k` to `Just (f k)`. This property of `Maybe` will be explored later on, when we use it in the formalisation. *(End of aside.)*

Initialisers. We model these using `n`-ary trees: initialisers can be a single initialising expression or a list of initialisers, allowing for arbitrary levels of nesting to match the shape of the variable being initialised. We elaborate `Tree` later.

```haskell
type Initialiser a = Tree (Expression a)
```
Function definitions. These contain a signature and a body, which is a compound statement.

```haskell
data Function a = Function { functionSignature :: Signature a, functionBody :: Compound a }  
    deriving (Read, Show, Eq)
```

Declaration specifiers. Unlike C, ACDC has only two specifiers for declarations: whether or not the declaration is a constant, and its type.

```haskell
data DeclarationSpecifiers = DS { dsConstness :: Constness, dsType :: Type }  
    deriving (Read, Show, Eq)
```

Types. We include `void` types here, although they may only be used to denote a lack of return value. The optional parameter for arrays is the array size.

```haskell
data Type = Void  
    | Short Signedness | Int Signedness | Long Signedness  
    | Double  
    | Char Signedness | Enum Identifier  
    | Array Type (Maybe Int) deriving (Read, Show, Eq)
```

The type `Constness` captures whether a declaration is constant, or a variable; `Signedness` captures whether an integral type is signed or unsigned.

```haskell
data Constness = Const  
    | NotConst deriving (Read, Show, Eq)
```

```haskell
data Signedness = Signed  
    | Unsigned deriving (Read, Show, Eq)
```

Struct specifiers. These contain an identifier, which names the struct type for later use, and a non-empty sequence of struct declarations:

```haskell
data StructSpecifier = StructSpecifier { structSpecifierId :: Identifier  
    , structSpecifierDecls :: Seq1 StructDeclaration } deriving (Read, Show, Eq)
```

Struct declarations. These contain a type and identifier. It makes no sense to have a constant struct member; thus, we do not use `DeclarationSpecifiers` here.

```haskell
data StructDeclaration = StructDeclaration { sdType :: Type, sdId :: Identifier }  
    deriving (Read, Show, Eq)
```

Enumeration specifiers. Similarly to struct specifiers, these contain an identifier labelling the enum, and a non-empty sequence of enumerators.

```haskell
data EnumSpecifier = EnumSpecifier { enumId :: Identifier  
    , enumEnumerators :: Seq1 Enumerator } deriving (Read, Show, Eq)
```

Enumerators. An enumerator may be explicit, and thus have a specific value assigned, or implicit, and thus take the value of the preceding enumerator (or zero, if it is the first enumerator). For both, the automatically generated function `enumeratorId` retrieves the identifier of the enumerator.

We give the `unit type ()` as the parameter to `Expression`. This signifies that enumeration expressions may not contain identifiers, as the unit type can hold no information.

```haskell
data Enumerator = EnumImplicit { enumeratorId :: Identifier }  
    | EnumExplicit { enumeratorId :: Identifier  
    , enumeratorExpression :: Expression () } deriving (Read, Show, Eq)
```

Parameters. A function’s formal parameter contains declaration specifiers and an identifier.

```haskell
data Parameter a = Parameter { parameterSpecs :: DeclarationSpecifiers  
    , parameterName :: a } deriving (Read, Show, Eq)```
Statements. This AST simplifies the various productions for statements in the ACDC grammar into one production, to make the traversal of statements easier later.

```haskell
data Statement a
    = NopStm
    | CompStm (Compound a)
    | ExprStm (Expression a)
    | Assignment (Assignment a)
    | Return (Maybe (Expression a))
    | DoWhile (Compound a) (Expression a)
    | For (Assignment a) (Expression a) (Assignment a) (Compound a)
    | IfThenElse (Expression a) (Compound a) (Compound a)
    | Switch (Expression a) (SwitchCases a)
    | While (Expression a) (Compound a)
  deriving (Read, Show, Eq)
```

Compound statements. These introduce a new scope, and contain lists of declarations and statements.

```haskell
data Compound a = Compound { compoundDeclarations :: [Declaration a], compoundStatements :: [Statement a] } deriving (Read, Show, Eq)
```

Switch cases. These are default cases or guarded cases (which recursively include switch cases).

```haskell
data SwitchCases a = Default (Compound a)
    | Case (Expression a) (Compound a) (SwitchCases a)
  deriving (Read, Show, Eq)
```

Assignments. Value-assignments and self-assignments—the ++ and -- operators—are handled here.

```haskell
data Assignment a = AssignSelf AssignSelfOperator (AssignmentTarget a)
    | AssignValue AssignValueOperator (AssignmentTarget a) (Expression a)
  deriving (Read, Show, Eq)
```

Expressions. We treat these as one production, which is ambiguous but simplifies traversing expressions.

```haskell
data Expression a
    = ArraySubscript (Expression a) (Expression a)
    | ConstantExpr Constant
    | CallExpr (FunctionCall a)
    | IdExpr a
    | Infix InfixOperator (Expression a) (Expression a)
    | Prefix PrefixOperator (Expression a)
    | StringExpr String
    | StructMember (Expression a) Identifier
  deriving (Read, Show, Eq)
```

Function calls. A function call is a function identifier and a list of zero or more argument expressions.

```haskell
data FunctionCall a = FC { fcFunction :: Identifier, fcArguments :: [Expression a] }
  deriving (Read, Show, Eq)
```

Assignment targets. These are encoded as below.

```haskell
data AssignmentTarget a
    = AssignArray (AssignmentTarget a) (Expression a)
    | AssignID a
    | AssignStruct (AssignmentTarget a) Identifier
  deriving (Read, Show, Eq)
```
Constants. These can be integer, characters, floating point numbers, or enumeration values.

```
data Constant = CInt Int | CChar Char | CFloat Float | CEnum Identifier
  deriving (Read, Show, Eq)
```

Identifiers. Identifiers not renamed—enum constants, function names, struct members, and so on—are modelled as strings.

```
type Identifier = String
```

We now define the \textit{ACDC} operators. Recall that these are a strict subset of those of C.

Self-assignment operators. \textit{AssignSelfOperator} is an enumeration of the operators available in \textit{ACDC} to perform self-assignments (increments and decrements):

```
data AssignSelfOperator = Decrement | Increment deriving (Read, Show, Eq)
```

Value-assignment operators. \textit{AssignValueOperator} is an enumeration of the operators available for assignments using an external value. This includes 'normal assignment' (\textit{AOpEquals}) as well as other forms of assignment in which the assigned value is a function of the original and given values.

```
data AssignValueOperator = AOpAdd | AOpDivide | AOpEquals | AOpModulus
  | AOpMultiply | AOpSubtract deriving (Read, Show, Eq)
```

Infix operators. \textit{InfixOperator} is an enumeration of the binary operators available in \textit{ACDC}:

```
data InfixOperator = Add | Divide | Equals | Greater
  | GreaterEquals | Less | LessEquals | LogicalAnd
  | LogicalOr | Modulus | NotEquals | Multiply
  | Subtract deriving (Read, Show, Eq)
```

Prefix operators. \textit{PrefixOperator} is an enumeration of the prefix operators available in \textit{ACDC} (of which there is only one, but we provide room for expansion in future work):

```
data PrefixOperator = LogicalNot deriving (Read, Show, Eq)
```

We now define our abstract syntax for \textit{Circus}, including a subset of the Z notation used therein.

\subsection{Circus}

Following is a module encoding the \textit{Circus} syntax in the form of a set of Haskell algebraic data types. Not all of the \textit{Circus} syntax is provided: thus, we properly define a subset of \textit{Circus}. Where applicable, we use the same naming conventions and order of elaboration for the syntax as those given in Oliveira [43].

\textbf{Circus models.} A \textit{Circus} model is a potentially empty list of \textit{Circus} paragraphs.

```
type Circus = [CircusPar]
```

\textbf{Circus paragraphs.} A \textit{Circus} paragraph is either a Z paragraph, a channel declaration, or a process declaration. We do not include channel set paragraphs.

```
data CircusPar = CPar ZParagraph
  | CChannel (Seq1 SimpleCDecl)
  | CProcDecl ProcDecl
  deriving (Read, Show, Eq)
```

\textbf{Channel declarations.} In our \textit{Circus} subset, these may either be untyped channels or typed channels.

```
data SimpleCDecl = UntypedChannel { channelNames :: Seq1 ZIdentifier }
  | TypedChannel { channelNames :: Seq1 ZIdentifier
    , channelType :: ZExpression
  } deriving (Read, Show, Eq)
```
Process declarations. We do not support generic process definitions; as such, processes are defined as having a name and definition.

\[
\text{data ProcDecl} = \text{Process ZIdentifier ProcDef deriving (Read, Show, Eq)}
\]

Process definitions. Process definitions can only be simple process definitions in this syntax: that is, there are no indexed or parameterised processes, as they are not used in the transformation process.

\[
\text{data ProcDef} = \text{SimpleProcDef Proc deriving (Read, Show, Eq)}
\]

Processes. We support explicit processes, compound processes created from the parallel composition of existing processes and synchronised by their given channel sets, processes that are the hiding of a channel set from another process, and a named reference to a process defined elsewhere. The parallel composition syntax is the set-explicit form used by Ribeiro.

\[
\text{data Proc} = \text{ExplicitProc [PPar] ZSchemaExp [PPar] Action}
\mid \text{PParallel (Seq1 (Proc, CSExp))}
\mid \text{Hide Proc CSExp}
\mid \text{PNamed ZIdentifier deriving (Read, Show, Eq)}
\]

These are the only process operators we need to create our models of ACDC programs.

Process paragraphs. Z paragraphs and namesets are not used in the process, so we only support named actions in process paragraphs.

\[
\text{data PPar} = \text{PParAction ZIdentifier ParAction deriving (Read, Show, Eq)}
\]

Paragraph actions. We need only use direct actions in this work.

\[
\text{data ParAction} = \text{PAction Action deriving (Read, Show, Eq)}
\]

Actions. An action is either a guarded command, an action named by its identifier, or a CSP action. We use a language extension to allow string literals to represent actions; the literal "string" with type Action is equivalent to AIdentifier"string".

\[
\text{data Action} = \text{ACommand Command} \mid \text{AIdentifier ZIdentifier} \mid \text{ACSPAction CSPAction deriving (Read, Show, Eq)}
\]

Actions form a monoid, a data type with an empty value and an associative appending operation \([\_\_\_]\). In this case, the empty value is Skip, and the associative operation is sequential composition.

CSP actions. The CSP actions we consider are Skip, prefixing of actions by communications, sequential composition of actions, interleaving of actions, the application of an action to one or more parameters given as Z expressions, and fixed-point recursion. We use the explicit-set interleaving syntax from Ribeiro, where each interleaved action is paired with the alphabet of variables on which it synchronises.

\[
\text{data CSPAction} = \text{Skip}
\mid \text{CPrefix Comm Action}
\mid \text{CSeqCompose Action Action}
\mid \text{CInterleave (Seq1 (Action, NSExp))}
\mid \text{CActionApply ParAction (Seq1 ZExpression)}
\mid \text{CFixedPoint ZIdentifier Action deriving (Read, Show, Eq)}
\]

Communications. These consist of a Z identifier naming the channel partaking in communication, and zero or more communication parameters defining inputs and outputs from that channel.

\[
\text{data Comm} = \text{Comm \{commIdentifier :: ZIdentifier, commParameters :: [CParameter]\} deriving (Read, Show, Eq)}
\]

Communication parameters. In our Circus subset, these are either inputs or outputs.

\[
\text{data CParameter} = \text{Input ZIdentifier} \mid \text{Output ZExpression deriving (Read, Show, Eq)}
\]
Guarded commands. Not all of the commands from Circus are given here: only those used in the process—assignment, if, var, val and res—are present. The latter three are combined into CVarBlock, to allow a single command to bind multiple types of variable, as in Ribeiro.

```
data Command = CAssign (Seq1 ZIdentifier) (Seq1 ZExpression) | CIF (Seq1 GAction) | CVarBlock [(VarType, ZDeclPart)] Action deriving (Read, Show, Eq)
```

Variable types. These are var, val, and res. We do not include vres, as it is unused in our translation process.

```
data VarType = Var | Val | Res deriving (Read, Show, Eq)
```

Guarded actions. A guarded action is a guarding predicate followed by an action. We do not include external choice of guarded actions in this subset.

```
data GAction = GAction ZPredicate Action deriving (Read, Show, Eq)
```

Channel-set expressions. For our purposes, a channel-set expression is a list of Z identifiers.

```
newtype CSExp = CSExp [ZIdentifier] deriving (Read, Show, Eq)
```

Name-set expressions. A name-set expression is also here restricted to a list of Z identifiers.

```
newtype NSExp = NSExp [ZIdentifier] deriving (Read, Show, Eq)
```

To define the Circus syntax fully, we also implement a working subset of the Z notation in a Haskell abstract syntax tree format. The syntax used here is based on that given by ISO [55]. This is a deviation from the Circus syntax given by Oliveira et al., which is based on the Z reference manual. This is a pragmatic encoding of Z, only embedding aspects needed in the project and containing extensions to the standard syntax needed for a straightforward elaboration of Circus models.

Z paragraphs. Here, a Z paragraph is taken to be a Z axiom (containing a schema text), or a free type (consisting of a free type identifier and zero or more branches).

```
data ZParagraph = Axiom ZSchemaText | FreeType ZIdentifier [ZBranch] deriving (Read, Show, Eq)
```

Z free type branches. These consist of a branch identifier, and an optional Z expression.

```
data ZBranch = ZBranch ZIdentifier (Maybe ZExpression) deriving (Read, Show, Eq)
```

Z expressions. Standard Z expressions include identifiers, function applications, integer literals, tuples, set and sequence literals, sequence types, Cartesian product types, and powerset types. As with Circus actions, we interpret string literals of ZExpression type as Z identifier expressions for convenience.

```
data ZExpression = ZIdExpr ZIdentifier | ZApplication ZExpression ZExpression | ZNumber Int | ZTuple [ZExpression] | ZSetLiteral [ZExpression] | ZSequenceLiteral [ZExpression] | ZSequence ZExpression | ZCartesian ZExpression ZExpression | ZPowerset ZExpression deriving (Read, Show, Eq) | ZFunction ZExpression ZExpression | ZPredefinedSet PredefinedSet
```
We now define the binary operators available to Z expressions in the output of the process.

```haskell
data ZBinOp = ZInfixExpression ZBinOp ZExpression ZExpression
  | ZPrefixExpression ZUnOp ZExpression
  | ZRealNumber Float
  deriving (Read, Show, Eq)
```

Z distinguishes syntactically between predicates, which are of Boolean type and can be used when a truth value is needed, and expressions, which cannot. Some operators, such as equality (and inequality and non-equality) and the predicate logic AND and OR, are defined only in predicates.

ACDC (and C) make no such distinction; expressions are used when truth values are required. The semantics of ACDC takes the result of an expression as true if and only if it is not (precisely) zero. This poses an implementation difficulty, as we must map ACDC’s binary operators into Z’s expression space, even though some of the Z analogues are available only in predicates. Furthermore, we need to capture the ACDC definition of truth in the definitions of the predicate operators.

Our solution is to introduce surrogates for the ACDC operators in terms of new Z operators, which we define below. We do not model the short-circuit semantics of the logical operators, as mentioned in the simplifying assumptions. Therefore, some properties that hold on these operators (such as commutativity and distributivity) do not hold for their ACDC equivalents.

As the logical operators are defined for any numerical value, we define them in terms of the set \( \mathbb{A} \). For example, consider C logical AND, which is true (\( \top \)) if and only if neither of its arguments are false (\( \bot \)):

\[
\begin{align*}
\text{cLAnd} &: \mathbb{A} \times \mathbb{A} \rightarrow \{0, 1\} \\
\forall x, y : \mathbb{A} \bullet x \text{cLAnd} y = 1 &\iff x \neq 0 \land y \neq 0
\end{align*}
\]

In Haskell this is represented by the production \( \text{CLAnd} \):

```haskell
= \text{CLAnd}
```

We define the remaining binary operators in Appendix A.

**Unary operators.** We define the standard Z negation operator, in addition to an encoding of the ACDC logical negation operator. The latter is defined by the following axiom:

\[
\begin{align*}
\text{CNot} &: \mathbb{A} \rightarrow \{0, 1\} \\
\forall x : \mathbb{A} \bullet \text{CNot} x = 1 &\iff x = 0
\end{align*}
\]

```haskell
data ZUnOp = ZNegation | CNot deriving (Read, Show, Eq, Enum, Bounded)
```

**Predefined sets.** PredefinedSet represents the basic mathematical number sets used to model ACDC types in a shallow manner. These are the naturals (\( \mathbb{N} \)) and integers (\( \mathbb{Z} \)), which are part of the Z mathematical tool-kit but defined in the syntax tree for convenience, and two extensions from ProofPower-Z: the reals (\( \mathbb{R} \)), and the universe type (\( \mathbb{U} \)), which represents a value of any type [54, sec. 2.1.1.7].

```haskell
data PredefinedSet = N | Z | R | U deriving (Read, Show, Eq, Enum, Bounded)
```

**Z declaration parts.** A Z declaration part is one or more Z basic declarations.

```haskell
newtype ZDeclPart = ZDeclPart (Seq1 ZDeclaration) deriving (Read, Show, Eq)
```

**Z declarations.** We only implement the first production in the ISO grammar for declarations: a basic declaration of a non-empty sequence of names, followed by the type expression for those names.

```haskell
data ZDeclaration = BasicDecl { basicDeclNames :: Seq1 ZIdentifier , basicDeclTypes :: ZExpression } deriving (Read, Show, Eq)
```
Predicates. In our subset of the Z notation, a Z predicate is a literal Boolean value, an predicate infix operation, the negation of another Z predicate, or a conjunction of two predicates separated by a new-line.

```haskell
data ZPredicate = PBoolean Bool |
PInfix PBinOp ZExpression ZExpression |
PNegation ZPredicate |\nPNConjunction ZPredicate ZPredicate
deriving (Read, Show, Eq)
```

Z predicates form a monoid under newline-conjunction, with PBoolean True as the empty value and PNConjunction, with special cases to preserve the monoid laws, as the associative operation.

Predicate binary operators. We implement the equality, set-membership and set non-membership operators.

```haskell
data PBinOp = Equality |
Membership |
NonMembership
deriving (Read, Show, Enum, Eq, Bounded)
```

Schema expressions. In this subset, a schema expression can only be an instance of schema text.

```haskell
data ZSchemaExp = SEText ZSchemaText deriving (Read, Show, Eq)
```

Schema text. The text of a Z schema introduces a Z declaration, followed by an optional predicate.

```haskell
data ZSchemaText = ZSchemaText [ZDeclPart] (Maybe ZPredicate) deriving (Read, Show, Eq)
```

Identifiers. We use the name ZIdentifier for identifiers in Z, to distinguish them from the Identifiers used in ACDC. They are, however, represented in the same manner.

```haskell
type ZIdentifier = String
```

Function calls. These are not part of our Z syntax, but capture function calls that have been translated into Z.

```haskell
data ZFunctionCall = ZFunctionCall { zfcIdentifier :: ZIdentifier , zfcArguments :: [ZExpression] , zfcHasReturn :: Bool } deriving (Read, Show)
```

4.4.3 General constructs

We implement some data types that are not part of the abstract syntax trees themselves, but on which the syntax trees are dependent. These are the Seq1 and Tree types.

Non-empty sequences. We handle non-empty sequences in the translation process. We could use the Haskell list type for these, but it has no type-level guarantee of non-emptiness. We therefore introduce a new type, Seq1.

```haskell
data Seq1 a = Base a |
Inductive a (Seq1 a)
deriving (Read, Show, Eq)
```

Not presented here is code for making Seq1 an instance of the Haskell type classes Functor, Foldable and Traversable. These instances allow the functions fmap, foldl’ and mapAccumL, amongst others, to be used on sequences later on.

The function toList converts a sequence to a list. It is automatically defined on Seq1 by its membership of the Foldable type class, so we need not define it here. We must, however, explicitly define its inverse, toSeq. This is a partial function: it will fail on empty lists. To encode this possibility, we define the result of toSeq as being Maybe (Seq1 a).

```haskell
toSeq :: [a] -> Maybe (Seq1 a)
toSeqP :: [a] -> Maybe (Seq1 a)
```

We define toSeq differently for the list of one item and the list of two or more items. The first case is translated into a present Base; the second is an application of Inductive to the first item in the list, followed by a recursive call to toSeq for the remaining list. In the second case, as the result of toSeq is in a Maybe context, we must lift the Inductive x into that context.

```haskell
toSeq [] = Nothing
toSeq [x] = Just (Base x)
toSeq (x : xs) = Inductive x <> toSeq xs
```

When non-emptiness of the list is assumed or required, it is convenient to dispel the Maybe context from toSeq. Thus, we also define toSeqP, which fails outright if the list is empty, and always returns a sequence otherwise.

```haskell
toSeqP :: [a] -> Seq1 a
toSeqP = fromMaybe (error "Cannot toSeqP the empty list.") <<< toSeq
```
Aside (Partial functions.). When a function \( f \) returns its result in a \( \text{Maybe} \) context, we denote any equivalent that does not—and terminates with an error when \( f \) would return \( \text{Nothing} \)—with the name \( fP \). (End of aside.)

N-ary trees. A Tree is a leaf of one value, or a branch, which contains a list of values. It is an instance of the type classes Functor, which allows it to be mapped over, and Foldable, which allows it to be folded over and converted to a list (using toList).

\[
data \text{Tree } a = \text{Leaf } a \mid \text{Branch } [\text{Tree } a] \quad \text{deriving (Read, Show, Eq)}
\]

We now formalise the translation phases we outlined earlier, starting with the first, flatten.

4.5 Flattening scopes

We begin our formalisation of this phase by introducing notation for the various types of flattening function.

4.5.1 Types of flattening

To avoid repeating similar function type signatures and emphasise similarities between functions, we define type synonyms for use in the type signatures for the flattening functions.

Flattening. A (scope-)flattening takes an environment \( e \) mapping local names to full names, and a syntactic construct \( c \) using local names, and returns the construct given by substituting the full name in \( e \) for each equivalent local name in \( c \).

\[
\text{type } \text{Flattening } a = \text{Env } \to a \to \text{FullName}
\]

Scoped flattening. These take the name of the current scope, as well as the current environment.

\[
\text{type } \text{ScopedFlattening } a = \text{ScopeName } \to \text{Flattening } a
\]

Environment-modifying flattening. Some syntactic constructs introduce bindings into their current scope: examples include function prototypes and definitions, enumerations, and variable declarations. Flattening for these constructs return, in addition to the renamed syntax, the environment resulting from updating the initial environment with these new bindings.

\[
\text{type } \text{EnvFlattening } a = \text{Env } \to a \to (\text{Env}; a \to \text{FullName})
\]

Scoped environment-modifying flattening. These are the EnvFlattening counterparts of ScopeFlattening: they accept a scope name, an environment, and a construct, and return the pair of the modified environment and renamed construct:

\[
\text{type } \text{ScopedEnvFlattening } a = \text{ScopeName } \to \text{EnvFlattening } a
\]

We now formalise the flattening process itself. We begin by considering the entire program.

4.5.2 Program

We begin with the renaming of an entire program, captured by flatten, which maps a program to an external declaration list.

\[
\text{flatten :: Program } \to a \to [\text{External } a \to \text{FullName}]
\]

Example. Consider a Program containing the following ACDC function:

\[
\begin{align*}
\text{struct } \text{diff}_\_\text{out } \text{diff}(\text{float } \text{input}, \text{float } k, \text{float } \text{in}_\text{mem}) \{ \\
\quad \text{struct } \text{diff}_\_\text{out } \text{result} = (0, 0); \\
\quad \text{result}.\text{output} = (k) * ((\text{input}) - (\text{in}_\text{mem})); \\
\quad \text{result}.\text{out}_\text{mem} = \text{input}; \\
\quad \text{return } \text{result}; \\
\}
\end{align*}
\]

Its flattening, as produced by flatten, is equivalent to the following pseudo-ACDC:

\[
\begin{align*}
\text{struct } \text{diff}_\_\text{out } \text{diff}::\text{input}, \text{float } \text{diff}::k, \text{float } \text{diff}::\text{in}_\text{mem}) \{ \\
\quad \text{struct } \text{diff}_\_\text{out } \text{diff}::\text{result} = (0, 0); \\
\quad \text{diff}::\text{result}.\text{output} = (\text{diff}::k) * ((\text{diff}::\text{input}) - (\text{diff}::\text{in}_\text{mem})); \\
\quad \text{diff}::\text{result}.\text{out}_\text{mem} = \text{diff}::\text{input}; \\
\quad \text{return } \text{diff}::\text{result}; \\
\}
\end{align*}
\]
The above is similar to part of the case study, and is representative of typical usage. (End of example.)

The function flatten has two roles. First, it renames a program, changing all LocalNames to FullNames. Then, it must collapse the program from a list of translation units to a list of external declarations. The first role is separated into flattenProgram. We pass an empty environment into flattenProgram, as there are no bindings to consider at the beginning of the program. For the second role, we extract the list of units from the renamed program, using programUnits, and concatenatively map toList to convert each unit from a sequence to a list, while also merging the unit lists into one.

\[
\text{flatten} = \text{concatMap toList} \circ \text{programUnits} \circ \text{flattenProgram}
\]

We now define flattenProgram itself. It is a simple Flatten: it need not return an environment.

\[
\text{flattenProgram} \colon \text{Flattening Program}
\]

We define flattenProgram using mapAccumL \cite{flanagan2005flattening}. This maps a function across a data structure from left to right, passing and returning an accumulator value on each call. We can thus traverse the units of the program, renaming each, while building an environment of seen definitions so that their references later in the program may be renamed.

We use mapAccumL with flattenUnit and the initial environment, producing a pair containing the final environment and the list of flattened units. We then use snd, which retrieves the second element of a pair, to discard the environment as it is not needed after this stage in the transformation.

Although the functions given above act on a list of program units, we represent programs with the opaque type Program. We thus need to translate from a Program to a list of units, and back. The former is performed by programUnits, a function automatically generated by Haskell from the record-notation definition of Program. We perform the latter by using the type constructor Program to build a new program from the renamed unit list. Composition of the above functions defines flattenProgram.

\[
\text{flattenProgram env} = \text{Program} \circ \text{snd} \circ \text{mapAccumL} \text{flattenUnit env} \circ \text{programUnits}
\]

This is an example of a function in which some parameters are elided: the definition of the function may be represented as a composition of other functions without explicitly naming all parameters.

We now define flattenUnit as a nested function of flattenProgram. Recall that a translation unit is a non-empty sequence of external declarations. We thus use mapAccumL to map flattenExternal, which considers external declarations, over the sequence, using the environment as an accumulator.

This accumulative map, which characterises flattenUnit, returns a pair containing the environment after considering the external declaration, as well as the renamed unit as a non-empty sequence of external declarations. Thus, the two uses of mapAccumL combine to flatten the program, given a method of flattening its external declarations.

\[
\text{where } \text{flattenUnit} = \text{mapAccumL} \text{flattenExternal}
\]

We now consider external declarations in detail.

### 4.5.3 External declarations

We handle external declarations using flattenExternal. External declarations add bindings to the global scope, so the Env provided to the function must be updated with these bindings. Thus, flattenExternal is an EnvFlattening.

\[
\text{flattenExternal} \colon \text{EnvFlattening External}
\]

We handle function definitions, global-scope declarations and enumerations using functions defined later: flattenFunction, flattenDeclaration and flattenEnum, respectively. We cover the remaining cases inside flattenExternal directly.

**Function prototypes.** We add the name of the function into the environment, and rename the parameters of its signature. Thus, we return a pair containing the environment updated with a mapping from the function’s name to its full name—the function’s name placed in the global scope—, and the renamed prototype.

The function setEnvGlobal takes an environment env and an identifier i, and yields env overridden with the mapping \((i, i)\); we implement the environment update in terms of it. We use flattenSignature to flatten the signature: this also returns an environment binding the renamed function parameters, which we discard as we do not need it.

**Example.** Consider the empty environment. After flattening the function prototype \(\text{void example(float k)}\), the environment will be \([\text{example}, \text{example}]\) and the parameter will be float example::k. (End of example.)

\[
\text{flattenExternal env (EDPrototype signature)} = (\text{env}’ \text{setEnvGlobal}’ \text{signatureName signature, EDPackage signature'})
\]

\[
\text{where } (_, \text{signature'}) = \text{flattenSignature env signature}
\]

**Aside (Quoted functions).** The above use of quotes converts a function into an infix operator: the expression above is equivalent to setEnvGlobal env (signatureName signature). (End of aside.)
Function definitions. We apply `flattenFunction` to the function inside the external declaration, which results in a pair of the environment after the function definition and the renamed function. We wish to return this pair as the result of `flattenExternal`, but since the `Function` that is the second element of the pair is not an `External`—the type of the second element of the pair returned by `flattenDeclaration`—we apply `second EDFunction`. The type constructor `EDFunction`, when applied to a `Function`, yields an `External`, and `second` applies the constructor to the second element of the pair.

```
flattenExternal env (EDFunction fun) = second EDFunction (flattenFunction env fun)
```

Global declarations are handled similarly, differing in which flattening function and type constructors are used. We formalise this in the appendix.

Structure specifiers. We ignore these: they add nothing to the environment and have no renamable contents.

```
flattenExternal env (EDStruct struct) = (env, EDStruct struct)
```

Enumeration specifiers. These add identifiers to the global scope, but need not be renamed. We handle them similarly to function prototypes, but handle the environmental changes using a separate function, `envEnum`.

