Final Report for AOARD Grant #FA2386-11-1-4070

Formal System Verification - Extension

June Andronick, Andrew Boyton, Gerwin Klein

18 October 2012

Principal Investigators: Gerwin Klein and June Andronick

Email: gerwin.klein@nicta.com.au, june.andronick@nicta.com.au
Institution: NICTA
Mailing Address: 223 Anzac Parade, Kensington NSW 2052, Australia
Phone: +61 2 8306 0578
Fax: +61 2 8306 0406

Period of Performance: 18 May, 2011 - 17 Oct, 2012

Abstract

The AOARD project #FA2386-11-1-4070 aims at providing a provably correct *initialiser* of componentised systems. Taking as input a description of the desired components and the desired authorised communication between them, the initialiser sets up the system and provides a proof that the resulting concrete machine state of the system matches the desired authority state. Within the scope of this project, we provide (1) a formal *specification* of the initialiser, in terms of the steps needed to create the components and their communication channels; and (2) substantial progress towards a formal proof that this specification is correct in that it either fails safely or produces the desired state.

This document is the final report of the project, presenting its scientific outcomes. Namely, we have completed the initialiser specification, we have created a verification framework enabling modular reasoning and proofs about the initialiser, and we have progressed substantially on the proof itself, demonstrating that such proofs are feasible.

Contents

1. Introduction

This final report describes the outcomes of AOARD project #FA2386-11-1-4070, "Formal System Verification - Extension". The project ran from 18 May 2011 to 17 October 2012.

This project is part of a larger research vision of building truly trustworthy software systems; in particular the formal verification of large, complex embedded systems. A first step in this vision was to provide a formally verified microkernel basis. We have previously developed the seL4 microkernel, together with a formal proof (in the theorem prover Isabelle/HOL [6]) of its functional correctness [2]. This means that all behaviours of the seL4 C source code are included in the high-level, formal specification of the kernel.

The approach taken to provide formal, code-level guarantees about the security or safety of systems on top of this kernel is to minimise the *trusted computing base* by designing the system architecture in a componentised way, where untrusted components can be isolated from trusted ones. With suitable architectures, the amount of code to be verified can be reduced from million of lines of code to tens of thousands. This MILS-style of security architecture [1] enables us to concentrate the verification effort on the trusted components. In our case, the isolation is provided by the underlying seL4 kernel, in terms of integrity and authority confinement [9], as well as confidentiality (in ongoing work).

The project we report on here focuses on the critical step of initialising the system into a state satisfying the specified architecture. Namely, the project goal is to:

> *(1) construct a formal, high-level specification of the system initialiser component in the theorem prover Isabelle/HOL that is connected to the existing formal specification of the seL4 microkernel and*
> *(2) prove that this specification of the system initialiser component is correct in the sense that it will always either fail safely at initialisation time or produce a system state that formally corresponds to the specified target.*

We begin the remainder of this report with background information on the seL4 microkernel and in particular its access control mechanism, which is critical to the system initialiser. After introducing notation, we then describe the exact project scope and finally explain in detail our formal specification of the initialiser and the outcomes of its proof of correctness at the end of this project. In particular, we describe our instantiation of *separation logic* that enables the decomposition of the proof into smaller statements that can be reasoned about independently.

While we did not finish the full proof in the available project time, we have made enough progress on it to show that this proof is feasible and can be completed in future work. In particular, we have covered the breadth of the specification with invariant and pre/post condition annotations on the top level, and we have investigated one deep slice of the proof, connecting one of the

most complex operations in the system initialiser down to the kernel call level. We expect the remaining operations to be suited for the same kind of reasoning, only with less complexity.

The main outcome of the project is the proof framework that is scalable enough for concrete state operations yet flexible enough to deal with the complexities of system initialisation. We managed to abstract significant parts of this framework into a general separation logic framework that is readily applicable to other applications as well. The separation logic framework has been published and its Isabelle/HOL sources are available for download from the Archive of Formal Proofs [4].

We attach the publication describing the separation logic framework at the end of this report. This framework is then instantiated in the work presented here in Section 6.

2. Background

2.1. seL4

The seL4 microkernel is a small operating system kernel designed to be a secure, safe, and reliable foundation for a wide variety of application domains. The microkernel is the only software running in the *privileged* mode (or *kernel* mode) of the processor. The privileged mode is used to protect the operating system from applications and applications from each other. Software other than the kernel runs in unprivileged mode (or *user* mode). As a microkernel, seL4 provides a minimal number of services to applications: threads, inter-process communication and virtual memory, as well as a capability-based access control. Threads are an abstraction of CPU execution that support running software. As shown in Figure 2.1, threads are represented in seL4 by their *thread control blocks* (TCB), that store a thread's context, virtual address space (VSpace) and capability space (CSpace). VSpaces contain a set of frames, generally organised in a hierarchical, architecture-dependent structure of page tables and page directories. CSpaces are kernel managed storage for *capabilities*. A capability is an unforgeable token that confers authority. In seL4, a thread may only invoke operations on objects when it has sufficient permissions. This means that every call to the kernel requires the presentation of a capability with the correct authority to perform the specified operation. A thread stores its capabilities in its CSpace, which is a directed acyclic graph of capability nodes (CNodes). These CNodes can be of various sizes and contain capabilities to other CNodes and objects. When a thread invokes seL4, it provides an index into this structure, which is traversed and resolved into a real capability. Communication between components is enabled by *endpoints*. When a thread wants to send a message to another thread, it must send it to an intermediate endpoint to which it has send access and to which the receiver has receive access, as shown in Figure 2.2.

The allocation of kernel objects in seL4 is performed by *retyping* an *untyped memory* region. Untyped memory is an abstraction of a power-of-two sized, size-aligned region of physical memory. Possession of capability to an untyped memory region provides sufficient authority to allocate kernel objects in this region: a user-level application can request that the kernel transforms that region into other kernel objects (including smaller untyped memory regions). At boot time, seL4 first pre-allocates the memory required for the kernel itself and then gives the remainder of the memory to the initial user task in the form of capabilities to untyped memory.

2.2. seL4 Verification

The seL4 microkernel was the first, and is still, to our knowledge, the only general-purpose operating system kernel that is fully formally verified for functional correctness. This means that there exists a formal, machine-checked proof that the C implementation of seL4 is a correct

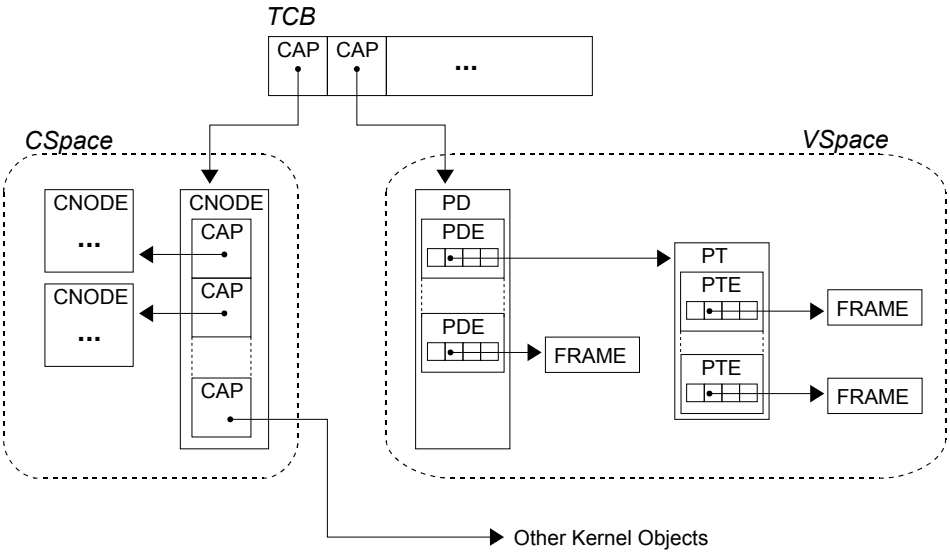

Figure 2.1.: Internal representation of an application in seL4.

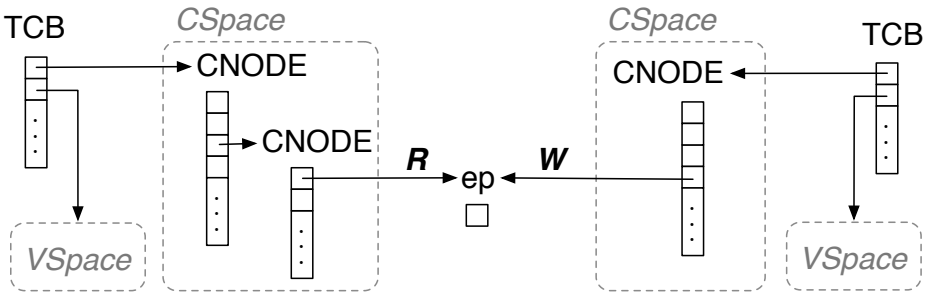

Figure 2.2.: seL4-based system with two threads that can communicate via an *endpoint*.

refinement of its functional, abstract specification. This proof assumes the correctness of the compiler, assembly code, boot code, management of caches, and the hardware. The technique used for formal verification is interactive, machine-assisted and machine-checked proof. Specifically, we use the theorem prover Isabelle/HOL [6].

As shown in Figure 2.3, the verification uses several specification layers. The top-most layer in the picture is the *abstract specification*: an operational model that is the main, complete specification of system behaviour. The next layer down is the *executable specification*, representing the design of the kernel. This layer is generated from a Haskell prototype of the kernel, aimed at bridging the gap between kernel development needs and formal verification requirements. Finally, the bottom layer is the high-performance C implementation of seL4, parsed into Isabelle/HOL using a precise, faithful formal semantics for the C programming language [7, 10].

The correspondence established by the refinement proof enables to conduct further proofs of any *Hoare logic* properties (see Section 6) at the abstract level, ensuring that the properties also hold for the refined model. This means that if a security property is proved in Hoare logic

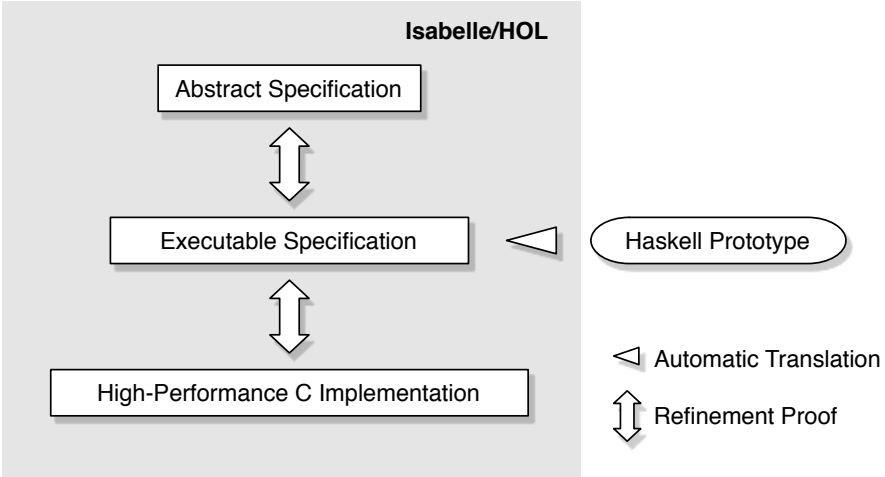

Figure 2.3.: The refinement layers in the verification of seL4 functional correctness.

about the abstract model (not all security properties can be), refinement guarantees that the same property holds for the kernel source code. Since proofs at the abstract level are easier to perform, this allows a significant reduction of effort in additional verifications, as illustrated by the proof that seL4 enforces integrity and authority confinement [9].

2.3. capDL

One of the design goals of the seL4 kernel was to capture all access control relevant state by capabilities. However, despite its capability-oriented design, seL4, like other microkernels, contains authority relevant to information flow and access control that is not conferred by capabilities. For example, the memory mapped into a thread's address space is not mapped using capabilities, nor are various hardware accesses such as IO ports.

Having all the protection state described by capabilities would enable reasoning about the access control and security of a system through capability distributions alone. It would also allow components and connections of user-level systems on top of seL4 to be described by their capability distribution alone. Such a description is basically a graph with objects as nodes and capabilities as edges. To reason about such specific graphs, we developed the capability distribution language capDL [5]. The capDL language unifies all information relevant to information flow and access control as explicit capabilities. In addition to the language itself, which describes snapshots of system states, we developed, in previous work, a kernel semantics for this language that describes the effect of each kernel operation on such states, and showed that it is a formally correct abstraction of existing models of seL4, with a complete refinement chain to the C code level, as shown in Figure 2.4. This high-level model of seL4, which will be referred to as the *seL4 capDL model*, fully encapsulates the protection state of the system.

