School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Verifying C Data Types for Cogent

by

Louis Francis Cheung

Thesis submitted as a requirement for the degree of

Computer Science Honours

Submitted: 4 January 2020 Student ID: z5062193

Supervisor: Dr. Christine Rizkallah Topic ID: NA

Abstract

Cogent is a restricted, pure, polymorphic functional programming language with uniqueness types and does not require a trusted runtime or garbage collector. It has a certifying compiler which produces an Isabelle/HOL shallow embedding of the given Cogent program, a C program and an Isabelle/HOL proof that the C program is a refinement of the shallow embedding. Cogent was designed to be used to implement trustworthy and efficient systems code. Cogent currently does not support recursion, instead recursion and other forms of iteration are defined in C functions and these can be called from Cogent using its foreign function interface (FFI).

The main task of this thesis is to verify the functional correctness of Cogent's 32-bit word array data type which is implemented in C. In addition, we investigate what steps would be required to allow us to prove functional correctness of word arrays with standard word sizes. Word arrays are a commonly used data structure, especially in low level systems. By providing a verified implementation of word arrays, we would reduce the cost of verification of any project that requires word arrays. This thesis is the first step towards attaining a verified library of commonly used ADTs, which cannot be implemented in Cogent natively.

Acknowledgements

I would like to acknowledge my supervisor Dr Christine Rizkallah and my assessor Aprof. June Andronick for their guidance and support throughout my thesis. I would also like to thank Liam O'Connor-Davis, Zilin Chen, Amos Robinson and Vincent Jackson for all the help they have provided during this thesis. I would like to thank my family friends for their continued support throughout the years.

Contents

Chapter 1

Introduction

A program should always realise the intentions of its users. This is especially true for high critical systems where failure can have disastrous consequences. Typically, programmers use tests to determine if their programs are correct. However, tests cannot prove that a program is free from unwanted behaviours, i.e. bugs, unless it exhaustively tests all possible inputs. This approach is infeasible if the set of possible inputs is large. The solution to this is formal verification. With formal verification, we can prove that program is correct with respect to its functional correctness specification, which is the formalisation of the program's intended behaviours. The feasibility of formal verification is not affected by the size of the input set.

Typically, ease of formal verification comes at the cost of a decrease in performance for the program. With a decrease in performance, a program will still work "correctly", however, the program may become slower or require more resources which limits where these programs can be used. In low level systems code, resources are limited and timing may be critical to its usability. For example, if it takes 10 seconds for an OS to register a key press, no one would be willing to use it on a daily basis even it is the most secure and reliable OS in the world.

Cogent [28] is a programming language designed to make formal verification easier for low level systems code, such as device drivers and file systems, without reducing perfor-

mance significantly. It is a purely functional, high level language with uniqueness types. It has a certifying compiler which produces an Isabelle/HOL [25] shallow embedding of the program, a C program, and an Isabelle/HOL proof that the generated C program is a correct refinement of the Isabelle/HOL shallow embedding. The Isabelle/HOL shallow embedding is a representation of the Cogent program's semantics using equivalent expressions in Isabelle/HOL. High performance low level systems are typically written in C because is fast and does not require many resources. Since Cogent programs compile to C programs, their performance should be similar to the performance of written C programs.

For most high level languages, a garbage collector is required for memory management, and thus memory safety. Cogent doesn't require a garbage collector because it uses uniqueness types for heap objects. Uniqueness types allow efficient and destructive updates, and static memory allocation. Uniquely typed object must be used exactly once, so Cogent heap object must eventually be freed and once it has been freed it cannot be used ever again. Therefore, memory safety is guaranteed by Cogent's type system. When compared to other high level languages, Cogent has a much smaller trusted computing base (TCB) since it does not require a garbage collector and its language runtime environment, which is the same as C's, is very small.

Cogent makes verification easier because it does not have mutable state and can be easily shallowly embedded into Isabelle/HOL. Shallow embeddings are easier to reason about because it is easier to apply equational reasoning Myreen [24], and since there is no mutable state, reasoning becomes as simple as reasoning about mathematical functions. In formal verification, we typically want to be able to substitute equivalent expressions or terms in our proofs, and this is exactly what equational reasoning allows us to do.

Recursion is not natively available in Cogent. This design choice was made so that certifying compilation could be implemented more easily. This is due to Isabelle/HOL requiring that all functions need to be total, i.e terminating and well-defined, in proofs for total correctness. Recursion can still be used in Cogent by defining the recursion

in C functions and calling these C functions through Cogent's FFI. However, these C functions need to be verified manually. For Cogent' use cases, recursion is only required when iterating over some abstract data types (ADT), such as arrays and lists. ADTs are designed to be reused in multiple programs, so if an ADT implementation is verified, its cost is amortized.

Despite these restrictions, Cogent is not a toy language. It has been successfully used to build a log structured file system called BilbyFS [1] as well as Linux's ext2 file system Amani et al. [2]. The file system operations **sync** and **iget** have partial correctness proofs. The proofs assume the correctness of several ADTs and Linux's unsorted block image (UBI) erase block management layer. To have proof of total correctness, both Linux's UBI layer and Cogent's ADT library would need to be verified.

It is believed that verifying Cogent's ADTs should be an easy task. As of yet, no work on this has been done. This thesis aims at to take the first step towards attaining a verified library of commonly used ADTs, which can be used in Cogent programs. This is achieved by verifying the correctness of Cogent's 32-bit word array ADT. In addition we investigate how we could extend these proofs for word arrays of standard word sizes, where possible. Besides verifying correctness of the standalone 32-bit word array ADT, we also need to ensure that the current implementation is compatible with Cogent's type system. These constraints for compatibility are defined in Cogent's frame constraints. To do we show that the current implementation of 32-bit word arrays respects Cogent's frame constraints. We chose 32-bit word arrays to be the first ADT to be verified because they are a commonly used ADT. They are also one of the ADTs used in BilbyFS, so by verifying 32-bit word arrays, we also increase the trustworthiness of the partial proofs of correctness for BilbyFS.

Chapter 2

Background and Related Work

We begin our investigation with an overview of formal verification, Cogent and its C ADT library, and existing verified ADTs in other languages.

2.1 Formal Verification

Formal verification has three main components, these are: defining a functional correctness specification, proving functional correctness, and validating that the proofs are correct. Each component is crucial, since if one is lacking or non-existent, the overall verification result either loses usefulness or credibility. In addition, each of these components can be quite difficult tasks, as it depends on the complexity of the functional correctness specification, level of proof infrastructure available, and the availability of verification tools.

2.1.1 Functional Correctness Specifications

The functional correctness specification is probably the most important component. This is because the functional correctness specification contains the intended behaviour of the program we want to verify, i.e. it is our definition of correctness. If it is lacking,

i.e. does not encapsulate the intended behaviours of the program we want verify or possibly incorrect, then even if we are able to verify correctness, this correctness which we obtained may not be the actual correctness we want. For example, suppose we have program which takes two numbers as input and returns the addition of those two numbers, and we want to verify this program is correct. Suppose that our functional correctness specification for this program is that it returns a number. Even if we verify functional correctness with strong guarantees, this result is meaningless because we would not be able to determine whether or not the program we wrote actually returned the addition of the two numbers it received.

Functional correctness specifications should be rich, two-sided, formal and live [4]. A rich specification describes how a program should behave in full detail, i.e. without leaving out the especially complex behaviours. This would help to prevent unexpected new "features" from popping up that often occurs in software development. A two-sided specification is a specification that actually matches how a user intends to use the proposed program and that the proposed program is actually implementable. This ensures that specification contains the intended behaviours of the program, i.e. we are building the correct program, and such a program can actually exist. Specifications should be written formally. This means that they are written in formal logic and their meanings are unambiguous. Using formal logic is important, since we need to be able to reason about the specification. This also makes it amenable to verification tools such as proof checkers. An ambiguous specification would hamper verification efforts as it would make it difficult to discern the correct behaviour from the specification. A specification is live if the specification is linked to an implementation of the program it specifies via machine-checkable proofs. This ensures that the specification is used to prove correctness about the implementation and the proofs are sound, assuming that the proof checker has strong guarantees of correctness. All of this ensures that if a program is verified with a functional correctness specification, which is rich, two-sided, formal and live, then we know, with strong guarantees, that the program is actually correct, and we know that we built the correct program.

The level of detail that is present in a functional correctness specifications can affect

the cost of verification. Specification with a low amount of detail can generally be written using logics with a low degree of expressiveness, such as propositional logic. Statements written in logics with low expressiveness are more amenable to automatic methods [19]. For example, propositional logic formulae can be transformed into a Boolean satisfiability (SAT) problems and solved using SAT solvers. These SAT solvers have become powerful enough to solve formulae with millions of variables [7]. Highly detailed specifications generally need to be written in logics which have a high degree of expressiveness, such as higher order logic, and proofs in these logics normally require more human guidance.

2.1.2 Proving Functional Correctness

Once a functional correctness specification and the program has been implemented, we are then able to prove function correctness. The first step is to embed the program into the formal logic which we will use to verify correctness. A key requirement for this step is that the language used to implement the program must have a formal semantic, i.e. each syntactic construct is assigned a logical meaning. With a formalisation, we know what each statement of our program means, and hence, we are able to avoid situations where the meaning of some particular statement is ambiguous or undefined, and we can actually reason about our program. Not all languages have formalisations, so we cannot formally prove correctness for programs written in these languages. Other languages only have formalisations for a subset of the language, like C [26]. We can verify the correctness of programs written in these languages so long as it is only written using the formalised subset. Once we have embedded our program into formal logic, preferably the same logic that we defined our specification in, we can then prove correctness of the program by reasoning about the embedding and specification.

We can embed a program into a formal logic either as a deep embedding or a shallow embedding. In a deep embedding, the program is modelled as a data type in the formal logic. In a shallow embedding, the program is represented using equivalent expressions in the formal logic. Depending on the implementation language and the logic used,

one of these approaches may be easier to do. Typically, if the language and logic have similar paradigms, then generating a shallow embedding may be easier. On the other hand if the language and logic do not have similar paradigms, generating a deep embedding may be easier.

Deep embeddings and shallow embeddings have their pros and cons. Deep embeddings are useful if we want to prove properties which involve the structure of the language. Shallow embeddings are more suited to prove properties about the meanings of expressions in the shallow embedding. This difference can be seen in the following equation:

$$3 + 4 = 7. \tag{2.1}$$

Using basic arithmetic, we know that the left hand side of equation 2.1 is equal to the right hand side, i.e. the left hand side is semantically equivalent to the right hand side. We can also see that the left hand side is structurally different from the right hand side as two numbers and a plus sign are different to a single number. When we prove functional correctness of a program, we are mainly concerned with the semantics of the program. Thus, a shallow embedding is generally easier to use when proving correctness. The reason for this is that in a shallow embedding, it is much easier to show that two terms are semantically equivalent by applying equational reasoning [24]. Equational reasoning, is basically substitution of equivalent terms. This type of reasoning is quite easy to do as even high school children use this type of reasoning when doing algebra. To show that two expressions are semantically equivalent in a deep embedding, evaluation and substitution functions must be defined. These evaluation and substitution functions themselves need to be proven correct which becomes difficult to do when variables can be renamed [34].

For programs written in a pure functional language, we can reason about them using standard mathematical techniques. For programs written in languages which have mutable state, such as C, we typically use program logics such as Floyd [12] which was later improved by Hoare [17]. In Hoare logic, the key element is the Hoare triple which is of the form:

$$\{P\}\ C\ \{Q\}. \tag{2.2}$$

This describes how the execution of the program code C changes the program state. In addition, if we have a valid Hoare triple, then if the precondition P holds before C is executed, then the postcondition Q is established after C is executed. For a simple imperative language, Hoare logic has several axioms and inference rules for the main constructs of the language, and from these rules, rules for other language constructs can be derived. The key axioms and rules are the following:

$$\frac{}{\{P\}\ \textbf{skip}\ \{P\}} \tag{2.3}$$

$$\frac{}{\{P[E/x]\}\ x := E\ \{P\}} \tag{2.4}$$

$$\frac{\{P\}\ S\ \{Q\},\ \{Q\}\ T\ \{R\}}{\{P\}\ S;\ T\{R\}} \tag{2.5}$$

$$\frac{\{P \wedge B\}\ S\ \{Q\},\ \{P \wedge \neg B\}\ T\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{endif}\ \{Q\}} \tag{2.6}$$

$$\frac{P \longrightarrow P',\ \{P'\}\ C\ \{Q'\},\ Q' \longrightarrow Q}{\{P\}\ C\ \{Q\}} \tag{2.7}$$

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \textbf{endwhile}\ \{P \wedge \neg B\}} \tag{2.8}$$

The Hoare axiom for the empty statement 2.3 states, since the **skip** statement, which does nothing, does not change the state, then whatever assertion held before **skip** still holds afterwards. The assignment axiom 2.4 states that if the assertion P, in which all occurrences of the free variable x have been replaced with the expression E, is true, then after the assignment $x := E$, P is true. The composition rule 2.5, states that for two programs S and T, if the postcondition of S is the precondition of T, then if the precondition of S is true, then after the program $S;\ T$ is executed, the postcondition of T is true. The conditional rule 2.6 states that the postcondition Q, which is common to both branches of the condition statement, is established if the precondition P is true. The consequence rule 2.7 allows the strengthening of the precondition and the weakening of the postcondition of a Hoare triple. The while rule 2.8 states that if the assertion P is preserved by the execution of the program C, then even after repeated executions of C, P still holds assuming that P held initially. The assertion P is known as a loop invariant.

With standard Hoare logic, only partial correctness can be proved, because in the Hoare triple 2.2, if C does not terminate then we can set the postcondition to be anything, since it will never be established. Often, the rules in Hoare logic are extended to include termination so that it can be used to prove total correctness, i.e. in the Hoare triple 2.2, the program C terminates and so the postcondition is guaranteed to be established.

2.1.3 Verification Tools and Proof Validation

In formal verification, we not only need to generate proofs of correctness, we also need to convince others that these proofs themselves are correct. Clearly hand written proofs which are only checked by human eyes would be prone to errors and thus unconvincing. To solve this, we can use verification tools to validate and even generate the proofs for us. Even though these verification tools could also potentially produce errors, they are less likely to, however, if they do, they tend to fail to generate proofs rather than generate unsound proofs. We provide a description of a few of the wide range of verification tools which exist. We focus on verification tools which are designed to verify C programs, since the main tasks of this project is to verify the correctness of C data types.

There exist many highly automated verification tools. These tools make verification very easy because they basically only require a push of a button to work. An example of this kind verification tool is SLAM2 [5]. SLAM2 uses the counterexample-guided abstraction refinement (CEGAR) algorithm to test if a given property holds for a program. CEGAR works by first generating an abstraction of the given program as a boolean program with the information relating to the property that we want to prove. It then applies model checking to determine if the model satisfies the property by attempting to find counterexamples. If it finds a counterexample, it then checks whether or not this counterexample is also a counterexample for the original program. If it is, we then know that the program did not satisfy the given property. If the counterexample does not apply to the original, it then adds the new information it obtained from the counter example to the abstraction. It then repeats this process

until it finds an actual counterexample or until it cannot find any and times out. Tools like SLAM2 are easy to use but can fail to actually prove the properties requested and the properties they can prove are limited.

Other verification tools require us provide hints or annotations to the code to guide the verification. These tools are also quite convenient, although not to the extent of more highly automated tools. However, they can generally prove more properties than highly automated tools. One such verification tool is VCC [11], which is an assertional, first order deductive code verifier for C programs. It uses ghost state and code in its reasoning, and requires the C code to be annotated with data invariants, loop invariants and function contracts. The majority of these annotations form the specification for the C code. Since the specification is mostly defined as assertions on the C code in first order logic, this still limits the scope of what can be proved.

Verification tools that require the most human interaction are interactive theorem provers. With these, we are able to verify programs with richer functional correctness specifications, such as the seL4 microkernel [20] and the CompCert C compiler [23]. These were proved in the theorem provers Isabelle/HOL [25] and Coq [6] respectively. In addition, the families of theorem provers which these two tools belong to, are less likely to produce proofs which are unsound. For the theorem provers like Coq, whose logic is constructive, we can transform constructive proofs about formulae into algorithms to generate explicit values for these formulae. The theorem provers like Isabelle/HOL, which are LCF style theorem provers [13], have a small kernel of functions. These functions, which correspond to the axioms and inference rules of the logic that is being implemented, can derive new theorems using only the inference rules and the axioms. Since all proof components are derived from the kernel, any unsoundness can only occur if the kernel itself is unsound. Since this is small, it is very unlikely.

In this thesis, we use Isabelle/HOL to verify correctness of Cogent's ADT library for the reasons that Cogent programs are shallowly embedded in Isabelle/HOL, which means that verification of Cogent programs will occur in Isabelle/HOL. This choice means that we do not have to later port different logics when verifying Cogent programs in

their entirety, i.e. without assuming correctness of the ADT library. In addition, there exists other verification tools which plugin to Isabelle/HOL which make verification of C programs easier.

Isabelle/HOL and C Verification

Isabelle/HOL is an interactive theorem prover whose logic is instantiated to classical higher order logic. It contains a rich library of already formalised theories. Proofs in Isabelle/HOL can be written in backward and forward style. Backward style proofs are similar to "discovery" proofs which works backwards from the goal to the given facts. Forward style proofs are similar to proper mathematical proofs that start from the facts to arrive at the goal. The former is easier when trying to discover a proof, but the latter is more readable. Isabelle/HOL also contains many automated tools like its term rewriting engine and automatic proof methods, making verification easier.

Sequential imperative programming languages, like C can easily be modelled, in Isabelle/HOL using SIMPL [31], which is a generic language model for sequential imperative languages. It is both deeply and shallowly embedded in Isabelle/HOL. Statements/commands in SIMPL are deeply embedded and basic operations and expressions, e.g. $x + 1 < 3$, can be shallowly embedded. SIMPL's core language constructs can be used to model variable assignment, branches, loops, sequential instructions, non-determinism, no operations, and exceptions. It can also model the following states of computation:

1. *Normal s*: The program is current executing normally with current state s, where s contains other state information.
2. *Abrupt s*: The program is currently propagating an exception.
3. *Fault f*: The program is in an irrecoverable state with fault *f*.
4. *Stuck*: The program is stuck, i.e. the execution cannot proceed any further.

There exists many theorems and lemmas proven about SIMPL. Hoare logic for both partial and total correctness has been developed for SIMPL as well as automatic methods such as a verification condition generator.

C programs can be embedded into Isabelle/HOL using the C-to-Isabelle parser [33] tool. This is tool that translates a large subset of C99 into SIMPL. Although the translation is not proven to be correct, the C-to-Isabelle parser attempts to make the most literal translation from C to SIMPL. That way, the translation is more likely to be correct. C's abrupt terminations commands **break** and **continue**, can be modelled using SIMPL's exceptions. For example, for the following C program:

```
while (condition) {
    break;
}
```

would be modelled as the following:

```
TRY {
    WHILE (condition) {
        exception = 'break';
        THROW;
    }
} CATCH {
    IF (exception == 'break')
        SKIP;
    ELSE
        THROW;
}
```

For places where undefined behaviour could occur, SIMPL's guard command can be used like assertions. A correct C program will satisfy all of the guards, while an incorrect

C program may violate some of the guards. SIMPL's guard command has the following form :

```
GUARD f g c
```

In the guard expression, if the condition `g` is satisfied, then the SIMPL command `c` would be executed. If the condition `g` is not satisfied, the execution state is set to *Fault* `f`. An example where guards may be used are expression which contain the division operation. A guard is used to ensure no division by 0 occurs, since division by 0 is undefined in C.

