Formal Memory Models for Verifying C Systems Code

Doctor of Philosophy
School of Computer Science and Engineering
The University of New South Wales

Harvey Tuch

2008

Abstract

Systems code is almost universally written in the C programming language or a variant. C has a very low level of type and memory abstraction and formal reasoning about C systems code requires a memory model that is able to capture the semantics of C pointers and types. At the same time, proof-based verification demands abstraction, in particular from the aliasing and frame problems.

In this thesis, we study the mechanisation of a series of models, from semantic to separation logic, for achieving this abstraction when performing interactive theorem-prover based verification of C systems code in higher-order logic. We do not commit common oversimplifications, but correctly deal with C's model of programming language values and the heap, while developing the ability to reason abstractly and efficiently. We validate our work by demonstrating that the models are applicable to real, security- and safety-critical code by formally verifying the memory allocator of the L4 microkernel. All formalisations and proofs have been developed and machine-checked in the Isabelle/HOL theorem prover.

Copyright Statement

Signed: _____

Date: _____

Authenticity Statement

I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.

Signed: _____

Date: _____

Originality Statement

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

Signed: _____

Date: _____

Portions of this work were previously published in the following papers:

- Harvey Tuch. *Structured Types and Separation Logic*. In Proceedings of the 3rd International Workshop on Systems Software Verification (SSV), 2008.
- Harvey Tuch, Gerwin Klein and Michael Norrish. *Types, Bytes and Separation Logic*. In Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2007.
- Harvey Tuch and Gerwin Klein. *A Unified Memory Model for Pointers*. In Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-12), 2005.
- Harvey Tuch, Gerwin Klein and Gernot Heiser. *OS Verification — Now!*. In Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS), 2005.

Acknowledgements

I would like to thank my family, in particular my parents Jeff and Sandy, and siblings Laurie and Sharni, for their encouragement, understanding and endless support for me throughout my studies.

The advice and guidance of my thesis supervisors has been invaluable. As a student of Gerwin Klein I have benefited from his insight, direction, expertise and patience. Gernot Heiser's vision, teaching and research instruction sparked my continuing interest in systems.

It has been a privilege to have worked in the Kernel Experimentation Group and L4.verified team at UNSW and NICTA, alongside many excellent researchers, engineers and fellow students, past and present. While there are too many individuals that I have had the pleasure of working with to name here, I would like to acknowledge the influential mentoring of Adam Wiggins.

The feedback received from anonymous paper reviewers, conference attendees, during my internship at Intel and while traveling is greatly appreciated, as is the insightful analysis and suggestions provided by my thesis examiners, Manuel Chakravarty, John Matthews and Tobias Nipkow. Comments on drafts of this thesis from my thesis supervisors, David Cock, Rafal Kolanski, Michael Norrish, Leonid Ryzhyk, Norbert Schirmer, Bastian Schlich, Thomas Sewell and Simon Winwood have been highly useful. Any remaining mistakes are entirely my own.

Finally, I would like to express my profound appreciation for my girlfriend Gabrielle Gareau and her wonderful ways.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Since its inception, computer science has strived as a discipline to develop methods that allow systems to be reasoned about with the same degree of exactness and clarity as a mathematical proof. In favour of this approach is the mechanistic nature of computation, an improving understanding of how to architect for this goal and advances in computer systems themselves yielding improved tools to assist. Working against this enterprise has been the immense and increasing complexity of computer-based systems and a patchy history of success.

At this juncture, there is increasing practical pressure to provide a high degree of assurance of a computer system's security and functionality. This pressure stems from the deployment of computer systems in mission-critical scenarios, the growing economic consequences of failure and the need to protect computing and communication infrastructure against attack. This requires end-to-end guarantees of system functionality, from applications down to hardware.

In this thesis, we address some of the challenges in reasoning about the correctness of the lowest level of the software stack. This poses challenges distinct to other layers as a result of the common choices of implementation languages — typically machine assembly, C or C++ — that have a very low level of type and memory abstraction, and the unshielded interaction with hardware. The lowest level of the software stack may be an operating system, hypervisor, firmware, garbage-collected language run-time, real-time executive, etc. Collectively, we refer to software at this level as *systems code*.

In the rest of this section we further develop the motivation for reasoning about the correctness of systems code, elaborate on the technical problems we intend to address during the course of the thesis, and provide the case for certain formal tools being of use in the pursuit of a solution.

1.1.1 C systems code

The vast majority of systems code today is implemented in the C programming language or some variant such as C++ and Objective-C. To understand why this is the case we consider first the history of C and then its nature as a language.

Along with moon landings, tie-dye shirts and the Beatles, the 1960s and 1970s brought us many significant systems developments, including the C language and Unix operating system. As described in Ritchie's detailed history of the development of the language [82], it followed earlier efforts at implementing Unix on the PDP-7 in assembler and the typeless B language. C was simple and small, suited to the machines of the day and limited in the degree of hardware abstraction of types and language constructs. Nonetheless, C evolved a non-trivial type system and portability while maintaining the correspondence between its primitives and hardware operations, hence providing efficient compiled output that was competitive with handwritten assembly code. For these reasons, the success of Unix and its derivatives, and perhaps also simply as a result of first-mover advantage, C has established itself as the language *de rigueur* of systems.

Today, Unix derivatives continue to enjoy great popularity and almost every significant operating system is implemented at some level in C or a variant. This is true in the commercial world (Windows XP/Vista, Mac OS X, Linux, Free/Open/NetBSD, Solaris, HP-UX, AIX), microkernel research (Mach, L4, Fluke, EROS, Exokernel, MINIX) and embedded systems (VxWorks, QNX, Windows CE). Not just traditional operating systems, but hypervisors (Xen) and even "safe" language runtimes (Sun Hotspot, Mono) follow the rule.

Systems impose on languages many abstraction breaking requirements and are not usually considered amenable to implementation in higher-level languages like Java and ML. For example, zero-copy I/O and address translation are crucial features and programmers demand the freedom to control data structure layout [87], in particular when optimising the cache and TLB footprint that is typically opaque in such languages. Inside the research community there are recent promising efforts at harnessing the gains of the last three decades of programming language research [8, 22, 29, 37, 46, 68, 89], with an emphasis on types and static checking, when implementing systems. However, these advances are yet to be popularised in industry and still face enormous scepticism from systems implementors who are highly obsessed with efficiency, sometimes to the extreme where clock cycles are the metric of choice.

Having established C's rise to dominance, we now consider the language itself. C is a sequential imperative language with (mostly) nested control structures. Side effects occur through expression evaluation and the language provides a number of primitive types and operations corresponding closely

in spirit and often in practice to CPU integer, floating point and address registers and instructions. C also allows programmers to derive further types from the primitive types in a curious mixture of first- and second-class types through its **struct**, **union** and array types.

The C type system evolved from the typeless B language, motivated by convenience, and in earlier versions there were few checks for type safety, for example it was possible to assign pointer values to integers without the need to cast [82]. Even today, it is easy to violate the C type system by its cast mechanism and through address arithmetic. The programmer is given, intentionally, access to low-level bit and byte representations of values in memory. There are no checks on array bounds when indexing — this would violate C's design philosophy. With great power comes great responsibility and it is easy to violate invariants of the system by accessing unallocated memory, exceeding the bounds of arrays, dereferencing a dangling pointer, etc.

A key aspect of systems programming is managing memory. C does not have garbage collection and the programmer is responsible for allocation and deallocation of memory through library calls. A systems implementor may even develop his or her own memory allocator that replaces this already low-level interface, enabling direct management of the physical memory in a system. Memory management in C is one of the sweet spots we target in this thesis.

We consider the above features of C's types and memory model to be one of the fundamental differentiators of formal systems code verification from the general problem of software correctness. We now elaborate on the specific problems that formal reasoning about C's memory interactions brings about.

Pointers, types and safety

Type safety is the guarantee that the evaluation of an expression does not result in the machine entering an erroneous state and that it yields the expected type. This has been shown for C when considering *strictly conforming* programs [1, 4–5] (see §2.1) by Norrish [74]. Unfortunately, systems code is by no means strictly conforming and we could say by definition requires the ability to violate the standard's strict rules on how memory can be accessed. As a result, when describing type safety with respect to a C program in this thesis, we refer to a looser notion, where we may require expressions that designate a memory object to have a type corresponding to the expected value stored in memory. Program fragments can be type-safe if all their expressions have this property and later we formalise what is meant by the expected value's type.

Memory management code tracks the free memory that can be allocated and also sometimes the memory that has been allocated. This is commonly done through pointer-linked data structures, and this use of what are also

called mutable inductively-defined data structures is the cause of a great degree of the difficulty in reasoning about such code formally. This difficulty, a direct consequence of the use of indirection, can be broken down as the *aliasing* [14] and *frame* [61] problems.

For an example of aliasing, consider a program with two pointer variables **int *** p and **int *** q and the following triple:

$$\{\!|\ \mathsf{True}\ |\!\}\ *p = 37; *q = 42; \{\!|\ *p = ?\ |\!\}$$

We are unable to ascertain the value pointed to by p as it may refer to the same location as q. We need to state that $p = q$ or $p \neq q$ in the pre-condition to be able to determine the value of $*p$ in the post-state. We refer to aliasing between pointers of the same type in this thesis as *intra-type aliasing*.

The aliasing problem is much worse for inductively-defined data structures, where it is possible that structural invariants can be violated, and where we need more sophisticated recursive predicates to stipulate aliasing conditions. These predicates appear in specifications, invariants and proofs, and their discovery is often a time consuming trial-and-error process.

The aliasing situation becomes untenable when code is type-unsafe and we are forced to seek improved methods. If instead we had a variable **float *** p:

$$\{\!|\ \mathsf{True}\ |\!\}\ *p = 3.14; *q = 42; \{\!|\ *p = ?\ |\!\}$$

then not only do we have to consider aliasing between pointers of different types, but also the potential for p to be pointing inside the encoding of $*q$ and vice versa. We talk about this phenomenon as *inter-type aliasing*.

The frame problem is apparent in Hoare triples. While specifications may mention some state that is affected by the intended behaviour of a program, it is hard to capture the state that is not changed. In the above example, a client verification that also dereferences a pointer r, not mentioned in the specification, has no information on its value after execution of the code fragment. This limits reusability and hence scalability of verifications.

There have been several solutions proposed in the literature to these problems, which we examine in §1.3. Our intention is not to propose a completely new method, but to study the mechanisation of these solutions, unite them in a common framework and adapt them to type-unsafe C systems code.

1.1.2 Formal verification

Formal reasoning with the usual process of pen-and-paper mathematical proof neither scales as we would like in software verification nor has the expected degree of rigour. For example, formal proofs can unintentionally abstract away critical details and miss boundary conditions, as Bloch observes with the

textbook binary search algorithm [12]. There are two main, complementary, approaches to technology that can assist here — algorithmic verification and theorem proving.

Algorithmic verification techniques typically target a limited subset of the language and a restricted class of properties where there are decidability results or procedures that are efficient in practice. Research in this area has produced impressive results in recent years with improvements in the underlying theory and increased available computing power. They can catch increasingly wider classes of programmer errors and even guarantee the absence of certain types of bugs. We touch on some of the algorithmic techniques that have been aimed at C systems code verification in §1.4. However, it is easy to lose track of the fact that such techniques only currently help with the low hanging fruit and the verification story does not end here. In particular, we wish to show more general properties such as functional correctness, i.e. that a system does what we intend it to do.

To provide for a wider set of properties, we use the theorem proving approach that involves describing the intended properties of the system and a model of its source code in a formal logic, and then deriving a mathematical proof showing that the model satisfies these properties. Only the expressiveness of the logic limits the properties that can be shown, at least in principle. Contrary to algorithmic techniques, theorem proving is usually not an automatic procedure unless we restrict ourselves to decidable logics, e.g. quantifier-free first-order logic. Such restrictions have the same drawback as algorithmic techniques, and so we require human interaction in our proofs. While modern theorem provers remove some of the tedium from the proof process by providing rewriting, decision procedures, automated search tactics, etc., it is ultimately the user who guides the proof, provides the structure, or comes up with suitably strong induction statements. Proofs are developed interactively but can be checked automatically for validity once derived, making the size and complexity of the proof irrelevant to soundness.

Proof creation has a high cost associated with it. To give an idea, it has been the author's experience [93, 94, 96] that, in an interactive theorem prover, verifying the functional correctness of C code can require between one and two orders of magnitude more proof steps than line count, and in a single person year, we may be limited to verifying no more than something in the order of 1000 lines-of-code (LoC). Little data exists for how proof-based projects scale, but it is unlikely to be linear.

The economics of verification have two significant consequences. First, the range of systems we can hope to verify is limited, but is still large enough to be practically interesting. Modern microkernels, with implementations around 10,000 LoC are hopefully within the realm of possibility. Verification of such systems can bring significant improvements to the reliability of the entire software stack as above the microkernel layer hardware protection domains limit the impact any incorrectly behaving software has on the

trusted computing base [83]. L4 [57] is a member of this class of kernels and the work undertaken during this thesis has been a part of the L4.verified project [28] that aims to prove that the behaviours of a C implementation for a next-generation L4 design conform to a high-level abstract API [26]. We use L4 examples in various places in the following[1].

The second consequence of cost is that it is important to have formal models that are as abstract as possible while remaining sound. The construction of these models for reasoning about memory is a recurring theme in this thesis.

1.2 HOL, Isabelle and Hoare logic

We use *higher-order logic* (HOL) as the logical substrate of our specifications and proofs. It goes beyond first-order logic in allowing quantification over functions and is typed. HOL is highly expressive and allows us to model most mathematical concepts in an intuitive manner. In fact, it is quite common to embed others logics and mathematical theories in HOL, and it has been successfully used in a number of hardware and software formalisations and verifications [21, 32, 38, 39, 48, 51, 74, 77].

The generic Isabelle theorem prover [78] has a HOL instantiation, called Isabelle/HOL, and we take this as our technological infrastructure. Isabelle is an interactive theorem proving system with powerful automated rewriting support [72], a high-level proof language [102], rich set of HOL libraries and is built on an LCF-style proof kernel [34, 35]. This last point, where LCF stands for *Logic for Computable Functions*, is of special interest as theorem provers themselves are complex and we trust them to be reliable. LCF-style architectures enhance our confidence in this by reducing the trusted components of the theorem prover to a minimal set of code that is isolated from the rest by the type system of ML, Isabelle's implementation language. It would not be a stretched analogy to liken it to the microkernel approach to system architecture.

Above the HOL layer, we use Schirmer's verification environment [85] that enables us to write HOL pre/post specifications, model our C code, and perform verification using the axiomatic techniques of Hoare [43]. We give more details of this environment in §2.5.1. Hoare triples, where a block of code is preceded by a pre-condition and followed by a post-condition, have already appeared in §1.1.1. Formally, $\{\!|\ P\ |\!\}\ c\ \{\!|\ Q\ |\!\}$ has the meaning that if the block of code c is entered in a state satisfying the pre-condition P and the code terminates, then the system will be in a state satisfying the post-condition Q. This is known as *partial correctness*[2].

[1] For logistical reasons these were based on an earlier implementation of L4, the L4Ka::Pistachio kernel [90].

[2] If we are also guaranteed termination then this becomes *total correctness*.

A Hoare logic is a set of rules that allow the step-by-step syntactic transformation of a triple to a set of logical statements, in our case HOL goals, which we refer to as *proof obligations*. A simple example is the transformation of the triple for **int** n:

$$\{\!|\ n > 1\ |\!\}\ n = n - 1; \{\!|\ n > 0\ |\!\}$$

to the HOL goal $n > 1 \longrightarrow n - 1 > 0$, which is trivially true.

A *verification condition generator* (VCG) can, when supplied suitable invariant annotations, automatically perform the transformation, freeing the user from this initial mechanical aspect of the proof process. Schirmer's entire verification environment, including the verification condition generator, is implemented in Isabelle/HOL, providing the soundness benefits of the LCF-style architecture, and allows the full HOL machinery to be available to the program verifier in specifications and proofs.

Above we dwell on details not just to provide some preliminaries, but because the choice of our underlying logic, tools and verification methodology are not entirely orthogonal to how we chose to represent memory. While we believe the developments in this thesis are more general than this specific point in the design space, there is no doubt that the tools and logics chosen influence the direction taken to a great extent.

1.3 Proving pointer programs

There are three approaches to reasoning about memory in C that we consider in this thesis — semantic, multiple typed heaps and separation logic. While these by no means provide exhaustive coverage of the literature, they are representative of the models that are commonly used today in Hoare logic verification. We mention some models used in other verification techniques and earlier work in §1.4.

Semantic models

When we go beyond a toy language that only allows a set of named discrete variables in a program's state space, and introduce pointers as a language feature, we want to be able to describe the effects of memory accesses and updates through pointer expressions.

A reasonable approach from a descriptive language semantics perspective is to regard memory simply as a function from some type *addr* representing addressable locations to some type *value*, i.e. $addr \rightharpoonup value$. This works fine for typeless languages, while for type-safe languages we can make *value* a disjoint union of languages types, e.g.:

$$\textbf{datatype}\ value \quad = \quad \mathsf{Int}\ int \mid \mathsf{Float}\ float \mid \mathsf{IntPtr}\ addr \mid \ldots$$

The semantics of access and update dereferences are easy to express as they translate to function application and update. Address arithmetic can be modeled by having *addr* be an integer type.

It is straightforward to adapt the Hoare logic assignment rule:

$$\{\!|\ P[x/v]\ |\!\}\ x = v;\ \{\!|\ P\ |\!\}$$

where $P[x/v]$ indicates that all occurrences of program variable x in assertion P are replaced with v. To do so, we treat memory as a variable with a function type. If this variable was called h then the rule would be:

$$\{\!|\ h\ p \neq \bot \wedge P[h/h(p \mapsto v)]\ |\!\}\ *p = v;\ \{\!|\ P\ |\!\}$$

We are not overwhelmed in the resultant proof obligations by function updates as the scope of a function update is limited to the enclosing function, loop or specification block.

With type-unsafe languages like C, we run into trouble with this typed model as it becomes quickly apparent in the programmer's model that an **int** representation is not contained in a single location, and we need to replace *value* with *byte*. We then require functions from language values to sequences of *byte*s and their inverses that map the other way. Heap function update is replaced by a series of function updates for the *byte* encoding. This is the model used to describe C and C++ semantics by Norrish [74] and Hohmuth et al [44] respectively. We adopt a similar model in Chapter 2 as the basis for our semantics.

While it is possible to formally reason in this way about the *effects* of pointer operations, we are still at a loss as to how to avoid the aliasing and frame problems. One can derive point-wise rewrites [44] that simplify proof obligations when we know that an update can be ignored, but there is no abstraction from the aliasing or frame problems. The next two models describe approaches where first inter-type aliasing and then intra-type aliasing and the frame problem are addressed.

Multiple typed heaps

With type-safe languages, we can rule out the adverse effects of inter-type aliasing in our memory model by having a separate heap variable for each language type in the program's state space, e.g. *float-heap* :: *float ptr* $\rightharpoonup$ *float*, *int-heap* :: *int ptr* $\rightharpoonup$ *int*, *int-ptr-heap* :: *int ptr ptr* $\rightharpoonup$ *int ptr*, etc. Updates to one heap do not affect others, and hence we get that any assertion that is only a function of an **int** heap is preserved across a **float *** update without any additional work needing to be done.

Bornat [14] describes how we can further rule out potential aliasing with structure or record types in the situation where there is no pointer arithmetic and these types are second-class, that is their values cannot be dereferenced

or assigned directly. With these restrictions, each field in each structured type can be given its own heap, as it is impossible for an update to one field to ever affect an access of another field in the same or different object.

Unfortunately, C does not guarantee type-safety and hence multiple typed heaps are unsound as a fundamental memory model for the language. They also do not support language features we require such as casts and pointer arithmetic. In addition, Bornat's restrictions do not apply to the language. We see later in this thesis how multiple typed heaps can be used as a proof technique without compromising soundness, and how abstract reasoning about first-class structured types can occur.

Separation logic

Multiple typed heaps only help with inter-type aliasing. A popular approach to managing the aliasing and frame problems currently is the *separation logic* of Reynolds, O'Hearn and others [45, 81]. Separation logic is an extension of Hoare logic that provides a language and inference rules for specifications and programs that both concisely allows for the expression of aliasing conditions in assertions and ensures modularity of specifications.

Separation logic introduces new logical connectives, separation conjunction $\wedge^*$ and implication $\longrightarrow^*$. We can now write $(p \mapsto 37 \wedge^* q \mapsto 42)$ to mean that in the heap, the dereferenced p and q map to their respective values and do so in disjoint regions of the heap. Separation conjunction implicitly includes anti-aliasing information, making specifications clearer and providing an intuitive way to write inductive definitions for data structures on the heap. Our earlier example becomes:

$$\{\!|\ (p \mapsto -) \wedge^* (q \mapsto -)\ |\!\}\ *p = 37; *q = 42; \{\!|\ (p \mapsto 37) \wedge^* (q \mapsto 42)\ |\!\}$$

Separation logic extends the usual Hoare logic rules with additional rules to manage heap assignments and dereferences, and with the *frame* rule:

$$\frac{\{\!|\ P\ |\!\}\ c\ \{\!|\ Q\ |\!\}}{\{\!|\ P \wedge^* R\ |\!\}\ c\ \{\!|\ Q \wedge^* R\ |\!\}}$$

The frame rule allow us to take an arbitrary triple for a pointer program and globalise it to be used in the proof of a calling procedure. This works because separation logic forces any heap state that might be shared with the caller to appear inside P or Q. A separation logic specification then tells the reader what the program does not do, as well as what it does.

Even though separation logic has been developed with low-level languages in mind, there are complications with type-unsafe languages. In particular, there is the problem of *skewed sharing* [81] related to inter-type aliasing, which we discuss at the end of §4.2.2. The frame rule as well requires special treatment to be applicable to C programs that use a more relaxed notion of memory safety — we consider this in §4.4.

1.4 Related work

The earliest work in program verification focused on establishing axioms to enable mathematical reasoning about programs in imperative languages. McCarthy [60] and Hoare [43] were influential in this endeavour. Later papers tackled rules for assignment, procedures, recursion and heap allocation primitives in the presence of pointers, reference variables and arrays. Cartwright and Oppen [19], Morris [64], Bijlsma [10] and Burstall [16] are examples of this. The result of applying these rules was a sequence of updates that could then be further reasoned about in proofs, as in semantic models, or in some cases the multiple typed heaps model, where distinct fields, arrays and language types could be used to reduce the effects of aliasing. The languages targeted were type-safe or typeless. Most work at this time was through pen-and-paper formalisation.

More recently, Bornat [14] revisited the work of Morris and Burstall, and produced a mechanised proof in the Jape editor of a number of examples including the Schorr-Waite graph marking algorithm. A similar mechanisation and study has been performed by Mehta and Nipkow [62] in Isabelle/HOL. The Caduceus tool [31] also uses what now seems to be commonly called the Burstall-Bornat model and Moy [65] extends this to cope with some well-behaved cases of unions and type casts.

Leroy and Blazy [56] have a memory model in the Coq theorem prover for C that is aimed at compiler verification. It contains a far more thorough approach to C's memory than we consider in this thesis, including the modeling of stack variables, but has in-built allocation primitives and is faithful to the C standard, making it not as suitable for offending systems code. In addition, verifying functional properties of C pointer programs requires higher-level models than those needed for reasoning about semantics and compiler transformations, e.g. Burstall-Bornat or separation logic, which are the focus of our work.

Norrish [74] and Hohmuth et al [44] provide mechanised C/C++ semantics in HOL and PVS respectively, which include low-level memory models, and provide the basis for our own semantic model in Chapter 2. In our HOL type encoding in §2.4, the approach is similar to that of Blume's [13] encoding of the C type system in ML that utilises phantom typing to express pointer types and operators for the purpose of a foreign-function interface.

Separation logic was also inspired by Burstall's work, and has been developed in the papers of Reynolds [80, 81], O'Hearn [75], Yang [105], Ishtiaq [45] and Calcagno [18]. This has since been mechanised for simple languages in Isabelle/HOL by Weber [100], Preoteasa [79] in PVS based on a predicate transformer semantics and Marti et al [59] in Coq for a version of C without dealing with its types. Tuch et al [96] gave the first treatment of separation logic that unified the byte-level and logical views of memory in Isabelle/HOL. Appel and Blazy [3] later gave a mechanised separation logic

for a C intermediate language in Coq with the strict standard's memory view.

We work with a subset of C, and there have been numerous other definitions of C subsets and variants aimed at taming the language for formalisation or as an intermediate language for transformations. These include C0 [53], Cminor [55], Clight [11], BitC [89], CCured [68] and CIL [69]. We make no special claim about our subset in relation to this body of research other than that it has a lower-level memory model than most and is well suited to the target verification environment.

Algorithmic techniques attract a lot of attention today. The main relevant approaches are software model checking, static analysis and separation logic decision procedures. C language software model checkers [86] include SLAM [4] and BLAST [42], which have had success in checking safety properties such as correct API use in device drivers. Similarly, Hallem et al [36] use static analyses to find bugs in system code. More sophisticated abstract domains are used in shape analyses [63, 84], which can show some structural invariants, such as the absence of loops in linked lists. Separation logic decision procedures [6] can also show similar properties. At this point in time, these techniques tend to be specialised for limited language fragments or data structures, but there are promising developments that may improve this situation [17].

We complete the related work with a brief look at what has been achieved to date in verifying functional properties of operating systems. Some of the earliest work on OS verification was in the PSOS [70] and UCLA Secure Unix [99] systems. The rudimentary tools available at the time meant that the proofs had to end at the design level; full implementation verification was not feasible. Later, Bevier [9] describes verification of process isolation properties down to object code level for the simplified KIT kernel in the Boyer-Moore theorem prover. A number of case studies [20, 27, 97] have modeled the IPC and scheduling subsystems of microkernels in PROMELA and analysed them with the SPIN model checker. Manually constructed, these abstractions were not necessarily sound, and so while useful for discovering concurrency bugs, they could not provide a guarantee of correctness. The VeriSoft project [33] is attempting to verify a whole system stack, including hardware, compiler, applications, and a simplified microkernel called VAMOS that is inspired by, but not very close to, L4. Most closely related to our case study in Chapter 6 is the successful verification of the kernel memory allocator from the teaching-oriented Topsy operating system by Marti et al [59] in Coq. The major difference is the heavy use of pointer arithmetic and casting in L4's memory allocator that we are able to handle confidently and conveniently due to our more detailed semantic model and type encoding.