Using capDL, we are able to precisely describe the protection state of a system we wish to run on seL4. For instance, the example system shown in Figure 2.2 can be formally described

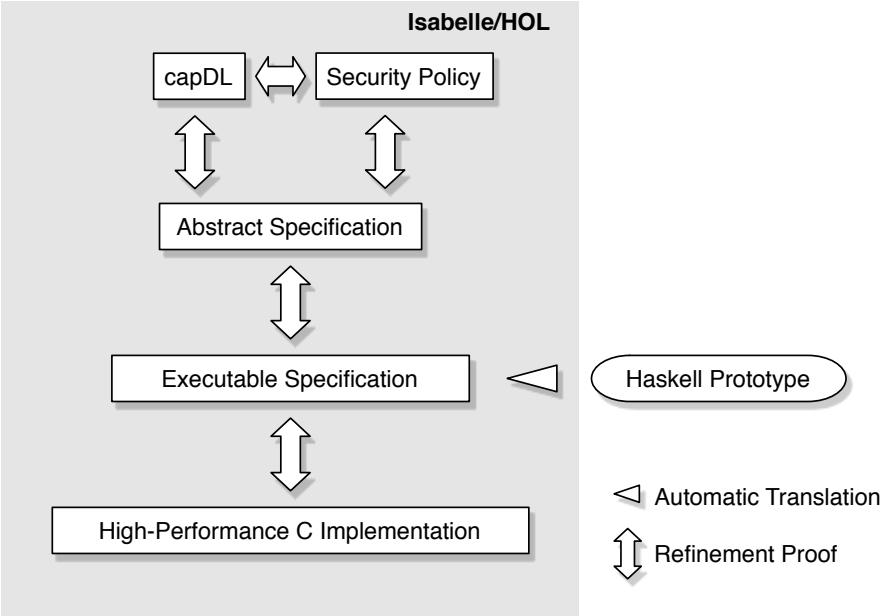

Figure 2.4.: capDL model in the seL4 refinement chain.

using the *capDL specification* shown in Figure 2.5. This capDL specification describes all system objects through a mapping from unique object identifiers to the objects themselves. Each object contains a set of capabilities, as well as other attributes, referred to as *fields*. For example, the object reference 0 maps to a TCB object whose CSpace root is the CNode at reference 2 and whose VSpace root is the page directory at reference 1. In turn, the CNode at reference 2 contains a set of capabilities, one of which is a capability to the CNode at reference 3, which itself contains the capability, with $Read$ right, to the endpoint at reference 4. Other than rights, capabilities may contain other attributes such as *guards* or other tags; objects also contain other capabilities or fields; and the capDL specification itself contains other information than the set of objects; but these are irrelevant for understanding the initialiser definition and proof.

In the project reported here, we use formal capDL specifications, similar to the one given in Figure 2.5, to describe the system to be initialised.

$$
\begin{aligned}
&(\!| \mathit{cdl\_objects} = \\
&\quad [\, 0 \mapsto \mathit{Tcb}\ (\!| \mathit{cdl\_tcsb\_caps} = [\, 0 \mapsto \mathit{CNodeCap}\ 2\ \mathit{guard}\ \mathit{guard\_size}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad 1 \mapsto \mathit{PageDirectoryCap}\ 1\ \mathit{True}\ \mathit{None}, \ldots], \\
&\qquad\qquad\qquad \mathit{cdl\_tcb\_fault\_endpoint} = \ldots, \\
&\qquad\qquad\qquad \mathit{cdl\_tcb\_intent} = \ldots\ |\!), \\
&\qquad 1 \mapsto \mathit{PageDirectory}\ \ldots, \\
&\qquad 2 \mapsto \mathit{CNode}\ (\!| \mathit{cdl\_cnode\_caps} = [\ldots,\ 2 \mapsto \mathit{CNodeCap}\ 3\ \mathit{guard}\ \mathit{guard\_size},\ \ldots], \\
&\qquad\qquad\qquad\qquad \mathit{cdl\_cnode\_size\_bits} = \ldots\ |\!), \\
&\qquad 3 \mapsto \mathit{CNode}\ (\!| \mathit{cdl\_cnode\_caps} = [0 \mapsto \mathit{EndpointCap}\ 4\ \mathit{badge}\ \{\mathit{Read}\}, \ldots], \\
&\qquad\qquad\qquad\qquad \mathit{cdl\_cnode\_size\_bits} = \ldots\ |\!), \\
&\qquad 4 \mapsto \mathit{Endpoint}, \\
&\qquad 5 \mapsto \mathit{Tcb}\ (\!| \mathit{cdl\_tcb\_caps} = [\, 0 \mapsto \mathit{CNodeCap}\ 6\ \mathit{guard}\ \mathit{guard\_size}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad 1 \mapsto \mathit{PageDirectoryCap}\ 7\ \mathit{True}\ \mathit{None}, \ldots], \\
&\qquad\qquad\qquad \mathit{cdl\_tcb\_fault\_endpoint} = \ldots, \\
&\qquad\qquad\qquad \mathit{cdl\_tcb\_intent} = \ldots\ |\!), \\
&\qquad 6 \mapsto \mathit{CNode}\ (\!| \mathit{cdl\_cnode\_caps} = [0 \mapsto \mathit{EndpointCap}\ 4\ \mathit{badge}\ \{\mathit{Write}\}], \\
&\qquad\qquad\qquad\qquad \mathit{cdl\_cnode\_size\_bits} = \ldots\ |\!), \\
&\qquad 7 \mapsto \mathit{PageDirectory}\ \ldots, \\
&\qquad \ldots], \\
&\ldots \\
&\mathit{current\_thread} = \ldots\ |\!)
\end{aligned}
$$

Figure 2.5.: capDL specification of Figure 2.2.

3. Notation

The Isabelle/HOL notation used in this report largely conforms to everyday mathematical notation and the conventions of functional programming languages such as Haskell or Standard ML.

The main concepts used in the specification are functions, type declarations and non-recursive function definitions.

3.1. Types and Functions

Function application follows the convention of functional programming. That means `f x`, often written as `f(x)` in mathematics, is the function `f` applied to argument `x`. Multiple arguments are delimited by spaces, i.e. `g x (3 + y)` is the function `g` applied to two arguments namely `x` and `3 + y`. Functions can be partially applied, i.e. not all arguments need be given at the call site. For the function `g` for instance, the term `g x` would denote a function that expects one further parameter. Predicates are functions that return values of type `bool`.

Function types are written with ⇒, for instance `ptr ⇒ nat ⇒ ptr`. This denotes a total function that takes a `ptr` and natural number of type `nat` as a parameter and returns a `ptr`. The notation `f :: ptr ⇒ nat` means that function `f` has type `ptr ⇒ nat`.

Isabelle allows polymorphic types with variables, distinguished from normal types by a quote character `'a`. For instance, `ptr ⇒ 'a ⇒ ptr` is a function that takes a `ptr` and a value of arbitrary type `'a`, and delivers a `ptr`. Finally, functions can be higher-order, i.e. they can again take functions as arguments. For example, `nat ⇒ (nat ⇒ nat) ⇒ nat` is a function whose first argument is a natural number and whose second argument is a function from `nat` to `nat`.

New types can be built from existing types by using type constructors. For instance, `ptr set` is a set of pointers, `'a list` is a list of elements of type `'a`. Note that `'a list` is variable in `'a`, but demands that all elements of the list have that same type `'a`. The type of pairs is denoted by ×, i.e. `(ptr × nat) set` is a set of pairs consisting of `ptr` and natural numbers. A concrete value of the pair type is written `(x,y)` as in standard mathematics.

New type constructors can be defined using the **datatype** command. The `option` type for instance is defined as a datatype and often used to add a special element to an existing type to indicate failure or undefinedness. Its definition is

datatype `'a option = None | Some 'a`

A lookup function could for instance return `None` for lookup failure and `Some r` to indicate a result `r`.

A similar type mechanism used in this specification is that of records. A record is a tuple with named fields. The definition

```
record point =
  x :: nat
  y :: nat
```

introduces a new type `point` that contains two fields x, and y, both of type `nat`. If p is of type `point`, the term `x p` is the x-field of p and `y p` the y-field of p, that is, record fields names can be used as accessor functions.

The term `(| x = 3, y = 7 |)` constructs a new record and stands for a `point` with x-field `3` and y-field `7`. To update an existing record p, we write `p (|x := 4|)`. This stands for the record that has the y-field of p and the x-field `4`.

3.2. Lambda

In this report we make use of anonymous functions, that is, functions that are used only locally and are not given a specific name or their own function definition. The notation is the standard λ from functional programming: $\lambda x.\ t$ is a function that takes an argument x and returns the result t (x may occur in t).

An example usage is the definition of the function update notation: $f(a := b)$ stands for the function that at position a returns the value b, and otherwise returns what f would have returned. Its formal definition is

$$f(a := b) \equiv \lambda x.\ \textit{if}\ x = a\ \textit{then}\ b\ \textit{else}\ f\ x$$

3.3. State Monad and Do-Syntax

Since the system initialiser in this report is inherently state-based, many functions in the specification modify state in some way. Isabelle/HOL provides a convenient specification mechanism for this that is also known from functional languages such as Haskell, the so-called *state monad*. A state monad is merely a function from a state $'s$ to a pair of new state and return value $'s \times 'r$. This can be thought of as modelling a usual C function that has a side effect (produces a new state) and returns a value. The syntax for chaining such functions together is the following:

```
do
    a ← f;
    b ← g a;
    h a b
od
```

In the example, first function f is called, returning value a, and potentially changing the underlying state. In this new state $g\ a$ executes, producing result b and potentially using f's result a. The whole `do`-block returns whatever $h\ a\ b$ returns, and produces the state after execution of h. This means the state is threaded through the functions implicitly, while the result values are passed explicitly. The type of each of the single functions is still $'s \Rightarrow 's \times 'r$, potentially with additional arguments if they make use of previous result values.

The idea is to make the syntax reminiscent of an imperative programming style while staying in a strictly functional setting where the state and its type could be made explicit if convenient. It also allows the type checker to enforce that certain functions only modify certain parts of the state.

The library this specification is based on extends such state monads with non-determinism and a mechanism to raise and check assertions. Non-determinism basically means that functions do not return only one state and result, but a set of possible new states and results. The type of such functions is abbreviated as `('s, 'v) nondet_monad` where `'s` is the type of the state and `'v` the type of the result value. If the function does not return a result value, the type `unit` is used for `'v`.

The following example shows a number of basic monad functions and further syntax.

```
do
    x ← gets field_name;
    v ← select {v. P x};
    z ← assert_opt v;
    return $ x + v
od
```

The meaning of these are in order of occurrence:

`gets` expects a function `'s ⇒ 'v` as argument and returns the result of applying this function to the state. It does not modify the state. It is usually used to extract one field from the state when the state is modelled as a record.

`select` nondeterministically picks an element from a set.

`assert_opt v` fails if the value `v` is `None`, and returns `x` if it is `Some x`.

`return` returns its argument and does not modify the state. It is used to perform computations that are state-independent.

`f $ x` applies function `f` to argument `x`. This alternative syntax for function application makes it possible to write fewer parentheses. For example, `f $ x + v` is equivalent to `f (x+v)`.

4. Project Scope

Given a formal capDL specification describing the target system, we need to produce initialiser code that runs at system startup to give a state where all the components have been created and their communication channels set up.

As explained in Section 2.1, when a seL4 system starts, the kernel creates an initial user task – the *root task* – with access rights to all of the memory not used by the kernel itself. More precisely, the kernel creates all the objects needed by the root task such as its TCB, capability space and virtual address space. Capabilities to untyped memory are stored in the capability space, together with capabilities to allow hardware access. At the end of this booting phase, the root task is enabled to run and starts executing.

Our work produces a custom root task to perform system initialisation. We provide both a *specification* describing its execution and progress towards a *proof* of its correctness.

The specification describes the code for allocating objects, managing capabilities to the objects, copying or transferring authority, managing and mapping frames, and setting any required data (such as thread instruction pointers). Manually performing these tasks for each given system is complicated, error prone, and inflexible. For instance, creating an object requires possessing a capability to an untyped memory region of suitable size, which in turn requires possessing a slot in the root task's CSpace to store this capability, and so on. A slight change in the desired system might require storing more capabilities, which may be difficult or impossible if the root task's CSpace size is hardcoded. Instead we choose to provide a *generic tool*, that takes any capDL specification as input, and *automatically* produces code to create and initialise the objects.

Together with the code, the tool also generates a proof asserting that after executing the code from the initial state after kernel booting, the system is in a final state *conformant* to the original capDL specification. We use a notion of conformance instead of equality for two reasons. First, as will be described in detail in Section 7.1, capDL specifications use object identifiers that need to be mapped to memory addresses. Second, we have not yet examined different options for deleting the root task after system initialisation. Therefore the root task stays dormant in memory, but will be provably isolated from the rest of the system. Future work could look into removing the root task from memory after it has finished initialising the system. In this project, the final state after initialisation will therefore contain some root task objects in addition to the objects described in the capDL specification.

As formal proof at the source code level requires significant effort, we follow the approach successfully used for seL4 security proofs, by performing the proof at the abstract specification level, and relating it to the source code through a separate refinement proof. We therefore describe the behaviour of the initialiser by a set of high-level instructions in the seL4 capDL model. We call this behaviour description the *initialiser specification* and we prove conformance with the capDL specification at the initialiser specification level. Later work (outside the scope of this project) will involve producing an implementation of the initialisation code in C, and proving a

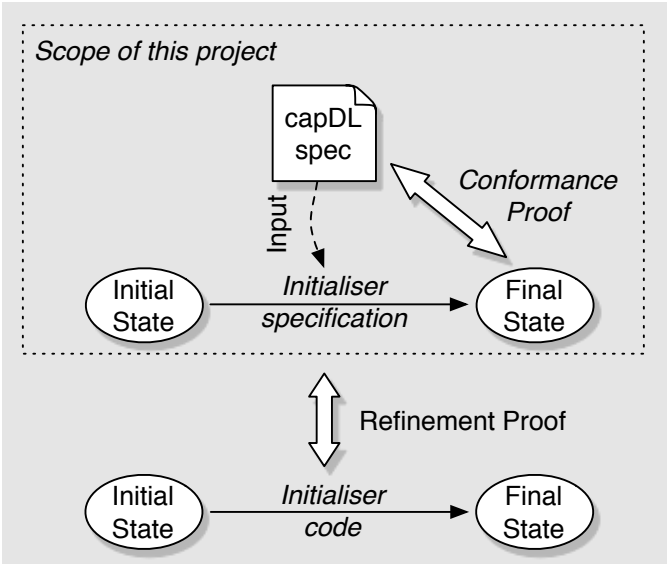

Figure 4.1.: Overview of initialiser correctness proof.

formal refinement between the initialiser specification and the C implementation, as illustrated in Figure 4.1.