The C SIMPL embedding produced by the C-to-Isabelle parser is not the easiest representation to reason about as it is designed to make the translation from C to SIMPL as literal as possible, and is partially deeply embedded. To make this more pleasant to reason about, we can use the tool AutoCorres [15, 16, 14]. AutoCorres takes the output of the C-to-Isabelle parser and transforms it into a monadic shallow embedding, where local variables are transformed into Isabelle/HOL bounded variables. Although AutoCorres is not verified, it produces a machine-checkable proof that its translation is correct. Thus, we do not need to trust its correctness. It uses a monad to represent the mutable state of the program. The monad that is used in the monadic shallow embedding is an exception monad[10] which is a type of state monad. It is defined as follows:

$$('s,'a,'e)\ monadE\ =\ 's\ \Rightarrow\ (('e\ +\ 'a)\ \times\ 's)\ set\ \times\ bool$$

The monad takes a state of type $'s$ and returns a tuple which contains a set of results of executions and a boolean flag which signals whether any of the executions failed. A result of an execution contains a return value and the resulting state. This return value can either be a normal return value of type $'a$ or an exception of type $'e$. A set of these is returned to model non-determinism. To model sequential execution the monadic bind operator $>>=_E$ is used. The state contains the heap which is modelled as a collection of heaps, one for each type, and a collection of valid pointer sets, one for each heap. AutoCorres comes with tactics like **wp**, which generates the weakest precondition of a Hoare triple.

2.2 Cogent

Cogent is a restricted, pure, polymorphic, functional language which is designed to be used to implement low-level systems without degrading performance or increasing the cost of verification significantly. The aim of this thesis is to make verification of Cogent programs even easier by taking the first step towards attaining a verified library of commonly used ADTs. This library of ADTs will consists of the data types which cannot be defined natively in Cogent. Before we describe Cogent's current ADT library, we describe how Cogent makes verification easier and why this lead to some data types not being implementable using only Cogent.

Currently, Cogent makes formal verification easier in several ways. Firstly, Cogent has certifying compiler which produces runnable C code, a shallow embedding in Isabelle/HOL, and a proof that the C code is a correct refinement of the Isabelle/HOL shallow embedding. This means that if we prove that the shallow embedding is correct then the C code is also correct so we don't have to directly reason about the C code. Secondly, since Cogent programs are shallowly embedded and Cogent is a purely functional language, and hence has no mutable state, we only need to apply basic mathematical techniques to prove correctness, and we do not encounter the problem of aliasing, since Cogent programs do not have mutable state. Thirdly, Cogent programs compile to C programs, and C programs do not require large language runtime environments typical of other high level languages so there TCB is smaller. This also means that Cogent programs should have similar performance to hand written C programs. Typically, low-level systems are written in C because it is fast and efficient with a small memory footprint. This makes Cogent an excellent choice for certain kinds of low-level systems, such file systems and device drivers. Finally, Cogent uses uniqueness types to handle safe and efficient memory management [29]. There are other languages, such as CakeML Kumar et al. [21], which also make verification easier. However, such languages currently are not suitable for low-level systems or are inferior to Cogent in terms of ease of verification.

Many programs require, some sort of looping behaviour to repeat sets of instructions

until stopping conditions are established. In functional languages, this is achieved through recursion. Currently, Cogent does not support recursion natively in order to make certifying compilation easier. There are projects underway to introduce a limited form of recursion to Cogent, however, even with this, there would still be some recursive function which would be implementable using only Cogent. To solve this issue, Cogent has a FFI to C, so that any recursion or looping behaviour can be defined in C functions and then these can be called from Cogent. These C functions need to be verified just like Cogent programs. However, unlike Cogent functions which would have nice Isabelle/HOL shallowly embeddings with no mutable state, the C functions would generally have embeddings which have less abstraction and mutable state, and thus, overall harder to verify. For low-level system programs, which are part of the domain programs which Cogent is suitable for, the majority of these would only really require iteration over some ADT. Our aim of developing a verified library of commonly used ADTs would rectify this issue of no native recursion.

2.2.1 Frame Constraints

All C functions that can be called from Cogent must respect Cogent's type system. To do this, the C functions must be well-typed and satisfy three frame constraints [28] which are defined as follows:

$$\textbf{Inertia: } p \notin w_i \wedge p \notin w_o \longrightarrow \mu_i(p) = \mu_o(p) \tag{2.9}$$

$$\textbf{Leak Freedom: } p \in w_i \wedge p \notin w_o \longrightarrow \mu_o(p) = \bot \tag{2.10}$$

$$\textbf{Fresh Allocation: } p \notin w_i \wedge p \in w_o \longrightarrow \mu_i(p) = \bot \tag{2.11}$$

where p is a Cogent pointer to a heap object, w_i and w_o are sets of pointers which are writeable, i.e. a set of pointers which we are allowed to modify or free or create, before and after a C function is called, and μ_i and μ_o are partial functions from Cogent pointers to their values, if the Cogent pointer is not valid, i.e. does not point to anything, then the result of applying these partial functions to these invalid pointers is $\bot$.

By enforcing that C functions are well-typed, we ensure that the no internal aliasing occurs in the return value. If a C function satisfies the frame constraints, then we know that the C function does not modify any state other than the writeable values which are passed to it. The **Inertia** constraint ensures that values passed in to the C function, which are not writeable, still retain the same values after the C function has been executed. In addition, no new non-writeable objects are created. The **Leak Freedom** constraint ensures that heap objects are not leaked in the C function. The **Fresh Allocation** constraint ensures that if a heap object is created in the C function, no Cogent pointers were pointing to the address of the new heap object prior to its creation. All C functions must satisfy these constraints so that they do not disrupt Cogent uniqueness type system.

A straightforward approach to ensure that these constraints are satisfied, would be to treat all heap objects as linearly typed objects when writing our C functions. Hofmann [18] outlines an approach to write C in a functional style where all heap objects are linearly typed. This approach would make it easier to show that the frame constraints are satisfied, however, it would be too restrictive and would not take advantage of C's speed and efficiency. Not all C pointers would need to satisfy the frame constraints. Internal C pointers which are not visible from Cogent are allowed to ignore these constraints [27]. Using recursive functions in C, especially in low level systems, is not advisable in general because loops are generally faster and there is a possibility of causing a stack overflow if the stack size is small and the recursion depth is deep with large stack frames. So in addition to proving functional correctness of C functions, we also need to prove that they satisfy the frame constraints.

2.2.2 Cogent's ADT Library

Cogent has an ADT library, which is accessed via its FFI. The ADTs in the library have specifications, however, these have not been formally defined, and hence, their implementations have not been formally verified. The implementations of the ADTs are written using antiquoted C. To use them, they are compiled into C by Cogent's

compiler. An example of an antiquoted C function is the following:

```
\label{verb:anti}
$ty:a $id:wordarray_get($ty:(((WordArray a)!, WordArrayIndex)) args)
{
        if (args.p2 >= (args.p1)->len) {
                return 0;
        }
        return (args.p1)->values[args.p2];
}
```

This function is the **wordarray_get** function from the word array ADT. This function returns the element at a given index if the index is valid. In this antiquoted function, we can see that types and the name of the function have been antiquoted. The reason that Cogent's ADT library is written in antiquoted C is that many of the ADTs which are commonly uses can be polymorphic and defining higher order functions in C would require parts of C which either have not been formalised or are not amenable to current verification tools.

Many ADTs, such as arrays, lists and trees, can be polymorphic. For example, we could have a array of 32-bit words, characters or any user defined type. In addition, most of their implementations in C would be similar with only minor differences due to the different types. So there would be a lot of unnecessary repetition. We could implement it in a single C data type, but this would involve nasty type coercions which would make verification difficult. In addition, it would not feasible to generate every possible instance, since programmers can define their own types. Antiquotation solves this as we can generate all the necessary data types at compilation.

Higher order functions are functions which take other functions as arguments. These higher order functions can also be polymorphic. For example, consider the function

map which is defined as follows, in a high level language similar to Haskell:

$$\begin{aligned} &\mathbf{map} \ :: \ (a \to b) \to [a] \to [b] \\ &\mathbf{map} \ f \ [] \ = \ [] \ | \\ &\mathbf{map} \ f \ (x \ \# \ xs) \ = \ (f \ x) \ \# \ (\mathbf{map} \ f \ xs). \end{aligned} \tag{2.12}$$

This function takes another function f and applies it to each element of a list. This function does not specify the exact type of the list or the function, however, there are constraints on the type of the function. In C, we would need to generate a **map** function for each combination of types a and b. Using antiquoted C, we can just define a single **map** function and only generate the **map** functions for the required types.

To actually write a higher order function in C for a concrete type, there are two approaches. These are taking a function pointer as an argument or taking a function ID as an argument. In the first method the function pointer is called just like a regular function in the body of the higher order function. In the second method, the higher order function calls a dispatch function which takes a function ID and the arguments to be passed to the relevant function. The dispatch function, which is basically a giant switch case statement, compares function IDs until it finds the relevant function. It then calls that function giving it the relevant arguments and then it returns the result to the higher order function. Cogent's ADT library is implemented using the second method because general function pointers are not supported in the verification tools which these ADT functions will be verified with. The dispatch functions are only generated at compile time because the set of functions that could be given to a higher order function is not known until compile time. So the call to the dispatch function in the higher order functions are also antiquoted.

To verify Cogent's ADT library implementations, we first need to compile all the antiquoted C to C, since antiquoted C does not have a formalism. To do this we need to instantiate all the types and instantiate the dispatch functions used in the higher order functions. This means that we need a proof for each type and dispatch function combination. In addition, the names of types and functions can change depending on the program being compiled. So in actual fact, we would need to generate a proof of

correctness each time an ADT is compiled. This technique is called translation validation Pnueli et al. [30]. Translation validation is a widely used technique often used in certifying compilation such as Cogent's compiler. It has also been used to extend the proof of correctness for the seL4 microkernel to the binary level [32] and in the verification tool AutoCorres to prove that the monadic shallow embedding it generates from a C SIMPL embedding is a correct refinement.

For Cogent's ADT library, the translation validation proofs should be almost as simple as replaying proofs. This is because the program logic basically remains the same regardless of the types and names of variables and functions. The only functions where differences would occur are the dispatch functions. Since the dispatch functions are just switch case statements, proving correctness should be straightforward. For the rest, changes to the names and a few minor changes due to the different types would be required. Proving functional correctness for a specific program should be enough to expose all the necessary proof tactics required to develop an automatic tactic to prove correctness, since the automatic tactic would basically replay the proofs with a few minor changes. The development of an automatic tactic is beyond the scope of this thesis but we found that it is important to keep in mind so that we could keep our proofs as generic as possible.

2.3 Existing Verified ADTs

In other programming languages, there exists many verified ADTs. Some examples of these are verified red black trees in Coq [3], verified linked data structures in Java [35], verified Haskell containers [8], and Isabelle Collection Framework (ICF) [22] which generates verified and efficient ADTs for Haskell and ML. Although the ADTs implementations differ, all of these examples use some form of data refinement to prove correctness. We use these examples as inspiration for our own verification task.

2.3.1 Data Refinement

Data refinement is the technique that defines a relation R which connects a concrete data type to an abstract data type. With this, we can then prove that the operations of the concreted data type are correct by showing that if the concrete and abstract data types are related, then both the concrete and abstract data types are still related after the concrete operations and its abstract counterparts are executed, and the results of the operations are equivalent.

As an example, suppose we implement a set with as a list and prove functional correctness of the insert operation. We can define the refinement relation between a list and set as follows:

$$R\ l\ s\ \equiv\ \{x \mid \forall\ i < \mathit{length}\ l.\ x\ =\ l\ !\ i\} = s \tag{2.13}$$

In the refinement relation R, it extracts all elements from a list l into a set and compares it with the set s. We can then use R as an equivalence relation between a list and a set. The functional correctness specification for insert can be defined like the following:

$$\mathit{insert}_s\ s\ x\ =\ s \cup \{x\} \tag{2.14}$$

The implementation for insert can be defined as adding the new element to the front of the list:

$$\mathit{insert}_c\ l\ x\ =\ x\ \#\ l \tag{2.15}$$

To prove function correctness we then need to show the following:

$$R\ l\ s\ \Longrightarrow R\ (\mathit{insert}_c\ l\ x)\ (\mathit{insert}_s\ s\ x) \tag{2.16}$$

We can then prove this is true by unfolding the definitions the refinement relation and the operations.

In this thesis, we take similar approach in our verification approach.

Chapter 3

Methodology

This section outlines the steps that were taken to prove both functional correctness and frame constraint satisfiability for 32-bit word arrays, and how these steps can be extended to other word arrays of standard word sized. The scope of this project was limited to only consider the core word array functions which are not platform specific and can be used to implement the other word array functions. These word array functions are:

- **wordarray_length:** This function takes a word array as input and returns the length of that word array.
- **wordarray_get:** This function takes a word array and an index as input and returns the element at the index if the index is in bounds, or it returns the value 0.
- **wordarray_put2:** This function takes a word array, an index and a value as input and returns the word array the value inserted at the given index. Note that if the index given is not in bounds then the word array is returned unchanged.
- **wordarray_fold_no_break:** This function takes a word array, a starting and ending index, a function, an accumulator and an observer as input and returns the result of applying the function to each element from the starting index up to

but not including the ending index with the accumulator and the observer. Note that this function is the similar to the higher order fold function used on lists except it works on a slice and takes an extra observer argument.

To generate the C implementations of the 32-bit word array functions, a trivial Cogent program which simply calls the word array functions was written and compiled with the Cogent compiler (version 2.9.0.0-c58cbc0e2a). Isabelle/HOL (June 2019 version) and AutoCorres (version 1.6.1) was use to verify correctness of the C implementation of 32-bit word arrays. The machine architecture parameter which AutoCorres requires was set to ARM, i.e. 32-bit machine. The antiquoted word array implementation, trivial Cogent program and the generated C program are provided in section A.1, section A.2 and section A.3 of the appendix.

3.1 Defining Functional Correctness

Word arrays are modelled as Isabelle/HOL's list of words, since both of these are finite sequences of words. In Isabelle/HOL, lists are the base type for finite sequences.

3.1.1 Abstraction Relation

To model a 32-bit word array as a list of 32-bit words, an abstraction relation must be defined, which states how a word array and list of words are equivalent. This is quite straight forward since if both have the same sequence of words and their sequences are of the same length, then they should be equivalent. Since the 32-bit word array data structure contains pointers, checking pointer validity must also be included in the abstraction relation so that equivalence only occurs if the word array is valid. A pointer is valid if the area in memory it is pointing at has actually been allocated. This constraint is also necessary to prevent memory becoming corrupted from the word array functions. Another requirement is that the maximum length of the word array can only be at most a quarter of the maximum word value. This is because the heap

size is bounded by the maximum word value, and each element in a 32-bit word array is 4 bytes in size. Note that if this abstract relation was defined for a 64-bit word array, the maximum length could only be at most an eighth of the maximum word value since a 64-bit word is 8 bytes in size. This constraint ensures that a word array that satisfies this constraint can actually exist. If a word array does not satisfy this constraint then it cannot possibly exist and hence is not valid.

3.1.2 List Counterparts for Word Array Functions

The word array functions, which are considered, all have similar list counterparts in Isabelle/HOL. For **wordarray_length** and **wordarray_put2**, their list counterparts *length* and *list_update*. The behaviours of the counterparts are exactly the same as the intended behaviours of the word array functions, so the functional correctness specification for these word array functions are their list counterparts with no modifications.

The Isabelle/HOL list function *nth* is almost an exact counterpart for **wordarray_get**, however, in the case where the given index is out of range, *nth* raises and exception whereas **wordarray_get** returns 0. Since Cogent programmers using **wordarray_get** would expect to receive the value 0 when **wordarray_get** is applied to an invalid index, the functional correctness specification for **wordarray_get** should incorporate this behaviour. The specification then for **wordarray_get** is simply and an if statement with the condition checking if the index is within range. If the index is within range then the result should be the same as the result from applying *nth*, otherwise the result should be 0.

For **wordarray_fold_no_break**, its counterpart is the list *fold* function, however, there are some slight differences. The first difference is that **wordarray_fold_no_break** can be applied to a slice of a word array by setting the starting and ending indices accordingly, whereas the list *fold* function operates on a whole list. This behaviour can easily be encoded by *fold* to the slice. To obtain the slice, the list functions *take*, which, given a natural number n, returns the first n elements in a list, and *drop*, which given a natural number n, returns the list with the first n elements removed, can be used. For

the *take* and *drop* functions, they take a natural number as an argument. However, the requested slice range can be invalid and which would mean argument passed to *take* and *drop* would appear to be negative at first glance. Since the argument passed *take* and *drop* is a natural number, these negative numbers would become 0, and thus the slice returned would be empty, which is the intended behaviour for invalid slice ranges. In the functional correctness specification, invalid slices are handled explicitly so that it is clear to others what the intended behaviour should be.

Although **wordarray_fold_no_break** is a higher order function, it doesn't actually take a function as an argument. Instead it takes a function ID as an argument and uses a dispatch function to call the relevant function for the reasons as described in subsection 2.2.2. The functional correctness specification for **wordarray_fold_no_break** takes an actual function as an argument. However, a dispatch function, which has the same behaviours as the C dispatch function, is also defined. In the refinement and frame constraint satisfiability proofs, this dispatch function is the function argument to specification for **wordarray_fold_no_break**.

Typically for higher order functions like *fold*, the function that it takes as an argument f only takes two arguments. One of these arguments corresponds to the accumulator value and the other corresponds to the current element which the function f is being applied to by *fold*. For **wordarray_fold_no_break**, the function that it takes as an argument f' also takes an additional observer argument which remains unchanged throughout the operation of **wordarray_fold_no_break**. Since the observer is also an argument of **wordarray_fold_no_break**, the function f' can be transformed into a function like f by wrapping f' in another function f'' which takes the same arguments as f and captures the observer argument in its internals like the following:

$$f''\ x\ acc\ = (\lambda x\ acc.\ f'\ x\ acc\ obsv) \tag{3.1}$$

This is how it is defined in the functional correctness specification.

3.2 Proving Functional Correctness

AutoCorres is used to embed the C implementation of 32-bit word arrays into Isabelle/HOL as a monadic shallow embedding with the mutable state represented as a non-deterministic state monad. Hoare logic is then used to show that the C word array functions are a refinement of the corresponding functional correctness specification. Note that all of these proofs are total correctness proofs, i.e. the word array functions terminate and return the same values as specified by the functional correctness specification. Since most of the functions do not contain loops, the termination proof is trivial. Only the function **wordarray_fold_no_break** contains a loop. However, it is quite easy to prove termination because it has a loop counter which is incremented each iteration, and once this loop counter has exceeds its termination value, the loop terminates.