1.5 Contributions

This thesis makes a number of contributions, primarily in the areas of interactive theorem proving, formal memory models and language semantics:

- A rigorous treatment of the multiple typed heaps and separation logic proof abstractions is provided, in a unified framework that demonstrates the soundness of these techniques and their relationship to the underlying byte-level view of system memory and each other. We mechanise this treatment in higher-order logic in the Isabelle theorem prover.
- A type encoding and semantics for C types and objects is developed in Isabelle/HOL. This makes elegant use of the HOL type system, reducing specification and proof type annotation overhead, and integrates with an existing Hoare logic environment and verification condition generator. General properties of C types as given by the standard are formalised and a mechanism to supply implementation-defined behaviour for specific compilers and architectures is established. We cope fully with many language features that are often ignored in language semantics — size, alignment, padding, type-unsafe casts and pointer address arithmetic, to name a few.
- Limitations in the multiple typed heaps and separation logic proof abstractions when structured types appear are exposed, and the models are extended to accommodate these types. We show that the earlier models are special cases of the generalised development and present new features that are available to proofs about pointer programs with structured types.
- A study in the application of the framework to the verification of a kernel memory allocator for the L4Ka::Pistachio microkernel implementation is given. This contains unsafe code that exercises our framework and provides an opportunity to compare the two proof abstractions studied in the same setting.

To the best of our knowledge, while some aspects of each contribution above are mentioned in §1.4, each point constitutes a novel contribution and no such study as this has been attempted before either in a theorem proving system or without.

1.6 Notation

This thesis makes heavy use of formal notation from Isabelle/HOL. Mostly this can be thought of as conforming to standard mathematical and functional

programming notation. Here we describe some of the non-standard aspects of the language and some key types. For a more thorough introduction, the Isabelle/HOL tutorial [73] is an excellent resource.

All definition, theorems and formulas are built up from *terms*. Each term t in HOL has a type τ. Most of the time this is implicit, with Isabelle performing type checking and inference, however we can make typing explicit through a type annotation $t{::}\tau$. Type variables are written α, β, γ, etc. Any two types α and β can be paired to give a new type $\alpha \times \beta$. Functions in HOL are total, and we write function types with $\alpha \Rightarrow \beta$. Type synonyms are introduced with the **types** keyword and algebraic datatypes (disjoint unions) with **datatype**, for example:

$$\textbf{datatype}\ \alpha\ \mathit{option} \quad = \quad \mathsf{None} \mid \mathsf{Some}\ \alpha$$

Compound types can be formed from pair types, e.g. tuples $\alpha \times \beta \times \gamma$, and Isabelle/HOL provides a means to introduce types with named fields using the **record** keyword, e.g.:

$$\begin{array}{lcl} \textbf{record}\ \mathit{point} & = & \mathsf{x} :: \mathit{nat} \\ & & \mathsf{y} :: \mathit{nat} \end{array}$$

Each field in a **record** has an access and update function supplied by Isabelle. For the field x above these are x::*point* $\Rightarrow$ *nat* and x-update::(*nat* $\Rightarrow$ *nat*) $\Rightarrow$ *point* $\Rightarrow$ *point* respectively. We use the syntax $v(\!|\mathsf{x} := k|\!)$ for x-update $(\lambda\text{-}.\ k)\ v$.

While HOL is a logic of total functions, we can model partial functions with the α *option* type, i.e. $\alpha \Rightarrow \beta$ *option* where None represents no mapping and Some a the existence of a mapping with value a. Since this is a frequent occurrence, we have special syntax, and write $\alpha \rightharpoonup \beta$, $\bot$ and $\lfloor a \rfloor$ respectively. There are also some additional concepts related to partial functions. The Some constructor has an underspecified inverse called the, satisfying the $\lfloor x \rfloor = x$. Function update is written $f(x := y)$ where $f{::}\alpha \Rightarrow \beta$, $x{::}\alpha$, $y{::}\beta$ and $f(x \mapsto y)$ stands for $f(x := \mathsf{Some}\ y)$. Domain restriction is written $f\restriction_A$ where $f{::}\alpha \rightharpoonup \beta$ and $(f\restriction_A)\ x = (\mathsf{if}\ x \in A\ \mathsf{then}\ f\ x\ \mathsf{else}\ \bot)$.

Finite integers are represented by the type α *word* where α determines the word length. This is backed by a bit-vector library with support for the usual arithmetic and bitwise operators. For succinctness, we use abbreviations like *word8* and *word32*. The functions $\mathbb{N}^{\Leftarrow}$ and $\mathbb{N}^{\Rightarrow}$ convert to and from natural numbers. Arithmetic operations on bit-vector values are modulo 2^n, where n is the word length.

Isabelle supports type classes [101] similar to, but more restrictive than Haskell's. Isabelle's type classes are axiomatic in the sense that a set of properties, or "axioms", can be associated with a class, forcing all types in that class to have the specified properties. The notation $\alpha{::}\mathit{ring}$ restricts

the type variable α to those types that support the axioms of class *ring*. So, restricting a polymorphic term t to a type in class *ring* appears as $t{::}\alpha{::}\mathit{ring}$. Type classes can be reasoned about abstractly, with recourse just to the defining axioms. Furthermore, a type τ can be shown to belong to a type class given a proof that the class's axioms hold for τ, a process referred to as *instantiation*. All abstract consequences of the class's axioms then follow for τ.

For every Isabelle/HOL type α we can derive a type α *itself*, consisting of a single element denoted by $\mathsf{TYPE}(\alpha)$. This reflects types at the term level and provides a convenient way to restrict the type of a term when working with polymorphic definitions. We exploit axiomatic type classes and α *itself* to a great extent in this thesis, as they allow Isabelle's type system to manage many of the type correctness and other typing issues we encounter.

Two final non-type related aspects of our notation deserve some attention. First, we represent addresses with bit-vectors, and write address intervals as $\{p..{+}n\}$, where p is the base address and n is the size of the interval. Intervals wrap around the end of the address space. Hoare triples are written $\{\!|P|\!\}\ c\ \{\!|Q|\!\}$ where P and Q are assertions and c is a program. In assertions, we use the syntax $'x$ to refer to the program variable x in the current state, while ${}^{\sigma}x$ means x in state σ. Program states can be bound across triples by $\{\!|\sigma.\ P|\!\}\ c\ \{\!|Q|\!\}$.

1.7 Outline

This thesis is structured as a series of successive models for reasoning about memory and pointer programs, each building upon the previous, followed by a case study.

In Chapter 2 we introduce a subset of C designed to remove many of the unnecessarily troublesome features of the language yet still be of relevance to systems verification. We present a model for C types and memory and give details of how C expressions and statements are translated to HOL.

Chapter 3 and Chapter 4 take the semantic model of Chapter 2 and develop proof techniques above it that allow for some abstraction in the treatment of memory updates and aliasing. Chapter 3 is concerned with the multiple typed heaps abstraction where we unify the low-level semantic byte granularity model with typed heaps. Chapter 4 explores a shallow embedding of separation logic in HOL with a focus on the validity of the frame rule in our setting.

Dealing with C's structured types, e.g. **struct**s and arrays, brings new challenges which we examine in Chapter 5. We generalise the developments of Chapter 3 and Chapter 4 and give some new rules for reasoning about C's non-primitive types.

Chapter 6 is a real-world case study, on the memory allocator of an implementation of the L4 microkernel, that both demonstrates the utility of the models in previous chapters when verifying C systems code and allows for a comparison of their effectiveness.

We conclude in Chapter 7 by summarising the results and experiences of this thesis and consider future directions that might be fruitful in further research.

Chapter 2

Semantic model

In this chapter we give a description of the semantics for the effects of C expressions and statements, with an emphasis on the aspects related to types and the memory model. This provides the formal grounding for C that we build on in later chapters where we develop proof techniques that allow us to effectively reason about C pointer programs.

Following some introductory definitions, we give a description of the assumptions that underlie our model and the strict subset of C utilised, termed C_{sys}. We present the state space and a simple model for C's store and heap, with semantics for accesses and updates affecting these components, in the next section. As this is an Isabelle/HOL mechanisation, detail is provided on how C types are encoded in HOL, taking advantage of theorem prover features such as type inference and polymorphism to provide convenient and compact descriptions of operations such as pointer arithmetic and reducing the burden of type annotations in specifications and proofs.

The focus in this chapter is on memory access and update, other semantic concerns with C have been treated elsewhere in the literature [74]. Of particular interest in the present thesis is how the usually safe and standard conforming semantics for C can be extended to cope with the system programmer's view of C as essentially a portable assembler layer, where the memory model is at odds with that of the strict standard. While it may be argued that code written with such a model in mind is by definition incorrect, the fact remains that the vast majority of operating system kernel implementations are written with C/C++ code in this way, and formal modeling and verification can still help increase the confidence in the correctness of the implementation. We formally state our semantic model here, achieving a separation of concerns between the verification enterprise and checking the validity of the model.

2.1 Execution model

There are several key concepts from the ISO/IEC draft standard [1] that we refer to in this chapter, introduced below. References to the relevant paragraphs are provided with [1, Clause–Paragraph] annotations. The C programming language has a long history and has been defined, standardised and then revised several times since 1978 [82]. This standard for C99 comes later than the ANSI C standard described by Kernighan and Ritchie [47] and specifies a technically different language, with additional features. However the spirit of the language is retained and the pertinent sections in this thesis are common to both, so we use the ISO/IEC standard here for the clarity its definitions bring.

The C standard does not provide a formal semantics for the language. It does however provide a natural language definition of an *abstract machine* [1, 5.1.2.3], supplying the *abstract semantics*. This machine has a notion of state, called the *execution environment* [1, 5.1.2], containing *objects* [1, 3.14][1]. As a C program is effectively evaluated by the abstract machine, at the granularity of (sub-)expressions and statement evaluation, the state transitions through *side effects* [1, 5.1.2.3–2]. In general there is non-determinism in the order of expression evaluation, however the standard defines a notion of *sequence points* that restricts this. One way of thinking about this is to view a program as having a trace semantics given by an execution DAG rooted at the initial state, branching during the evaluation of expressions, depicted in Fig. 2.1. We include sequence points as special nodes in the DAG. The branching of the abstract semantics is not merely on the choice of which sub-expression to evaluate when there is a choice, but can involve partial evaluation of sub-expressions, and can be captured as a partial order [30]. The edge labels in this DAG are *actions* — object accesses and updates. Externally visible actions are those related to **volatile** objects and are denoted α_i. Termination of the program may also be observed and is labeled δ. Other actions are not further relevant and are labeled τ.

The *actual semantics* are provided by a set of hardware and software that constitute the *implementation*. The standard places restrictions on the behaviour of an implementation. A *behaviour* in the standard is considered an "external appearance or action" [1, 3.4]. Ignoring files and I/O, we take the trace set derived from the execution DAG to constitute these behaviours. The standard mandates certain deterministic steps in the execution DAG, and leaves others as:

- Implementation-defined [1, 3.4.1] — here the implementation may make a choice, but it must be consistent with documentation.

[1] The standard also mentions files [1, 5.1.2.3–2], but this is only relevant in the presence of library functions that interact with these components of state, which are outside the scope of this thesis.

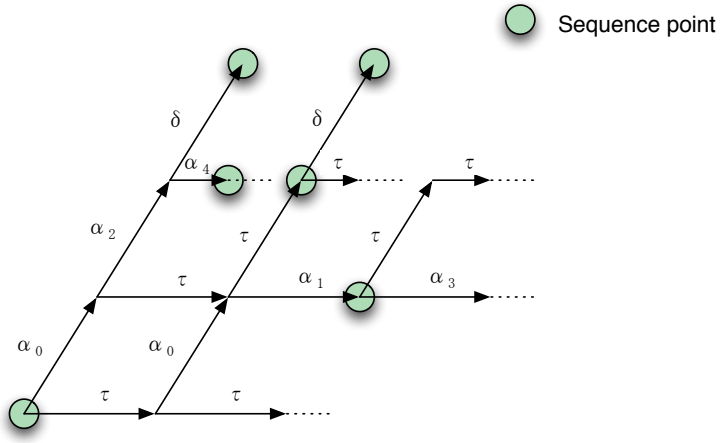

Figure 2.1: Execution DAG for abstract C machine.

- Undefined [1, 3.4.3] — the standard imposes no requirements. This would appear in the execution DAG as a node branching with every possible action.
- Unspecified [1, 3.4.4] — the abstract semantics give more than one possibility, i.e. branches the execution DAG in a stated manner.

Conforming programs [1, 4–7] are those programs that a *conforming implementation* [1, 4–6] considers valid and implements the intended semantics for. Conforming implementations should provide a subset of the behaviours of the abstract semantics. The standard provides the notion of a *strictly conforming* program [1, 4–5], imposing a well-formedness requirement on the DAG in its abstract semantics. Any branching as a result of implementation behaviour, i.e. not an input action, should lead to states that produce identical trace sets.

2.2 C_{sys} assumptions

In a treatment of language semantics suitable for use as the basis of a program verification tool, i.e. not purely descriptive, it is desirable to gain abstraction where possible for the class of programs to be verified to simplify the resulting proof obligations. This necessitates making certain assumptions and restrictions. As well, since target systems code tends not to be strictly conforming, and may rely on implementation-defined, unspecified or even undefined behaviour as a result of knowledge about a particular implementation, we require the ability to incorporate known features of implementation behaviour in the semantics, for example the size of a pointer representation.

The implementation behaviours may be derived from documentation, which is rarely formal, from header files, architecture reference manuals and ABIs, compiler internals, or even, quite commonly, the behaviours observed. There is clearly a possibility of error in this process, however without a fully formalised and verified C implementation we are left with the task of managing this error. We state our assumed behaviours clearly in the following, noting where they deviate from the standard. Should an assumption be mistaken, it is possible to update the model and recheck proofs, giving a more rigorous approach to unifying the programmer's model and the implementation. Outside of the scope of this thesis, there is the orthogonal task of checking correspondence between the model presented below and implementation behaviour — there exists an active body of literature on compiler and hardware verification [7, 11, 15, 53–55, 98].

Here we give some of the significant assumptions underpinning the semantics used in this thesis and their justification:

- Sequential execution is assumed. It is of course desirable to provide a concurrent semantics, as even non-preemptable systems code requires this formal support on SMP systems. However, this goes beyond the C standard and we consider this outside of the scope of the thesis, requiring consideration of rather involved issues such as memory ordering on contemporary processors [58]. The later case studies do not feature concurrency, demonstrating the utility of this work with this restriction.

- Only the standard C control structures are supported. Code may not modify either itself or its execution stack other than through standard language features. We also disallow function pointers. In this thesis we are primarily interested in the verification of systems code responsible for memory management. As a result, we provide a simple and sound model of C that does not feature any extensions such as notifications, continuations or context switches. For example, while initialisation code for an operating system kernel may require explicitly manipulating stack frames and the kernel's own virtual memory mappings, the vast majority of kernel code, including the case studies we present later, do not do this and hence it is possible to use a model that elides this.

- The heap is assumed to be a subset of physical memory that functions, with alignment restrictions, as a map between addresses and bytes. No aliasing via address translation or cache incoherence is expected. Work in this area by Tews [92] has started to examine how we may establish this for those parts of a systems program that do not violate the abstraction and Kolanski [50] intends to extend the later separation logic embedding in this thesis to overcome the limitation.

- C_{sys} eschews some of the more troublesome features of C, such as non-deterministic ordering in expression evaluation. This is not a case of expressing the ordering explicitly, as this is unspecified and not even necessarily consistent in an implementation. Instead, expressions are restricted such that they remain within a syntactic subset of standard C yet have deterministic side effects. These restrictions are described in §2.5. While they aid in making the semantics simpler and proofs more tractable, the actual restrictions are somewhat orthogonal to the models presented in this thesis, and so we do not consider this to be a significant limitation.
- The C standard library is not included. We consider this a feature of our development, as the systems we target are freestanding implementations that do not include the library, and we may wish to verify the library implementation itself where this does exist.
- Low-level details of data structure layout and a direct correspondence between pointer values and addresses is required in §2.3.1. Since the motivation in developing the models and abstractions in this thesis is reasoning about code that relies on these details for its correctness, it is reasonable to assume the implementation makes these consistent and knowable.
- The pre-processor is assumed to have been run prior to verification, so we do not include any discussion of C's macro language and other pre-processing directives. **typedef** synonyms are also assumed to be handled similarly.

Other technical C_{sys} limitations are introduced later in this chapter, in particular we forbid taking the address of automatic variables and restrict some aggregate types.

2.3 State space

The execution environment or state for C programs is modelled with a **record** in Schirmer's verification environment [85]. Program variables are treated as fields of the **record** and the heap is also a member field with a function type.

Example 2.3.1. A program whose entire state is a single variable $n::int$ could be modelled with state:

$$\textbf{record}\ \mathit{simple\text{-}state} \quad = \quad n\text{-}' \ ::\ \mathit{int}$$

where the $\text{-}'$ suffix allows syntactic identification of variable field names. A shallow embedding of an increment operation on this state space would be $\lambda s.\ s(\!|n\text{-}' := n\text{-}'\ s\ +\ 1|\!)$.

In this section we define our object model and detail how the verification environment's state representation is used to model C's heap and variable store.

2.3.1 Relaxed Object Lifetime Model (ROLM)

According to the standard, an *object* [1, 3.14–1] is a

> region of data storage in the execution environment, the contents of which can represent values

An *lvalue* is an expression that designates some object [1, 6.3.2.1–1], e.g. $*(p + 1)$.

Objects have a lifetime [1, 6.2.4–2], described by a *storage duration* [1, 6.2.4–1], during which the object is guaranteed to exist and may be accessed and updated through lvalues. *Static* storage duration is the same as that of the program's lifetime [1, 6.2.4–3], *automatic* is restricted to the scope of a block [1, 6.2.4–5] and *allocated* storage duration is determined by explicit invocations of **malloc** and **free** [1, 7.20.3–1].

Objects may overlap hierarchically, for example a **struct** object contains member objects in its storage region. An object that has been initialised with a value of a specific type may be accessed through an expression expecting a value of that type, but the region itself does not necessarily have a type associated with it, i.e. the type of an object depends on the value stored in it, or more precisely the expression used to designate it [1, 6.3.2.1–1]. This is a result of C not having a typed memory allocator, and hence calls to **malloc** return a **void ***, requiring appropriate casting and initialisation prior to use. It is a necessary condition that the base of the object be appropriately *aligned* prior to use. Alignment is determined by type [1, 3.2–1], and if a type α has alignment n then n divides the base address of any object of type α. In practice, $\exists\, k.\ n = 2^k$, since the alignment restriction is motivated by architectural restrictions on the lower-order bits of addresses, and we make this restriction in the rest of the thesis.

In this work we adopt less stringent requirements on the lifetime of objects than the ISO C standard and impose some restrictions on the use of pointer-related operators. The intent is that this view corresponds to one acceptable to system implementers and is shared by the compiler and other aspects of the load- and run-time utilised.

malloc and **free** are in a sense both language constructs and library functions. In systems running outside of a hosted environment, e.g. a microkernel or standalone JVM, one often requires explicit control over allocated storage, and the ability to implement one or more allocators to manage regions of memory. We also wish to have the ability to verify these allocator implementations, which are themselves typically written in C.

Example 2.3.2. In the L4Ka::Pistachio kernel there are two memory allocators arranged in hierarchy:

- **alloc/free** — manages regions of memory of size and alignment a multiple of 1K for the purpose of page tables and thread control blocks.
- **mdb_alloc_buffer/mdb_free_buffer** — based on storage acquired through **alloc** allows smaller allocations for the purpose of nodes in a tree data structure tracking the relationship of pages and frames between address spaces.

Implementing these memory allocators requires that the allocated storage duration be sidestepped in favour of a view of memory in which we are able to manage the allocation of objects, their values and types in C code.

A further motivation for venturing into the realm of the implementation-defined behaviour of the object model is the necessity that interactions with memory-mapped devices take place at specific physical addresses. Implementing critical functionality in C, such as timer and interrupt control, requires that one is able to take integer values representing these addresses, cast them to a corresponding pointer and be able to consider the target object valid. The C standard in fact allows for this technique in *address constants* [1, 6.6–9]. This is required elsewhere too, as pointer representations are often directly visible in systems code, e.g. in alignment tests or explicit pointer construction from a base and offset derived from different calculations.

Definition 2.3.1. The *relaxed object lifetime model* (ROLM) assumption is a notion of object lifetime that dispenses with the allocated storage duration and is detailed below. ROLM is motivated by the above examples requiring implementation-defined (and what the standard considers unspecified and undefined) behaviour to be taken into account.

First we draw a distinction between *automatic* program variables and other objects. Automatic variables are stored in the execution stack. All other objects, i.e. those with *static* storage duration and *allocated* storage, exist in the heap component of state, modelled as a single field described in §2.3.2. We focus the discussion on heap objects here and deal with automatic variables in §2.3.3.

We consider an α object in the heap to be an interval of length size-of TYPE(α) based at a particular address. Objects exist in a potentially non-contiguous subset of the domain of the heap, leaving some non-accessible memory for the implementation to place the code segment, automatic variables and other runtime machinery.

Within the heap, we do not impose any restrictions on which addresses may be used for objects beyond the alignment requirements — e.g. a block of C code is free to regard $\{x..+\text{size-of TYPE}(\textit{float})\}$ as containing a **float** and then later have $\{x + 2..+\text{size-of TYPE}(\textit{short})\}$ contain a **short** value.

Objects in the heap no longer have an explicit lifetime, and storage duration is determined by program semantics. Pointers are considered to have a direct correspondence with addresses and to be convertible through casts between different pointer types and integer representations. Programs may directly manipulate pointer contents as a result. It is clear that this is a very liberal view of heap memory and objects, but also this is a view that is adopted commonly in systems programming idioms and practice.

The C standard introduces constraints on initialisation and use of objects. For automatic variables, an *indeterminate value* [1, 3.17.2] is placed in the object each time the scoping block is entered [1, 6.2.4–5]. Any attempt to use this value prior to initialisation results in undefined behaviour [1, J.2–1]. **malloc**ed objects and padding components of structured types [1, 7.21.4.1–2] also receive indeterminate values [1, 7.20.3.3–2], but objects with static storage duration are automatically initialised to 0 or NULL values [1, 6.7.8–10].

In different verification scenarios and implementations, one may wish to adopt a more or less strict notion of initialisation. In ROLM, we leave this as a parameter of the framework we are developing. In the translation provided later in this chapter, we make the choice of having values of automatic variables when entering a block underspecified and static and heap objects deriving values from previous contents. To avoid the incorrect dependence on these values, one can use the guard mechanism of §2.5.5 to protect expressions that have such references. This may require some additional book-keeping state as in Norrish's [74] "initialised addresses" state component.

2.3.2 Heap

It is common in language semantics to treat the heap or memory as a partial function *int* $\rightharpoonup$ *lang-val*, where *int* is the type of addresses and *lang-val* the type of all language values. While greatly simplifying the formalisation, this makes several assumptions that are not valid in our setting:

- Addresses range over an infinite integer type. In C, addresses are constrained by a finite addressable memory, which affects the semantics of pointer arithmetic and memory allocation, e.g. $*(x+1) = y$ may in fact be a NULL pointer dereference.

- Value representations are atomic. C language types have representations spanning multiple locations, and it is possible to have value updates at one location affect values in other cells. This calls for a semantic model that both captures values' storage sizes and reflects these update semantics accurately. E.g. $*x = 0xdeadbeef$ affects not only the byte at location x, but also the bytes at locations $x + 1$, $x + 2$, and $x + 3$. An additional complication is alignment, a per-type restriction

on address validity. For example, 16-bit **short** values may be forced to be stored at even addresses. Expressing alignment conditions in dereferencing and update semantics requires a constant byte granularity for addressing.

- Heap partiality. Heap partiality is often used in the heap dereferencing semantics in memory or type-safety checks. Much weaker variants of these properties hold for C programs and it is not always necessary to introduce them in the dereference semantics. This is particularly important in making it possible to verify low-level code that manages details such as the layout of its own address space or implements the functionality of **malloc**.

To overcome these limitations, we adopt a view of memory close to that of hardware. In our model, heap memory state is a total function from addresses, represented by a bit-vector type corresponding to machine addresses, to bytes, also a bit-vector type. This function is a field in the state record, treated by the verification environment as a variable. On a machine with 32-bit addresses and 8-bit bytes the heap memory state will be:

$$\begin{array}{lllll} \textbf{types} & \mathit{addr} & = & \mathit{word32} \\ & \mathit{byte} & = & \mathit{word8} \\ & \mathit{heap\text{-}mem} & = & \mathit{addr} \Rightarrow \mathit{byte} \end{array}$$

The connection between this low-level view of state and typed language values in the theorem prover is made in §2.4.

2.3.3 Store

There are two kinds of variables according to the ISO C standard, those with scope and lifetime tied to some function or block, known as *automatic* variables, and those lacking this dependency, called *static* variables. In our setting, we map automatic variables to distinct fields of the state record, and variables with static storage duration are considered to be part of the heap, with all accesses translated as indirection through the heap function.

Automatic variables

Each automatic variable in the program appears as a field in the state record. The convention we employ here is to name these fields with a -′ suffix, e.g. *n*-′. This supports the use of an antiquotation syntax in assertions, where ´*n* refers to the variable in the current state and ^{s}n in state *s*, i.e. *n*-′ *s*. The type of this field is an encoding of the C type in the HOL type system, described in §2.4. The verification environment limits the lifetime of these variables to the enclosing block. A difficulty arises when there are two or more variables of the same name with different types — we simply rule this

	Usage		**Size (words)**		
	Pass	**Return**	1	2	> 2
C++ method invocation	385	68	352	64	0
Pass-by-reference	8	28	2	20	6

Table 2.1: Address derivation in the L4Ka::Pistachio μ-kernel source.

out as a possibility in the C_{sys} subset, but this is a technical issue which could be resolved with appropriate namespace mangling.