```
flattenExternal env (EDEnum spec) = (envEnum env spec, EDEnum spec)
```

We now consider the flattening of function definitions.

4.5.4 Function definitions

Function definitions are renamed by `flattenFunction`, which takes an environment and function, and returns a pair of resulting environment and renamed function. As such, it is an `EnvFlattening Function`:

```
flattenFunction :: EnvFlattening Function
```

We proceed initially as with function prototypes. We take the name of the function from the signature and add it to the environment, thus recording the function name in the environment if it has not yet been encountered in a a prototype. This resulting environment, `env'`, is the one returned by the flattening.

We flatten the signature using `flattenSignature`. This time, we retain the environment it returns—which maps the parameters of the signature to their full names—, binding it to `envLocal`. We do not propagate `envLocal` outside of this function: the function parameters are not visible to the rest of the program.

We also scope-flatten the compound-statement body of the function. We create a `ScopeName` for the compound statement by appending the name of the function onto the global scope using `pushScope`.

We now rename the compound statement using `flattenCompound`, which takes the scope name, local environment and compound, and returns the renamed compound. Finally, we reconstruct the function using its original prototype and renamed body, and return it alongside the new global environment.

Example. Consider again the function `diff` from the example for `flatten`. When applied to the empty environment and the `Function` corresponding to the `diff`, `flattenFunction` yields a pair whose first element is the environment `[(diff, diff)]` (registering the function in the global scope), and whose second element is a `Function` corresponding to the flattened function from earlier. When renaming the body, the function’s scope name is `diff`, and the local environment is `[(diff, diff), (input, diff::input), (k, diff::k), (mem, diff::mem)]`. (End of example.)

We now formalise the above.

```
flattenFunction env (Function sig body) = (env', Function sig' body')
where funcName = signatureName sig
  funcScope = global 'pushScope' funcName
  env' = env 'setEnvGlobal' funcName
  (envLocal, sig') = flattenSignature env' sig
  body' = flattenCompound funcScope envLocal body
```

We now consider the flattening of function signatures.

4.5.5 Signatures

The function `flattenSignature` renames signatures by renaming their parameters; the other aspects are unmodified. The resulting environment is the input environment, `env`, but with the function’s parameters mapped to names qualified by the function’s scope name. To do this, we override `env` with a new environment containing these mappings, which we calculate using `envParameters`. We perform the override, giving priority to the new environment, using `setEnv`.
flattenSignature :: EnvFlattening Signature
flattenSignature env sig = (env', sig { signatureParameters = params' })
    where params' = map (flattenParameter env') params
        env' = signatureParameters sig
        funcName = signatureName sig
        funcParams = map parameterName params

The function envParameters, given a list of parameter names, generates a new environment mapping each parameter name to its fully qualified name in terms of the function scope.

envParameters :: Identifier → [LocalName] → Env

As an Env is a list of pairs of a LocalName and a FullName, representing the mapping from a scope-local name to its full, scope-qualified name, we construct these pairs for each parameter. We use the fan-out operator, &&&. which routes one argument to two functions and returns the pair of both results. The parameter name is already a LocalName, so we use the identity function id to yield the from side of the Env mapping. We construct the full names being the to side of the mapping, with the function name providing the scope, using fullName.

envParameters fname = map (id &&& fullName (global 'pushScope' fname))

To rename parameters according to the environment we just created, we use flattenParameter.

flattenParameter :: Flattening Parameter
flattenParameter env p = p { parameterName = env 'getEnv' parameterName p }

4.5.6 Statements

Considered next are statements, beginning with compound statements.

Compound statements

Compound statements, the element of ACDC’s syntax responsible for defining new scopes, are renamed by flattenCompound. In addition to the block environment, this also takes a scope name, which is to be prepended onto the names of any variables defined in the compound statement. Any changes to the environment are not propagated outside of the compound statement, and, therefore, no new Env is returned. It is therefore an example of a ScopedFlattening:

flattenCompound :: ScopedFlattening Compound

Example. Consider the following compound statement:

{ unsigned int variable;
  variable = 5;
}

If renamed by flattenCompound with the scope name [function, block], the result is:

{ unsigned int function::block::variable;
  function::block::variable = 5;
}

(End of example.) We flatten the declarations and statements in the compound statement using the nested functions flattenD and flattenS respectively. These are defined later on.

flattenCompound sname env (Compound ds ss) = Compound ds' ss'
    where (env', ds') = flattenD ds
          ss' = flattenS ss

We now define flattenD and flattenS. These are only used for compounds, so we define them as nested functions: they thus have access to the parameters of flattenCompound. The first, flattenD, is a trivial application of mapAccuml.

flattenD = mapAccuml (flattenDeclaration sname) env

We also use mapAccuml for flattenS. Here, the accumulator is the scope counter, which begins at zero. This counter records the number of compound statements that have already been defined inside this compound statement, and is used to ensure that each new, nested compound statement has a unique scope name (the result of appending the scope counter at its definition
to the scope name of its parent compound statement). As the scope counter may be increased by each statement, but the amount of increase depends on the type of statement considered, it must be used as an accumulator across the statements of the compound statement.

The environment from `flattenD` is not changed by statements, and is thus passed to `flattenStatement` unmodified for each statement. We use `snd` to discard the final counter value, as it is not needed.

\[
\text{flatten}S = \text{snd} \circ \text{mapAccumL} (\text{flattenStatement sname env'}) 0
\]

We now consider the flattening of each type of statement in turn.

**Statements**

Statements are considered by the function `flattenStatement`.

\[
\text{flattenStatement :: ScopeName} \\
\quad \rightarrow \text{Env} \\
\quad \rightarrow \text{Integer} \\
\quad \rightarrow \text{Statement LocalName} \\
\quad \rightarrow (\text{Integer, Statement FullName})
\]

**Null statements.** These are not renamed, and do not affect the counter. The scope name and environment are ignored, and both the counter and statement are returned unmodified.

\[
\text{flattenStatement _ _ _ _ counter (NopStm) = (counter, NopStm)}
\]

**Scope-less statements.** We next consider statements that do have effects, but do not contain compound statements, and thus do not introduce a scope. Thus, the scope name is ignored, and the counter is unmodified. We consider assignments here: we handle returns and expression statements similarly, but leave the formalisation to the appendix.

\[
\text{flattenStatement _ _ env _ _ counter (Assignment assignment) = (counter, Assignment assignment')} \\
\text{where assignment'} = \text{flattenAssignment (flattenAssignmentTarget env)} \\
\text{(flattenExpression env)} \\
\text{assignment}
\]

The next cases involve compound statements, and thus the incrementing of the scope counter.

**If-then-else.** These contain two compounds: one used if the expression is true, and the other used if it is false. We calculate their scope names by appending the current counter, and the counter plus one, to the current scope name for the true and false compound respectively.

*Example.* If the statement is in the function \( f \), and the counter is at 1, the scope names of the true and false compounds will be \( f::1 \) and \( f::2 \) respectively. (End of example.)

Once we have the scope names, we can rename the elements of the if-then-else statement. The condition expression and both compound statements are renamed by their respective functions, with the calculated scope names passed to `flattenCompound` for each compound statement. Since any variables defined inside the compound statement are only in scope for that compound statement, the environment is not modified and needs not be returned.

We return a pair of the new counter value—the original plus two—and the renamed expression.

\[
\text{flattenStatement sname env counter (IfThenElse expression true false)} \\
= (counter, IfThenElse expression' true' false') \\
\text{where expression'} = \text{flattenExpression env expression} \\
\text{true'} = \text{flattenCompound snameTrue env true} \\
\text{false'} = \text{flattenCompound snameFalse env false} \\
\text{snameTrue} = \text{scope 'pushScope' show counter} \\
\text{snameFalse} = \text{scope 'pushScope' show counterFalse} \\
\text{counterFalse} = \text{counter + 1} \\
\text{counter'} = \text{counterFalse + 1}
\]

The renaming of switch statements is handled in a similar way and presented in the appendix.
**Statements introducing one new scope.** As these involve a similar change to the counter and scope name, we handle the remaining cases together. The new scope is given the name resulting from pushing the current scope counter onto the current scope name: for example, if the scope name at the statement is `function` and this is the first new scope, the compound statement will have scope name `function::0`. The counter is incremented by one, to reflect the single new scope, and the renamed statement is returned along with it.

\[
\text{flattenStatement } \text{sname } \text{env } \text{counter } \text{statement} = (\text{counter}', \text{statement}') \text{ where } \\
\text{statement'} = \text{case statement of } \\
(\text{CompStm compound}) \rightarrow \text{CompStm (flattenCompound sname' env compound)}
\]

The remaining cases are given in the appendix.

\[
\text{sname'} = \text{sname 'pushScope' show counter} \\
\text{counter'} = \text{counter + 1}
\]

We now consider the flattening of syntax associated with assignments.

**Assignments**

These are handled by `flattenAssignment`.

\[
\text{flattenAssignment} \\
:: (\text{AssignmentTarget LocalName }\rightarrow \text{AssignmentTarget FullName}) \\
\rightarrow (\text{Expression LocalName }\rightarrow \text{Expression FullName}) \\
\rightarrow \text{Assignment LocalName} \\
\rightarrow \text{Assignment FullName}
\]

This function is defined in terms of two other functions: a function to rename assignment targets, and one to rename expressions. These are substituted when `flattenAssignment` is used.

**Self-assignments.** We rename these using the function provided to rename assignment targets:

\[
\text{flattenAssignment } a \_ (\text{AssignSelf op lhs}) = \text{AssignSelf op (a lhs)}
\]

**Value assignments.** The target is renamed as above, and the value is renamed using the expression-renaming function.

\[
\text{flattenAssignment } a \ e (\text{AssignValue op lhs rhs}) = \text{AssignValue op (a lhs) (e rhs)}
\]

We now consider assignment targets.

**Assignment targets**

The function `flattenAssignmentTarget` flattens assignment targets.

\[
\text{flattenAssignmentTarget} :: \text{Flattening AssignmentTarget}
\]

**Example.** Consider an assignment assigning to the target `variable[index]`. The environment at the assignment is

\[
[(\text{variable, function::variable}), (\text{index, function::0::index})]
\]

Then, `flattenAssignmentTarget` will yield the new target `function::variable[function::0::index]`. *(End of example.)*

**Variables.** This is the base case, and involves substituting, for the variable name (which will be a `LocalName`), the `FullName` to which the environment maps it:

\[
\text{flattenAssignmentTarget env (AssignID i) = AssignID (env 'getEnv' i)}
\]

**Array indices.** This is an inductive case, and involves the flattening of the recursive assignment target defining the array, as well as the flattening of the expression providing the array index.
Example. Consider the assignment target \texttt{array[index1][index2]}. This will be seen by \texttt{flattenAssignmentTarget} as an assignment to the index \texttt{index2} of an assignment target \texttt{x}, where \texttt{x} is to be considered next. The expression \texttt{index2} is renamed: let \texttt{index2R} is the result.

We then consider \texttt{x} by applying \texttt{flattenAssignmentTarget}, with the same environment as the original application. This will be considered as an assignment to the index \texttt{index1} of another target \texttt{y}, thus producing a renamed index \texttt{index1R} and another recursion \texttt{y} to rename.

Finally, \texttt{y} is an \texttt{AssignID}, whose identifier is \texttt{array}, and which we can rename using the rule above to \texttt{arrayR}. Reconstructing the full target, we have \texttt{arrayR[index1R][index2R]}. (End of example.)

\begin{verbatim}
flattenAssignmentTarget env (AssignArray array ix) = AssignArray array' ix'
where array' = flattenAssignmentTarget env array
      ix' = flattenExpression env ix
\end{verbatim}

Struct members. These are similar to array indices, but the member is a constant identifier, and therefore is not renamed. The formal definition is in the appendices.

We now move to the consideration of declarations.

4.5.7 Declarations

Declarations are renamed by \texttt{flattenDeclaration}.

\begin{verbatim}
flattenDeclaration :: ScopedEnvFlattening Declaration
\end{verbatim}

A declaration is a set of declaration specifiers, which we need not modify, followed by a local name and an initialiser. We rename the declaration to its full name, which is constructed from the current scope name and the local name, then consider the initialiser. Since the initialiser is optional, we lift the \texttt{flattenInitialiser} function so it renames the initialiser if it exists and has no effect if it does not. \texttt{flattenInitialiser} is defined as the mapping of \texttt{flattenExpression} over the initialiser, as the initialiser is a \texttt{Functor}, this mapping can be expressed using the \texttt{map} operator.

\begin{verbatim}
flattenDeclaration sname env (Declaration ds fname ini) = (env', Declaration ds fname' ini')
where fname' = fullName sname fname
      env' = setEnvLocal env sname fname
      ini' = flattenInitialiser <<< ini
      flattenInitialiser = (flattenExpression env<<<)
\end{verbatim}

We now consider the scope-flattening of expressions.

4.5.8 Expressions

An expression is flattened by \texttt{flattenExpression}:

\begin{verbatim}
flattenExpression :: Flattening Expression
\end{verbatim}

Identifier expressions. These are renamed to their full names as per the environment:

\begin{verbatim}
flattenExpression env (IdExpr i) = IdExpr (env 'getEnv' i)
\end{verbatim}

For example, with an environment mapping \texttt{input} to \texttt{diff::input}, the expression \texttt{input} will become \texttt{diff::input}.

All other cases involve the distribution of \texttt{flattenExpression} through the other forms of Expression, renaming recursive inclusions of Expressions and leaving other items untouched. The formal definitions are available in the appendix.

We now consider the scope-flattening of enumerations.

4.5.9 Enumerations

To process enumeration specifier, we specify \texttt{envEnum}, which adds the bindings from a specifier into a global environment.

\begin{verbatim}
envEnum :: Env -> EnumSpecifier -> Env
\end{verbatim}

Enumerations consist of an identifier, which is ignored, and a non-empty sequence of enumerators. We extract the enumerator sequence using the record syntax accessor function \texttt{enumSpecifierEnumerators}. The enumerators are then considered by performing a left fold with the function \texttt{envEnumerator} and the initial environment. This applies \texttt{envEnumerator} with the first enumerator and initial environment, returning a new environment, which is then applied with the second enumerator to the same function, and so on until all enumerators have been added into the environment.
The function envEnumerator handles the enumerators themselves, by adding the identifier of the enumerator into a given environment and returning the resulting new environment.

\[
\text{envEnumerator :: Env} \rightarrow \text{Enumerator} \rightarrow \text{Env}
\]

**Example.** The enumerators example; and example\_two = 5; when considered by envEnumerator, add to the environment the mappings (example, example) and (example\_two, example\_two) respectively. (End of example.)

Enumerators can either be implicit, in which case they take the value of the previous enumerator in the enumeration plus one (or zero, if there is no such value), or explicit, in which case they take the value of the constant expression given. In both cases, the identifier is given by enumeratorId, an automatically generated record accessor.

\[
\text{envEnumerator env} = \text{setEnvGlobal env} \circ \text{enumeratorId}
\]

We next define the second phase: the capturing of declarations of interest. These are the function definitions and global-level constant declarations; all others are ignored.

### 4.6 Capturing declarations

We begin our discussion of this phase by discussing the data types we use to store the captured declarations, which will be used later on whenever a declaration of interest must be accessed.

Later stages of our translation process do not consider full programs, but instead consider various constructs capturing the necessary information therefrom in a more workable form. We now introduce the first, the \(\text{Decl}m\).

A \(\text{Decl}\) is a construct containing information about declarations of interest. We implement it as a list of \(\text{DeclProfiles}\), which are discussed later on. It records global-level constants and functions, as well as any declarations and statements therein.

\[
\text{newtype Decl t m} = \text{Decl} \{ \text{declProfiles} :: [\text{DeclProfile}] \} \text{ deriving (Read, Show, Eq)}
\]

The types \(t\) and \(m\) given to \(\text{Decl}\), but unused in the definition, are **phantom types**. These add additional information to a \(\text{Decl}\) via the type system, and prevent the mixing of \(\text{Decl}s\) at different stages of usage. We use the phantom type \(t\) to mark whether the \(\text{Decl}\) is untimed (temporally abstracted), using the empty types \(\text{Timed}\) and \(\text{Untimed}\) as markers.

\[
\begin{align*}
\text{data Timed} \\
\text{data Untimed}
\end{align*}
\]

The second phantom type \(m\) determines whether the \(\text{Decl}\) only has constants, or if it is mixed.

\[
\begin{align*}
\text{data Mixed} \\
\text{data ConstantsOnly}
\end{align*}
\]

We now define the elements of a \(\text{Decl}\), namely \(\text{DeclProfiles}\). A \(\text{DeclProfile}\) is a mathematical object capturing a declaration, and is defined differently for functions and constants. A profile for a function contains its identifier and full function definition. A constant’s profile is its identifier and declaration. All declarations and statements in the \(\text{DeclProfile}\) take FullNames: the process of constructing \(\text{DeclProfiles}\) must occur after the scope flattening of the program.

\[
\begin{align*}
\text{data DeclProfile} &= \text{DPFunction} \{ \text{dpId :: Identifier, dpFunction :: Function FullName} \} \\
&\quad | \text{DPConstant} \{ \text{dpId :: Identifier, dpConstantDecl :: Declaration FullName} \} \\
&\quad \text{deriving (Read, Show, Eq)}
\end{align*}
\]

Having defined \(\text{Decl}s\) and \(\text{DeclProfiles}\), we now elaborate the translation process phase concerning the capture of declarations. This phase of the translation process is characterised by declCapture. This function captures information about declarations in a scope-flattened program, and is a mapping from a potentially empty list of external declarations to a \(\text{Decl}\). As we have not yet performed temporal abstraction, and the \(\text{Decl}\) may contain constants and functions, the full type of the result is \(\text{Decl Timed Mixed}\). We here define \(\text{declCapture}\) and a selection of its sub-functions; additional definitions are available in Appendix A.

\[
\text{declCapture :: [External FullName]} \rightarrow \text{Decl Timed Mixed}
\]

Each external declaration may now be considered individually. As we treat the profiles of a \(\text{Decl}\) as a list of pairs, we may map the function declCaptureExternal, which captures one external declaration, over the list of external declarations. However, an external declaration may be of a kind we do not need to capture. We need not record enumerations and structures in the \(\text{Decl}\), and we do not record global declarations unless they represent constants. In the case of enums, this is because they are no longer needed for the translation process; in the other cases, there are other constructs we use, such as the external outputs and inputs lists and structure maps introduced later, that capture the useful information associated with these declarations.

Therefore, declCaptureExternal returns \(\text{Maybe DeclProfile}\) (recall from page 25 that \(\text{Maybe}\) represents potentially absent values). This means we cannot use \text{map}: we would have a list of \(\text{Maybe DeclProfiles}\), as opposed to one of \(\text{DeclProfiles}\). Instead, we use mapMaybe, which yields the values of all results from the mapping that are present (that is, of the form \text{Just a}). This returns a list of \(\text{DeclProfiles}\): to convert this to a \(\text{Decl}\), we apply the type constructor \(\text{Decl}\) to the result.
Example. Consider the fragment of **ACDC** below.

```c
const unsigned int constant = 5;
unsigned int not_a_constant;
enum example {
  EXPLICIT_ENUMERATOR = 5;
  IMPLICIT_ENUMERATOR;
};
struct structure {
  signed int test;
};
void function(signed int function::argument) {
  function::argument ++;
}
```

The only declarations captured by `declCapture` here are `const unsigned int constant = 5;` and the function `function()`. The global variable, enum and struct yield `Nothing` when captured, and are thus removed from the list of profiles that are sent to `Decl` to create the final profile. *(End of example.)*

```
declCapture = Decl \circ mapMaybe declCaptureExternal
```

We now define `declCaptureExternal`, which captures the declarations present in an external declaration.

```
declCaptureExternal :: External FullName \rightarrow Maybe DeclProfile
```

We first consider the case of enum declarations, struct declarations and function prototypes: these are not captured and result in `Nothing`. For example, we have the definition below.

```
declCaptureExternal (EDEnum _) = Nothing
```

We then consider the declarations and function definitions. Function definitions are handled by `declCaptureFunction`, which maps from a function to zero or one profiles.

```
declCaptureFunction :: Function FullName \rightarrow Maybe DeclProfile
```

The function definition is a declaration to be captured, so we include it in the `Decl`. Recall that we are only interested in functions (which cannot be nested) and global constants (which cannot appear in functions): thus, we need not capture any declarations inside the function.

Since every part of the function will be used later in the process, we simply copy in the entire `Function`, extracting its name from its signature to be given to the profile.

```
declCaptureFunction f = Just (DPFunction ((signatureName \circ functionSignature) f) f)
```

Declarations. The function `declCaptureDeclaration` captures an internal declaration.

Example. The global-scope constant `const unsigned int test = 5;` yields a `DeclProfile` of

```
DPConstant "test"
  Declaration (DS Const (Int Unsigned) "test"
    (Leaf (ConstantExpr (CInt 5)))
```

(In the above, "test" properly represents a `FullName` in the global scope, with identifier "test".)

The global-scope variable `unsigned int test = 5;` yields nothing: it is not a constant. *(End of example.)*

```
declCaptureDeclaration :: Declaration FullName \rightarrow Maybe DeclProfile
```

Since the only internal declarations we capture are constants at global level, we select only constant declarations for further consideration. Other declarations are ignored, and return `Nothing`.

To perform the capture, we must first find the identifier of the constant. The partial function `flattenGlobalName` converts a `FullName` to a basic `Identifier`, succeeding only if the `FullName` is in the global scope. The result of `flattenGlobalName` is thus given inside a `Maybe` context.

We then construct a `DPConstant` profile by attaching the flattened name to the original declaration body, which is captured as the variable `d`. Using `DPConstant` in infix operator form, we can apply `d` to create a new function that takes the flattened name and returns the desired profile. However, both the flattened name and the result of `declCaptureDeclaration` are wrapped in a `Maybe` context, whereas the profile-building function accepts and returns plain values.

Recall from page 25, that `Maybe a` is a functor: thus, we can lift functions of type `a → b` to type `Maybe a → Maybe b`. We now make use of this ability, using the functor mapping operator `<$>`, which lifts the function on the left hand side and applies it to the argument on the right hand side. Thus, we complete the definition of `declCaptureDeclaration`. 


This concludes the discussion of the second phase of the translation process. We now move to formalising the third, namely the abstraction of temporal information.

### 4.7 Abstracting temporal information

This phase is realised by the function \textit{abstractTime}, which maps from timed \texttt{Decl}s to untimed \texttt{Decl}s.

\[
\text{abstractTime} :: \text{Decl Timed Mixed} \rightarrow \text{Decl Untimed Mixed}
\]

To abstract from the temporal information, we first analyse the declarations to find variables related to the passage of time, and then remove references to them from the active declarations. These two stages are handled by the functions \textit{analyseTime} and \textit{removeTime} respectively.

\[
\text{abstractTime} \ decl = \text{removeTime} (\text{analyseTime} \ decl) \ decl
\]

The analysis and removal sub-phases are the subject of subsection 4.7.1 and subsection 4.7.2 respectively.

#### 4.7.1 Analysing for temporal information

**Analysis function types.** We first introduce a type synonym for the function types. \texttt{TAnalyser} represents the general form of the analysis functions: it takes a syntactic construct, and returns a list of names of variables that have participated in a temporal function call.

\[
\text{type} \ T\text{Analyser} \ a = a \rightarrow [\text{FullName}]
\]

We now define the analysis functions.

**Decl.** Given a \texttt{Decl}, \textit{analyseTime} calculates a list of full names of program variables participating in a temporal activity.

\[
\text{analyseTime} :: T\text{Analyser} \ (\text{Decl Timed Mixed})
\]

Recall our assumption that variables either participate directly in a temporal activity, or not at all. We thus need not compute the transitive closure of \textit{analyseTime}. If the assumption did not hold, we would need to retrieve variables being assigned from temporal variables, and so on.

To find the variables reaching a temporal activity, we concatenatively map \textit{analyseDeclProfile} over each \texttt{DeclProfile} in the \texttt{Decl}, retrieved using the record accessor \texttt{declProfiles}. We do this because each instance of \textit{analyseDeclProfile} returns the list of temporal variables in that profile, and we desire a single list of the temporal variables across all profiles. We then use \texttt{nub}, which removes duplicates from a list, to ensure each temporal variable is only represented once.

\[
\text{analyseTime} = \text{nub} \circ \text{concatMap} \text{analyseDeclProfile} \circ \text{declProfiles}
\]

**Declaration profiles.** We analyse individual \texttt{DeclProfiles} with \textit{analyseDeclProfile}.

\[
\text{analyseDeclProfile} :: T\text{Analyser} \ \text{DeclProfile}
\]

We assume that the result of a temporal expression is never used again, so we need not search the initialisers of declarations for temporal expressions. Thus, constants, which are only given value by initialisers, always return the empty list. The bracket syntax below is shorthand for a pattern match that matches the type constructor \texttt{DPConstant}, but ignores all of the fields of the constant record.

\[
\text{analyseDeclProfile} \ \texttt{DPConstant} \ [] = []
\]

When the declaration is a function, as is the case in all other cases of \textit{analyseDeclProfile}, we consider that function’s compound statement using \textit{analyseCompound}.

\[
\text{analyseDeclProfile} \ f = \text{analyseCompound} (\text{dpFunctionBody} \ f)
\]
Compound statements. The function `analyseCompound` traverses a compound statement to find temporal variables.

```haskell
analyseCompound :: TAnalyser (Compound FullName)
```

Here, we introduce the first instance of a common pattern used throughout the formalisation. Since many of the intermediate phases of the translation involve traversing through parts of the `Decl` to build a list of results, we provide a series of traversal functions. These reduce the problem of traversing a syntactic construct into that of traversing certain types of sub-construct, and, given functions that implement those smaller traversals, construct the overall traversal on the original construct. Their formalisations are given in section A.8.

The function `traverseCompoundVDE` takes functions for converting variable names in assignment targets, declarations and expressions to lists, and, given a compound statement, returns the result of traversing a compound using those functions and concatenating their results over every instance of each of the above syntactic constructs. We implement `analyseCompound` in terms of this traversal; since we do not consider declarations or assignment target variable names, the traversal function we give for both, x, always returns the empty list.

```haskell
analyseCompound = traverseCompoundVDE (const []) (const []) (analyseExpression False)
```

We explain the `False` argument to `analyseExpression` later. As the traversal above handles statements and assignment targets in terms of the analysis functions given, we need not consider them separately.

Expressions. We consider these with `analyseExpression`.

```haskell
analyseExpression :: Bool -> TAnalyser (Expression FullName)
```

The Boolean parameter, `inTemporal`, is true if, and only if, the function is being called inside a temporal function call. If `inTemporal` is true, the analyser records all variables it finds as being temporal variables, as they are being used inside a temporal function call. Otherwise, the analyser assumes the variables are not related to a temporal call unless it detects such a call: `inTemporal` is then set to true when analysing inside that call. This approach allows us to use the same function for handling both general program expressions and the sub-expressions of temporal function calls. Later on, we will use the variable-capturing behaviour of `analyseExpression True` to achieve a different result.

Example. Consider the expression `var1 + sleep(var2 + var3)`, outside any temporal function calls. (This expression is meaningless, as `sleep` returns no value, but will suffice as an example.) When initially considering the whole expression, `inTemporal` will be false. However, upon considering the argument to the function `sleep`, we note that `sleep` is a temporal function, and `inTemporal` is set to true for each of its arguments. Thus, the expression `var2 + var3` is considered with `analyseExpression True`, and thus `var2` and `var3` are recorded as temporal variables. (End of example.)

Constants and literals are ignored; the formalisation of this behaviour is in the appendix. Identifiers are returned if, and only if, the Boolean is true, and otherwise discarded, which is implemented by the list comprehension below.

```haskell
analyseExpression inTemporal (IdExpr i) = [i | inTemporal]
```

For call expressions, where the function is a delay function (its name is `time` or `sleep`), `inTemporal` is switched on and the identifiers in each argument are captured. Otherwise, `analyseExpression` distributes over the function’s arguments.

```haskell
analyseExpression inTemporal (CallExpr (FC name args)) = concatMap (analyseExpression inTemporal') args
  where inTemporal' = isDelayFunction name ∨ inTemporal
```

Finally, `analyseExpression` distributes over all other expressions.

4.7.2 Removing unused temporal information

Before we define the functions removing temporal variables, we introduce a type synonym for functions participating in the removal. A `TRemover` takes the list of full names calculated by the analysis stage and a syntactic construct, and returns that construct with uses of the named variables removed where possible.

```haskell
type TRemover a = [FullName] → a → a
```

We now define the temporal variable removal functions, starting with the function for `Decls`.

**Decls.** The function `removeTime` removes the declarations that correspond to the identifiers found by `analyseTime`. It is not quite a `TRemover`, as it changes the type of the `Decl` to reflect its untimed nature.

```haskell
removeTime :: [FullName] → Decl Timed Mixed → Decl Untimed Mixed
```

We map `removeTimeDeclProfile` across the declarations in the `Decl`. We use `declProfiles` and `Decl` to extract the profiles from the old `Decl`, and create a new untimed `Decl`, respectively.

```haskell
removeTime names = Decl ◦ map (removeTimeDeclProfile names) ◦ declProfiles
```
Individual declarations. The function `removeTimeDeclProfile` removes temporal variable usage from a `DeclProfile`.

```haskell
removeTimeDeclProfile :: TRemover DeclProfile
```

As we are only investigating statements, `removeTimeDeclProfile` ignores constant declarations.