For the word array functions which do not modify the given word array, the Hoare triple is of the form:

$$\{\lambda s.\ \alpha\ s\ xs\ w\}\ f_C\ args_C\ \{\lambda r\ s'.\ \alpha\ s'\ xs\ w\ \wedge\ r = f_S\ args_S\} \tag{3.2}$$

In the Hoare triple, α is the abstraction relation which takes a state, a list xs and a word array w. The states s and s' are the states before and after the operation f_C. The operation f_C is the word array function, whose specification is f_S. The arguments to f_C are $args_C$, which contains w, and $args_S$, which contains xs, are the equivalent arguments. The meaning of this Hoare triple is that if a word array and list are equivalent then after applying the function f_C, the word array and list are still equivalent and the result of f_C is the same as the result from f_S when applied to equivalent arguments. For word array functions which modify the given word array, the Hoare triple is of the form:

$$\{\lambda s.\ \alpha\ s\ xs\ w\}\ f_C\ args_C\ \{\lambda r\ s'.\ \alpha\ s'\ (f_S\ args_S)\ r\} \tag{3.3}$$

This Hoare triple means that if a list and word array are equivalent, then after applying f_C to modify the word array, the modified word array is equivalent to the list modified by the specification of f_C which is f_S.

3.3 Proving Frame Constraint Satisfiability

To prove frame constraint satisfiability, the frame constraints had to be transformed so that they would be applicable to C pointers. The frame constraints for C pointers were of the form

$$\begin{aligned} \mathit{frame}\ s\ W_i\ s'\ W_o\ I\ \equiv\ & (\forall p.p \in W_i \wedge p \notin W_o \wedge p \notin I \longrightarrow v\ s'\ p) \\ & (\forall p.p \notin W_i \wedge p \in W_o \wedge p \notin I \longrightarrow v\ s\ p) \\ & (\forall p.p \notin W_i \wedge p \notin W_o \wedge p \notin I \\ & \quad \longrightarrow (v\ s\ p\ =\ v\ s'\ p) \wedge v\ s\ p \longrightarrow (h\ s\ p = h\ s'\ p) \end{aligned} \tag{3.4}$$

where s and s' are the C states before and after an abstract function, such as a word array function, is called, W_i is the input set of writeable pointers and W_o is the output set of writeable pointers, I is the set of pointers which are allowed to ignore the frame constraints, v is the function that checks if a pointer is valid, and h is the function that maps pointers to their values. Since only C pointers which are visible from Cogent must satisfy the frame constraints, it was necessary to define an ignore set I. Cogent's heap representation is different to AutoCorres's heap representation. AutoCorres abstracts the C heap into multiple typed heaps, while the heap in Cogent is still represented as a single heap. This meant that the frame constraints had to be replicated for each different heap that exists for a given Cogent program.

Initially, the frame constraints were defined without ignore sets. This meant that all C pointers would need to satisfy the frame constraints. For the word array functions, having an ignore set is not necessary since all of the pointers they manipulate are visible from Cogent. In the general case, this is not true as some ADT implementations may require C pointers which do not satisfy the frame constraints but are not visible to Cogent.

The frame constraint satisfiability proofs is similar to the refinement proofs for functional correctness since Hoare logic is used. The Hoare triple used is of the form:

$$\{\lambda s.\ \alpha\ s\ xs\ w\}\ f_C\ \mathit{args}_C\ \{\lambda r\ s'.\ \mathit{frame}_{\tau_1}\ s\ W_{\tau_1 i}\ s'\ W_{\tau_1 o}\ I_{\tau_1} \wedge \mathit{frame}_{\tau_2}\ s\ W_{\tau_2 i}\ s'\ W_{\tau_2 o}\ I_{\tau_2} \ldots\} \tag{3.5}$$

where $frame_{\tau_i}$ is the set of frame constraints for type τ_i. Note that in order to show frame constraint satisfiability, the frame constraints for each type must hold. For the majority of the typed heaps, the input and output writeable pointer sets would be empty since the word array functions do not manipulate any pointer of that type. This makes proving frame constraints satisfiability for those typed heaps trivial as frame constraint satisfiability then follows from the definition of the frame constraints. For **wordarray_fold_no_break**, the Hoare triple is slight different to cater for the possible functions it can take as arguments.

Since the ignore sets should only contain C pointers which are not visible from Cogent, the ignore sets for the functions **wordarray_length**, **wordarray_get** and **wordarray_put2** are empty. For all of these functions, the only pointers they have access to are the word array and its elements, and these are all visible from Cogent. Note that for **wordarray_get** and **wordarray_length**, the input and output writeable pointer sets are also empty because these are read-only functions. The input and output writeable pointer sets for **wordarray_put2** contains the pointer to the element that is modified. For **wordarray_fold_no_break**, the ignore sets are set to some arbitrary sets. This is because the functions that can be called from **wordarray_fold_no_break** may modify data structures which may have internal C pointers which are not accessible from Cogent. To ensure that the pointers which the word array encompasses still satisfy the frame constraints, the Hoare triple for **wordarray_fold_no_break** also includes in the precondition that the word array and all its elements are not in the ignore sets. The input and output writeable pointer sets are also empty for **wordarray_fold_no_break** because it should not modify the word array or its elements and any pointers. Note that the function which **wordarray_fold_no_break** takes as argument may manipulate pointers, however, **wordarray_fold_no_break** itself does not do any pointer manipulation.

3.4 Extending Our Proof Method

To extend the proof methods to word arrays with standard word sizes, it is sufficient to modify the functional correctness specification and proofs to reflect the change of the word size. For example, to change the specification for 32-bit word arrays so that it becomes a specification for 64-bit word arrays, the types used in the specification would from 32-bit words to 64-bit words. For the proofs, besides changing a few of the types like in the specification and altering the abstraction relation so that the length constraint properly reflects the change in word size, the pointer validity and heap functions, which are generated by AutoCorres, which are used in the abstraction relation and proofs would also need to be changed to the pointer validity and heap functions for the relevant pointers. For example, the function **is_valid_w32** is used to check the validity of pointers to 32-bit words. This validity function would then become **is_valid_w64**, assuming that the pointer it was being applied to became a pointer to 64-bit words.

Since the C implementations for the word array functions are generated each time the compiler is run, each time the compiler is run, the new C implementation would need to be verified again and frame constraint satisfiability would also need to be proved again. However, the compiler generates the same C code each time modulo alpha conversion and minor changes for **wordarray_fold_no_break**. For **wordarray_fold_no_break**, changes can occur to the dispatch function which it calls due to functions being added an removed from the dispatch set. The parts of the proof not relating to the dispatch function would not be affected by these changes, and the proofs about the dispatch functions would only require slight modifications. So the current proofs, with slight modifications to rectify alpha conversion and the changes to the dispatch functions, can be re-applied each time the compiler is run to show functional correctness and frame constraint satisfiability.

Chapter 4

Results and Discussion

We have defined a functional correctness specification for 32-bit word arrays which can be extended to other word arrays with standard word sizes. We have also proven functional correctness and frame constraint satisfiability for the functions **wordarray_length**, **wordarray_get**, **wordarray_put2** and **wordarray_fold_no_break**. All of the specifications and proofs are provided in section A.4 and section A.5 in the appendix. These are also available on GitHub [9].

4.1 Word Array to List of Words Abstraction Relation

As described in subsection 3.1.1, we defined the abstraction relation α as follows:

$$\begin{aligned} \alpha\ s\ xs\ w\ \equiv\ & is\_valid\_WordArray\_u32\_C\ s\ w\ \wedge \\ & (unat\ (\mathbf{w\_l}\ s\ w))\ =\ length\ xs\ \wedge \\ & 4 \times length\ xs\ \leq\ unat\ max\_word\ \wedge \\ & \mathbf{u32list}\ s\ xs\ (\mathbf{w\_p}\ s\ w). \end{aligned} \tag{4.1}$$

This abstraction relation takes a non deterministic state monad s, which represents the state of a C program, a list of 32-bit words and a pointer to a 32-bit word array, and returns if the list and word array are equivalent at the given C state. It does this

by first checking if the pointer to the 32-bit word array is valid by using the function *is_valid_WordArray_u32_C*, which is generated by AutoCorres. It then compares the length of the 32-bit word array with the length of the list. To do this it extracts the length field from the 32-bit word array pointer using the function **w_l**. The lengths of the word array and list are then checked to make sure they have a valid size. The elements of the word array and the elements of the list are then compared using the function **u32list**. The function **w_p** is used to extract the start of the array of 32-bit words. The functions **w_l** and **w_p** are functions which we defined using functions generated by AutoCorres which map pointers to their values, and extract the length and array field from a word array struct. The functions *length* and *unat* are in-built Isabelle/HOL functions for determining the length of a list and for converting words to natural numbers. The term *max_word* is the maximum possible word value. This is different depending on the architecture for the machine. For example, on a 32-bit machine, *max_word* would be equal to $2^{32} - 1$.

The function **u32list** is the specialisation of the function **arrlist** for 32-bit words. The function **arrlist** is defined recursively as:

$$\begin{aligned} &\textbf{arrlist}\ h\ v\ []\ p = \textbf{True}\ | \\ &\textbf{arrlist}\ h\ v\ (x\#xs)\ p = v\ p \wedge h\ p = x \wedge \textbf{arrlist}\ h\ v\ xs\ (p +_p 1). \end{aligned} \tag{4.2}$$

arrlist takes a function h which returns the 32-bit word that a pointer is pointing to, a function v that checks the validity of a pointer, a list and pointer to a 32-bit word. Note that a pointer to a 32-bit word is the same as an array of 32-bit words, so the element with address $p +_p 1$ would be the second element of the array which starts at p. The function **arrlist** iterates over the list and compares each element of the list with each element of the array whilst also checking that the each element of the array is valid. If **arrlist** returns true, we know that the array and list are equivalent assuming that the array and list have the same length, otherwise the array and list are not equivalent. In our abstraction relation, we check that the lengths of the word array and list are equal, so this assumption is valid. Note that it could still be the case that the length of the word array as defined in the word array struct and the actual length are different, however, in the case that the actual length is shorter, **arrlist** would return false, and

in the case where the actual length is longer, the word array functions prevent access to the elements beyond the defined length so it is as if they do not exists.

4.2 Word Array Functional Correctness Specification

As described in subsection 3.1.2, the functional correctness specifications for the word array functions are defined using their list counterparts with **w_length**, **w_get**, **w_put** and **w_fold** being the specifications for **wordarray_length**, **wordarray_get**, **wordarray_put2** and **wordarray_fold_no_break** respectively. In Isabelle/HOL they defined as follows:

$$\textbf{w\_length}\ w = \mathit{length}\ w$$

$$\textbf{w\_get}\ w\ i = (\textbf{if}\ \mathit{length}\ w \leq i\ \textbf{then}\ 0\ \textbf{else}\ w\ !\ i)$$

$$\textbf{w\_put}\ w\ i\ v = w[i := v]$$

$$\textbf{w\_fold}\ w\ frm\ to\ f\ acc\ obsv =$$

$$(\textbf{if}\ frm \leq to\ \textbf{then}\ acc$$

$$\textbf{else}\ \mathit{fold}\ (\lambda x\ y.\ f\ x\ y\ obsv)\ (\mathit{take}\ (to - frm)\ (\mathit{drop}\ frm\ w))\ acc)$$

Note that a program can be proven correct even if it does not do what it is intended to do. This occurs when the specification is incorrect. The intended behaviour for the word array functions that we are verifying is that they should behave the same as their list counterparts, excluding in the case of error handling. Our specification is defined using their list counterparts so we believe our specification is correct with a high degree of certainty.

4.3 Refinement and Frame Constraint Satisfiability Proofs

For all the refinement and frame constraint satisfiability proofs, we applied the weakest precondition tactic **wp** defined by AutoCorres. As the name suggests, the **wp** tactic

generates the weakest precondition which, if true, then implies that the postcondition is true for that given Hoare triple. Once this tactic was applied and loops were annotated with invariants and measures, the proofs followed from the definitions of the abstraction relation α and the frame constraints with addition lemmas derived from these and a few conversions between words, natural numbers and integers.

4.3.1 Addition Lemmas

The majority of the proof effort went into proving addition lemmas used in the refinement and frame constraint satisfiability proofs. All of the addition lemmas which were needed in order to prove functional correctness and frame constraint satisfiability were derived from the abstraction relation α and the frame constraints Some of these were quite trivial to derive such as **abs_length_eq** and **frame_triv_w32**. The lemma **abs_length_eq**, which is defined as:

$$\begin{aligned} &\textbf{abs\_length\_eq:} \\ &\quad \alpha\ s\ xs\ w \Longrightarrow \mathit{unat}\ (\mathbf{w\_l}\ s\ w) = \mathit{length}\ xs, \end{aligned} \tag{4.3}$$

states that if a word array and list are equivalent then their lengths are equal. The lemma **frame_triv_w32**, which is defined as:

$$\begin{aligned} &\textbf{frame\_triv\_w32:} \\ &\quad \mathit{frame\_word32\_ptr}\ s\ W\ s'\ W, \end{aligned} \tag{4.4}$$

states that the frame constraint holds if both the state and writeable pointer sets remain unchanged. Note for each typed heap, similar lemma was defined and all these were combined into the lemma **frame_triv**. There were also a few lemmas which required more effort to prove and these were the key lemmas necessary to prove functional correctness and frame constraint satisfiability.

The majority of the lemmas that required more effort to prove were related to validity of pointers to elements in a word array and the equivalence of between elements in a word array and list. This group of lemmas was the most important because most of the refinement and frame constraint satisfiability proofs were about word array functions

which involved accessing or modifying an element in the word array. The main lemmas were the following:

arrlist_nth_valid:

$$\textbf{arrlist}\ h\ v\ xs\ p \wedge i < \mathit{length}\ xs \Longrightarrow v\ (p +_p i) \tag{4.5}$$

arrlist_nth_value:

$$\textbf{arrlist}\ h\ v\ xs\ p \wedge i < \mathit{length}\ xs \Longrightarrow h\ (p +_p i) = xs\ !\ i \tag{4.6}$$

to_arrlist:

$$(\forall k < \mathit{length}\ xs.\ v\ (p +_p\ \mathit{int}\ k) \wedge h\ (p +_p\ \mathit{int}\ k) \Longrightarrow \textbf{arrlist}\ h\ v\ xs\ p \tag{4.7}$$

arrlist_nth_valid states that if a list xs and array starting at p are found to be equivalent by **arrlist**, then a pointer $p +_p i$ that points to the ith element in the array and i is a valid index, then the pointer is a valid pointer. Before accessing/modifying an element in the word array, it is necessary to show that the location in memory of the element which is about to be accessed/modified is actually valid, so **arrlist_nth_valid** is used to discharge this proof obligation.

arrlist_nth_value states that if a list xs and array starting at p are found to be equivalent by **arrlist**, then the element of the array which is pointed to by the pointer $p +_p i$ is equal to the ith element of the list xs if i is a valid index. In our proofs, showing that the ith element of a word array and the ith element of a list was proof obligation that appeared quite often. **arrlist_nth_value** was used to discharge these proof obligation.

to_arrlist states that if a list xs and array starting at pointer p are equivalent, then **arrlist** will also find that the list xs and the array starting at p to be equivalent. This alternate definition of equivalence and validity was more convenient to use when the word array and list were modified. **to_arrlist** was used to switch between the two definitions.

The lemmas **arrlist_nth_valid**, **arrlist_nth_value** and **to_arrlist** were all proven using structural induction on the list xs. In our refinement and frame constraint satis-

fiability proofs, we never explicitly see the function **arrlist** in our assumptions or in our proof obligations. This is because it is hidden in the definition of our abstraction relation. In our proofs, we could have unfolded the definitions until **arrlist** appeared, however, this would have made our proofs more cluttered, so we decided to abstract these lemmas. To do this, we first combined **arrlist_nth_valid**, **arrlist_nth_value** into a single lemma **arrlist_nth**. We also combined **arrlist_nth_valid**, **arrlist_nth_value** and **to_arrlist** into the lemma **arrlist_all_nth** which states that each element of an array is valid and equal to an element in a list if and only if **arrlist** states that the array is valid and equal to the list, assuming that the array and list have the same length.

The lemmas **arrlist_nth** and **arrlist_all_nth** were then specialised for arrays of 32-bit words and lists of 32-bit words in the lemmas **u32list_nth** and **u32list_all_nth** by replacing **arrlist** with **u32list**. **u32list_nth** was further transformed into the lemma $\boldsymbol{\alpha}$**_nth** so that it uses the abstraction relation α rather than the definition **u32list**. This form made it easier to use because it reduced the number of times we needed to unfold the definition of the abstraction relation α which in turn reduced the size of the assumption set in our proofs. Although we did apply a similar process **u32list_all_nth**, we never used this lemma in our proofs because the abstraction relation α contains more information than **u32list** so it was hard to unify with the assumptions and proof obligation. In the proofs of refinement and frame constraint satisfiability, only $\boldsymbol{\alpha}$**_nth** and **u32list_all_nth** were used as these unified best with the assumptions and proof obligations.

Other lemmas which were quite important were the lemmas that relate to the equality of pointer addresses of elements in an array. These lemmas where necessary because in some proofs, we had to prove the following:

$$p +_p k = p +_p i \longrightarrow v = xs[i := v] \; ! \; k. \tag{4.8}$$

This proof obligation is true if $k = i$, however, from $p +_p k = p +_p i$, we cannot infer that $k = i$ unless additional facts were known. The necessary facts that was required was that either there was no overflow in the pointer arithmetic or that the overflow

occurred in a certain way. Pointer arithmetic differs from real number arithmetic in two ways. Firstly, pointers have a finite size, so any arithmetic operation can cause an overflow or an underflow. Secondly, pointer arithmetic only includes adding a word to a pointer or subtracting a word from a pointer. For pointer addition, the term $p +_p i$ is equivalent to $p + i \times s$, where s is the size of the type that p points to and i is a word. For example, if p points to a 32-bit word, then s would be equal to 4 since a 32-bit word is 4 bytes long. Pointer subtraction is defined similarly. Since pointer addition is defined as $p + i \times s$, overflow can occur in either the addition or the multiplication operation. The only allowable overflow is the overflow in the addition operation.

In the proof obligation 4.8, if the overflow occurs in the multiplication operation and $p +_p i = p +_p j$, then there are cases where $i \neq j$. Using a small example to illustrate this, suppose that pointers are 3-bit words, which means that the word arithmetic is modulo 8. Let the pointer $p = 0$ and let the size of the objects that pointers point to be 2 bytes long. Then the proof obligation 4.8 would simplify to the following:

$$i \times 2 = j \times 2 \longrightarrow i = j. \tag{4.9}$$

If $i = 2$ and $j = 6$, then $2 \times 2 \equiv 4 \ mod\ 8$ and $6 \times 2 \equiv 4 \ mod\ 8$. Clearly $2 \neq 6$ even though $2 \times 2 \equiv 6 \times 2 \ mod\ 8$, so this proof obligation 4.8 would be false. Clearly, we need to have that in pointer arithmetic, no overflow occurs in the multiplication operation.