Our treatment of automatic variables produces reasonable verification goals when programs operate on local state as we have abstracted from the details of stack frames and variable allocation. The most significant drawback of this is that we are unable to easily support references or pointers to automatic variables, and these are excluded from the C_{sys} subset. We justify this on the grounds that the focus in this thesis is on memory models and proof abstractions for pointer programs featuring inductively-defined data structures. If we accept the hypothesis that the main use of references to local storage in systems code is to allow a callee to return by reference rather than by value, then this has little influence on such data structures — indirection is being used merely to improve efficiency by reducing copying.

The validity of this hypothesis is not immediately obvious. To provide some evidence for this claim, we have undertaken a study of the use of pointers to automatic variables in the L4Ka::Pistachio 0.4 kernel, targeting the ARM/StrongARM/PLEB platform. The compiler flags were extended in the build scripts with the `-fdump-tree-original` option, instructing GCC to generate the corresponding abstract syntax tree for each function. Occurrences of the & operator were identified through `addr_expr` nodes. Stack-allocated arrays were also examined manually with `grep`. Kernel debug functions were ignored, as were method invocations on anonymous objects. The results from 137 function bodies in 25 files are summarised in Table 2.1. The first column gives the number of occurrences of pass-by-reference where & appears. This is subdivided into cases where the data is being passed and returned, and between the cases where the call is an implicit self-reference in a C++ method invocation and where an explicit pointer parameter is given. The second column gives the size of the data being transferred.

Almost all operations taking the address of an automatic variable are C++ method invocations, where a self-reference for the invoked object is provided. These are typically word-sized objects such as **threadid_t**. When operator overloading is utilised, sometimes a reference is also provided as one of the arguments to another object, e.g. for equality comparisons, or when more than one value is effectively returned as in an overloaded post-increment. Since the objects are only word sized, it would be possible and in fact likely

that references would not be used in a pure C implementation of the kernel, as the indirection is not warranted in these cases.

The remaining appearances of & occur explicitly in the code. The bulk of these are related to the **lookup_mapping** and **calculate_errorcodes** functions that return 2 words. A call to **perform_exregs** requires the transfer in and out by reference of 6 words. Rewriting these to take and return a **struct** would be possible and an optimising compiler should be able to hide the overhead if this is a problem.

No use of pointer casting or arithmetic on the result of local & obtained pointers was observed. Neither was the linking of automatic storage to or from inductively-defined data structures in the store or heap. All stack-allocated arrays were used only within the declaring functions, and only directly through array indexing and update, i.e. they were not passed as arguments nor converted to pointer values. Automatic arrays could then, for this code base, be modelled as fixed-sized lists of the appropriate type.

In summary, it can be seen that the use of automatic storage is significantly different to the general case of memory in Pistachio, and that prohibiting taking the address of automatic variables does not reduce the applicability of C_{sys} to the kernel.

In other code bases, it may be necessary to model these features. A reasonable solution in the absence of recursion would be to treat automatic variables as static variables (described below), having fixed addresses allocated in some global memory pool, with addresses given by some underspecified function. It would then be necessary to introduce guards in the language embedding to avoid the possibility of dangling pointers to stack frames.

Static variables

Static variables are referred to through their addresses, contained in a pointer constant of the same name with an *-addr* suffix, e.g. variable *n* would have its address in *n-addr*. All accesses and updates to these variables are translated as dereferences. A locale [5] provides the assumptions that these constants reside in a distinct region of memory and are disjoint — to discharge the obligation this assumption introduces we require that more concrete details are made available about variable allocation. Since this is an entirely standard assumption to make, usually inside the semantics tacitly, this does not need to occur for most verifications other than those where there is genuine concern over whether there may be sufficient space for static variable placement.

In contrast to automatic variables, there are static variables in the Pistachio source code that have their addresses taken, linked with inductive data structures on the heap and are involved in pointer casts, e.g. *kfree-list* in §6.2. Hence these variables do play a role in the proof abstractions related to memory and this level of abstraction is required to provide sensible semantics for these language features.

Example 2.3.3. Table 2.2 provides a running example in this chapter, finding the minimum value x in some ordered binary tree y. The state space representation for this program, *globals state*, in the verification environment is[2]:

consts		x-addr	::	*word32 ptr*
		y-addr	::	*bin-tree ptr*
record *globals*	=	h-state	::	*heap-mem*
record γ *state*	=	t-′	::	*bin-tree ptr*
		tree-min-ret-′	::	*word32*
		globals	::	γ

The distinction between the scoped variables used to model C's automatic variables and globally persistent variables, in this case used for the heap mapping function, is made explicit through the use of distinct **record**s. In addition to the function parameter t, a variable is used for the return value *tree-min-ret*. This is a feature of the verification environment that allows a clean separation of procedure call statements and the later use of the returned value in expressions. Types are explained in the next section.

```
int x;

struct bin_tree {
    int item;
    struct bin_tree *l, *r;
} y;

int tree_min(struct bin_tree *t)
{
    while (t->l)
        t = t->l;

    return t->item;
}

void f(void)
{
    x = tree_min(&y);
}
```

Table 2.2: **tree_min** definition.

[2]This example is based on the preceding discussion in this chapter. The state representation used in actual verifications includes additional global fields introduced later, such as the heap type description in Chapter 3 and exception type variable in §2.5.4.

2.4 Type encoding

Each language type is assigned a unique type in the theorem prover's logic. This allows for both an intuitive definition of language operators as functions in HOL, and the harnessing of the theorem prover's type inference mechanism to avoid unnecessary type annotations in assertions and proofs.

For each C language type, the following steps are taken:

1. A unique type is assigned to model it in the theorem prover. For example, for **char**, we can use *word8*. If another language type had a similar representation in HOL, we would need to create a distinct type, based on or with the same properties as *word8*. This is because later we rely on the fact that distinct language types have distinct HOL types to recreate the abstraction of multiple typed heaps in Chapter 3. If there existed synonyms here, we would be left with the undesirable result of identical typed heaps for the responsible language types.

2. The type is placed in a type class providing functions to allow heap dereferencing expression semantics to be defined.

   Definition 2.4.1. The α::*c-type* class introduces several constants that connect the low-level byte representation, referred to in the standard as the *object representation* [1, 6.2.6.1–4], and HOL values for encoded language types:

   $$\begin{array}{lll} \textbf{consts} & \mathsf{to\text{-}bytes} & :: \ \alpha \Rightarrow \mathit{byte\ list} \\ & \mathsf{from\text{-}bytes} & :: \ \mathit{byte\ list} \rightharpoonup \alpha \\ & \mathsf{typ\text{-}tag} & :: \ \alpha\ \mathit{itself} \Rightarrow \mathit{typ\text{-}tag} \\ & \mathsf{typ\text{-}info} & :: \ \alpha\ \mathit{itself} \Rightarrow \mathit{typ\text{-}info} \end{array}$$

   Example 2.4.1. Fig. 2.2 illustrates the relationship between the functions in the α::*c-type* class for a 32-bit integer type with a mixed-endian representation.

   The functions to-bytes and from-bytes convert between Isabelle values and lists of bytes suitable for writing to or reading from the raw heap state. from-bytes is partial to allow for object representations that do not correspond to a value of a type, known as *trap representations* [1, 6.2.6.1–5]. The function typ-tag associates a unique *type tag* with each α::*c-type*, providing a means of treating language types as first-class values in HOL. Type tags are somewhat similar to the standard's notion of a *tag* [1, 6.7.2.3], which is essentially the identifier naming a **struct**, **union** or **enum** type. *typ-info* contains enough structure to allow size and alignment information for the type to be calculated, as described in §2.4.1. We use this to define functions size-of :: α::*c-type itself* $\Rightarrow$

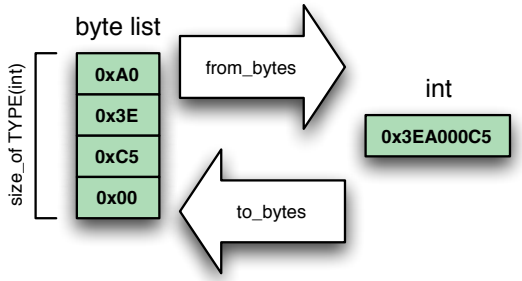

Figure 2.2: Mixed-endian integer encoding.

nat and align-of :: α::*c-type itself* $\Rightarrow$ *nat* respectively. The syntax T_τ and T_t are used for typ-info T and typ-tag T respectively.

Structured types may have internal padding fields that are not reflected at the HOL level [1, 6.2.6.1–6, 6.7.2.1–13]. Correct treatment of these either requires failure semantics for padding field access, which can be achieved with the technique described in §2.5.5, or a model for the implementation's treatment of padding. The latter may require to-bytes to be extended as in §5.2.2 where we revisit the function with a focus on structures. The standard also allows for the fact that to-bytes may provide more than a single object representation [1, 6.2.6.1–8]. If such behaviour were to be modelled we would also need to extend these definitions using non-determinism or the technique described in the footnote of page 132.

3. A set of properties are automatically shown by the C_{sys} parser to hold for the encoded type, placing the type in the α::*c-type* subclass α::*mem-type*. These properties are provided by the C standard and/or implementation and support proofs featuring language types.

   Definition 2.4.2. The α::*mem-type* axiomatic type class requires the following properties to hold on an α::*c-type* for instantiation:

$$\begin{array}{rl}
\text{from-bytes (to-bytes } x) = \lfloor x \rfloor & [\textsc{Inv}] \\
|\text{to-bytes } (x::\alpha)| = \text{size-of TYPE}(\alpha) & [\textsc{Len}] \\
0 < \text{size-of TYPE}(\alpha) & [\textsc{SzNZero}] \\
\text{size-of TYPE}(\alpha) < |\text{addr}| & [\textsc{MaxSize}] \\
\text{align-of TYPE}(\alpha) \text{ dvd } |\text{addr}| & [\textsc{Align}] \\
\text{align-of TYPE}(\alpha) \text{ dvd size-of TYPE}(\alpha) & [\textsc{AlignDvdSize}]
\end{array}$$

where the constant |addr| represents the size of the address space, e.g. 2^{32}.

These conditions follow mostly from requirements in the C standard, with the exception of the alignment constraint [Align] which we add to make pointer arithmetic better behaved, and which holds with the power-of-two alignment restriction. Providing the intended semantics of operators such as assignment [1, 6.5.16–3] implies [Inv]. [Len] gives that object representations have a constant size for all values [1, 6.2.6.1–4]. The [SzNZero] axiom is stated explicitly in [1, 6.2.6.1–2] and [MaxSize] is an obvious prerequisite for any type whose value is to be stored in an object. Finally, for tiling in an array, [AlignDvdSize] is needed.

Following this process, the encoded language type may be used in the program's state and appear inside translated expressions and statements.

2.4.1 Type information

Definition 2.4.3. To derive the required size and alignment information from *typ-info*, we include the structure of the type in the value:

$$\textbf{datatype}\ \textit{typ-info} \quad = \quad \mathsf{TypScalar}\ \textit{nat}\ \textit{nat}\ \mid \\ \mathsf{TypAggregate}\ (\textit{typ-info} \times \textit{field-name})\ \textit{list}$$

where *field-name* is a *string*.

This nested recursive definition describes a tree structure, capable of describing C's aggregate types as well as its scalar types. The structure of a scalar type consists of the constructor TypScalar followed by its size and alignment. Size and alignment for aggregate types is implicit and calculated below. There is not a one-to-one correspondence between fields in this structure and those in a C **struct**, as fields in this definition are also intended to explicitly represent padding.

Example 2.4.2. Fig. 2.3 gives several instances of *typ-info*. The C type corresponding to the type information is indicated in parentheses at each node in the trees. The alignment in the case of **long** is less than the size, and in the **struct x** and array examples the size and alignment are such that no additional padding is required.

Definition 2.4.4. Type size is calculated with:

$$\mathsf{size\text{-}of}\ t \equiv \mathsf{typ\text{-}size}\ t_\tau$$

where typ-size is defined in Table 2.3.

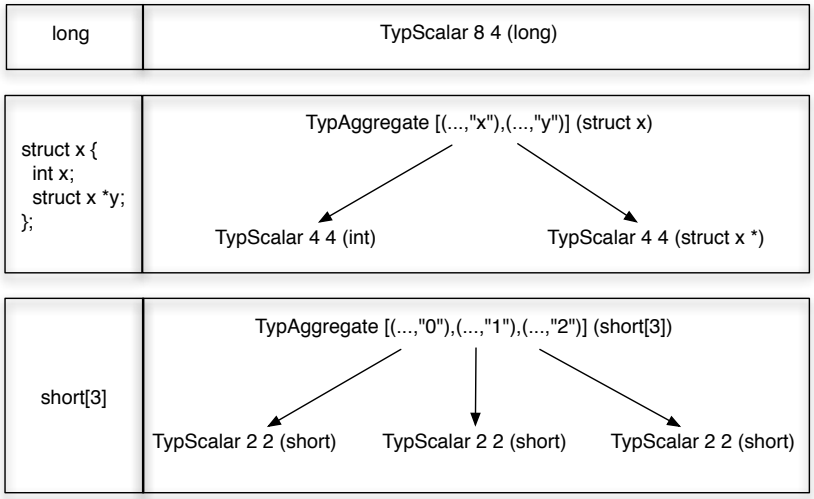

Figure 2.3: Example *typ-info*s.

typ-size (TypScalar sz $algn$)	$\equiv$	sz
typ-size (TypAggregate ts)	$\equiv$	typ-size-list ts
typ-size-list []	$\equiv$	0
typ-size-list $(x{\cdot}xs)$	$\equiv$	typ-size-pair x + typ-size-list xs
typ-size-pair $\langle t,n \rangle$	$\equiv$	typ-size t

Table 2.3: typ-size definition.

An advantage of explicit representation of padding is that size can be found easily, without having to simultaneously reason about alignment, as would be necessary if padding had to be inferred while finding the size.

Definition 2.4.5. Type alignment is calculated with:

$$\text{align-of } t \equiv \text{typ-align TYPE}(\alpha)_\tau$$

where typ-align is defined in Table 2.4.

typ-align (TypScalar sz $algn$)	$\equiv$	$algn$
typ-align (TypAggregate ts)	$\equiv$	typ-align-list ts
typ-align-list []	$\equiv$	0
typ-align-list $(x{\cdot}xs)$	$\equiv$	max (typ-align-pair x) (typ-align-list xs)
typ-align-pair $\langle t,n \rangle$	$\equiv$	typ-align t

Table 2.4: typ-align definition.

As required by the C standard, structured types must be aligned such that all members are aligned [1, 6.7.2.1–12]. From this we can infer that the offset of a field must be divisible by the alignment of a field's type so that this property holds when placed at the base of the address space. Hence the aggregate alignment must be also divisible by the field's type alignment. Thus aggregate type alignment is a multiple of the LCM of the individual field alignments. Since we are restricting alignment to 2^n for some n, this is equivalent to a multiple of the maximum of field alignments.

Defn. 2.4.5 captures a necessary condition. The implementation may impose a stricter alignment. In that case, the above definition would require strengthening and/or additional alignment guards can be introduced.

In Chapter 5 we extend *typ-info* and reformulate the theory such that the relationship between to-bytes, from-bytes, typ-tag and typ-info is made clearer. There is a potential for a mismatch between *typ-info*s. For example, the *typ-info* for a **short** may state the alignment is 2, but a **short** field of some other **struct** type may have alignment as 1. This does not pose a problem for the theory in Chapter 3 and Chapter 4, as these developments do not utilise the internal structure of types, but in Chapter 5 we also discuss the well-formedness and consistency conditions that must hold to avoid problems like this.

2.4.2 Scalar types

C provides a number of types to manipulate integer, floating-point data and addresses. These are fixed for a specific implementation [1, 6.2.6.1–2]. In this section we detail how they are treated in the semantics and where appropriate how they are instantiated as α::*c-type*s and α::*mem-type*s.

Void

The **void** type is an *incomplete type* [1, 6.2.5–1], meaning it lacks sufficient information to calculate size, and may not be made complete [1, 6.2.5–19]. We mention it here as its derived pointer type is somewhat more interesting. It is an empty type, but since such types are not available in Isabelle/HOL's type system, we model this with the singleton type *unit*. As there is no size information we do not instantiate **void** as an α::*mem-type*.

Integer

C provide a number of distinct integer types of various sizes, given by the types **char**, **short int**, **int**, **long int** and **long long int** [1, 6.2.5–4]. These are available in both **signed** and **unsigned** flavours [1, 6.2.5–6]. Enumerated types are additional integer types, which consist of a set of constants from one of these types [1, 6.2.5–16].

The relationship between the ranges of values represented by different integer types with the same sign is given by $|char| \leq |short\ int| \leq |int| \leq |long\ int| \leq |long\ long\ int|$ [1, 6.2.5–8, 6.3.1.1–1]. The non-negative range of a **signed** type is a subset of the range of a corresponding **unsigned** type [1, 6.2.5–9]. There are minimum sizes given by the standard for the integer types, found in the header file `limits.h`, e.g. **unsigned int**s must have a maximum value of at least $2^{16} - 1$ [1, 5.2.4.2.1–1].

The standard provides some specification of how integer types are encoded. It mandates a binary representation [1, 6.2.6.2–1] and requires that bits in the object representation either are used to encode the value or are used for padding. It also leaves much to the implementation, for example, whether signed types are sign and magnitude, one's complement or two's complement [1, 6.2.6.2–2]. Padding bits can also be used for trap representations, e.g. as parity bits. The ordering of the bits representing the value is not specified, and so there are various possibilities such as little-, mixed- or big-endian encodings.

In this thesis we mostly make use of the **unsigned** integer types, and provide the details for this instantiation below. It is not difficult to see how this may be extended to **signed** and **enum** integer types. We represent these types with bit-vectors of the appropriate width[3], in contrast to the usual approach of using mathematical integer types such as *nat* and *int*.

Definition 2.4.6. Unsigned integer types are α::*c-types*[4]:

$$\begin{array}{lcl}
\mathsf{to\text{-}bytes}\ w & \equiv & \mathsf{rev}\ (\mathsf{word\text{-}rsplit}\ w) \\
\mathsf{from\text{-}bytes}\ bs & \equiv & \mathsf{if}\ |bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha\ word)\ \mathsf{then}\ \lfloor \mathsf{word\text{-}rcat}\ (\mathsf{rev}\ bs) \rfloor\ \mathsf{else}\ \bot \\
t_t & \equiv & \mathit{''word''}\ @\ \mathsf{replicate}\ (\mathsf{len\text{-}of}\ \mathsf{TYPE}(\alpha))\ \mathsf{CHR}\ \mathit{''1''} \\
t_\tau & \equiv & \mathsf{let}\ sz = \mathsf{len\text{-}of}\ \mathsf{TYPE}(\alpha)\ \mathsf{div}\ 8\ \mathsf{in}\ \mathsf{TypScalar}\ sz\ sz
\end{array}$$

where word-rsplit takes a bit-vector and returns a representation as a list of bytes, most-significant-byte first, and word-rcat has the property that word-rcat (word-rsplit w) $= w$. The tag is simply a string and we make it unique for each distinct integer type by appending a unary encoding of the width. len-of provides the number of bits in the index type, where t::α::*len8 word itself*. The α::*len8* type class restricts the index type to bit lengths divisible by a byte, with the axioms:

$$\begin{array}{rl}
8\ \mathsf{dvd}\ \mathsf{len\text{-}of}\ \mathsf{TYPE}(\alpha) & [\textsc{Len8Dv8}] \\
\mathsf{len\text{-}of}\ \mathsf{TYPE}(\alpha)\ \mathsf{div}\ 8 < |\mathsf{addr}| & [\textsc{Len8Sz}] \\
\mathsf{len\text{-}of}\ \mathsf{TYPE}(\alpha)\ \mathsf{div}\ 8\ \mathsf{dvd}\ |\mathsf{addr}| & [\textsc{Len8Dvd}]
\end{array}$$

[3] The implementation currently uses the same type for both signed and unsigned integers of a given width. This does not have soundness implications, but could reduce the gains from the multiple typed heaps abstraction developed in the next chapter. In any case, this can be fairly easily rectified with some additional phantom typing in the bit-vector library.

[4] Here we use a 32-bit little-endian encoding as might be witnessed on the x86 or ARM architectures.

Theorem 2.4.1. *Unsigned integer types are* α*::mem-types:*

Proof. We discharge the proof obligations resulting from each of the class axioms:

- [Inv] — from rev (rev xs) = xs, word-rcat (word-rsplit w) = w and |word-rsplit w| = len-of TYPE(α) div *8*.
- [Len] — also from |word-rsplit w| = len-of TYPE(α) div *8*.
- [SzNZero] — by [Len8Dv8].
- [MaxSize] — by [Len8Sz].
- [Align] — size and alignment are the same, hence [Len8Dvd].
- [AlignDvdSize] — as with [Align].

□

Example 2.4.3. Alternative encodings are also supported. For example, an implementation with a singly redundant encoding of **unsigned char**s could have the instantiations performed with the following definitions:

to-bytes w	$\equiv$	$[w, w]$
from-bytes bs	$\equiv$	if $\lvert bs\rvert = 2 \wedge bs_{[0]} = bs_{[1]}$ then $\lfloor bs_{[0]} \rfloor$ else $\bot$
t_t	$\equiv$	$''uchar''$
t_τ	$\equiv$	TypScalar *2 2*

Floating point

C has three floating-point types [1, 6.2.5–10], **float**, **double** and **long double**. Since these do not appear inside the systems we study in this thesis we omit an encoding. We are not presently aware of any limitation in using our style of type encoding for floating-point types.

Pointer

From any α*::c-type* a pointer type may be derived [1, 6.2.5–20]. This applies recursively. The C standard provides few restrictions on the representation and alignment of pointers [1, 6.2.5–26], and it is possible for pointers derived from different types to be represented differently. They are convertible to integer types and back in an implementation-defined manner [1, 6.3.2.3–5]. In C_{sys} we assume an implementation that has an interchangeability with word-sized integer address representations.

We introduce a distinct Isabelle pointer type for each Isabelle type, used to model C pointer types:

$$\textbf{datatype}\ \alpha\ ptr \quad = \quad \mathsf{Ptr}\ addr$$

The additional α on the left-hand side can now be used to associate the pointer type information with pointer values in Isabelle's type system. Since the type variable does not appear on the right-hand side it is a phantom type. Nonetheless, the type information is used to constrain the action of various pointer operators by making use of the type information associated with an α value. The destructor ptr-val retrieves the address from a pointer value; we write $p_{\&}$ as an abbreviation of ptr-val p. The pointer types α::*c-type ptr* can be shown to be instances of α::*mem-type* for all pointer types.

Definition 2.4.7. Pointer types are α::*c-type*s:

$$\begin{array}{lcl}
\mathsf{to\text{-}bytes}\ p & \equiv & \mathsf{rev}\ (\mathsf{word\text{-}rsplit}\ p_{\&}) \\
\mathsf{from\text{-}bytes}\ bs & \equiv & \mathsf{if}\ |bs| = 4\ \mathsf{then}\ \lfloor \mathsf{Ptr}\ (\mathsf{word\text{-}rcat}\ (\mathsf{rev}\ bs))\rfloor\ \mathsf{else}\ \bot \\
p_t & \equiv & \mathsf{TYPE}(\alpha)_t\ @\ ''{+}ptr'' \\
p_\tau & \equiv & \mathsf{TypScalar}\ 4\ 4
\end{array}$$

Again, we create a unique string for each pointer type's tag to allow us to treat C types as first-class values in HOL.

Theorem 2.4.2. *Pointer types are* α::*mem-types:*

Proof. By Thm. 2.4.1 as encoding is just a special case of Defn. 2.4.6. □

Later on, in §2.5.6, we need to express the effects of pointer addition. The following polymorphic function provides this.

Definition 2.4.8. Pointer addition is defined for p::α::*c-type ptr* as:

$$\mathsf{Ptr}\ p\ +_p\ n \equiv \mathsf{Ptr}\ (p + n * \mathbb{N}^{\Rightarrow}\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)))$$

2.4.3 Aggregate types

C allows for the grouping of related objects as aggregate types. Here we explore the treatment of these types in the C_{sys} type encoding.

Array

As with pointers, any α::*c-type* has a corresponding derivable array type [1, 6.2.5–20]. Arrays are not first-class types in C. There are two distinct situations in which they require treatment. First, when appearing as objects in the heap. Here, array expressions are given pointer semantics [1, 6.3.2.1–3]. We do not require pointer arithmetic to be restricted to the bounds of a

memory object, as with ROLM, in contrast to the strict C standard [1, 6.5.6–8] which provides for undefined behaviour here. In the other situation, where arrays appear as members of other objects or as automatic variables, they must have a constant size and are represented using the techniques of Harrison [39]. In this case, array α::*c-type* definitions and α::*mem-type* instantiations are similar to those of **struct**s, as the fixed size allows us to treat each object in the array like a field.

Structure

structs can be used to describe a sequentially ordered collection of α::*c-type* objects [1, 6.2.5–20]. We can model a **struct** using Isabelle/HOL's **record** or **datatype** types, to-bytes/from-bytes using the member types' functions and type structure in *typ-info*. We also need to insert padding fields between fields appearing in the declaration and at the end to ensure alignment restrictions are met. While it is possible to describe **struct**s in this way, as we did in an earlier encoding [96], and instantiate with an Isabelle/HOL tactic as α::*mem-type*s, we present a more compelling approach in Chapter 5.

structs may contain bitfields as members, which are integer types with a specifiable width [1, 6.2.6.1–4]. They present a problem in our setting as they differ from other α::*c-type*s by operating on a potentially sub-byte granularity. The solution is to rule them out as fields of regular **struct**s, but instead allow a special form **struct** which may only contain bit-fields that fit exactly inside their corresponding integer type. This also has the benefit of simplifying the understanding required of the compiler's **struct** packing.