```haskell
removeTimeDeclProfile _ constant\[\{\}\] = constant
```

For functions, however, we need to investigate the function body, which is a compound statement. We use Haskell’s record notation to create a new function that is equal to the original function except in the case of the body, which is replaced with the result of applying `removeTimeCompound names` on the original statements.

```haskell
removeTimeDeclProfile names (DPFunction fname f) = DPFunction fname
\(f \{\text{functionBody} = \text{removeTimeCompound names (functionBody f)}\}\)
```

We now consider the removal process for the statements inside functions.

Statements. Statements are handled by `removeTimeStatement`.

```haskell
removeTimeStatement :: TRemover (Statement FullName)
```

The only statement types we remove directly are expression and assignment statements. We do not remove return statements. To do so would change the type and meaning of the function, and, due to our assumptions, no temporal information should reach a return statement.

Null and return statements are returned unmodified: see the appendices for the formalisation. Compound statements are handled by `removeTimeCompound`, which is defined later.

```haskell
removeTimeStatement names e (CompStm c) = CompStm (removeTimeCompound names c)
```

We now consider statements involving expressions. For such statements, we check whether the expression contains a temporal variable. If it does, we remove it. An exception is made if the expression is a call to a delay function: we retain these so the translation process can use them to implement timing-based semantics later on. Here, `getAny` is needed due to `hasTemporalVariables` not directly returning a boolean; we explain this later.

```haskell
removeTimeStatement names e (ExprStm expression)
\(\begin{align*}
| \text{isDelayExpression expression} = e &= e \\
| \text{getAny (hasTemporalVariables names expression)} &= \text{NopStm} \\
| \text{otherwise} &= e
\end{align*}\)
```

Assignment statements are removed if the assignment is marked as temporal by `assignmentTemporal`, which we define later.

```haskell
removeTimeStatement names a (Assignment assignment)
\(\begin{align*}
| \text{getAny (assignmentTemporal names assignment)} &= \text{NopStm} \\
| \text{otherwise} &= a
\end{align*}\)
```

Since we assume the temporal variables are not used in any way other than assignment and temporal functions, we distribute through the other statements’ compound statements while ignoring their expression components.

Compound statements. The function `removeTimeCompound` removes temporal variable usage in compound statements.

```haskell
removeTimeCompound :: TRemover (Compound FullName)
```

We again consider only the statements in the compound statement. We thus use the record update syntax to signify that the resulting `Compound` differs only from the input in its statements, which are mapped across by `removeTimeStatement`.

```haskell
removeTimeCompound names c = c \{\text{compoundStatements = statements'}\}
\text{where statements'} = \text{map (removeTimeStatement names) statements}
\text{compoundStatements} = \text{compoundStatements c}
```

The function `removeTimeSwitchCases`, which considers switch cases, is similar.

Having defined the removal functions for the elements of the ACDC abstract syntax tree of interest, we now define `assignmentTemporal`. This function takes the list of computed temporal variables and an assignment, and determines whether the assignment reaches one of the temporal variables.
Example. Assume that the variables \([a, b, c, d, e]\) have been analysed as temporal. Then, the assignments \(a = 3;\), \(f[a] = 4;\), and \(g = b;\) yield true when \(\text{assignmentTemporal}\) is applied, as they involve temporal variables. The assignments \(h = 3;\), \(i[j] = 4;\) and \(g = k;\) yield false, as they do not. (End of example.)

Note that we remove assignments if they involve array subscripts containing temporal variables. Our simplifying assumptions mean we cannot have any use of a temporal variable outside of implementing the cyclic executive time behaviour. Whilst these assumptions mean that such an assignment is likely useless, it is still a valid assignment so long as the variable is not used elsewhere.

The traversals used so far have operated on lists. As traversals are defined on any monoid, we can instead use the monoid \(\text{Any}\), which has the result of calculating a Boolean value that is true if \(\text{any}\) of the applications sub-functions returns \(\text{Any True}\). Given a function deciding whether an assignment target reaches temporal variables, and a similar function for expressions, we can implement \(\text{assignmentTemporal}\) as a traversal yielding true if either the target or any expressions reach temporal variables. We use \(\text{Any}\) to enter the \(\text{Any}\) context, and \(\text{getAny}\) to leave.

Aside (Repeated arguments). Consider \(f x = g(hx)(ix)(jx)\ldots\). We can rewrite \(f\) as \(f = g \triangleleft_{\triangleleft} h \triangleleft_{\triangleleft} i \triangleleft_{\triangleleft} j \ldots\), eliding the repeated argument \(x\). (The \(\triangleleft\) operator is precisely that used for lifting functions into functor and monadic contexts.) We use this idiom in \(\text{assignmentTemporal}\), and throughout the rest of the translation process. (End of aside.)

We use \(\text{anyTargetTemporal}\) to determine whether an assignment target contains temporal variables. To define it, we use the traversal \(\text{traverseTargetVE}\), with traversals for determining whether an assignment target variable name and any array subscript expressions relate to temporal variables. Respectively, we provide \(\text{varInNames}\), which flags an assignment’s target variable as temporal if it belongs to the list of temporal variables, and \(\text{hasTemporalVariables}\). Again, we use \(\text{Any}\) so that the traversal returns true if any application of the sub-functions yields true.

\[
\begin{align*}
\text{assignmentTemporal} :: [\text{FullName}] \rightarrow \text{Assignment FullName} \rightarrow \text{Any} \\
\text{assignmentTemporal} = \text{traverseAssignmentTE} \triangleleft_{\triangleleft} \text{anyTargetTemporal} \triangleleft_{\triangleleft} \text{hasTemporalVariables}
\end{align*}
\]

We now elaborate the process of finding the reachability graph for a program. This process is captured by \(\text{reachability}\), which maps untimed \(\text{Decls}\), as constructed in the previous phase, to their \(\text{RGraphs}\).

\[
\text{reachability} :: \text{RFinder} (\text{Decl Untimed Mixed})
\]

We define \(\text{reachability}\) as the transitive closure of the investigation of all functions in the \(\text{Decl}\). We use the transitive closure because the main body of \(\text{reachability}\) only returns immediate dependencies between functions, and our reachability graph
must include all dependency relations. We ignore constants as we seek function dependencies, thus only function calls inside existing functions need be considered.

To consider all functions, we use another traversal function. The function `traverseDeclF` implements a traversal of a `Decl` in terms of a traversal function for function prototypes. This traversal—`rFunction` in this case—takes the function name as an additional parameter.

We use the nested function `rFunction` to traverse functions. Here, dependencies of functions are captured by exploring their bodies (and ignoring their formal parameters), using `rCompound`. `rCompound` is an `InnerRFinder`, and thus takes the name of the function as its first parameter, as supplied by `traverseDeclF` to `rFunction`.

\[
\text{reachability} = \text{transitive} \circ \text{traverseDeclF} \ rFunction
\]
\[
\text{where } rFunction \ f = rCompound \ f \circ \text{functionBody}
\]

**Compound statements.** Recall that function calls are only found inside expressions. Thus, the only items of interest in a compound are expressions, including those in declaration initialisers. We use `traverseCompoundVE`, which traverses a compound given traversals for assigned variable names and expressions. The former are ignored—their traversal always returns the empty list—, and the latter is `rExpression`, defined later on. We now define `rCompound`.

\[
rCompound :: \text{InnerRFinder} \ (\text{Compound FullName})
\]
\[
rCompound \ f = \text{traverseCompoundVE} \ (\text{const} \ []) (rExpression \ f)
\]

**Expressions.** Expressions are handled by `rExpression`, which we implement with a case analysis.

\[
rExpression :: \text{InnerRFinder} \ (\text{Expression FullName})
\]

Constants, identifiers and strings are ignored by `rExpression`, and return the empty list.

A function call is analysed to check whether it calls one of the temporal functions `time()` or `sleep()`. This check is performed by `isDelayFunction`. If it is, the call is ignored; otherwise, the target function name is added to the reachability graph, and its arguments are considered by concatenatively mapping `rExpression`.

\[
rExpression \ f = \text{CallExpr} \ (FC \ identifier \ args)
\]
\[
\begin{align*}
| \ & \text{isDelayFunction} \ identifier = [] \\
| \ & \text{otherwise} \quad = (f, identifier) : \text{concatMap} (rExpression \ f) \ args
\end{align*}
\]

The `:` operator prepends a value onto a list, and is equivalent to the concatenation of a list containing the left hand side, and the right hand side.

**Example.** Consider the function calls `example()`, `sleep(3)`, `nested(argument())`, and `nested(time())`, within the body of a function `test()`.

The first is a non-delay function call with no arguments, and thus yields the dependency `(test, example)`. The second is a delay function, and thus yields no dependences. The third is a non-delay function, and thus we record `(test, nested)`, but, in this instance, one of the arguments is itself a function call, so we also record `(test, argument)`. Similarly, for the final call, we record `(test, nested)` as before, but the argument is a delay function and is thus ignored. (**End of example.**)

We distribute `rExpression` over all other types of expression.

### 4.9 Capturing variable usage

In this phase, we capture how and where variables are used. This phase is captured by `variableUsage`, which maps from untimed `Decl` to `VUsage` constructs. Before elaborating `variableUsage`, we introduce a type synonym to simplify this phase’s declarations: `VFinder` represents a mapping from a syntactic construct to a list of variable records.

\[
\text{type } VFinder a = a \rightarrow [\text{VariableRecord}]
\]

We now present `variableUsage` and its sub-functions; elided definitions are in Appendix A. `variableUsage` is not a `VFinder`, as it returns a `VUsage` and not a list of `VariableRecords`.

\[
\text{variableUsage :: Decl Untimed Mixed } \rightarrow \text{VUsage}
\]

As variables cannot be used in constants, we again make use of `traverseDeclF` to consider each function in the `Decl`. The nested function `vFunction`, which considers each function, returns a list of one element, which is a mapping from the function’s name to the list of records of variables used in that function. Since variables are only used in the bodies of functions, we use `functionBody` to extract the function’s compound statement, and `vCompound` to discover the variable usages inside it.

\[
\text{variableUsage} = \text{traverseDeclF} \ vFunction
\]
\[
\text{where } vFunction \ f = [(f, (vCompound \circ \text{functionBody}) \ f)]
\]
**Compound statements.** These are considered by \(v\text{Compound}\).

\[v\text{Compound} :: \text{VFinder} \ (\text{Compound FullName})\]

We use \(\text{traverseCompoundVE}\) to implement \(v\text{Compound}\) in terms of \(v\text{AssignedVariable}\) and \(v\text{Expression}\). This function handles declarations, statements and assignments by applying one of those functions when the relevant syntactic construct is found.

\[v\text{Compound} = \text{traverseCompoundVE} \ v\text{AssignedVariable} \ v\text{Expression}\]

**Assigned variables.** We consider the name of a variable inside an assignment target using \(v\text{AssignedVariable}\).

\[v\text{AssignedVariable} :: \text{VFinder} \ \text{FullName}\]

Assignment entails a variable write, so we record the variable that is the target of the assignment with a status of Write. The function \(\langle \text{VariableRecord} \ \text{Write} \rangle\) maps a variable name to such a record. As the output of a \(\text{VFinder}\) is a list, we need to promote the single record to a list of one element, which is done using \(\text{return}\).

\[v\text{AssignedVariable} = \text{return} \circ \langle \text{VariableRecord} \ \text{Write} \rangle\]

We now consider the capturing of variable usage from expressions. This is performed by \(v\text{Expression}\).

\[v\text{Expression} :: \text{VFinder} \ (\text{Expression FullName})\]

The only expressions of interest are identifier expressions, which imply the variable with that identifier is being read.

\[v\text{Expression} \ (\text{IdExpr identifier}) = \{\langle \text{VariableRecord identifier Read} \rangle\}\]

Generally, \(v\text{Expression}\) distributes over all other expression types in a manner similar to that of the reachability phase. Constant and string expressions return the empty list, and delay functions are ignored.

### 4.10 Retrieving struct definitions

This phase derives a mapping from struct identifiers in the scope-resolved ACDC program to lists of their struct declarations, which capture members and expected types. We return this mapping in the form represented by the type synonym \(\text{StructMap}\).

This form allows the use of standard functions such as \(\text{lookup}\), which finds a value by its key in a list of key–value pairs.

\[
\text{type} \ \text{StructMap} = [(\text{Identifier}; \text{StructDeclaration})]
\]

This phase is characterised by the function \(\text{getStructs}\), which maps from a flattened ACDC program to a list of \(\text{StructSpecs}\). We use another traversal, \(\text{traverseExternalListDFPNR}\), which takes traversals for declarations, functions, prototypes, enum specifiers and struct specifiers and yields a traversal on external declaration lists. We are only interested in structs, so the only traversal we supply is that on struct specifiers; the others return the empty list.

The struct traversal is \(\text{getStructSpecifier}\), which we define as a nested function. This extracts the name of the specifier and its sequence of declarations, and returns a single list containing a tuple of the name and the list equivalent of the declarations.

\[
\text{getStructs} :: [\text{External FullName}] \rightarrow \text{StructMap}
\]

\[
\text{getStructs} = \text{traverseExternalListDFPNR} \ (\text{const} []) \ (\text{const} []) \ (\text{const} []) \ (\text{const} []) \ \text{getStructSpecifier}
\]

\[
\text{where} \ \text{getStructSpecifier} \ (\text{StructSpecifier identifier}) = [(\text{identifier}; \text{toList declarations})]
\]

We now consider the retrieval of variable and function types.

### 4.11 Retrieving variable and function types

This phase of the process derives a function mapping each variable in the scope-resolved ACDC program to its declared type, and each function to its declared return type. We first define a type synonym, \(\text{TypeGet}\), for the functions in this phase.

\[
\text{type} \ \text{TypeGet a} = a \rightarrow [(\text{FullName}; \text{Type})]
\]

We now elaborate the phase, which is characterised by the function \(\text{getTypes}\). As with \(\text{getStructs}\), we use \(\text{traverseExternalListDFPNR}\). Here, we supply \(\text{getTypesDeclaration}\) and \(\text{getTypesFunction}\) as the traversals on declarations and functions, and returning the empty list for all other traversals.

\[
\text{getTypes} :: \text{TypeGet [External FullName]}
\]

\[
\text{getTypes} = \text{traverseExternalListDFPNR} \ \text{getTypesDeclaration} \ \text{getTypesFunction}
\]

\[
(\text{const} []) \ (\text{const} []) \ (\text{const} [])
\]
**Functions.** We use the function traversal `traverseFunctionPC`, which is implemented in terms of traversals of function signatures and compounds. Here, they are `getTypesSignature` and `getTypesCompound` respectively.

\[
\text{getTypesFunction} : \text{TypeGet (Function FullName)} \\
getTypesFunction = \text{traverseFunctionPC getTypesSignature getTypesCompound}
\]

**Signatures.** We find the type of the function whose signature is being considered by extracting it from the signature’s declaration specifiers. We then construct a mapping from the full name of the function to this type. Next, we consider the types of the parameters in the signature using a concatenative map of `getTypesParameter`, passing the function name so that it can be used to build the full names of each parameter. We then attach the function type to the resulting type map.

\[
\text{getTypesSignature} : \text{TypeGet (Signature FullName)} \\
getTypesSignature (Signature ds fname parameters) \\
= (globalName fname, dsType ds) : \text{concatMap getTypesParameter parameters}
\]

**Parameters.** We fan-out a parameter to a pair of functions: the left-hand function returns the parameter’s full name via attaching the function name to the parameter’s local name as a scope, and the right-hand function returns the type nested inside the parameter’s specifiers. This yields a pair of full-name and type, which we need to promote to a list: we do so by using the function `return`.

\[
\text{getTypesParameter} : \text{TypeGet (Parameter FullName)} \\
getTypesParameter = \text{return o (parameterName \& \& parameterType)}
\]

**Compound statements.** We implement the type-get for compound statements in terms of the traversal `traverseCompoundTDE`. This implements the traversal in terms of traversals for assignment targets, declarations and expressions. Since we are only getting types from declarations, we supply `getTypesDeclaration`, leaving the other two traversals as returning empty lists. This traversal will reach every declaration in the program, and, thus, every local variable’s type.

\[
\text{getTypesCompound} : \text{TypeGet (Compound FullName)} \\
getTypesCompound = \text{traverseCompoundTDE (const [] getTypesDeclaration (const [])}
\]

**Declarations.** We handle these similarly to parameters, substituting `declarationName` and `declarationType` into the fan-out. Having elaborated the pre-processing phases, we may now discuss the construction of the Circus model itself.

### 4.12 Building the Circus model

In addition to the outputs of the previous phases, we need to provide `circusify` with additional information about the cyclic executive architecture of the program. This module introduces the `CircusifyInput` type, which bundles all of the information required in this phase into one data structure.

A `FunctionMap` is a list of triples of step function, frame number, and function called by that step function on that frame.

\[
\text{type TypeMap} = [(FullName, Type)] \\
\text{type FunctionMap} = [(Identifier, Frame, Identifier)]
\]

A `CircusifyInput` captures all information needed to produce a Circus model: the names of the main functions; the name of the frame counter; the number of frames in the executive; the names of external outputs and inputs; the name of the timer function; and the type map, struct map, reachability graph, variable usage and `Decl` from earlier. We also take a `function map`, as defined above.

\[
\text{data CircusifyInput} = \text{CircusifyInput} \\
\{ \\
\text{mainFunctions :: [Identifier]} , \text{frameCounter :: FullName} \\
\text{, functionMap :: FunctionMap, frameCount :: FrameCount} \\
\text{, externalInputs :: [FullName], externalOutputs :: [FullName]} \\
\text{, timerFunction :: Identifier, types :: TypeMap} \\
\text{, structs :: StructMap, rgraph :: RGraph} \\
\text{, usage :: VUsage, decl :: Decl Untimed Mixed} \\
\} \text{ deriving (Read, Show)}
\]

We introduce a type synonym for sub-phases. A `CircusSubPhase` maps a `CircusifyInput` to a partial Circus model.

\[
\text{type CircusifySubPhase} = \text{CircusifyInput \rightarrow Circus}
\]
We now consider the overall shape of this phase. The phase is defined by \textit{circusify}, which takes an input context and returns its corresponding \textit{Circus} model.

\[
\text{circusify} : \text{CircusifyInput} \to \text{Circus}
\]

We define \textit{circusify} in terms of a series of sub-phases, each mapping a \textit{CircusifyInput} to a partial \textit{Circus} model. We promote the input to a one-item list using \textit{return}. As both our list of sub-processes and their input are in list form, and lists are a form of applicative functor, we can apply the former to the latter using the applicative functor application operator \(\circ\). This yields a list of sub-models created by sending the input to each sub-phase, which we concatenate into one model.

\[
circusify = \text{concat} \circ (\text{subPhases} \circ \text{return})
\]

\[
\text{where subPhases} = [\text{circusifyStructs}, \text{circusifyChannels}, \text{circusifyConstants}, \text{circusifyProcesses}, \text{circusifyTimer}, \text{circusifyProgram}]
\]

We now elaborate the modules for each of the sub-processes in turn.

4.12.1 Structs

Our modelling of structs in \textit{Circus} takes the form of functions that map from \(Z\) representations of the struct member names to universe-type values representing the member value. We represent member names by creating a \(Z\) free type for each struct type, whose members are the names of each member of that struct type.

This sub-phase, captured by \textit{circusifyStructs}, generates free types for each struct type in the \textit{CircusifyInput}.

\[
\text{circusifyStructs} : \text{CircusifySubPhase}
\]

The function \textit{circusifyStructs} is a composition of \textit{structs}, which extracts the \textit{StructMap} of the input context, and a mapping of the composition of \textit{structCToZ}, which converts a struct entry into a \(Z\) paragraph, and \textit{CPar}, which converts a \(Z\) paragraph to a \textit{Circus} paragraph.

\[
\text{circusifyStructs} = \text{map} (\text{CPar} \circ \text{structCToZ}) \circ \text{structs}
\]

We now define \textit{structCToZ}, which considers a single entry in a \textit{StructMap}.

\[
\text{structCToZ} :: \text{Identifier, [StructDeclaration]} \to \text{ZParagraph}
\]

We define \textit{structsCToZ} in terms of two other functions: \textit{structCToZ}, which considers a single struct, and \textit{structDeclCToZ}, which considers a single struct member declaration. We now define the former, which maps pairs of struct names and member lists to \(Z\) paragraphs.

We first use the operator \(***\) to apply \textit{structFreeTypeName} and the mapping of map \textit{structDeclCToZ} over the struct name and declaration list respectively. We now have a pair of free type name and declaration list, which are the arguments taken by the type constructor \textit{FreeType} to complete the definition. However, \textit{FreeType} expects the arguments separately, whereas we have them in a pair.

We can convert a function on two arguments into one accepting a pair containing both arguments using \textit{uncurry}. After converting \textit{FreeType} to a pair-wise function, we apply it to the pair to form the \(Z\) paragraph.

\[
\text{structCToZ} = \text{uncurry FreeType} \circ (\text{structFreeTypeName} *** \text{map structDeclCToZ})
\]

We now define \textit{structDeclCToZ}, which considers individual struct member declarations. Each member translates to a simple \(Z\) free type branch.

\[
\text{structDeclCToZ} :: \text{Declaration} \to \text{ZBranch}
\]

\[
\text{structDeclCToZ} \text{ (Declaration _member)} = \text{ZBranch member Nothing}
\]

We apply a convention of prepending the free type name of a struct \(s\) with \('ST', hence \('STs'\). This transformation is applied to struct names via \textit{structFreeTypeName}, which is formalised in the appendices.

We next consider how to construct \textit{Circus} channels for the shared and external variables in the \textit{ACDC} program.

4.12.2 Channels

We define the channels corresponding to shared variables, internal and external channels from the \textit{ACDC} program using the function \textit{circusifyChannels}.

\[
\text{circusifyChannels} : \text{CircusifySubPhase}
\]

The function \textit{circusifyChannels} combines \textit{static channels}, which are part of our understanding of the cyclic executive model, and \textit{calculated channels}, which come from the \textit{ACDC} program itself. The static channels are \textit{frame}, which holds the current frame, and \textit{end_cycle}, which is used to force synchronisation at the end of each cycle.
To calculate the other channels, we first extract the variables needing channels from the input context with `channelVariables`, which we define later. This returns a pair of the variable's full name and optional Z type; we run the full name through `channelName` to convert from the variable name to that of its corresponding channel. To convert the pairs of channel name and type to `Circus` paragraphs, we use `channelPar`, which we elaborate later.

```haskell
circusifyChannels = map channelPar o (static++) o calculate
where static = [("frame", Just (Int Unsigned)), ("end_cycle", Nothing)]
calculate = map (first channelName) o channelVariables
```

We now define `channelVariables`, which extracts from an input context a list of pairs of channel name and optional type.

```haskell
channelVariables :: CircusifyInput -> [(FullName, Maybe Type)]
```

We find the channel variables by first taking the shared and external variables of the input, less the frame counter and constants. We then remove any duplicates in the resulting list with `nub`. We then map a fan-out, combining each variable name with its type from the input (if it has one), over the resulting list of channel variables.

```haskell
channelVariables input = (map (id &&& typeOf input) o nub) channels
where channels = (shared ++ external) \ \ [counter]
    shared = sharedVariables mains input \ \ consts
```

Many of the values above are extracted from the input context, as follows.

```haskell
constS = programConstantFullNames input
counter = frameCounter input
external = externalVariables input
mains = mainFunctions input
```

The functions `channelName` and `channelIdentifier` convert full names and identifiers to channel names respectively. The former is defined by composing latter with `flattenGlobalNameP`, which maps full names in the global scope down to single identifiers, and is undefined for other full names.

```haskell
channelName :: FullName -> ZIdentifier
channelName = channelIdentifier o flattenGlobalNameP
```

We adopt Ribeiro’s convention of constructing channel identifiers from variable identifiers by suffixing them with ‘SH’ (‘SHared variable’). This is captured by `channelIdentifier`, which we formalise in the appendices.

**Example.** The variable `var_1` would have a corresponding channel named `var_1SH`, and the variable `i` would be mapped to `iSH`. (End of example.)

The function `channelPar` takes a pair of channel identifier and optional type, and constructs the `Circus` paragraph of the corresponding channel.

```haskell
channelPar :: (ZIdentifier, Maybe Type) -> CircusPar
```

We first promote the identifier to a one-member sequence using `Base`. Then, we use the function `addType`, which converts the pair of name sequence and type to a typed or untyped channel, depending on whether the type is a value or `Nothing`. Finally, to make the channel into a `Circus` paragraph, we apply the type constructors `Base` and `CChannel`.

```haskell
channelPar = CChannel o Base o addType o first Base
```

When the channel has a type associated with it (`vtype` is not `Nothing`), we use a production of the `Circus` AST that adds in the Z expression equivalent of that type, producing a typed channel. Otherwise, we create an untyped channel. The function `addType'` selects the appropriate production by considering the optional type.

However, the function `addType'` has two parameters, the type and the sequence of channel names. We have these two parameters, but in the wrong order, and in a pair. This mismatch can be fixed by applying `flip` to `addType'`, swapping the two parameters’ positions, and `uncurry`, which converts the function to one accepting its two parameters in a pair. This corrected function is `addType`.

```haskell
where addType' = maybe UntypedChannel (flip TypedChannel o typeCToZ)
addType = uncurry (flip addType')
```

We now elaborate `sharedVariables`.