In the proof obligation 4.8, if the overflow only occurs in the addition operation, then we can infer $k = i$. This is because $p +_p i = p +_p j$ can be rewritten as $p + i \times s = p + j \times s$. From this, we can subtract p from both sides by the cancellation law for addition. Since we know that $i \times s = j \times s$, and there is no overflow in the multiplication, then $i = j$ from the cancellation law for multiplication. Note that the cancellation law for multiplication is only true if there is no overflow in the multiplication. This was one of the lemmas we had to prove. We defined it as follows:

word_mult_cancel_right:

$$\begin{aligned} &a \geq 0 \wedge b \geq 0 \wedge c \geq 0 \wedge uint\ a \times uint\ c \leq uint\ max\_word \wedge \\ &uint\ b \times uint\ c \leq uint\ max\_word \Longrightarrow a \times c = b \times c \longleftrightarrow c = 0 \vee a = b \end{aligned} \tag{4.10}$$

The proof of this followed from the representation of words and from the definition of word arithmetic. We chose to define no overflow by casting words to integers and then comparing the result of the multiplication with the maximum word value. With this definition of no overflow, it made it easier to derive from our abstraction relation, that the pointers in word array do not overflow in the multiplication operation. In our abstraction relation, we had that the lengths of the word array and list were less than or equal to a quarter of the maximum word value. Note that a 32-bit word is 4 bytes long, so any valid index multiplied by 4 would be less than the maximum word size. Therefore, our abstraction relation guarantees that pointers do not overflow in the multiplication of the pointer addition, and hence the word array does not wrap over itself. Initially, our abstraction relation did not put restrictions on the size of the word arrays and lists. It was only after we faced this issue and realised that it was due to pointer overflow that we included length constraint in the abstraction relation. We actually attempted alternate ways to show that no overflow occurs, such as modifying the **arrlist** function to have that $p < p +_p 1$, however the length constraint was the easiest and most successful way that we discovered to define no overflow. The constraint that the lengths of word arrays and lists can only have a maximum length at most a quarter of the maximum word value is actually rather intuitive. Since the size of memory on a machine is at most maximum word value, then for a 32-bit word array which has elements of size 4 bytes, it is impossible for a 32-bit word array, which does not wrap over itself, to have a length more than a quarter of the maximum word value.

To avoid having to unfold the definition of pointer arithmetic in the proofs, we proved the lemma **ordered_indices** which is defined as:

ordered_indices:

$$\begin{aligned} &size\_of\ (\text{TYPE}\ ('a)) > 0\ \wedge\ m \times size\_of\ (\text{TYPE}\ ('a)) \leq unat\ max\_word\ \wedge \\ &n \times size\_of\ (\text{TYPE}\ ('a)) \leq unat\ max\_word\ \wedge\ n < m \\ &\quad \Longrightarrow p +_p int\ n \neq p +_p int\ m \end{aligned} \tag{4.11}$$

where $'a$ is the type that the pointer p points to and $size\_of\,(\text{TYPE}\ ('a))$ returns the size of the type $'a$. The proof of this followed from the definition of pointer arithmetic and

other word facts. We then derived the lemma **distinct_indices** which was almost the same as **ordered_indices** except that the assumption $n < m$ was changed to $n \neq m$. We then defined **wordarray_same_address**, which we proved using **distinct_indices**. This lemma is defined as:

wordarray_same_address:

$$\alpha\ s\ xs\ w \wedge n < \mathit{length}\ xs \wedge m < \mathit{length}\ xs \tag{4.12}$$
$$\Longrightarrow (\mathbf{w\_p}\ s\ w) +_p \mathit{int}\ n = (\mathbf{w\_p}\ s\ w) +_p \mathit{int}\ m \longrightarrow n = m$$

where **w_p** extracts the start of the array from a pointer to a word array. Only **wordarray_same_address** is used directly in the refinement proofs as this lemma was easiest to unify with the assumptions and proof obligations.

4.3.2 wordarray_fold_no_break

The refinement and frame constraint satisfiability proofs for **wordarray_fold_no_break** required the addition of a loop invariants and measures for termination. At each iteration, we needed to show that invariant still holds and that the measure decreases. These are proved in **fold_loop_inv** and **fold_loop_frame_inv**.

The maximum number of iterations that occurs in **wordarray_fold_no_break** is the following:

$$\min(to,\ \mathit{length}\ xs) - frm, \tag{4.13}$$

assuming that $frm \leq \min(to,\ \mathit{length}\ xs)$. When this is not the case, there will be 0 iterations. From this, a simple measure for termination can be defined as follows:

$$\lambda i.\, \min \mathit{unat}\ to,\ \mathit{length}\ xs) - \mathit{unat}\ i, \tag{4.14}$$

where i is the current index that **wordarray_fold_no_break** is operating on. In our proofs we show that this measure decreases at each iteration and terminates once this measure reaches 0.

The loop invariants for the refinement and frame constraint satisfiability proofs both contained that the abstraction relation holds and the current index i is greater than or

equal to the starting index frm. Since **wordarray_fold_no_break** does not modify the word array, the abstraction relation should hold throughout the operation. The current index i should always be greater than or equal to the starting index frm because it is initialised to be equal to frm and it is incremented after each iteration. Also included in the invariants was what the stopping index was equal to which was $\min(to,\ \mathbf{w\_l}\ s\ w)$, where **w_l** extracts the length from a pointer to a word array. This was only included so that the **wp** tactic would not remove this from the assumptions. For the refinement proof, our postcondition was the result from the **wordarray_fold_no_break** is the same as the result from our specification **w_fold**, so we decided to include the following in our invariant:

$$\mathbf{w\_fold}\ xs\ frm\ to\ f\ acc\ obsv = \mathbf{w\_fold}\ xs\ i\ to\ f\ acc_i\ obsv, \tag{4.15}$$

where acc_i is the value of the accumulator when at the index i and it is returned by **wordarray_fold_no_break** when $i = to$ or when it reaches the end of the word array. By adding this to the invariant, we guarantee that by the end of the loop, we have the following:

$$\mathbf{w\_fold}\ xs\ frm\ to\ f\ acc\ obsv = acc_i \tag{4.16}$$

which is what we need to prove show refinement. This is because initially $i = frm$ and $acc_i = acc$. After the loop has finished executing, $i = \min(to,\ length\ xs)$, so **w_fold** $xs\ i\ to\ f\ acc_i\ obsv$ simplifies to acc_i. For the frame constraint satisfiability proof, our postcondition is that the frame constraints hold after applying **wordarray_fold_no_break**. So in our invariant, we have that the frame constraints hold throughout the loop for each change of state, i.e. $frame\ s\ W_i\ s_i\ W_o\ I$ holds at each iteration, where s_i is the state at when the current index is i. From this, it easy to show that the frame constraints hold because if the invariant holds, then s_i becomes the state after **wordarray_fold_no_break** after is executed, so the frame constraints hold after **wordarray_fold_no_break** is executed. To show that the invariant is true, we also need to show that it hold before the loop. Since the input and output writeable pointer sets were the same in all of our proofs, showing that the frame constraints held initially was trivial by applying the lemma **frame_triv**. Once we defined the invariants, the proofs that the invariants were preserved followed from the definitions of the

abstraction relation, specification, frame constraints, and the dispatch function, as well the additional lemmas as described in subsection 4.3.1.

In our frame constraint satisfiability proofs, the writeable pointer sets were empty. In most cases, the writeable pointer sets will be empty because we are applying a fold operation onto an array of words and most likely, the type of the accumulator will also be a word. However, in some cases the accumulator argument may contain pointers. In our current proof, these pointers would be included in the ignore set, however, this means that we do no prove frame constraint satisfiability for these pointers. To solve this, we would need to set the writeable pointer sets to contain the relevant pointers and remove these from the ignore sets in our proof. Slight modification would need to be made to the proof, however, for the most part, the same proof tactics would apply.

In our proofs about invariant preservation and loop termination, we needed to unfold the definitions for the C and Isabelle/HOL dispatch functions and all the functions that can be called by the C dispatch function. In the case that additional C functions are made available or made unavailable to the C dispatch function, our Isabelle/HOL dispatch function equivalent would need to be modified to reflect these changes and the lemmas **fold_loop_inv** and **fold_loop_frame_inv** would need to be modified to also unfold the definitions of the additional C functions and to remove references to the definitions of the C functions which are no longer present in the C dispatch function. In our frame constraint satisfiability proof, the sets of writeable pointers would also need to changed accordingly. The proofs for the frame constraint satisfiability would also need to change slightly however the proofs should follow from the definitions of the frame constraints.

Since the accumulator and the observer argument can be of any type, multiple **wordarray_fold_no_break** would be generated for each of the different types. For each of these, we would need to show refinement and frame constraint satisfiability. However, each of these proofs will be almost exactly the same. The only location that would differ, besides the names of types and function, is when we are proving that the invariant holds and the loop terminates. This difference is only due to the dispatch function

having a different set of available functions. This can be solved in the same way as when additional functions are made available and when functions are made unavailable to the dispatch function as described previously.

4.4 Bug Discovery

Initially, the structure for a word array was defined in C as:

```
struct WordArray_u32 {
    int len;
    u32 *values;
} ;
```

Notice that the length is define as a signed value. From this definition, it is possible to define a word array of negative length by either initially defining a negatively sized word array or by causing an overflow in the length value. In the word array functions that we verified, this was not an issue because the length was always used in expressions which used unsigned values. In C, a signed value will be implicitly cast to an unsigned value if it is used in an expression with an unsigned value. Even though no issues arose, it does not make sense to have the length of a word array represented with a signed value. In addition, in other functions that we did not verify, this may be an issue. So, changing the length representation to an unsigned value removes all potential bugs that could arise from having a negative length value.

4.5 Observations on Frame Constraint Satisfiability

From a Cogent program, it is easy to see which pointers are writeable and which ones are read-only. From C, this is not obvious as all of this type information is lost. In our frame constraint satisfiability proofs, we manually compared the Cogent program to the C program to determine the writeable pointer sets in C. This is possible

to do because we can discern the input writeable pointer set from the types of the arguments and the output writeable set from the type of the return value. For **wordarray_get**, **wordarray_length** and **wordarray_put2** this is sufficient because the writeable pointer sets do not change. However, this approach relies on the proof engineer to correctly define the writeable sets. An automatic tactic to generate the input and output writeable sets would be more suitable. This does not solve the problem of guaranteeing that the correct input and output writeable sets are defined, however, if any errors occurred, they would be systematic and more easily traceable. In addition for **wordarray_fold_no_break**, the writeable pointer sets depend on the function that is passed to it. So an automatic tactic would be very helpful here.

Frame constraints need to be defined for each type used in a program and they need to be added to the postconditions for all the frame constraint satisfiability proofs. In our proofs we do this manually. This was possible because the Cogent program we used was very small. If a larger Cogent program was used, this would have become quite tedious. An automatic tactic would be suitable for this.

The frame constraint satisfiability proof for **wordarray_put2** is actually stronger than what is required. This is because the whole word array is treated as being writeable, however, in our proofs we chose to only have the element that is updated be part of the writeable pointer sets. This choice was made simply because it was easier to define the writeable pointer set this way. Is it easy to prove that if the frame constraint holds for input and output writeable sets which are equal, where the set is equal to some W, then the frame constraint holds for input and output writeable sets which are equal, where the set is equal to W' and $W \subseteq W'$.

Chapter 5

Future Work

Earlier, we mentioned that there are other word array functions which we chose not to verify since they are only variants of the functions which we did verify and they can be implemented, albeit inefficiently, with the functions we verified. These functions are **wordarray_fold**, **wordarray_map_no_break** and **wordarray_map**. Note that **wordarray_map** and **wordarray_map_no_break** are not the same as Isabelle/HOL's list **map** function. In fact, their intended behaviour is similar to Haskell's **mapAccum** function. All of these functions can be implemented using **wordarray_fold_no_break** and **wordarray_put2** so that they return the same results, however, the current implementations for **wordarray_fold**, **wordarray_map_no_break** and **wordarray_map** would perform better. Since our goal is to build verifiable systems code which is efficient and has good performance, verifying these functions would enable us to build more efficient systems which have good performance and yet still verifiable. Currently, we have defined a function correctness specification for these functions in Isabelle/HOL. We have also proved several high level properties about their functional correctness specification. In addition we have verified functional correctness of **wordarray_map_no_break** for a simple Cogent program. The specifications and proofs are provided in A.4 and section A.5 of the appendix.

In our work, we verified the functional correctness and frame constraint satisfiability

for 32-bit word arrays for a specific Cogent program, since the C implementations only exist once a Cogent program is compiled. We want to have functional correctness and frame constraint satisfiability for 32-bit word arrays in all Cogent programs. For all the word array functions besides **wordarray_fold_no_break**, functional correctness can be shown by re-applying our functional correctness proofs and frame constraint satisfiability proofs. Note that there may be some alpha conversions required because the compiler may change the name of the C types, which then means that the names of some functions which are generated by AutoCorres will also have different names. For **wordarray_fold_no_break**, slight changes would need to be made to the proofs as described in subsection 4.3.2. For frame constraint satisfiability, an automatic tactic would be required to generate all the frame constraints for each type and extract the writeable pointer sets from the function types. Slight modifications to the frame constraint satisfiability proofs would need to be made as described in section 4.5. In addition, we would like to extend this result to all word arrays with standard word sizes. This would also require an automatic tactic to do, however, this tactic would only need to alter the current proofs for 32-bit word arrays as described in section 3.4.

Chapter 6

Conclusion

In our proposed work, we have defined a functional correctness specification for 32-bit word arrays, verified functional correctness of several core word array functions, and we have proved that these functions satisfy Cogent's frame constraints. Our specification and proofs can also be easily extended so that they apply to word arrays with standard word sizes and arbitrary Cogent programs.

Our functional correctness specification shallowly embeds all of the intended behaviours of the key word array functions in Isabelle/HOL without mutable state. This makes it easier to use in other functional correctness proofs for Cogent programs. In addition, the intended behaviours of the word array functions are defined using standard list functions, so many of the available lemmas about lists can be used in conjunction with our specification.

Although we proved functional correctness and frame constraint satisfiability for a specific Cogent program, the majority of our functional correctness proofs are designed so that they hold regardless of the Cogent program with minor alpha conversions due to Cogent's compiler which can change the names of types and functions. For the other proofs, these should only require slight modifications. We have investigated where such changes might occur and what changes would need to be made.

Our work has set the foundations for future work in verifying functional correctness and frame constraint satisfiability for other C data types. Specifically, our work on frame constraint satisfiability would apply to all C data types, and our functional correctness specification and proofs can be used as inspiration for other C data types which have similar intended behaviours, such as standard arrays. Our work is the first step in our quest for attaining a fully verified library of ADTs which cannot be implemented in Cogent natively. A fully verified library of ADTs would greatly reduce the cost of formal verification of Cogent programs and all the ADTs they rely on.

Bibliography

[1] Sidney Amani. *A Methodology for Trustworthy File Systems.* PhD thesis, UNSW, 2016. URL `https://ts.data61.csiro.au/publications/nicta{_}full{_}text/9502.pdf`.

[2] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. COGENT: Verifying High-Assurance File System Implementations. In *ASPLOS '16Proceedings Twenty-First Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, volume 61, pages 175–188, 2016. ISBN 9781450340915. doi: 10.1145/2872362.2872404.

[3] Andrew W Appel. Efficient Verified Red-Black Trees. *Library (Lond).*, pages 1–15, 2011.

[4] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. Position paper: the science of deep specification. *Philos. Trans. R. Soc. A Math. Phys. Eng. Sci.*, 375(2104), 2017. ISSN 1364503X. doi: 10.1098/rsta.2016.0331.

[5] Thomas Ball, Ella Bounimova, R. Kumar, and Vladimir Levin. SLAM2: Static driver verification with under 4% false alarms. *Form. Methods Comput. Aided Des.*, pages 35 –42, 2010. URL `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp={&}arnumber=5770931`.

[6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science An EATCS Series. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-642-05880-6. doi: 10.1007/978-3-662-07964-5. URL `http://link.springer.com/10.1007/978-3-662-07964-5`.

[7] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, pages 454–464, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-44585-2.

[8] Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. Ready, Set, Verify! Applying Hs-to-coq to

Real-world Haskell Code (Experience Report). *Proc. ACM Program. Lang.*, 2 (ICFP):89:1—-89:16, jul 2018. ISSN 2475-1421. doi: 10.1145/3236784. URL http://doi.acm.org/10.1145/3236784.

[9] Cheunglo. Cheunglo/verifed_word_arrays: Verified word arrays, January 2020. URL https://doi.org/10.5281/zenodo.3594554.

[10] David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, volume 5170 LNCS, pages 167–182, 2008. ISBN 3540710655. doi: 10.1007/978-3-540-71067-7-16. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.436.4858{&}rep=rep1{&}type=pdf.

[11] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Thomas Moskal Michałand Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving High. Order Logics*, pages 23–42, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03359-9.

[12] Robert W. Floyd. *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993. ISBN 978-94-011-1793-7. doi: 10.1007/978-94-011-1793-7_4. URL https://doi.org/10.1007/978-94-011-1793-7_4.

[13] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1979. ISBN 978-3-540-09724-2. doi: 10.1007/3-540-09724-4. URL http://link.springer.com/10.1007/3-540-09724-4.

[14] David Greenaway. *Automated proof-producing abstraction of C code*. phdthesis, University of New South Wales, 2014. URL https://ts.data61.csiro.au/publications/nicta{_}full{_}text/8758.pdfhttp://handle.unsw.edu.au/1959.4/54260.

[15] David Greenaway, June Andronick, and Gerwin Klein. Bridging the gap: Automatic verified abstraction of C. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 7406 LNCS:99–115, 2012. ISSN 03029743. doi: 10.1007/978-3-642-32347-8_8.

[16] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain. *SIGPLAN Not.*, 49(6):429–439, jun 2014. ISSN 0362-1340. doi: 10.1145/2666356.2594296. URL http://doi.acm.org/10.1145/2666356.2594296.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259.

[18] Martin Hofmann. A Type System for Bounded Space and Functional In-Place Update–Extended Abstract. In *Proc. 9th Eur. Symp. Program. Lang. Syst.*, ESOP '00, pages 165–179, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67262-1. doi: 10.1007/3-540-46425-5_11. URL `http://dl.acm.org/citation.cfm?id=645394.651913`.

[19] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1): 27–69, February 2009.

[20] Gerwin Klein, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, and Rafal Kolanski. seL4: Formal Verification of an OS Kernel. In *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Princ. - SOSP '09*, page 207, New York, New York, USA, 2009. ACM Press. ISBN 9781605587523. doi: 10.1145/1629575.1629596. URL `http://portal.acm.org/citation.cfm?doid=1629575.1629596`.

[21] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Principles of Programming Languages (POPL)*, pages 179–191. ACM Press, January 2014. doi: 10.1145/2535838.2535841. URL `https://cakeml.org/popl14.pdf`.

[22] Peter Lammich and Andreas Lochbihler. The isabelle collections framework. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 339–354, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14052-5.

[23] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4): 363–446, 2009. ISSN 01687433. doi: 10.1007/s10817-009-9155-4. URL `https://xavierleroy.org/publi/compcert-backend.pdf`.

[24] Magnus O. Myreen. Functional programs: Conversions between deep and shallow embeddings. In *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, volume 7406 LNCS, pages 412–417. Springer, Berlin, Heidelberg, 2012. ISBN 9783642323461. doi: 10.1007/978-3-642-32347-8_29. URL `http://link.springer.com/10.1007/978-3-642-32347-8{_}29`.

[25] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-43376-7. doi: 10.1007/3-540-45949-9. URL `http://link.springer.com/10.1007/3-540-45949-9`.

[26] Michael Norrish. C formalised in hol. Technical report, University of Cambridge, Computer Laboratory, 1998.