Union

unions describe a collection of overlapping objects [1, 6.2.5–20]. Only one of these objects may be accessible at any time [1, 6.7.2.1–14], although if some members have a common prefix then any fields in the common prefix are accessible through any relevant member [1, 6.5.2.3–5].

The treatment of **union**s in ROLM and the type encoding presents several problems. One may first ask what Isabelle/HOL type may correspond to these C types. A disjoint union in the form of a **datatype** is the most straightforward answer, but a **datatype** carries around information additional to the current member's value, i.e. which member is active. This information does not appear in the memory contents of the object representing the **union**, except in some well-behaved cases such as tagged unions, where a common field in each **union** member provides a tag indicating the currently active member. A fixed array of *byte*s could be used to model union values, possibly abstracted as a quotient type with to-bytes representation equivalence, at the expense of losing a conveniently typed encoding.

Under ROLM, objects may overlap but only hierarchically. We could

simply consider a snapshot of the current active members in an aggregate type featuring **union**s as providing the object type, which may be expressed in terms of **struct**s, however this requires each time an active member changes the entire object have its type changed. This is particularly a problem for a **union** nested deep inside an aggregate type as assignment has effects that are non-local.

To avoid these problems, we introduce some restrictions that could allow **union**s to work in C_{sys}. We can drop **union**s as first-class types, prohibit them as automatic variables and from being nested inside aggregate types. This would then allow us to give a semantics in terms of their members and express the . and −> operators in terms of pointer casts.

Performing a similar analysis with **union**s as in §2.3.3, we found 18 type definitions in ARM L4Ka::Pistachio. Of these, 12 were simply reinterpretations of a type as a sequence of words or bytes, 2 were tagged unions, 2 supported disjoint use of the same memory based on external tags or implicit in the code path, 1 contained only a single field and 1 supported a non-trivial punning of page table entries as address space specific variables. Three of the **union**s were nested, the last being the most tricky. Here, the **union** supporting the pun contained an array of page table entries, themselves a **union** of disjoint **union**s. We hypothesise that it would be reasonable to rewrite the code to use explicit casting in the nested cases to support the above restrictions.

Since no **union** types were present in our case study, they are currently unimplemented in our translation and we leave to future work an improved treatment of C **union**s.

2.5 C_{sys}-*com* translation

The syntax for the C_{sys} language is provided in Appendix A. The grammatical constructs are mostly a subset of corresponding constructs in the C standard, however there are several places where new non-terminals are introduced — here the language recognised does not change as the additional non-terminals have a correspondence with a production in the standard C grammar. In addition, we do not describe **union**s, bit-fields or **enum**s. While in previous sections we have stated sensible encodings for these types, these are not yet implemented in the translation tool at the time of writing.

Semantics are provided through a two-stage process. First, the abstract syntax tree for C_{sys} undergoes a mapping, described below, to the *com* language introduced in §2.5.1, then the existing operational semantics for *com* [85] yields a program's meaning.

Fig. 2.4 provides the flow for the C_{sys}-*com* mapping, referred to henceforth as *translation*, and the verification process.

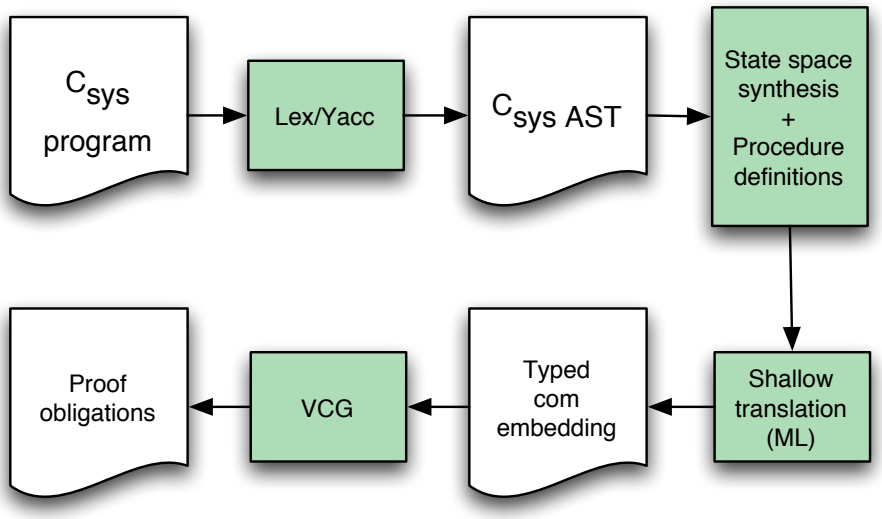

Figure 2.4: C_{sys} translation and verification processes.

2.5.1 *com* syntax and semantics

In this section we summarise the *com* language and verification environment of Schirmer [85]. This language is intended to be a generic target for embeddings of sequential imperative programming languages, and provides a number of primitive constructs which the language features of C_{sys} can be mapped to. We consider the use of an existing verification framework, complete with a machine-checked HOL operational semantics, Hoare logic and verification condition generator, a distinct advantage, as it has allowed us to focus on the goals of the thesis without having to develop these specialised but somewhat standard theories and tools from scratch, as is common in the verification literature.

Definition 2.5.1. The syntax of the *com* language is given by:

types σ *bexp*	=	σ *set*
datatype (σ, η, χ) *com*	=	Skip
		| Basic "$\sigma \Rightarrow \sigma$"
		| Spec "$(\sigma \times \sigma)$ *set*"
		| Seq "(σ, η, χ) *com*" "(σ, η, χ) *com*"
		| Cond "σ *bexp*" "(σ, η, χ) *com*" "(σ, η, χ) *com*"
		| While "σ *bexp*" "(σ, η, χ) *com*"
		| Call η
		| DynCom "$\sigma \Rightarrow (\sigma, \eta, \chi)$ *com*"
		| Guard χ "σ *bexp*" "(σ, η, χ) *com*"
		| Throw
		| Catch "(σ, η, χ) *com*" "(σ, η, χ) *com*"

The type parameters allow one to select a state space σ suitable for an embedding. In our case we synthesise a state space during translation based on §2.3. η is used for procedure names, and χ to name fault conditions.

Basic models the atomic transitions in the embedded language's state space, e.g. variable assignment or heap update. Seq provides sequential composition, Cond and While conditional and iterative statements respectively. It should be observed that the condition expressions are side-effect free. Spec can be used to describe non-deterministic transitions, but we do not make any use of this in the following. Function call semantics map to Call and DynCom statements. Throw and Catch give an exception mechanism with which one can model abrupt termination in language statements like **return**. Finally, Guards lead to failure semantics if guard expressions do not hold prior to execution of parameter programs. We drop the χ parameter when describing Guard programs in the following, as from a soundness point-of-view the cause of failure semantics is irrelevant.

com provides a mixed embedding, where statements are deeply embedded using the **datatype** constructors, but expressions and the core language state transformers are HOL sets and functions. *com* statements form the primitives for our embedding, where C_{sys} statements such as **for** and **break** translate to multiple statements in *com*.

Further details on *com* are available from Schirmer [85], who describes a small- and big-step operational semantics, Hoare logics for partial and total correctness, soundness and completeness proofs, a verification condition generator and the embedding of many higher-level language features together with the C0 [53] language.

Details of the translation of C_{sys} statements and expressions to *com* are given in the next sections.

2.5.2 Notation

As the translation takes place externally to the theorem prover's logic, we do not have the luxury of Isabelle/HOL machine-checked theories to describe the process. Instead we introduce some semi-formal notation here. We write:

- $\langle X \rangle_T$ for the Isabelle/HOL α::*c-type* of C_{sys} expression X, described in §2.5.3. When we wish to introduce a type variable that is restricted to the class of integer types, α::*c-type* is written τ^A, and for α::*c-type* *ptr*s we use τ^P.
- $\langle P \rangle_S$ for the *com* object translation of C_{sys} statement P, described in §2.5.4.
- $\langle X \rangle_G$ for the guard for C_{sys} expression X, described in §2.5.5.
- $\langle X \rangle_E$ for the side-effect free C_{sys} expression X's embedding $\sigma \Rightarrow \langle X \rangle_T$, where σ is the program's state space type. This is described in §2.5.6.
- $\langle X \rangle_A$ for the function $\sigma \Rightarrow addr$ giving the address of lvalue X's object in program state σ, described in §2.5.7.
- $\langle X \rangle_L$ for the state update function $\langle X \rangle_T \Rightarrow \sigma \Rightarrow \sigma$, that takes a value of type $\langle X \rangle_T$ and updates the program's state σ, representing assignment to modifiable lvalue expression X. This is described in §2.5.7.
- $\langle f \rangle_F$ for the vector $\langle r, p_1, p_2, \ldots, p_n \rangle$, where f is a C_{sys} function with n formal parameters, r is a variable in the program's state space for passing f's return value and p_m gives a variable for each formal parameter for value passing during calls.
- $\bot$ for translation failure resulting from a missing rule.

2.5.3 Types

The typing of C_{sys} expressions is described in Table 2.5. This is a type translation, rather than judgement, as the intent here is to describe a process rather than provide an explicit semantics. There is not sufficient information in this table to perform type checking, as the expression, statement and lvalue translations also contain type restrictions that do not require duplication here. Table 2.6, Table 2.7 and Table 2.8 give various type promotion conversions that occur inside expressions, in the situation where an operator requires unification of the types in its sub-expressions or with the expected type of the expression.

The translation mostly follows the standard for this subset, however on occasion we have made some implementation assumptions, in particular we assume **long** and **int** to be the same width in the conversions, **short** to be

TypCond
$\langle e_0 \; ? \; e_1 \; : \; e_2 \rangle_T \;=\; \mathsf{cond\text{-}conv}\ \langle e_1 \rangle_T \ \langle e_2 \rangle_T$

TypPlusPtr
$$\frac{\langle e_0 \rangle_T = \tau^P \vee \langle e_1 \rangle_T = \tau^P}{\langle e_0 + \; e_1 \rangle_T \;=\; \tau^P}$$

TypPlus
$$\frac{\langle e_0 \rangle_T = \tau_0^A \qquad \langle e_1 \rangle_T = \tau_1^A}{\langle e_0 + \; e_1 \rangle_T \;=\; \mathsf{arith\text{-}conv}\ \tau_0^A \ \tau_1^A}$$

TypSubPtr
$$\frac{\langle e_0 \rangle_T = \tau^P \qquad \langle e_1 \rangle_T = \tau^A}{\langle e_0 - e_1 \rangle_T \;=\; \tau^P}$$

TypSubPtrDiff
$$\frac{\langle e_0 \rangle_T = \tau^P \qquad \langle e_1 \rangle_T = \tau^P}{\langle e_0 - e_1 \rangle_T \;=\; \mathit{sint}}$$

TypSub
$$\frac{\langle e_0 \rangle_T = \tau_0^A \qquad \langle e_1 \rangle_T = \tau_1^A}{\langle e_0 - e_1 \rangle_T \;=\; \mathsf{arith\text{-}conv}\ \tau_0^A \ \tau_1^A}$$

TypArith
$\langle e_0 \ f \ e_1 \rangle_T \;=\; \mathsf{arith\text{-}conv}\ \langle e_0 \rangle_T \ \langle e_1 \rangle_T$ where $f \in \{*, /, \%, \&, \mid, \hat{}\}$

TypLogicRelEq
$\langle e_0 \ f \ e_1 \rangle_T \;=\; \mathit{sint}$ where $f \in \{\,||\,, \&\&, <, <=, >, >=, ==, !=\}$

TypShift
$\langle e_0 \ f \ e_1 \rangle_T \;=\; \mathsf{integral\text{-}promote}\ \langle e_0 \rangle_T$ where $f \in \{<<, >>\}$

TypCast
$\langle (t) e \rangle_T \;=\; \tau$ where τ is an α::*c-type* encoding for type t

TypSizeOf
$\langle \mathbf{sizeof}\ e \rangle_T \;=\; \mathit{sint}$

TypAddrOf
$\langle \&e \rangle_T \;=\; \langle e \rangle_T \ \mathit{ptr}$

TypDeref
$$\frac{\langle e \rangle_T = \tau \ \mathit{ptr}}{\langle *e \rangle_T \;=\; \tau}$$

TypUnary
$\langle f \ e \rangle_T \;=\; \mathsf{integral\text{-}promote}\ \langle e \rangle_T$ where $f \in \{+\,, -, \tilde{}\,\}$

TypNot
$\langle !e \rangle_T \;=\; \mathit{sint}$

TypDot
$\langle e\,.\,n \rangle_T \;=\; \tau$ where $\langle e \rangle_T$ is a **struct** type with field n::τ

TypArrow
$\langle e{-}{>}n \rangle_T \;=\; \tau$ where $\langle e \rangle_T$ is a pointer to a **struct** type with field n::τ

TypArray
$$\frac{\langle e_0 \rangle_T = \tau \ \mathit{ptr} \vee \langle e_0 \rangle_T = (\tau,n) \ \mathit{array}}{\langle e_0 [e_1] \rangle_T \;=\; \tau}$$

TypConst
$\langle c \rangle_T \;=\; \mathit{sint}$ where c is a constant

TypVar
$\langle v \rangle_T \;=\; \tau$ where v::τ is an in-scope variable

TypParen
$\langle (e) \rangle_T \;=\; \langle e \rangle_T$

Table 2.5: Type translations.

arith-conv	*ulong*	τ	=	*ulong*
	τ	*ulong*	=	*ulong*
	uint	*slong*	=	*ulong*
	slong	*uint*	=	*ulong*
	slong	τ	=	*slong*
	τ	*slong*	=	*slong*
	uint	τ	=	*uint*
	τ	*uint*	=	*uint*
	τ_0	τ_1	=	*sint*

Table 2.6: Arithmetic type conversions [1, 6.3.1.8–1].

cond-conv	τ_0^A	τ_1^A	=	arith-conv τ_0^A τ_1^A
	τ	τ	=	τ
	τ^P	*void ptr*	=	*void ptr*
	void ptr	τ^P	=	*void ptr*

Table 2.7: Conditional type conversions [1, 6.5.15–5].

narrower than **int** and an interchangeability of pointer values and **unsigned int**s. The semantics for conversion are given in Table 2.9. Here, scast and ucast are polymorphic (in the size and sign of the word) bit-vector library functions that perform the appropriate size and sign conversions. Not all conversions are valid in all situations, e.g. the conversions performed in an assignment are more restricted than in a type cast. The details are deferred to the specific translation rule below.

2.5.4 Statements

Here we describe the translation of the `statement` non-terminal in Appendix A. C_{sys} has most of the statements found in the C standard [1, 6.8], however it lacks some of the statements that result in control flow that is cumbersome to express in *com*. Specifically, **switch** and **goto** statements are not in the language. **switch** is not a simple case distinction in C, but provides the ability to enter a **case** anywhere inside the statement's scope [1, 6.8.4.2–4]. Both statements violate the block structure of a program, and their semantics

integral-promote	{*uchar*, *schar*, *ushort*, *sshort*}	=	*sint*
	τ^A	=	τ^A

Table 2.8: Integral type promotion [1, 6.3.1.1–2].

convert	τ	τ	f	$=$	f
	$\{$*schar*, *sshort*, *sint*, *slong*$\}$	τ^A	f	$=$	scast $\circ$ f
	$\{$*uchar*, *ushort*, *uint*, *ulong*$\}$	τ^A	f	$=$	ucast $\circ$ f
	$\{$*schar*, *sshort*, *sint*, *slong*$\}$	τ^P	f	$=$	scast $\circ$ ptr-val $\circ$ f
	$\{$*uchar*, *ushort*, *uint*, *ulong*$\}$	τ^P	f	$=$	ucast $\circ$ ptr-val $\circ$ f
	τ^P	τ^A	f	$=$	Ptr $\circ$ ucast $\circ$ f
	τ^P	τ^P	f	$=$	Ptr $\circ$ ptr-val $\circ$ f

Table 2.9: Expression type conversions [1, 6.3].

have been treated elsewhere in the literature [91].

Expression

In C, expression statements may contain arbitrary expressions. Expressions may have side effects such as assignment and function invocation. This introduces non-determinism as the order of evaluation is not fully specified. While *com*'s Spec statement provides a natural target for this, the resulting proof obligations after verification condition generation are not as pleasant as those that arise when the language is fully deterministic. This can be achieved with some benign changes to syntax. First, we disallow all side effects in expressions and then provide several limited forms of expression statements to produce side effects deterministically. Any C program can in principle be rewritten to the form described here, with some additional state to hold intermediate results. Any standard C compiler will continue to recognise the expression statements below as they are indistinguishable from cases of the C expression statement.

The translation for expression statements in C_{sys} is given in Table 2.10. Table 2.11 provides the restrictions on the types of expressions in an assignment. Assignment makes a relatively straightforward use of the Basic and Guard primitives, with type conversion performed for compatible assignments. Function calls are more complicated, as they involve several steps and more than one assignment. The call::$(\sigma \Rightarrow \sigma) \Rightarrow \eta \Rightarrow (\sigma \Rightarrow \sigma \Rightarrow \sigma) \Rightarrow (\sigma \Rightarrow \sigma \Rightarrow (\sigma,\eta,\chi)\ com) \Rightarrow (\sigma,\eta,\chi)\ com$ translation is provided by the verification environment and makes use of DynCom, Seq and Call to provide parameter-passing function call semantics. The first argument gives the state update that is performed prior to the function call, in our case this is call-by-value state update. The second argument is the function name. After the function completes, various parts of the program's state space may have been mutated. We wish to discard the effects on local variables and call takes a third argument which provides a means of selecting which parts of the state to preserve — in our case we take the heap and other static variables in the globals component of the program's state space in the post-function

call state t and set the new state to the pre-function call state s with this component updated. Finally, we supply call with a continuation that, in the case of a combined assignment and function call, updates the post-function call state with the return value. The LHS of the assignment may only have the form of a single variable expression to prevent dependence on state that might be updated during the function call in the evaluation of the modifiable lvalue expression. Further details of call are provided by Schirmer [85].

Compound

The translation for compound statements is detailed in Table 2.12, where d_n represents declarations and c_n statements. The main point of interest here is a detail that is lacking from the translation — we do not describe how variable scoping is handled. This is due to additional checks performed by the translation tool that reject programs with variables declared in a block where outside the block there is already a visible variable with the same name. This restriction is not fundamental, with namespace mangling we could ensure all variables have unique names.

Where explicit initialisers are missing, we assign arbitrary as a value, preventing much of interest being shown. A more conservative approach would be to introduce additional initialisation guards for expressions that feature potentially uninitialised variables and/or force the value to depend on program location and state.

Selection

Table 2.13 contains the translation for selection statements. Here we limit ourselves to **if** statements.

Iteration

while and **for** statements in Table 2.14 are implemented using the exception mechanism of the verification environment to allow for abrupt termination of the loop with **continue** and **break** statements. An additional variable is introduced to the program's state space, called global-exn-var, with type:

$$\textbf{datatype}\ \textit{c-exntype} \quad = \quad \mathsf{Break} \mid \mathsf{Continue} \mid \mathsf{Return}$$

and checked when an exception is detected to provide the correct loop semantics. A similar wrapper for the entire function is generated to catch **return** exceptions.

Since the comma operator is not included in ordinary expressions in C_{sys}, **for** statements have this syntax as a special case to list expression statements.

do statements can be handled as above, although are not currently implemented.

$$\langle e_0 := e_1 \rangle = (\lambda s.\ \langle e_0 \rangle_L\ (\mathsf{convert}\ \langle e_0 \rangle_T\ \langle e_1 \rangle_T\ \langle e_1 \rangle_E\ \mathrm{s})\ s)$$

StmtExprAssign

$$\frac{\mathsf{assign\text{-}valid}\ \langle e_0 \rangle_T\ \langle e_1 \rangle_T}{\langle e_0 = e_1; \rangle_S\ = \mathsf{Guard}\ (\langle e_0 \rangle_G \cap \langle e_1 \rangle_G)\ (\mathsf{Basic}\ \langle e_0 := e_1 \rangle)}$$

StmtExprCallAssign

$$\frac{\begin{array}{c}\langle f \rangle_F = \langle r, p_1, \ldots, p_n \rangle \\ \mathsf{assign\text{-}valid}\ \langle e_0 \rangle_T\ \langle r \rangle_T \\ \mathsf{assign\text{-}valid}\ \langle p_1 \rangle_T\ \langle e_1 \rangle_T \quad \ldots \quad \mathsf{assign\text{-}valid}\ \langle p_n \rangle_T\ \langle e_n \rangle_T \\ e_0 \text{ is a variable expression}\end{array}}{\begin{array}{c}\langle e_0 = f(e_1, e_2, \ldots, e_n); \rangle_S\ = \\ \mathsf{Guard}\ (\langle e_0 \rangle_G \cap \langle e_1 \rangle_G \cap \langle e_2 \rangle_G \cap \ldots \cap \langle e_n \rangle_G) \\ (\mathsf{call}\ (\langle p_1 := e_1 \rangle \circ \ldots \circ \langle p_n := e_n \rangle)\ f\ (\lambda s\ t.\ s(\!|\ \mathsf{globals} := \mathsf{globals}\ t\ |\!)) \\ (\lambda s\ t.\ \mathsf{Basic}\ (\langle e_0 \rangle_L\ (\mathsf{convert}\ \langle e_0 \rangle_T\ \langle r \rangle_T\ \langle r \rangle_E\ \mathrm{t}))))\end{array}}$$

StmtExprCall

$$\frac{\begin{array}{c}\langle f \rangle_F = \langle r, p_1, \ldots, p_n \rangle \\ \mathsf{assign\text{-}valid}\ \langle p_1 \rangle_T\ \langle e_1 \rangle_T \quad \ldots \quad \mathsf{assign\text{-}valid}\ \langle p_n \rangle_T\ \langle e_n \rangle_T\end{array}}{\begin{array}{c}\langle f(e_1, e_2, \ldots, e_n); \rangle_S\ = \\ \mathsf{Guard}\ (\langle e_1 \rangle_G \cap \langle e_2 \rangle_G \cap \ldots \cap \langle e_n \rangle_G) \\ (\mathsf{call}\ (\langle p_1 := e_1 \rangle \circ \ldots \circ \langle p_n := e_n \rangle)\ f\ (\lambda s\ t.\ s(\!|\ \mathsf{globals} := \mathsf{globals}\ t\ |\!)) \\ (\lambda s\ t.\ \mathsf{Skip}))\end{array}}$$

StmtExprPostInc

$$\frac{}{\langle e{+}{+}; \rangle_S\ =\ \langle e = e + 1; \rangle_S}$$

StmtExprPostDec

$$\frac{}{\langle e{-}{-}; \rangle_S\ =\ \langle e = e - 1; \rangle_S}$$

StmtExprEmpty

$$\frac{}{\langle ; \rangle_S\ =\ \mathsf{Skip}}$$

Table 2.10: Expression statement translation.

assign-valid	τ_0^A	τ_1^A
	τ	τ
	τ^P	*void ptr*
	void ptr	τ^P

Table 2.11: Valid assignment expression types.

StmtCompound

$$\frac{}{\langle\{d_0\ d_1\ \ldots\ d_n\ c_0\ c_1\ \ldots\ c_m\}\rangle_S\ =\ \langle d_0\rangle_S\ \text{`Seq`}\ \langle d_1\rangle_S\ \text{`Seq`}\ \ldots\ \text{`Seq`}\ \langle d_n\rangle_S\ \text{`Seq`}\ \langle c_0\rangle_S\ \text{`Seq`}\ \langle c_1\rangle_S\ \text{`Seq`}\ \ldots\ \text{`Seq`}\ \langle c_m\rangle_S\ \text{`Seq`}\ \mathsf{Skip}}$$

StmtDecl

$$\frac{}{\langle t\ v;\rangle_S\ =\ \langle v = \mathsf{arbitrary};\rangle_S}$$

StmtDeclAssign

$$\frac{}{\langle t\ v\ =\ e;\rangle_S\ =\ \langle v\ =\ e;\rangle_S}$$

Table 2.12: Compound statement translation.

Jump

Jump statements, in Table 2.15, set the exception cause variable and throw an exception to modify the control flow. In the case of **return** this also involves updating the function return variable in the program's state when the function's return type is not *void*.

$\mathsf{cond\text{-}valid}\ e = \exists \tau.\ \langle e\rangle_T = \tau^A \vee \langle e\rangle_T = \tau^P$

StmtIf

$$\frac{\mathsf{cond\text{-}valid}\ e}{\langle \textbf{if}\ (e)\ c\rangle_S\ = \mathsf{Guard}\ \langle e\rangle_G\ (\mathsf{Cond}\ \{s.\ \langle e\rangle_E\ s \neq 0\}\ \langle c\rangle_S\ \mathsf{Skip})}$$

StmtIfElse

$$\frac{\mathsf{cond\text{-}valid}\ e}{\langle \textbf{if}\ (e)\ c_0\ \textbf{else}\ c_1\rangle_S\ = \mathsf{Guard}\ \langle e\rangle_G\ (\mathsf{Cond}\ \{s.\ \langle e\rangle_E\ s \neq 0\}\ \langle c_0\rangle_S\ \langle c_1\rangle_S)}$$

Table 2.13: Selection statement translation.

$$\textsf{loop-body}\ c\ d = \textsf{Seq}\ (\textsf{Catch}\ c \\ \quad (\textsf{Cond}\ \{s.\ \textsf{global-exn-var}\ s = \textsf{Continue}\}\ \textsf{Skip}\ \textsf{Throw}))\ d$$

$$\textsf{while}\ e\ c = \textsf{Guard}\ \langle e\rangle_G\ (\textsf{While}\ \{s.\ \langle e\rangle_E\ s \neq 0\}\ (\textsf{Seq}\ c\ (\textsf{Guard}\ \langle e\rangle_G\ \textsf{Skip})))$$

$$\textsf{loop}\ e\ c\ d = \textsf{Catch}\ (\textsf{while}\ e\ (\textsf{loop-body}\ \langle c\rangle_S\ \langle d\rangle_S)) \\ \quad (\textsf{Cond}\ \{s.\ \textsf{global-exn-var}\ s = \textsf{Break}\}\ \textsf{Skip}\ \textsf{Throw})$$

StmtWhile

$$\frac{\textsf{cond-valid}\ e}{\langle \textbf{while}\ (e)\ c\rangle_S\ =\ \textsf{loop}\ e\ c\ \textsf{Skip}}$$

StmtFor

$$\frac{\textsf{cond-valid}\ e}{\begin{array}{c}\langle \textbf{for}\ (c_0, c_1, \ldots, c_n;\ e;\ d_0, d_1, \ldots, d_m)\ c\rangle_S\ = \\ \langle c_0;\rangle_S\ \text{`}\textsf{Seq}\text{`}\ \langle c_1;\rangle_S\ \text{`}\textsf{Seq}\text{`}\ \ldots\ \text{`}\textsf{Seq}\text{`}\ \langle c_n;\rangle_S\ \text{`}\textsf{Seq}\text{`}\ \textsf{Skip}\ \text{`}\textsf{Seq}\text{`} \\ \textsf{loop}\ e\ c\ (\langle d_0;\rangle_S\ \text{`}\textsf{Seq}\text{`}\ \langle d_1;\rangle_S\ \text{`}\textsf{Seq}\text{`}\ \ldots\ \text{`}\textsf{Seq}\text{`}\ \langle d_m;\rangle_S\ \text{`}\textsf{Seq}\text{`}\ \textsf{Skip})\end{array}}$$

Table 2.14: Iteration statement translation.