To summarise, within the scope of this project:

- we have defined a formal specification `init_system` as a set of high level instructions, which takes a capDL specification `spec` as input, and sets up objects and capabilities as required by `spec`;

- we have made substantial progress in proving that executing `init_system spec` from a state where only the root task's objects exist results in a final state where each object mentioned in `spec` maps to an object in the state's heap, and the only other existing objects are the root task's objects. The desired property is stated formally as:

  If `well_formed spec` *and* `injective` φ *and*
  `object_ids = dom (cdl_objects spec)` *then*

  $\{\!|\!\langle\!\langle$ `root_objects` $\rangle\!\rangle\!|\!\}$ `init_system spec`
  $\{\!|\!\langle\!\langle \bigwedge^{*}$ `map (object_done spec` φ`) object_ids` $\wedge^{*}$ `root_objects` $\rangle\!\rangle\!|\!\}$

  This statement will be explained in depth in Section 7. It assumes a wellformedness condition about `spec` and the existence of an injection φ mapping object identifiers to memory addresses. It uses the predicate `object_done` stating that a given object is successfully created according to `spec`, and iterates this predicate over the list `object_ids` of all identifiers of the objects to be created in `spec`.

5. Specification of the System Initialiser

In this section, we present the outcome of the first goal of this project, as stated in Section 1, namely the formal specification `init_system` of the system initialiser. This specification takes a capDL specification `spec` as input and is defined as a set of high level instructions setting up objects and capabilities as required by `spec`. The full formalisation of the specification is given in Appendix A. Here we give an overview of its structure and formalisation of the main steps.

5.1. Top-level definition

The initialiser is the root task running just after kernel booting, i.e. from a state where its TCB, capability space and virtual address space have been created by the kernel, and where it possesses capabilities to all of the memory regions not used by the kernel itself. Such a state is illustrated in Figure 5.1.

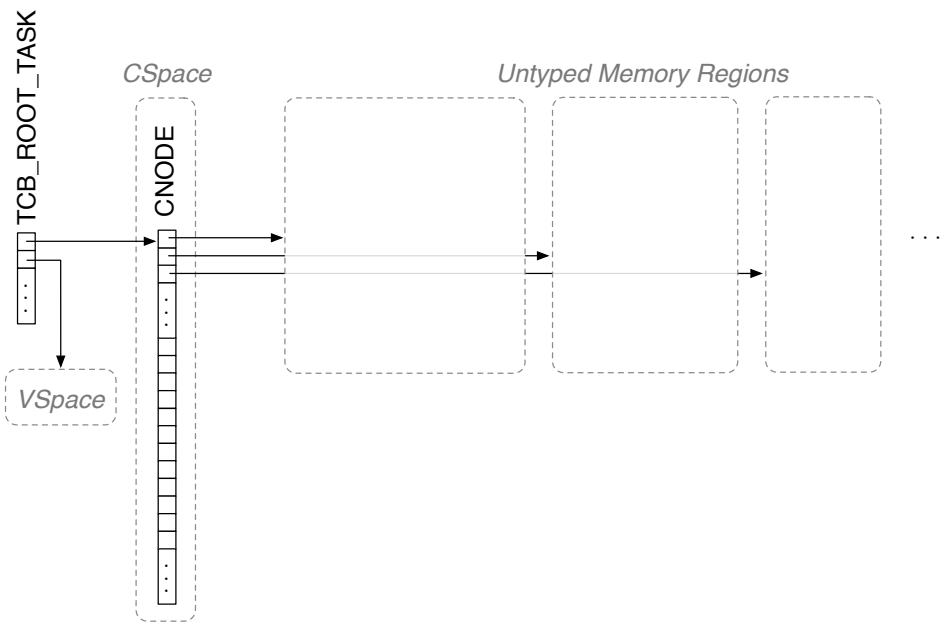

Figure 5.1.: Initial state, after kernel booting.

The top-level definition of the specification is the following:

```
init_system spec =
do bootinfo ← get_bootinfo;
```

```
   parse_bootinfo bootinfo;
   parse_spec spec;
   create_objects spec;
   duplicate_cnode_caps spec;
   init_vspace spec;
   init_tcbs spec;
   init_cspace spec;
   start_threads spec
od
```

The specification is divided into the following well-defined separate phases.

- Firstly, the initialiser processes the information provided by the kernel (in the initial state), and the capDL specification itself (three first lines of `init_system` body).
- Secondly it creates all the objects specified in the capDL specification, and then duplicates all the capabilities to the newly created objects in order to be able to later move some of those capabilities into some component's CSpace while keeping a copy of the capability (fourth and fifth line of `init_system` body).
- Finally, it initialises each of these objects by type, including installing the capabilities into the capability storage objects, and sets all threads to be runnable (sixth to ninth lines of `init_system` body).

Each instruction operates on a state of the system, modelled by a record containing the kernel state, as well as bookkeeping information used by the initialiser:

```
record user_state =
  capdl_to_sel4_orig :: cdl_object_id ⇒ cdl_cptr option
  capdl_to_sel4_copy :: cdl_object_id ⇒ cdl_cptr option
  capdl_objects      :: cdl_object_id list
  free_untypeds      :: cdl_cptr list
  free_slots         :: word32 list
  kernel_state       :: cdl_state
```

The first two fields, `capdl_to_sel4_orig` and `capdl_to_sel4_copy`, are used to store the original and duplicates of all the caps in the system. These will be used when initialising the capability spaces of the system, as we will see in Section 5.4. The field `capdl_objects` stores the objects of `spec` that need to be created. This is just a list created from the mapping contained in `spec`. The field `free_untypeds` stores a list of the capabilities to untyped memory regions that haven't been used yet. The field `free_slots` keeps track of the free slots in the initialiser's CSpace, to be able to use them as destination slots when creating new objects.

Finally, since we take `user_state` as the full state of the entire system for this specification, we also include `kernel_state` in the record. This state will not directly be changed by the operations in the specification, but only through seL4 API calls which connect to our previous functional specification of the seL4 kernel.

The clear separation into phases is designed to assist in the formal proof of the initialiser's correctness, as it makes it relatively easy to specify the state of the system at any given point. Each phase is now described in detail, illustrated by the example given in Figure 2.2 and Figure 2.5.

5.2. Information processing

In a first phase, the initialiser analyses the boot information provided by seL4. The function `parse_bootinfo` initialises the bookkeeping information by setting `free_untypeds` to the list of capabilities to *all* the untyped memory regions, and setting `free_slots` to all of the available slots in the initialiser's capability space. The number of free capability slots provided to the root task is specified at compile time for seL4. It is therefore possible to make sure that there will be enough free slots available for initialisation of a specific system. Finally, the fields `capdl_to_sel4_orig` and `capdl_to_sel4_copy` are set to empty mappings.

The specification `spec` is then parsed to extract the list of objects that need to be created and this is stored in the `capdl_objects` field of the state.

5.3. Object creation and capability duplication

In a second phase, the initialiser creates all the objects required by the capDL specification `spec`. Objects are created sequentially from the untyped memory regions, as illustrated in Figure 5.2.

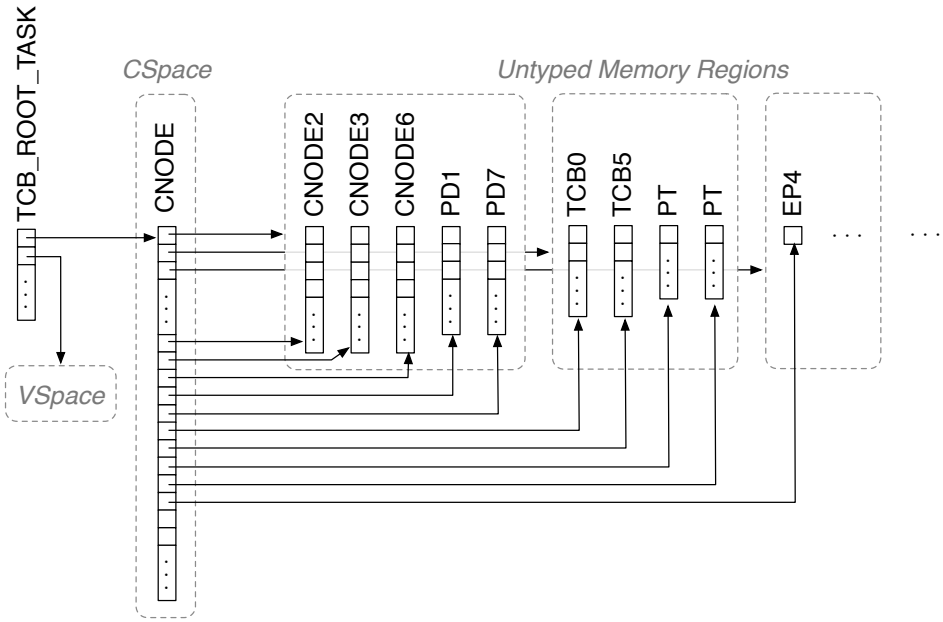

Figure 5.2.: Creation of objects for the system described in Figure 2.2 and Figure 2.5.

Each untyped memory region can be retyped incrementally to create new objects. Any ordering of the creation of objects, and any choice of the untyped memory region they are created from, is safe. Some orderings may however make inefficient use of space as seL4 requires all objects to be aligned to their size. Ordering the creation of objects from the largest object to the smallest, and from the largest untyped memory region to the smallest, is conjectured to be optimal, but proving it is outside the scope of this work. At this stage, the creation is just done in order of appearance

in the list `capdl_objects` (extracted from the mapping in `spec`). The `create_objects` function is therefore just applying a function `create_object` to each of the element of the list:

```
create_objects spec =
do objects ← get_capdl_objects;
   mapM_x (create_object spec) objects
od
```

where `mapM_x` runs the given operation on each element of the given list, and where `create_object` is defined as:

```
create_object spec object_id =
do free_slot ← get_free_slot;
   object ← get_spec_object spec object_id;
   object_type ← return $ object_type object;
   object_size ←
   return $ of_nat $ object_at_pointer_size_bits spec object_id;
   retype_untypeds free_slot object_type object_size;
   add_sel4_cap object_id OrigCap free_slot;
   next_free_slot
od
```

The capabilities to each of these newly created objects are stored in the free capability slots of the initialiser. The creation is done using the function `retype_untypeds`, which in turn calls the seL4 service `seL4_Untyped_Retype`:

```
retype_untyped free_slot free_untyped type size_bits ≡
do root ← return seL4_CapInitThreadCNode;
   node_index ← return 0;
   node_depth ← return 0;
   node_offset ← return free_slot;
   no_objects_to_create ← return 1;
   seL4_Untyped_Retype free_untyped type size_bits root node_index
    node_depth node_offset no_objects_to_create
od
```

The `seL4_Untyped_Retype` kernel call takes as input a capability to an untyped memory region (here the first available in `free_untypeds`), the size and type of object to be created, and information about where the capability to this newly created object should be stored, given by a CSpace root (here the root task's CSpace root), and slot in this CSpace (here the next available in `free_slots`).

The `create_object` function then calls `add_sel4_cap` function which simply stores the locations of the capabilities to the newly created object into `capdl_to_sel4_orig` field of the state. This bookkeeping will be used later when initialising the capability spaces: if `spec` specifies that a given thread needs to possess a capability to a given object, the initialiser needs to copy its own capability to that object into the thread's CSpace. In some cases, the original capability should be *moved* (instead of *copied*) into a thread's CSpace. There is a distinction in seL4 between *master* capabilities (which are the original capabilities acquired at object creation), and *derived* capabilities (which are copies of these original capabilities). Master capabilities confer more authority as they allow for the complete deletion of an object. When initialising a

system, we may want to provide some of the objects with master capabilities. In this case, if the initialiser moves its own (master) capability to a CNode A into another CNode, it can no longer access CNode A to add more capabilities to it. Rather than defining a complex dependency ordering between the various moves of CNode capabilities, the initialiser duplicates all its CNode capabilities by copying them into its own CSpace. The master capabilities can then be moved where needed in the system's CNodes and the initialiser can still access all the CNodes using the duplicated capabilities.

This operation is done by the `duplicate_cnode_caps` function, which in turn calls the seL4 service `seL4_CNode_Copy` and updates the `capdl_to_sel4_copy` field of the state.

5.4. Object initialisation

Objects are created by the kernel in a default state. To make the system state match the desired capDL specification, the content of created objects must be initialised, which is done per object type: the virtual address spaces are first initialised, then the thread control blocks, and finally the capability spaces.

As explained in Section 2.1, the virtual address space of each thread in seL4 consists of a page directory containing either page tables or large frames. Each page table in turn contains a number of frames. The initialisation of virtual address spaces consists of mapping the required page tables and large frames into the page directories, and mapping the frames into the page tables, as directed by the capDL specification. The formalisation `init_vspace` is straightforward: it iterates (using the `mapM_x` function) over all the page directories specified in `spec`, then for each such page directory, iterates over all of its slots; for each page directory slot, it first maps the required page tables and large frames using the seL4 services `seL4_PageTable_Map` and `seL4_Page_Map` respectively. It then iterates over all the slots in each page table and maps the required framed, using `seL4_Page_Map` again.