[27] Liam O'Connor. *Type Systems for Systems Types*. phdthesis, University of New South Wales, 2019. URL `http://handle.unsw.edu.au/1959.4/64238`.

[28] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: bringing down the cost of verification. *ACM SIGPLAN Not.*, 2016. doi: 10.1145/3022670.2951940.

[29] Liam O'Connor-Davis, Gabriele Keller, Sidney Amani, Toby Murray, Gerwin Klein, Zilin Chen, and Christine Rizkallah. CDSL Version 1 Simplifying Verification with Linear Types. Technical report, NICTA, Sydney, Australia, oct 2014.

[30] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation Validation. In *Proc. 4th Int. Conf. Tools Algorithms Constr. Anal. Syst.*, number Ep 20897 in TACAS '98, pages 151–166, Berlin, Heidelberg, 1998. Springer-Verlag. ISBN 3-540-64356-7. doi: 10.1007/bfb0054170. URL `http://dl.acm.org/citation.cfm?id=646482.691453`.

[31] Norbert Schirmer. *Verification of sequential imperative programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.128.388{&}rep=rep1{&}type=pdfhttp://www-wjp.cs.uni-saarland.de/leute/private{_}homepages/nschirmer/pub/schirmer{_}phd.pdf`.

[32] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. *ACM SIGPLAN Not.*, 48(6):471, 2013. ISSN 03621340. doi: 10.1145/2499370.2462183.

[33] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, Bytes, and Separation Logic. In Martin Hofmann and Matthias Felleisen, editor, *ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, pages 97–108, Nice, France, jan 2007. ACM. ISBN 1-59593-575-4.

[34] Martin Wildmoser and Tobias Nipkow. Certifying Machine Code Safety: Shallow Versus Deep Embedding. In Konrad Slind, , Annette Bunker, , and Ganesh Gopalakrishnan, editors, *Theorem Proving High. Order Logics*, pages 305–320. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-30142-4. doi: 10.1007/978-3-540-30142-4_22. URL `http://isabelle.in.tum.de/{~}nipkow/pubs/tphols04.pdfhttp://link.springer.com/10.1007/978-3-540-30142-4{_}22`.

[35] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. *ACM SIGPLAN Not.*, 43(6):349, 2008. ISSN 03621340. doi: 10.1145/1379022.1375624.

Appendix

A.1 Antiquoted C Implementation for Word Arrays

This section contains the antiquoted C implementation for Word Arrays.

```
/*
 * Copyright 2016, NICTA
 *
 * This software may be distributed and modified according to
    the terms of
 * the GNU General Public License version 2. Note that NO
    WARRANTY is provided.
 * See "LICENSE_GPLv2.txt" for details.
 *
 * @TAG(NICTA_GPL)
 */

struct $id:(WordArray a) {
        unsigned int len;
        $ty:a* values;
};

typedef struct $id:(WordArray a) $id:(WordArray a);

/*
 * Copyright 2018, Data61
 * Commonwealth Scientific and Industrial Research
    Organisation (CSIRO)
 * ABN 41 687 119 230.
 *
 * This software may be distributed and modified according to
    the terms of
 * the GNU General Public License version 2. Note that NO
    WARRANTY is provided.
 * See "LICENSE_GPLv2.txt" for details.
 *
```

```
 * @TAG(DATA61_GPL)
 */

$ty:a $id:wordarray_get($ty:(((WordArray a)!, WordArrayIndex))
    args)
{
        if (args.p2 >= (args.p1)->len) {
                return 0;
        }
        return (args.p1)->values[args.p2];
}


u32 $id:wordarray_length($ty:((WordArray a)!) array)
{
        return array->len;
}


$ty:((WordArray a, acc)) $id:wordarray_map_no_break($ty:(
   WordArrayMapNoBreakP a acc obsv) args)
{
        $ty:((WordArray a, acc)) ret;
        $ty:(ElemAO a acc obsv) fargs;
        $ty:((a, acc)) fret;
        u32 to = args.to > args.arr->len ? args.arr->len :
           args.to;
        u32 i;

        fargs.acc = args.acc;
        fargs.obsv = args.obsv;
        for (i = args.frm; i < to; i++) {
                fargs.elem = args.arr->values[i];
                fret = (($spec:(WordArrayMapNoBreakF a acc
                   obsv)) args.f)(fargs);
                args.arr->values[i] = fret.p1;
                fargs.acc = fret.p2;
        }
        ret.p1 = args.arr;
        ret.p2 = fargs.acc;
        return ret;
}

$ty:(WordArrayMapRE a acc rbrk) $id:wordarray_map($ty:(
   WordArrayMapP a acc obsv rbrk) args)
{
```

```
        $ty:(<Iterate () | Break rbrk>) default_variant = { .
           tag = TAG_ENUM_Iterate};
        $ty:((WordArray a, acc)) init_ret = {.p1 = args.arr, .
           p2 = args.acc };
        $ty:(WordArrayMapRE a acc rbrk) ret = {.p1 = init_ret,
            .p2 = default_variant };

        /* setup in case we don't ever loop at all */
        ret.p2.tag = TAG_ENUM_Iterate;

        $ty:(ElemAO a acc obsv) fargs = { .obsv = args.obsv };
        u32 i;

        for (i = args.frm; i < args.to && i < args.arr->len; i
           ++) {
                fargs.elem = args.arr->values[i];
                fargs.acc = ret.p1.p2;

                $ty:(LRR (a,acc) rbrk) fret = (($spec:(
                   WordArrayMapF a acc obsv rbrk)) args.f)(
                   fargs);
                args.arr->values[i] = fret.p1.p1; // <T>

                ret.p1.p2 = fret.p1.p2; // acc
                ret.p2 = fret.p2;

                if (fret.p2.tag == TAG_ENUM_Break) {
                        break;
                }
        }

        return ret;
}

$ty:(LoopResult acc rbrk) $id:wordarray_fold($ty:(
   WordArrayFoldP a acc obsv rbrk) args)
{
        $ty:(ElemAO a acc obsv) fargs;
        $ty:(LoopResult acc rbrk) fret;
        u32 to = args.to > args.arr->len ? args.arr->len :
           args.to;
        u32 i;

        fargs.obsv = args.obsv;
        fargs.acc = args.acc;
        fret.tag = TAG_ENUM_Iterate;
```

```
        fret.Iterate = args.acc;
        for (i = args.frm; i < to; i++) {
                fargs.elem = args.arr->values[i];
                fret = (($spec:(WordArrayFoldF a acc obsv rbrk)
                   ) args.f) (fargs);

                if (fret.tag == TAG_ENUM_Break)
                        return fret;
                fargs.acc = fret.Iterate;
        }
        return fret;
}

$ty:(acc) $id:wordarray_fold_no_break($ty:(
   WordArrayFoldNoBreakP a acc obsv) args)
{
        $ty:(ElemAO a acc obsv) fargs;
        u32 to = args.to > args.arr->len ? args.arr->len :
           args.to;
        u32 i;

        fargs.obsv = args.obsv;
        fargs.acc = args.acc;

        for (i = args.frm; i < to; i++) {
                fargs.elem = args.arr->values[i];
                fargs.acc = (($spec:(WordArrayFoldNoBreakF a
                   acc obsv)) args.f) (fargs);
        }

        return fargs.acc;
}


$ty:(WordArray a) $id:wordarray_put2($ty:(WordArrayPutP a)
   args)
{
        if (likely(args.idx < (args.arr)->len)) {
                args.arr->values[args.idx] = args.val;
        }

        return args.arr;
}
```

A.2 Trivial Cogent Program

This section contains our trivial Cogent program which causes the Cogent compiler to produce the C implementation for 32-bit word arrays.

```
include <gum/common/wordarray.cogent>

-- Standard ADT methods which are not higher order functions

wordarray_get_u32 : ((WordArray U32)!, WordArrayIndex) -> U32
wordarray_get_u32 x = wordarray_get x

wordarray_length_u32 : (WordArray U32)! -> U32
wordarray_length_u32 x = wordarray_length [U32] x

wordarray_put2_u32 : WordArrayPutP U32 -> WordArray U32
wordarray_put2_u32 x = wordarray_put2 x

---- ADT methods which are higher order functions
--
wordarray_map_no_break_u32 : all(a,b). WordArrayMapNoBreakP
   U32 a b -> (WordArray U32, a)
wordarray_map_no_break_u32 x = wordarray_map_no_break x

wordarray_fold_no_break_u32 : all(a). WordArrayFoldNoBreakP
   U32 a () -> a
wordarray_fold_no_break_u32 x = wordarray_fold_no_break x

---- Function bodies used to modify elements or in higher
   order functions
--
inc : ElemAO U32 () () -> (U32, ())
inc #{elem, acc, obsv} = (elem + 1, acc)

sum_bod : ElemAO U32 U32 () -> U32
sum_bod #{elem, acc, obsv} = elem + acc

mul_bod : ElemAO U32 U32 () -> U32
mul_bod #{elem, acc, obsv} = elem * acc
---- Tests to make higher order functions exist
--
---- For map_no_break
--
inc_arr : WordArray U32 -> (WordArray U32, ())
inc_arr wa = let end = wordarray_length wa !wa and
```

```
                arg = #{arr = wa, frm = 0, to = end, f = inc,
                    acc = (), obsv = ()}
            in wordarray_map_no_break_u32 arg
---- For fold_no_break

sum : (WordArray U32)! -> U32
sum wa = let e = wordarray_length_u32 wa and
             arg = #{arr = wa, frm = 0, to = e, f = sum_bod,
                acc = 0, obsv = ()}
         in wordarray_fold_no_break arg
mul : (WordArray U32)! -> U32
mul wa = let e = wordarray_length_u32 wa and
             arg = #{arr = wa, frm = 0, to = e, f = mul_bod,
                acc = 0, obsv = ()}
         in wordarray_fold_no_break arg
```

A.3 Generated C Program From the Trivial Cogent Program

This section contains the generated C program from our trivial Cogent program. The C implementations for the 32-bit word arrays are:

- **wordarray_length_0**
- **wordarray_get_0**
- **wordarray_put2_0**
- **wordarray_fold_no_break_0**
- **wordarray_map_no_break_0**

```
/*
This file is generated by Cogent

*/

typedef void *SysState;
typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned int u32;
typedef unsigned long long u64;
typedef struct unit_t {
            int dummy;
```

```
        } unit_t;
typedef struct bool_t {
            u8 boolean;
        } bool_t;
enum {
    LET_TRUE = 1,
};
enum {
    LETBANG_TRUE = 1,
};
enum untyped_func_enum {
    FUN_ENUM_inc,
    FUN_ENUM_inc_arr,
    FUN_ENUM_mul,
    FUN_ENUM_mul_bod,
    FUN_ENUM_sum,
    FUN_ENUM_sum_bod,
    FUN_ENUM_wordarray_fold_no_break_0,
    FUN_ENUM_wordarray_get_0,
    FUN_ENUM_wordarray_get_u32,
    FUN_ENUM_wordarray_length_0,
    FUN_ENUM_wordarray_length_u32,
    FUN_ENUM_wordarray_map_no_break_0,
    FUN_ENUM_wordarray_map_no_break_u32_0,
    FUN_ENUM_wordarray_put2_0,
    FUN_ENUM_wordarray_put2_u32,
};
typedef enum untyped_func_enum untyped_func_enum;
typedef untyped_func_enum t11;
typedef untyped_func_enum t12;
typedef untyped_func_enum t13;
typedef untyped_func_enum t14;
typedef untyped_func_enum t4;
typedef untyped_func_enum t15;
typedef untyped_func_enum t8;
typedef untyped_func_enum t16;
struct WordArray_u32 {
    unsigned int len;
    u32 *values;
};
typedef struct WordArray_u32 WordArray_u32;
struct t1 {
    WordArray_u32 *p1;
    u32 p2;
};
typedef struct t1 t1;
```

```
struct t2 {
    WordArray_u32 *arr;
    u32 idx;
    u32 val;
};
typedef struct t2 t2;
struct t3 {
    u32 acc;
    u32 elem;
    unit_t obsv;
};
typedef struct t3 t3;
struct t5 {
    u32 acc;
    WordArray_u32 *arr;
    t4 f;
    u32 frm;
    unit_t obsv;
    u32 to;
};
typedef struct t5 t5;
struct t6 {
    unit_t acc;
    u32 elem;
    unit_t obsv;
};
typedef struct t6 t6;
struct t7 {
    u32 p1;
    unit_t p2;
};
typedef struct t7 t7;
struct t9 {
    unit_t acc;
    WordArray_u32 *arr;
    t8 f;
    u32 frm;
    unit_t obsv;
    u32 to;
};
typedef struct t9 t9;
struct t10 {
    WordArray_u32 *p1;
    unit_t p2;
};
typedef struct t10 t10;
```

```
static inline u32 wordarray_get_0(t1);
static inline u32 wordarray_length_0(WordArray_u32 *);
static inline WordArray_u32 *wordarray_put2_0(t2);
static inline u32 wordarray_fold_no_break_0(t5);
static inline t10 wordarray_map_no_break_0(t9);
static inline __attribute__((pure)) u32 wordarray_get_u32(t1);
static inline __attribute__((pure)) u32 wordarray_length_u32(
   WordArray_u32 *);
static inline WordArray_u32 *wordarray_put2_u32(t2);
static inline t10 wordarray_map_no_break_u32_0(t9);
static inline __attribute__((const)) t7 inc(t6);
static inline t10 inc_arr(WordArray_u32 *);
static inline __attribute__((const)) u32 mul_bod(t3);
static inline __attribute__((pure)) u32 mul(WordArray_u32 *);
static inline __attribute__((const)) u32 sum_bod(t3);
static inline __attribute__((pure)) u32 sum(WordArray_u32 *);
static inline t10 dispatch_t11(untyped_func_enum a2,
   WordArray_u32 *a3)
{
    return inc_arr(a3);
}
static inline u32 dispatch_t12(untyped_func_enum a2,
   WordArray_u32 *a3)
{
    switch (a2) {

      case FUN_ENUM_mul:
        return mul(a3);

      case FUN_ENUM_sum:
        return sum(a3);

      case FUN_ENUM_wordarray_length_0:
        return wordarray_length_0(a3);

      default:
        return wordarray_length_u32(a3);
    }
}
static inline u32 dispatch_t13(untyped_func_enum a2, t1 a3)
{
    switch (a2) {

      case FUN_ENUM_wordarray_get_0:
        return wordarray_get_0(a3);
```

```
      default:
        return wordarray_get_u32(a3);
    }
}
static inline WordArray_u32 *dispatch_t14(untyped_func_enum a2
   , t2 a3)
{
    switch (a2) {

      case FUN_ENUM_wordarray_put2_0:
        return wordarray_put2_0(a3);

      default:
        return wordarray_put2_u32(a3);
    }
}
static inline u32 dispatch_t4(untyped_func_enum a2, t3 a3)
{
    switch (a2) {

      case FUN_ENUM_mul_bod:
        return mul_bod(a3);

      default:
        return sum_bod(a3);
    }
}
static inline u32 dispatch_t15(untyped_func_enum a2, t5 a3)
{
    return wordarray_fold_no_break_0(a3);
}
static inline t7 dispatch_t8(untyped_func_enum a2, t6 a3)
{
    return inc(a3);
}
static inline t10 dispatch_t16(untyped_func_enum a2, t9 a3)
{
    switch (a2) {

      case FUN_ENUM_wordarray_map_no_break_0:
        return wordarray_map_no_break_0(a3);

      default:
        return wordarray_map_no_break_u32_0(a3);
    }
}
```

```
typedef u32 ErrCode;
typedef u32 WordArrayIndex;
typedef t6 inc_arg;
typedef WordArray_u32 *inc_arr_arg;
typedef t10 inc_arr_ret;
typedef t7 inc_ret;
typedef WordArray_u32 *mul_arg;
typedef t3 mul_bod_arg;
typedef u32 mul_bod_ret;
typedef u32 mul_ret;
typedef WordArray_u32 *sum_arg;
typedef t3 sum_bod_arg;
typedef u32 sum_bod_ret;
typedef u32 sum_ret;
typedef t5 wordarray_fold_no_break_0_arg;
typedef u32 wordarray_fold_no_break_0_ret;
typedef t1 wordarray_get_0_arg;
typedef u32 wordarray_get_0_ret;
typedef t1 wordarray_get_u32_arg;
typedef u32 wordarray_get_u32_ret;
typedef WordArray_u32 *wordarray_length_0_arg;
typedef u32 wordarray_length_0_ret;
typedef WordArray_u32 *wordarray_length_u32_arg;
typedef u32 wordarray_length_u32_ret;
typedef t9 wordarray_map_no_break_0_arg;
typedef t10 wordarray_map_no_break_0_ret;
typedef t9 wordarray_map_no_break_u32_0_arg;
typedef t10 wordarray_map_no_break_u32_0_ret;
typedef t2 wordarray_put2_0_arg;
typedef WordArray_u32 *wordarray_put2_0_ret;
typedef t2 wordarray_put2_u32_arg;
typedef WordArray_u32 *wordarray_put2_u32_ret;
static inline __attribute__((pure)) u32 wordarray_get_u32(t1
   a1)
{
    t1 r2 = a1;
    u32 r3 = wordarray_get_0(r2);

    return r3;
}
static inline __attribute__((pure)) u32 wordarray_length_u32(
   WordArray_u32 *a1)
{
    WordArray_u32 *r2 = a1;
    u32 r3 = wordarray_length_0(r2);
```

```
    return r3;
}
static inline WordArray_u32 *wordarray_put2_u32(t2 a1)
{
    t2 r2 = a1;
    WordArray_u32 *r3 = wordarray_put2_0(r2);

    return r3;
}
static inline t10 wordarray_map_no_break_u32_0(t9 a1)
{
    t9 r2 = a1;
    t10 r3 = wordarray_map_no_break_0(r2);

    return r3;
}
static inline __attribute__((const)) t7 inc(t6 a1)
{
    u32 r2 = a1.elem;
    unit_t r3 = a1.acc;
    unit_t r4 = a1.obsv;
    u32 r5 = 1U;
    u32 r6 = r2 + r5;
    t7 r7 = (t7) {.p1 = r6, .p2 = r3};

    return r7;
}
static inline t10 inc_arr(WordArray_u32 *a1)
{
    WordArray_u32 *r2 = a1;
    u32 r3;

    if (LETBANG_TRUE)
        r3 = wordarray_length_0(r2);
    else
        ;

    u32 r4 = 0U;
    t8 r5 = FUN_ENUM_inc;
    unit_t r6 = (unit_t) {.dummy = 0};
    unit_t r7 = (unit_t) {.dummy = 0};
    t9 r8 = (t9) {.arr = r2, .frm = r4, .to = r3, .f = r5, .
       acc = r6, .obsv =
                  r7};
    t10 r9 = wordarray_map_no_break_u32_0(r8);
```

```
    return r9;
}
static inline __attribute__((const)) u32 mul_bod(t3 a1)
{
    u32 r2 = a1.elem;
    u32 r3 = a1.acc;
    unit_t r4 = a1.obsv;
    u32 r5 = r2 * r3;

    return r5;
}
static inline __attribute__((pure)) u32 mul(WordArray_u32 *a1)
{
    WordArray_u32 *r2 = a1;
    u32 r3 = wordarray_length_u32(r2);
    u32 r4 = 0U;
    t4 r5 = FUN_ENUM_mul_bod;
    u32 r6 = 0U;
    unit_t r7 = (unit_t) {.dummy = 0};
    t5 r8 = (t5) {.arr = r2, .frm = r4, .to = r3, .f = r5, .
       acc = r6, .obsv =
                  r7};
    u32 r9 = wordarray_fold_no_break_0(r8);

    return r9;
}
static inline __attribute__((const)) u32 sum_bod(t3 a1)
{
    u32 r2 = a1.elem;
    u32 r3 = a1.acc;
    unit_t r4 = a1.obsv;
    u32 r5 = r2 + r3;

    return r5;
}
static inline __attribute__((pure)) u32 sum(WordArray_u32 *a1)
{
    WordArray_u32 *r2 = a1;
    u32 r3 = wordarray_length_u32(r2);
    u32 r4 = 0U;
    t4 r5 = FUN_ENUM_sum_bod;
    u32 r6 = 0U;
    unit_t r7 = (unit_t) {.dummy = 0};
    t5 r8 = (t5) {.arr = r2, .frm = r4, .to = r3, .f = r5, .
       acc = r6, .obsv =
                  r7};
```

```
    u32 r9 = wordarray_fold_no_break_0(r8);

    return r9;
}
u16 u8_to_u16(u8 x)
{
    return (u16) x;
}
u32 u8_to_u32(u8 x)
{
    return (u32) x;
}
u64 u8_to_u64(u8 x)
{
    return (u64) x;
}
u8 u16_to_u8(u16 x)
{
    return (u8) x;
}
u32 u16_to_u32(u16 x)
{
    return (u32) x;
}
u8 u32_to_u8(u32 x)
{
    return (u8) x;
}
u16 u32_to_u16(u32 x)
{
    return (u16) x;
}
u64 u32_to_u64(u32 x)
{
    return (u64) x;
}
u32 u64_to_u32(u64 x)
{
    return (u32) x;
}
u8 u64_to_u8(u64 x)
{
    return (u8) x;
}
u16 u64_to_u16(u64 x)
{
```

```
    return (u16) x;
}
u32 wordarray_get_0(t1 args)
{
    if (args.p2 >= args.p1->len)
        return 0;
    return args.p1->values[args.p2];
}
u32 wordarray_length_0(WordArray_u32 *array)
{
    return array->len;
}
t10 wordarray_map_no_break_0(t9 args)
{
    t10 ret;
    t6 fargs;
    t7 fret;
    u32 to = args.to > args.arr->len ? args.arr->len : args.to
       ;
    u32 i;

    fargs.acc = args.acc;
    fargs.obsv = args.obsv;
    for (i = args.frm; i < to; i++) {
        fargs.elem = args.arr->values[i];
        fret = dispatch_t8(args.f, fargs);
        args.arr->values[i] = fret.p1;
        fargs.acc = fret.p2;
    }
    ret.p1 = args.arr;
    ret.p2 = fargs.acc;
    return ret;
}
u32 wordarray_fold_no_break_0(t5 args)
{
    t3 fargs;
    u32 to = args.to > args.arr->len ? args.arr->len : args.to
       ;
    u32 i;

    fargs.obsv = args.obsv;
    fargs.acc = args.acc;
    for (i = args.frm; i < to; i++) {
        fargs.elem = args.arr->values[i];
        fargs.acc = dispatch_t4(args.f, fargs);
    }
```

```
    return fargs.acc;
}
WordArray_u32 *wordarray_put2_0(t2 args)
{
    if (__builtin_expect(!!(args.idx < args.arr->len), 1))
        args.arr->values[args.idx] = args.val;
    return args.arr;
}
```