2.5.5 Guards

The generation of guards that perform run-time checks is given in Table 2.16. Guards are a semantic construction used during verification and do not result in any modification to the code or compilation.

Only a few examples of guards are included to provide a flavour. Our framework allows a verifier to customise the guards to a particular verification scenario. For example, a verification that is aimed at a strict interpretation of the standard may require guards on pointer and array arithmetic to ensure that bounds are not violated, checks on the correct initialisation of variables, for trap representations, etc. In another scenario, we may wish to remove the NULL pointer guard to allow the operating system to interact with the base of memory. An important use of guards could be to limit the heap to a subset of the address space to prevent unintentional damage to the run-time stack, page tables and code segment.

The other aspects of translation remain mostly unchanged when we modify the guard translation, lowering the cost of specialising to a particular implementation, although in some cases it may be necessary to track some additional state, e.g. the initialisation state of variables [74].

Definition 2.5.2. Pointer expressions have guards on alignment and NULL values:

throw r = Seq (Basic ($\lambda s.\ s(\!|$ global-exn-var := r $|\!)$)) Throw

StmtContinue

$$\overline{\langle \textbf{continue};\rangle_S \ = \ \text{throw Continue}}$$

StmtBreak

$$\overline{\langle \textbf{break};\rangle_S \ = \ \text{throw Break}}$$

StmtReturnVoid

$$\overline{\langle \textbf{return};\rangle_S \ = \ \text{throw Return}}$$

StmtReturn

$$\frac{\langle f\rangle_F = \langle r,p_1,\ldots,p_n\rangle \quad \text{statement enclosed by function } f}{\langle \textbf{return } e;\rangle_S \ = \ \text{Seq } (\langle r = e;\rangle_S) \ (\text{throw Return})}$$

Table 2.15: Jump statement translation.

c-null-guard p	$\equiv$	$0 \notin \{p_{\&}..+\text{size-of TYPE}(\alpha)\}$
ptr-aligned p	$\equiv$	align-of TYPE(α) dvd $\mathbb{N}^{\Leftarrow}$ $p_{\&}$
c-guard (p::α ptr)	$\equiv$	ptr-aligned $p \wedge$ c-null-guard p

The pointer guard definitions are polymorphic and only need to be specified once for all pointer types.

2.5.6 Side-effect free expressions

The shallow embedding of side-effect free expressions is given in Table 2.17 and Table 2.18. Pointer addition is restricted to where the integer sub-expression is an *uint*, and pointer subtraction completely elided. This already relies on implementation-defined behaviour, namely the equivalence of addresses and *uint*, and could be generalised to other integer types with more such detail. Pointer relational and equality operators only allow same typed sub-expressions to be compared, rather than also allowing the *void ptr* in comparisons. Again, these limitations are not fundamental.

The embedding is split between two tables as it is quite lengthy, with the second table containing the embedding of expressions that interact with the program's state.

Definition 2.5.3. Heap dereferences in expressions, e.g. $*$p + 1 are given a semantics by first lifting the raw heap state with a polymorphic lift function, e.g. lift $s\ p\ +\ 1$ where s is the current state.

heap-list :: *heap-mem* $\Rightarrow$ *nat* $\Rightarrow$ *addr* $\Rightarrow$ *byte list*		
heap-list h *0* p	$\equiv$	[]
heap-list h (*Suc* n) p	$\equiv$	$h\ p$·heap-list $h\ n\ (p\ +\ 1)$

GuardCond
$\langle e_0 \; ? \; e_1 : e_2 \rangle_G \;=\; \langle e_0 \rangle_G \cap \langle e_1 \rangle_G \cap \langle e_2 \rangle_G$

GuardArith
$\langle e_0 \; f \; e_1 \rangle_G \;=\; \langle e_0 \rangle_G \cap \langle e_1 \rangle_G$ where $f \in \{+, -, *, \&, |, \hat{}\}$

GuardDivMod
$\langle e_0 \; f \; e_1 \rangle_G \;=\; \langle e_0 \rangle_G \cap \langle e_1 \rangle_G \cap \{s.\ \langle e_1 \rangle_E \; s \neq 0\}$ where $f \in \{/, \%\}$

GuardLogicRelEq
$\langle e_0 \; f \; e_1 \rangle_G \;=\; \langle e_0 \rangle_G \cap \langle e_1 \rangle_G$ where $f \in \{||, \&\&, <, <=, >, >=, ==, !=\}$

GuardShift
$\langle e_0 \; f \; e_1 \rangle_G \;=\; \langle e_0 \rangle_G \cap \langle e_1 \rangle_G \cap \{s.\ \langle e_1 \rangle_E \; s \geq 0\}$ where $f \in \{<<, >>\}$

GuardCast
$\langle (t)e \rangle_G \;=\; \langle e \rangle_G$

GuardDeref
$\langle *e \rangle_G \;=\; \langle e \rangle_G \cap \{s.\ \textsf{c-guard}\ (\langle e \rangle_E \; s)\}$

GuardUnary
$\langle f \; e \rangle_G \;=\; \langle e \rangle_G$ where $f \in \{+, -, \tilde{}, !, \&, \textbf{sizeof}\}$

GuardDot
$\langle e.n \rangle_G \;=\; \langle e \rangle_G$

GuardArrow
$\langle e{->}n \rangle_G \;=\; \langle *e \rangle_G$

GuardArray
$$\frac{\langle e_0 \rangle_T = (\tau, n)\ \mathit{array}}{\langle e_0[e_1] \rangle_G \;=\; \langle e_0 \rangle_G \cap \langle e_1 \rangle_G}$$

GuardArrayDegen
$$\frac{\langle e_0 \rangle_T = \tau\ \mathit{ptr}}{\langle e_0[e_1] \rangle_G \;=\; \langle *(e_0 + e_1) \rangle_G}$$

GuardConst
$\langle c \rangle_G \;=\; \mathcal{U}$

GuardVar
$\langle v \rangle_G \;=\; \mathcal{U}$

GuardParen
$\langle (e) \rangle_G \;=\; \langle e \rangle_G$

Table 2.16: Guard translations.

h-val :: *heap-mem* ⇒ α::*c-type ptr* ⇀ α
h-val *h p* ≡ from-bytes (heap-list *h* (size-of TYPE(α)) $p_{\&}$)

lift :: *heap-mem* ⇒ α::*c-type ptr* ⇒ α
lift *h* ≡ λ*p*. the (h-val *h p*)

This is a core concept in the expression semantics, as it provides the formal machinery for splitting values across multiple locations in the heap. lift and h-val are polymorphic, with their types inferred from context, e.g. from an applied pointer. A *byte list* of the type's size is retrieved from memory with heap-list and lifted with from-bytes to the HOL level. It should be noted that this is a semantic model and updates to the heap state affect all lifted heaps, regardless of type.

The **struct** field . operator has two cases. If the first expression does not have an address in the heap then it must be an automatic variable and we retrieve the value from the program state by applying the **record** access function. Otherwise, we use the &($p \rightarrow n$) operator, defined in Chapter 5, to obtain the address of the field. This is a HOL term that takes an α::*c-type ptr p* and field name *n* and gives the address of the field in the heap (see Defn. 5.2.8). A pointer of the field type can then be formed and the value lifted from the heap.

2.5.7 Lvalues

The address of an lvalue's object can be found with Table 2.19. Modifiable lvalue expressions can be embedded as state update functions with Table 2.20.

Definition 2.5.4. Updates to lvalues in the heap are given a semantics with heap-update:

heap-update-list :: *addr* ⇒ *byte list* ⇒ *heap-mem* ⇒ *heap-mem*
heap-update-list *p* [] *h* ≡ *h*
heap-update-list *p* (*x*·*xs*) *h* ≡ heap-update-list (*p* + *1*) *xs* (*h*(*p* := *x*))

heap-update :: α::*c-type ptr* ⇒ α ⇒ *heap-mem* ⇒ *heap-mem*
heap-update *p v h* ≡ heap-update-list $p_{\&}$ (to-bytes *v*) *h*

For example, the assignment ∗p = ∗q + 5 is translated to the state transformer λ*s*. heap-update *p* (lift *s q* + *5*) *s*. As with lift, this is a place where the semantics maps between the HOL value and *byte list* representation. We develop rules to abstract these terms when they appear in proof obligations in the following chapters.

ExprCond

$$\frac{\begin{array}{c}\mathsf{cond\text{-}valid}\ e_0 \\ \mathsf{convert}\ \langle e_0\ ?\ e_1\ :\ e_2\rangle_T\ \langle e_1\rangle_T\ \langle e_1\rangle_E = e_1^R \\ \mathsf{convert}\ \langle e_0\ ?\ e_1\ :\ e_2\rangle_T\ \langle e_2\rangle_T\ \langle e_2\rangle_E = e_2^R\end{array}}{\langle e_0\ ?\ e_1\ :\ e_2\rangle_E\ =\ \lambda s.\ \mathit{if}\ \langle e_0\rangle_E\ s \neq 0\ \mathit{then}\ {e_1}^R\ s\ \mathit{else}\ {e_2}^R\ s}$$

ExprPlus

$$\frac{\langle e_0\rangle_T = \tau^P \qquad \langle e_1\rangle_T = \mathit{uint}}{\langle e_0 +\ e_1\rangle_E\ =\ \lambda s.\ (\langle e_0\rangle_E\ s)\ +_p\ (\langle e_1\rangle_E\ s)}$$

ExprPlus2

$$\frac{\langle e_0\rangle_T = \mathit{uint} \qquad \langle e_1\rangle_T = \tau^P}{\langle e_0 +\ e_1\rangle_E\ =\ \lambda s.\ (\langle e_1\rangle_E\ s)\ +_p\ (\langle e_0\rangle_E\ s)}$$

ExprArith

$$\frac{\begin{array}{c}\langle e_0\rangle_T = \tau_0^A \qquad \langle e_1\rangle_T = \tau_1^A \qquad \mathsf{convert}\ \langle e_0\ f\ e_1\rangle_T\ \tau_0^A\ \langle e_0\rangle_E = e_0^R \\ \mathsf{convert}\ \langle e_0\ f\ e_1\rangle_T\ \tau_1^A\ \langle e_1\rangle_E = e_1^R \qquad f \in \{+, -, *, /, \%, \&, |, \hat{}\} \\ g \text{ is a HOL bit-vector operator corresponding to } f\end{array}}{\langle e_0\ f\ e_1\rangle_E\ =\ \lambda s.\ g\ (e_0^R\ s)\ (e_1^R\ s)}$$

ExprLogic

$$\frac{\begin{array}{c}\mathsf{cond\text{-}valid}\ e_0 \qquad \mathsf{cond\text{-}valid}\ e_1 \\ f \in \{||, \&\&\} \qquad g \text{ is a HOL boolean operator corresponding to } f\end{array}}{\langle e_0\ f\ e_1\rangle_E\ =\ \lambda s.\ \mathit{if}\ g\ (\langle e_0\rangle_E\ s \neq 0)\ (\langle e_1\rangle_E\ s \neq 0)\ \mathit{then}\ 1\ \mathit{else}\ 0}$$

ExprRelEq

$$\frac{\begin{array}{c}\langle e_0\rangle_T = \tau_0^A \qquad \langle e_1\rangle_T = \tau_1^A \qquad \mathsf{convert}\ \langle e_0 + e_1\rangle_T\ \tau_0^A\ \langle e_0\rangle_E = e_0^R \\ \mathsf{convert}\ \langle e_0 + e_1\rangle_T\ \tau_1^A\ \langle e_1\rangle_E = e_1^R \qquad f \in \{<, <=, >, >=, ==, !=\} \\ g \text{ is a HOL relational operator corresponding to } f\end{array}}{\langle e_0\ f\ e_1\rangle_E\ =\ \lambda s.\ \mathit{if}\ g\ (e_0^R\ s)\ (e_1^R\ s)\ \mathit{then}\ 1\ \mathit{else}\ 0}$$

ExprRelEqPtr

$$\frac{\begin{array}{c}\langle e_0\rangle_T = \tau^P \qquad \langle e_1\rangle_T = \tau^P \qquad f \in \{<, <=, >, >=, ==, !=\} \\ g \text{ is a HOL relational operator corresponding to } f\end{array}}{\langle e_0\ f\ e_1\rangle_E\ =\ \lambda s.\ \mathit{if}\ g\ (\mathsf{ptr\text{-}val}\ e_0^R\ s)\ (\mathsf{ptr\text{-}val}\ e_1^R\ s)\ \mathit{then}\ 1\ \mathit{else}\ 0}$$

ExprShift

$$\frac{\begin{array}{c}\langle e_0\rangle_T = \tau_0^A \qquad \langle e_1\rangle_T = \tau_1^A \qquad \mathsf{convert}\ \langle e_0\ f\ e_1\rangle_T\ \tau_0^A\ \langle e_0\rangle_E = e_0^R \\ f \in \{<<, >>\} \qquad g \text{ is a HOL bit-vector operator corresponding to } f\end{array}}{\langle e_0\ f\ e_1\rangle_E\ =\ \lambda s.\ g\ (e_0^R\ s)\ (\mathbb{N}^{\Leftarrow}\ (\langle e_1\rangle_E\ s))}$$

ExprUnary

$$\frac{\begin{array}{c}\langle e\rangle_T = \tau^A \qquad \mathsf{convert}\ \langle f\ e\rangle_T\ \tau^A\ \langle e\rangle_E = e^R \\ f \in \{+,-,\tilde{}\} \qquad g \text{ is a HOL bit-vector operator corresponding to } f\end{array}}{\langle f\ e\rangle_E\ =\ \lambda s.\ g\ (e^R\ s)}$$

ExprNot

$$\frac{\mathsf{cond\text{-}valid}\ e}{\langle !e\rangle_E\ =\ \lambda s.\ \mathit{if}\langle e\rangle_E\ s = 0\ \mathit{then}\ 1\ \mathit{else}\ 0}$$

Table 2.17: Side-effect free expression translations.

ExprAddrOf
$\langle \& e \rangle_E \;=\; \lambda s.\ \mathsf{Ptr}\ (\langle e \rangle_A\ s)$

ExprDeref
$\langle * e \rangle_E \;=\; \lambda s.\ \mathsf{lift}\ (\mathsf{h\text{-}state}\ s)\ (\langle e \rangle_E\ s)$

ExprSizeOf
$\langle \mathsf{sizeof}\ e \rangle_E \;=\; \lambda s.\ \mathsf{size\text{-}of\ TYPE}(\langle e \rangle_T)$

ExprCast
$\langle (t) e \rangle_E \;=\; \mathsf{convert}\ \langle (t) e \rangle_T\ \langle e \rangle_T\ \langle e \rangle_E$

ExprDot
$$\frac{\langle e \rangle_A = \bot}{\langle e\,.\,n \rangle_E \;=\; \lambda s.\ n\ (\langle e \rangle_E\ s)}$$

ExprArrow
$\langle e{-}{>}n \rangle_E \;=\; \langle (*e)\,.\,n \rangle_E$

ExprDotLvalue
$$\frac{\langle e \rangle_A \neq \bot}{\langle e\,.\,n \rangle_E \;=\; \lambda s.\ \mathsf{lift}\ (\mathsf{h\text{-}state}\ s)\ (\mathsf{Ptr}\ (\&(\langle \& e \rangle_E\ s) {\rightarrow} n))}$$

ExprArrayPtr
$$\frac{\langle e_0 \rangle_T = \tau\ ptr}{\langle e_0 [e_1] \rangle_E \;=\; \langle *(e_0 +\ e_1) \rangle_E}$$

ExprArray
$$\frac{\langle e_0 \rangle_T = (\tau, n)\ array}{\langle e_0 [e_1] \rangle_E \;=\; \lambda s.\ \mathsf{array\text{-}index}\ (\langle e_0 \rangle_E\ s)\ (\mathbb{N}^{\Leftarrow}\ (\langle e_1 \rangle_E\ s))}$$

ExprConst
$\langle c \rangle_E \;=\; \lambda s.\ \mathbb{N}^{\Rightarrow}\ c$

ExprParen
$\langle (e) \rangle_E \;=\; \langle e \rangle_E$

ExprVarAuto
$$\frac{v \text{ is automatic}}{\langle v \rangle_E \;=\; v\text{-}'}$$

ExprVarStatic
$$\frac{v \text{ is static}}{\langle v \rangle_E \;=\; \lambda s.\ \mathsf{lift}\ (\mathsf{h\text{-}state}\ s)\ v\text{-}addr}$$

Table 2.18: Side-effect free expression translations (cont.).

2.5.8 Example translation

Example 2.5.1. Table 2.21 contains a translation of **tree_min**'s function body in Exmp. 2.3.3. This appears to be a quite large and unwieldy object for such a simple program, but this is processed by the VCG before the program verifier interacts with it. The resulting proof obligations, as seen in later examples, are reasonable since the VCG is able to simplify significantly — e.g. the Catchs are never used other than in a trivial way and disappear.

Acknowledgments

The content in this chapter represents joint work with Michael Norrish and Gerwin Klein. The aspects of the translation process not related to the

AddrDeref
$\langle *e \rangle_A \;=\; \lambda s.\ \textsf{ptr-val}\ (\langle e \rangle_E\ s)$

AddrDot
$\langle e\,.\,n \rangle_A \;=\; \lambda s.\ \&(\langle \&e \rangle_E\ s) {\rightarrow} n$

AddrArrow
$\langle e {->} n \rangle_A \;=\; \langle (*e)\,.\,n \rangle_A$

AddrArray
$$\frac{\langle e_0 \rangle_T = \tau\ ptr}{\langle e_0 [e_1] \rangle_A \;=\; \langle *(e_0 +\ e_1) \rangle_A}$$

AddrVarStatic
$$\frac{v \text{ is static}}{\langle v \rangle_A \;=\; \lambda s.\ \textsf{ptr-val}\ v\text{-}addr}$$

Table 2.19: Lvalue address translations.

type encoding, **struct** operators, pointer and heap semantics have been formalised after the fact, using a combination of the C standard and an ML implementation of the translation process developed by Michael Norrish.

MLvalDeref
$\langle *e \rangle_L \;=\; \lambda v\ s.\ \mathsf{heap\text{-}update}\ (\langle e \rangle_E\ s)\ v\ s$

MLvalDot
$$\frac{\langle e \rangle_A = \bot}{\langle e\,.\,n \rangle_L \;=\; \lambda v\ s.\ \langle e \rangle_L\ ((\langle e \rangle_E\ s)(\!|\ n := v\ |\!))\ s}$$

MLvalDotHeap
$$\frac{\langle e \rangle_A \neq \bot}{\langle e\,.\,n \rangle_L \;=\; \lambda v\ s.\ \mathsf{heap\text{-}update}\ (\mathsf{Ptr}\ (\&(\langle \&e \rangle_E\ s) \rightarrow n))\ v\ s}$$

MLvalArrow
$\langle e{-}{>}n \rangle_L \;=\; \langle (*e)\,.\,n \rangle_L$

MLvalArrayPtr
$$\frac{\langle e_0 \rangle_T = \tau\ ptr}{\langle e_0[e_1] \rangle_L \;=\; \langle *(e_0 +\ e_1) \rangle_L}$$

MLvalArray
$$\frac{\langle e_0 \rangle_T = (\tau,n)\ array}{\langle e_0[e_1] \rangle_L \;=\; \lambda v\ s.\ \langle e_0 \rangle_L\ (\mathsf{array\text{-}update}\ (\langle e_0 \rangle_E\ s)\ (\mathbb{N}^{\Leftarrow}\ (\langle e_1 \rangle_E\ s))\ v)\ s}$$

MLvalVarAuto
$$\frac{v \text{ is automatic}}{\langle v \rangle_L \;=\; \lambda k\ s.\ s(\!|\ v\text{-}' := k\ |\!)}$$

MLvalVarStatic
$$\frac{v \text{ is static}}{\langle v \rangle_L \;=\; \mathsf{heap\text{-}update}\ v\text{-}addr}$$

Table 2.20: Modifiable lvalue translations.

```
Catch (
   Catch (
      Guard {s. c-guard (t-' s)} (
         While {s. lift (h-state s) (Ptr &(t-' s→[''l'']) ≠ 0} (
            Catch (
               Guard {s. c-guard (t-' s)} (
                  Basic (λs. s(| t-' := lift (h-state s) (Ptr &(t-' s→[''l'']))|))
               )
            ) (
               Cond {s. global-exn-var s = Continue} Skip Throw
            ) `Seq`
            Skip `Seq`
            Guard {s. c-guard (t-' s)} Skip
         )
      )
   ) (
      Cond {s. global-exn-var s = Break} Skip Throw
   ) `Seq`
   Guard {s. c-guard (t-' s)} (
      Basic (λs. s(| tree-min-ret-' := lift (h-state s)
                                            (Ptr &(t-' s→[''item'']))|))
   ) `Seq`
   Basic (λs. s(| global-exn-var := Return |)) `Seq`
   Throw
) Skip
```

Table 2.21: **tree_min** *com* translation.

Chapter 3

Unified memory model

3.1 Inter-type aliasing

The pointer dereferencing semantics presented in Chapter 2 are sufficient to describe the behaviour of a C program, but they do not provide an adequate basis for writing assertions or describing proofs for the obligations resulting from verification condition generation. Aliasing, under the ROLM assumption, in C is apparent in several possibilities:

- Conventional aliasing between pointers of the same type, as introduced in §1.3.

  Example 3.1.1. For two **char** pointers the following Hoare triples hold:

  $\{\!|\sigma.\ ´p \neq ´q\ |\!\}\ *p = 1; *q = 2; \{\!|\ *(^{\sigma}p) = 1\ |\!\}$
  $\{\!|\sigma.\ ´p = ´q\ |\!\}\ *p = 1; *q = 2; \{\!|\ *(^{\sigma}p) = 2\ |\!\}$

  We refer to this as *intra-type* aliasing and its treatment in conjunction with the aliasing issues identified below is the subject of Chapter 4.

- Pointers of different types may point to the same object. This may be intentional, as in physical sub-typing [88], or unwanted, where aliasing should not occur if the program is correct.

  Example 3.1.2. Consider the following declarations:

  struct bin_tree { **int** item; **struct** bin_tree *left , *right ; };
  struct llist { **int** item; **struct** llist *next; };

  If linked data structures of these types are intended to make disjoint use of memory then the following triple regarding ordered insertion into a list should be true:

  $\{\!|\sigma.$ balanced-tree $´p\ |\!\}$
  insert-ordered$((p{::}$*bin-tree ptr*$)\rightarrow$item, $(q{::}$*llist ptr*$))$;
  $\{\!|$ balanced-tree $^{\sigma}p\ |\!\}$

Unfortunately, additional information is required to rule out aliasing in the pre-condition. Stating $p_{\&} \neq q_{\&}$ is insufficient, as it is possible that the node modified during insertion in the list rooted at q is also linked as a node in the tree rooted at p. We require a stronger conjunct, as with intra-type aliasing, giving disjointness between the addresses of pointers occurring in the tree and those in the list.

If C were type-safe then predicates ruling out potential aliasing between differently typed pointers would be unnecessary[1] and it is troubling that the program verifier is burdened with the requirement to add these predicates to assertions and reason about them.

- Pointers do not alias only on an object's base address, but may alias inside the encoding. This again may be a deliberate feature in a program, such as using an **unsigned int *** to efficiently read consecutive 32-bit blocks in a **struct** when computing a checksum of the same width, or performing endianess conversion on a received network packet. However, this further complicates the problem of having to state aliasing conditions.

  Alignment provides some relief, for example two **int**s with 4 byte size and alignment may not alias into the encodings of one another under ROLM. In general, however, the C standard and implementations we consider only guarantee that $\mathsf{align\text{-}of}\ \mathsf{TYPE}(\alpha)\ \mathsf{dvd}\ \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)$, a condition required to allow tiling of objects in arrays. Hence it may still be possible for a **struct bin_tree *** pointer to reference inside a **llist** object representation. The addresses that need to be considered in an anti-aliasing predicate must include the entire heap footprint of each object.

Collectively we refer to the last two aliasing possibilities as *inter-type* aliasing.

While ROLM allows programs the ability to exhibit inter-type aliasing, it should be the case that well-written systems code has very little of this form of aliasing, confining it to only the parts where this is necessary, such as the motivating examples in §2.3.1. By remaining inside the type-safe fragment of the language it is possible to leverage what type checks the compiler and other pointer analysis tools offer, in addition to simplifying the proofs we consider here.

Based on this observation, we would like to ignore inter-type aliasing in most proofs. The multiple typed heaps model from §1.3 provides an abstract state representation that allows for this, but does not apply under either ROLM or many other reasonable implementation specialisations of the strict

[1] If we ignore updates through member objects in **struct**s, an assumption that is revisited in Chapter 5.