Thread initialisation `init_tcbs` consists mainly of setting the VSpace root and CSpace root of all TCBs, as well as other information about the thread's context (such as the address of the IPC buffer for communication). This is done by iterating over the list of TCBs specified in `spec`:

```
init_tcbs spec = do tcbs ← get_tcbs spec;
                    mapM_x (init_tcb spec) tcbs
                 od
```

where each TCB is initialised using seL4 service `seL4_TCB_Configure`.

The most complex initialisation operation, which we also investigated most closely in the formal proof, is the one of capability spaces `init_cspace`. Capability spaces are initialised similarly to the virtual address spaces, with the complication of needing to distinguish between capabilities that are *moved* and the ones that are *copied*. Moreover, unlike virtual address spaces which are fixed, two-level data structures, capability spaces can be arbitrary directed acyclic graphs.

At this stage, the initialiser would have created all the CNodes objects in a default state and it now needs to fill them with the desired capabilities as specified in the capDL specification. This

is done one CNode at a time and, in each CNode, one slot at a time. For example, take the *spec* instance given in Figure 2.2 and Figure 2.5, in particular:

```
cdl_objects spec 3 ↦
  CNode ⦇ cdl_cnode_caps = [0 ↦ EndpointCap 4 badge {Read} ... ],
          cdl_cnode_size_bits = ... ... ⦈
```

Additionally, for the following assume we are initialising the slot number 0 of the CNode of reference 3, i.e. we want the newly created CNode at the address corresponding to the reference 3 to contain, at its slot number 0, the capability to the already created endpoint at the address corresponding to reference 4.

First we need to find, in the root task's CSpace, the capability to the actual newly created CNode object. We need to look for the duplicated capability in case the original capability to this CNode has already been moved. We use the *capdl_to_sel4_copy* field of the state to get where this duplicated capability is stored in the root task's CSpace. Then we need to find, in the root task's CSpace, the capability to the targeted object of the capability (here the endpoint of reference 4). We need to know if this capability should be copied or moved into the CNode. This is indicated in the specification by a mechanism (not explained here) which specifies what is the *parent* of each capability (i.e. what is the capability it has been derived from). If a capability has no parent, then it is the original capability. In that case, the initialiser needs to move its own original capability to the object into the CNode. Of course, the specification might require that some other CNode also contain that same capability. It can not be the original anymore, so a copy will be enough. It is technically easier for the initialiser to locate and copy its own original capability. This means that it should not have moved it yet. For these reasons, the *init_cspace* function proceeds in two steps: first it initialises the CNodes with the capabilities that need to be copied, and only afterwards it handles the ones that need to be moved:

```
init_cspace spec =
do cnodes ← get_cnodes spec;
   mapM_x (init_cnode spec Copy) cnodes;
   mapM_x (init_cnode spec Move) cnodes
od
```

where each CNode is initialised by iterating over its slots:

```
init_cnode spec mode cnode =
do cnode_slots ← return $ slots_of_list spec cnode;
   mapM_x (init_cnode_slot spec mode cnode) cnode_slots
od
```

Now each slot is initialised as explained above. The formalisation is the following, explained just after.

```
init_cnode_slot spec mode dest dest_slot ≡
do src_cap ← assert_opt $ opt_cap (dest, dest_slot) spec;
   assert (¬ is_untyped_cap src_cap);
   src ← return $ cap_object src_cap;
   dest_obj ← get_spec_object spec dest;
```

```
    dest_size ← return $ object_size_bits dest_obj;
    rights ← return $ cap_rights src_cap;
    orig ← return $ is_none $ opt_parent (dest, dest_slot) spec;
    data ← return $ cap_data src_cap;
    dest_root ← get_sel4_cap dest DerivedCap;
    dest_index ← return $ dest_slot;
    dest_depth ← return $ dest_size;
    src_root ← return seL4_CapInitThreadCNode;
    src_index ← get_sel4_cap src OrigCap;
    src_depth ← return 0x20;
    if src_cap = NullCap then return True
    else if mode = Move ∧ orig
         then seL4_CNode_Mutate dest_root dest_index dest_depth src_root
               src_index src_depth data
         else if mode = Copy ∧ ¬ orig
              then seL4_CNode_Mint dest_root dest_index dest_depth src_root
                    src_index src_depth rights data
              else return True
od
```

We explain the formalisation starting from the end. The move or copy is done using the seL4 API call `seL4_CNode_Mutate` and `seL4_CNode_Mint` respectively. These functions take information (root, index and depth) about the destination (the CNode which the capability should be moved/copied into), and about the source (the capability that needs to be moved/copied), as well as some data (for endpoints, this would be badges, for CNodes, that would be a potential guard). In the case of a copy, the specification might require that the capability only confer some specific rights (instead of the maximum rights the original capability confers). Therefore `seL4_CNode_Mint` takes rights as an extra argument.

In our case, the destination information (`dest_root`, `dest_index` and `dest_depth`) is computed from the (derived) capability that the root tasks holds for the CNode identified by `dest` in the specification. The source information (`src_root`, `src_index` and `src_depth`) is computed from the (original) capability that the root tasks holds for the object identified by `src`, where `src` is the target object of the capability `src_cap` that `spec` requires to be in CNode `dest` at slot `dest_slot`. The `rights` and `data` are also computed from what `src_cap` requires in `spec`.

The last action of the initialiser is to set each thread state to runnable. It then becomes dormant in memory, isolated from the rest of the system.

6. Separation Logic

When reasoning about the system initialiser specification, we wish to precisely capture the semantics of each operation. Axiomatic semantics are commonly represented in *Hoare logic*, using *Hoare triples*. These triples, written $\{\!|P|\!\}\ f\ \{\!|Q|\!\}$ express that if P is true before an operation f, then Q will be true afterwards.[1]

Many of the operations in the initialiser specification either iterate over a list of objects, or over the capability slots of an object. Expressing that an operation affects only a single capability slot, and chaining such operations together is difficult in traditional Hoare logic. For instance, to express the fact that a loop correctly initialises all the thread control blocks, we need to express that the initialisation of each TCB is correct *and that it does not affect the other TCBs*. We also need to express that the composition of these operations performs as expected. Similarly, when reasoning about the initialisation of specific capability slots *within an object*, we need to express that the capability slots are correctly initialised and that the other capability slots in the object are unaffected. Such reasoning in Hoare logic requires explicitly description of the unmodified parts of the state.

Separation logic — an extension to Hoare logic first introduced by Reynolds [8] — is designed for modular reasoning about operations that only manipulate portions of the state. In this work we have defined a separation logic to reason independently about various objects of a system state and various parts within objects. This separation logic is an instantiation of an abstract separation algebra we developed [3]. We partition all the objects of the logical abstract state — the memory heap — into smaller disjoints heaps, to allow us to specify that an operation affects only a portion of the memory heap. We also allow partitioning within objects to specify that an operation affects only a particular slot of an object and leaves other slots and fields unchanged. For example, we can partition a thread control block into three portions, as shown in Figure 6.1. This logical partitioning allows modular reasoning about the initialisation specification, that is we can effectively initialise each part separately.

Formally, we express that P and Q are true for separate parts of a heap by separation conjunction $P \wedge^* Q$.[2] We also define the lifted separation conjunction operator $\bigwedge^* Ps$ which joins a list of separation predicates Ps with separation conjunction (i.e. $\bigwedge^* [P1, P2] = P1 \wedge^* P1$).

This definition of separation conjunction allows us to prove concise rules about loops, such as the following rule about $mapM\_x$, which runs the operation f on each object in the list xs.

$$(\bigwedge R\ x.\ x \in xs \Longrightarrow \{\!|«P\ x \wedge^* R»|\!\}\ f\ x\ \{\!|«Q\ x \wedge^* R»|\!\}) \Longrightarrow$$
$$\{\!|«\bigwedge^* map\ P\ xs \wedge^* R»|\!\}\ mapM\_x\ f\ xs\ \{\!|«\bigwedge^* map\ Q\ xs \wedge^* R»|\!\}$$

[1] Hoare logic is normally used to prove partial correctness — if f terminates, then Q will be true. Termination can be proven separately, if desired.

[2] $P \wedge^* Q$ is defined as $(P \wedge^* Q)\ h = (\exists p\ q.\ p\ \#\#\ q \wedge h = p + q \wedge P\ p \wedge Q\ q)$, that is, there exists two disjoint memory heaps, p and q, that add to give h, and for which P is true of p and Q is true of q.

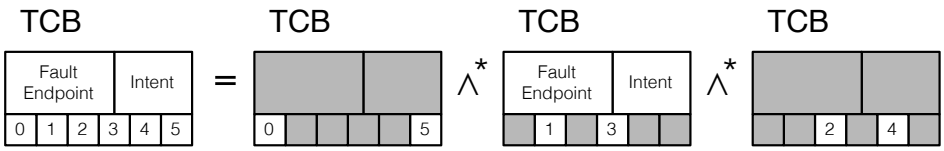

Figure 6.1.: The decomposition of a thread control block into three parts, one containing capability slots 0 and 5, another containing the fields (the fault endpoint and intent) and capability slots 1 and 3 and another containing capability slots 2 and 4.

The arbitrary R in this rule specifies that any predicate on the rest of the state is preserved, which means that the rest of the heap must be unchanged.

The state used in separation logic predicates has two parts, the memory heap containing the objects, and an additional *component map*, which states which parts of an object are "owned" by an object (and which is used in the definition of the heap partitioning). To apply separation predicates to the kernel state we use the syntax $<P>$, and to apply separation predicates to the initialiser state we use the syntax «P».

By allowing partitions within objects, we can specify that an operation like `set_cap` changes only a single slot, and leaves everything else (all other objects and all other slots) unchanged. We express this property using the following Hoare triple.

$$\{\!|<ptr \mapsto_c old\_cap \wedge^* R>|\!\}\ set\_cap\ ptr\ cap\ \{\!|<ptr \mapsto_c cap \wedge^* R>|\!\}$$

We have defined predicates that express when an object identifier maps to an object ($ptr \mapsto_o object$), when a capability reference maps to a capability in an object ($cap\_ref \mapsto_c cap$), and also when an object identifier maps to an object containing just slots and no fields ($ptr \mapsto_S obj$), fields and no slots ($ptr \mapsto_f obj$), or just a single slot ($cap\_ref \mapsto_s obj$). This allows us to decompose the reasoning about the initialisation of an object into reasoning about initialising the capabilities for an object separately from initialising the fields of an object.

We use this separation logic to reason about the semantics of the initialiser specification and about the capDL model of seL4 itself. We proved the above rule for `set_cap` about the seL4 capDL model and other leaf functions of the kernel, and used these to prove concise semantics for the behaviour of the seL4 kernel operations in specific contexts. These semantics, which are preserved by refinement and thus hold for the seL4 C code by our previous functional correctness proof for seL4, are used when proving properties about the initialiser specification.

6.1. Separation State

The separation algebra we use for these proofs is defined on a separation state that is lifted from the capDL state.

Since in our previous formalisation, objects are defined in Isabelle as records and datatypes, and since Isabelle does not allow a natural definition of "partial" records, we encode the partitioning of objects by associating an additional ownership set with each object. The objects can then be

divided into their individual capability slots, and their fields.[3] This ownership set can consist of capability slots (which are natural numbers) and/or fields. We encode this using the type `cdl_component`.

datatype `cdl_component = Slot nat | Fields`

This object ownership is added to our previous seL4 capDL model as *ghost state*.[4]

We define the separation state, which is a projection from the capDL kernel state, as an object heap and a function of object ownerships.

datatype `sep_state` = SepState `cdl_heap cdl_component_map`

where `cdl_heap` = `cdl_object_id ⇒ cdl_object option`

and `cdl_component_map` = `cdl_object_id ⇒ cdl_component set`

6.2. Definition of separation state addition and disjunction

To create a separation algebra, we define state addition and state disjunction, and show that these definitions obey the axioms of our abstract separation algebra. Two states are defined as being disjoint, if, wherever they both have objects in the same address, these objects have the same type and the corresponding ownerships are neither empty nor colliding. Empty ownerships for an object are not allowed because they would allow the construction of a predicate which constrains the type, but does not consume ownership, which would preclude the construction of a predicate specifying total ownership of an object, as is needed to retype an object. Thus, we do not allow such predicates by making them always disjoint from any other predicate.

State addition is defined by the addition of object heap and the addition of object ownerships. Object addition adds the fields and capabilities owned by the two objects (using a right-override[5]). Any unowned fields and capabilities are reset to a default value, thus always given a "cleaned" object. This is necessary to ensure commutativity of addition. Ownership addition is defined as the union of the ownerships of the objects.

Using these definitions of state addition and state disjunction we instantiate our abstract separation algebra, and show that these definitions obey the axioms of the algebra. This gives us a separation algebra for capDL.

[3] Even though the framework would be general enough to handle this case, we do not decompose the different fields of an object, since it is not necessary in this proof.

[4] Ghost state are variables that are only written to, and can be removed without changing the operation of a program. They are useful for referring to intermediate state properties in annotations.

[5] Right-override is associative and commutative when the objects are disjoint, as required by the abstract separation algebra axioms described in [3].

7. Proof

In this section, we describe the verification framework we have developed for reasoning about the system initialiser, as well as describe the proof outcomes so far.

We explain the top-level correctness theorem of the system initialiser in Section 7.1, the wellformedness assumptions on the input specification in Section 7.2, and the invariants needed in Section 7.3.

Section 7.4 explains how we use separation algebra for this proof, and Section 7.5 explains how we use it to describe predicates about separate parts of objects.

Finally, we describe how we subdivide the proof of the initialiser in Section 7.6, how we specify that the system initialiser sets up capabilities correctly in Section 7.7, and go into detail about the proof of `init_cspace` and local reasoning about loops in Section 7.8.