```haskell
sharedVariables :: [Identifier] -> CircusifyInput -> [FullName]
```
We discover the shared variables by considering, for each pair of main functions, their reachable variables. Any variables reachable in both functions is a shared variable. We concatenatively map `sharedInFunction` over the list of functions, to calculate the variables each function shares with other functions, and use `nub` to remove any duplicates.

\[
\text{sharedVariables fNames input} = \text{nub} \left( \text{concatMap} \text{sharedInFunction} \ fNames \right)
\]

where `sharedInFunction = \text{concatMap} \ \circ \ \text{sharedBetween} \ \circ \ \text{others}
\]

\[
\text{sharedBetween} = \text{intersect} \ \text{on} \ \text{reachable}
\]

\[
\text{reachable} \ x = \text{reachableVariables input} \left[ x \right]
\]

\[
\text{mains} = \text{mainFunctions input}
\]

We now consider the constants in an ACDC program.

### 4.12.3 Constants

The function `circusifyConstants` generates a Circus axiom describing the global-level constants:

\[
\text{circusifyConstants} :: \text{CircusifyInput} \rightarrow \text{Circus}
\]

We need to check to make sure there are constants in the `Decl`: if there are none, we need not generate an axiom. Otherwise, we build an axiom using `circusifyConstantsDecl` and `circusifyConstantsPredicate` to generate the declaration and predicates necessary. The decision as to whether to create the axiom is implemented by the list comprehension over `hasConstants`.

\[
\text{circusifyConstants input} = \left[ \text{axiomPar input} \mid \text{hasConstants (decl input)} \right]
\]

where `axiomPar = (\text{CPar} \ o \ \text{Axiom}) \ o \ (\text{ZSchemaText} \ \circ \ \text{circusifyConstantsDecl} \ \circ \ Just \ o \ \text{circusifyConstantsPredicate})`

We use `circusifyConstantsDecl` to construct the list of constant declarations. First, we use `programConstantIdentifiers`, which returns identifiers for all constants in an input context. Then, we map `constDecl` over the constant names. Since both these functions take the input context, we use the applicative functor syntax to make this extra argument implicit.

\[
\text{circusifyConstantsDecl} :: \text{CircusifyInput} \rightarrow [\text{ZDeclPart}]
\]

\[
\text{circusifyConstantsDecl} = \text{map} \ \circ \ \text{constDecl} \ \circ \ \text{programConstantIdentifiers}
\]

where `constDecl input name = name : constType input name`

\[
\text{constType input} = \text{variableZTypeError input} \circ \ \text{globalName}
\]

We now formalise `circusifyConstantsPredicate`.

\[
\text{circusifyConstantsPredicate} :: \text{CircusifyInput} \rightarrow \text{ZPredicate}
\]

We extract the `DeclProfiles` of all constants in the input context. We then convert these to predicates using `toPred`. Predicates form a monoid under newline-conjunction, so we map the conversion using `foldMap`, which concatenates the created predicates into a single predicate by the monoidal append (newline-conjunction). If there were no predicates, the conjoined predicate is `true`; however, this situation is never reached, as there can only be no predicates if there are no initialised constants, and we do not elaborate the axiom in this case. We use `return` to lift the result into a `Maybe` context.

We now elaborate `circusifyConstantsPredicate`.

\[
\text{circusifyConstantsPredicate input} = (\text{foldMap} \ \text{toPred} \ o \ \text{constantProfiles} \ o \ \text{decl}) \ input
\]

We define `toPred` on constant profiles with valid initialisers. We convert the initialiser into a Z expression using `initialiserToZ`, and create a predicate that the constant equals that expression. Any other type of profile found yields an error; this case is formalised in the appendices.

First, we convert the tree to a list of pairs of value and element, using the tree utility function `treeToPathList`. Then, we apply a folding map of `pathToPred`, which generates a predicate for each value assigned into the constant and conjoins them with newline-conjunction.

We define `pathToPred` as a split over `exprToCircus`, which changes the value to a Z expression, and `pathToCircusP`, which is a partial function that converts each path through the initialiser tree into a Z expression referring to the value being initialised. This is defined in terms of the more general utility function `convertInitialiserPath`, which takes functions for converting identifiers, arrays and structs, a struct map, the type of the constant, the initialiser path and the name of the constant. Note that we increment array indices by one when converting from ACDC to Z; this is a consequence of using Z sequences to model arrays.
Example. Consider the declaration `const signed int arr[4] = 1, 2, 3, 4`. The initialiser, after `treeToListPath`, will be converted to the list `[(1, 0), (2, 1), (3, 2), (4, 3)]`. The application of the split function yields `[(1, array1), (2, array2), (3, array3), (4, array4)]`, and the final list of predicates before newline-conjunction is `[array1 = 1, array2 = 2, array3 = 3, array4 = 4]`. (End of example.)

\[
\text{where toPred (DPConstant name d) | Just i ← declarationInitialiser d} \\
\quad \text{where pathToPred = equal o (exprToCircus *** pathToCircusP)} \\
\quad \text{equal = uncurry (flip (=))} \\
\quad \text{exprToCircus e = evalState (circusifyExpression input e) []} \\
\quad \text{pathToCircusP = fromJust o pathToCircus} \\
\quad \text{pathToCircus p = convertInitialiserPath ZIdExpr} \\
\quad \text{pArray = ZArray \ldots} \\
\quad \text{pStruct = ZStruct \ldots} \\
\quad \text{pArray index rec = ZApplication rec (ZNumber (index + 1))} \\
\quad \text{pStruct member rec = ZApplication rec (ZIdExpr member)}
\]

4.12.4 General processes

The function `circusifyProcesses` translates the ACDC main functions to Circus processes.

\[
\text{circusifyProcesses :: CircusifySubPhase}
\]

We do not consider the timer function here, as the semantics of the timer is different. We again make use of list comprehensions to select whether to elaborate a process based on whether it is not the timer function.

\[
\text{circusifyProcesses input} = \text{concatMap process (mainFunctions input)} \\
\text{where process f = [circusifyProcess input f | f \neq timerFunction input]}
\]

Single processes. The function `circusifyProcess` maps ACDC main functions to Circus processes.

\[
\text{circusifyProcess :: CircusifyInput \rightarrow Identifier \rightarrow CircusPar}
\]

We define `circusifyProcess` in terms of a more general function, `circusifyGeneralProcess`, as we use the same framework for the timer function. We pass a list containing the frame counter as the list of variables to hide from the process’s state.

\[
\text{circusifyProcess = generalProcess \$> foldMap o circusifyStatement \$> return o frameCounter \$> id}
\]

General processes. We now define `generalProcess`, which takes a function from statement lists to actions, a list of reachable variables to be hidden in the process state, an input context and the name of the function whose process is to be created, and returns the Circus paragraph of that process.

\[
\text{generalProcess :: ([Statement FullName] \rightarrow Action) \rightarrow [FullName] \rightarrow CircusifyInput \rightarrow Identifier \rightarrow CircusPar}
\]

If the function name cannot be looked up in the input context’s Decl, or belongs to a constant and not a function, `generalProcess` signals an error, which we formalise in the appendices. Otherwise, it generates a process definition for the function, using `circusifyProcessState` to generate the state, `circusifyProcessActions` to generate the process actions, and the given statement function on the result of converting the function body with `compoundToStatements` to generate the process main action.

\[
\text{generalProcess sf hidden input (declOf input \rightarrow Just (DPFunction function (Function body)))} \\
\quad = \text{simpleProcess function (ExplicitProc [] (circusifyProcessState input hidden function)} \\
\quad \text{(circusifyProcessActions input function)} \\
\quad \text{(sf (compoundToStatements (structs input) body)))}
\]
Process state. We define the state schema of a main function process using the function \( \text{circusifyProcessState} \).

\[
\text{circusifyProcessState} :: \text{CircusifyInput} \rightarrow [\text{FullName}] \rightarrow \text{Identifier} \rightarrow \text{ZSchemaExp}
\]

We take the reachable variables of the function, minus those that are to be hidden. Then, we convert each variable to a Z declaration by taking its type from the input context, converting its name to a Z identifier, and assembling the declaration via \( \text{singleTypeZDecl} \). We then assemble the schema expression using \( \text{SEText} \), \( \text{ZSchemaText} \), and no schema predicate.

\[
\text{circusifyProcessState} \text{ input hidden fun} = \text{SEText} (\text{ZSchemaText} (\text{mapMaybe reachables}) \text{ Nothing})
\]

where \( \text{toDecl i} = \text{fullNameToZ i} : : \text{variableZTypeError input i} \)

reachableV ariables input fun = \text{reachables = reachableVariables input} \text{ [fun]} \backslash \backslash \text{ hidden}

Process actions Each non-main function reached by a main function must be recorded as a C\text{ircus} action. We handle this with \( \text{circusifyProcessActions} \).

\[
\text{circusifyProcessActions} :: \text{CircusifyInput} \rightarrow \text{Identifier} \rightarrow [\text{PPar}]
\]

We take the reachability graph from the input context, and \( \text{mapMaybe} \) the partial function process to consider each pair therein. In process, we form a guard on the first element of the pair, which will be used to remove any graph pairs that are not reached from function. Taking the second element—the reached function—, we look-up its declaration profile in the input, which returns \( \text{Maybe DeclProfile} \), then lift and use \( \text{circusifyProcessAction} \) on the result. We then use an uncurried form of the monadic then operator to cause the guard to override the process action with \text{Nothing} if the graph pair did not concern function. We thus collect the process actions of every function in the graph that is reachable from function and has a declaration in the input. We here use \( \text{fmap} \), which is a function equivalent to \( \langle \& \rangle \).

\[
\text{circusifyProcessActions} \text{ input function} = \text{fmap toPar process (rgraph input)}
\]

where \( \text{process} = \text{uncurry (\&)} \circ (\text{guard} \circ (\exists \text{ function}) \& \& \text{ toPar}) \)

\( \text{toPar} = \text{fmap (circusifyProcessAction input)} \circ \text{declOf input} \)

The function \( \text{circusifyProcessAction} \) considers a single function. It is given the function in the form of a DeclProfile, and raises an error if the profile is not a function.

\[
\text{circusifyProcessAction} :: \text{CircusifyInput} \rightarrow \text{DeclProfile} \rightarrow \text{PPar}
\]

\( \text{circusifyProcessAction input (DPFunction name function)} = (\text{toPar} \circ \text{addProcessActionVars function} \circ \text{addLocalVariables function} \circ \text{actionFromStatements} \circ \text{statementsFromFunction}) \circ \text{functionBody} \)

where \( \text{toPar} = \text{PParAction name} \circ \text{PAction} \)

\( \text{statementsFromFunction} = \text{compoundToStatements (structs input)} \circ \text{functionBody} \)

If the function is a step function, we translate it into an action using \( \text{circusifyStep} \). Otherwise, we simply translate its statements.

We assume a function is a step function if, and only if, its name ends with ‘_step’.

\[
\text{actionFromStatements} \mid \text{isStep} = \text{circusifyStep input name}
\]

| otherwise = \text{foldMap (circusifyStatement input)}

\( \text{isStep} = "\_\_\text{step}" \circ \text{isSuffixOf} \circ \text{name} \)

The function \( \text{addLocalVariables} \) adds a local variable block to a process action. It places the action inside a var declaration if, and only if, there is at least one local variable in the function.

\[
\text{addLocalVariables} :: \text{Function FullName} \rightarrow \text{Action} \rightarrow \text{Action}
\]

We define \( \text{addLocalVariables} \) in terms of a stage \( \text{findLocals} \) that collects the local variables from the function, and \( \text{addVarBlock} \) that converts the local variable list into a function on Actions that adds the appropriate variable block.

\[
\text{addLocalVariables function} = \text{addVarBlock (findLocals function)}
\]

In \( \text{findLocals} \), we traverse the function body, collecting each declaration. The traversal \( \text{traverseCompoundTDE} \) recursively considers each declaration in compound statements inside the function body: thus, we find all local variables in the function.

\[
\text{where} \text{ findLocals} = \text{traverseCompoundTDE (const []) return (const [])} \circ \text{functionBody}
\]

In \( \text{addVarBlock} \), we convert the list of variables to a sequence using \( \text{toSeq} \). This returns the sequence in a Maybe context, being \text{Nothing} if the list was empty. We can thus use \( \text{maybe} \) to make \( \text{addLocalVariables} \) be the identity on actions if there were no local variables, and construct a variable block if there were some.

\[
\text{addVarBlock} \circ \text{toSeq}
\]

\( \text{makeVarBlock vs} = \text{ACCommand} \circ \text{CVarBlock (makeVarEntry vs)} \)

\( \text{makeVarEntry} = \text{return} \circ (\text{Var,}) \circ \text{ZDeclPart} \circ \text{fmap toZDecl} \)
The nested function \texttt{toZDecl} constructs the Z declaration for one local variable. We fan-out the parameter to two functions, \texttt{vName} and \texttt{vType}, which produce a one-value sequence containing the variable name and the Z type of the variable respectively. These are the two arguments we need to pass to \texttt{BasicDecl}, but they are inside a pair and thus cannot be used directly as arguments, so we use \texttt{uncurry}.

\begin{align*}
  \text{toZDecl} &= \text{uncurry } \text{BasicDecl } \circ (\text{vName } \&\& \& \text{vType}) \\
  \text{vName} &= \text{Base } \circ \text{fullNameToZ } \circ \text{declarationName} \\
  \text{vType} &= \text{typeCToZ } \circ \text{dsType } \circ \text{declarationSpecs}
\end{align*}

The function \texttt{addProcessActionVars} adds variable declarations to a process action. It constructs \texttt{val} declarations for any parameters to the action, and a \texttt{res} declaration for any result to be retrieved from the process action. The parameters, and whether a result is returned, are extracted from the function declaration.

\texttt{addProcessActionVars :: Function FullName }\to\text{ Action }\to\text{ Action}

We build the \texttt{val} and \texttt{res} aspects of the variable declaration separately, using \texttt{val} and \texttt{res} respectively, creating a list of two \texttt{Maybe} values. We use \texttt{catMaybes} to remove the \texttt{Maybe} context, by removing instances of \texttt{Nothing} from the list. We now have a list containing the \texttt{val}, the \texttt{res}, both, or neither, which we pass to \texttt{addVars} to introduce the variable block.

\begin{align*}
  \text{addProcessActionVars } &= \text{addVars } \circ (\text{val } \&\&\& \text{res}) \circ \text{functionSignature} \\
  \text{addProcessActionVars } &= \text{addVars } \circ \text{vs; rs}
\end{align*}

If neither the \texttt{val} nor the \texttt{res} are present, then we do not introduce a variable block. Otherwise, we do, using a composition of \texttt{ACommand} and \texttt{CVarBlock}. We use \texttt{catMaybes} to construct a list containing the present values from \texttt{(vs, rs)}.

\begin{align*}
  \text{where addVars (Nothing, Nothing) = id} \\
  \text{addVars (vs, rs) = ACommand } \circ \text{CVarBlock (catMaybes [vs, rs])}
\end{align*}

We retrieve the parameters from the signature with \texttt{signatureParameters}. Then, we convert the parameters to a sequence using \texttt{toSeq}: this returns the sequence in a \texttt{Maybe} context, with the value \texttt{Nothing} if the sequence was empty. We then lift the function \texttt{valFromSeq}, which converts a sequence of parameters into a \texttt{val} entry, into this context. Thus, the final \texttt{val} will exist (not be \texttt{Nothing}) if, and only if, there were parameters in the signature.

The function \texttt{valFromSeq} maps \texttt{paramDecl} over each item in the sequence. It then constructs a \texttt{htypeZDeclPart} from the resulting sequence, and uses a \texttt{tuple section} to place the declaration into the second part of a pair whose first part is \texttt{Val}. The tuple section notation is a Glasgow Haskell Compiler extension to Haskell.

\begin{align*}
  \text{val} &= (\text{valFromSeq<\$>}) \circ \text{toSeq } \circ \text{signatureParameters} \\
  \text{valFromSeq} &= (\text{Val, } ) \circ \text{ZDeclPart } \circ (\text{paramDecl<\$>})
\end{align*}

If the return type in the signature is \texttt{void}, then there is no result, and \texttt{res} returns \texttt{Nothing}. Otherwise, a \texttt{res} declaration is introduced for the return value, which we assign to the variable \texttt{_result}.

\begin{align*}
  \text{res (signatureType } \to t) &= (\text{Res, "_result" : typeCToZ t ) } \&\& \text{guard (t } \neq \text{ Void)}
\end{align*}

The nested function \texttt{paramDecl} constructs the Z declaration for one parameter. It is similar to \texttt{toZDecl} in the local variables construction, but we use \texttt{parameterName} and \texttt{parameterSpecs} in place of their declaration equivalents.

The function \texttt{compoundToStatements} converts a compound statement, with declarations and statements, into a list of only statements. It converts each declaration to an equivalent initialising statement using \texttt{declarationsToStatements}.

\begin{align*}
  \text{compoundToStatements :: StructMap } \to\text{ Compound FullName }\to\text{ [Statement FullName]} \\
  \text{compoundToStatements snap (Compound ds ss) = declarationsToStatements snap ds ++ ss}
\end{align*}

\subsection{4.12.5 The timer process}

The sub-phase \texttt{circusifyTimer} introduces the timer process. We use \texttt{generalProcess} as in \texttt{circusifyProcess}, but here the process action is elaborated by \texttt{timerAction}, the name by \texttt{timerFunction}, and the set of hidden reachable variables is always empty.

\begin{align*}
  \text{circusifyTimer :: CircusifySubPhase} \\
  \text{circusifyTimer = return } \circ (\text{generalProcess } \circ\$\text{ timerAction } \circ\$\text{ const } [ ] \circ\$ \text{id } \circ\$ \text{timerFunction})
\end{align*}

We handle the statements in the timer process using \texttt{timerAction}. We summarise its behaviour here, but formalise it in the appendices. The timer action is elaborated differently based on whether the timer function begins with a delay statement. If it does, we replace the delay with a high-level model of a cyclic executive timer, including a write to the frame counter and an end-of-cycle synchronisation, and then consider the remaining statements with \texttt{circusifyStatements}. This replaces the concrete implementation of the timer with a more shallow, high-level \texttt{Circus} model of its rôle in the cyclic executive.

Otherwise, we consider each statement in the timer function separately. Whenever we find an infinite loop, we assume it contains a delay-based timer as characterised above, and thus recursively consider that infinite loop as a timer action, adding operations to initialise and increment the frame counter if they are not already made explicit in the program itself. Otherwise, we translate the statement as normal, using \texttt{circusifyStatement}.
4.12.6 The program process

The function `circusifyProgram` produces the Circus sub-model that represents the overall program as a process.

It is defined as the parallel composition, made by `makeParallel`, of the main function processes, with all internal channels and the frame count channel hidden from the environment.

```plaintext
circusifyProgram :: CircusifySubPhase
circusifyProgram input = [simpleProcess "Program" procDef]
  where procDef = (hidemakeParallel input)
    hidemakeParallel = fromMaybe (error "Need >=1 process. ")  
```

We now elaborate `hidemakeParallel`, which adds the channel-hiding clause to the program. We hide all channels that are not representing external variables, as well as the `frame` channel.

```plaintext
hidemakeParallel :: CircusifyInput -> Proc -> Proc
hidemakeParallel = (hidemakeParallel)  
  where hidden = "frame" : map channelName (channels \\ externals)  
        channels = (map fst \ channelVariables) input  
        externals = externalVariables                  
```

We now elaborate `makeParallel`.

```plaintext
makeParallel :: CircusifyInput -> [Identifier] -> Maybe Proc
makeParallel input = (\makeParallel)  
  where makeParallel = fromMaybe (error "Need >=1 process. ")  
```

We map `parallelPairFromFunction` over each main function name, producing a list of pairs of process and synchronisation alphabet. We then use `toList` to produce `Nothing` if the list is empty, or the equivalent sequence inside `Just` if it is not. Then, we lift `PParallel`, which constructs a parallel composition of processes from a sequence of process–alphabet pairs, into the `Maybe` context. Thus, we either have `Nothing`, or the desired parallel composition.

```plaintext
makeParallel input = (\fmap PParallel)  
  where parallelPairFromFunction = PNamed \&\& CSExp \&\& alphabet  
```

To construct the process’s alphabet, we first discover its reachable variables by promoting the process name to a list and supplying it to `reachableVariables`. We then take the intersection of this list with the variables in the program that have channels, and convert the resulting list of reachable variables with channels into a list of channel names by mapping `channelName` over them. Finally, we add the two channels that are always in the alphabet of by processes in our architecture: ‘frame’ and ‘end_cycle’.

```plaintext
alphabet = (\((\map channelName o cV arsOnly o rV ars o return\)) input  
```

4.12.7 Step functions

The step functions are where our translation process deviates the most from standard ACDC semantics. This is because the step functions are intimately linked with the cyclic executive architecture, thus their translation must incorporate the semantics of that architecture at a high level.

We cater for two styles of step function: a frame scheduler based function, which selects the appropriate task by checking the frame counter via a switch statement, and a delay-based step function, which uses delaying statements to keep in synchronisation with the frames of the process. Of these two styles, we only fully elaborate the former, and translate the latter by first introducing a frame scheduler.

The function `circusifyStep` translates the body of a step function to an action.

```plaintext
circusifyStep :: CircusifyInput -> Identifier -> [Statement FullName] -> Action
```

We leave the precise definition of `circusifyStep` to the appendices. Informally, `circusifyStep` first creates a frame scheduler if none already exists. We assume that a frame scheduler exists if, and only if, the first statement of the step function is a switch statement. We introduce one by splitting the frame scheduler body at each delay statement, moving the resulting partitions into new compound statements, and building the scheduler switch statement from those compounds starting with the first frame.

Next, the frame scheduler is translated into Circus as with any other statement, using `circusifyStatement`. This produces a guarded if block whose guards each activate on different frames. We then introduce an input communication on the frame channel, to provide the step function with the correct frame count and synchronise it with the timer.

The step function is responsible for co-ordinating the reading of external inputs for its process at the start of the frame, the writing of external outputs at the end of the frame, and the communication, over channels, with other step functions to synchronise the processes’ local copies of shared variables. This last requirement is achieved via the reading and writing, in interleaving, between channels and local variables, such that the step function writes values that other step functions need to
read, and reads values that other step functions have written to in the interim. These concerns are handled by \texttt{circusifyStepSharedGAction}, which we formalise below.

\[
\text{circusifyStepSharedGAction} :: \text{CircusInput} \to \text{Identifier} \to \text{GAction} \to \text{GAction}
\]

Using Haskell’s pattern matching, we can reach into the guarded action’s predicate and extract the frame number; we also fail if we discover that the guarded action is not a supported form of frame scheduler, and do nothing if we reach an empty default clause.

The first pattern considered is a non-empty action guarded on a particular frame being active. We proceed to change the guarded action, using \(\ast \to \) as shorthand for introducing the modified action \(\text{GAction}\).

\[
\text{GAction} p\bullet(PInfix \text{ Equality } (\text{ZIdExpr } \_ ) (\text{ZNumber frameNum})) \to p \to \text{action}'
\]

The new action depends on whether the frame on which this action is triggered is the final frame. If it is, we append a communication to force synchronisation on the end of the cycle. We here introduce the \textit{monoidal append} operator \(\circ\), which is a generalisation of list appending to monoids; in this case, as actions are a monoid under sequential composition, it represents the sequential composition of its operands.

\[
\text{where action'} \mid \text{frame ‘lastFrameIn’ count } = \text{mainAction } \circ \text{endCycle}
\]

\[
\mid \text{otherwise } = \text{mainAction}
\]

We now consider \(\text{mainAction}\). This consists of a sequential composition of a phase of reading and writing, the original action, and the act of writing back to the channels:

\[
\text{mainAction } = \text{preSync } \circ \text{action } \circ \text{postSync}
\]

We now consider \(\text{preSync}\) and \(\text{postSync}\). The former is an interleaving of a write phase, communicating any variable whose current value is held by this step function and whose value is sought by another step function, and a read phase, accepting said communications from step functions and external channels holding required values.

\[
\text{preSync } = \text{interleave } (\text{writeSync } \cup \text{readSync})
\]

\[
\text{readSync } = \text{makeReadInterleave } (\text{vRead ‘union’ vIn})
\]

\[
\text{writeSync } = \text{makeWriteInterleave } vWrite
\]

The latter is a second write phase, communicating the external outputs written during the step function.

\[
\text{postSync } = \text{interleave } (\text{makeWriteInterleave } vOut)
\]

We now consider the creation of the variable sets \(\text{vRead}, \text{vWrite}, \text{vOut},\) and \(\text{vIn}\). These are handled by functions \(\text{variablesToRead}, \text{variablesToWrite}, \text{variablesToOutput}\) and \(\text{variablesToInput}\) respectively. These four functions take a common set of outputs and are composed with the same post-processing functions; this common boilerplate is handled by \(\text{makeV}\). The last two cases require an extra argument: the set of all external outputs and inputs in the program respectively.

\[
[\text{vRead}, \text{vWrite}, \text{vOut}, \text{vIn}] = \text{map makeV}
\]

\[
[\text{variablesToRead}, \text{variablesToWrite}, \text{variablesToOutput} \text{extOut}, \text{variablesToInput} \text{extIn}]
\]

\[
\text{makeV } f = \text{map makePair } (f \text{vmap count frame function})
\]

Finally, \(\text{makePair}\) is a fan-out into a pair of the input variable converted to a channel identifier, and the input.

\[
\text{makePair } = \text{channelIdentifier } \& \& \& \text{id}
\]

The values \(\text{extIn}, \text{extOut}, \text{vmap}\) and \(\text{count}\) are the external inputs and outputs, \(\text{VMap}\), and frame count from the input context respectively.

The other considered case corresponds to the default case of the switch statement, must be empty, and is ignored. If neither of the two considered patterns match, the process fails.

We now consider the construction of interleavings. First, we elaborate \textit{interleave}, which takes a list of pairs of actions and alphabets and returns \(\text{Skip}\) if the list is empty, and the interleaving of those actions on those alphabets otherwise.

\[
\text{interleave} :: ((\text{Action}, \text{NSExp})] \to \text{Action}
\]

\[
\text{interleave } = \text{maybe skip } (\text{ACSPAction } \circ \text{CIInterleave }) \circ \text{toSeq}
\]

We now consider the creation of the lists of action–alphabet pairs. Read and write interleavings are implemented in terms of \(\text{makeInterleave}\), which takes a function converting a pair of channel and variable into a read or write action and a function that processes the synchronisation sets for the interleave, and returns a function from lists of channel–variable pairs to a pair of the interleaving action and all variables being synchronised. We first create a type synonym, \textit{Interleaver}, for functions that produce interleave pairs.

\[
\text{type Interleaver } = [(\text{ZIdentifier}, \text{ZIdentifier})] \to [(\text{Action}, \text{NSExp})]
\]
Read-interleaves. In read-interleaves, we must compute synchronisation sets for each interleaving pair. This is handled by `makeInterleave`; we provide the function `id` to instruct `makeInterleave` to pass its computed sets into the interleaving without modification or discarding.

```haskell
makeReadInterleave :: Interleaver
makeReadInterleave = makeInterleave makeRead id
    where makeRead channel var = (channel .? "x") \rightarrow (var = ZIdExpr "x")
```

Write-interleaves. Unlike read-interleaves, we do not need synchronising alphabets. Thus, the function for handling synchronisation sets always returns the empty list, and we discard the final set of variables using `fst`.

```haskell
makeWriteInterleave :: Interleaver
makeWriteInterleave = makeInterleave makeWrite (const [])
    where makeWrite channel var = (channel ! ZIdExpr var) \rightarrow skip
```

Interleaves in general. We now elaborate `makeInterleave`. This maps a fan-out of `actionF`, uncurried to accept a pair, and `makeNS`, across the set of pairs of channel and variable.

```haskell
makeInterleave :: (ZIdentifier ! ZIdentifier ! Action) \rightarrow ([ZIdentifier] \rightarrow [ZIdentifier]) \rightarrow Interleaver
makeInterleave actionF syncF = map (uncurry actionF &&& makeNS)
    where makeNS = NSExp syncF return snd
```

We now elaborate `circusifyVMap`. Since every argument to `functionMapToVMap` is the result of applying a function to the input, we can make this definition more concise by using applicative functor syntax. This effectively lifts `functionMapToVMap` such that it takes the input argument and distributes it to each of its own arguments.

```haskell
circusifyVMap :: CircusifyInput \rightarrow VMap
circusifyVMap = functionMapToVMap <$> map dpId constantProfiles <*> rgraph <*> vusage <*> functionMap
functionMapToVMap constantNames r v = concatMap vmapFunction
    where vmapFunction (step; frame; call) = (map makeEntry \filter isNotConstant) vmapMapping
        where makeEntry = (step, frame,)
            vmapMapping = reachableVUsage r [call] v
        isNotConstant = (\(gmap globalName constantNames) \circ variableName
```

The functions considering sets of channel synchronisations have a common shape: we introduce a type synonym, `SyncFunction`, to represent it.

```haskell
type SyncFunction = VMap \rightarrow FrameCount \rightarrow Frame \rightarrow Identifier \rightarrow [ZIdentifier]
```

We first consider the calculation of sets of variables representing external inputs that must be read by a step function, and external outputs that must be written back. This is handled by `variablesToInput` and `variablesToOutput` respectively.

These functions are similar, differing only in their inputs and whether they consider the variables in the VMap that are being read (in the case of external inputs) or written (in the case of external outputs). Thus, we factor the similarities out into a separate function, `variablesToSyncWithEnv`.