A.4 32-bit Word Array Functional Correctness Specification

This section contains the functional correctness specifications for 32-bit word arrays. The word array functions that are specified are the following:

- **wordarray_length**
- **wordarray_get**
- **wordarray_put2**
- **wordarray_fold_no_break**
- **wordarray_fold**
- **wordarray_map_no_break**
- **wordarray_map**

There are validation proofs for the following:

- **wordarray_length**
- **wordarray_get**
- **wordarray_put2**
- **wordarray_fold_no_break**
- **wordarray_map_no_break**

In addition, we proved several high level properties for **wordarray_fold_no_break** and **wordarray_map_no_break**.

theory *WordArraySpec*
imports *HOL−Word.Word Word-Lib.HOL-Lemmas*
begin

type-synonym *u32 = 32 word*

datatype *('a, 'b) LoopResult = Iterate 'a | Break 'b*

1 List helper lemmas

lemma *take-1-drop*:
i < length xs ⟹ take (Suc 0) (drop i xs) = [xs ! i]
apply *(induct xs arbitrary: i)*
apply *(simp add: take-Suc-conv-app-nth)*
by *(simp add: drop-Suc-nth)*

lemma *take-drop-Suc*:
i < l ∧ i < length xs ⟹
take (l − i) (drop i xs) = (xs ! i) # take (l − Suc i) (drop (Suc i) xs)
apply *clarsimp*
by *(metis Cons-nth-drop-Suc Suc-diff-Suc take-Suc-Cons)*

2 wordarray_length specification

definition *w-length :: u32 list ⇒ nat*
where
w-length w = length w

2.1 Correctness

lemma *w-length-length-eq*:
w-length xs = length xs
by *(simp add: w-length-def)*

3 wordarray_get specification

definition *w-get :: u32 list ⇒ nat ⇒ u32*
where
w-get w i = (if length w ≤ i then 0 else w ! i)

3.1 Correctness

lemma *w-get-get-eq*:
i < length xs ⟹ w-get xs i = xs ! i
by *(simp add: w-get-def)*

4 wordarray_put2 specification

definition *w-put* :: *u32 list* ⇒ *nat* ⇒
u32 ⇒ *u32 list*
where
w-put w i v = *w*[*i* := *v*]

4.1 Correctness

lemma *w-put-listupdate-eq*:
w-put xs i v= *xs*[*i* := *v*]
by (*simp add*: *w-put-def*)

5 wordarray_fold_no_break specification

definition *w-fold* :: *u32 list* ⇒ *nat* ⇒ *nat* ⇒ (*u32* ⇒ *'a* ⇒ *'b* ⇒ *'a*) ⇒
'a ⇒ *'b* ⇒ *'a*
where
w-fold w frm to f acc obs =
(*if to* ≤ *frm*
then acc
else
fold (λ*x y*. *f x y obs*)
(*take* (*to* − *frm*) (*drop frm w*)) *acc*)

5.1 w_fold helper lemmas

lemma *w-fold-step*: ⟦*frm* < *length xs*; *frm* < *to*⟧ ⟹
w-fold xs frm to f acc obsv = *w-fold xs* (*Suc frm*) *to f* (*f* (*xs* ! *frm*) *acc obsv*)
obsv
by (*simp add*: *w-fold-def take-drop-Suc*)

lemma *w-fold-preserve*:
⟦*frm* ≤ *i*; *i* < *min* (*length xs*) *to*; *w-fold xs frm to f acc obsv* = *w-fold xs i to f*
nacc obsv⟧ ⟹
w-fold xs frm to f acc obsv = *w-fold xs* (*Suc i*) *to f* (*f* (*xs* ! *i*) *nacc obsv*) *obsv*
apply (*simp add*: *w-fold-def*)
apply *safe*
apply (*simp add*: *take-drop-Suc*)
apply (*simp add*: *take-drop-Suc*)
done

lemma *w-fold-end*:
⟦*frm* ≤ *i*; *i* ≥ *min* (*length xs*) *to*; *w-fold xs frm to f acc obsv* = *w-fold xs i to f*
nacc obsv⟧ ⟹
w-fold xs frm to f acc obsv = *nacc*
apply (*simp add*: *w-fold-def*)
done

5.2 w_fold test

primrec *summer* :: *u32* ⇒ *u32* ⇒ *unit* ⇒ *u32* **where**
summer elem acc () = *elem* + *acc*

value *w-fold* [*0*, *1*, *2*, *3*, *4*, *5*] *1 1 summer 0* ()

5.3 Correctness

lemma *w-fold-fold-eq-whole*:
w-fold xs 0 (*length xs*) *f acc obsv* = *fold* (λ*a b*. *f a b obsv*) *xs acc*
apply (*simp add*: *w-fold-def*)
done

lemma *w-fold-fold-eq-slice*:
w-fold xs frm to f acc obsv = *fold* (λ*a b*. *f a b obsv*) (*take* (*to* − *frm*) (*drop frm xs*)) *acc*
apply (*simp add*: *w-fold-def*)
done

6 wordarray_fold specification

definition *w-fold-break* :: *u32 list* ⇒ *nat* ⇒ *nat* ⇒
(*u32* ⇒ *'a* ⇒ *'b* ⇒ (*'a*, *'c*) *LoopResult*) ⇒
'a ⇒ *'b* ⇒ (*'a*, *'c*) *LoopResult*
where
w-fold-break w frm to f acc obs =
(*if to* ≤ *frm*
then (*Iterate acc*)
else
fold (λ*x y*. (*case y of*
Iterate acc ⇒ *f x acc obs* |
Break acc ⇒ *Break acc*))
(*take* (*to* − *frm*) (*drop frm w*)) (*Iterate acc*))

7 wordarray_map_no_break specification

definition *w-map* :: *u32 list* ⇒ *nat* ⇒ *nat* ⇒ (*u32* ⇒ *'a* ⇒ *'b* ⇒ *u32* × *'a*) ⇒
'a ⇒ *'b* ⇒ *u32 list* × *'a*
where
w-map w frm to f acc obs =
(*if to* ≤ *frm*
then (*w*, *acc*)
else
let (*ys*, *vacc*) =
fold (λ*x* (*xs*, *vacc'*).
let (*y*, *vacc''*) = *f x vacc' obs*
in (*xs* @ [*y*], *vacc''*))
(*take* (*to* − *frm*) (*drop frm w*)) ([], *acc*)

in ((*take frm w*) @ *ys* @ (*drop to w*), *vacc*))

7.1 w_map helper lemmas

lemma *w-map-bounds*:
to ≥ *length xs* ⟹ *w-map xs frm to f acc obsv* = *w-map xs frm* (*length xs*) *f acc obsv*
apply (*simp add*: *w-map-def*)
done
lemma *w-map-internal*:
length (*fst* (*fold* (λ*x p*. (*fst p* @ [*fst* (*f x* (*snd p*) *obsv*)], *snd* (*f x* (*snd p*) *obsv*))) *xs* (*ls*, *acc*)))
= *length ls* + *length xs*
apply (*induct rule*: *rev-induct*)
apply *simp*
apply (*simp add*: *split-def*)
done

lemma *w-map-length*:
length (*fst* (*w-map xs frm to f acc obsv*)) = *length xs*
apply (*induct rule*: *rev-induct*)
apply (*simp add*: *w-map-def*)
apply (*clarsimp simp add*: *w-map-def*)
apply (*case-tac frm* > *length xs*)
apply *simp*
apply (*case-tac frm* = *length xs*) **thm** *split-def*
apply (*simp add*: *split-def*)
apply *simp*
apply (*case-tac to* < *length xs*)
apply *simp*
apply (*simp add*: *split-def*)
apply (*case-tac to* = *length xs*)
apply *simp*
apply (*simp add*: *split-def*)
apply *simp*
apply (*simp add*: *split-def*)
done

lemma *w-map-length-same*:
length xs = *length ys* ⟹ *length* (*fst* (*w-map xs frm to f acc obsv*))
= *length* (*fst* (*w-map ys frm*′ *to*′ *f*′ *acc*′ *obsv*′))
apply (*induct xs arbitrary*: *ys rule*: *rev-induct*)
apply *simp*
apply (*simp add*: *w-map-def*)
by (*simp add*: *w-map-length*)

lemma *w-map-length-Cons*:
length (*fst* (*w-map* (*x* # *xs*) *frm to f acc obsv*))
= *Suc* (*length* (*fst* (*w-map xs frm*′ *to*′ *f*′ *acc*′ *obsv*′)))

apply (*induct xs*)
apply (*subst w-map-length*)
apply (*subst w-map-length*)
apply *simp*
apply (*subst w-map-length*)
apply (*subst w-map-length*)
apply *simp*
done

lemma *w-map-length-append*:
length (*fst* (*w-map* (*xs* @ [*x*]) *frm to f acc obsv*))
= *Suc* (*length* (*fst* (*w-map xs frm' to' f' acc' obsv'*)))
apply (*induct xs*)
apply (*subst w-map-length*)
apply (*subst w-map-length*)
apply *simp*
apply (*subst w-map-length*)
apply (*subst w-map-length*)
apply *simp*
done

lemma *w-map-append-list*:
to ≤ *length xs* ⟹
fst (*w-map* (*xs* @ *ys*) *frm to f acc obsv*) = *fst* (*w-map xs frm to f acc obsv*) @ *ys*
apply (*simp add*: *w-map-def*)
apply *clarsimp*
apply (*simp add*: *split-def*)
done

lemma *w-map-append-acc*:
to ≤ *length xs* ⟹
snd (*w-map* (*xs* @ *ys*) *frm to f acc obsv*) = *snd* (*w-map xs frm to f acc obsv*)
apply (*simp add*: *w-map-def*)
apply (*simp add*: *split-def*)
done

lemmas *w-map-append* = *w-map-append-list w-map-append-acc*

lemma *w-map-step-list*:
frm ≤ *to* ⟹ *fst* (*w-map xs frm* (*Suc to*) *f acc obsv*) =
(*fst* (*w-map xs frm to f acc obsv*))[*to* := *fst* (*f* (*xs* ! *to*) (*snd* (*w-map xs frm to f acc obsv*)) *obsv*)]
apply (*induct xs rule*: *rev-induct*)
apply (*simp add*: *w-map-def*)
apply (*case-tac to* ≥ *length xs*)
apply (*case-tac to* ≥ *length* (*xs* @ [*x*]))
apply (*subst* (*asm*) *w-map-length*[**where** *frm* = *frm* **and** *to* = *to* **and** *f* = *f* **and** *acc* = *acc* **and** *obsv* = *obsv*,*symmetric*])

apply *simp*
apply (*subgoal-tac length* (*fst* (*w-map* (*xs* @ [*x*]) *frm to f acc obsv*)) = *Suc* (*length xs*))
apply *simp*
apply (*subst w-map-def*)
apply (*subst w-map-def*)
apply *simp*
apply (*simp add*: *w-map-length*)
apply (*subgoal-tac to* = *length xs*)
defer
apply *simp*
defer
apply *simp*
apply (*simp* (*no-asm*) *add*: *w-map-def*)
apply (*rule conjI*)
apply *clarsimp*
apply (*simp add*: *split-def*)
apply *clarsimp*
apply (*simp add*: *split-def*)
apply (*subst upd-conv-take-nth-drop*)
apply *simp*
apply (*subst w-map-internal*)
apply *simp*
apply *simp*
apply (*subgoal-tac length* (*fst* (*fold* (λ*x p*. (*fst p* @ [*fst* (*f x* (*snd p*) *obsv*)], *snd* (*f x* (*snd p*) *obsv*))) (*drop frm xs*) ([], *acc*))) = *length xs* − *frm*)
apply *simp*
apply (*subst w-map-internal*)
apply *simp*
apply *simp*
apply (*subgoal-tac snd* (*w-map* (*xs* @ [*x*]) *frm to f acc obsv*) = *snd* (*w-map xs frm to f acc obsv*))
apply *simp*
apply (*subgoal-tac* (*xs* @ [*x*]) ! *to* = *xs* ! *to*)
apply *simp*
apply (*subst w-map-append*)
apply *linarith*
apply (*subst w-map-append*)
apply *simp*
apply *clarsimp*
apply (*subst list-update-append1*)
apply (*simp add*: *w-map-length*)
apply *simp*
apply (*simp add*: *nth-append*)
apply (*simp add*: *w-map-append*)
done

lemma *w-map-step-acc*:
⟦*frm* ≤ *to*; *Suc to* ≤ *length xs*⟧

$\Longrightarrow$ *snd* (*w-map xs frm* (*Suc to*) *f acc obsv*)
snd (*f* (*xs* ! *to*) (*snd* (*w-map xs frm to f acc obsv*)) *obsv*)
apply (*induct xs rule*: *rev-induct*)
apply (*clarsimp simp add*: *w-map-def*)
apply *clarsimp*
apply (*case-tac Suc to* $\leq$ *length xs*)
apply (*simp add*: *w-map-append*)
apply (*simp add*: *nth-append*)
apply (*simp add*: *nth-append*)
apply (*simp add*: *w-map-def*)
apply (*simp add*: *split-def*)
done

lemmas *w-map-step* = *w-map-step-list w-map-step-acc*

lemma *w-map-endpoint*:
to < *length xs* $\Longrightarrow$ *fst* (*w-map xs frm to f acc obsv*) ! *to* = *xs* ! *to*
apply (*induct xs rule*: *rev-induct*)
apply (*simp add*: *w-map-def*)
apply *clarsimp*
apply (*simp add*: *w-map-append*)
apply (*case-tac to* = *length xs*)
apply *simp*
apply (*metis w-map-length nth-append-length*)
apply (*subgoal-tac to* < *length xs*)
apply *simp*
apply (*simp add*: *w-map-length nth-append*)
apply *arith*
done

7.2 Correctness

lemma *no-update-acc*:
⟦∀ *x ob a*. *snd* (*f x a ob*) = *a*; *xs* ≠ []⟧
$\Longrightarrow$ *snd* (*fold* (λ*x p*. (*fst p* @ [*fst* (*f x* (*snd p*) *obsv*)], *snd p*)) *xs* ([], *acc*)) = *acc*
apply (*induct xs rule*: *rev-induct*)
apply *simp*
apply *simp*
apply (*case-tac xs* = [])
apply *simp*
apply *simp*
done

lemma *w-map-map-eq-whole*:
⟦∀ *x ob a*. *snd* (*f x a ob*) = *a* ⟧
$\Longrightarrow$ *fst* (*w-map xs 0* (*length xs*) *f acc obsv*) = *map* (λ*x*. *fst* (*f x acc obsv*)) *xs*
apply (*induct xs rule*: *rev-induct*)
apply (*simp add*: *w-map-def*)
apply *simp*

apply (*subst w-map-step*)
apply *simp*
apply (*subst w-map-append*)
apply *simp*
apply (*subst list-update-append*)
apply (*subst w-map-length*)
apply *simp*
apply (*subst w-map-append*)
apply *simp*
apply (*simp* (*no-asm*) *add*: *w-map-def*)
apply (*simp add*: *split-def*)
apply *clarsimp*
apply (*subgoal-tac snd* (*fold* (λ*x p*. (*fst p* @ [*fst* (*f x* (*snd p*) *obsv*)], *snd p*)) *xs* ([], *acc*)) = *acc*)
apply *simp*
apply (*rule no-update-acc*)
apply *simp*
apply *simp*
done