These proofs of the initialiser are built on and connect to previous work in which we formally proved that the capDL kernel model is a correct abstraction of the seL4 C code. This means, the model and proof presented here are firmly grounded on the real seL4 system call API.

7.1. Top-Level Theorem

The aim of this proof is a theorem about the initialiser specification stating that, at the end of the initialisation, all objects in the system either existed at the start (those for the root task itself, created during kernel booting), or that they are in conformance with the capDL specification.

In capDL specifications, systems are described as a mapping from object identifiers (names) to objects. We treat these object identifiers simply as identifiers, rather than as memory addresses. When objects are created, their location in memory is decided by an allocator that runs at system initialisation. The initialiser ensures that there is an injective function φ between the identifiers of the objects in the capDL specification and the memory addresses of corresponding objects in the initialised system. In other words, no two objects are mapped to the same memory address. The use of an online allocator allows more adaptability to hardware and specification changes. However, it also means that the capDL specification cannot specify the memory addresses for objects. The only time where physical memory addresses are relevant for user-level execution is when mapping device frames into memory and for DMA. The verified version of seL4 currently does not support DMA, and provides special pre-allocated pages for known devices, so this no new restriction. In a potential extension of the seL4 kernel that supports DMA and memory allocation by physical address, this kernel mechanism could be used by the system initialiser to enable mapping physical memory precisely when desired. However, this is presently outside the scope of this work.

To reason about the relation between the capDL specification and the concrete kernel state, the injection between object identifiers and memory addresses maps the object graph of the capDL

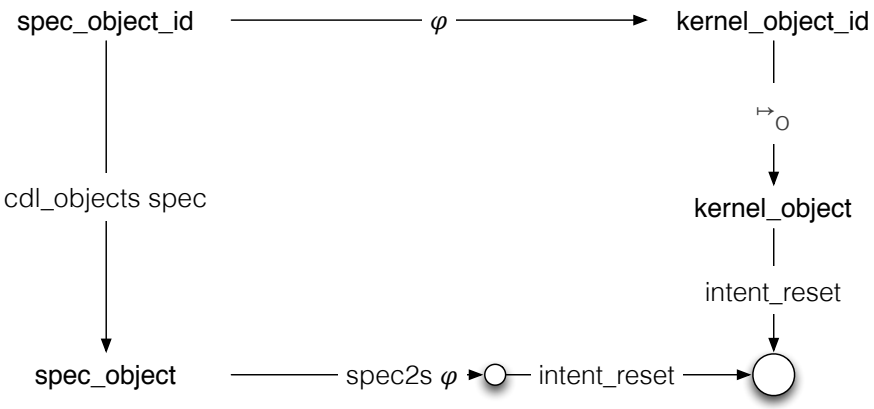

Figure 7.1.: Diagrammatic representation of `object_done` definition.

specification to the one of the kernel state. Technically, such a mapping is called an injective homomorphism, or monomorphism. This monomorphism is used to map all object identifiers in the capDL specification, as well as all object identifiers within the capabilities of each object — the latter defined by the function `spec2s` (which takes the injection φ as a parameter). Each of the objects in the capDL specification will be created at a particular memory address and the capabilities in the kernel will refer to the memory addresses of the corresponding objects, not the identifiers used in the capDL specification.

To be able to phrase the top-level correctness property, we define the function `object_done` that indicates that a given object in the capDL specification has been created, and that it corresponds to the respective object in the kernel state as illustrated in Figure 7.1. The formal definition is the following.

```
object_done spec φ spec_object_id ≡
λs. ∃kernel_object_id kernel_object spec_object.
       φ spec_object_id = Some kernel_object_id ∧
       (kernel_object_id ↦o kernel_object) s ∧
       cdl_objects spec spec_object_id = Some spec_object ∧
       intent_reset (spec2s φ spec_object) = intent_reset kernel_object
```

It can be read as: the object identified by `spec_object_id` in the capDL specification `spec` is said to be *done* in state `s` if there exists a corresponding `kernel_object_id` mapped to `spec_object_id` via the injection φ such that the object pointed to by `spec_object_id` is (almost) the same as the one pointed to by `kernel_object_id`. It is only *almost* the same because each thread control block contains an *intent* field in the specification, used to encode the kernel instruction that the thread wishes to execute, and which our proof does not make any guarantee about its content. The intent is therefore overwritten to a default value through `intent_reset` before comparing objects.

An alternative to the `intent_reset` formulation in this definition would be to include the reset into the projection from the kernel state into the separation logic state. Under the additional

(reasonable) assumption that the initial specification has all intents set to default, the statement of the top-level theorem could be simplified at the cost of making the proof framework slightly more complex. Our initial experiments in this direction have been encouraging, but are not completed yet.

With either version of this definition, the top-level result of the system initialiser is:

Proposition.

If `well_formed spec` *and* `injective φ` *and*
`object_ids = dom (cdl_objects spec)` *then*

```
⦃«root_objects»⦄ init_system spec
⦃«⋀* map (object_done spec φ) object_ids ∧* root_objects»⦄
```

It states that each object of the capDL specification has been correctly initialised, and that the other objects created by the kernel booting remain unchanged.

7.2. Assumptions on the capDL Specification

This section explains in more detail the assumptions of the top-level theorem, in particular the wellformedness constraints on input specifications.

The capDL language allows the description of infeasible systems, for instance systems with objects having infinite number of capability slots, or more slots than the object size allows. The initialiser can obviously not configure systems described by such capDL specifications. The proof of correctness therefore assumes that the capDL specification is `well_formed`. In addition to these obvious restrictions, the proof has extracted a number of more subtle constraints on system architectures that are formalised in this section.

In addition to these necessary constraints on the input specification, we have additional assumptions that express current limitations of the system initialiser, such as not allowing objects to access untyped memory. We aim to remove these in future work.

We formalise all of these constraints with the predicates `well_formed`, `well_formed_caps` and `well_formed_cap`.

```
well_formed spec ≡
∀ obj_id.
   case cdl_objects spec obj_id of None ⇒ True
   | Some obj ⇒
       well_formed_caps spec obj ∧
       object_size_bits obj < 32 ∧
       object_size_bits (object_default_state obj) = object_size_bits obj ∧
       dom (object_slots (object_default_state obj)) =
       dom (object_slots obj) ∧
       (is_cnode obj ⟶ 0 < object_size_bits obj)
```

This predicate says: Objects must have well formed capabilities, they must be less than 32 bits in size (as the verified version of seL4 runs on a 32 bit system), the size must match the default

size of the object (or positive for CNodes, which do not have a default size)[1], and the domain of the capability map must be the same as the default for the type of object. The capability map for an object is a partial mapping, and objects can have *real capabilities*, `NullCap`s, or `None` in each slot, so the domains of the capability maps mean that an object's capabilities will only occur in the correct slots. The capDL language distinguishes between real capabilities which directly correspond to seL4 API capabilities, and virtual capDL capabilities that are used to represent implicit authority or virtual memory mappings.

```
well_formed_caps spec obj ≡
∀slot.
   slot ∈ dom (object_slots obj) ⟶
   (∃cap_obj cap.
       object_slots obj slot = Some cap ∧
       (cap ≠ NullCap ⟶
        well_formed_cap cap ∧
        ¬ is_untyped_cap cap ∧
        cdl_objects spec (cap_object cap) = Some cap_obj ∧
        cap_type cap = object_type cap_obj ∧
        (is_cnode obj ⟶ ¬ vm_cap_has_asid cap ∧ ¬ is_fake_vm_cap cap)))
```

Well formed capabilities must be of the same type as the object they point to. As mentioned, capDL distinguishes between the "real" seL4 API capabilities for virtual memory that are installed in a capability space, and artificial capabilities that denote the presence of virtual memory mappings in the hardware. All capabilities in a capability space must be such real capabilities, and cannot have a hardware ASID associated with them yet. ASID stands for *address space identifier* and is a hardware feature that the kernel manages for user-space. Each virtual memory mapping, when it becomes active, must be mapped to an ASID. The restriction basically expresses that the pages described in the capDL state must not already be previously mapped.

```
well_formed_cap cap ≡ case cap of
    UntypedCap _              ⇒ True
  | EndpointCap _ b _         ⇒ b < 2 ^ badge_bits
  | AsyncEndpointCap _ b r    ⇒ (b < 2 ^ badge_bits) ∧
                                (r ⊆ {AllowRead, AllowWrite})
  | CNodeCap _ g gs           ⇒ (gs < guard_bits) ∧ (g < 2 ^ gs)
  | TcbCap _                  ⇒ True
  | FrameCap _ r _ _ _        ⇒ r ∈ {vm_read_write, vm_read_only}
  | PageTableCap  _ _ _       ⇒ True
  | PageDirectoryCap _ _ _    ⇒ True
  | _                         ⇒ False
```

Each type of capability has particular constraints, which are requirements of seL4 (or the underlying hardware) — the badges of endpoint and asynchronous endpoints cannot be too large, asynchronous endpoints cannot have grant rights (as they cannot transfer capabilities), a CNode's guard cannot be bigger than a predefined size and the guard must be smaller than the $2^{guardsize}$ and frames must be read-only or read-write. At present, the system initialiser is unable to initialise hardware IRQ or ASID control capabilities. The latter two are intended for trusted

[1] If `obj` is a CNode, then `object_size_bits (object_default_state obj) = object_size_bits obj` is always true.

supervisor components only. Since the system initialiser is intended for setting up a static system configuration, this is not a practical restriction. Nevertheless, we think this constraint can be lifted in future work to enable the initialisation of dynamic systems with trusted supervisor components.

7.3. Invariants

This section starts going into the proof of the top-level theorem itself and describes the two main global invariants that are needed for the proof. The invariant demands

- that the root task is the only thread that runs, and
- that the capabilities held by the root task point to the correct kernel objects.

While these are reasonably lightweight invariants to prove, we eventually aim to eliminate the need for global invariants altogether to reduce proof effort. Using separation logic, we believe we can achieve this and use local, compositional reasoning instead.

The latter invariant we were already able in an experiment to describe using separation logic predicates as described in Section 7.7, the former we believe can be captured by the separation algebra itself as described in Section 7.4, by extending the separation logic state to include the scheduling data structures of the kernel.

Eliminating these global invariants in favour of local reasoning would simplify the proof further.

7.4. Local reasoning

This section describes the key reasoning feature that makes the proof tractable: local reasoning.

To prove results about the capDL model of seL4 using the separation logic we defined in Section 6, we prove what is known as the *frame rule* explicitly for the leaf functions of the kernel. Because we use a shallow embedding of capDL within Isabelle, we need to prove this frame rule manually for each leaf function of the kernel, but can then use these for proving similar rules for more complicated functions.

The frame rule states that the effects of a function are local, i.e. that the effects of a function are constrained to the domain of the separation logic predicate in the precondition. If that is the case, then any predicate that is separate from this precondition will remain unaffected by the function. We can use this to implicitly state which parts of the state remain invariant without naming these parts of the state explicitly each time. The following shows an example of this technique.

By allowing partitions within objects, we can specify that an operation such as `set_cap` changes only a single capability slot, and leaves everything else (all other objects and all other slots) unchanged. We express this property using the following Hoare triple.

$$\{\!|\langle ptr \mapsto_c old\_cap \wedge^* R\rangle|\!\}\ set\_cap\ ptr\ cap\ \{\!|\langle ptr \mapsto_c cap \wedge^* R\rangle|\!\}$$

The arbitrary R in this rule specifies that any predicate on the rest of the state is preserved, which means that the rest of the heap must be unchanged. In standard separation logic, there is one frame

rule that applies to all functions. In a shallow embedding such as ours, we bake the statement of the frame rule into each basic hoare triple by attaching an arbitrary R as above.

We proved this implicit frame rule for the the leaf functions of the kernel, i.e. for setting of an object's slots or fields, object creation and object deletion. These rules depend on the exact instantiation of the separation algebra, and were used to motivate the definition of the separation algebra instantiation for this proof.

At present, the only remaining non-local operation in the kernel is the scheduling of threads in the system. This is because the scheduler can globally choose the next thread and set its state to running, i.e. in this setting scheduling appears as an inherently global operation. However, we believe we will be able to modify the capDL specification slightly together with the separation algebra instantiation to model the same behaviour in a local way by making the set of runnable threads an explicit object in the system. This change will of course require an update to the existing functional correctness proof to re-establish the connection to the C code of seL4.

We have used a similar technique to make another behaviour of the seL4 capDL model local: at various point in the kernel, the so-called intent of a thread is reset to a nondeterministic value. The *intent* models the system call a thread is about to make. Certain operations the kernel can perform, such as changing registers in a thread, can change this intent. On the abstraction level of capDL, we do not model what the precise effect of such a register change is. Instead, we demand that such operations are only executed when safe.

7.5. Object predicate decomposition

To make use of local reasoning as introduced in the previous section, we need to be able to decompose properties of objects into small pieces that single operations transform locally. This section describes such predicates.

As described in Section 6, we have predicates that specify that an specific object exists in the state at a particular position ($ptr \mapsto_o object$).

The separation algebra we defined allows us to decompose this predicate into predicates describing partitions of objects. We can divide a predicate about an object into two predicates about the fields and the capability slots of an object.

$$p \mapsto_o obj = (p \mapsto_f obj \wedge^* p \mapsto_S obj)$$

Each of these object predicates specify that the separation state consists of an object with the specified properties (and that the unspecified parts of the object are in a default state), and the state ownership correspond the the ownerships of the particular predicate. Thus, $(ptr \mapsto_f object)\ s$ specifies that the object heap of s consists of an object with fields the same as $object$ at address ptr, the capabilities of the object in the heap has an empty capability map, and that the state ownership is that this object owns its fields (and nothing else).