```haskell
variablesToInput, variablesToOutput :: [FullName] \rightarrow SyncFunction
variablesToInput = variablesToSyncWithEnv Read
variablesToOutput = variablesToSyncWithEnv Write
variablesToSyncWithEnv :: Status \rightarrow [FullName] \rightarrow SyncFunction
    = map fullNameToZ
```
A shared variable \( v \) should be written to a channel by a step function \( s \) on a frame \( f \) if there is another step function \( t \) that is reading \( v \) on \( f \), and the last function to write to \( v \) was \( s \). Similarly, \( v \) should be read from a channel by \( s \) on \( f \) if \( s \) is reading \( v \) on \( f \), and the last function to write to \( v \) was \( s \).

Since these two cases differ only in two ways—the inversion of the logic concerning functions and the inclusion of external inputs into the read case—the formalisation of \( \text{variablesToRead} \) and \( \text{variablesToWrite} \) is implemented in terms of a general function, \( \text{variablesToSync} \). The difference in function logic is handled by taking the appropriate operator for comparing the function \( s \) to any functions reading \( v \) on \( f \), which is equality for reads and non-equality for writes, and \( \text{variablesToRead} \) takes an extra parameter containing the list of external inputs to add to the result of \( \text{variablesToSync} \). We do not need to provide another operator for comparing \( s \) to the last function to write \( v \), as it is always the negation of the operator for comparing \( s \) to functions reading \( v \) on \( f \).

\[
\text{variablesToSync} :: (\text{Identifier} \to \text{Identifier}) \to \text{SyncFunction}
\]

\[
\text{variablesToSync} \circ \text{op} \vmap \text{count} \text{frame} \text{function}
\]

\[
\text{where} \quad \text{names} = \text{variablesInVMap} \circ \text{fPred} \circ \text{Read} \circ \vmap
\]

\[
\text{vars} = \text{variablesPerFrame} \circ \text{names} \circ \vmap
\]

\[
\text{access} = \text{lastAccessedByFunction} \circ \text{op} \circ \text{fPred} \circ \text{count} \text{frame}
\]

\[
\text{fPred} = \text{op} \circ \text{function}
\]

We now define \( \text{variablesToSync} \). We first select the variables from the \( \text{VMap} \) that have been read in the current frame, and satisfy the function predicate. Recall that for the reading set, the reading function must be the step function; for the writing set, it must be another function. We then use \( \text{variablesPerFrame} \) to find the variables from this set that have been written by some function, and return them as a list of pairs containing a variable name and a list of frame–function writing pairs. Then, we further filter this list to those variables that were last written by the current function (for writing sets) or a different function (for reading sets). Finally, we extract the variable names from the remaining pairs, and convert them to \( \text{Z} \) identifiers.

\[
\text{variablesToRead}, \text{variablesToWrite} :: \text{SyncFunction}
\]

\[
\text{variablesToRead} = \text{variablesToSync} \circ \text{=} \text{fSy}
\]

\[
\text{variablesToWrite} = \text{variablesToSync} \circ \text{̸=} \text{fSy}
\]

We now define \( \text{variablesPerFrame} \). We first select all \( \text{write} \) records from the \( \text{VMap} \) that reference a variable in our list of names. We then apply the utility function \( \text{coalesce} \), formalised in the appendices, which takes a flat list of tuples of variable name, frame and writing function, and groups them by name, producing a list of pairs of name and lists of frame and function.

\[
\text{variablesPerFrame} :: [\text{FullName}] \to \text{VMap} \to [[(\text{Frame}, \text{Identifier})]]]
\]

\[
\text{variablesPerFrame} \circ \text{vnames} = \text{coalesce} \circ \text{concatMap} \text{translate}
\]

\[
\quad \text{where} \quad \text{translate} = (\text{function}, \text{frame}, \text{VariableRecord} \circ \text{var} \circ \text{vStatus}) \mapsto [(\text{var} \circ \text{frame} \circ \text{function}) \mid \text{Write} \equiv \text{vStatus} \land \text{var} \in \text{vnames}]
\]

The function \( \text{lastAccessedByFunction} \) takes a predicate concerning a function name, the frame count of the scheduler and the current frame. It then, given a list of pairs of frames in which a variable is accessed and functions performing that access, finds the function that last accessed the variable before the current frame, and returns the result of the predicate over its name.

\[
\text{lastAccessedByFunction} :: (\text{Identifier} \to \text{Bool}) \to \text{FrameCount} \to \text{Frame} \to ((\text{Frame}, \text{Identifier}) \to \text{Bool})
\]

\[
\text{lastAccessedByFunction} \circ \text{correctFunction} \circ \text{count} \text{frame} \text{function} \partial \text{readers}
\]

\[
= (\text{lastAccessFrame} < \text{thisFrame}) \land
\]

\[
\text{coalesce} \text{frame} \text{function} \partial \text{readers}
\]

\[
= \text{coalesce} \text{frame} \text{function} \partial \text{readers}
\]

\[
\text{variablesToWrite} \circ \text{lastAccessedByFunction} \circ \text{correctFunction} \circ \text{count} \text{frame} \text{function} \partial \text{readers}
\]

\[
\text{variablesToRead} \circ \text{lastAccessedByFunction} \circ \text{correctFunction} \circ \text{count} \text{frame} \text{function} \partial \text{readers}
\]

\[
\text{variablesToSync} \circ \text{lastAccessedByFunction} \circ \text{correctFunction} \circ \text{count} \text{frame} \text{function} \partial \text{readers}
\]

\[
\text{variablesToWrite} \circ \text{lastAccessedByFunction} \circ \text{correctFunction} \circ \text{count} \text{frame} \text{function} \partial \text{readers}
\]

\[
\text{variablesToRead} \circ \text{lastAccessedByFunction} \circ \text{correctFunction} \circ \text{count} \text{frame} \text{function} \partial \text{readers}
\]

\[
\text{variablesToSync} \circ \text{lastAccessedByFunction} \circ \text{correctFunction} \circ \text{count} \text{frame} \text{function} \partial \text{readers}
\]
We elaborate the case of function calls below.

\[
\begin{align*}
\text{circusifyExpression} &:: \text{CircusifyInput} \\
&\rightarrow [\text{Expression FullName}] \\
&\rightarrow \text{State} [\text{ZFunctionCall}] [\text{ZExpression}] \\
\text{circusifyExpression} &\text{= mapM o circusifyExpression}
\end{align*}
\]

The function \textit{circusifyExpression}, then, is the stateful computation considering a single expression.

\[
\begin{align*}
\text{circusifyExpression} &:: \text{CircusifyInput} \\
&\rightarrow \text{Expression FullName} \\
&\rightarrow \text{State} [\text{ZFunctionCall}] \text{ ZExpression} \\
\text{circusifyExpression input expression} &\text{= case expression of}
\end{align*}
\]

We elaborate much of \textit{circusifyExpression} in the appendices, as it is better explained here in summary. The activity of \textit{circusifyExpression} on each type of expression is as follows:

**Constants** These are converted to Z expressions using \textit{constantToZ}, and returned with no state changes.

**Array subscripts** We translate the array expression first, then the array subscript expression, with the monadic bind ensuring function calls are sequenced in that order. We then add one to the subscript expression, to map from array indices to sequence indices, and represent the subscript as a Z function application of the subscript to the array.

**Struct members** We translate the struct expression, then apply the identifier expression of the struct member to it.

**Function calls** We elaborate the case of function calls below.

**Identifiers** These are converted to Z expressions using \textit{fullNameToZ} and \textit{ZIdExpr}, and returned with no state changes.
Prefix and infix expressions  We convert the operator to its Z equivalent, apply this operator to the type constructor for Z
prefix or infix expressions respectively, and then use applicative functor syntax to lift the resulting function to a stateful
computation and apply the operand computations to the lifted function.

Strings  These are converted to a sequence of Z integers, each representing the ASCII code-point of its respective character.

We now consider function calls in detail. If the function is a delay function, we ignore it and return an arbitrary value. If the
temporal abstraction phase was sound, this value will never be reached.

\[(\text{CallExpr} \ (\text{FC} \ \text{name} \ \text{arguments})) | \text{isDelayFunction} \ \text{name} \to \text{return} \ (\text{ZNumber} \ 0)\]

Otherwise, we operate in two stages. First, we alter the state of our computation by adding in the new function call, using
addCall. Next, we create and return the Z expression corresponding to the variable that will, in the action in which this
expression exists, hold the result of that function call, using makeResultExpression. The two stages are separated by the monadic
then operator \(\gg\), which sequences addCall before makeResultExpression, but only considers the former’s changes to the state
of the computation and discards any value it may have.

\[
\text{addCall} = \text{append} \gg \text{(callFromArguments} \ <$\text{circusifyArguments}$) \\
\text{callFromArguments} = \text{[ZFunctionCall name as hasReturn]} \\
\text{circusifyArguments} = \text{circusifyExpressions input arguments} \\
\text{append call} = \text{modify} \ (++\text{call}) \\
\text{makeResultExpression} = \text{(ZIdExpr \ currentResultId} \ <$\text{get}$) \\
\text{currentResultId} = \text{functionCallResultIdentifier} \ \text{length}
\]

A function has a return value if, and only if, it does not return the void type. We check this by finding the function’s return
type via the input context’s type map, and comparing it against Void.

\[
\text{hasReturn} = \text{maybe (error "Could not find type." \ (\# Void) (input ‘typeOf’ globalName name)}
\]

4.12.9 Statements

The function circusifyStatement converts an ACDC statement into a Circus action.

\[
\text{circusifyStatement} :: \text{CircusifyInput} \to \text{Statement FullName} \to \text{Action}
\]

Wherever an expression is found inside a statement, we receive the resulting translation hidden inside a stateful computation.
We must run the computation with an initial state to produce a pair of final state (the final function call list) and value (the
translated expression). In general, we approach this by performing the statement translation inside the stateful context, and
then using the function circusifyFunctionCalls to extract the translated statement, sequence any function calls from expressions,
and wrap the statement inside a var block binding the call results to their result variables. The initial state is provided by
circusifyFunctionCalls, and is the empty function call list.

As with expressions, it is generally more convenient to outline the approach taken for translating statements, and formalise
in the appendix. The effect of circusifyStatement on each type of statement is as follows:

Null statements and empty return statements  These are translated to Skip.

Valued return statements  We apply circusifyExpression to the value, attach it to a guarded assignment command giving its
value to _result, and sequence any function calls.

Expression statements  As their only effect is via their function calls, we sequence the expression calls over Skip.

Assignments  These are handled by circusifyAssignment.

Compound statements  These are handled by circusifyCompound.

If statements  These are handled by circusifyIfThenElse.

Switch statement  We first translate the switch expression, then handle the cases using circusifySwitchCases. These translate
easily into guarded if statements.

Infinite while loops  These translate into infinitely recurring fixed points. The body is translated using circusifyCompound.

General while loops  These are handled similarly to the above, but the body and recursion are placed in the true branch of a
call to circusifyIfThenElse, where the condition is the while condition and the false branch is Skip.

Do-while and for loops  These are first translated into equivalent while-loop based constructs, and then elaborated as above.
Switch statements. These are handled by \textit{circusifySwitchCases}.

\begin{align*}
circusifySwitchCases &:: \text{CircusifyInput} \\
& \rightarrow \text{ZExpression} \\
& \rightarrow \text{SwitchCases FullName} \\
& \rightarrow [\text{ZExpression}] \\
& \rightarrow \text{State [ZFunctionCall] (Seq1 GAction)}
\end{align*}

We handle non-default switch cases by converting the condition expression to a Z expression and function call list. We construct a guard that is true if, and only if, that Z expression equals its condition, and recurse on the remaining switch cases.

\begin{align*}
circusifySwitchCases \text{ input expr (Case cond compound rest) seen} &= \text{makeCase} \left\llll \text{makeCond} \\
\text{where makeCond} &= \text{circusifyExpression input cond} \\
\text{makeCase condZ} &= \text{Inductive (caseAction condZ) } \iff \text{nextActions condZ} \\
\text{nextActions condZ} &= \text{circusifySwitchCases input expr rest (seen ++ [condZ])} \\
\text{caseAction condZ} &= \text{GAction (expr 'predEqual' condZ)} \\
& \quad \text{(circusifyCompound input compound)}
\end{align*}

The default case evaluates to an action with a guard that is true if, and only if, the value of the switch expression is not any of the previously seen switch cases. Since the default case has no condition expression, it yields no function calls.

\begin{align*}
circusifySwitchCases \text{ input expr (Default compound) seen} &= \text{return (Base (notSeen } \iff \text{circusifyCompound input compound))} \\
\text{where notSeen} &= \text{expr 'predNotIn' ZSetLiteral seen}
\end{align*}

If-then-else. We translate if-then-else statements using \textit{circusifyIfThenElse}. It takes the input context, an expression, an action to apply if the expression is true and an action to apply if the expression is false.

We first convert the expression to a Z expression, then use \textit{expressionPredicate} to construct a predicate that is true if, and only if, the Z expression evaluates to true using \textit{ACDC}'s semantics. Then, we construct the two guarded commands for the if and else legs of the statement, combine them into a guarded if action using \textit{ifActionList}, and sequence the function calls from the expression over the result.

\begin{align*}
circusifyIfThenElse &:: \text{CircusifyInput} \rightarrow \text{Expression FullName} \rightarrow \text{Action} \rightarrow \text{Action} \rightarrow \text{Action} \\
circusifyIfThenElse \text{ input cond ifLeg elseLeg} &= \text{circusifyFunctionCalls action} \\
\text{where action} &= \text{actionFromPredicate } \iff \text{ifPredicate} \\
\text{actionFromPredicate p} &= \text{ifActionList [ifCircus p, elseCircus p]} \\
\text{ifCircus p} &= p \iff \text{ifLeg} \\
\text{elseCircus p} &= \text{pnot p } \iff \text{elseLeg} \\
\text{ifPredicate} &= \text{expressionPredicate } \iff \text{ifExpression} \\
\text{ifExpression} &= \text{circusifyExpression input cond}
\end{align*}

Compound statements. These are handled by \textit{circusifyCompound}.

\begin{align*}
circusifyCompound &:: \text{CircusifyInput} \rightarrow \text{Compound FullName} \rightarrow \text{Action} \\
\text{circusifyCompound input } (\text{Compound decls stms}) &= \text{foldMap (circusifyStatement input stms)} \\
\text{where stms'} &= \text{declarationsToStatements (structs input) decls ++ stms}
\end{align*}

Expressions to predicates. We convert Z captures of ACDC expressions to Z predicates using \textit{expressionPredicate}.

\begin{align*}
\text{expressionPredicate} &:: \text{ZExpression} \rightarrow \text{ZPredicate} \\
\text{expressionPredicate} &= \text{pnot o ('predEqual' ZNumber 0)}
\end{align*}
Converting declarations to statements. The function `declarationsToStatements` generates a series of assignment statements capturing the initial assignments present in a given list of declarations. It requires a `StructMap`, as it must map initialiser lists for structs back to the struct members given.

```
declarationsToStatements :: StructMap -> [Declaration FullName] -> [Statement FullName]
```

We define `declarationsToStatements` as a concatenative mapping of `declarationToStatement`.

```
declarationsToStatements smap = concatMap (declarationTo Statements smap)
```

The function `declarationToStatement` maps a declaration to a list of statements.

```
declarationToStatement :: StructMap ! Declaration FullName ! [Statement FullName]
```

Our goal is to replace the initialisers in a declaration with explicit assignments, so we must deal with the potential for the initialiser to be absent. Since this case is handled by the initialiser taking a `Maybe` value, we can use the `maybe` function to return the empty list when an initialiser is absent, and apply a further function if it is present:

```
declarationToStatements smap (Declaration ds name ini) =
   maybe [] initialiserToStatements ini
   where initialiserToStatements = map toStatement . treeToPathList
        toStatement (value, path) = Assignment (AssignValue AOpEquals (target path) value)
        target path = fromJust (treeTarget smap (dsType ds) path name)
```

4.12.10 Assignments

The function `circusifyAssignment` translates ACDC assignments into Circus actions.

```
circusifyAssignment :: CircusifyInput -> Assignment FullName -> Action
```

Self-assignment. These are translated into their equivalent value assignments (target++ becomes target += 1, and target -- becomes target -= 1). The appropriate value-assignment operator is selected by `circusifyAssignSelfOperator`.

```
circusifyAssignment input (AssignSelf op target) =
   circusifyAssignment input (AssignValue vop target (ConstantExpr (CInt 1)))
   where vop = circusifyAssignSelfOperator op
```

Value-assignment. To translate these, we find the name of the variable being assigned to using the record accessor `assignmentName`. We then translate the value expression to using `circusifyExpression`, noting the resulting Circus expression and list of function calls used in that expression.

If the assignment changes an array index or struct member of the assigned variable, we reflect this by having the Circus assign assignment to the variable the variable’s current value overridden with the index or member changed; this technique recurses if we are assigning to an index of an index, and so on. The function `circusifyAssignmentRhs` implements this translation, returning the corrected assignment value and any function calls from array indices in the assignment target.

We build the assignment action from the variable name and translated value using the operator, then add the function calls from the value expression and assignment target into the resulting action.

```
circusifyAssignment input (AssignValue op target value) = circusifyFunctionCalls ((name=) §> rhs)
   where rhs = circusifyAssignmentRhs input op target
        expressionFromValue = circusifyExpression input value
        rhsFromExpression = circusifyAssignmentRhs input op target
        name = fullNameToZ (assignmentName target)
```

We now consider the right-hand side of an assignment, using `circusifyAssignmentRhs`. This is a stateful computation that computes the entire assignment to the right of the assignment operator, and is implemented by first converting the assignment target into a list of expressions representing each recursion level of the assignment target, then applying that list to a lifted form of `assignmentRhs`.

```
circusifyAssignmentRhs :: CircusifyInput
   -> AssignValueOperator
   -> AssignmentTarget FullName
   -> ZExpression
   -> State [ZFunctionCall] ZExpression
```

```
circusifyAssignmentRhs input op target rhs
```
= assignmentRhs (circusifyAssignValueOperator op) rhs <\Rightarrow>
targetToExpressions input target

We now elaborate assignmentRhs.

assignments :: (ZExpression → ZExpression → ZExpression)
→ ZExpression
→ [ZExpression]
→ ZExpression

Simple assignments. When the target list has one element, we are assigning the value to the entire target given in that list. Thus, we can apply the operator function to that target and value to produce the right hand side of the assignment:

assignmentRhs op expr [x] = op x expr

Assignments to structs and arrays. When the target list has more than one element, it is describing an assignment to part of a compound value, such as a member of a struct or element of an array. The head of the list is the member or element, and the rest of the list defines the rest of the target.

We represent the assignment of part of a compound structure by assigning to that structure the current structure value, overridden with a new mapping from the member or element being assigned to its new value. Since the new value may depend on the current value, we invoke the operator as above but provide the result of app on the target, which calculates the function application that returns the value of the target. Similarly, the result of app on the tail of the list yields the value of the compound structure itself, which we then override with the new mapping.

Finally, because the compound structure could be an element of another compound structure, we recurse down the target list. The assignment operator is replaced in the recursive calls with a simple assignment, as we are now assigning a value to the wider structure.

assignmentRhs op expr (x : xs) = assignmentRhs (circusifyAssignValueOperator AOpEquals)
(app xs ⊕ ZSetLiteral [x → op (app (x : xs)) expr])

where app = foldr1 (flip Z.Application)

When the target list is empty, the function is undefined.

Example. Consider the following set of assignments.

variable = value;
variable += value;
variable[5] = value;
variable[5] += value;
variable.member = value;
variable.member += value;

The resulting Circus assignments are below.

variable := value
variable := variable + value
variable := variable ⊕ {5 → value}
variable := variable ⊕ {5 → (variable5) + value}
variable := variable ⊕ {member→ value}
variable := variable ⊕ {member→ (variable.member) + value}

(End of example.)

We now elaborate targetToExpressions.

targetToExpressions :: CircusifyInput
→ AssignmentTarget FullName
→ State [ZFunctionCall] [ZExpression]

Identifier assignments. We simply return a single-value list containing the Z equivalent of the identifier.

targetToExpressions _ (AssignID name) = return [(ZIdExpr ο fullNameToZ) name]
Struct assignments. We recursively consider the rest of the target, and append the struct member to the resulting list.

\[
\text{targetToExpressions input (AssignStruct rec member)} = (\text{ZIdExpr member}::\triangleleft) \text{targetToExpressions input rec}
\]

Array assignments. This is similar to the situation with structs, but we must translate the array subscript expression. As usual, we increment the index by one, to map from ACDC arrays to Z sequences.

\[
\text{targetToExpressions input (AssignArray rec index) = addIndex \triangleleft targetToExpressions input rec}
\]

\[
\text{where addIndex exprs = (\text{flip handleCalls) \circ (\text{uncurry (\text{resultVariableCommand})}])}}
\]

\[
\begin{align*}
\text{addOne} &= (\text{zplus ZNumber 1}) \\
\text{addIndex exprs} &= (\text{uncurry (\text{resultVariableCommand})})
\end{align*}
\]

4.12.11 Function calls

Function calls in ACDC may contain side effects. In order to represent function calls in Circus, we separate them from the expressions in which they are found, and place them in the actions in which those expressions appear. This task is performed by circusifyFunctionCalls, which takes a chain of function calls calculated by circusifyExpression, sequences them before the calling action, and wraps the resulting action in a variable introduction.

\[
\text{circusifyFunctionCalls :: State [ZFunctionCall] Action} \rightarrow \text{Action}
\]

The action given to circusifyFunctionCalls is inside a state monad. To run stateful computations, we use runState, which takes a stateful computation and an initial state, and returns the pair of final value and state (the final list of function calls that must be added). We apply runState with the empty list of function calls, returning the translated action and list of function calls, for which the action must be prepared.

We then use the preparation function to construct, if necessary, a command constructing the resultN variables for each function that does indeed have a return value. If such a command exists, the aforementioned action is placed inside it; otherwise, the action is passed through unmodified.

\[
\begin{align*}
\text{circusifyFunctionCalls} &= \text{uncurry (\text{flip handleCalls})} \circ (\text{runState}) \\
\text{resultVariableCommand} &= \text{foldMap functionAction} \circ \text{zip [1..]}
\end{align*}
\]

We now consider resultVariableCommand.

\[
\text{resultVariableCommand :: [ZFunctionCall]} \rightarrow \text{Action} \rightarrow \text{Action}
\]

We begin by, again, zipping the call chain with the list of call positions. We then perform a concatenative map using toIdentifier, which produces a list containing the resultN identifier for each \(N\)th function call that has a return value. We then create the function on Actions using wrapAction, which is the identity on its action parameter if there was no \(resultN\) identifier, and an encoding of the corresponding var command otherwise.

We first use toSeq to convert the identifier list into a Maybe Seq1. If this sequence is non-empty, we then use makeDecl to convert it into a Z declaration part with the identifiers assigned to declarations of the universe type. We then add Var to produce a pair of VarType and ZDeclPart, lift it to a list of one value, and apply the \(\bullet\) operator. This converts the list into a function that will add the corresponding var declaration to an action.

\[
\begin{align*}
\text{resultVariableCommand} &= \text{wrapAction} \circ \text{concatMap toIdentifier} \circ \text{zip [1..:
\text{Int}..]}} \\
\text{where wrapAction} &= \text{maybe id ((\bullet) \circ \text{return} \circ (\text{Var}_{\rceil}) \circ \text{makeDecl})} \\
\text{makeDecl} &= (\text{\langle singleTypeZDecl' ZPredefinedSet U\rangle}) \\
\text{toIdentifier (k, call)} &= [\text{FunctionCallResultIdentifier} k \mid \text{zfcHasReturn call}]
\end{align*}
\]
We now elaborate \textit{functionAction}.

\[
\text{functionAction} :: (\text{Int}, \text{ZFunctionCall}) \rightarrow \text{Action}
\]

The function \textit{applyList} produces a function application from an action and a list of arguments to that action. When there are no arguments, it is the identity on the given action. We use \textit{applyList} to apply the function call to its list of arguments, as well as an extra argument, if applicable, for the return argument.

\[
\text{functionAction} \ (k, \text{call}) = \text{applyList} \ (\text{AIdentifier} \ (\text{zfcIdentifier} \ \text{call})) \ (\text{zfcArguments} \ \text{call} + + \ returnArg)
\]

Whether the return argument exists depends on whether the function call has a return value: if it does not, the return argument does not exist.

\[
\text{where} \ returnArg = [\text{ZIdExpr} \ (\text{functionCallResultIdentifier} \ k) \ | \ \text{zfcHasReturn} \ \text{call}]
\]

We adopt a convention of naming function call result variables \textit{resultN}, where \textit{N} is the position, starting from one, of the function call in the chain of calls. The variable \textit{resultN} is skipped over if the \textit{N}th function has no result (its return type is void). This convention is codified by \textit{functionCallResultIdentifier}, which is formalised in the appendices.

\textbf{Example.} The result of the first function in a call chain will use the variable \textit{result1}. The third will use \textit{result3}, even if the first, second, or both first and second functions had no return value. \textbf{(End of example.)}

\section*{4.13 Final considerations}

We have now concluded the formalisation of the main aspects of our translation process. Not all of the code that forms the process prototype is elaborated here: elided definitions are, as mentioned previously, available in the appendices, and some utility modules have been left out of the report entirely. Every piece of code, however, is provided in the supplemental files that come with this report.

Although the process as formalised was based initially on that given by Ribeiro \cite{Ribeiro}, and many aspects of the process are highly similar up to the change of target language and meta-language, some areas have been radically changed. This is mainly due to the Haskell environment offering more concise, idiomatic, or high-level equivalents to the approach taken by Ribeiro. This can be seen most notably in the phases that build constructs from the program declarations: we use Haskell’s treatment of functions as first-class data values to isolate the similarities between phases into a pattern of traversals, and implement the differences as small helper functions atop the common framework. This promotes a high level of code-reuse in the process and, thus, reduces its error surface.

In the next chapter, we proceed to evaluate the process just formalised.
5 Evaluation

To have a degree of confidence that the process produces valid models given input satisfying the assumptions, we evaluate the process using several approaches. We outline these in section 5.1, before discussing the results in section 5.2. In section 5.3 we discuss aspects of the overall quality of our work not covered by our formal evaluation strategies.

5.1 Approaches used

We use several approaches to evaluate our translation process, which we list below. These approaches are sound, but not complete; we discuss this issue at the end of this chapter.

**Type checking.** The most basic form of validation we use on the process is the Haskell type checker. As a part of the Haskell compilation process, the code is subjected to a phase of type-checking that fails if any type errors are discovered. Since Haskell has a strong, static type system, a successful type-checking run gives confidence that the program provided is free of a class of errors that may have persisted in other languages or in a mathematical meta-language.

Coupled with a defensive use of types to capture transformations inside the process, this guarantees that our work performs a well-formed series of computations that are provided with, and result in, values of the correct type. For example, we cannot accidentally implement `flatten` as only collapsing the program into a list of external declarations, as the type explicitly captures the naming convention used in the program, and `flatten` is defined as mapping from a program in terms of `LocalNames` to a declaration list over `FullNames`. The type checker would catch this; had we used a less rigorous type system, this would not be the case. The type system alone, however, is not a sufficient guard against all possible errors. Haskell has a value, `undefined`, that inhabits all types; we could have defined `flatten` as `undefined` and only found this error during run-time.

**Compiler warnings.** Another diagnostic we use is the presence of *compiler warnings*, which are emitted by the compiler when valid, but dubious, code is found. For example, the Glasgow Haskell Compiler (GHC) can produce warnings for incomplete pattern matches, which could suggest a definition was not finished. We use the `-Wall` (all warnings) option with GHC to maximise the depth of warnings generated.

**Unit testing.** Unit testing is a common strategy for testing programs. As we have built a prototype program implementing the translation process in Haskell, we can make use of it via the HUnit library. We define test runs that assert that the application of parts of the process to sample data yield the expected results.

We have defined a series of unit tests for aspects of the process, and take the passing of unit tests as a necessary condition for accepting the process. Some of the unit tests are defined using the examples from Ribeiro [8] as a basis: we do this to compare our process to that of Ribeiro and also because the examples given serve as a good indicator of whether the process is correct for the inputs considered in the examples. We did not systematically unit test the program, unit testing coming late in the development of the process, but our general philosophy was to capture both our and Ribeiro’s examples in automated form (to gain confidence that the examples were correct representations of the process’s behaviour), and also create unit tests to exercise known failures in the process so any corrections could be checked for soundness.

**Property checking.** The QuickCheck system [56] tests whether Boolean properties hold over arbitrary members of given data types. It randomly generates valid members of those types, and checks the property for a given number of those random members in an attempt to falsify the property.

Testing with QuickCheck is similar to model-checking. It does not require a finite input domain, but the random and limited nature of input generation means that it is not complete (it could fail to falsify properties that do not hold for inputs not selected). It is, however, sound, given an appropriate set of properties.

We use a small amount of QuickCheck properties in an attempt to falsify statements that must be true of a correct implementation. For example, there is a property that, for each step of our transitive closure calculation, for every two pairs in the original list `(a, b)` and `(b, c), (a, c)` is in the list after the step. Another property states that `rotateFrame` is closed over the range from 1 to the frame count inclusive. These are simple properties, yet their failure suggests fundamental flaws in the basic algorithms that underpin our work.

**Case study analysis.** To demonstrate the usage, feasibility and validity of the translation process, we apply it to a case study in ACDC. This is a translation of the PID case study from Ribeiro [8]. There are advantages to this re-use:

1. The case study is feasible: it has been successfully translated using the Ada counterpart to our translation process;
2. We have an existing Circus model and, ideally, our process will generate an equivalent C implementation model;
3. The case study is an implementation of a Simulink diagram and is thus representative of our process’s context.

Having discussed our approaches to evaluation, we now summarise the results. A short-form test log is given in Appendix C, and the Circus model produced from the case study forms Appendix D.
5.2 Results

Type checking. We have successfully compiled our translation process using the Glasgow Haskell Compiler (GHC), version 7.8.2, with no compiler warnings under -Wall. The successful compilation implies that the type checking was successful.

Automated tests. All of the tests we created to evaluate the process, including QuickCheck properties, pass. As QuickCheck selects random inputs when checking a property, there is no guarantee that those properties are indeed true of all values.

The use of examples from Ribeiro, and the passing of the tests based on those examples, gives a degree of confidence that the aspects of the translation process covered by those examples are at least as sound as their counterparts in Ribeiro’s process.

We did not unit-test the Circus emitter, partly because it was not considered to be part of the main translation process, and partly due to the difficulty of judging the correctness of the \texttt{ACDC} scripts: as we did not follow a formal specification of the presentation of Circus in \texttt{H}TEX, there would be many different yet equally acceptable ways to format the same Circus model. Instead, we opted to use a case study to evaluate the quality of the final output of the process.

An example of a problem we were able to capture, and later confirm the successful removal of, via unit testing was a bug in the function call handling. An early version of the process did not correctly pass information about the function calls already discovered when converting statements and expressions to Circus, with the result that multiple function calls within the same action were assigned the same \texttt{resultN} variable. This was fixed by making the call passing implicit via the state monad.

We also discovered an error in the calculation of variables to read from channels, carried over from Ribeiro’s process. A unit test that checked that the application of this calculation produced the results given by Ribeiro flagged that an incorrect action were assigned the same

Case study. We used our technique to generate the Circus model of the ACDC version of the proportional-integral-derivative controller used as a case study by Ribeiro [8]. We convert the source code from Ada to the equivalent ACDC, but otherwise do not significantly modify the case study; the resulting model should be comparable to that of the Ada original.

Given this source code and a manually created CircusifyInput, and using the ACDC parser and Circus emitter we created, our process produced a Circus model, which we include in the appendices. We did not perform a rigorous analysis of the model’s correctness, but instead appeal to its similarity to the model produced by Ribeiro, as well as the evaluation of the various parts of the process covered by the automated tests and type-checking procedures.

With the models provided in Ribeiro’s report as a guideline, we were able to identify errors in the generation and emission of the Circus models by our translation process that were not discovered by our automated tests. For example, the model of the case study timer was discovered on inspection of the Circus model to be attempting to read from the \texttt{frame} channel, as opposed to writing it. As this meant no process was writing to the channel, this would have caused a deadlock in the model.

5.3 Qualitative evaluation

Time and feasibility concerns mean that our work is not as easy-to-use, as general or as rigorous as it could be. Of particular issue are the simplifying assumptions from section 4.2. Assumptions 1 and 2 make using the process more difficult, as the input source code must be pre-processed into a format the process can accept, and even then the input must be parsed into the abstract syntax. Assumptions 3, 6 and 7 harm the generality of the process, as they add additional restrictions on those of ACDC. Assumptions 4 and 5 reduce the rigour of the process, as we only model ACDC’s numeric operations in a shallow manner that does not capture the semantics of integer overflow and floating-point precision errors.

With these assumptions, however, we were able to achieve success with the case study comparable to that seen in Ribeiro’s work. Aside from the requirements of pre-processing the ACDC input—which itself was partially automatable using a shell script and the topological sorting already found by Ribeiro—, and creating the CircusifyInput—which was a manual process but not difficult—, there were no difficulties or manual interventions when running the process. Qualitatively, the process appears to be a step in the correct direction, but with room for improvement.

5.4 Final considerations

Given the positive results from our evaluation methods, we feel that our work was successful. However, as our evaluation techniques are not complete, we cannot be fully certain.

Ideally, we would like to perform a rigorous proof of correctness of our technique. This would make sense: our work is intended to be used as part of a strategy for automatically proving the correctness of implementations of control-law diagrams. For this strategy to be truly useful in the safety-critical domain, we need more confidence than the evaluation presented here can give. However, this step was beyond the scope of our work, and indeed beyond our skill level.

The unit and QuickCheck tests used are included in the source code for the process: see Appendix B for more details.

Having evaluated our work, we now move on to the report conclusions.
6 Conclusions

We have presented a working prototype for a modelling technique that, for a program written in ACDC and following our assumptions, produces its Circus model. Although the process is mainly defined in terms of a mapping from the ACDC abstract syntax tree to that of Circus, we also created a proof-of-concept pair of parser and emitter for the evaluation of our work.

In the rest of this section, we explore the outcomes of our work in detail. In section 6.1, we discuss the consequences of our use of Haskell in the formalisation of the process. Our translation process is based on work by Ribeiro to formalise the equivalent process for Ada. There are differences between our respective approaches, which we discuss in section 6.2. We then, in section 6.3, consider the context we explored at the start of our work, and how our final results fit into it. Our work, while a contribution to the field of modelling of state-rich reactive programs in C, has room for improvement and extension. In section 6.4, we discuss potential future work.

6.1 Haskell as a language for model generation

Our use of Haskell in developing the translation process was initially restricted to assembling Haskell versions of Z-notation translation phases for type-checking and evaluation purposes. Haskell proved itself to be more than capable, however, of hosting the final implementation.

Haskell provided us with many tools to provide a concise definition of the translation process. The traversal-based approach to many of the pre-processing phases relies on Haskell’s support for the treatment of functions as values, as well as abstractions based on concepts in abstract algebra, such as monoids. The state monad and accumulativemaps are solid frameworks on which we could hang solutions to the problems of dealing with side-effects in expressions and scope-flattening.

The type system and compiler warnings, whose availability was the original reason we used Haskell, identified many errors in the process that would otherwise have been missed. The HUnit and QuickCheck testing systems helped us detect regressions, failings and spurious assumptions in the process. HLint, which suggests style-improving rewrites for Haskell source, was useful for cleaning up the process code.

Our experiences with Haskell yield confidence that its use for automated translation processes such as ours is beneficial. Haskell lends itself to a natural, concise and easily evaluated specification of such processes, and the wealth of tools available for checking, refining, executing and formatting it add to its suitability.

6.2 Differences from the Ada process

As evidence that our work is an individual contribution, we note differences from the previous work by Ribeiro.

Target language. A clear distinction between our work and that of Ribeiro is the target language. Whereas Ribeiro targets a subset of Ada with a pre-defined architecture, we target a subset of C, albeit using the same architecture.

Changes to our process due to the different target language include the lack of distinction between in and out parameters (in ACDC, all parameters are input values, and there is only one output from a function, namely its return value); the lack of a phase for normalising function call arguments (in ACDC, only positional arguments are supported); and the lack of namespaces for variables and functions (which simplifies the renaming of both).

Meta-language. Whereas Ribeiro provided a formal specification of a translation process in the Z notation, our process is an executable prototype in Haskell. We gain an easier evaluation of the process and the ability to apply it directly to real-life code, with the disadvantage of a less rigorous specification that uses a notation foreign to the relevant field of study.

Techniques for evaluation. Our creation of a prototype allows us to use evaluation techniques found in the realm of software engineering, such as unit testing and type checking. With a parser and emitter attached, we can input the case studies into the compiled Haskell program and survey the output. Since we use the same case studies as those in Ribeiro, ported to ACDC, we can use the examples given in that report as test cases for the process, to demonstrate that the process produces equivalent results.

Lack of function call standardisation. Unlike the Ada process, the C modelling process does not need a step for standardising function calls, as C has an exact correspondence in all situations between function formal parameters and call arguments. There is only one way to define a parameter in a function call, which matches the convention used in the Circus models.

Additional phases. We add two phases, GetTypes and GetStructs, which build maps of types and structures used in the final phase. We use these maps to construct shallow Z models of the types of variables where feasible, which is a more detailed treatment than the purely universe-typed approach used by Ribeiro.

We also construct a basic parser and emitter for evaluation purposes. These are not formalised here, but are included in the supplemental files.
Use of generic traversals. Much of the definitions of early phases of our process are given in terms of a set of generic functions for traversing the ACDC syntax. This allows a very compact definition of the phases, and is a marked difference from Ribeiro.

Increased automation. When attached to an ACDC parser and Circus emitter, such as those built for the evaluation of the work, our translation process can be operated on real ACDC programs to generate Circus models. This is a higher degree of automation from that available in the Ada process, which has a partial encoding in Haskell but is primarily a non-executable formalisation in the Z notation.

6.3 Revisiting the context

At the beginning of this report, we posed three questions to motivate our work and its contributions to the concept of proof via refinement. We now revisit those questions:

- Did we extend the process to languages other than Ada? In introducing ACDC, and producing a prototype process for deriving models of implementations in ACDC, we have shown that at least this part of the proof strategy can be extended beyond Ada.

- Did we automate the process? Our translation process is more automated than its ancestor, but some aspects still need work. The need to sort, concatenate and pre-process the ACDC code before use, to construct an input context manually in a format more readable by Haskell than by humans, and to deal with a large amount of simplifying assumptions and issues related to shallow modelling all detract from our ability to plug ACDC programs into the process and effortlessly derive useful models.

- Did we reproduce the previous success of the process? Our process indeed was able to produce a Circus model comparable to that found by Ribeiro. This implies that not only was our work successful in reproducing the results of the prior work in the field, but also that the concept of producing models of implementations via a formal technique is once again shown to be feasible. Again, however, our process would need a more rigorous assessment of correctness before it can be deemed useful in the safety-critical sector.

We conclude that, by our original motivation, we have succeeded in making a contribution to the field of interest, even if our work needs much extension and refinement to become part of the trusted tool-set of formal methods. We can hope that, eventually, techniques for proof by refinement similar to ours will allow widespread and easy access to the right degree of rigour in software engineering, to prevent disasters like the Therac-25 case.

6.4 Future work

Future extensions to our work fall into three categories: extensions to ACDC, removal of the simplifying assumptions of the process, and development of the process itself.

6.4.1 Extensions to ACDC

ACDC is a subset of C that is demonstrated to be useful for applications representative of those in industry. However, it is a conservative subset, restricted to the elements of C that are easily modelled in Circus, and the lack of features such as pointers and bitwise arithmetic limit the general application of ACDC in the field.

Pointers. A major shortcoming of ACDC is that there is no ability to take a reference to a variable. The full power of C pointers is unsound—it brings much potentially unsafe behaviour and prevents modular reasoning—but complete absence is incomplete—it forbids construction of perfectly safe programs. A compromise is the use of a structured, restricted form of pointer usage, for example one based on ownership types [57].

Unions. We do not support unions because of their potential for misuse. However, unions are an efficient way of representing variables whose type is exactly one of a range of types at any given time, and their exclusion thus reduces the applicability of ACDC in resource-constrained settings. Future work may look at implementing unions in ACDC in a way that restricts the safety issues, such as enforcing tagging of unions.

bitwise operators. We do not model bitwise operators, as to do so would involve an elaboration of their semantics in Z. However, bitwise operations are part of C’s expressive power for dealing with low-level abstractions, and omitting them significantly reduces the application of ACDC as a low-level systems and embedded software language. Future work may involve implementing a Z semantics for bitwise operators.
6.4.2 Removal of simplifying assumptions

We make many simplifying assumptions about the input that exist only to ease our technique’s implementation in the allotted time-span. These limit the technique’s usefulness, and many could be lifted. For example, the assumption that the program does not use the short-circuit semantics of logical AND and OR could be lifted by implementing said semantics, for example by considering AND and OR as implicitly containing an if-then-else statement whereby the second term is evaluated if, and only if, the first term does not yield a definitive truth value.

6.4.3 Development of the translation process

Our translation process is partially automated—it accepts as input ACDC source code and emits Circus models in \LaTeX format—and, as we discussed in the evaluation, is sufficient for producing an implementation model of a simple but representative case study. However, the process could be improved. For example, the process could be made more fully automated and suitable for integration into larger automated work-flows. Also, the \LaTeX formatting of the Circus models is basic and does not conform to any parseable standard, the models are shallow embodiments of ACDC semantics, and error detection and correction is crude. Future work could improve these aspects of the process.

Automation. The process requires its C source code to be provided in one file, which requires manual topological sorting, concatenation and pre-processing of the input code. Improving this situation is feasible, and would be a simple next step in fully automating the process.

The calculated Circus model should be given in a format more suitable for further parsing and automated reasoning. The abstract syntax form of the model may be emitted and parsed by other Haskell programs, but is not easily used elsewhere. The \LaTeX encoding used in the evaluation does not follow any formal specification of the presentation of Circus, and is also unsuitable for parsing.

Generalisation. Our process is specific to ACDC, but many of the concepts we use are portable to other languages. Many of them were adapted from, and remain similar to, those used by Ribeiro for Ada.

One approach would be to consider the modelling of programs written in a common intermediate language, such as the LLVM intermediate representation. LLVM IR is targeted by C and other languages, and has a more suitable semantics for modelling in Circus, using static single assignment to solve the problem of tracking changes to variables through side-effects. LLVM has been used in the generation of CSP models of real-time operating system schedulers, to prove refinement of CSP-OZ specifications.

Depth of modelling. Our models do not fully capture the semantics of ACDC. For example, the ACDC type system is not accurately modelled, and concerns such as integer overflow and the maximum and minimum values of variables are ignored with the optimistic assumption that the program cannot ever misbehave with respect to them. Future work may look to capturing a more realistic model of ACDC types.

Circus does not model the explicit passage of time. To be modelled successfully in Circus, an ACDC program using temporal effects must belong to an architecture wherein those effects can be abstracted into a higher level pattern of communicating sequential processes with synchronisation, such as the cyclic executive we consider. This does not generalise well to the realm of all interesting ACDC programs; thus, the most refined modelling process would target a modelling notation supporting timed state-rich reactive systems. Targeting the CircusTime extension of Circus would be a natural extension of our work.

[39]
A Additional Definitions

Here we formalise aspects of the translation process omitted from the main body of the report. This does not include all of the code produced for our work: see the supplemental files for full code listings.

A.1 Syntax

A.1.1 Z encodings of ACDC operators

Logical OR. C logical OR is false (0) if and only if both of its arguments are false (0):

\[
\begin{align*}
\text{cLOr} : & \times \to \{0, 1\} \\
\forall x, y : & \bullet x \text{cLOr} y = 0 \iff x = 0 \land y = 0
\end{align*}
\]

Its representation in the AST is CLoR:

| CLoR

Equality. C equality is true (1) if and only if both of its arguments are equal:

\[
\begin{align*}
\text{cEq} : & \times \to \{0, 1\} \\
\forall x, y : & \bullet x \text{cEq} y = 1 \iff x = y
\end{align*}
\]

This is represented by CEq:

| CEq

Non-equality. C non-equality is false (1) if and only if both of its arguments are not equal:

\[
\begin{align*}
\text{cNeq} : & \times \to \{0, 1\} \\
\forall x, y : & \bullet x \text{cNeq} y = 1 \iff x \neq y
\end{align*}
\]

It is represented by CNeq:

| CNeq

Inequalities. These are defined in broadly the same way as equality and non-equality:

\[
\begin{align*}
\text{cLess} : & \times \to \{0, 1\} \\
\forall x, y : & \bullet x \text{cLess} y = 1 \iff x < y
\end{align*}
\]

\[
\begin{align*}
\text{cLessEq} : & \times \to \{0, 1\} \\
\forall x, y : & \bullet x \text{cLessEq} y = 1 \iff x \leq y
\end{align*}
\]

\[
\begin{align*}
\text{cMore} : & \times \to \{0, 1\} \\
\forall x, y : & \bullet x \text{cMore} y = 1 \iff x > y
\end{align*}
\]

\[
\begin{align*}
\text{cMoreEq} : & \times \to \{0, 1\} \\
\forall x, y : & \bullet x \text{cMoreEq} y = 1 \iff x \geq y
\end{align*}
\]

Their Haskell definitions follow:
We now define infix operators found in the Z mathematical tool-kit: relational overrides, maplets, and arithmetic operations.