8 wordarray_map specification

definition *w-map-break* :: *u32 list* ⇒ *nat* ⇒ *nat* ⇒
(*u32* ⇒ *'a* ⇒ *'b* ⇒ (*u32* × *'a*, *'c*) *LoopResult*) ⇒
'a ⇒ *'b* ⇒ (*u32 list* × *'a*, *'c*) *LoopResult*
where
w-map-break w frm to f acc obs =
(*if to* ≤ *frm*
then Iterate (*w*, *acc*)
else
let ret = *fold*
(λ*x r*.
case r of
Iterate (*xs*, *vacc*) ⇒
let r' = *f x vacc obs*
in
(*case r' of*
Iterate (*x'*, *vacc'*) ⇒ *Iterate* (*xs* @ [*x'*], *vacc'*) |
Break v ⇒ *Break v*) |
Break v ⇒ *Break v*)
(*take* (*to* − *frm*) (*drop frm w*)) (*Iterate* ([], *acc*))
in
case ret of
Iterate (*ys*, *acc'*) ⇒ *Iterate* ((*take frm w*) @ *ys* @ (*drop to w*), *acc'*) |
Break v ⇒ *Break v*)
end

A.5 32-bit Word Array Refinement and Frame Constraint Satisfiability Proofs

This section contains the refinement proofs and frame constraint satisfiability proofs for 32-bit word arrays. We have refinement proofs for the following word array functions:

- **wordarray_length**
- **wordarray_get**
- **wordarray_put2**
- **wordarray_fold_no_break**
- **wordarray_map_no_break**

We have frame constraint satisfiability proofs for the following:

- **wordarray_length**
- **wordarray_get**
- **wordarray_put2**
- **wordarray_fold_no_break**

In addition, we proved several key lemmas about our abstraction relation, frame constraints, and pointers.

theory *WordArrayT*
imports *AutoCorres.AutoCorres WordArraySpec*
begin

install-C-file *main-pp-inferred.c*
autocorres [*ts-rules* = *nondet*] *main-pp-inferred.c*

context *main-pp-inferred* **begin**

type-synonym *u32* = *32 word*

1 Frame Constraints

definition *frame-word32-ptr* :: *lifted-globals* ⇒ (*word32 ptr*) *set* ⇒ *lifted-globals* ⇒ (*word32 ptr*) *set* ⇒ (*word32 ptr*) *set* ⇒ *bool*
where
frame-word32-ptr s p_i *s'* p_o *ign* ≡ ∀ *p* :: *word32 ptr*.
(*p* ∈ p_i ∧ *p* ∉ p_o ∧ *p* ∉ *ign* ⟶ ¬ *is-valid-w32 s' p*) — leak freedom
∧ (*p* ∉ p_i ∧ *p* ∈ p_o ∧ *p* ∉ *ign*⟶ ¬ *is-valid-w32 s p*) — fresh allocation
∧ (*p* ∉ p_i ∧ *p* ∉ p_o ∧ *p* ∉ *ign* ⟶
(*is-valid-w32 s p* = *is-valid-w32 s' p*) ∧ *is-valid-w32 s p* ⟶
heap-w32 s p = *heap-w32 s' p*) — inertia

definition *frame-WordArray-u32-C-ptr* :: *lifted-globals* ⇒ (*WordArray-u32-C ptr*) *set* ⇒ *lifted-globals* ⇒ (*WordArray-u32-C ptr*) *set* ⇒ (*WordArray-u32-C ptr*) *set* ⇒ *bool*
where
frame-WordArray-u32-C-ptr s p_i *s'* p_o *ign* ≡ ∀ *p* :: *WordArray-u32-C ptr*.
(*p* ∈ p_i ∧ *p* ∉ p_o ∧ *p* ∉ *ign* ⟶ ¬ *is-valid-WordArray-u32-C s' p*) — leak freedom
∧ (*p* ∉ p_i ∧ *p* ∈ p_o ∧ *p* ∉ *ign* ⟶ ¬ *is-valid-WordArray-u32-C s p*) — fresh allocation
∧ (*p* ∉ p_i ∧ *p* ∉ p_o ∧ *p* ∉ *ign* ⟶
(*is-valid-WordArray-u32-C s p* = *is-valid-WordArray-u32-C s' p*) ∧
(*is-valid-WordArray-u32-C s p* ⟶
heap-WordArray-u32-C s p = *heap-WordArray-u32-C s' p*)) — inertia

lemmas *frame-def* = *frame-WordArray-u32-C-ptr-def frame-word32-ptr-def*

lemma *frame-triv-w32-ptr*: *frame-WordArray-u32-C-ptr state P state P ign*
using *frame-WordArray-u32-C-ptr-def* **by** *blast*

lemma *frame-triv-w32*: *frame-word32-ptr state P state P ign*
using *frame-word32-ptr-def* **by** *blast*

lemmas *frame-triv = frame-triv-w32-ptr frame-triv-w32*

lemma *frame-weaken-w32*:
⟦ $A \subseteq A'$; *frame-word32-ptr s A s' A' I*⟧ ⟹ *frame-word32-ptr s A' s' A' I*
apply (*clarsimp simp add*: *frame-word32-ptr-def*)
by *blast*

lemma *frame-combine-w32*:
⟦*frame-word32-ptr s A s' A' I_A*; *frame-word32-ptr s B s' B' I_B*;
$I_A \cap A = \{\}$; $I_A \cap A' = \{\}$; $I_B \cap B = \{\}$; $I_B \cap B' = \{\}$; $I = (I_A \cup I_B) - (A \cup A' \cup B \cup B')$⟧
⟹ *frame-word32-ptr s* $(A \cup B)$ *s'* $(A' \cup B')$ *I*
apply (*clarsimp simp add*: *frame-word32-ptr-def*)
apply (*rule conjI*)
apply *clarsimp*
apply (*erule-tac x = p* **in** *allE*)+
apply *clarsimp*
apply (*erule disjE*; *clarsimp*; *blast*)
apply (*rule conjI*)
apply *clarsimp*
apply (*erule-tac x = p* **in** *allE*)+
apply *clarsimp*
apply (*erule disjE*; *clarsimp*; *blast*)
apply *clarsimp*
done

lemma *frame-weaken-w32-ptr*:
⟦ $A \subseteq A'$; *frame-WordArray-u32-C-ptr s A s' A I*⟧ ⟹ *frame-WordArray-u32-C-ptr s A' s' A' I*
apply (*clarsimp simp add*: *frame-WordArray-u32-C-ptr-def*)
by *blast*

lemma *frame-combine-w32-ptr*:
⟦*frame-WordArray-u32-C-ptr s A s' A' I_A*; *frame-WordArray-u32-C-ptr s B s' B' I_B*;
$I_A \cap A = \{\}$; $I_A \cap A' = \{\}$; $I_B \cap B = \{\}$; $I_B \cap B' = \{\}$; $I = (I_A \cup I_B) - (A \cup A' \cup B \cup B')$⟧
⟹ *frame-WordArray-u32-C-ptr s* $(A \cup B)$ *s'* $(A' \cup B')$ *I*
apply (*clarsimp simp add*: *frame-WordArray-u32-C-ptr-def*)
apply (*rule conjI*)
apply *clarsimp*
apply (*erule-tac x = p* **in** *allE*)+
apply *clarsimp*
apply (*erule disjE*; *clarsimp*; *blast*)
apply (*rule conjI*)
apply *clarsimp*
apply (*erule-tac x = p* **in** *allE*)+

apply *clarsimp*
apply (*erule disjE*; *clarsimp*; *blast*)
apply *clarsimp*
done

lemmas *frame-weaken* = *frame-weaken-w32 frame-weaken-w32-ptr*

lemmas *frame-combine* = *frame-combine-w32 frame-combine-w32-ptr*

2 Helper Lemmas

2.1 Miscellaneous Pointer Lemmas

lemma *cptr-add-add*[*simp*]:
$p +_p m +_p n = p +_p (m + n)$
by (*cases p*) (*simp add*: *algebra-simps CTypesDefs.ptr-add-def*)

2.2 Helper Word Lemmas

lemma *word-mult-cancel-right*:
fixes *a b c* :: (*'a*::*len*) *word*
assumes $0 \leq a$ $0 \leq b$ $0 \leq c$
assumes *uint a* $*$ *uint c* $\leq$ *uint* (*max-word* :: (*'a*::*len*) *word*)
assumes *uint b* $*$ *uint c* $\leq$ *uint* (*max-word* :: (*'a*::*len*) *word*)
shows $a * c = b * c \longleftrightarrow c = 0 \vee a = b$
apply (*rule iffI*)
using *assms*
apply (*unfold word-mult-def word-of-int-def*)
apply (*clarsimp simp*:*Abs-word-inject max-word-def uint-word-of-int m1mod2k uint-0-iff*)
apply *fastforce*
done

lemmas *words-mult-cancel-right* =
word-mult-cancel-right[**where** *'a=8*]
word-mult-cancel-right[**where** *'a=16*]
word-mult-cancel-right[**where** *'a=32*]
word-mult-cancel-right[**where** *'a=64*]

lemma *order-indices*:
fixes *ptr* :: (*'a*::*c-type*) *ptr*
assumes $n *$ *size-of* (*TYPE*(*'a*)) $\leq$ *unat* (*max-word* :: *32 word*)
assumes $m *$ *size-of* (*TYPE*(*'a*)) $\leq$ *unat* (*max-word* :: *32 word*)
assumes $n < m$
assumes *size-of* (*TYPE*(*'a*)) > 0
shows *ptr* $+_p$ *int n* $\neq$ *ptr* $+_p$ *int m*
apply (*simp add*: *ptr-add-def*)
apply (*rule notI*)
using *assms*

apply (*subst* (*asm*) *word-mult-cancel-right*; *simp add*: *uint-nat*)
apply (*subst le-unat-uoi*[**where** *z* = *max-word*:: *32 word*])
apply (*metis le-def multi-lessD nat-less-le neq0-conv*)
apply (*subst le-unat-uoi*[**where** *z* = *max-word*:: *32 word*])
apply (*metis le-def multi-lessD nat-less-le neq0-conv*)
apply (*metis assms*(*1*) *int-ops*(*7*) *of-nat-mono*)
apply (*subst le-unat-uoi*[**where** *z* = *max-word*:: *32 word*])
apply (*metis le-def multi-lessD nat-less-le neq0-conv*)
apply (*subst le-unat-uoi*[**where** *z* = *max-word*:: *32 word*])
apply (*metis le-def multi-lessD nat-less-le neq0-conv*)
apply (*metis int-ops*(*7*) *of-nat-mono*)
apply (*erule disjE*)
using *le-unat-uoi* **apply** *fastforce*
by (*metis* (*mono-tags*, *hide-lams*) *le-unat-uoi less-irrefl-nat mult-cancel2 of-nat-mult*)

lemma *distinct-indices*:
fixes *ptr* :: (*'a*::*c-type*) *ptr*
assumes *n* ∗ *size-of* (*TYPE*(*'a*)) ≤ *unat* (*max-word* :: *32 word*)
assumes *m* ∗ *size-of* (*TYPE*(*'a*)) ≤ *unat* (*max-word* :: *32 word*)
assumes *n* ≠ *m*
assumes *size-of* (*TYPE*(*'a*)) > *0*
shows *ptr* +$_p$ *int n* ≠ *ptr* +$_p$ *int m*
using *assms*
apply (*case-tac n* < *m*)
apply (*erule order-indices*; *clarsimp*)
by (*metis linorder-neqE-nat order-indices*)

2.3 arrlist

fun *arrlist* :: (*'a* :: *c-type ptr* ⇒ *'b*) ⇒ (*'a* :: *c-type ptr* ⇒ *bool*) ⇒ *'b list* ⇒ *'a ptr* ⇒ *bool*
where
arrlist h v [] *p* = *True* |
arrlist h v (*x* # *xs*) *p* = (*v p* ∧ *h p* = *x* ∧ *arrlist h v xs* (*p* +$_p$ *1*))

lemma *arrlist-nth-value*:
assumes *arrlist h v xs p i* < *length xs*
shows *h* (*p* +$_p$ *i*) = *xs* ! *i*
using *assms*
proof (*induct i arbitrary*: *p xs*)
case *0*
then show *?case*
apply(*cases xs*)
by(*auto*)
next
case (*Suc i*)
then have *i* < *length* (*tl xs*) **by** (*cases xs*) *auto*
have *h* (*p* +$_p$ *1* +$_p$ *int i*) = *tl xs* ! *i*
apply(*rule Suc*)

```
    apply(case-tac [!] xs)
   using Suc by auto
 then show ?case using Suc by (auto simp: nth-tl)
qed


lemma arrlist-nth-valid:
 assumes arrlist h v xs p  i < length xs
 shows v (p +p i)
 using assms
proof (induct i arbitrary: p xs)
 case 0 then show ?case by (cases xs) auto
next
 case (Suc i)
 then have i < length (tl xs) by (cases xs) auto
 have v ((p +p 1) +p int i)
  apply (rule Suc.hyps[where xs=tl xs])
   apply (case-tac [!] xs)
  using Suc by auto
 then show ?case by simp
qed

lemmas arrlist-nth = arrlist-nth-value arrlist-nth-valid

lemma case-Suc:
 ∀ i < Suc n.  P i ⟹ (∀ i < n . P (Suc i)) ∧ P 0
 by simp

lemma to-arrlist:
 assumes val-heap: ∀ k <  length xs. v (p +p int k) ∧ h (p +p int k) = xs ! k
 shows arrlist h v xs p
 using assms
 apply (induct xs arbitrary: p)
  apply simp
 apply simp
 apply (rule conjI)
  apply (erule-tac x = 0 in allE)
  apply clarsimp
 apply (rule conjI)
  apply (erule-tac x = 0 in allE)
  apply clarsimp
 apply (drule-tac x = p +p 1 in meta-spec, simp)
 apply (case-tac xs, simp)
 apply (drule case-Suc, simp)
 done


lemma arrlist-all-nth:
 arrlist h v xs p ⟷
```

$(\forall k < \mathit{length}\ \mathit{xs}.\ v\ (p +_p \mathit{int}\ k) \wedge h\ (p +_p \mathit{int}\ k) = \mathit{xs}\ !\ k)$
apply (*intro iffI conjI allI impI*)
apply (*clarsimp simp*: *arrlist-nth-valid*)
apply (*clarsimp simp*: *arrlist-nth-value*)
apply (*erule to-arrlist*)
done

3 Word Array Abstraction

definition *u32list* :: *lifted-globals* ⇒ *u32 list* ⇒ *u32 ptr* ⇒ *bool*
where
u32list s xs p ≡ *arrlist* (*heap-w32 s*) (*is-valid-w32 s*) *xs p*

lemmas *u32list-nth* =
arrlist-nth
[**where** *h*=*heap-w32 s* **and** *v*=*is-valid-w32* (*s* :: *lifted-globals*) **for** *s*,
simplified u32list-def[*symmetric*]]

lemmas *u32list-all-nth* =
arrlist-all-nth
[**where** *h*=*heap-w32 s* **and** *v*=*is-valid-w32* (*s* :: *lifted-globals*) **for** *s*,
simplified u32list-def[*symmetric*]]

abbreviation *w-val h w* ≡ *values-C* (*h w*)
abbreviation *w-len h w* ≡ *len-C* (*h w*)

abbreviation *w-p s w* ≡ *w-val* (*heap-WordArray-u32-C s*) *w*
abbreviation *w-l s w* ≡ *w-len* (*heap-WordArray-u32-C s*) *w*

definition α :: *lifted-globals* ⇒ *u32 list* ⇒ *WordArray-u32-C ptr* ⇒ *bool*
where
α *s xs w* ≡
u32list s xs (*w-p s w*)
∧ (*unat* (*w-l s w*)) = *length xs*
∧ *is-valid-WordArray-u32-C s w*
∧ *4* ∗ *length xs* ≤ *unat* (*max-word*:: *32 word*)

lemma *abs-length-eq*:
α *s xs w* ⟹ *unat* (*w-len* (*heap-WordArray-u32-C s*) *w*) = *length xs*
by (*clarsimp simp add*: α-*def*)

lemma α-*nth*:
fixes *i* :: *int*
assumes
α *s xs w*
0 ≤ *i i* < *length xs*
i = *int n*

shows *heap-w32 s ((w-val (heap-WordArray-u32-C s) w) +$_p$ i) = xs ! n*
and *is-valid-w32 s ((w-val (heap-WordArray-u32-C s) w) +$_p$ i)*
using *u32list-nth assms*
unfolding *α-def*
by *auto*

lemma *α-all-nth*:
α s xs w ⟷ (unat (w-l s w)) = length xs
∧ is-valid-WordArray-u32-C s w
*∧ 4 * length xs ≤ unat (max-word:: 32 word)*
∧ (∀ k<length xs. is-valid-w32 s ((w-p s w) +$_p$ int k) ∧ heap-w32 s ((w-p s w) +$_p$ int k) = xs ! k)
apply (*rule iffI*)
apply (*clarsimp simp*: *α-def u32list-all-nth*)
apply (*clarsimp simp*: *α-def u32list-all-nth*)
done

4 Refinement and Frame Proofs

4.1 wordarray_get

lemma *wordarray-get-refinement*:
{|λs. α s xs w |}
wordarray-get-0' (t1-C w i)
{|λr s. α s xs w ∧ r = w-get xs (unat i) |}!
apply (*unfold wordarray-get-0'-def*)
apply *clarsimp*
apply *wp*
apply (*clarsimp simp*: *w-get-def*)
apply (*safe*; *clarsimp simp*: *uint-nat word-le-nat-alt α-nth abs-length-eq*)
apply (*simp-all add*: *α-def*)
done

lemma *wordarray-get-frame*:
{|λs. state = s ∧ α s xs w |}
wordarray-get-0' (t1-C w i)
{|λr s. frame-WordArray-u32-C-ptr state {} s {} {} ∧
frame-word32-ptr state {} s {} {}|}!
apply(*unfold wordarray-get-0'-def*)
apply(*clarsimp*)
apply(*wp*)
apply (*safe*; *clarsimp simp*: *uint-nat word-le-nat-alt frame-triv α-nth abs-length-eq*)
apply (*simp add*: *α-def*)
done

4.2 wordarray_length

lemma *wordarray-length-refinement*:
{|λs. α s xs w|}

wordarray-length-0′ w
{|λ*r s*. *α s xs w* ∧ (*unat r*) = *w-length xs*|}!
apply(*unfold wordarray-length-0′-def*)
apply(*unfold w-length-def*)
apply(*clarsimp*)
apply(*wp*)
by (*clarsimp simp*: *α-def*)

lemma *wordarray-length-frame*:
{|λ*s*. *state* = *s* ∧ *α s xs w* |}
wordarray-length-0′ w
{|λ*r s*. *frame-WordArray-u32-C-ptr state* {} *s* {} {} ∧
frame-word32-ptr state {} *s* {} {}|}!
apply(*unfold wordarray-length-0′-def*)
apply(*clarsimp*)
apply(*wp*)
apply (*clarsimp simp*: *frame-triv α-def*)
done