Since, in capDL, an object's capability slots are a partial mapping from slots to capabilities, an object can have a theoretical infinite number of slots. Whilst the type does not constrain the number of capability slots of an object, the domain of the capability map of a wellformed object is determined by the size and type of the object. This wellformedness constraint is described in detail in Section 7.2. Such wellformed objects will have either *real* capabilities or $NullCap$s

in the expected slots, and no caps in the other slots. We use this to define the ownership of the empty slots of an object (*ptr ↦E object*) which is used for the decomposition of the predicate describing the slots of an objects.

```
⟦dom (object_slots obj) = slots; distinct slots; slots ≠ UNIV⟧
⟹ obj_id ↦S obj =
    (⋀* map (λslot. (obj_id, slot) ↦s obj) slots ∧* obj_id ↦E obj)
```

These predicate decompositions are used to decompose the predicate defined in Section 7.6 that defines when an object is correctly initialised.

The difficulty in setting up the framework is to find the sweet spot between local behaviour of the kernel, the separation algebra instantiation, and simple object decomposition rules as above.

7.6. Decomposition of the Final Theorem

This section describes how the predicates defined above can be used to decompose the top-level correctness theorem of the system initialiser into its parts.

The separation algebra we defined in Section 6 has two main properties that allow the proof of the system initialiser. Firstly, as explained in Section 7.4, because it is a separation algebra, we are able to reason separately that the different phases turn a particular type of object (such as virtual memory objects, or capability space objects), from being in an empty, default state (*object_empty*), to being in an initialised state (*object_done*). These results, once proven, compose in the obvious manner.

The second advantage of this particular separation algebra, is the decomposability of the predicates describing the initialisation of an object (such as *object_empty* and *object_done*) into predicates such as when the object's fields are done (*object_fields_done*), or when an object slot is done (*object_slot_done*). This proof is built on the decomposition of the object predicates such as *ptr ↦o object* as explained in Section 7.5. As *p ↦o obj = (p ↦f obj ∧* p ↦S obj)*, the property *object_done* can be similarly decomposed into stating that an object's fields are correctly initialised, and that its capability slots are correctly initialised.

```
object_done spec spec2s_ids spec_object_id =
(object_fields_done spec spec2s_ids spec_object_id ∧*
 object_slots_done spec spec2s_ids spec_object_id)
```

The definition that an object's capability slots are initialised can be similarly decomposed into stating that each individual capability slot is initialised.

```
⟦dom (slots_of obj_id spec) = slots; distinct slots; slots ≠ UNIV⟧
⟹ object_slots_done spec φ obj_id =
    (⋀* map (object_slot_done spec φ obj_id) slots ∧*
     object_empty_slots_done spec φ obj_id)
```

Such a decomposition is generally a difficult result to prove and not necessarily true, but our separation algebra gives us this result with relative ease. To prove this lemma, we observe that the definition of *object_slots_done* specifies that there exists *kernel_object_id*,

kernel_object and *spec_object* satisfying a particular property. The left hand side contains only one *kernel_object_id*, *kernel_object* and *spec_object*, but the right hand side folds the predicate *object_slot_done* over a list of slots, and thus we have a list of potentially different *kernel_object_id*, *kernel_object* and *spec_object* satisfying the right hand side.

Lifting *kernel_object_id* and *spec_object* outside the folding is simple because there can be only one given a constant φ and *spec*. We can lift the *kernel_object* outside the fold by noting that

$$((obj\_id,\ slot_1) \mapsto s\ obj_1 \wedge^*\ (obj\_id,\ slot_2) \mapsto s\ obj_2)\ s \Longrightarrow$$
$$\exists\, obj.\ ((obj\_id,\ slot_1) \mapsto s\ obj \wedge^*\ (obj\_id,\ slot_2) \mapsto s\ obj)\ s$$

We use this decomposition of the predicates *object_empty* and *object_done* in Section 7.8 to decompose the proof that *init_cspace* correctly initialises the capability spaces of the objects from the specification.

7.7. Initialiser's CSpace

This section prepares the proof of connecting one of the initialiser operations to the kernel API level. A key specification problem in this proof is to formalise the contents of the initialiser data structures and how they relate to each other.

After the initialiser has created all of the objects required in the second phase of operation, it stores the master capabilities to each of these objects within its own capability space, saving the information about which capability slot contains the master capability to a particular object in its own data structure *opt_sel4_cap*. As described in Section 5, the initialiser keeps a duplicate copy of CNode capabilities for its own addressing. The information about these *OrigCap*s and *DerivedCap*s are both stored in the data structure *opt_sel4_cap*.

To specify that the initialiser has the necessary capabilities stored in the correct slots (as specified by the initialiser's internal bookkeeping in *opt_sel4_cap*), we define the predicate *seL4_cap_at*, which takes an injection φ mapping object identifiers to memory addresses, a mapping from object identifiers to cap pointers (to be taken from *opt_sel4_cap*), and the object identifier of the CNode being initialised (*obj_id*), and specifies that a capability exists in the root task's capability space at the specified slot points to the CNode being initialised.

$$\begin{aligned}
&seL4_cap_at\ \varphi\ seL4_caps\ spec\ obj_id \equiv \\
&\lambda s.\ \exists\, cap_ptr\ slot\ obj\ kobj_id. \\
&\qquad ((root_cnode_id,\ slot) \mapsto c\ default_cap\ (object_type\ obj)\ \{kobj_id\})\ s \wedge \\
&\qquad seL4_caps\ obj_id = Some\ cap_ptr \wedge \\
&\qquad unat\ cap_ptr = slot \wedge \\
&\qquad cap_ptr < 2^{root_cnode_size} \wedge \\
&\qquad cdl_objects\ spec\ obj_id = Some\ obj \wedge \varphi\ obj_id = Some\ kobj_id
\end{aligned}$$

A similar predicate *seL4_spec_obj_caps_at* specifies that for a given CNode, the capability required to initialise the specified slot exists in the root capability space at the specified location.

```
seL4_spec_obj_cap_at φ seL4_caps spec obj_id slot ≡
λs. ∃spec_cap.
       seL4_cap_at φ seL4_caps spec (cap_object spec_cap) s ∧
       opt_cap (obj_id, slot) spec = Some spec_cap ∧
       ¬ is_untyped_cap spec_cap
```

We define a predicate `seL4_null_cap_at` to describe the empty capability slots of the root capability space after the capabilities have been moved into the required CNodes.

We lift the predicates `seL4_cap_at` and `seL4_null_cap_at` to work on lists of objects, `seL4_spec_obj_caps_at` to work on both objects, and lists of objects.

7.8. Initialising the capability spaces

This section looks at connecting one of the operations in the initialiser specification to the seL4 kernel API formalisation. It draws together the results of the sections above.

The most complicated objects to initialise are the capability storage objects, because they can be arranged in a graph, potentially with loops, and because they are initialised in a two phases process to ensure the distinction between *original* capabilities and *derived* capabilities as described in Section 5.

Due to its complexity, this operation of the initialiser is the one we have investigated in detail, as its proof should ensure that the proof framework developed is sufficient to express and prove the desired properties for the entire initialiser. We believe that the proof of the initialisation of other objects will be similar to the initialisation of capability spaces, only simpler.

As mentioned, the initialisation of capability spaces is divided into two phases — copying the capabilities that should be *derived* capabilities first, then moving the capabilities that should be *master* capabilities second.

```
init_cspace spec =
do cnodes ← get_cnodes spec;
   mapM_x (init_cnode spec Copy) cnodes;
   mapM_x (init_cnode spec Move) cnodes
od
```

The detailed pre/post specification of the copy phase of `init_cspace` is the following.

```
⦃λs. «⋀* map (object_empty spec t) objects ∧*
      ⋀* map (seL4_spec_obj_caps_at t (S OrigCap) spec) objects ∧*
      ⋀* map (seL4_cap_at t (S DerivedCap) spec) objects ∧*
      root_cspace ∧* R»
      s ∧
     S = (λtype obj_id. opt_sel4_cap obj_id type s) ∧
     well_formed spec ∧ objects = cnodes spec s⦄
mapM_x (init_cnode spec Copy) objects
⦃λs. «⋀* map (cnode_half_done spec t) objects ∧*
      ⋀* map (seL4_spec_obj_caps_at t (S OrigCap) spec) objects ∧*
      ⋀* map (seL4_cap_at t (S DerivedCap) spec) objects ∧*
      root_cspace ∧* R»
      s ∧
```

```
S = (λtype obj_id. opt_sel4_cap obj_id type s) ∧
well_formed spec ∧ objects = cnodes spec s⦄
```

This theorem states that iterating *init_cnode* over each of the CNodes of the specification *spec* transforms each of these CNodes from being in a default state, to being half-initialised (that is, having the derived capabilities, but not the master capabilities required).

For this to be true, the initialiser needs all of the original master capabilities required to copy into the CNodes (*seL4_cap_at*), and derived capabilities to address the CNodes themselves (*seL4_spec_obj_caps_at*). These lookups require the data structure *opt_sel4_cap* which is taken from the initialiser's state, and this theorem proves that the data structure remains unchanged by the operation.

Proving this property requires reasoning about complex loops. Thankfully, the separation algebra makes reasoning about such loops possible using rules such as the follow, which work for iteration over objects, and iteration over the capability slots of an object.

```
(⋀R t x.
    x ∈ xs ⟹
    ⦃λs. «P t x ∧* I s ∧* R» s ∧ t = g s⦄ f x
    ⦃λs. «Q t x ∧* I s ∧* R» s ∧ t = g s⦄) ⟹
⦃λs. «⋀* map (P t) xs ∧* I s ∧* R» s ∧ t = g s⦄ mapM_x f xs
⦃λs. «⋀* map (Q t) xs ∧* I s ∧* R» s ∧ t = g s⦄
```

Recall that *init_cspace* iterates the function *init_cnode* that initialises each signle CNode, which in turn iterates *init_cnode_slot* which initialises a single slot in a CNode by extracting and transforming the necessary information from the capDL specification of the desired system state and then calling the seL4 API function *seL4_CNode_Mint*.

The proof of *init_cspace* simply reduces the problem using the *mapM_x* rule above first to a single CNode initialisation, and then to a single slot initialisation. Due to the compositionality of this rule and with the additional help of the decomposition rules of Section 7.6, this doubly nested loop that would otherwise need complex invariant specifications becomes easily tractable. The decomposition rules and object predicates are designed such that the decomposition happens precisely along the same boundaries.

To connect the proof to the low-level seL4 API function, we then transform our lemma about the initialisation of a CNode slot into a specification of the state required for the seL4 API call *seL4_CNode_Mint* by expanding the definitions of *seL4_cap_at* and *seL4_spec_obj_caps_at* and observing that these provide sufficient information for local reasoning about the kernel call. This local reasoning is what makes the complexity reduction in *mapM_x* rule possible.

In total, this gives us a formal link between the system initialisation specification and the functional correctness proofs of seL4.

8. Conclusion and Future Work

This report has presented the outcomes so far of our work on proving correctness of user-level system initialisation.

The critical task of initialising components and communication channels is tedious and error prone when conducted manually. This work addresses this task by providing a generic initialiser, able to automatically build seL4-based componentised systems from precise configuration descriptions, and by conducting a formal proof of specification correctness.

In this report, we have described in detail our formal Isabelle/HOL specification of the initialiser, which takes a capDL description of the desired initial system state as input and then iterates over this description to produce that desired system state. We have shown a general separation logic framework that can be used to reason about such user-level systems, we have produced a proof framework to reason about user-level executions on top of a formally verified microkernel API, and we have made significant progress towards a mechanised Isabelle/HOL proof of the correctness of the system initialiser specification.

In more detail, the outcomes of the project are the following.

- a complete formal specification of system initialisation;
- a separation algebra suitable for compositional reasoning about the semantics of both seL4 and the initialiser specification;
- the formal instantiation of the separation algebra to system-initialiser and kernel state;
- a formal definition of the correctness property required from the initialiser;
- a decomposition of the correctness property of the whole initialisation into properties about the various phases focusing on smaller parts of the system, including precise pre/post annotations for each phase, enabling compositional verification;
- a formal description of wellformed desired initial system states that is practically usable and can be automatically checked;
- global invariants on user and kernel state that hold during system initialisation, encoded in local separation logic reasoning;
- substantial progress on the correctness proof of the initialiser specification itself, both in breadth over the whole specification as well as in depth towards connecting with the kernel correctness proof.

While the full specification correctness proof has not been achieved in the time frame of this project, it is close enough to completion and has covered enough of the complexity of the specification to clearly demonstrate the feasibility of its completion in future work.

A side effect of the formalisation and proof effort was the explicit extraction of assumptions which initial system states can be safely achieved, and which state descriptions are malformed. This explicit formal wellformedness condition on the capDL description of initial system states embodies system-architecture assumptions the initialiser must make to successfully create the desired state. These wellformedness conditions are executable and can be automatically checked before the system initialiser runs. This means system architects without formal methods expertise can get design-time feedback on whether a system can be safely initialised or not.

The work in this project shows that the formal verification of user-level components on top of a microkernel API with a fully formal connection between kernel correctness and user-level specification is feasible.

Bibliography

[1] Jim Alves-Foss, Paul W. Oman, Carol Taylor, and Scott Harrison. The MILS architecture for high-assurance embedded systems. *Int. J. Emb. Syst.*, 2:239–247, 2006.