C standard. In the following, we unify the semantic model in §2.3.2 with this abstract view of memory, providing the best of both worlds — the ability to express the semantics of programs that exhibit inter-type aliasing where needed and multiple typed heaps as a proof abstraction where the program remains within a type-safe fragment of C.

3.2 Heap type description

3.2.1 Ghost variable

Definition 3.2.1. The notion of object lifetime and type can be recovered by introducing an additional *ghost* component in the program state, which we call the *heap type description*:

$$\textbf{types}\ \textit{heap-typ-desc} = \textit{addr} \rightharpoonup \textit{typ-tag option}$$

This captures the implicit mapping from addresses to types and object footprints. The heap type description is partial, since it only maps memory actually used by the program. Each object representation has the *typ-tag* corresponding to its type stored at the base address. The rest of the *heap footprint* of the value is also mapped, but with a $\lfloor\bot\rfloor$ value padding instead of a tag. It is helpful to consider the three possible cases for entries at an address — $\bot$ meaning no value is present, $\lfloor\lfloor t\rfloor\rfloor$ that the location represents the base of a footprint for some object of type t and $\lfloor\bot\rfloor$ giving some location inside the footprint other than the base.

To understand why this is useful, consider a heap type description *addr* $\rightharpoonup$ *typ-tag*, where only the base of the object footprint is stored. In principle, this could also describe the intended mapping if we add a well-formedness invariant [93]:

$$\begin{array}{l}\mathsf{wf\text{-}heap}\ d \equiv \\ \forall x\ y\ t.\ d\ x = \lfloor t\rfloor \wedge 0 < y \wedge y < \mathsf{typ\text{-}size}\ t \longrightarrow d\ (\mathbb{N}^{\Rightarrow}\ (\mathbb{N}^{\Leftarrow}\ x + y)) = \bot\end{array}$$

This requires that wf-heap is carried around in proofs and assertions and that it is reestablished on updates to the heap type description resulting from retype operations. Further problematic is that pointer validity is non-monotonic, i.e. if $d \subseteq_m d'$ then if a pointer is valid in d it should be valid in d'. This would complicate the later separation logic development of Chapter 4, where performing map addition $d +\!\!+ d'$, with wf-heap d, wf-heap d' and dom $d \cap$ dom $d' = \emptyset$, does not preserve pointer validity or even the wf-heap invariant.

The heap type description is a ghost variable. It exists as a proof convenience and plays no role in the semantic interpretation — programs

are free to violate this mapping and do anything that ROLM permits. The proof following verification condition generation will not benefit from the information contained in the heap type description in this case, but remains sound.

3.2.2 Validity

Definition 3.2.2. We write $d,g \models_t p$ to mean that the pointer p::α::*c-type ptr* is valid in heap type description d with guard g:

$$\begin{aligned}&\mathsf{valid\text{-}footprint}\ d\ x\ t\ n \equiv d\ x = \lfloor\lfloor t\rfloor\rfloor \wedge \\ &\qquad\qquad (\forall y.\ y \in \{x + 1..{+}n - 1\} \longrightarrow d\ y = \lfloor\bot\rfloor)\end{aligned}$$

$$d,g \models_t p \equiv \mathsf{valid\text{-}footprint}\ d\ p_{\&}\ \mathsf{TYPE}(\alpha)_t\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)) \wedge g\ p$$

The guard g strengthens the assertion to restrict validity based on the language's pointer dereferencing rules. For example, alignment can be captured with d,ptr-aligned $\models_t p$. The stronger assertion is motivated by the need to satisfy the guard proof obligation generated whenever a pointer is dereferenced — if it is necessary to establish validity of a pointer p for the purpose of a proof about a code fragment involving p, it is usual that one or more guard related proof obligations for p will also need to be discharged. By parameterising the guard, the framework presented in this chapter becomes reusable in settings with different guard restrictions.

The type signature of valid-footprint is *heap-typ-desc* $\Rightarrow$ *addr* $\Rightarrow$ *typ-tag* $\Rightarrow$ *nat* $\Rightarrow$ *bool* and does not contain any type variables, unlike the polymorphic $\models_t$. By removing the dependency on the type variable we are able to write Isabelle/HOL definitions involving validity where there is no free type variable on the RHS of the definition unmatched on the LHS, such as $\doteq$ in §3.5, as one cannot quantify over types in Isabelle/HOL. This idiom of splitting a definition into a polymorphic wrapper for a monomorphic definition is used later in the definitions of Chapter 5 as well.

Example 3.2.1. A heap memory and type description state is given in Fig. 3.1. Locations in the footprint of a valid pointer are shaded. If the heap type description is the variable d, then d,ptr-aligned $\models_t s$ and d,ptr-aligned $\models_t u$, but $\neg\ d$,ptr-aligned $\models_t t$ and $\neg\ d$,ptr-aligned $\models_t v$. The pointer t::*char ptr* is not valid because the correct heap footprint is missing from the heap type description, and v::*int ptr* is invalid because it is unaligned.

Theorem 3.2.1. *Pointer validity is monotonic:*

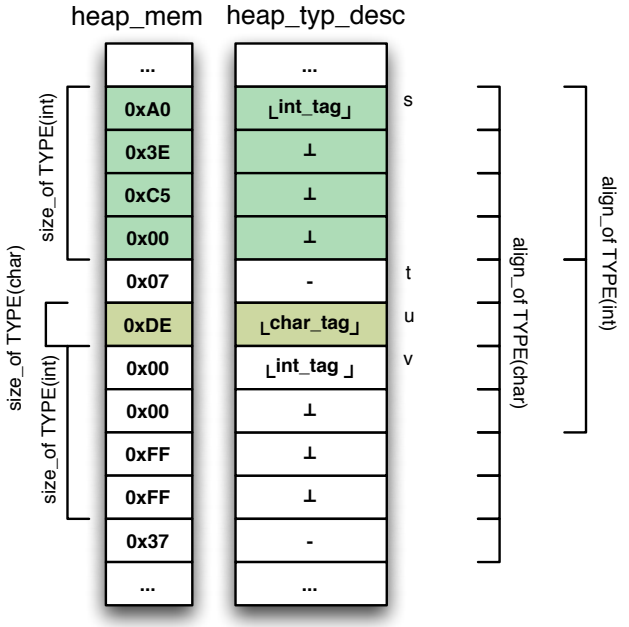

Figure 3.1: Example heap state.

$$\frac{d \subseteq_m d'}{d,g \models_t p \longrightarrow d',g \models_t p}$$

Proof. By unfolding Defn. 3.2.2. □

Theorem 3.2.2. *Two valid pointers, p::α::c-type ptr and q::β::c-type ptr, with unequal address values do not have overlapping footprints:*

$$\frac{d,g \models_t p \qquad d,g' \models_t q \qquad p_\& \neq q_\&}{\{p_\&..+\text{size-of TYPE}(\alpha)\} \cap \{q_\&..+\text{size-of TYPE}(\beta)\} = \emptyset}$$

Proof. Observe that $p_\& \notin \{q_\&..+\text{size-of TYPE}(\beta)\}$, since if it were it would have to be a member of the interval $\{q_\& + 1..+\text{size-of TYPE}(\beta) - 1\}$ as $p_\& \neq q_\&$. From $d,g' \models_t q$ and the definition of validity we then have that $d\ p_\& = \lfloor\bot\rfloor$. However, from $d,g \models_t p$ and the definition of validity $d\ p_\& = \lfloor\lfloor\text{TYPE}(\alpha)_t\rfloor\rfloor$.

Similarly, $q_\& \notin \{p_\&..+\text{size-of TYPE}(\alpha)\}$. If there is overlap then either $p_\& \in \{q_\&..+\text{size-of TYPE}(\beta)\}$ or $q_\& \in \{p_\&..+\text{size-of TYPE}(\alpha)\}$, so a contradiction is obtained.

□

Definition 3.2.3. A pointer p::α::*c-type ptr* is *safe* w.r.t. a heap type description d if its footprint is a subset of the domain of d:

$$\text{ptr-safe } p\ d \equiv \{p_{\&}..+\text{size-of TYPE}(\alpha)\} \subseteq \text{dom } d$$

Theorem 3.2.3. *Pointer validity implies safety:*

$$\frac{d,g \models_t p}{\text{ptr-safe } p\ d}$$

Proof. By unfolding Defn. 3.2.3 and Defn. 3.2.2. □

3.2.3 Retyping

In this section we describe how the heap type description can be updated by the ptr-tag function, a process also referred to as *retyping*.

Definition 3.2.4. ptr-tag updates the heap type description such that it includes the given pointer p's footprint:

$$\begin{aligned}
&\text{ptr-clear } p\ n\ d && \equiv\ d(\{p_{\&} + 1..+n - 1\}\{\mapsto\}\bot)\\
&\text{ptr-set } p\ t\ d && \equiv\ d(p_{\&} \mapsto \lfloor t_t \rfloor)\\
&\text{ptr-tag } (p::\alpha\ ptr) && \equiv\ \text{ptr-set } p\ \text{TYPE}(\alpha) \circ \text{ptr-clear } p\ (\text{size-of TYPE}(\alpha))
\end{aligned}$$

It is intended that retyping occurs once for all variables of static storage duration. If automatic variables were included in the heap area they would also require retyping on block entry/exit. During a program's execution, retyping occurs at the call sites of and inside memory (de)allocator functions, where the intended type and use of a block of memory changes.

Theorem 3.2.4. *Following retyping, a target pointer $p::\alpha::$mem-type ptr is valid:*

$$\frac{g\ p}{\text{ptr-tag } p\ d,g \models_t p}$$

Proof. Unfold definitions and note that size-of $\text{TYPE}(\alpha) < |\text{addr}|$, hence the required footprint can be represented in the heap type description. □

Theorem 3.2.5. *A previously valid pointer $q::\beta::$mem-type ptr remains valid across a retype as long as its footprint and $p::\alpha::$mem-type ptr's are disjoint:*

$$\frac{d,g \models_t q \qquad \{p_{\&}..+\text{size-of TYPE}(\alpha)\} \cap \{q_{\&}..+\text{size-of TYPE}(\beta)\} = \emptyset}{\text{ptr-tag } p\ d,g \models_t q}$$

Proof. We have $x \notin \{p_{\&}..+\text{size-of TYPE}(\alpha)\} \implies \text{ptr-tag } p\ d\ x = d\ x$ by noting $0 <$ size-of $\text{TYPE}(\alpha)$ prevents the disjointness assumption from not accounting for the effect of ptr-set with empty type object representations. The rule then follows by unfolding and application of this fact. □

One of the motivations in developing the heap type description and a notion of validity was the desire to avoid having to reason about footprint intervals. The above rule requires consideration of the entire footprint however, though the need to invoke this rule occurs usually only at memory (de)allocation time. Based on the above rule we can derive additional rules that do not involve interval reasoning in their use:

- The specifications for allocators can explicitly provide for validity preservation.

  Example 3.2.2. Consider an allocator for **struct llist** nodes. Internally it may use types distinct to the rest of the program and perform a retype immediately before a successful return. A property such as the following could then be provided:

  $\{\!|\sigma.\ \acute{}d,g \models_t \acute{}(p::\ \alpha\ ptr) \wedge \mathsf{typ\text{-}tag}\ \mathsf{TYPE}(\alpha) \notin \mathit{ll\text{-}alloc\text{-}typ\text{-}tags}\ |\!\}$
  $q = \mathit{ll\text{-}alloc}();$
  $\{\!|\ \acute{}d,g \models_t {}^{\sigma}p\ |\!\}$

  Here, any pointer p has its validity preserved during the execution of **ll_alloc** providing it is not of a type modified by the allocator's execution, i.e. not in the set of types used to implement the free list.

- Application-specific rules based on the types utilised and the relationship between retype targets and existing valid pointers can be added.

  Example 3.2.3. If a retype is performed on the address of a valid pointer $r::\gamma\ ptr$ and the size of the object representation of the target type β is less than or equal to the size of γ's representation, then a distinct pointer $p::\alpha\ ptr$ has its validity preserved:

$$\frac{d,g' \models_t r \qquad q_\& = r_\& \qquad \begin{array}{c} d,g \models_t p \\ \mathsf{size\text{-}of}\ \mathsf{TYPE}(\beta) \leq \mathsf{size\text{-}of}\ \mathsf{TYPE}(\gamma) \\ \mathsf{TYPE}(\alpha)_t \neq \mathsf{TYPE}(\gamma)_t \vee p_\& \neq q_\& \end{array}}{\mathsf{ptr\text{-}tag}\ q\ d,g \models_t p}$$

  Proof. We first establish

$$\frac{\mathsf{valid\text{-}footprint}\ d\ p\ s\ m \qquad \{p..{+}m\} \cap \{q_\&..{+}\mathsf{size\text{-}of}\ \mathsf{TYPE}(\beta)\} = \emptyset \qquad 0 < m}{\mathsf{valid\text{-}footprint}\ (\mathsf{ptr\text{-}tag}\ q\ d)\ p\ s\ m}$$

  by Defn. 3.2.2, interval reasoning and observing from Defn. 3.2.4:

$$\frac{x \notin \{q_\&..{+}\mathsf{size\text{-}of}\ \mathsf{TYPE}(\beta)\}}{\mathsf{ptr\text{-}tag}\ q\ d\ x = d\ x}$$

From this we can deduce

$$\frac{q_{\&} = r \qquad \begin{array}{c}\textsf{valid-footprint}\ d\ p\ s\ m \\ \textsf{size-of}\ \textsf{TYPE}(\beta) \leq n\end{array} \qquad \begin{array}{c}\textsf{valid-footprint}\ d\ r\ t\ n \\ s \neq t \vee p \neq q_{\&}\end{array} \qquad 0 < m}{\textsf{valid-footprint}\ (\textsf{ptr-tag}\ q\ d)\ p\ s\ m}$$

since

$$\frac{\textsf{valid-footprint}\ d\ p\ s\ m \qquad \textsf{valid-footprint}\ d\ r\ t\ n \qquad p \neq r}{\{p..{+}m\} \cap \{r..{+}n\} = \emptyset}$$

using the same reasoning as in the proof of Thm. 3.2.2, which solves the goal after unfolding with Defn. 3.2.2. □

3.2.4 Annotations

Code annotations are added to update the heap type description when the intended type of a region of memory changes. Syntactically, they are C comments of the form:

$$/{*}{*}\ AUXUPD\text{: }(g,\ f)\ {*}/$$

where f is an expression that may depend on any program variable or the heap type description and yields a new heap type description, and g is a guard predicate on the current state. A guard could require that retypes only affect locations in the existing domain of the heap type description, such as ptr-safe in §4.4, providing a form of memory safety on retypes. An example annotation for such a retype would be:

$$/{*}{*}\ AUXUPD\text{: }(\textsf{ptr-safe}\ {'p}\ {'d},\ \textsf{ptr-tag}\ ({'p}::{'a}\ ptr)\ {'d})\ {*}/$$

Annotations are translated to the *com* language as the following statement:

$$\textsf{Guard SafetyError}\ \{s.\ g\ s\}\ (\textsf{Basic}\ (\lambda s.\ s(\!|\ d := f\ s\ |\!)))$$

Annotations are flexible and allow for additional retype operations to be defined for an application, e.g. for retyping arbitrary sized arrays inside an allocator. They only introduce additional guards and modify the state of the heap type description ghost variable. Consequently, there is no effect on soundness.

3.3 Lifting

So far, the effect of updates on the lifted heap can only be expressed pointwise; we can determine that a heap derived with lift at pointer p is not affected by an update at pointer q if both are valid. We cannot determine that if the **float** incarnation of the lifted heap changes, the whole **unsigned int** incarnation, as a function, is unaffected.

This means that if we had, for instance, a heap invariant or abstraction function for a linked list structure that only uses the **unsigned int *** incarnation of the lifted heap, we would need to prove a separate rule for that abstraction function to show that it remains unchanged under **float** updates — even if the abstraction function explicitly states that all its pointers are valid.

In this section we lift the *heap-mem* and *heap-typ-desc* state to a set of independent heap functions, providing the ability to write assertions and reason about multiple typed heaps in proofs. This follows a two-stage process, where first the two components are combined and then transformed into a polymorphic lifting function. The split facilitates later layering of the separation logic embedding. We describe the stages in the process here and then the properties of the composed lifting function.

1. The two heap related components are combined into a single function.

   Definition 3.3.1. The first stage results in an intermediate *heap-state*.

   $$\textbf{types}\ \textit{heap-state} = \textit{addr} \rightharpoonup \textit{typ-tag option} \times \textit{byte}$$

   Definition 3.3.2. The function lift-state takes as a single parameter a *heap-mem* $\times$ *heap-typ-desc* tuple, filters out locations that are $\bot$ in the heap type description, removing values that should not affect the final lifted typed heaps, and yields a *heap-state*:

   $$\mathsf{lift\text{-}state} \equiv \lambda(h,\ d)\ x.\ \mathsf{case}\ d\ x\ \mathsf{of}\ \bot \Rightarrow \bot \mid \lfloor t \rfloor \Rightarrow \lfloor (t,\ h\ x) \rfloor$$

   Theorem 3.3.1. *Equality between lifted heaps is pointwise component equality modulo the heap type description domain:*

   $$(\mathsf{lift\text{-}state}\ (h,\ d) = \mathsf{lift\text{-}state}\ (h',\ d')) = ((\forall x \in \mathsf{dom}\ d.\ h\ x = h'\ x) \wedge d = d')$$

   Proof. It is required that both heap type descriptions have the same domain and are pointwise equal, since they are directly visible after the lift-state in the first component of the range. The proof is completed by applying definitions and function extensionality. □

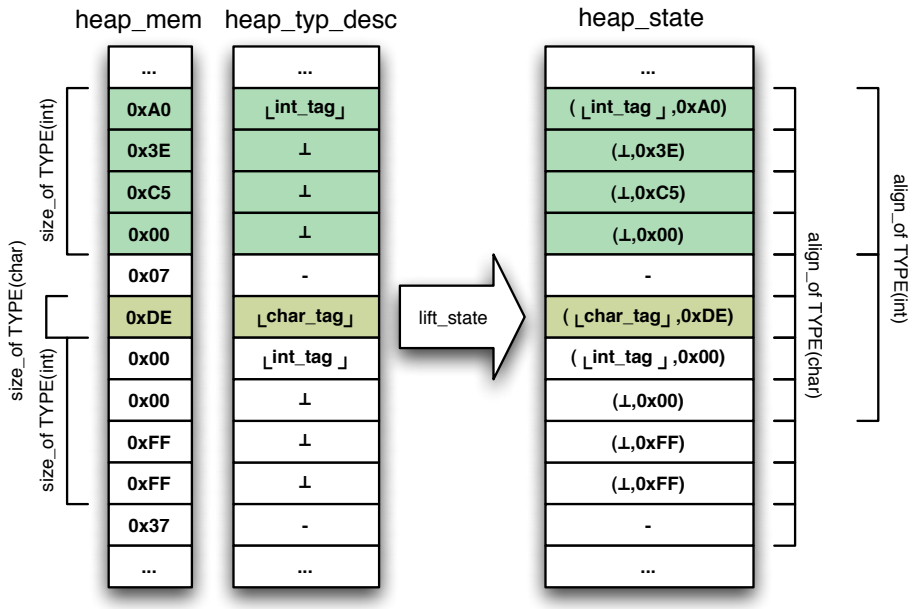

Figure 3.2: First stage lifting.

Example 3.3.1. Fig. 3.2 illustrates the effect of lift-state on a running example for this section. The only locations dropped in the lifted state are those with $\bot$ in the heap type description — even incorrectly aligned locations and those with invalid footprints are preserved. These are filtered out at the next lifting stage.

The following additional definitions are useful later when operating at the *heap-state* level.

Definition 3.3.3. Projection functions proj-h and proj-d, from the intermediate *heap-state*, that satisfy $x \in \mathsf{dom}\ d \Longrightarrow \mathsf{proj\text{-}h}\ (\mathsf{lift\text{-}state}\ (h,\ d))\ x = h\ x$ and $\mathsf{proj\text{-}d}\ (\mathsf{lift\text{-}state}\ (h,\ d)) = d$ can be defined as:

$$\mathsf{proj\text{-}h}\ s \equiv \lambda x.\ \mathsf{case}\ s\ x\ \mathsf{of}\ \bot \Rightarrow \mathsf{arbitrary} \mid \lfloor (t,\ k) \rfloor \Rightarrow k$$
$$\mathsf{proj\text{-}d}\ s \equiv \lambda x.\ \mathsf{case}\ s\ x\ \mathsf{of}\ \bot \Rightarrow \bot \mid \lfloor (t,\ k) \rfloor \Rightarrow \lfloor t \rfloor$$

Definition 3.3.4. These projection functions then allow us to define lifted validity and heap-list on *heap-states* with $s,g \models_s p$ and heap-list-s respectively:

$$s,g \models_s p \equiv \mathsf{proj\text{-}d}\ s,g \models_t p$$
$$\mathsf{heap\text{-}list\text{-}s}\ s\ n\ p \equiv \mathsf{heap\text{-}list}\ (\mathsf{proj\text{-}h}\ s)\ n\ p$$

2. The second lifting stage results in multiple typed heaps again. We supply a single polymorphic definition that provides a distinct heap for each language type. The intended heap type in a specification or proof is implicit — there are usually no type annotations. Instead the type is discovered from use through Isabelle's type inference, based on the phantom pointer types in §2.4.2.

   Definition 3.3.5. The lift-typ-heap function, with the type signature α *ptr-guard* $\Rightarrow$ *heap-state* $\Rightarrow$ (α::*c-type ptr* $\rightharpoonup$ α), restricts the domain such that the only values affecting the resultant heap are inside the heap footprint of valid pointers of the corresponding type. It also converts appropriately sized *byte list*s at the address of valid pointers to typed values:

   $$\begin{array}{l}\mathsf{lift\text{-}typ\text{-}heap}\ g\ s \equiv \\ (\mathsf{from\text{-}bytes} \circ \mathsf{heap\text{-}list\text{-}s}\ s\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)) \circ \mathsf{ptr\text{-}val})\restriction_{\{p\ \mid\ s,g \models_s p\}}\end{array}$$

   This is equivalent to the following definition, which is sometimes easier to reason about:

   $$\begin{array}{l}\mathsf{lift\text{-}typ\text{-}heap}\ g\ s \equiv \\ \lambda p.\ \mathsf{if}\ s,g \models_s p\ \mathsf{then}\ \mathsf{from\text{-}bytes}\ (\mathsf{heap\text{-}list\text{-}s}\ s\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha))\ p_\&)\ \mathsf{else}\ \bot\end{array}$$

   Example 3.3.2. In the term lift-typ-heap c-guard $s\ q = \lfloor -1 \rfloor \longrightarrow (\exists j.$ lift-typ-heap c-guard $s\ p = \lfloor j \rfloor \wedge k = j + 1)$, where *1*::*word32*, type inference gives that p is of type **unsigned int *** and q is some **signed** pointer type, with these types then forming the domain of the lifted heaps respectively. Hence two distinct heaps occur in the term.

   Example 3.3.3. Fig. 3.3 illustrates the effect of lift-typ-heap on a *heap-state* resulting from lift-state. As well as now mapping to multiple distinct heaps, this lifting stage filters out invalid pointer locations and performs decoding of the object representation.

Definition 3.3.6. The two stages are combined with lift_τ, shown in Fig. 3.4:

$$\mathsf{lift}_\tau\ g \equiv \mathsf{lift\text{-}typ\text{-}heap}\ g \circ \mathsf{lift\text{-}state}$$

Like lift, lift_τ is polymorphic and returns a heap abstraction of type α *typ-heap* $= \alpha$ *ptr* $\rightharpoonup \alpha$. The program text itself can continue to use the functions lift and heap-update, while pre/post conditions and invariants use the stronger lift_τ to make more precise statements.

The characteristic properties used in later proofs are provided below.

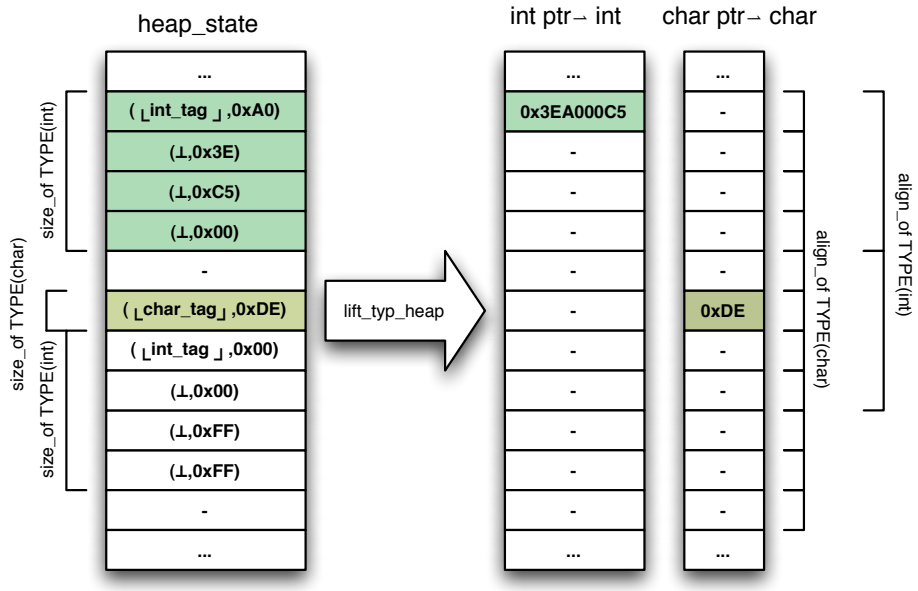

Figure 3.3: Second stage lifting.