```
| CLess
| CLessEq
| CMore
| CMoreEq
```

A.2 Flattening scopes

A.2.1 External declarations

Global-scope declarations. These are handled by `flattenDeclaration`. This takes a scope name, which here is the global (empty) scope name, represented by the constant `global`:

```
flattenExternal env (EDDeclaration declaration)
= second EDDeclaraiion (flattenDeclaration global env declaration)
```

A.2.2 Statements

```
flattenStatement _ env counter (Return expression)
= (counter, Return expression')
where expression' = flattenExpression env expression

flattenStatement _ env counter (ExprStm expression)
= (counter, ExprStm expression')
where expression' = flattenExpression env expression

Switch statements. These may contain an arbitrary number of compound statements.

```
flattenStatement sname env counter (Switch expression cases)
= second (Switch expression') flattenedCases
where expression' = flattenExpression env expression
    flattenedCases = flattenSwitchCases env sname counter cases

(While expression compound) →
    While
        (flattenExpression env expression)
        (flattenCompound sname' env compound)

(DoWhile compound expression) →
    DoWhile
        (flattenCompound sname' env compound)
        (flattenExpression env expression)

(For initialiser invariant post compound) →
    For
        (flattenAssignment (flattenAssignmentTarget env)
            (flattenExpression env) initialiser
        )
        (flattenExpression env invariant)
        (flattenAssignment (flattenAssignmentTarget env)
            (flattenExpression env)
```
post
) (flattenCompound sname' env compound)
_ = error "Bug: This should have been matched earlier."

Struct members. flattenAssignmentTarget env (AssignStruct struct member)
= AssignStruct struct' member
where struct' = flattenAssignmentTarget env struct

Switch cases
A set of switch cases is renamed by flattenSwitchCases:

flattenSwitchCases :: Env
    → ScopeName
    → Integer
    → SwitchCases LocalName
    → (Integer, SwitchCases FullName)

The base case of flattenSwitchCases is the default statement:

flattenSwitchCases env sname counter (Default compound)
= (counter + 1, Default compound')
where compound' = flattenCompound sname' env compound
    sname' = pushScope sname (show counter)

Non-default case statements are inductively defined on SwitchCases.

flattenSwitchCases env sname counter
(Case expression compound cases)
= (counter', Case expression' compound' cases')
where
    expression' = flattenExpression env expression
    compound' = flattenCompound sname' env compound
    (counter', cases')
        = flattenSwitchCases env sname (counter + 1) cases
    sname' = pushScope sname (show counter)

A.2.3 Expressions

flattenExpression _ (ConstantExpr c) = ConstantExpr c
flattenExpression _ (StringExpr s) = StringExpr s
flattenExpression env (Infix op lhs rhs)
    = Infix op lhs' rhs'
where lhs' = flattenExpression env lhs
    rhs' = flattenExpression env rhs
flattenExpression env (Prefix op expression)
    = Prefix op expression'
where expression' = flattenExpression env expression
flattenExpression env (ArraySubscript array index)
    = ArraySubscript array' index'
where array' = flattenExpression env array
    index' = flattenExpression env index
flattenExpression env (CallExpr (FC function arguments))
    = CallExpr (FC function arguments')
where arguments' = map (flattenExpression env) arguments
flattenExpression env (StructMember struct member)
    = StructMember struct' member
where struct' = flattenExpression env struct
A.3 Capturing declarations

\begin{align*}
\text{declCaptureExternal (EDStruct \_)} & = \text{Nothing} \\
\text{declCaptureExternal (EDPrototype \_)} & = \text{Nothing} \\
\text{declCaptureExternal (EDDeclaration dec)} & = \text{declCaptureDeclaration dec} \\
\text{declCaptureExternal (EDFunction fun)} & = \text{declCaptureFunction fun}
\end{align*}

A.4 Abstracting temporal information

A.4.1 Analysis

\begin{align*}
\text{analyseExpression \_ (ConstantExpr \_)} & = [ ] \\
\text{analyseExpression \_ (StringExpr \_)} & = [ ] \\
\text{analyseExpression inTemporal (Infix \_ lhs rhs)} & = \text{analyseExpression inTemporal lhs} + + \\
& \quad \text{analyseExpression inTemporal rhs} \\
\text{analyseExpression inTemporal (Prefix \_ expr)} & = \text{analyseExpression inTemporal expr} \\
\text{analyseExpression inTemporal (ArraySubscript array subscript)} & = \text{analyseExpression inTemporal array} + + \\
& \quad \text{analyseExpression inTemporal subscript} \\
\text{analyseExpression inTemporal (StructMember struct \_)} & = \text{analyseExpression inTemporal struct}
\end{align*}

A.4.2 Removal

\begin{align*}
\text{removeTimeStatement \_ NopStm} & = \text{NopStm} \\
\text{removeTimeStatement \_ r\bullet(Return \_)} & = r
\end{align*}

\begin{align*}
\text{removeTimeStatement names (DoWhile compound expression)} & = \text{DoWhile (removeTimeCompound names compound) expression} \\
\text{removeTimeStatement names (For l m r compound)} & = \text{For l m r (removeTimeCompound names compound)} \\
\text{removeTimeStatement names (IfThenElse cond true false)} & = \text{IfThenElse cond (removeTimeCompound names true)} \\
& \quad (removeTimeCompound names false) \\
\text{removeTimeStatement names (Switch cond cases)} & = \text{Switch cond (removeTimeSwitchCases names cases)} \\
\text{removeTimeStatement names (While expression compound)} & = \text{While expression (removeTimeCompound names compound)}
\end{align*}

\textbf{Switch cases.} The function \text{removeTimeSwitchCases} considers switch cases.

\text{removeTimeSwitchCases :: TRemover (SwitchCases FullName)}

As previous, we are concerned only with the compound statements available in the switch cases.

\begin{align*}
\text{removeTimeSwitchCases names (Default compound)} & = \text{Default (removeTimeCompound names compound)} \\
\text{removeTimeSwitchCases names (Case expression compound rest)} & = \text{Case expression} \\
& \quad (removeTimeCompound names compound) \\
& \quad (removeTimeSwitchCases names rest)
\end{align*}

\text{hasTemporalVariables :: [FullName] \rightarrow Expression FullName \rightarrow Any}

\text{hasTemporalVariables names = Any \circ \neg \circ \text{null} \circ \text{intersect names} \circ \text{analyseExpression True}
A.5 Capturing functions called by the ACDC main functions

\[
\begin{align*}
    & r\text{Expression (}\text{ConstantExpr}\text{)} = [] \\
    & r\text{Expression (}\text{IdExpr}\text{)} = [] \\
    & r\text{Expression (}\text{StringExpr}\text{)} = [] \\
    & r\text{Expression }\text{fname (}\text{ArraySubscript array index}\text{)} = r\text{Expression }\text{fname array} ++ r\text{Expression }\text{fname index} \\
    & r\text{Expression }\text{fname (}\text{Infix }\text{lhs rhs}\text{)} = r\text{Expression }\text{fname lhs} + + r\text{Expression }\text{fname rhs} \\
    & r\text{Expression }\text{fname (}\text{Prefix expression}\text{)} = r\text{Expression }\text{fname expression} \\
    & r\text{Expression }\text{fname (}\text{StructMember struct}\text{)} = r\text{Expression }\text{fname struct}
\end{align*}
\]

A.6 Capturing variable usage

\[
\begin{align*}
    & v\text{Expression (}\text{ConstantExpr}\text{)} = [] \\
    & v\text{Expression (}\text{StringExpr}\text{)} = [] \\
    & v\text{Expression (}\text{ArraySubscript array index}\text{)} = v\text{Expression array} ++ v\text{Expression index} \\
    & v\text{Expression c (}\text{CallExpr FC arguments}\text{)} \mid \text{isDelayExpression c} = [] \\
    & \quad \text{otherwise} = \text{concatMap vExpression arguments} \\
    & v\text{Expression (}\text{Infix }\text{lhs rhs}\text{)} = v\text{Expression lhs} + + v\text{Expression rhs} \\
    & v\text{Expression (}\text{Prefix expression}\text{)} = v\text{Expression expression} \\
    & v\text{Expression (}\text{StructMember struct}\text{)} = v\text{Expression struct}
\end{align*}
\]

A.7 Building the Circus model

\[
\begin{align*}
    & \text{declOf} :: \text{CircusifyInput} \\
    & \quad \rightarrow \text{Identifier} \\
    & \quad \rightarrow \text{Maybe DeclProfile} \\
    & \text{declOf input name} = \text{lookupDecl name }((\text{declProfiles }\circ \text{decl}) \text{ input}) \\
    & \text{where} \\
    & \quad \text{lookupDecl []} = \text{Nothing} \\
    & \quad \text{lookupDecl identifier (}x : xs\text{)} \\
    & \quad \quad \text{dpId }x \equiv \text{identifier} = \text{Just }x \\
    & \quad \quad \text{otherwise} = \text{lookupDecl name }xs \\
\end{align*}
\]

\[
\begin{align*}
    & \text{externalVariables} :: \text{CircusifyInput} \\
    & \quad \rightarrow [\text{FullName}] \\
    & \text{externalVariables} = \text{uncurry union }\circ (\text{externalInputs }\&\& \text{externalOutputs})
\end{align*}
\]

\[
\begin{align*}
    & \text{typeof} :: \text{CircusifyInput} \\
    & \quad \rightarrow \text{FullName} \\
    & \quad \rightarrow \text{Maybe Type} \\
    & \text{typeof input name} = \text{lookup name }\langle \text{types input} \rangle
\end{align*}
\]

\[
\begin{align*}
    & \text{programConstants} :: \text{CircusifyInput} \\
    & \quad \rightarrow [\text{DeclProfile}] \\
    & \text{programConstants} = \text{constantProfiles }\circ \text{decl}
\end{align*}
\]
programConstantIdentifiers :: CircuifyInput → [Identifier]

programConstantIdentifiers = map dpId ◦ programConstants

programConstantFullNames :: CircuifyInput → [FullName]

programConstantFullNames = map (globalName ◦ dpId) ◦ programConstants

A.7.1 Structs

structFreeTypeName :: Identifier

→ ZIdentifier

structFreeTypeName = ("ST"++)

A.7.2 Channels

channelIdentifier :: ZIdentifier → ZIdentifier

channelIdentifier = (++"SH")

A.7.3 General processes

On all cases other than that considered in the report body, generalProcess raises the following error.

generalProcess _ _ _ = error "Constant or nonexistent function taken as main procedure."

circusifyProcessAction _ _ = error "Unexpected declaration or lack thereof."

A.7.4 The timer process

timerAction :: CircuifyInput → [Statement FullName] → Action

If the timer statement list begins with a delay statement, we introduce a synchronising write to the frame counter, followed by a synchronisation on the end_cycle channel. Then, we sequentially compose the remaining statements in the timer as translated by circusifyStatement, via foldMap.

timerAction input ((isDelayStatement → True) : xs) = ("frame" ◦ ZIdExpr framec) → (endIf ◦ xs')

where framec = fullNameToZ (frameCounter input)

endIf = ifActionList [pnot endFrame ↔ skip, endFrame ↔ endCycle]

endFrame = ZIdExpr framec 'predEqual' frameCountZ

frameCountZ = (ZNumber ◦ frameCountInt ◦ frameCount) input

xs' = foldMap (circusifyStatement input) xs

If the first statement is not a delay, we instead consider each statement with circusifyTimerStatement.

timerAction input xs = foldMap (circusifyTimerStatement input) xs

We now elaborate circusifyTimerStatement. This differs from circusifyStatement in that, if it finds an infinite loop, it considers the body as a timer action, and implements frame counter initialisation and incrementing if no explicit frame counter already exists in the case study.

circusifyTimerStatement :: CircuifyInput → Statement FullName → Action

circusifyTimerStatement input (While (ConstantExpr (CInt 1)) (Compound _ statements)) |

haveCounter input = rec body

| otherwise = initCounter ◦ rec (body ◦ tickCounter)
where  

\[ \text{where} \quad \text{body} = \text{timerAction input statements} \]

\[ \text{rec} = \mu X. (\circ \langle X \rangle) \]

\[ \text{initCounter} = \text{counter} = ZNumber 1 \]

\[ \text{tickCounter} = \text{counter} = \text{rotateFrameZ} (ZNumber \text{numFrames}) \]

\[ (ZNumber 1) \]

\[ (ZIdExpr \text{counter}) \]

\[ \text{counter} = \text{fullNameToZ} (\text{frameCounter input}) \]

\[ \text{numFrames} = (\text{frameCountInt } \circ \text{frameCount}) \text{ input} \]

\[ \text{haveCounter} = \text{any isCounter } \circ (\text{reachableV Usage } \& \& \text{rgraph } \&\& \text{mainFunctions } \&\& \text{vusage}) \]

\[ \text{isCounter} = (\exists \text{frameCounter input}) \circ \text{variableName} \]

In the general case, \text{circusifyTimerStatement} falls back to \text{circusifyStatement}.

\[
\text{circusifyTimerStatement input statement} = \text{circusifyStatement input statement}
\]

**A.7.5 Step functions**

\[
\text{circusifyStep input f} = \text{circusifyStepShared input f } \circ \text{action}
\]

We translate the step function differently depending on whether it contains a frame scheduler. We assume that it does if, and only if, the first statement in the step function is an swit statement. This query is represented by \text{checkScheduler}, which returns an Either context. This context is similar to Maybe, but holds one of two values (a left value, or a right value). If there is a frame scheduler in the form of a swit statement, we return that single statement as a right value; otherwise, we pass through the statement list as the left value.

\[
\text{where checkScheduler (Switch \{ \} : \_)} = \text{Right s} \\
\text{checkScheduler s} = \text{Left s}
\]

The definition of action when the step function contains a frame scheduler is simply the result of \text{circusifyStatement} on the scheduler. When the step function does not contain a frame scheduler, we first apply \text{addFrameScheduler} to introduce one, then perform the translation.