[2] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating system kernel. *CACM*, 53(6):107–115, Jun 2010.

[3] Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Mechanised separation algebra. In Lennart Beringer and Amy Felty, editors, *3rd ITP*, volume 7406 of *LNCS*, pages 332–337, Princeton, New Jersey, Aug 2012. Springer-Verlag.

[4] Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Separation algebra. *Archive of Formal Proofs*, May 2012. http://afp.sf.net/entries/Separation_Algebra.shtml.

[5] Ihor Kuz, Gerwin Klein, Corey Lewis, and Adam Walker. capDL: A language for describing capability-based systems. In *1st APSys*, pages 31–36, New Delhi, India, Aug 2010.

[6] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[7] Michael Norrish. C-to-Isabelle parser, version 0.7.2. http://ertos.nicta.com.au/software/c-parser/, Jan 2012.

[8] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[9] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *2nd ITP*, volume 6898 of *LNCS*, pages 325–340, Nijmegen, The Netherlands, Aug 2011. Springer-Verlag.

[10] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *34th POPL*, pages 97–108. ACM, 2007.

A. System Initialiser Specification

definition
```
  parse_bootinfo :: cdl_bootinfo ⇒ unit u_monad
```
where
```
  parse_bootinfo bootinfo ≡ do
    modify (λs. s (|capdl_to_sel4_orig := empty|));
    modify (λs. s (|capdl_to_sel4_copy := empty|));

    (untyped_start, untyped_end) ← return $ bi_untypes bootinfo;
    untyped_list ← return $ set [untyped_start .e. (untyped_end - 1)];
    ordered_untyped_list ← select {xs. set xs = untyped_list &
                                       distinct xs};
    modify (λs. s (|free_untypeds := ordered_untyped_list|));

    (free_slot_start, free_slot_end) ← return $ bi_free_slots bootinfo;
    modify (λs. s (|free_slots := [free_slot_start .e. (free_slot_end - 1)]|))
  od
```

definition
```
  parse_spec :: cdl_state ⇒ unit u_monad
```
where
```
  parse_spec spec ≡ do
    cdl_object_list ← select {xs. set xs = dom (cdl_objects spec) &
                                  distinct xs};
    modify (λs. s (|capdl_objects := cdl_object_list|))
  od
```

definition
```
  retype_untyped :: cdl_cptr ⇒ cdl_cptr ⇒ cdl_object_type ⇒
                    word32 ⇒ bool u_monad
```
where
```
  retype_untyped free_slot free_untyped type size_bits ≡ do
    root ← return seL4_CapInitThreadCNode;
    node_index  ← return (0::word32);
    node_depth  ← return (0::word32);
    node_offset ← return free_slot;

    no_objects_to_create ← return 1;

    seL4_Untyped_Retype free_untyped type size_bits root node_index
                        node_depth node_offset no_objects_to_create
  od
```

```
function
  retype_untypeds :: cdl_cptr ⇒ cdl_object_type ⇒ word32 ⇒ unit u_monad
where
  retype_untypeds free_slot type size_bits s = (do
    free_untyped ← get_free_untyped;
    success ← retype_untyped free_slot free_untyped type size_bits;
    if success
    then return ()
    else (do
            next_free_untyped;
            retype_untypeds free_slot type size_bits
          od)
    od) s
```

```
definition
  create_object :: cdl_state ⇒ cdl_object_id ⇒ unit u_monad
where
  create_object spec object_id = do
    free_slot   ← get_free_slot;
    object      ← get_spec_object spec object_id;
    object_type ← return $ object_type object;
    object_size ← return $ of_nat $ object_at_pointer_size_bits spec object_id;
    retype_untypeds free_slot object_type object_size;
    add_sel4_cap object_id OrigCap free_slot;
    next_free_slot
  od
```

```
definition create_objects :: cdl_state ⇒ unit u_monad
where
  create_objects spec = do
    objects ← get_capdl_objects;
    mapM_x (create_object spec) objects
  od
```

```
definition duplicate_cap :: cdl_state ⇒ cdl_object_id ⇒ unit u_monad
where
  duplicate_cap spec cap ≡ do
    rights     ← return $ UNIV;
    free_slot   ← get_free_slot;

    dest_root  ← return free_slot;
    dest_index ← return (0::word32);
    dest_depth ← return (0::word32);

    src_root   ← get_sel4_cap cap OrigCap;
    src_index  ← return (0::word32);
    src_depth  ← return (0::word32);

    success ← seL4_CNode_Copy dest_root dest_index dest_depth
```

```
                                src_root src_index src_depth rights;
    assert success;

    add_sel4_cap cap DerivedCap free_slot;
    next_free_slot
  od


definition duplicate_cnode_caps :: cdl_state ⇒ unit u_monad
where
  duplicate_cnode_caps spec = do
    cnodes ← get_cnodes spec;
    mapM_x (duplicate_cap spec) cnodes
  od



definition init_tcb :: cdl_state ⇒ cdl_object_id ⇒ unit u_monad
where
  init_tcb spec tcb = do
    cdl_tcb         ← assert_opt $ opt_thread tcb spec;
    cdl_cspace_root ← assert_opt $ opt_cap (tcb, tcb_cspace_slot) spec;
    cdl_vspace_root ← assert_opt $ opt_cap (tcb, tcb_vspace_slot) spec;
    cdl_ipcbuffer   ← assert_opt $ opt_cap (tcb, tcb_ipcbuffer_slot) spec;
    ipcbuf_addr     ← return $ tcb_ipc_buffer_address cdl_tcb;
    priority        ← return $ tcb_priority cdl_tcb;

    sel4_tcb         ← get_sel4_cap tcb OrigCap;
    sel4_cspace_root ← get_sel4_cap (cap_object cdl_cspace_root) OrigCap;
    sel4_vspace_root ← get_sel4_cap (cap_object cdl_vspace_root) OrigCap;
    sel4_ipcbuffer   ← get_sel4_cap (cap_object cdl_ipcbuffer) OrigCap;
    sel4_fault_ep    ← return $ cdl_tcb_fault_endpoint cdl_tcb;

    sel4_cspace_root_data ← return $ guard_as_rawdata cdl_cspace_root;
    sel4_vspace_root_data ← return 0;

    success ← seL4_TCB_Configure sel4_tcb sel4_fault_ep priority
                                 sel4_cspace_root sel4_cspace_root_data
                                 sel4_vspace_root sel4_vspace_root_data
                                 ipcbuf_addr sel4_ipcbuffer;
    assert success
  od


definition init_tcbs :: cdl_state ⇒ unit u_monad
where
  init_tcbs spec = do
    tcbs ← get_tcbs spec;
    mapM_x (init_tcb spec) tcbs
  od


definition set_asid :: cdl_state ⇒ cdl_object_id ⇒ bool u_monad
```

```
where
  set_asid spec page = do
    sel4_asid_pool ← return seL4_CapInitThreadASIDPool;
    sel4_page ← get_sel4_cap page OrigCap;
    seL4_ASIDPool_Assign sel4_asid_pool sel4_page
  od
```

```
definition map_page :: cdl_state ⇒ cdl_object_id ⇒ cdl_object_id
                       ⇒ cdl_right set ⇒ word32 ⇒ bool u_monad
where
  map_page spec page pd rights vaddr = do
    cdl_page  ← assert_opt $ opt_object page spec;
    sel4_page ← get_sel4_cap page OrigCap;
    sel4_pd   ← get_sel4_cap pd OrigCap;

    vmattribs ← assert_opt $ opt_vmattribs cdl_page;

    if (pt_at page spec) then
       seL4_PageTable_Map sel4_page sel4_pd vaddr vmattribs
    else if (frame_at page spec) then
      seL4_Page_Map sel4_page sel4_pd vaddr rights vmattribs
    else
      return False
  od
```

```
definition map_page_table_slot :: cdl_state ⇒ cdl_object_id ⇒ cdl_object_id
                                  ⇒ word32 ⇒ cdl_cnode_index ⇒ bool u_monad
where
  map_page_table_slot spec pd pt pt_vaddr pt_slot = do
    page_cap     ← assert_opt $ opt_cap (pt, pt_slot) spec;
    page         ← return $ cap_object page_cap;

    page_vaddr   ← return $ pt_vaddr + (of_nat pt_slot >> small_frame_size);
    page_rights  ← return (cap_rights page_cap);

    map_page spec page pd page_rights page_vaddr
  od
```

```
definition map_page_directory_slot :: cdl_state ⇒ cdl_object_id
                                  ⇒ cdl_cnode_index ⇒ unit u_monad
where
  map_page_directory_slot spec pd pd_slot = do
    page_cap    ← assert_opt $ opt_cap (pd, pd_slot) spec;
    page        ← return $ cap_object page_cap;
    page_slots  ← return $ slots_of_list spec page;

    page_vaddr  ← return $ of_nat pd_slot  >> (pt_size + small_frame_size);
    page_rights ← return (cap_rights page_cap);

    map_page spec page pd page_rights page_vaddr;
```

```
    mapM_x (map_page_table_slot spec pd page page_vaddr) page_slots
  od


definition map_page_directory :: cdl_state ⇒ cdl_object_id ⇒ unit u_monad
where
  map_page_directory spec pd = do
    pd_slots ← return $ slots_of_list spec pd;
    set_asid spec pd;
    mapM_x (map_page_directory_slot spec pd) pd_slots
  od


definition init_vspace :: cdl_state ⇒ unit u_monad
where
  init_vspace spec = do
    pds ← get_pds spec;
    mapM_x (map_page_directory spec) pds
  od

datatype init_cnode_mode = Move | Copy


definition init_cnode_slot :: cdl_state ⇒ init_cnode_mode
     ⇒ cdl_object_id ⇒ cdl_cnode_index ⇒ bool u_monad
where
  init_cnode_slot spec mode dest dest_slot ≡ do
    src_cap    ← assert_opt $ opt_cap (dest, dest_slot) spec;
    assert     (¬ is_untyped_cap src_cap);
    src        ← return $ cap_object src_cap;

    dest_obj   ← get_spec_object spec dest;
    dest_size  ← return $ object_size_bits dest_obj;

    rights     ← return $ cap_rights src_cap;

    orig       ← return $ (is_none $ opt_parent (dest, dest_slot) spec);
    data       ← return $ cap_data src_cap;

    dest_root  ← get_sel4_cap dest DerivedCap;
    dest_index ← return $ of_nat dest_slot;
    dest_depth ← return $ of_nat dest_size;

    src_root   ← return seL4_CapInitThreadCNode;
    src_index  ← get_sel4_cap src OrigCap;
    src_depth  ← return (32::word32);
    if (src_cap = NullCap) then
      return True
    else if (mode = Move ∧ orig) then
      seL4_CNode_Mutate dest_root dest_index dest_depth
                        src_root src_index src_depth data
    else if (mode = Copy ∧ ¬ orig) then
      seL4_CNode_Mint dest_root dest_index dest_depth
```

```
                    src_root src_index src_depth rights data
    else
      return True
  od


definition init_cnode :: cdl_state ⇒ init_cnode_mode ⇒ cdl_object_id
                                                        ⇒ unit u_monad
where
  init_cnode spec mode cnode = do
    cnode_slots ← return $ slots_of_list spec cnode;
    mapM_x (init_cnode_slot spec mode cnode) cnode_slots
  od


definition init_cspace :: cdl_state ⇒ unit u_monad
where
  init_cspace spec = do
    cnodes ← get_cnodes spec;
    mapM_x (init_cnode spec Copy) cnodes;
    mapM_x (init_cnode spec Move) cnodes
  od


definition start_thread :: cdl_state ⇒ cdl_object_id ⇒ unit u_monad
where
  start_thread spec tcb = do
    cdl_tcb  ← assert_opt $ opt_thread tcb spec;
    sel4_tcb ← get_sel4_cap tcb OrigCap;
    ip ← return $ tcb_ip cdl_tcb;
    sp ← return $ tcb_sp cdl_tcb;
    regs ← return [ip, sp];
    success ← seL4_TCB_WriteRegisters sel4_tcb True 0 2 regs;
    assert success
  od

definition start_threads :: cdl_state ⇒ unit u_monad
where
  start_threads spec = do
    tcbs ← get_tcbs spec;
    mapM_x (start_thread spec) tcbs
  od

definition init_system :: cdl_state ⇒ unit u_monad
where
  init_system spec =
    do
      bootinfo ← get_bootinfo;
      parse_bootinfo bootinfo;
      parse_spec spec;
      create_objects spec;
      duplicate_cnode_caps spec;
      init_vspace spec;
```

```
  init_tcbs spec;
  init_cspace spec;
  start_threads spec
od
```

Attachments

- Short paper *Mechanised Separation Algebra*, published at the 3rd International Conference on Interactive Theorem Proving (ITP'12).

Mechanised Separation Algebra

Gerwin Klein, Rafal Kolanski, and Andrew Boyton

[1] NICTA, Sydney, Australia*
[2] School of Computer Science and Engineering, UNSW, Sydney, Australia
{first-name.last-name}@nicta.com.au

Abstract. We present an Isabelle/HOL library with a generic type class implementation of separation algebra, develop basic separation logic concepts on top of it, and implement generic automated tactic support that can be used directly for any instantiation of the library. We show that the library is usable by multiple example instantiations that include structures such as a heap or virtual memory, and report on our experience using it in operating systems verification.

Keywords: Isabelle, Separation Logic

1 Introduction

The aim of this work is to support and significantly reduce the effort for future separation logic developments in Isabelle/HOL by factoring out the part of separation logic that can be treated abstractly once and for all. This includes developing typical default rule sets for reasoning as well as automated tactic support for separation logic. We show that both of these can be developed in the abstract and can be used directly for instantiations.