Lemma 3.3.2. *Lifted validity after* lift-state *is equivalent to pointer validity on the underlying heap state:*

$$\text{lift-state } (h,\ d),g \models_s p = d,g \models_t p$$

Proof. Note proj-d (lift-state $(h,\ d)$) $= d$ by expanding definitions and extensionality, the result follows from unfolding Defn. 3.2.2 and simplifying. □

Lemma 3.3.3. *The value of the lifted* heap-list-s *on a* lift-state *at the address given by a valid pointer* $p::\alpha$ *ptr, when considering a byte list of length* size-of TYPE(α) *is equivalent to* heap-list *applied to the underlying heap state:*

$$\frac{d,(\lambda x.\ \text{True}) \models_t p}{\begin{array}{c}\text{heap-list-s } (\text{lift-state } (h,\ d))\ (\text{size-of TYPE}(\alpha))\ p_\& \\ = \text{heap-list } h\ (\text{size-of TYPE}(\alpha))\ p_\&\end{array}}$$

Proof. Via induction on k we can establish $\{n..{+}k\} \subseteq \text{dom } d \Longrightarrow$ heap-list-s (lift-state $(h,\ d)$) $k\ n =$ heap-list $h\ k\ n$. Thm. 3.2.3 and Defn. 3.2.3 can then be used to derive from pointer validity an instantiation of this result to complete the proof. □

Theorem 3.3.4. *An alternative definition of* lift_τ *that provides a connection with* lift *is:*

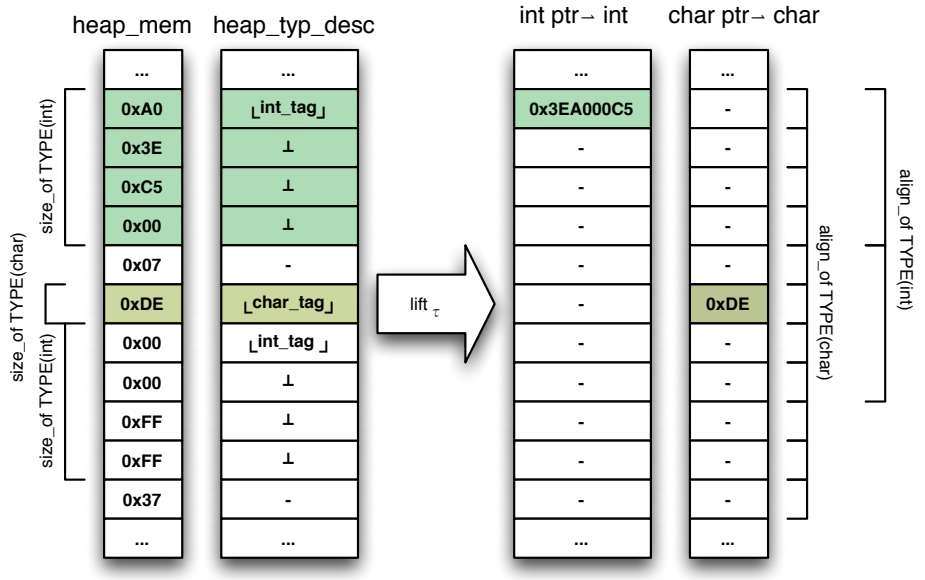

Figure 3.4: Combined lifting.

$$\mathsf{lift}_\tau\ g\ (h,\ d) \equiv \lambda p.\ \mathsf{if}\ d,g \models_t p\ \mathsf{then}\ \mathsf{h\text{-}val}\ h\ p\ \mathsf{else}\ \bot$$

Proof. After expanding lift_τ and the alternate definition for lift-typ-heap, letting $s = \mathsf{lift\text{-}state}\ (h,\ d)$, Lem. 3.3.2 and Lem. 3.3.3 yield the above. □

Corollary. *Existence of a typed heap mapping at p implies validity:*

$$\frac{\mathsf{lift}_\tau\ g\ (h,\ d)\ p = \lfloor x \rfloor}{d,g \models_t p}$$

This alternate definition is somewhat simpler than the two-stage lifting process, but we develop the framework based on the original definition as the intermediate *heap-state* also supports the later separation logic embedding and we are able to then reuse proofs developed on this definition. This provides a link in unifying the multiple typed heaps and separation logic memory views.

Theorem 3.3.5. *Pointwise component equality modulo pointer validity is sufficient for lifted heap equality:*

$$\frac{\bigwedge p.\ d,g \models_t p = d',g \models_t p \qquad \bigwedge p.\ d,g \models_t p \longrightarrow (\forall x{\in}\{p_{\&}..{+}\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)\}.\ h\ x = h'\ x)}{\mathsf{lift}_\tau\ g\ (h,\ d) = \mathsf{lift}_\tau\ g\ (h',\ d')}$$

Proof. First we simplify the goal with Thm. 3.3.4. The validity equivalence assumption then reduces the goal to comparison of h-val applied to the two heaps h and h', at locations where there exist valid pointers. We can establish $(\bigwedge x.\ x \in \{p..+n\} \Longrightarrow h\ x = h'\ x) \Longrightarrow$ heap-list $h\ n\ p =$ heap-list $h'\ n\ p$ by induction on n and then apply the second assumption to instantiate this to complete the proof. □

3.4 Rewriting

In this section we present the key results of this chapter, where a set of rewrites are derived to tie together the memory semantics and proof abstraction of the previous section.

3.4.1 Proof obligations

Specifications now feature occurrences of lift_τ on the pre- and post-state and the program semantics make use of lift and heap-update. Following verification condition generation, we are left with proof obligations featuring a mix of these terms.

Example 3.4.1. Table 3.1 presents a definition for a function containing a sequence of assignments containing pointer dereferences, together with a specification.

$\forall \sigma\ k.\ \{\!|\sigma.\ \Phi\ \acute{}q = \lfloor k \rfloor \wedge \mathcal{D} \models_t \acute{}c \wedge \mathcal{D} \models_t \acute{}p|\!\}$
$\quad \acute{}f\text{-}ret :== f(\acute{}p, \acute{}q, \acute{}c)$
$\quad \{\!|\acute{}f\text{-}ret = 2 * k^2 + (\mathsf{if}\ {}^\sigma p = {}^\sigma q\ \mathsf{then}\ k^2\ \mathsf{else}\ k) + 5|\!\}$

```
int f(int *p, int *q, char *c)
{
    *c = 5;
    *p = *q * *q;
    *q = *p + *p + *q;

    return *q + *c;
}
```

Table 3.1: **f** specification and definition.

Here, and in the rest of this thesis, ${}^s\mathcal{H}$ is used as an abbreviation for the heap state in program state s, ${}^s\mathcal{D}$ is used for just the heap type description and ${}^s\Phi$ for the lifted heap state lift_τ c-guard ${}^s\mathcal{H}$ in assertions to provide greater clarity. The antiquotation ^{s}- is dropped when referring to the current state in these abbreviations.

The post-condition is complicated by the possible aliasing between parameters p and q, causing the second assignment to exhibit rather different behaviour in each case.

A fragment of the resulting proof obligation features the mix of abstraction levels mentioned above:

$$\begin{gathered}[\![(\mathsf{lift}_\tau{}^c\ (h,d))\ q = \lfloor k \rfloor;\ d \models_t c;\ d \models_t p]\!] \Longrightarrow \\ \ldots\ \mathsf{lift}\ (\mathsf{heap\text{-}update}\ q\ (3 * (\mathsf{lift}\ (\mathsf{heap\text{-}update}\ c\ 5\ d)\ q * \\ \mathsf{lift}\ (\mathsf{heap\text{-}update}\ c\ 5\ d)\ q))\ (\mathsf{heap\text{-}update}\ q\ (\mathsf{lift}\ (\mathsf{heap\text{-}update} \\ c\ 5\ d)\ q * \mathsf{lift}\ (\mathsf{heap\text{-}update}\ c\ 5\ d)\ q)\ (\mathsf{heap\text{-}update}\ c\ 5\ d))) \\ \ldots = 3 * k^2 + 5\end{gathered}$$

To solve this problem we provide conditional rewrites in Thm. 3.4.1, Thm. 3.4.5, and Thm. 3.4.6, that can be applied with Isabelle's simplifier and leave only lift_τ terms in the goal when updates are type-safe and enough detail is known in the pre-state[2].

3.4.2 Conditional rewrite set

Theorem 3.4.1. *The value of* lift *at valid pointers is equivalent to the value at the same location in the corresponding typed heap:*

$$\frac{d,g \models_t p}{\mathsf{lift}\ h\ p = \mathsf{the}\ (\mathsf{lift}_\tau\ g\ (h,\ d)\ p)}$$

Proof. Follows from Thm. 3.3.4 and the definition of lift. □

Lemma 3.4.2. *The heap function resulting from encoding a value v and updating a heap function at p::α::*mem-type ptr can have a* size-of TYPE(α) *byte list read back from p and decoded to v:*

$$\mathsf{h\text{-}val}\ (\mathsf{heap\text{-}update}\ p\ v\ h)\ p = \lfloor v \rfloor$$

Proof. First, via structural induction on vs, we establish the stronger intermediate result for lists:

$$\frac{|vs| \leq |\mathsf{addr}|}{\mathsf{heap\text{-}list}\ (\mathsf{heap\text{-}update\text{-}list}\ p\ vs\ h)\ |vs|\ p = vs}$$

The base case is trivial, and the $v{\cdot}vs$ case can be seen from

$$\frac{0 < k \qquad k \leq |\mathsf{addr}| - |vs|}{\mathsf{heap\text{-}update\text{-}list}\ (p + \mathbb{N}^{\Rightarrow}\ k)\ vs\ h\ p = h\ p}$$

[2]E.g. In Exmp. 3.4.1 we required $\mathcal{D} \models_t \acute{c}$. In a proof system for a type-safe language we would still require knowledge of a reference's validity, so this does not introduce new overhead.

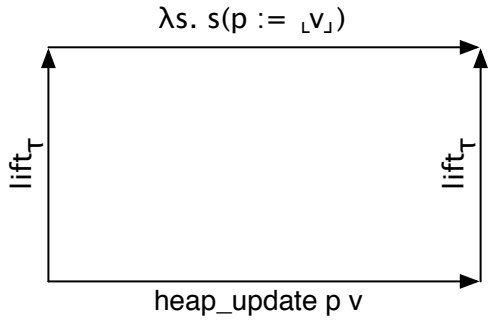

Figure 3.5: Lifted heap updates when the heap is of the same type.

which also follows by structural induction on *vs*.

The intermediate result can be instantiated with to-bytes v and size-of TYPE(α), and with the α::*mem-type* properties and relevant definitions gives the lemma. □

Lemma 3.4.3. *Heap updates do not affect heap reads providing the regions do not overlap:*

$$\frac{\{p..+|v|\} \cap \{q..+k\} = \emptyset}{\mathsf{heap\text{-}list}\ (\mathsf{heap\text{-}update\text{-}list}\ p\ v\ h)\ k\ q = \mathsf{heap\text{-}list}\ h\ k\ q}$$

Proof. By induction on k. □

Lemma 3.4.4. *Updates to the heap function at a valid pointer p do not affect* h-val *at distinct valid pointer locations:*

$$\frac{d,g \models_t p \qquad d,g' \models_t q \qquad p_\& \neq q_\&}{\mathsf{h\text{-}val}\ (\mathsf{heap\text{-}update}\ p\ v\ h)\ q = \mathsf{h\text{-}val}\ h\ q}$$

Proof. Starting with Lem. 3.4.3, we can then apply Defn. 2.5.4, Defn. 2.5.4 and Thm. 3.2.2 to give a complete proof. □

Theorem 3.4.5. *The effect of* heap-update *at p::α::mem-type ptr on lifted heaps of type α typ-heap is function update. Fig. 3.5 depicts this rewrite.*

$$\frac{d,g \models_t p}{\mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d) = \mathsf{lift}_\tau\ g\ (h,\ d)(p \mapsto v)}$$

Proof. We first simplify the goal with Thm. 3.3.4, then by extensionality we compare the functions point-wise, case splitting on whether the point is equal to p. Lem. 3.4.2 and Lem. 3.4.4 finish the proof for each case. □

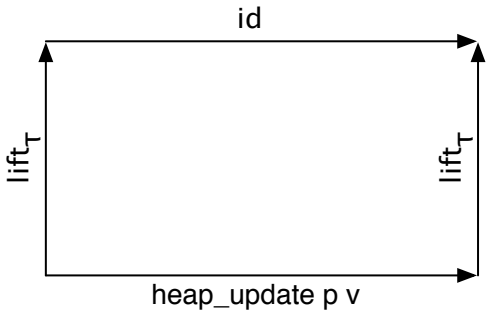

Figure 3.6: Lifted heap updates when the heap is of a different type.

Theorem 3.4.6. *There is no effect of a* heap-update *at* p::α::*mem-type ptr on lifted heaps of type* β *typ-heap, where* α *and* β *are distinct. Fig. 3.6 depicts this rewrite.*

$$\frac{d,g' \models_t p \qquad \mathsf{TYPE}(\alpha)_t \neq \mathsf{TYPE}(\beta)_t}{\mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d) = \mathsf{lift}_\tau\ g\ (h,\ d)}$$

Proof. Similar to Thm. 3.4.5, except only require Lem. 3.4.4 since valid pointers are distinct always based on type. □

Example 3.4.2. We continue Exmp. 3.4.1 by applying the above rewrites, grouped as *c-typ-rewrs* together with some additional rewrites including the corollary from Thm. 3.3.4 and some for handling pointer guards on alignment and non-NULLness, to discharge the proof obligation:

by (*auto simp*: *c-typ-rewrs*)

It is sometimes necessary to supply the simplifier with additional facts. As a technical aside, Isabelle's simplifier does not pursue conditions in the rewrites that introduce schematic variables (i.e. do not appear on the LHS of the rewrite) unless they are a direct match with the assumptions. With this rewrite set we have d, g and g' potentially appearing as schematics, so validity conditions are not always further reduced. One way to get around this is to manually instantiate these variables in the rules prior to application, however, this is not so straightforward since d may be bound by the goal's meta-quantifier[3].

The problem here is that information is lost during verification condition generation, during application of the assignment weakest pre-condition rule,

[3]In our setup g and g' correspond to c-guard which can be handled in this manner as it is a constant.

if the two heap state variables are treated as distinct components of state. Not only is the intended interleaving of heap-updates and ptr-tags lost, but the intended heap type description for a lift reduction is not present. The solution is to bundle the heap components as a tuple *heap-mem* $\times$ *heap-typ-desc* in the program's state.

Definition 3.4.1. During C translation, lifts, heap-updates and ptr-tags interact with the tuple heap state through projection and update functions:

$$\begin{aligned} \textsf{hrs-mem} &\equiv \textsf{fst} \\ \textsf{hrs-mem-update}\ f &\equiv \lambda(h,\ d).\ (f\ h,\ d) \\ \textsf{hrs-htd} &\equiv \textsf{snd} \\ \textsf{hrs-htd-update}\ f &\equiv \lambda(h,\ d).\ (h,\ f\ d) \end{aligned}$$

This forces a serialisation of heap state and type description updates, and provides a target type description for lift reductions. E.g. where before we had lift h we now have lift (hrs-mem $(h,\ d)$).

In this thesis, rules are given as earlier in this section, but when applied they require expressing in a form based on the tuple state and Defn. 3.4.1.

3.4.3 Rewrite properties

Since the rewrite set of the previous section is the core contribution of this chapter, we address some important properties of the rules here. Ideally we would like to prove that they are confluent, terminating and complete, in the sense that they perform the desired goal transformation. Unfortunately we are unable to do so inside Isabelle/HOL proper, since these are meta-statements, but we provide some justification here for these properties holding.

The desired result of applying the rewrites is that any lift $p\ h'$ and $\textsf{lift}_\tau\ g$ (heap-update $p\ v\ h',\ d$) terms are reduced to terms that only reference the heap state through $\textsf{lift}_\tau\ g\ (h,\ d)$ where $(h,\ d)$ is the original state. By original state we refer to the state in the resultant proof obligations corresponding to the pre-state. It is assumed that the required validity information for all pointers in the terms to be reduced is available, either directly in the assumption or through conditional rewrites supplied by the user — even with the usual multiple typed heaps view weakest pre-condition Hoare rules produce obligations requiring this information. In addition we only consider simple blocks here. Weakest pre-condition loop proof obligations are similar in appearance, with the invariant taking the place of the pre/post conditions. Function call rules introduce potentially multiple heap states and type descriptions — this may require multiple rewrites targeting different heap type descriptions and use of the function specifications to connect the states in between, but in each specific rewrite there will only be one target heap type description. The functions need to have sufficient specifications to allow

this to work, again this would be required to tackle the validity conditions in a type-safe language.

We proceed to show completeness by showing lifts and then heap-updates can always be eliminated through rewriting. First note that any lift h' p can be reduced to the (lift$_\tau$ g (h', d) p) using Thm. 3.4.1, providing d,g $\models_t$ p. Since changes to the heap type description do not occur in the type-safe fragment, d refers to the original state, and so all lift h' p terms can be reduced, as we are assuming we can obtain validity for p.

This leaves terms in the form lift$_\tau$ g (heap-update p v h', d) to reduce — the only way in which the semantics produces updates to the *heap-mem* component of the heap state is through heap-update. Thm. 3.4.5 and Thm. 3.4.6 can reduce these terms as required, based on whether the type of p and the lifted heap match. Validity is taken from the original heap type description as above. The typ-tag disequality is handled by unfolding the definitions and comparing literals. Therefore, the set of rewrites is complete.

At each step, the LHS of only one rule matches any term and subterms that may be further reduced are left unchanged. The rewrite set is hence confluent.

Finally, the number of lifts and heap-updates decreases with each rule. The additional goals arising from conditions in the rewrites produce either direct matches with the assumptions or are not further reduced by this set. Hence the rewrite set is terminating.

These last two properties are given for the rewrite set of the previous section, but if the user supplies additional rewrites or function specifications are added, they may need to be reconsidered on a case-by-case basis.

3.4.4 Rules for unsafe code

The rules of §3.4.2 apply when dereferences respect the heap type description, i.e. when the C program is inside the type-safe fragment of the language. When the code strays, all is not lost, as additional rewrites can be developed, at the expense of additional proof effort. We consider this to be a satisfactory situation — TANSTAAFL [40].

Example 3.4.3. In Table 3.2 a pointer cast is added to Exmp. 3.4.1, and all arithmetic is **unsigned**.

The rewrite set in §3.4.2 gets stuck on this example, unable to reduce the lift on the heap access through the dereferenced pointer resulting from the cast. The following rule can be proven to hold for little-endian encodings:

$$\frac{\text{hrs-htd } s,g \models_t q}{\text{ucast (lift (hrs-mem } s\text{) (ptr-coerce } q\text{))} = \text{the (lift}_\tau\ g\ s\ q\text{) mod } 256}$$

where ptr-coerce q = Ptr $q_{\&}$. Adding this to the rewrite set allows auto to complete the proof.

$\forall\, \sigma\ k.\ \{\!|\sigma.\ \Phi\ \acute{}q = \lfloor k \rfloor \wedge \mathcal{D} \models_t \acute{}c \wedge \mathcal{D} \models_t \acute{}p|\!\}$
$\acute{}g\text{-}ret :== g(\acute{}p, \acute{}q, \acute{}c)$
$\{\!|\acute{}g\text{-}ret = 2 * k^2 + (\mathsf{if}\ {}^{\sigma}p = {}^{\sigma}q\ \mathsf{then}\ k^2\ \mathsf{else}\ k)\ \mathsf{mod}\ 256 + 5|\!\}$

```
unsigned int g(unsigned int *p, unsigned int *q, char *c)
{
    *c = 5;
    *p = *q * *q;
    *q = *p + *p + *(unsigned char *)q;

    return *q + *c;
}
```

Table 3.2: **g** specification and definition.

The rule added for the example depended on endianess, the size of encodings, sign and the semantics of integer promotion in C. Rules for unsafe code can be highly application and/or implementation specific, but there are situations where it is possible to be a bit more general, such as the following theorem.

Example 3.4.4. If a retype is performed on the address of a valid pointer $r::\gamma\ ptr$ and the size of the value representation of the target $\beta::mem\text{-}type$ is less than or equal to the size of γ's representation, then distinct typed lifted heaps $\alpha::mem\text{-}type\ typ\text{-}heap$ are equivalent to their value prior to the retype:

$$\frac{d,g' \models_t r \qquad q_{\&} = r_{\&} \qquad \mathsf{size\text{-}of}\ \mathsf{TYPE}(\beta) \leq \mathsf{size\text{-}of}\ \mathsf{TYPE}(\gamma) \qquad \mathsf{TYPE}(\alpha)_t \neq \mathsf{TYPE}(\beta)_t \qquad \mathsf{TYPE}(\alpha)_t \neq \mathsf{TYPE}(\gamma)_t}{\mathsf{lift}_\tau\ g\ (h,\ \mathsf{ptr\text{-}tag}\ q\ d) = \mathsf{lift}_\tau\ g\ (h,\ d)}$$

The proof follows from Thm. 3.3.4, the rule from Exmp. 3.2.3 and a similar rule that can be derived in the other direction.

A piece is missing from the story however and that is how we handle the affected heaps. Inside allocators this is a messy business as it will involve the details of the data structures used to manage free memory, but clients should not be exposed to this. Exmp. 3.5.2 in §3.5 provides an idea of how this can achieved.

3.5 Typed heap equivalence

3.5.1 Inter-type framing

When writing specifications for functions that mutate the heap state, it is desirable to be able to express what the the code does not do, as well as what

it does, i.e. which areas in the heap remain unchanged. This is an example of the frame problem [61]. At the granularity of typed heaps, we call this problem *inter-type framing*. In this section we provide predicates and rules to express in specifications concisely the typed heaps that are unchanged by the code. We take advantage of the feature in the C type encoding that allows language types to be expressed as first-class values in the theorem prover in doing so.

Definition 3.5.1. The *heap footprint* for a language type is the set of locations that fall within the footprint of valid pointers of that type in the heap type description:

$$\mathsf{heap\text{-}footprint}\ d\ t\ n \equiv \{x \mid \exists y.\ \mathsf{valid\text{-}footprint}\ d\ y\ t\ n \wedge x \in \{y\} \cup \{y..+n\}\}$$

The key predicate used in specifications follows from this.

Definition 3.5.2. Two raw heap states s::*heap-mem* $\times$ *heap-typ-desc* and s' are identical modulo a set of language types given by a type tag set T if for all $t \notin T$ the heap states have identical heap footprints for t and are pointwise equivalent inside these heap footprints:

$$\begin{array}{l} s \doteq_f^{-T} s' \equiv \\ \forall\, t\ x.\ t \notin T \wedge \\ \quad (x \in \mathsf{heap\text{-}footprint}\ (\mathsf{snd}\ s)\ t\ (f\ t) \vee \\ \quad\ x \in \mathsf{heap\text{-}footprint}\ (\mathsf{snd}\ s')\ t\ (f\ t)) \longrightarrow \\ \quad \mathsf{lift\text{-}state}\ s\ x = \mathsf{lift\text{-}state}\ s'\ x \end{array}$$

The second parameter, f, is a function *typ-tag* $\Rightarrow$ *nat*. This supplies the type size information that is only otherwise available through the size-of function, which requires α *itself*, i.e. not the tag alone. It can be thought of as a cache, with later rules having consistency conditions that need to be resolved prior to being able to make use of the predicate in relation to specific typed heaps. We drop f in examples where it is a constant.

Theorem 3.5.1. $\doteq$ *is monotonic and has the usual equivalence relation properties:*

$$\frac{T \subseteq T'}{s \doteq_f^{-T} s' \longrightarrow s \doteq_f^{-T'} s'} \quad [\textsc{HMono}]$$

$$\frac{sa \doteq_f^{-T} s' \qquad s' \doteq_f^{-T'} s}{sa \doteq_f^{-T \cup T'} s} \quad [\textsc{HUn}]$$

$$s \doteq_f^{-T} s \quad [\textsc{HRefl}]$$

$$(s \doteq_f^{-T} s') = (s' \doteq_f^{-T} s) \quad [\textsc{HSym}]$$

Proof. Follows from unfolding the $\doteq$ definition. □

3.5.2 Callee rules

Inside a function that modifies the heap, rules are required to establish $\doteq$ in the specification.

Theorem 3.5.2. *Heap updates through a valid pointer p::α::mem-type ptr only affect locations in the heap footprint of the target type:*

$$\frac{d,g \models_t p}{(\textsf{heap-update}\ p\ v\ h,\ d) \doteq_f^{-\{\textsf{TYPE}(\alpha)_t\}} (h,\ d)}$$

Proof. heap-updates only modify locations inside the target pointer footprint. This footprint is inside the target type's heap footprint. Since p is valid, it cannot directly alias another valid pointer of a different type. Thm. 3.2.2 gives that valid footprints do not overlap in this case, so other types' heap footprints will be disjoint and hence unaffected. □

Corollary. *The above can be expressed as a rewrite suitable for reducing over multiple heap updates:*

$$\frac{d,g \models_t p \qquad \textsf{TYPE}(\alpha)_t \in T}{((\textsf{heap-update}\ p\ v\ h,\ d) \doteq_f^{-T} (h',\ d')) = ((h,\ d) \doteq_f^{-T} (h',\ d'))}$$

In addition, rules are required to cope with ptr-tag, but these will be application specific.

Example 3.5.1. A typed allocator for **unsigned int**s is specified in Table 3.3.

The pre-condition contains only the allocator's invariant as the predicate alloc-inv, which is also preserved in the post-condition (clients require this for future calls). The details of the predicate are specific to the implementation, but depend only on the state of two allocator-related global variables.

In the post-condition, there are two possibilities. Either the allocator returns NULL, indicating potentially resource exhaustion, or a non-NULL **unsigned int *** pointer that has not been previously valid in the heap type description. The next two conjuncts describe how pointer validity and the lifted typed heap for **unsigned int** change as a result of a successful outcome.

In all cases, the $\doteq$ component states that the only affected heaps are those belonging to allocator variables and **unsigned int**. This is required by clients that have valid pointers to objects of other types that need to be preserved across the call.