This choice is represented by \text{either}, which applies its first parameter if given a left value and its second if given a right value. We thus select \text{addFrameScheduler} if the statement body is a left value (it has no frame scheduler), and the identity function if it is a right value (it has a frame scheduler).

\[
\text{action} = \text{circusifyStatement input } \circ \text{either (addFrameScheduler counter count) id } \circ \text{checkScheduler}
\]

\[
\text{counter} = \text{frameCounter input}
\]

\[
\text{count} = \text{frameCountInt } (\text{frameCount input})
\]

**Adding a frame scheduler.** We convert a delay-based step function to a frame scheduler using \text{addFrameScheduler}. This takes the name of the frame counter, the number of frames in the system, and the step function’s statement list. It produces a single statement defining the frame scheduler.

\[
\text{addFrameScheduler :: FullName } \rightarrow \text{Int } \rightarrow \text{[Statement FullName]} \rightarrow \text{Statement FullName}
\]

We split the statement list into a list of compound statements, each representing a frame, using \text{delayGroup}. We convert these statements into a series of pairs of frame number and switch case, using \text{casesFromFrames}. We discard the frame numbers with \text{snd}, then insert the cases into the scheduler swit statement with \text{makeSwitch}.

\[
\text{addFrameScheduler counter count} = \text{makeSwitch } \circ \text{snd } \circ \text{toCases } \circ \text{delayGroup}
\]

In \text{delayGroup}, we split the list of statements whenever we see a delay statement, as we assume a delay statement represents the end of a frame. The auxiliary function \text{isDelayStatement} is given to the function \text{splitWhen}, a part of the \text{split} Haskell library. This returns a list of lists of statements, which we convert to compound statements by mapping \text{Compound} with an empty declaration list.

\[
\text{where delayGroup} = \text{map (Compound []) } \circ \text{splitWhen isDelayStatement}
\]

We then convert the frame compound statements to a series of switch cases. We do this using a right fold, whose base case is a pair of the frame count and an empty default case. Thus, we consider each frame from right to left, building the cases by attaching the previously constructed cases as the recursive term of the new case.

\[
\text{toCases} = \text{foldr toCase (count, Default (Compound [] []))}
\]
The folding step takes the next block to consider, as well as that block’s frame number and the cases corresponding to the blocks that succeed the one being considered. We decrement the frame by one, such that the next folding step uses the correct frame number for the next leftmost block. We construct the next switch case, guarded on the frame counter being equal to the frame we currently consider, by attaching the block as the case’s body and the previously seen cases as its recursive term.

\[
toCase \text{ block } (frame, cases) = (frame - 1, \text{Case} \ (\text{ConstantExpr} \ (\text{CInt} \ frame)) \ \text{block cases})
\]

Finally, makeSwitch completes the switch statement, using the frame counter as the expression on which the switch statement is focused.

\[
\text{makeSwitch} = \text{Switch} \ (\text{IdExpr} \ \text{counter})
\]

We now elaborate circusifyStepShared.

\[
circusifyStepShared :: \text{CircusifyInput} \rightarrow \text{Identifier} \rightarrow \text{Action} \rightarrow \text{Action}
\]

Since we assume the frame scheduler has been translated to the guarded command language as an if construct with one guarded action per frame, we define circusifyStepShared by traversing over those actions.

\[
circusifyStepShared \ \text{input function} \ (\text{ACommand} \ (\text{CIf} \ \text{gactions})) = \ "frame" \ . ? \ \text{counter} \ \rightarrow \ \text{ifAction} \ \text{gactions}'
where \ \text{counter} = (\text{fullNameToZ} \ . \ \text{frameCounter}) \ \text{input}
\]

\[
gactions' = \text{fmap} \ \text{cssGAction} \ \text{gactions}
\]

\[
circusifyStepShared _ _ _ = \text{error} \ "\text{Unexpected step function body."}
\]

\[
extIn = \text{externalInputs} \ \text{input}
extOut = \text{externalOutputs} \ \text{input}
vmap = \text{circusifyVMap} \ \text{input}
count = \text{frameCount} \ \text{input}
\]

\[
a \bullet \text{GAction} \ (\text{PInfix NonMembership} \ (\text{ZIdExpr} \ _)) \ (\text{ACSPAction} \ \text{Skip})
\]

\[
\rightarrow a
\]

\[
_ \rightarrow \text{error} \ "\text{Unexpected switch case in step function } " + + \ \text{function}
\]

\[
\text{coalesce} :: [(\text{FullName}; \text{Frame}; \text{Identifier})] \rightarrow [(\text{FullName}; [(\text{Frame}; \text{Identifier})])]\n\]

\[
\text{coalesce entries} = \text{map} \ \text{getWithName} \ (\text{varNames} \ \text{entries})
where \ \text{varNames} = \text{nub} \ \text{map} \ \text{varName}
\]

\[
\text{getVarName} \ (a; _; b) = a
\]

\[
\text{getWithName} = \text{id} \ \&\& \ \text{map} \ \text{removeName} \ \text{entries} \ \text{hasName}
\]

\[
\text{hasName} \ b = (b \equiv) \ \text{varName}
\]

\[
\text{removeName} \ (_; a; b) = (a; b)
\]

### A.7.6 Expressions

**Constants.** We extract the constant from its \text{ConstantExpr}, and apply the function \text{constantToZ} to convert it to the appropriate Z expression. We then apply the function \text{pure}, which is shorthand for returning an expression from \text{circusifyExpression} with no function calls, and is defined below.

\[
(\text{ConstantExpr} \ \text{constant}) \rightarrow (\text{return} \ \circ \ \text{constantToZ} \ \text{constant})
\]

**Arrays.** Arrays are mapped in our translation to sequences, which are functions from indices to array elements. An array access expression is thus an application of the index to the array function. However, since Z sequences begin indexing at one and ACDC arrays start at zero, we must increment the subscript by one.

\[
(\text{ArraySubscript} \ \text{array} \ \text{subscript}) \rightarrow \text{addSubscript} \ link \ \text{translateArray}
where \ \text{translateArray} = \text{circusifyExpression} \ \text{input array}
\]

\[
\text{addSubscript} \ \text{zArray} = (\text{ZApplication} \ \text{zArray} \ \circ \ \text{addOne})
\]

\[
\text{addOne} = (\text{zplus} ' \text{ZNumber} \ 1)
\]
Structs. These are handled in a similar way to arrays, except the argument to the function is a free type element representing the respective struct member.

\[(\text{StructMember struct member}) \rightarrow (\text{ZApplication} \text{ZIdExpr member}) \langle \$ \rangle \text{circusifyExpression input struct}\]

Identifier expressions. We simply convert the identifier to a Z identifier, wrap it in the type constructor \text{ZIdExpr} to promote it to an expression, and return it as a function-call-less expression.

\[(\text{IdExpr identifier}) \rightarrow (\text{return} \circ \text{ZIdExpr} \circ \text{fullNameToZ}) \text{identifier}\]

Infix expressions. This is a straightforward procedure of recursively considering the left and right expressions, mapping the ACDC operation to its Z definition, and returning a \text{ZInfixExpression}:

\[(\text{Infix op lhs rhs}) \rightarrow \text{ZInfixExpression zop} \langle \$ \rangle \text{makeExpr lhs} \langle \$ \rangle \text{makeExpr rhs} \]

\[\text{where makeExpr} = \text{circusifyExpression input} \]
\[\text{zop} = \text{circusifyInfixOperator op}\]

Prefix expressions. We consider these in a similar fashion to infix expressions, but need only consider one operand. We acquire the expression, function calls and function call count from recursively considering the operand with \text{circusifyExpression}, convert the expression itself to a \text{ZPrefixExpression} with the Z-equivalent prefix operator found using \text{circusifyPrefixOperator}, and return the triple of the new expression alongside the function calls and call count from the operand:

\[(\text{Prefix op operand}) \rightarrow \text{ZPrefixExpression zop} \langle \$ \rangle \text{operandExpr} \]

\[\text{where operandExpr} = \text{circusifyExpression input operand} \]
\[\text{zop} = \text{circusifyPrefixOperator op}\]

String literals. We interpret these as arrays of characters, and translate them into literal sequences of numbers.

\[(\text{StringExpr str}) \rightarrow (\text{return} \circ \text{ZSequenceLiteral} \circ \text{map} (\text{ZNumber} \circ \text{fromEnum})) \text{str}\]

Constants are converted to Z expressions by \text{constantToZ}.

\[\text{constantToZ} :: \text{Constant} \rightarrow \text{ZExpression}\]

For integers, we convert from the ACDC type for integer literals, \text{CInt}, to the Z type for integer literals, \text{ZNumber}.

\[\text{constantToZ} (\text{CInt integer}) = \text{ZNumber integer}\]

Floating point numbers and enumeration constants are similar, substituting \text{CFloat} and \text{ZRealNumber}, and \text{CEnum} and \text{ZIdExpr} respectively.

\[\text{constantToZ} (\text{CFloat float}) = \text{ZRealNumber float}\]
\[\text{constantToZ} (\text{CEnum enum}) = \text{ZIdExpr enum}\]

For character literals, we use \text{fromEnum}, which converts from a Haskell character to its integer equivalent, and then convert this integer to Z using \text{ZNumber}.

\[\text{constantToZ} (\text{CChar char}) = \text{ZNumber} (\text{fromEnum char})\]

Finally, \text{fullNameToZ} maps \text{FullNames} to Z identifiers suitable for the Circus model. It is implemented in terms of the show function for \text{FullNames}, and is elaborated in the appendices.

\[\text{fullNameToZ} :: \text{FullName} \rightarrow \text{ZIdentifier}\]

\[\text{fullNameToZ} = \text{show}\]
APPENDIX A. ADDITIONAL DEFINITIONS

A.7.7 Statements

Null statements and value-less return statements do nothing, and thus translate to Skip.

\[
\text{circusifyStatement } \_ (\text{NopStm}) = \text{skip}
\]
\[
\text{circusifyStatement } \_ (\text{Return Nothing}) = \text{skip}
\]

Valued return statements. We translate the return value using \text{circusifyExpression}, then lift it into an assignment guarded command to assign its value to \_result. We then sequence the function calls for the return action.

\[
\text{circusifyStatement input} (\text{Return (Just expr)}) = \text{circusifyFunctionCalls action}
\]
\[
\text{where action} = (\_result\_): = <\!> \text{circusifyExpression input expr}
\]

Expression statements. Since these have no visible effect besides function calls contained within, we ignore the Circus expression computed for the expression but sequence the calls:

\[
\text{circusifyStatement input} (\text{ExprStm expr}) = \text{circusifyFunctionCalls action}
\]
\[
\text{where action} = \text{skip} <\!> \text{circusifyExpression input expr}
\]

Assignments. We provide the input context and the assignment to \text{circusifyAssignment}, which translate the assignment.

\[
\text{circusifyStatement input} (\text{Assignment a}) = \text{circusifyAssignment input a}
\]

Compound statements are similar, substituting \text{CompStm} for \text{Assignment} and \text{circusifyCompound} for \text{circusifyAssignment}.

\[
\text{circusifyStatement input} (\text{CompStm c}) = \text{circusifyCompound input c}
\]

If statements. These are translated using \text{circusifyIfThenElse}.

\[
\text{circusifyStatement input} (\text{IfThenElse expr ifC elseC}) = \text{circusifyIfThenElse input expr (comp ifC) (comp elseC)}
\]
\[
\text{where comp} = \text{circusifyCompound input}
\]

Switch statements. These translate easily into the guarded command equivalent: the translation is, however, left to the function \text{circusifySwitchCases}.

\[
\text{circusifyStatement input} (\text{Switch expr cases}) = \text{circusifyFunctionCalls (makeSwitch =< circusifyExpression input expr)}
\]
\[
\text{where makeSwitch exprZ} = \text{ifAction <\!> circusifySwitchCases input exprZ cases []}
\]

Infinite while loops. We convert these into simple Circus recursion. We introduce \(\mu x y\) to denote a Circus fixed point: it is shorthand for \(\text{ACSPAction (CFixedPoint x y)}\). Recall that, in actions, string literals denote named actions.

\[
\text{circusifyStatement input} (\text{While (ConstantExpr (CInt 1)) body}) = \mu \"x\" (\text{circusifyCompound input body} \diamond \"x\")
\]

General while loops. The general while loop terminates when the loop condition fails to hold.

\[
\text{circusifyStatement input} (\text{While expr body}) = \mu \"x\" (\text{circusifyIfThenElse input expr (circusifyCompound input body} \diamond \"x\") \text{skip})
\]

Do-while statements. A do-while statement has the semantics of a while statement, except that the compound statement is executed at least once before the condition is checked. We elaborate a do-while statement by transforming it into a while statement, but prepending it with a duplicate of its body.

\[
\text{circusifyStatement input} (\text{DoWhile body expression}) = \text{circusifyCompound input body} \diamond \text{circusifyStatement input} (\text{While expression body})
\]
For loops. A for loop is syntactic sugar over a particular form of while loop. We translate for loops into their equivalent while loops, then elaborate those. We move the initialising assignment outside of the loop, move the updating assignment to the end of the loop body, then change the loop to a while loop with the same condition.

\[
\text{circusifyStatement input (For initialiser condition update body)}
\]
\[
= \text{circusifyAssignment input initialiser } \circ \text{circusifyStatement input (While condition body')}
\]
\[
\text{where body' = body } \{ \text{compoundStatements = statements'} \}
\]
\[
\text{statements' = statements }++ \text{[Assignment update]}
\]
\[
\text{statements} = \text{compoundStatements body}
\]

A.7.8 Assignments

\[
\text{assignmentRhs } _-[] = \text{error "Bug: empty assignment target list."}
\]

A.7.9 Function calls

\[
\text{functionCallResultIdentifier :: Integral a}
\]
\[
\Rightarrow a
\]
\[
\rightarrow \text{ZIdentifier}
\]
\[
\text{functionCallResultIdentifier} = ("\text{result"}++) \circ \text{show} \circ \text{toInteger}
\]

A.8 Traversal functions

External declarations. The traversal traverseExternalDFPNR traverses external declarations using traversals for declarations, functions, prototypes, enum specifiers, and struct specifiers.

\[
\text{traverseExternalDFPNR :: (Declaration a } \rightarrow m)
\]
\[
\rightarrow (\text{Function } a \rightarrow m)
\]
\[
\rightarrow (\text{Signature } a \rightarrow m)
\]
\[
\rightarrow (\text{EnumSpecifier } a \rightarrow m)
\]
\[
\rightarrow (\text{StructSpecifier } a \rightarrow m)
\]
\[
\rightarrow \text{External } a \rightarrow m
\]
\[
\text{traverseExternalDFPNR d } _-_-_-_-_- (\text{EDDeclaration } x) = d x
\]
\[
\text{traverseExternalDFPNR } _-_-_-_-_- (\text{EDFunction } x) = f x
\]
\[
\text{traverseExternalDFPNR } _-_-_-_-_- (\text{EDPrototype } x) = p x
\]
\[
\text{traverseExternalDFPNR } _-_-_-_-_- (\text{EDEnum } x) = n x
\]
\[
\text{traverseExternalDFPNR } _-_-_-_-_- (\text{EDStruct } x) = r x
\]

We also provide traverseExternalListDFPNR, which operates on a list of external declarations.

\[
\text{traverseExternalListDFPNR :: Monoid m}
\]
\[
\Rightarrow (\text{Declaration a } \rightarrow m)
\]
\[
\rightarrow (\text{Function } a \rightarrow m)
\]
\[
\rightarrow (\text{Signature } a \rightarrow m)
\]
\[
\rightarrow (\text{EnumSpecifier } a \rightarrow m)
\]
\[
\rightarrow (\text{StructSpecifier } a \rightarrow m)
\]
\[
\rightarrow [\text{External } a] \rightarrow m
\]
\[
\text{traverseExternalListDFPNR dfpnr} = \text{foldMap} (\text{traverseExternalDFPNR dfpnr})
\]

Decls. traverseDeclF is a generic function for traversing the functions of a Decl. It takes a traversal over functions, qualified by the function name, and returns a traversal over Decls.

\[
\text{traverseDeclF :: Monoid m}
\]
\[
\Rightarrow (\text{Identifier } \rightarrow \text{Function FullName } \rightarrow m)
\]
\[
\rightarrow \text{Decl a Mixed}
\]
\[
\rightarrow m
\]
\[
\text{traverseDeclF} f = \text{foldMap} (\text{traverseDeclProfileF f}) \circ \text{declProfiles}
\]
Declaration profiles.  `traverseDeclProfileDF` is a generic function for performing a traversal of a `DeclProfile`. It takes functions for performing traversal on constant declarations and function declarations, both qualified by the name of the declaration, and delegates to them where necessary.

\[
\begin{align*}
\text{traverseDeclProfileDF} &: (\text{Identifier} \to \text{Declaration FullName} \to m) \\
& \quad \to (\text{Identifier} \to \text{Function FullName} \to m) \\
& \quad \to \text{DeclProfile} \\
& \quad \to m \\
\text{traverseDeclProfileDF} \ d = \ (\text{DPConstant i declaration}) = \ d \ i \ \text{declaration} \\
\text{traverseDeclProfileDF} \ f = \ (\text{DPFunction i function}) = \ f \ i \ \text{function}
\end{align*}
\]

If we ignore constants and only traverse the functions, the resulting traversal is `traverseDeclProfileF`.

\[
\begin{align*}
\text{traverseDeclProfileF} &: \text{Monoid m} \\
& \Rightarrow (\text{Identifier} \to \text{Function FullName} \to m) \\
& \Rightarrow \text{DeclProfile} \\
& \Rightarrow m \\
\text{traverseDeclProfileF} = \text{traverseDeclProfileDF} \ (\text{const (const mempty)})
\end{align*}
\]

Functions.  `traverseFunctionPC` takes traversals for a function’s signature (the P standing for ‘prototype’) and body, and returns a traversal of the entire function.

\[
\begin{align*}
\text{traverseFunctionPC} &: \text{Monoid m} \\
& \Rightarrow (\text{Signature a} \to m) \\
& \Rightarrow (\text{Compound a} \to m) \\
& \Rightarrow \text{Function a} \\
& \Rightarrow m \\
\text{traverseFunctionPC} \ p \ c = \ \text{uncurry} \ (\odot) \circ (p \circ \text{functionSignature} \ & \& c \circ \text{functionBody})
\end{align*}
\]

Statements.  `traverseStatementTCEW` is a generic function for performing a traversal of statements in an AST, creating a monoid (for example, a list). It accepts four functions which perform further traversal on assignment targets, expressions, compound statements and switch cases, and delegates to them where necessary.

\[
\begin{align*}
\text{traverseStatementTCEW} &: \text{Monoid m} \\
& \Rightarrow (\text{AssignmentTarget a} \to m) \\
& \Rightarrow (\text{Compound a} \to m) \\
& \Rightarrow (\text{Expression a} \to m) \\
& \Rightarrow (\text{SwitchCases a} \to m) \\
& \Rightarrow \text{Statement a} \\
& \Rightarrow m \\
\text{traverseStatementTCEW} \ t \ c \ e \ w \ \text{statement} = \ \text{case} \ \text{statement} \ of \\
& \text{NopStm} \rightarrow \text{mempty} \\
& (\text{CompStm compound}) \rightarrow c \ \text{compound} \\
& (\text{ExprStm expression}) \rightarrow e \ \text{expression} \\
& (\text{Return Nothing}) \rightarrow \text{mempty} \\
& (\text{Return (Just expression)}) \rightarrow e \ \text{expression} \\
& (\text{Assignment asst}) \rightarrow ta \ \text{asst} \\
& (\text{DoWhile body cond}) \rightarrow e \ \text{cond} \odot c \ \text{body} \\
& (\text{For l m r body}) \rightarrow m\text{concat} \ [ta\ l \\
& , e\ m \\
& , ta\ r \\
& , c\ \text{body}] \\
& \text{(IfThenElse cond true false)} \rightarrow e \ \text{cond} \odot c\ \text{true} \odot c\ \text{false} \\
& (\text{Switch subject cases}) \rightarrow e \ \text{subject} \odot w\ \text{cases} \\
& (\text{While cond body}) \rightarrow e \ \text{cond} \odot c\ \text{body} \\
\text{where} \ ta = \text{traverseAssignmentTE} \ t \ e
\end{align*}
\]
Assignment. \(\text{traverseAssignmentTE}\) traverses an assignment, given functions for traversing assignment targets and expressions.

\[
\text{traverseAssignmentTE} :: \text{Monoid} \; m \\
\Rightarrow (\text{AssignmentTarget} \; a \to m) \\
\Rightarrow (\text{Expression} \; a \to m) \\
\Rightarrow \text{Assignment} \; a \\
\Rightarrow m
\]

For self-assignments, we simply traverse the assignment target and return the resulting monoid.

\[
\text{traverseAssignmentTE} \; t \_ (\text{AssignSelf} \; \text{lhs}) = t \; \text{lhs}
\]

For value-assignments, we traverse the assignment target and the value expression, and return the concatenation of both.

\[
\text{traverseAssignmentTE} \; t \; e (\text{AssignValue} \; \text{lhs} \; \text{rhs}) = t \; \text{lhs} \diamond e \; \text{rhs}
\]

Assignment targets. \(\text{traverseTargetVE}\) traverses an assignment, given functions for handling identifiers and expressions.

\[
\text{traverseTargetVE} :: \text{Monoid} \; m \\
\Rightarrow (a \to m) \\
\Rightarrow (\text{Expression} \; a \to m) \\
\Rightarrow \text{AssignmentTarget} \; a \\
\Rightarrow m
\]

\[
\text{traverseTargetVE} \; v \; e = tf \\
\text{where}
\]

For identifier assignments, we use the identifier function provided.

\[
tf (\text{AssignID} \; \text{identifier}) = v \; \text{identifier}
\]

For array targets, we traverse the subscript expression with the expression function. As array targets are recursive, we then also traverse the recursive target.

\[
tf (\text{AssignArray} \; \text{array} \; \text{subscript}) = e \; \text{subscript} \diamond tf \; \text{array}
\]

Similarly, for struct targets, we recursively analyse the struct itself.

\[
tf (\text{AssignStruct} \; \text{struct}) = tf \; \text{struct}
\]

Switch cases. \(\text{traverseSwitchCasesCE}\) traverses a switch case set in terms of an expression function and a compound statement function.

\[
\text{traverseSwitchCasesCE} :: \text{Monoid} \; m \\
\Rightarrow (\text{Compound} \; a \to m) \\
\Rightarrow (\text{Expression} \; a \to m) \\
\Rightarrow \text{SwitchCases} \; a \\
\Rightarrow m
\]

\[
\text{traverseSwitchCasesCE} \; c \; e = tw \\
\text{where}
\]

For default cases, we simply return the traversal of the case body.

\[
tw (\text{Default} \; \text{compound}) = c \; \text{compound}
\]

For a normal case, we traverse the current case’s activation expression and body, as well as the cases after it. The results from each traversal are concatenated.

\[
tw (\text{Case} \; \text{expr} \; \text{compound} \; \text{rest}) = e \; \text{expr} \diamond c \; \text{compound} \diamond tw \; \text{rest}
\]

Combining \(\text{traverseStatementTCEW}\) and \(\text{traverseSwitchCasesCE}\) on the same functions for expressions and compounds gives \(\text{traverseStatementTCE}\).

\[
\text{traverseStatementTCE} :: \text{Monoid} \; m \\
\Rightarrow (\text{AssignmentTarget} \; a \to m) \\
\Rightarrow (\text{Compound} \; a \to m) \\
\Rightarrow (\text{Expression} \; a \to m) \\
\Rightarrow \text{Statement} \; a \\
\Rightarrow m
\]

\[
\text{traverseStatementTCE} \; t \; c \; e \\
= \text{traverseStatementTCEW} \; t \; c \; e \; (\text{traverseSwitchCasesCE} \; c \; e)
\]
**Compound statements.** `traverseCompoundDS` allows one to implement a traversal of compound statements given a function working on declarations and one working on statements.

\[
\text{traverseCompoundDS} :: \text{Monoid}\ m \\
\Rightarrow (\text{Declaration} \: a \rightarrow m) \\
\Rightarrow (\text{Statement} \: a \rightarrow m) \\
\Rightarrow \text{Compound} \: a \\
\Rightarrow m
\]

\[
\text{traverseCompoundDS} \: d \: s \: (\text{Compound} \: \text{declarations} \: \text{statements}) \\
= \text{foldMap} \: d \: \text{declarations} \downarrow \text{foldMap} \: s \: \text{statements}
\]

If we join `traverseStatementTCE` and `traverseCompoundDS`, we arrive at a function that can traverse compounds given rules for traversing assignment targets, declarations and expressions. We call this traversal `traverseCompoundTDE`.

\[
\text{traverseCompoundTDE} :: \text{Monoid}\ m \\
\Rightarrow (\text{AssignmentTarget} \: a \rightarrow m) \\
\Rightarrow (\text{Declaration} \: a \rightarrow m) \\
\Rightarrow (\text{Expression} \: a \rightarrow m) \\
\Rightarrow \text{Compound} \: a \\
\Rightarrow m
\]

\[
\text{traverseCompoundTDE} \: t \: d \: e = c \\
\text{where} \ c = \text{traverseCompoundDS} \: d \: (\text{traverseStatementTCE} \: t \: c \: e)
\]

If we join `traverseCompoundTDE` with `traverseAssignmentTarget`, using the same function for expressions, we get a function that takes assigned identifier, declaration and expression traversals and returns a traversal for compound statements. We name this `traverseCompoundVDE` and define it below.

\[
\text{traverseCompoundVDE} :: \text{Monoid}\ m \\
\Rightarrow (a \rightarrow m) \\
\Rightarrow (\text{Declaration} \: a \rightarrow m) \\
\Rightarrow (\text{Expression} \: a \rightarrow m) \\
\Rightarrow \text{Compound} \: a \\
\Rightarrow m
\]

\[
\text{traverseCompoundVDE} \: v \: d \: e = \text{traverseCompoundTDE} \: t \: d \: e \\
\text{where} \ t = \text{traverseTargetVDE} \: v \: e
\]

Finally, we can join `traverseCompoundVDE` with `traverseDeclarationE`, we get `traverseCompoundVE`.

\[
\text{traverseCompoundVE} :: \text{Monoid}\ m \\
\Rightarrow (a \rightarrow m) \\
\Rightarrow (\text{Expression} \: a \rightarrow m) \\
\Rightarrow \text{Compound} \: a \\
\Rightarrow m
\]

\[
\text{traverseCompoundVE} \: v \: e = \text{traverseCompoundVDE} \: v \: d \: e \\
\text{where} \ d = \text{traverseDeclarationE} \: e
\]

**Declarations.** We can traverse the initialisers of a declaration using `traverseDeclarationI`, which takes a traversal over initialisers and returns a traversal over declarations.

\[
\text{traverseDeclarationI} :: \text{Monoid}\ m \\
\Rightarrow (\text{Initialiser} \: a \rightarrow m) \\
\Rightarrow \text{Declaration} \: a \\
\Rightarrow m
\]

\[
\text{traverseDeclarationI} \: i = \text{maybe mempty} \: i \circ \text{declarationInitialiser}
\]

By joining `traverseDeclarationI` with `traverseInitialiser`, we get `traverseDeclarationE`.

\[
\text{traverseDeclarationE} :: \text{Monoid}\ m \\
\Rightarrow (\text{Expression} \: a \rightarrow m) \\
\Rightarrow \text{Declaration} \: a \\
\Rightarrow m
\]

\[
\text{traverseDeclarationE} \: e = \text{traverseDeclarationI} \: (\text{traverseInitialiserE} \: e)
\]
A.9 Syntax utility functions

These listings also include samples of the unit and property tests used in the translation process.

A.9.1 ACDC

We test the function signatureName here, though it is defined as part of the record notation for Signature.

\[
\begin{align*}
\text{testSignatureName} &:: \text{Test} \\
\text{testSignatureName} &= \text{testGroup} \text{'signatureName'} \\
&[\text{testCase} \text{"Retrieves the name of 'unsigned int example()'"}]
\text{signatureName} (\text{Signature} (\text{DS NotConst} (\text{Int Unsigned}))) \\
&\text{"example"} \\
&\] \\
&\text{'}? = \text{"example"} \\
&\]

\[
\text{testCase} \text{"Retrieves the name of 'struct s test()'"}
\text{signatureName} (\text{Signature} (\text{DS NotConst} (\text{Struct} "s") \text{"test"} [\]}) \\
&\text{'}? = \text{"test"} \\
&\]

The function signatureParameterNames retrieves the identifiers of all formal parameters in a function signature.

\[
\begin{align*}
\text{signatureParameterNames} &:: \text{Signature} \rightarrow [\text{a}] \\
\text{signatureParameterNames} &= \text{map} \text{parameterName} \circ \text{signatureParameters} \\
\end{align*}
\]

\[
\begin{align*}
\text{testSignatureParameterNames} &:: \text{Test} \\
\text{testSignatureParameterNames} &= \text{testGroup} \text{'signatureParameterNames'} \\
&[\text{testCase} \text{"A signature with no parameters"}]
\text{signatureParameterNames} (\text{Signature} (\text{DS NotConst} (\text{Int Unsigned}))) \\
&\text{"example"} \\
&\] \\
&\] \\
&\text{testCase} \text{"A signature with some parameters"}
\text{signatureParameterNames}
\text{Signature} (\text{DS NotConst} (\text{Struct} "struct")) \\
&\text{"params"} \\
&\text{Parameter} (\text{DS Const} (\text{Int Signed})) \text{"a"} \\
&\ ] \\
&\text{'}? = [\text{"a", "b"}] \\
&\]

The function signatureType returns the return type of a signature. Likewise, parameterType returns the type of a parameter, and declarationType the type of a declaration.

\[
\begin{align*}
\text{signatureType} &:: \text{Signature} \rightarrow \text{Type} \\
\text{signatureType} &= \text{dsType} \circ \text{signatureSpecs} \\
\end{align*}
\]

**Initialisers.** The function traverseInitialiserE traverses an initialiser’s constituent expressions by concatenatively mapping an function from an expression to a list of values. Since Initialiser is a form of Tree, and a Tree is an instance of Foldable, we can use the function foldMap to perform the traversal.

\[
\begin{align*}
\text{traverseInitialiserE} &:: \text{Monoid} m \\
&\Rightarrow (\text{Expression} a \rightarrow m) \\
&\rightarrow \text{Initialiser} a \\
&\rightarrow m \\
\text{traverseInitialiserE} &= \text{foldMap} \\
\end{align*}
\]
APPENDIX A. ADDITIONAL DEFINITIONS

\[
\text{parameterType :: Parameter } a \rightarrow \text{Type} \\
\text{parameterType } = \text{dsType } \circ \text{parameterSpecs} \\
\text{declarationType :: Declaration } a \rightarrow \text{Type} \\
\text{declarationType } = \text{dsType } \circ \text{declarationSpecs}
\]

The function \text{assignmentName} retrieves the base name of an assignment target.

\[
\text{assignmentName :: AssignmentTarget } a \rightarrow a \\
\text{assignmentName (AssignID identifier) } = \text{identifier} \\
\text{assignmentName (AssignArray rec _) } = \text{assignmentName rec} \\
\text{assignmentName (AssignStruct rec _) } = \text{assignmentName rec}
\]

The function \text{treeTarget} takes a map of struct names to their declarations, a \text{Type}, a list of indices signifying a path in an initialiser tree, and an identifier. It traverses the \text{Type}, using each index \(i\) in turn to mean the \(i\)th member of that type, and constructs an assignment target that will assign to the appropriate part of the identified variable.