4.3 wordarray_put2

lemma *wordarray-same-address*:
⟦*α s xs w*; *n* < *length xs*; *m* < *length xs*⟧ ⟹
((*w-val* (*heap-WordArray-u32-C s*) *w*) +$_p$ *int n* = (*w-val* (*heap-WordArray-u32-C s*) *w*) +$_p$ *int m*)
= (*n* = *m*)
apply (*simp add*: *α-def*)
using *distinct-indices* **apply** *force*
done

lemma *wordarray-distinct-address*:
⟦*α s xs w*; *n* < *length xs*; *m* < *length xs*⟧ ⟹
(*w-val* (*heap-WordArray-u32-C s*) *w*) +$_p$ *int n* ≠ (*w-val* (*heap-WordArray-u32-C s*) *w*) +$_p$ *int m*
⟶ *n* ≠ *m*
apply (*intro allI impI*)
apply *clarsimp*
done

lemma *wordarray-put2-frame*:
{|λ*s*. *state* = *s* ∧ *α s xs w* |}
wordarray-put2-0′ (*t2-C w i v*)
{|λ*r s*. *frame-WordArray-u32-C-ptr state* {} *s* {} {} ∧
frame-word32-ptr state { (*w-p s w*) +$_p$ *uint i*} *s* {(*w-p s w*) +$_p$ *uint i* } {}
|}!
apply (*unfold wordarray-put2-0′-def*)
apply *clarsimp*

apply *wp*
apply *clarsimp*
apply (*rule conjI*; *clarsimp simp*: *uint-nat word-less-def abs-length-eq* α*-nth*)
apply (*simp add*: *frame-WordArray-u32-C-ptr-def frame-word32-ptr-def* α*-def*)
apply (*simp add*: *frame-triv* α*-def*)
done

lemma *wordarray-put2-refinement*:
$\{\!|\lambda s.\ \alpha\ s\ xs\ w\ |\!\}$
wordarray-put2-0' (*t2-C w i v*)
$\{\!|\lambda r\ s.\ \alpha\ s\ (w\text{-}put\ xs\ (unat\ i)\ v)\ r|\!\}!$
apply (*unfold wordarray-put2-0'-def w-put-def*)
apply *clarsimp*
apply *wp*
apply (*clarsimp simp*: *uint-nat*)
apply (*rule conjI*)
apply (*clarsimp simp*: *word-less-nat-alt abs-length-eq*)
apply (*rule conjI*)
apply (*simp* (*no-asm*) *add*: α*-def u32list-all-nth*)
apply (*rule conjI*)
apply *clarsimp*
apply (*rule conjI*)
apply (*clarsimp simp*: *wordarray-same-address* α*-nth*)
apply (*clarsimp simp*: *wordarray-distinct-address* α*-nth*)
apply (*simp add*: α*-def*)
apply (*clarsimp simp*: α*-nth* α*-def*)
apply (*clarsimp simp*: *not-less word-le-nat-alt* α*-def*)
done

lemma *wordarray-put-no-fail*:
$\alpha\ s\ xs\ w \Longrightarrow \neg\ snd\ (wordarray\text{-}put2\text{-}0'\ (t2\text{-}C\ w\ i\ v)\ s)$
apply (*monad-eq simp*: *wordarray-put2-0'-def*)
apply *safe*
apply (*simp add*: α*-def*)
apply *clarsimp*
apply (*frule abs-length-eq*)
apply (*subgoal-tac unat i* $<$ *length xs*)
apply (*metis* (*full-types*) α*-def int-unat u32list-all-nth w-length-def*)
by (*simp add*: *word-less-nat-alt*)

4.4 wordarray_fold_no_break

fun *fold-dispatch* :: *int* $\Rightarrow$ *t3-C* $\Rightarrow$ *word32*
where
fold-dispatch n args = (*if n* = *sint FUN-ENUM-mul-bod*
then t3-C.elem-C args $*$ *t3-C.acc-C args*
else t3-C.elem-C args + *t3-C.acc-C args*)

definition *f-n*

where
f-n ≡ (λ*n a1 a2 a3*. *fold-dispatch* (*sint n*) (*t3-C a2 a1 a3*))

lemma *unat-min*:
unat (*min a b*) = *min* (*unat a*) (*unat b*)
by (*simp add*: *min-of-mono′ word-le-nat-alt*)

lemma *fold-loop-inv*:
{|λ*s*. α *s xs w* ∧ *r* = (*xa*, *y*) ∧ *y* < *ret* ∧ *ret* = *min to* (*w-l s w*) ∧ *frm* ≤ *y* ∧
xd = *t3-C.elem-C-update* (λ*a*. *heap-w32 s* ((*w-p s w*) +$_p$ *uint y*)) *xa* ∧
w-fold xs (*unat frm*) (*unat to*) (*f-n f-num*) *acc obsv* =
w-fold xs (*unat y*) (*unat to*) (*f-n f-num*) (*t3-C.acc-C xa*) *obsv*|}
dispatch-t4′ (*sint f-num*) *xd*
{|λ*xe a*. α *a xs w* ∧ *ret* = *min to* (*len-C* (*heap-WordArray-u32-C a w*)) ∧ *frm* ≤ *y* + *1* ∧
w-fold xs (*unat frm*) (*unat to*) (*f-n f-num*) *acc obsv* =
w-fold xs (*unat* (*y* + *1*)) (*unat to*) (*f-n f-num*) *xe obsv* ∧
min (*unat to*) (*length xs*) − *unat* (*y* + *1*) <
(*case r of* (*fargs*, *i*) ⇒ λ*s*. *min* (*unat to*) (*length xs*) − *unat i*) *s*|}!
unfolding *dispatch-t4′-def mul-bod′-def sum-bod′-def*
apply *wp*
apply (*clarsimp simp*: *word-less-nat-alt unatSuc2 less-is-non-zero-p1 word-le-nat-alt*)
apply (*clarsimp simp*: *unat-min abs-length-eq*)
apply (*simp add*: *diff-less-mono2*)
apply (*case-tac unat frm* < *length xs*)
apply (*case-tac unat frm* < *unat to*)
apply (*simp add*: α*-nth uint-nat*)
apply (*simp* (*no-asm*) *add*: *f-n-def*)
apply (*clarsimp simp*: *w-fold-step*)
apply *simp-all*
done

lemma *wordarray-fold-refinement*:
{|λ*s*. α *s xs w* |}
wordarray-fold-no-break-0′ (*t5-C acc w f-num frm obsv to*)
{|λ*r s*. α *s xs w* ∧ *r* = *w-fold xs* (*unat frm*) (*unat to*) (*f-n f-num*) *acc obsv*|}!
unfolding *wordarray-fold-no-break-0′-def*
apply *wp*
apply(*subst whileLoop-add-inv*[**where** *I* = λ(*fargs*, *i*) *s*. α *s xs w* ∧ *ret* = *min to* (*w-l s w*) ∧
frm ≤ *i* ∧
w-fold xs (*unat frm*) (*unat to*) (*f-n f-num*) *acc obsv* =
w-fold xs (*unat i*) (*unat to*) (*f-n f-num*) (*t3-C.acc-C fargs*) *obsv*
and
M = λ((*fargs*, *i*), *s*). (*min* (*unat to*) (*length xs*)) − *unat i*])
apply *wp*
apply *simp*

apply (*rule fold-loop-inv*)
apply *wp*
apply *wp*
apply *wp*
apply (*clarsimp simp*: *word-less-nat-alt uint-nat* α*-nth abs-length-eq unat-min*)
apply (*rule conjI*, *simp*)+
apply (*simp add*: α*-def*)
apply (*safe*; *clarsimp simp add*: *not-less word-le-nat-alt w-fold-def abs-length-eq*)
apply *wp*
apply *wp*
apply *wp*
apply *clarsimp*
apply *safe*
apply (*simp add*: *min.commute min.strict-order-iff*)
using *min.strict-order-iff* **apply** *fastforce*
apply (*simp add*: α*-def*)
done

lemma *fold-loop-frame-inv*:

⦃$\lambda s.\ \alpha\ s\ xs\ w \wedge r = (xa,\ y) \wedge y < ret \wedge ret = min\ to\ (w\text{-}l\ s\ w) \wedge frm \leq y \wedge$
xd = *t3-C.elem-C-update* ($\lambda a.$ *heap-w32 s* ((*w-p s w*) $+_p$ *uint y*)) *xa* $\wedge$
frame-WordArray-u32-C-ptr state {} *s* {} *A* $\wedge$ *frame-word32-ptr state* {} *s* {} *B*⦄
dispatch-t4' (*sint f*) *xd*
⦃$\lambda xe\ a.\ \alpha\ a\ xs\ w \wedge ret = min\ to\ (len\text{-}C\ (heap\text{-}WordArray\text{-}u32\text{-}C\ a\ w)) \wedge frm \leq y + 1 \wedge$
frame-WordArray-u32-C-ptr state {} *a* {} *A* $\wedge$ *frame-word32-ptr state* {} *a* {} *B* $\wedge$
min (*unat to*) (*length xs*) − *unat* (*y* + *1*) <
(*case r of* (*fargs*, *i*) $\Rightarrow \lambda s.$ *min* (*unat to*) (*length xs*) − *unat i*) *s*⦄!
unfolding *dispatch-t4'-def sum-bod'-def mul-bod'-def*
apply *wp*
apply (*clarsimp simp*: *word-le-nat-alt unatSuc2 less-is-non-zero-p1 word-less-nat-alt*)
by (*simp add*: *diff-less-mono2 abs-length-eq unat-min*)

lemma *wordarray-fold-frame*:
⦃$\lambda s.\ \alpha\ s\ xs\ w \wedge state = s \wedge w \notin A \wedge (\forall i < length\ xs.\ p +_p i \notin B)$⦄
wordarray-fold-no-break-0' (*t5-C acc w f frm obsv to*)
⦃$\lambda r\ s.$ *frame-WordArray-u32-C-ptr state* {} *s* {} *A* $\wedge$
frame-word32-ptr state {} *s* {} *B* ⦄!
unfolding *wordarray-fold-no-break-0'-def*
apply *wp*
apply(*subst whileLoop-add-inv*[**where** *I* =λ(*fargs*, *i*) *s*. α *s xs w* $\wedge$ *ret* = *min to* (*w-l s w*) $\wedge$
frm $\leq$ *i* $\wedge$ *frame-WordArray-u32-C-ptr state* {} *s* {} *A* $\wedge$
frame-word32-ptr state {} *s* {} *B* **and**
M = λ((*fargs*, *i*), *s*). (*min* (*unat to*) (*length xs*)) −

unat i])
apply *wp*
apply *simp*
apply (*rule fold-loop-frame-inv*)
apply *wp*
apply *wp*
apply *wp*
apply (*clarsimp simp*: *word-less-nat-alt uint-nat* α*-nth abs-length-eq unat-min*)
apply (*rule conjI*, *simp*)+
apply (*simp add*: α*-def*)
apply *clarsimp*
apply *wp*
apply *wp*
apply *wp*
apply (*clarsimp simp*: *frame-triv frame-def*)
apply *safe*
apply (*simp add*: *min.commute min.strict-order-iff*)
apply (*metis min.idem min.strict-order-iff neqE*)
apply (*simp add*: α*-def*)
done

4.5 wordarray_map_no_break

fun *map-dispatch* :: *int* $\Rightarrow$ *t6-C* $\Rightarrow$ *word32* $\times$ *unit-t-C*
where
map-dispatch n args = (*if n* = *sint FUN-ENUM-inc then* ((*t6-C.elem-C args* + *1*), (*t6-C.acc-C args*))
else ((*t6-C.elem-C args* + *1*), (*t6-C.acc-C args*)))
definition *m-n*
where
m-n = (λ*n a1 a2 a3*. *map-dispatch* (*sint n*) (*t6-C a2 a1 a3*))

lemma *map-loop*:
⦃λ*s*. *b* = *y* $\wedge$ *y* < *ret* $\wedge$ *ret* = *min to* (*w-l s w*) $\wedge$ (*frm* $\leq$ *ret* $\longrightarrow$ *y* $\leq$ *ret*) $\wedge$
(*ret* < *frm* $\longrightarrow$ *frm* = *y*) $\wedge$ *frm* $\leq$ *y* $\wedge$
α *s* (*fst* (*w-map xs* (*unat frm*) (*unat y*) (*m-n m-num*) *acc obsv*)) *w* $\wedge$
xd = *t6-C.elem-C-update* (λ*a*. *heap-w32 s* ((*w-p s w*) $+_p$ *uint y*)) *xa* $\wedge$
(*t6-C.acc-C xd*) = *snd* (*w-map xs* (*unat frm*) (*unat y*) (*m-n m-num*) *acc obsv*)
⦄
dispatch-t8' (*sint n-num*) *xd*
⦃λ*xe s*.
α (*heap-w32-update* (λ*a b*. *if b* = *values-C* (*heap-WordArray-u32-C s w*) $+_p$ *uint y then t7-C.p1-C xe else a b*) *s*)
(*fst* (*w-map xs* (*unat frm*) (*unat* (*y* + *1*)) (*m-n m-num*) *acc obsv*)) *w* $\wedge$
t7-C.p2-C xe = *snd* (*w-map xs* (*unat frm*) (*unat* (*y* + *1*)) (*m-n m-num*) *acc obsv*) $\wedge$
ret = *min to* (*len-C* (*heap-WordArray-u32-C s w*)) $\wedge$
(*frm* $\leq$ *ret* $\longrightarrow$ *y* + *1* $\leq$ *ret*) $\wedge$ (*ret* < *frm* $\longrightarrow$ *frm* = *y* + *1*) $\wedge$ *frm* $\leq$ *y* + *1* $\wedge$

min (*unat to*) (*length xs*) − *unat* (*y* + *1*) < *min* (*unat to*) (*length xs*) − *unat b* ∧
is-valid-w32 s (*values-C* (*heap-WordArray-u32-C s w*) +$_p$ *uint y*) ∧ *is-valid-WordArray-u32-C s w*⦘!
apply (*unfold dispatch-t8′-def inc′-def*)
apply *wp*
apply *clarsimp*
apply (*rule conjI*)
apply (*subst unatSuc2*)
using *less-is-non-zero-p1* **apply** *fastforce*
apply (*simp* (*no-asm*) *add*: α*-def*)
apply (*clarsimp simp add*: *u32list-all-nth*)
apply (*rule conjI*)
apply *clarsimp*
apply (*rule conjI*)
apply *clarsimp*
apply (*rule conjI*)
apply (*simp add*: α*-def*)
apply (*metis u32list-all-nth w-map-length*)
apply (*subgoal-tac uint y* = *int k*)
apply (*subst w-map-step*)
using *word-le-nat-alt* **apply** *blast*
apply (*subgoal-tac k* = *unat y*)
apply *simp*
apply (*subst nth-list-update-eq*)
apply (*simp add*: *w-map-length*)
apply (*simp add*: *m-n-def*)
apply (*simp add*: α*-nth*(*1*) *w-map-endpoint w-map-length*)
apply (*metis int-unat nat-int.Rep-eqD*)
apply (*clarsimp simp add*: *uint-nat word-less-nat-alt abs-length-eq w-map-length*[*symmetric*])
apply (*metis wordarray-same-address w-map-length-same*)
apply *clarsimp*
apply (*rule conjI*)
apply (*simp add*: α*-def u32list-all-nth w-map-length*)
apply (*subst w-map-step*)
using *word-le-nat-alt* **apply** *blast*
apply (*subgoal-tac k* ≠ *unat y*)
apply (*simp add*: α*-nth*(*1*) *w-map-length*)
apply (*metis int-unat*)
apply (*rule conjI*)
apply (*simp add*: α*-def u32list-all-nth w-map-length*)
apply (*simp add*: α*-def u32list-all-nth w-map-length*)
apply (*rule conjI*)
apply (*subst unatSuc2*)
using *less-is-non-zero-p1* **apply** *fastforce*
apply (*subst w-map-step*)
using *word-le-nat-alt* **apply** *blast*
apply (*metis Suc-le-eq abs-length-eq w-map-length unat-mono*)
apply (*simp add*: *m-n-def*)

apply (*rule conjI*)
apply *clarsimp*
using *inc-le* **apply** *blast*
apply (*rule conjI*)
apply *clarsimp*
apply (*meson min-less-iff-conj not-less-iff-gr-or-eq*)
apply (*rule conjI*)
apply (*metis* (*mono-tags*, *hide-lams*) *add.commute less-is-non-zero-p1 min.strict-coboundedI2 min-def word-le-less-eq word-overflow*)
apply (*rule conjI*)
apply (*subgoal-tac length xs* > *unat y*)
apply (*subgoal-tac unat to* > *unat y*)
apply (*meson diff-less-mono2 less-is-non-zero-p1 min-less-iff-conj word-less-nat-alt word-overflow*)
using *unat-mono* **apply** *blast*
apply (*simp add*: *abs-length-eq w-map-length word-less-nat-alt*)
apply (*rule conjI*)
apply (*subgoal-tac unat y* < *length xs*)
apply (*simp add*: *α-nth*(*2*) *w-map-length uint-nat*)
apply (*simp add*: *abs-length-eq w-map-length word-less-nat-alt*)
apply (*simp add*: *α-def*)
done

lemma *wordarray-map-refinement*:
{|λ*s*. *α s xs w* |}
wordarray-map-no-break-0′ (*t9-C acc w n-num frm obsv to*)
{|λ*r s*. *α s* (*fst* (*w-map xs* (*unat frm*) (*unat to*) (*m-n m-num*) *acc obsv*)) *w* ∧
(*t10-C.p2-C r*) = *snd* (*w-map xs* (*unat frm*) (*unat to*) (*m-n m-num*) *acc obsv*)|}!
unfolding *wordarray-map-no-break-0′-def*
apply *wp*
apply(*subst whileLoop-add-inv*[**where** *I* =λ(*fargs*, *i*) *s*.
α s (*fst* (*w-map xs* (*unat frm*) (*unat i*) (*m-n m-num*) *acc obsv*)) *w* ∧
(*t6-C.acc-C fargs*) = *snd* (*w-map xs* (*unat frm*) (*unat i*) (*m-n m-num*) *acc obsv*) ∧
ret = *min to* (*w-l s w*) ∧ (*frm* ≤ *ret* ⟶ *i* ≤ *ret*) ∧ (*frm* > *ret* ⟶ *frm* = *i*) ∧ *frm* ≤ *i*
and
M = λ((*fargs*, *i*), *s*). (*min* (*unat to*) (*length xs*)) − *unat i*])
apply *wp*
apply *clarsimp*
apply (*wp map-loop*)
apply *wp*
apply *wp*
apply *wp*
apply *clarsimp*
apply (*rule conjI*)
apply *simp*
apply (*metis* (*full-types*) *α-def int-unat u32list-all-nth w-length-def word-less-nat-alt*)

```
      apply clarsimp

      apply (case-tac min to (len-C (heap-WordArray-u32-C s w)) < frm)
       apply (case-tac b > to)
        apply (clarsimp simp add: w-map-def)
        apply (rule FalseE)
        apply (simp add: dual-order.strict-implies-order unat-mono)
       apply clarsimp
       apply (subgoal-tac unat frm > length xs)
        apply (simp add: w-map-def)
     apply (metis (mono-tags, hide-lams) abs-length-eq w-map-length min-less-iff-disj
unat-mono)
      apply clarsimp
      apply (subgoal-tac frm ≤ to ∧ frm ≤ len-C (heap-WordArray-u32-C s w))
       apply clarsimp
       apply (case-tac b = to)
        apply simp
       apply (subgoal-tac b < to)
        apply clarsimp
        apply (metis abs-length-eq w-map-bounds w-map-length word-le-nat-alt)
  using word-le-less-eq apply blast
      apply (simp add: min-less-iff-disj word-le-not-less)
     apply wp
    apply wp
   apply wp
  apply wp
  apply clarsimp
  apply (simp add: w-map-def)
  apply (rule conjI)
   apply (simp add: min.commute min.strict-order-iff)
  apply (simp add: α-def)
  using min.strict-order-iff apply fastforce
  done

end
end
```