The library supports users by enforcing a clear axiomatic interface that defines the basic properties a separation algebra provides as the underlying structure for separation logic. While these properties may seem obvious for simple underlying structures like a classical heap, more exotic structures such as virtual memory or permissions are less straight-forward to establish. The library provides an incentive to formalise towards this interface, on the one hand forcing the user to develop an actual separation algebra with actual separation logic behaviour, and on the other hand rewarding the user by supplying a significant amount of free infrastructure and reasoning support.

Neither the idea of separation algebra nor its mechanisation is new. Separation algebra was introduced by Calcagno et al [2] whose development we follow, transforming it only slightly to make it more convenient for mechanised instantiation. Mechanisations of separation logic in various theorem provers are plentiful, we have ourselves developed multiple versions [5, 6] as have many others. Similarly a

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. This work was in part funded by AOARD grant #FA2386-11-1-4070

number of mechanisations of abstract separation algebra exist, e.g. by Tuerk [7] in HOL4, by Bengtson et al [1] in Coq, or by ourselves in Isabelle/HOL [5].

The existence of so many mechanisations of separation logic is the main motivation for this work. Large parts of developing a new separation logic instance consist of boilerplate definitions, deriving standard properties, and often re-invented tactic support. While separation algebra is used to justify the separation logic properties of specific developments [5], or to conduct a part of the development in the abstract before proceeding to the concrete [1, 7], the number of instantiations of these abstract frameworks so far tends to be one. In short, the library potential of separation algebra has not been exploited yet in a practically re-usable way. Such lightweight library support with generic interactive separation logic proof tactics is the contribution of this paper.

A particular feature of the library presented here is that it does not come with a programming language, state space, or a definition of hoare triples. Instead it provides support for instantiating your own language to separation logic. This is important, because fixing the language, even if it is an abstract generic language, destroys most of the genericity that separation algebra can achieve. We have instantiated our framework with multiple different language formalisations, including both deep and shallow embeddings. The library is available for download from the Archive of Formal Proofs [4].

In Sec 2 we show the main interface of the separation algebra class in Isabelle/HOL and describe how it differs from Calcagno et al. Sec 3 describes the generic tactic support, and Sec 4 describes our experience with example instances.

2 Separation Algebra

This section gives a brief overview of our formulation of abstract separation algebra. The basic idea is simple: capture separation algebra as defined by Calcagno et al [2] with Isabelle/HOL type class axioms, develop separation logic concepts in the abstract as far as can be done without defining a programming language, and instantiate simply using Isabelle's type class instantiations. This leads to a lightweight formalisation that carries surprisingly far.

Calcagno et al define separation algebra as *a cancellative, partial commutative monoid* $(\Sigma, \cdot, u)$. *A partial commutative monoid is given by a partial binary operation where the unity, commutativity and associativity laws hold for the equality that means both sides are defined and equal, or both are undefined.* [2]

For a concrete instance, think of the carrier set as a heap and of the binary operation as map addition. The definition induces separateness and substate relations, and is then used to define separating conjunction, implication, etc. Since the cancellative property is needed primarily for completeness and concurrency, we leave it out at the type class level. If necessary, it could be introduced in a second class on top. The definition above translates to the following class axioms.

```
x ⊕ 0 = Some x        x ⊕ y = y ⊕ x        a ++ b ++ c = (a ++ b) ++ c
```

where `op ⊕::'a ⇒ 'a ⇒ 'a option` is the partial binary operator and `op ++::'a option ⇒ 'a option ⇒ 'a option` is the ⊕ operator lifted to strict partiality

with None ++ x = None. From this the usual definitions of separation logic can be developed. However, as to be expected in HOL, partiality makes the $\oplus$ operator cumbersome to instantiate; especially the third axiom often leads to numerous case distinctions. Hence, we make the binary operator total, re-using standard + as syntax. Totality requires us to put explicit side conditions on the laws above and to make disjointness ## a parameter of the type class leading to further axioms. The full definition of separation algebra with a total binary operator is

```
class sep_algebra = zero + plus +
    fixes op ##::'a ⇒ 'a ⇒ bool
    assumes x ## 0  and  x ## y ⟹ y ## x  and  x + 0 = x
    assumes x ## y ⟹ x + y = y + x
    assumes ⟦x ## y; y ## z; x ## z⟧ ⟹ x + y + z = x + (y + z)
    assumes ⟦x ## y + z; y ## z⟧ ⟹ x ## y
    assumes ⟦x ## y + z; y ## z⟧ ⟹ x + y ## z
```

This form is precisely as strong as Calcagno et al's formulation above in the sense that either axiom set can be derived from the other. The last two axioms are encapsulated in the original associativity law. The more intuitive form `x ## y ⟹ x + y ## z = (x ## z ∧ y ## z)` is strictly stronger.

While 7 axioms may seem a higher burden than 3, the absence of lifting and type partiality made them smoother to instantiate in our experience, in essence guiding the structure of the case distinctions needed in the first formulation.

Based on this type class, the definitions of basic separation logic concepts are completely standard, as are the properties we can derive for them. Some definitions are summarised below.

```
P ∧* Q   ≡ λh. ∃x y. x ## y ∧ h = x + y ∧ P x ∧ Q y
P ⟶* Q  ≡ λh. ∀h'. h ## h' ∧ P h' ⟶ Q (h + h')
x ≼ y    ≡ ∃z. x ## z ∧ x + z = y
□        ≡ λh. h = 0
⋀* Ps    ≡ foldl (op ∧*) □ Ps
```

On top of these, we have formalised the standard concepts of pure, intuitionistic, and precise formulas together with their main properties. We note to Isabelle that separating conjunction forms a commutative, additive monoid with the empty heap assertion. This means all library properties proved about this structure become automatically available, including laws about fold over lists of assertions.

From this development, we can set up standard simplification rule sets, such as maximising quantifier scopes (which is the more useful direction in separation logic), that are directly applicable in instances.

The assertions we cannot formalise on this abstract level are maps-to predicates such as the classical $p \mapsto v$. These depend on the underlying structure and can only be done on at least a partial instantiation.

Future work for the abstract development could include a second layer introducing assumptions on the semantics of the programming language instance. It then becomes possible to define locality, the frame rule, and (best) local actions generically for those languages where they make sense, e.g. for deep embeddings.

3 Automation

This section gives a brief overview of the automated tactics we have introduced on top of the abstract separation algebra formalisation.

There are three main situations that make interactive mechanical reasoning about separation logic in HOL frameworks cumbersome. Their root cause is that the built-in mechanism for managing assumption contexts does not work for the substructural separation logic and therefore needs to be done manually.

The first situation is the application of simple implications and the removal of unnecessary context. Consider the goal `(P ∧* p ↦ v ∧* Q) h ⟹ (Q ∧* P ∧* p ↦ -) h`. This should be trivial and automatic, but without further support requires manual rule applications for commutativity and associativity of ∧* before the basic implication between `p ↦ v` and `p ↦ -` can be applied. Rewriting with AC rules alleviates the problem somewhat, but leads to unpleasant side effects when there are uninstantiated schematic variables in the goal. In a normal, boolean setting, we would merely have applied the implication as a forward rule and solved the rest by assumption, having the theorem prover take care of reordering, unification, and assumption matching.

While in a substructural logic, we cannot expect to always be able to remove context, at least the order of conjuncts should be irrelevant. We expect to apply a rule of the form `(P ∧* Q) h ⟹ (P' ∧* Q) h` either as a forward, destruction, or introduction rule where the real implication is between `P` and `P'` and `Q` tells us it can be applied in any context. Our tactics `sep_frule`, `sep_drule`, and `sep_rule` try rotating assumptions and conclusion of the goal respectively until the rule matches. If `P` occurs as a top-level separation conjunct in the assumptions, this will be successful, and the rule is applied. This takes away the tedium of positional adjustments and gives us basic rule application similar to plain HOL. The common case of reasoning about heap updates falls into this category. Heap update can be characterised by rules such as `(p ↦ - ∧* Q) h ⟹ (p ↦ v ∧* Q) (h(p ↦ v))` if `h` is a simple heap map. If we encounter a goal with an updated heap `h(p ↦ v)` over a potentially large separating conjunction that mentions the term `p ↦ v`, we can now make progress with a simple `sep_rule` application.

Note that while the application scenario is instance dependent, the tactic is not. It simply takes a rule as parameter.

The second situation is reasoning about heap values. Again, consider a simple heap instantiation of the framework. The rule to apply would be `(p ↦ v ∧* Q) h ⟹ the (h p) = v`. The idea is similar to above, but this time we extend Isabelle's substitution tactic to automatically solve the side condition of the substitution by rotating conjuncts appropriately after applying the equality. It is important for this to happen atomically to the user, because the equality will instantiate the rule only partially in `h` and `p`, while the side condition determines the value `v`. Again, the tactic is generic, the rule comes from the instantiation.

The third situation is clearing context to focus on the interesting implication parts of a separation goal after heap update and value reasoning are done. The idea is to automatically remove all conjuncts that are equal in assumption and

conclusion as well as solve any trivial implications. The tactic *sep_cancel* that achieves this is higher-level than the tactics above, building on the same principles.

Finally, we supply a low-level tactic *sep_select n* that rotates conjunct *n* to the front while leaving the rest, including schematics, untouched.

With these basic tactics in place, higher-level special-purpose tactics can be developed much more easily in the future. The rule application and substitution tactics fully support backtracking and chaining with other Isabelle tactics.

One concrete area of future work in automation is porting Kolanski's machinery for automatically generating mapsto-arrow variants [5], e.g. automatically lifting an arrow to its weak variant, existential variant, list variant, etc, including generating standard syntax and proof rules between them which could then automatically feed into tools like *sep_cancel*. Again, the setup would be generic, but used to generate rules for instances.

4 Instantiations

We have instantiated the library so far to four different languages and variants of separation logic.

For the first instance, we took an existing example for separation logic that is part of the Isabelle distribution and ported it to sit on top of the library. The effort for this was minimal, less than an afternoon for one person, and unsurprisingly the result was drastically smaller than the original, because all boilerplate separation logic definitions and syntax declarations could be removed. The example itself is the classic separation logic benchmark of list reversal in a simple heap of type *nat* ⇒ *nat option* on a language with a VCG, deeply embedded statements and shallowly embedded expressions.

The original proof went through almost completely unchanged after replacing names of definitions. We then additionally shrunk the original proof from 24 to 14 lines, applying the tactics described above and transforming the script from technical details to reasoning about semantic content.

While a nice first indication, this example was clearly a toy problem. A more serious and complex instance of separation logic is a variant for reasoning about virtual memory by Kolanski [5]. We have instantiated the library to this variant as a test case, and generalised the virtual memory logic in the process.

The final two instantiations are taken from the ongoing verification of user- and kernel-level system initialisation in the seL4 microkernel [3]. Both instantiations are for a shallow embedding using the nondeterministic state monad, but for two different abstraction levels and state spaces. In the more abstract, user-level setting, the overall program state contains a heap of type *obj_id* ⇒ *obj option*, where *obj* is a datatype with objects that may themselves contain a map *slot* ⇒ *cap option* as well as further atomic data fields. In this setting we would like to not just separate parts of the heap, but also separate parts of the maps within objects, and potentially their fields. The theoretical background of this is not new, but the technical implementation in the theorem prover is nontrivial. The clear interface of separation algebra helped one of the authors with no prior

experience in separation logic to instantiate the framework within a week, and helped an undergraduate student to prove separation logic specifications of seL4 kernel functions within the space of two weeks. This is a significant improvement over previous training times in seL4 proofs.

The second monadic instance is similar in idea to the one above, but technically more involved, because it works on a lower abstraction level of the kernel, where in-object maps are not partial, some of the maps are encoded as atomic fields, and the data types are more involved. To use these with separation logic, we had to extend the state space by ghost state that encodes the partiality demanded by the logic. The instantiation to the framework is complete, and we can now proceed with the same level of abstraction in assertions as above.

Although shallow embeddings do not directly support the frame rule, we have found the approach of baking the frame rule into assertions by appending `∧* P` to pre- and post-conditions productive. This decision is independent of the library.

5 Conclusion

We have presented early work on a lightweight Isabelle/HOL library with an abstract type class for separation algebra and generic support for interactive separation logic tactics. While we have further concrete ideas for automation and for more type class layers with deeper support of additional separation logic concepts, the four nontrivial instantiations with productive proofs on top that we could produce in a short amount of time show that the concept is promising.

The idea is to provide the basis for rapid prototyping of new separation logic variants on different languages, be they deep or shallow embeddings, and of new automated interactive tactics that can be used across a number of instantiations.

Acknowledgements We thank Matthias Daum for his comments on this paper.

References

1. J. Bengtson, J. Jensen, F. Sieczkowski, and L. Birkedal. Verifying object-oriented programs with higher-order separation logic in Coq. In *ITP*, volume 6898 of *LNCS*, pages 22–38. Springer, 2011.
2. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proc. 22nd LICS*, pages 366–378. IEEE, 2007.
3. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220. ACM, Oct 2009.
4. G. Klein, R. Kolanski, and A. Boyton. Separation algebra. *Archive of Formal Proofs*, May 2012. http://afp.sf.net/entries/Separation_Algebra.shtml.
5. R. Kolanski. *Verification of Programs in Virtual Memory Using Separation Logic*. PhD thesis, School Comp. Sci. & Engin., Jul 2011.
6. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *34th POPL*, pages 97–108, Nice, France, Jan 2007. ACM.
7. T. Tuerk. A formalisation of smallfoot in HOL. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, LNCS, pages 469–484. Springer, Aug 2009.