This is a partial function, as an index could refer to a non-existent member of a type. It is defined in terms of a more general function, \text{convertInitialiserPath}, which performs a conversion from a path through an initialiser into an arbitrary value.

\[
\text{treeTarget} :: [[(Identifier, [StructDeclaration])]] \\
\rightarrow \text{Type} \\
\rightarrow [\text{Int}] \\
\rightarrow a \\
\rightarrow \text{Maybe (AssignmentTarget } a) \\
\text{treeTarget } = \text{convertInitialiserPath } aId aArray aStruct \\
\text{where } aId \text{, } aArray index rec = \text{AssignArray rec (ConstantExpr (CInt index))} \\
\text{aStruct } = \text{flip AssignStruct}
\]

\[
\text{convertInitialiserPath } :: (a \rightarrow b) \\
\rightarrow (\text{Int } \rightarrow b \rightarrow b) \\
\rightarrow (\text{Identifier } \rightarrow b \rightarrow b) \\
\rightarrow [[(Identifier, [StructDeclaration])]] \\
\rightarrow \text{Type} \\
\rightarrow [\text{Int}] \\
\rightarrow a \\
\rightarrow \text{Maybe } b
\]

We define \text{convertInitialiserPath} inductively:

**Base case.** When the list of indices is empty, we yield the identifier to the identifier function.

\[
\text{convertInitialiserPath } iF [] \text{ identifier } = \text{Just (iF identifier)}
\]

**Inductive case.** The most interesting treeTargets are arrays and structs.

For an array, the \text{convertInitialiserPath} is defined as the \text{convertInitialiserPath} of the array’s recursive type if and only if the current index is within the defined bounds of the array. (If the array has no defined bounds, we assume they are infinite).

\[
\text{convertInitialiserPath } iF aF sF s \text{ Array recType bound } (\text{index : indices }) \text{ identifier } \\
\text{guard (inBound bound index) } \Rightarrow \text{arrayWrap } \Rightarrow \text{recurse} \\
\text{where inBound } a b = 0 \leq b \land \text{maybe True (b< a) a} \\
\text{recurse} = \text{convertInitialiserPath } iF aF sF s \text{ recType indices identifier} \\
\text{arrayWrap } = aF \text{ index}
\]

For a struct, the \text{convertInitialiserPath} is defined as the \text{convertInitialiserPath} of the \(i\)th member of the structure, where \(i\) is the current index. If the requested struct does not exist, or there is no \(i\)th member, \text{Nothing} is returned.

\[
\text{convertInitialiserPath } iF aF sF s \text{ Struct structId } (\text{index : indices }) \text{ identifier } \\
\text{lookupStruct } \Rightarrow \text{lookupIndex } \Rightarrow \text{structDeclToTarget} \\
\text{where lookupStruct } = \text{lookup structId s} \\
\text{lookupIndex declList } \\
| \text{index < 0 } \lor \text{length declList } \leq \text{index } = \text{Nothing} \\
| \text{otherwise } = \text{Just (declList ! index)}
\]
structDeclToTarget structDecl
    = structWrap structDecl ≫ recurse structDecl
recurse (StructDeclaration recType _) = convertInitialiserPath iF aF sF s recType indices identifier
structWrap (StructDeclaration _ recId) = sF recId

Finally, any attempt to initialise a member of any other type is an error, and thus we return Nothing:

customInitialiserPath _ _ _ _ _ _ _ = Nothing

testTreeTarget :: Test
testTreeTarget = testGroup "treeTarget"
    [testCase "Array out of bounds (unsigned int foo[10]; foo[10]); failure"
        (treeTarget [] (Array (Int Unsigned) (Just 10)) [10] "foo"
                ●? = Nothing
        ]
    ,testCase "Array in bounds (unsigned int foo[10]; foo[9])"
        (treeTarget [] (Array (Int Unsigned) (Just 10)) [9] "foo"
                ●? = Just (AssignArray (AssignID "foo")
                        (ConstantExpr (CInt 9))
        )
    ,testCase "Array with indeterminate bounds (unsigned int foo[]; foo[9])"
        (treeTarget [] (Array (Int Unsigned) Nothing) [9] "foo"
                ●? = Just (AssignArray (AssignID "foo")
                        (ConstantExpr (CInt 9))
        )
    ,testCase "Simple identifier"
        (treeTarget [] [Int Unsigned] [] "foo"
                ●? = Just (AssignID "foo")
        )
    ,testCase "Simple identifier being accessed as an array; failure"
        (treeTarget [] [Int Unsigned] [9] "foo"
                ●? = Nothing
        )
    ]

testACDCUtils :: Test
testACDCUtils = testGroup "ACDC.Utils"
    [testSignatureName
        ,testSignatureParameterNames
        ,testTreeTarget
    ]

A.9.2 Circus

The function simpleProcess defines a simple process at the CircusPar level:

    simpleProcess :: ZIdentifier -> Proc -> CircusPar
    simpleProcess p = CProcDecl o Process p o SimpleProcDef

The constant skip is shorthand for a skip action:

    skip :: Action
    skip = ACSPAction Skip

µ is shorthand for a fixed point action.

    µ :: ZIdentifier -> Action -> Action
    µ i = ACSPAction o CFixedPoint i

The function apply is shorthand for an action consisting of applying one action to another:
apply :: Action \to Seq1 ZExpression \to Action
apply f = ACSPAction \circ CActionApply (PAction f)

We generalise apply to lists, noting that an application with zero arguments is just the function action itself:

applyList :: Action \to \mathbb{[}ZExpression\mathbb{]} \to Action
applyList = flip applyList'
  where applyList' = maybe id (flip apply) \circ toSeq

The function ifAction lifts a CIf to an Action.

ifAction :: Seq1 GAction \to Action
ifAction = ACommand \circ CIf
ifActionList :: \mathbb{[}GAction\mathbb{]} \to Action
ifActionList = maybe skip ifAction \circ toSeq

The function assignAction lifts a CAssign to an Action:

assignAction :: Seq1 ZIdentifier \to Seq1 ZExpression \to Action
assignAction x y = ACommand (CAssign x y)

The function singleAssignAction, and its operator form \( \_ = \_ \), is shorthand for the action of assigning one variable.

singleAssignAction, (=) :: ZIdentifier \to ZExpression \to Action
singleAssignAction x y = assignAction (Base x) (Base y)

We introduce infix notation for guarded actions:

(\_!\_\_\_\_) :: ZPredicate \to Action \to GAction
(\_!\_\_\_\_) = GAction

We introduce infix notation for prefixes and simple input and output communications.

(\_\_\_\_\_\_) :: Comm \to Action \to Action
(\_\_\_\_\_\_) a = ACSPAction \circ CPrefix a

(\_\_\_\_\_\_\_) a b = Comm a [Input b]

(\_\_\_\_\_\_\_\_) a b = Comm a [Output b]

The function pnot implements predicate negation.

pnot :: ZPredicate \to ZPredicate
pnot (PNegation x) = x
pnot x = PNegation x

A property of pnot is that it cannot introduce a double negation onto a predicate that does not already have one:

pnotDoubleNegation :: ZPredicate \to Property
pnotDoubleNegation p = notAlreadyDoubleNegation p => p
  where notAlreadyDoubleNegation (PNegation (PNegation _)) = False
        notAlreadyDoubleNegation _ = True

predEqual, (. =) :: ZExpression \to ZExpression \to ZPredicate
predEqual = PInfix Equality
(=) = predEqual
predIn :: ZExpression \to ZExpression \to ZPredicate
predIn = PInfix Membership
predNotIn :: ZExpression \to ZExpression \to ZPredicate
predNotIn = PInfix NonMembership
override, (\_ \_\_\_\_\_\_) :: ZExpression \to ZExpression \to ZExpression
override = ZInfixExpression RelationalOverride
(\_ \_\_\_\_\_) = override
mapsto, \(\mapsto\) :: \(\text{ZExpression} \rightarrow \text{ZExpression} \rightarrow \text{ZExpression}\)
mapsto = \text{ZInfixExpression Maplet}

\(\mapsto\) = mapsto

zmod :: \(\text{ZExpression} \rightarrow \text{ZExpression} \rightarrow \text{ZExpression}\)
zmod = \text{ZInfixExpression ZMod}

zplus :: \(\text{ZExpression} \rightarrow \text{ZExpression} \rightarrow \text{ZExpression}\)
zplus = \text{ZInfixExpression ZPlus}

zminus :: \(\text{ZExpression} \rightarrow \text{ZExpression} \rightarrow \text{ZExpression}\)
zminus = \text{ZInfixExpression ZMinus}

We now introduce shorthand for Z declaration parts: singleTypeZDecl, which is shorthand for a declaration part with only one type, and singleZDecl, which is shorthand for a declaration part with only one ID. The latter is given the operator \(\_\_\_\_\).

\quad singleTypeZDecl :: Seq1 \text{ZIdentifier} \rightarrow \text{ZExpression} \rightarrow \text{ZDeclPart}
singleTypeZDecl ids ztype = ZDeclPart (Base (BasicDecl ids ztype))

\quad singleZDecl, \(\_\_\_\_\_\_\_\) :: \text{ZIdentifier} \rightarrow \text{ZExpression} \rightarrow \text{ZDeclPart}
singleZDecl identifier = singleTypeZDecl (Base identifier)

\(\_\_\_\_\_\_\_\) = singleZDecl

The function \text{varCommand} is shorthand for a guarded command introducing variables. The operator \(\bullet\) is provided for it.

\quad varCommand, \(\bullet\) :: \([\text{VarType}, \text{ZDeclPart}] \rightarrow \text{Action} \rightarrow \text{Action}\)
varCommand vars = ACommand \circ CVarBlock vars
\(\bullet\) = varCommand
B Supplemental Files and Running the Process

This project report should have been accompanied by a series of supplemental files. In this appendix, we describe the files, and how to use them to evaluate the project code.

B.1 Supplemental files

demo.tex A \LaTeX file, used to run the case study demo.

mbw500-project.cabal A Cabal file, describing in a machine-readable format the translation process, its dependencies, and how to build and test it.

LICENSE A dummy license file, required for Cabal to work.

src/ The Literate Haskell source code for the translation process, including tests.

scripts/ A series of Bourne shell and Awk scripts used to prepare the project report and evaluate the process on the case study.

testcode/ Code snippets used in some of the unit tests.

case_study/ The source code for the ACDC case study, as well as 'concat.c', which is a pre-processed and concatenated form of the source code ready to input into the process; 'context', which is a serialised CircusifyInput containing the context needed to process the case study; and 'topological', a topological sorting (based on that from Ribeiro) of the case study source code needed to produce the concatenated form.

B.2 Evaluating the process

The supplemental files are sufficient to run the translation process on the case study to generate the Circus model provided in Appendix D. The process of performing this process is outlined below.

- Install the Glasgow Haskell Compiler, version 7.8.2 or above, the Cabal-Install package system, version 1.20 or above, and a \LaTeX distribution;

- Download the Community Z Tools Circus style file, circus.sty [60], and the Z style file, czt.sty [61];

- Extract the supplemental files to a directory and change to it;

- Run ./scripts/demo.sh in the base directory. This will create a new Cabal sandbox, configure and build the program, and run it on the case study. This should output a file, demo.pdf, in the base directory. This may produce dependency resolution errors, in which case the best tactic is to try and manually install the versions of packages given in mbw500-project.cabal and try demo.sh again.

- Once the main project has built, one can optionally run the test suite using cabal test.
C Automated test log

This is the test log automatically generated for the project when it was last subjected to automatic testing.

Test suite test-mbw500-project: RUNNING...

ACDC.Delay:
'isDelayFunction':
   Admits 'sleep' as a delay: [OK]
   Admits 'time' as a delay: [OK]
   Rejects 'printf': [OK]
   Is case sensitive: [OK]

ACDC.Parse:
Prefix expressions:
   Logical NOT: [OK]
Self-assignment expressions:
   Increment: [OK]
   Decrement: [OK]

ACDC.Utils:
'signatureName':
   Retrieves the name of 'unsigned int example()': [OK]
   Retrieves the name of 'struct s test()': [OK]
'signatureParameterNames':
   A signature with no parameters: [OK]
   A signature with some parameters: [OK]
'treeTarget':
   Array out of bounds (unsigned int foo[10]; foo[10]); failure: [OK]
   Array in bounds (unsigned int foo[10]; foo[9]): [OK]
   Array with indeterminate bounds ( unsigned int foo[]; foo[9]): [OK]
   Simple identifier: [OK]
   Simple identifier being accessed as an array; failure: [OK]

Circusify:
Circusify.Assignments:
   assignmentRhs:
      Basic assignment to a scalar: [OK]
      Additive assignment to a scalar: [OK]
      Basic assignment to an array: [OK]
      Additive assignment to an array: [OK]
   circusifyAssignment:
      Function calls in array and expression: [OK]

Circusify.Constants:
   circusifyConstantsPredicate:
      Predicate for one scalar constant: [OK]
      Predicate for two scalar constants: [OK]
      Predicate for an array: [OK]

Circusify.Channels:
   channelIdentifier:
      var_1 becomes var_1SH: [OK]
      i becomes iSH: [OK]
   channelPar:
      An untyped channel: [OK]
      A typed channel: [OK]

Circusify.ChannelSync:
   variablesToRead:
      Ribeiro table, p64, vread:
         Frame 1 - step task_1_step: [OK]
         Frame 1 - step task_2_step: [OK]
         Frame 1 - step task_3_step: [OK]
         Frame 2 - step task_1_step: [OK]
         Frame 2 - step task_2_step: [OK]
         Frame 2 - step task_3_step: [OK]
   variablesToWrite:
      Ribeiro table, p64, vwrite:
Frame 1 - step task_1_step: [OK]
Frame 1 - step task_2_step: [OK]
Frame 1 - step task_3_step: [OK]
Frame 2 - step task_1_step: [OK]
Frame 2 - step task_2_step: [OK]
Frame 2 - step task_3_step: [OK]
variablesToWrite:
  Ribeiro table, p64, vout:
    Frame 1 - step task_1_step: [OK]
    Frame 1 - step task_2_step: [OK]
    Frame 1 - step task_3_step: [OK]
    Frame 2 - step task_1_step: [OK]
    Frame 2 - step task_2_step: [OK]
    Frame 2 - step task_3_step: [OK]
variablesPerFrame:
  Ribeiro p61 ex2: [OK]
coalesce:
  Empty list: [OK]
  List with one item: [OK]
  List with multiple items: [OK]
lastAccessedByFunction:
  Ribeiro p61 ex3: [OK]
Circusify.Expressions:
  fullNameToZ:
    [function, block] name -> function::block::name: [OK]
Circusify.FunctionCalls:
  circusifyFunctionCalls:
    One function call with return value and no arguments.: [OK]
    One function call with no return value and no arguments.: [OK]
functionCallResultIdentifier:
  1 -> result1: [OK]
Circusify.Input:
  declOf:
    Retrieves a valid function: [OK]
    Retrieves a valid constant: [OK]
    Does not retrieve an invalid identifier: [OK]
  typeOf:
    A global variable that is in the type map: [OK]
    A local variable that is in the type map: [OK]
    A name that isn't in the type map: [OK]
Circusify.Process:
  addLocalVariables:
    No local variables.: [OK]
    Local variables with no scoping.: [OK]
  addProcessActionVars:
    Function with no parameters and no return value.: [OK]
    Function with no parameters and a return value.: [OK]
    Function with parameters but no return value.: [OK]
    Function with parameters and return value.: [OK]
Circusify.Step:
  addFrameScheduler:
    Empty scheduler: [OK]
    Simple delay-based step function: [OK]
  functionMapToVMap:
    An empty FunctionMap should return an empty VMap: [OK]
    Example from Ribeiro: [OK]
Circusify.Structs:
  structCToZ:
    Example 1: a typical struct: [OK]
    Example 2: an empty struct: [OK]
  structDeclCToZ:
    Example 1: a signed integer: [OK]
    Example 2: a floating point number: [OK]
Circusify.Types:
  typeCToZ:
    struct structName: [OK]
    unsigned int[]): [OK]

Circusify.Variables:
  isLocalVariableRecordOf:
    Variable that is a local variable of the given function: [OK]
    Variable that is not a local variable of the given function: [OK]
    Variable that is not a local variable of any function: [OK]

Constructs.Frame:
  rotateFrame:
    Zero: [OK]
    Forwards: [OK]
    Forwards with wrap: [OK]
    Backwards: [OK]
    Backwards with wrap: [OK]
    Rotating is closed over the range [1 .. count]: [OK, passed 100 tests]
    Rotating by a value then its negation is the identity: [OK, passed 100 tests]
    Rotating by 0 is the identity for valid frame counts: [OK, passed 100 tests]

Constructs.VMap:
  'variablesInVMap' on an empty VMap will fail: [OK, passed 100 tests]
  variablesInVMap:
    Variables that are in the VMap:
      1, test, Read: [OK]
      2, test, Read: [OK]
    Variables that are not in the VMap:
      3, test, Read: [OK]
      2, test, Write: [OK]
      1, no, Write: [OK]
  Ribeiro varRead example, p61: [OK]

Preprocess:
  Preprocess.AbstractTime:
    assignmentTemporal:
      a = 3 -> Any True: [OK]
      f[a] = 4 -> Any True: [OK]
      g = b -> Any True: [OK]
      h = 3 -> Any False: [OK]
      i[j] = 4 -> Any False: [OK]
      g = k -> Any False: [OK]
    hasTemporalVariables:
      a -> Any True: [OK]
      a + 4 -> Any True: [OK]
      3 -> Any False: [OK]
  Preprocess.DeclCapture:
    declCaptureDeclaration:
      Report example (valid constant): [OK]
      Report example (ignored variable): [OK]
  Preprocess.Flatten:
    'flattenAssignmentTarget':
      'test = ...' becomes 'scoped::test = ...': [OK]
      'test.member = ...' becomes 'scoped::test.member = ...': [OK]
  Code equality tests:
    Flattening flatten-general-in.c yields flatten-general-out.c: [OK]
    Flattening flatten-number-in.c yields flatten-number-out.c: [OK]
    Flattening flatten-while-in.c yields flatten-while-out.c: [OK]
    Flattening flatten-switch-in.c yields flatten-switch-out.c: [OK]
  Flatten.Functions:
    flattenParameter:
      Simple parameter.: [OK]
    flattenSignature:
void empty();: [OK]
unsigned int hasParams(float a, float b);: [OK]

Flatten.Enums:
  'envEnumerator':
    Registers an implicit enum: [OK]
    Registers an explicit enum: [OK]
  'flattenExpression':
    Flattens a scope-local name: [OK]
    Flattens a global name: [OK]

getStructs:
  Nothing defined: [OK]
  One struct defined: [OK]
  Two structs and some noise defined: [OK]

Preprocess.Reachability:
  Function calls:
    example(): [OK]
    sleep(3): [OK]
    nested(argument()): [OK]
    nested(time()): [OK]

Utils.Transitive:
  'transitive':
    Preserves the empty list: [OK]
    Produces the correct closure (simple): [OK]
    Produces the correct closure (less simple): [OK]
    Cannot reduce the number of unique items: [OK, passed 100 tests]
    Is idempotent: [OK, passed 100 tests]
    Produces a list with no duplicates: [OK, passed 100 tests]
    Contains all elements in its input: [OK, passed 100 tests]
  'transitiveStep':
    Preserves the empty list: [OK]
    Contains (a, c) for every (a, b) and (b, c): [OK, passed 100 tests]
    Can only grow the number of unique elements in a list: [OK, passed 100 tests]
    Contains all elements in its input: [OK, passed 100 tests]

Utils.Tree:
  Each path leads back to its element: [OK, passed 100 tests]
  map fst . treeToPathList === toList: [OK, passed 100 tests]
  toList (on trees):
    Single leaf: [OK]
    Branch of leaves: [OK]
    Branch of branches of leaves: [OK]
    Complex structure: [OK]

<table>
<thead>
<tr>
<th>Properties</th>
<th>Test Cases</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passed</td>
<td>13</td>
<td>132</td>
</tr>
<tr>
<td>Failed</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>13</td>
<td>132</td>
</tr>
</tbody>
</table>

Test suite test-mbw500-project: PASS
Test suite logged to: dist/test/mbw500-project-0.1.0.0-test-mbw500-project.log
D \textit{Circus} model of the case study

\[ ST_{\text{diff	extunderscore out}} ::= \text{out	extunderscore mem} \mid \text{output} \]

\textbf{channel} frame : \mathbb{N}

\textbf{channel} end\_cycle

\textbf{channel} errorSH : \mathbb{R}

\textbf{channel} iSH : \mathbb{R}

\textbf{channel} dSH : \mathbb{R}

\textbf{channel} kpSH : \mathbb{R}

\textbf{channel} kiSH : \mathbb{R}

\textbf{channel} kdSH : \mathbb{R}

\textbf{channel} positionSH : \mathbb{R}

\textbf{process} exec\_1 \triangleq \\
\text{begin} \\
\text{state} \left[ p : \mathbb{R}, kp : \mathbb{R}, error : \mathbb{R}, position : \mathbb{R}, i : \mathbb{R}, d : \mathbb{R} \right] \\
\text{task\_1\_init} \triangleq \text{Skip} \\
\text{task\_1\_step} \triangleq \\
\left( \begin{array}{c}
\text{if \text{current\_frame} = 1} \\
\left( \begin{array}{c}
\text{if \text{current\_frame} = 2} \\
\left( \begin{array}{c}
\text{calc\_proportion} \\
\text{calc\_output} \\
\text{end\_cycle} \rightarrow \text{Skip}
\end{array} \right)
\right)
\end{array} \right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
\right)
process exec_2 ≡
begin
  state [ i : R, error : R, ki : R ]
  task_2_init ≡
    init_integral
  task_2_step ≡
    frame?current_frame →
      ( \ if current_frame = 1 →
          ( ( errorSH?x → ( error := x ) ), { error } )
          ( ( kiSH?x → ( ki := x ) ), { ki } )
          calc_integral
      )
      \ if current_frame = 2 →
        ( ( iSH!i → Skip ), {} )
        end_cycle → Skip
      \ if current_frame ∉ { 1, 2 } →
        Skip
      fi
    init_integral ≡
    i := 0.0
  calc_integral ≡
    var result1 : U •
    ( integ( error, ki, i, result1 ) ; )
    i := result1
  integ ≡
    val integ::input : R, integ::k : R, integ::output : R;
  res_result : R •
    ( integ::output := integ::output + ( integ::k * integ::input ) ;
      _result := integ::output
    )
  task_2_init
  μX • ( task_2_step ; )
end
process exec_3 ≡
begin
task_3_init ≡
init_derivative

init_derivative

diff_mem := 0.0
calc_derivative ≡

var calc_derivative::dout : STdiff_out → U •
var result1 : U •

( diff( error, kd, diff_mem, result1 ) ;
  calc_derivative::dout := result1;
  diff_mem := calc_derivative::dout output;
  d := calc_derivative::dout output)
diff ≡

val diff::input : R, diff::k : R, diff::in_mem : R;
res_result : STdiff_out → U •

var diff::ds : STdiff_out → U •

( diff::ds := diff::ds ⊕ { output := { diff::k * ( diff::input - diff::in_mem ) } } ;
  _result := diff::ds
  _result := diff::ds )

• task_3_init;
μX • task_3_step ;
end

process exec_0 ≡
begin
state [ current_frame : N ]
init_frame ≡
current_frame := 1
next_frame ≡
current_frame := ( current_frame mod 2 ) + 1

• init_frame;
μX •

μX • current_frame → ( if ¬ ( current_frame = 2 ) →
  Skip
  current_frame = 2 →
  ( end_cycle → Skip )
  fi ;
  next_frame

process Program ≡

( ( exec_0 ) ; ( frame, end_cycle )
 ( exec_1 ) ; ( errorSH, isH, dSH, kpSH, positionSH, frame, end_cycle )
 ( exec_2 ) ; ( errorSH, isH, kiSH, frame, end_cycle )
 ( exec_3 ) ; ( errorSH, dSH, kdSH, frame, end_cycle )
 ) \ { frame, iSH, dSH }
E Glossary

**ACDC** The subset of C we define in chapter 3. ACDC programs are the input to our technique.

**ACDC main function** An entry point to a particular task scheduler in our cyclic executive. This concept is different from the `main()` function in C and ACDC.

**Concatenative map** A map from a list of a to a list of lists of b, followed by the concatenation of each such list to form one list of b. Generally implemented by `concatMap`.

**Functor** A data structure that can be mapped over, preserving its structure and mutating its inner values. Examples include lists, `Maybe` constructs, and non-empty sequences.

**Maybe** A type representing an optional value [62]. Given a type a, the type `Maybe a` can hold either the value `Nothing`, representing an absence of a value, or `Just a`, representing a presence. `Maybe` is a form of functor.

**Monad** A data structure that captures a computational context. A monad allows values to be lifted into it using `return`, and provides an operator `>>=` for binding the value from one computation into the creation of another.

**Monoid** A data structure with an empty value, `mempty`, and an associative binary operation, `mappend` (or the `<*>` operator) [52]. Monoids found in our technique are lists (`mempty` is the empty list, `mappend` is list concatenation), Booleans under `Any` (`mempty` is false, `mappend` is disjunction), and Circus actions (`mempty` is `Skip`, `mappend` is sequential composition).

**Partial function** A function that is undefined on some of its input domain. We either explicitly express this partiality using `Maybe`, or simply terminate the process with an error if an unsupported value is found. When a pair of functions exist that implement both styles, we use the convention of labelling the latter form with `P` (for partial).

**Phase** A discrete step in the translation process, defined by a top-level function and potentially more sub-functions.

**Scope-flattening** The renaming of variables in an ACDC program such that each variable name is fully qualified in terms of the scope in which it is defined.

**Translation process** Unless otherwise qualified, refers to the technique for translating ACDC programs to Circus models elaborated in chapter 4.

**Traversal function** A function that traverses an ACDC syntactic construct, given functions for traversing aspects of that construct, and applies said functions when those aspects are reached. Each traversal returns a monoid, and performs a monoidal concatenation of the results of its sub-traversals. Much of the pre-processing phases of the project are defined in terms of traversals.

E.1 Haskell syntax used in the project

For conciseness, we make use of high-level Haskell concepts and combinators in the project deliverable. Below is a reference guide to some of the less obvious syntax used.

We first outline the operators we define ourselves to capture Circus concepts in a more concise manner. Then, we summarise the Haskell operators and idioms we use in the project.

<table>
<thead>
<tr>
<th>Syntax (in report)</th>
<th>Syntax (in code)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>µ</td>
<td>µ</td>
<td>Fixed point action</td>
</tr>
<tr>
<td>≡</td>
<td>≡</td>
<td>Assignment guarded command for one variable</td>
</tr>
<tr>
<td>←→</td>
<td>&quot;→</td>
<td>Guarded action</td>
</tr>
<tr>
<td>‒→</td>
<td>.→</td>
<td>Communications prefixing</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>Input communication for one variable</td>
</tr>
<tr>
<td>!</td>
<td>!</td>
<td>Output communication for one variable</td>
</tr>
<tr>
<td>. =</td>
<td>.=</td>
<td>Predicate equality</td>
</tr>
<tr>
<td>⊕</td>
<td>⊗</td>
<td>Z relational override infix expression</td>
</tr>
<tr>
<td>⊳</td>
<td>⊳</td>
<td>Z maplet infix expression</td>
</tr>
<tr>
<td>. :</td>
<td>. :</td>
<td>Z declaration part with one declaration</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>Variable block introduction</td>
</tr>
</tbody>
</table>

Table E.1: Custom operators defined in the translation process.
<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax (in report)</th>
<th>Syntax (in code)</th>
<th>Description</th>
<th>Example</th>
<th>Defined at</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functor map/lift</td>
<td>fmap</td>
<td>fmap</td>
<td>Given a function ( f ) and a functor context ( c ), lifts ( f ) into ( c ). Often implies the mapping of ( f ) over the value(s) in ( c ).</td>
<td>On lists, <em>fmap</em> is <em>map</em>; it is the analogous function on <em>Seq1s</em> and <em>Trees</em>. On <em>Maybe</em>, <em>fmap</em> ( f ) <em>Nothing</em> is <em>Nothing</em> and <em>fmap</em> ( f ) <em>Just</em> ( a ) is <em>Just</em> ( f(a) ). For stateful computations, <em>fmap</em> lifts a function on values to a state-ignoring computation.</td>
<td>Page 53</td>
</tr>
<tr>
<td>Infix <em>fmap</em></td>
<td>&lt;$&gt;</td>
<td>&lt;$&gt;</td>
<td>As <em>fmap</em>, but in infix operator form.</td>
<td>See <em>fmap</em>. The syntax given here is equivalent to ( f i = a(b i)(c i)(d i) ).</td>
<td>Page 41</td>
</tr>
<tr>
<td>Repeated argument</td>
<td>( a &lt;$&gt; b &lt;$&gt; c &lt;$&gt; d )</td>
<td>( a &lt;$&gt; b &lt;$&gt; c &lt;$&gt; d )</td>
<td>Use of infix functor map and applicative functor application operators to elide an argument applied to every argument of a function.</td>
<td>[1, 2, 3] ( \diamond ) [4, 5, 6] is [1, 2, 3, 4, 5, 6].</td>
<td>Page 45</td>
</tr>
<tr>
<td>Monoidal append</td>
<td>( \diamond )</td>
<td>&lt;&gt;</td>
<td>Appends one monoidal value to another, using the appropriate operation: for lists, concatenation; for <em>Circus</em> actions, sequential composition; for <em>Z</em> predicates, newline conjunction; for Booleans under <em>Any</em>, disjunction, etc.</td>
<td></td>
<td>Page 56</td>
</tr>
<tr>
<td>Fan-out</td>
<td>&amp;&amp;&amp;</td>
<td>&amp;&amp;</td>
<td>Given two functions ( f ) and ( g ), is the function ( h x = (f x, g x) ).</td>
<td>[ (+10) &amp;&amp;&amp; ( +100 ) ] 1 = (11, 110).</td>
<td>Page 36</td>
</tr>
<tr>
<td>Split</td>
<td>***</td>
<td>***</td>
<td>Given two functions ( f ) and ( g ), is the function ( h (x, y) = (f x, g y) ).</td>
<td>[ (+10) *** ( +100 ) ] (1, 10) = (11, 110).</td>
<td>Page 49</td>
</tr>
<tr>
<td>Monadic bind</td>
<td>( \gg = \ll )</td>
<td>( \ggg = \lll )</td>
<td>Passes the value from one monadic computation into another, in the direction of the operator.</td>
<td>For stateful computations, binding sends the value from one computation to another while implicitly also passing the state.</td>
<td>Page 59</td>
</tr>
<tr>
<td>Monadic then</td>
<td>( \gg )</td>
<td>( \gg )</td>
<td>Sequences one monadic computation before another, in the direction of the operator, but discards its value.</td>
<td>For stateful computations, <em>then</em> performs the state changes in the leftmost computation, and passes the resulting state implicitly into the next computation.</td>
<td>Page 60</td>
</tr>
</tbody>
</table>

Table E.2: Summary of Haskell syntactic forms used in the project.