A simple implementation is also contained in Table 3.3. It only manages one memory location, so would not be very useful in practice, but serves an

```
∀ σ. {|σ. alloc-inv (Φ free-flag-addr) (Φ free-mem-addr)|}
     ´alloc-ret :== alloc()
     {|(´alloc-ret = NULL ∨
       ¬ σD ⊨t ´alloc-ret ∧
       (∀ p. D ⊨t p = (σD ⊨t p ∨ p = ´alloc-ret)) ∧ Φ = σΦ(´alloc-ret ↦ 0)) ∧
      H ≐−{TYPE(free-mem)t, TYPE(free-flag)t, TYPE(word32)t} σH ∧
      alloc-inv (Φ free-flag-addr) (Φ free-mem-addr)|}
```

```
struct free_mem {
    int block;
} free_mem;

struct free_flag {
    char flag;
} free_flag ;

unsigned int *alloc(void)
{
    if ( free_flag . flag ) {
        unsigned int *p;

        /** AUXUPD:
            (λx. True, ptr-tag ((ptr-coerce free-mem-addr)::word32 ptr)) */

        free_flag . flag = 0;

        p = (unsigned int *)&(free_mem.block);
        *p = 0;

        return p;
    }

    return 0;
}
```

Table 3.3: **alloc** specification and definition.

illustrative purpose. More complete allocator verifications are the subject of Chapter 6.

Two global variables are used, declared as **struct**s to hide implementation details and make clear the distinction from normal client variables. The invariant is defined as:

$$\mathsf{alloc\text{-}inv}\ \mathit{ff}\ \mathit{fm} \equiv \exists k.\ \mathit{ff} = \lfloor k \rfloor \wedge (\mathit{flag}\ k \neq 0 \longrightarrow \mathit{fm} \neq \bot)$$

While this is a simple function, the verification is non-trivial, as it requires developing additional rules for the retype operation. The main proof is about 50 lines, with about 15 lines relating to invariant preservation. The most significant rules required are the typed heap rewrites, Thm. 3.5.2, Thm. 3.2.4, Exmp. 3.4.4, and the following, derived for this example, but not included in the line count as they are somewhat reusable:

$$\frac{d,g \models_t q \qquad p_\& = q_\& \qquad \mathsf{size\text{-}of}\ \mathsf{TYPE}(\beta) \leq \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) \qquad \{\mathsf{TYPE}(\alpha)_t,\ \mathsf{TYPE}(\beta)_t\} \subseteq T \qquad (h,\ d) \doteq_f^{-T} (h',\ d)}{(h,\ \mathsf{ptr\text{-}tag}\ p\ d) \doteq_f^{-T} (h',\ d)}$$

$$\frac{d,g' \models_t q \qquad p_\& = q_\& \qquad \mathsf{size\text{-}of}\ \mathsf{TYPE}(\beta) \leq \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) \qquad g\ p}{\mathsf{ptr\text{-}tag}\ p\ d,g \models_t r = (p = r \vee d,g \models_t r)}$$

where q::α::*mem-type ptr* and p::β::*mem-type ptr*.

It can be seen that verifying unsafe code is not easy, but, as the next example in this section demonstrates, once this unsafe code is isolated behind a specification, type-safe client code verification is much simpler.

3.5.3 Caller rules

Several additional $\doteq$-related rewrites are useful in the verification of calling functions.

Theorem 3.5.3. *Heap type descriptions that are equivalent in the footprint of types given by $\doteq$ have equivalent pointer validity for these types:*

$$\frac{(h,\ d) \doteq_f^{-T} (h',\ d') \qquad \mathsf{TYPE}(\alpha)_t \notin T \qquad \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) = f\ \mathsf{TYPE}(\alpha)_t}{d,g \models_t p = d',g \models_t p}$$

*where p::α::*c-type ptr*.*

Proof. Since the footprint of p is inside $\mathsf{heap\text{-}footprint}\ d\ \mathsf{TYPE}(\alpha)_t$, the definition of $\doteq$ states it remains unchanged. The rest of the proof follows by unfolding the definitions and some interval reasoning. □

Theorem 3.5.4. *Heap states that are equivalent in the footprint of types given by $\doteq$ have equivalent typed lifted heaps for these types:*

$$\frac{(h,\ d) \doteq_f^{-T} (h',\ d') \qquad \mathsf{TYPE}(\alpha)_t \notin T \qquad \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) = f\ \mathsf{TYPE}(\alpha)_t}{\mathsf{lift}_\tau\ g\ (h,\ d) = \mathsf{lift}_\tau\ g\ (h',\ d')}$$

Proof. Using, and by similar reasoning as, Thm. 3.5.3. □

Example 3.5.2. Continuing Exmp. 3.5.1, a calling function **h** is specified and verified here. The Hoare triple and source code for **h**, together with a helper function **cube**, are given in Table 3.5 and Table 3.4.

In contrast to **alloc**, the proof obligations for each of these functions required less than 10 lines of proof script, mostly applications of rewrites. No heap-related rules other than the generic rewrites presented in this chapter as theorems were required and reasoning was mostly automatic.

3.6 Example: In-place list reversal

The examples given in this chapter so far have contained relatively straightforward simple blocks of code and function calls. We conclude this chapter by giving the standard in-place list reversal example from the literature in Table 3.6, which features a linked inductively-defined data structure, abstraction predicate in specifications, iteration, and pointer casts, i.e. the features that create non-trivial aliasing conditions. Mehta and Nipkow [62] use the same example in their more abstract setting.

This features an unsafe pointer cast, however this does not complicate the proof as it respects the heap type description as implied by the abstraction predicate in the pre-condition.

The list abstraction predicate is defined as:

$$\begin{array}{lcl} \mathsf{list}\ s\ []\ i & \equiv & i = \mathsf{NULL} \\ \mathsf{list}\ s\ (x{\cdot}xs)\ i & \equiv & \exists j.\ i_{\&} = x \wedge x \neq 0 \wedge s\ i = \lfloor j \rfloor \wedge \mathsf{list}\ s\ xs\ (\mathsf{Ptr}\ j) \end{array}$$

The loop invariant again is almost the same as in Mehta and Nipkow. We use distinct to say that the list zs does not contain duplicate addresses and rev to reverse the abstract HOL list:

$$\begin{array}{l} \{\!|\exists\, xs\ ys. \\ \quad \mathsf{list}\ \Phi\ xs\ \acute{}i \wedge \\ \quad \mathsf{list}\ \Phi\ ys\ (\mathsf{Ptr}\ \acute{}j) \wedge \\ \quad \mathsf{rev}\ zs = \mathsf{rev}\ xs\ @\ ys \wedge \mathsf{distinct}\ (\mathsf{rev}\ zs) \wedge \mathcal{H} \doteq^{-\{\mathsf{TYPE}(word32)_t\}} \sigma_{\mathcal{H}} |\!\} \end{array}$$

To give a more concrete idea of what the proof obligations and verification proofs look like, we give the proof obligations here and then describe the bulk of the proof after.

$\forall\, \sigma\ k.\ \{\!|\sigma.\ \Phi\ \acute{}p = \lfloor k \rfloor|\!\}$
$\quad cube(\acute{}p)$
$\quad \{\!|\Phi = {}^{\sigma}\Phi(^{\sigma}p \mapsto k^3) \wedge \mathcal{H} \doteq^{-\{\mathsf{TYPE}(word32)_t\}} {}^{\sigma}\mathcal{H} \wedge \mathcal{D} = {}^{\sigma}\mathcal{D}|\!\}$

```
void cube(unsigned int *p)
{
    *p = *p * *p * *p;
}
```

Table 3.4: **cube** specification and definition.

$\forall\, \sigma\ j\ k.$
$\quad \{\!|\sigma.\ \Phi\ \acute{}p = \lfloor j \rfloor \wedge$
$\qquad \Phi\ \acute{}q = \lfloor k \rfloor \wedge$ alloc-inv (Φ free-flag-addr) (Φ free-mem-addr)$|\!\}$
$\quad \acute{h}\text{-}ret :== h(\acute{}p, \acute{}q)$
$\quad \{\!|(\acute{h}\text{-}ret = \mathsf{NULL} \vee \Phi\ \acute{h}\text{-}ret = \lfloor 8 * j^3 +$ ucast $k \rfloor) \wedge$
$\quad$ alloc-inv (Φ free-flag-addr) (Φ free-mem-addr)$|\!\}$

```
unsigned int *h(unsigned int *p, unsigned char *q)
{
    unsigned int *r;

    r = alloc();

    if (!r)
        return 0;

    *p = 2 * *p;

    cube(p);

    *r = *p + *q;

    return r;
}
```

Table 3.5: **h** specification and definition.

```
∀ zs σ.
   {|σ. list Φ zs 'i|}
   'reverse-ret :== reverse('i)
   {|list Φ (rev zs) (Ptr 'reverse-ret) ∧ H ≐−{TYPE(word32)t} σH|}


word_t reverse(word_t *i) {
    word_t j = 0;

    while (i) {
        word_t *k = (word_t *)*i;

        *i = j;
        j = (word_t)i;
        i = k;
    }

    return j;
}
```

Table 3.6: **reverse** specification and definition.

1. $\bigwedge zs\ t\text{-}hrs\ i.$
   list (lift$_\tau{}^c$ $t\text{-}hrs$) $zs\ i \Longrightarrow$
   $\exists\, xs\ ys.$
   list (lift$_\tau{}^c$ $t\text{-}hrs$) $xs\ i \wedge$
   list (lift$_\tau{}^c$ $t\text{-}hrs$) ys (Ptr (ucast 0)) $\wedge$
   rev zs = rev xs @ $ys \wedge$
   distinct (rev zs) $\wedge$ $t\text{-}hrs \doteq^{-}\{\mathsf{TYPE}(word32)_t\}$ $t\text{-}hrs$
2. $\bigwedge zs\ t\text{-}hrs\ t\text{-}hrsa\ j\ i.$
   $[\![\exists\, xs\ ys.$
   list (lift$_\tau{}^c$ $t\text{-}hrsa$) $xs\ i \wedge$
   list (lift$_\tau{}^c$ $t\text{-}hrsa$) ys (Ptr j) $\wedge$
   rev zs = rev xs @ $ys \wedge$
   distinct (rev xs @ ys) $\wedge$ $t\text{-}hrsa \doteq^{-}\{\mathsf{TYPE}(word32)_t\}$ $t\text{-}hrs$;
   $i \neq$ NULL$]\!]$
   $\Longrightarrow$ $i \neq$ NULL $\wedge$
   ptr-aligned $i \wedge$
   $(\exists\, xs\ ys.$
   list (lift$_\tau{}^c$ (hrs-mem-update (heap-update $i\ j$) $t\text{-}hrsa$)) xs
   (Ptr (lift (hrs-mem $t\text{-}hrsa$) i)) $\wedge$
   list (lift$_\tau{}^c$ (hrs-mem-update (heap-update $i\ j$) $t\text{-}hrsa$)) ys
   (Ptr $i_{\&}$) $\wedge$
   rev zs = rev xs @ $ys \wedge$
   distinct (rev xs @ ys) $\wedge$
   hrs-mem-update (heap-update $i\ j$)
   $t\text{-}hrsa \doteq^{-}\{\mathsf{TYPE}(word32)_t\}$ $t\text{-}hrs$)
3. $\bigwedge zs\ t\text{-}hrs\ t\text{-}hrsa\ j\ i.$

$$
\begin{array}{l}
[\![\exists\, xs\ ys.\\
\quad \mathsf{list}\ (\mathsf{lift}_\tau{}^c\ t\text{-}hrsa)\ xs\ i\ \wedge\\
\quad \mathsf{list}\ (\mathsf{lift}_\tau{}^c\ t\text{-}hrsa)\ ys\ (\mathsf{Ptr}\ j)\ \wedge\\
\quad \mathsf{rev}\ zs = \mathsf{rev}\ xs\ @\ ys\ \wedge\\
\quad \mathsf{distinct}\ (\mathsf{rev}\ zs)\ \wedge\ t\text{-}hrsa \doteq -\{\mathsf{TYPE}(word32)_t\}\ t\text{-}hrs;\\
\neg\ i \neq \mathsf{NULL}]\!]\\
\Longrightarrow \mathsf{list}\ (\mathsf{lift}_\tau{}^c\ t\text{-}hrsa)\ (\mathsf{rev}\ zs)\ (\mathsf{Ptr}\ j)\ \wedge\\
\quad t\text{-}hrsa \doteq -\{\mathsf{TYPE}(word32)_t\}\ t\text{-}hrs
\end{array}
$$

where $\mathsf{lift}_\tau{}^c$ is an abbreviation for lift_τ c-guard.

Most of the proof is simply rewriting based on the following rules.

Lemma 3.6.1.

$$\mathsf{list}\ s\ xs\ \mathsf{NULL} = (xs = [\,]) \qquad [\textsc{empty}]$$

$$\frac{\mathsf{list}\ s\ xs\ p}{\mathsf{distinct}\ xs} \qquad [\textsc{distinct}]$$

$$\frac{\mathsf{list}\ s\ xs\ p \qquad p \neq \mathsf{NULL}}{p_\& \in set\ xs} \qquad [\textsc{mem}]$$

$$\frac{x_\& \notin set\ xs}{\mathsf{list}\ (s(x \mapsto v))\ xs\ p = \mathsf{list}\ s\ xs\ p} \qquad [\textsc{ign}]$$

$$\frac{\mathsf{list}\ (\mathsf{lift}_\tau{}^c\ (h,\ d))\ xs\ p \qquad q_\& \in set\ xs}{d \models_t q} \qquad [\textsc{valid}]$$

Proof. [EMPTY] and [MEM] follow from case splitting on xs. The proof for [DISTINCT] involves inductive proofs to establish $[\![\mathsf{list}\ s\ xs\ p;\ \mathsf{list}\ s\ ys\ p]\!] \Longrightarrow xs = ys$ and $\mathsf{list}\ s\ (xs\ @\ ys)\ p \Longrightarrow \exists\, q.\ \mathsf{list}\ s\ ys\ q$ first. [VALID] follows from $[\![\mathsf{list}\ s\ xs\ p;\ q_\& \in set\ xs]\!] \Longrightarrow \exists\, x.\ s\ q = \lfloor x \rfloor$ and Thm. 3.3.4. Finally, [IGN] is the result of an induction on xs giving $\mathsf{Ptr}\ `\ \mathsf{set}\ xs \subseteq X \Longrightarrow \mathsf{list}\ (s\restriction_X)\ xs\ p = \mathsf{list}\ s\ xs\ p$.

Most of these proofs are single line calls to Isabelle's induction and auto tactics with the appropriate facts. □

The proof is completed with the following lines of tactic script following the VCG invocation (where the typed heap rewrites have already been added to the default simplification set):

apply (*clarsimp simp del*: *distinct-rev*)
apply (*case-tac xs*, *fastsimp*)
apply *clarsimp*
apply (*rule-tac x*=*list* **in** *exI*)
by (*auto dest*: *lift*$_\tau$*-h-t-valid*)

In total, the proof script was 90 lines, which does not appear unreasonable for a non-trivial transformation of a linked data structure. When the automation in the proof is taken into consideration, it appears to be comparable to other efforts in the literature [62, 85, 100].

Chapter 4

Separation logic embedding

4.1 Intra-type aliasing and framing

The previous chapter presented a proof technique that tames inter-type aliasing. However, the problem of *intra-type aliasing* remains, where two valid same-typed pointers may have identical values. Dealing with this often requires explicit anti-aliasing invariants on typed heaps, a problem that has been recognised in the literature [14].

Example 4.1.1. Consider a list ADT where head and tail pointers are maintained for efficient append operations:

struct list_obj { **struct** llist *list_hd, * list_tl ; };

A naive specification for the allocation, cons and append operations might be, based on suitable definitions for the allocator invariant and list abstraction predicate, list-alloc-inv and list respectively:

$\{\!|$ list-alloc-inv Φ $|\!\}$ p = *list-alloc*(); $\{\!|$ list Φ [] $'p$ $\wedge$ list-alloc-inv Φ $|\!\}$
$\{\!|\sigma.$ list Φ xs $'p$ $|\!\}$ *list-cons*(p, x); $\{\!|$ list Φ $(^{\sigma}x\#xs)$ $^{\sigma}p$ $|\!\}$
$\{\!|\sigma.$ list Φ xs $'p$ $\wedge$ list Φ ys $'q$ $|\!\}$ *list-append*(p, q); $\{\!|$ list Φ $(xs@ys)$ $^{\sigma}p$ $|\!\}$

The pre-condition for **list_append** is unfortunately not strong enough to verify the following implementation of **list_append**:

```
void list_append(struct list_obj *p, struct list_obj *q)
{
    p->list_tl->next = q->list_hd;
    p->list_tl = q->list_tl;
}
```

This is due to the possibility that the lists described by p and q might overlap in some way (e.g. as a result of a previous **list_append** call), and adjusting these pointers will introduce a cycle in the list, as in Fig. 4.1.

Repairing this situation requires an additional predicate to be added to the pre-condition stating the no-overlap condition and then the pre- and

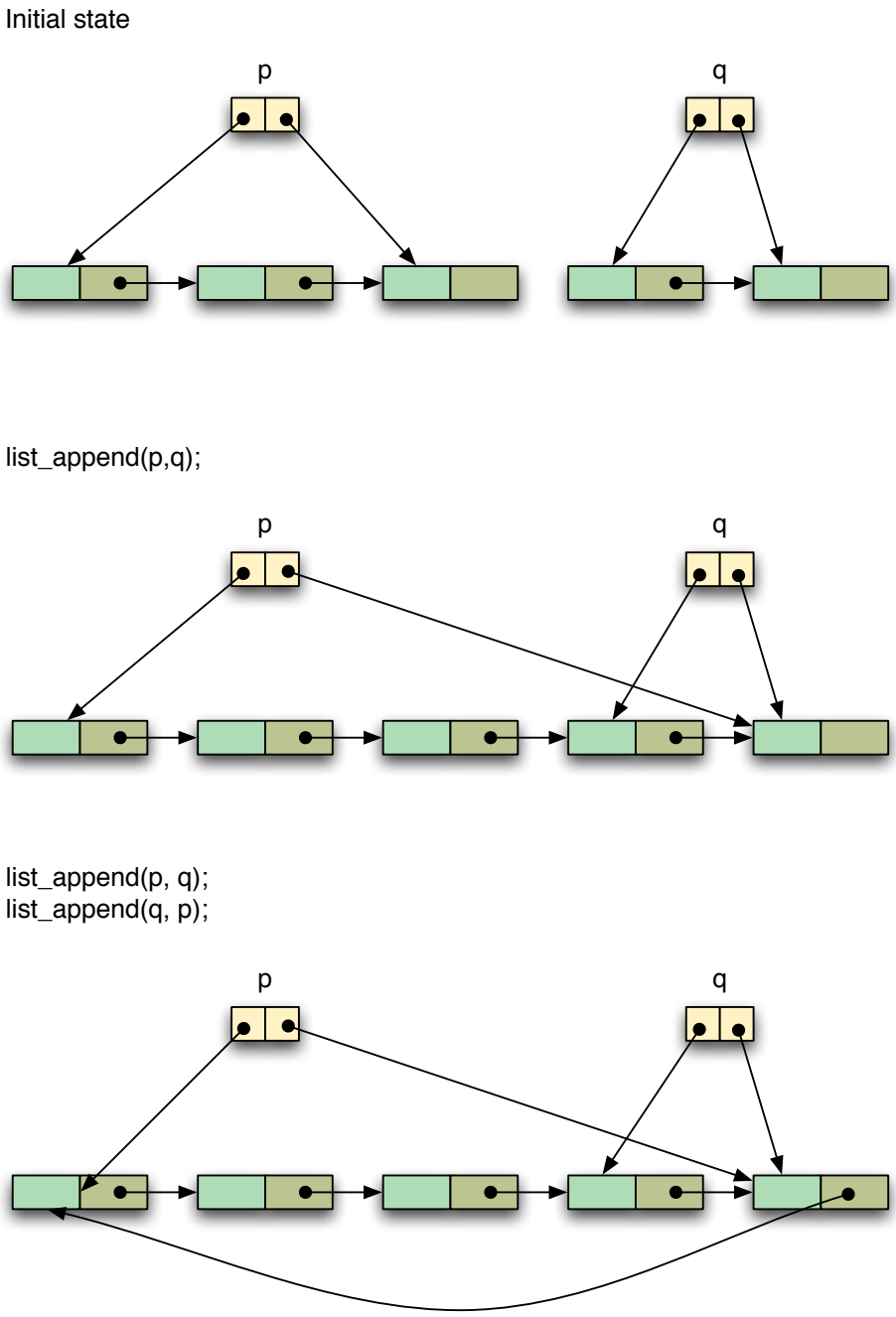

Figure 4.1: Pre-, intermediate and post-state for two consecutive invocations of **list_append**.

post-conditions and invariants on all the other operations to be updated to propagate this information so that clients may satisfy it. A less adhoc and more transparent means of achieving this is desirable. One further motivating factor in seeking a better way is that there is a hidden cost in providing these anti-aliasing predicates — the program verifier's time. As they can be non-obvious and are discovered as a result of a failed proof effort, this can be costly. Invariant discovery is already hard enough without the addition of aliasing concerns.

Similarly, while providing a solution to the frame problem at the typed heap level in §3.5, we cannot yet conveniently express frame conditions about regions of the heap of the same type as those locations modified by a specified function or block.

Example 4.1.2. Suppose a client performs a **list_append**:

```
p = list_alloc ();
q = list_alloc ();
r = list_alloc ();

/* ... p, q & r filled with data ... */

list_append(p, q);
```

After this point, it is not possible to say anything about the list r, since the specification does not express any frame conditions. The h-id-except predicate is not much use here as all lists share the same typed heaps, **struct list_obj** and **struct llist**.

Separation logic provides an approach in which anti-aliasing information may be expressed implicitly in assertions, potentially simplifying specifications and proofs. A significant feature of this assertion language is that it offers a general and scalable solution to the frame problem.

In this chapter we present a development of separation logic based on the preceding memory model. Utilising the HOL encoding of C types, the heap type description and typed heap lifting functions, an embedding of separation logic is described that is able to express assertions about C variables and pointers, rather than the usual typeless and memory-safe languages targeted in the literature. Another novel aspect of this work is that it builds on and effectively reuses two existing foundations — the classical Hoare logic verification environment and the multiple typed heaps memory model development. This necessitates a different approach to tackling proof obligations, as discussed in §4.3 and a careful consideration of the soundness of the frame rule in §4.4.

Figure 4.2: Empty heap predicate.

4.2 Shallow embedding

4.2.1 Definitions

Below we describe a shallow embedding for separation assertions, where the semantic constructs of assertions are translated to HOL, as opposed to a deep embedding where the syntax of assertions would be considered a distinct type in the logic. There is a tradeoff involved in the choice of embedding approach — shallow embeddings are often more pragmatic and expressive, while deep embeddings allow for language meta-theory reasoning and proof optimisations [104]. We opt for the former since our focus in this work is on applications to code verification. In this chapter we do show many properties of the language's connectives and the relation to existing features of the verification environment, but these do not go as far as, for example, showing a completeness result for the proof rules.

Separation assertions are modelled as predicates on *heap-state*s, applied in assertions of the verification environment to the result of the first lifting stage of §3.3. For example, a loop invariant with the separation assertion P is written $\{\!|\ P\ (\mathsf{lift\text{-}state}\ \mathcal{H})\ |\!\}$, which we abbreviate as $\{\!|P^{sep}|\!\}$. Automatic variables can be referenced in separation assertions using the usual antiquotation mechanism, described in §2.3.3, for variables in the verification environment. They do not require special treatment as there already exist Hoare rules for these — here we in fact treat the heap state as a variable and then build additional support for separation logic assertions above.

Definition 4.2.1. As in the development of Reynolds [81] there is an empty heap predicate (Fig. 4.2):

$$\square \equiv \lambda s.\ s = \mathsf{empty}$$

Definition 4.2.2. The definition of the singleton heap assertion is more involved in our embedding and is provided below. $p \mapsto_g v$ asserts that the heap contains exactly one mapping matching the guard g, at the location given by pointer p::α::*c-type ptr* to value v (Fig. 4.3):

$$p \mapsto_g v \equiv \lambda s.\ \mathsf{lift\text{-}typ\text{-}heap}\ g\ s\ p = \lfloor v \rfloor \wedge \mathsf{dom}\ s = \{p_{\&}..{+}\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)\}$$

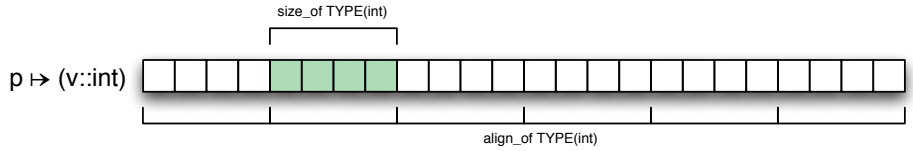

Figure 4.3: Singleton heap predicate.

The guard is an addition to the usual $p \mapsto v$ and serves the same purpose as in Defn. 3.2.2, i.e. strengthening the assertion to aid in discharging guard proof obligations and thereby making the treatment of guards in the framework generic.

The singleton heap assertion requires that predicates be on *heap-state*s, while the empty heap assertion does not, only that the state be a partial function. In the following, definitions of the separation connectives and their properties, with the exception of those involving a singleton heap assertion, are expressed and proven about polymorphic assertions (α, β) *map-assert* $= (\alpha \rightharpoonup \beta) \Rightarrow$ *bool*. The theory is hence somewhat reusable with different heap models, such as in Chapter 5. Here, *heap-state* predicates are the instantiation (*addr*, *typ-tag option* $\times$ *byte*) *map-assert*.

Definition 4.2.3. There are two significant separation connectives, conjunction and implication:

$$\begin{array}{lcl} s_0 \perp s_1 & \equiv & \mathsf{dom}\ s_0 \cap \mathsf{dom}\ s_1 = \emptyset \\ s_0 +\!\!+ s_1 & \equiv & \lambda x.\ \mathsf{case}\ s_1\ x\ \mathsf{of}\ \perp \Rightarrow s_0\ x \mid \lfloor y \rfloor \Rightarrow \lfloor y \rfloor \\ P \wedge^* Q & \equiv & \lambda s.\ \exists s_0\ s_1.\ s_0 \perp s_1 \wedge s = s_0 +\!\!+ s_1 \wedge P\ s_0 \wedge Q\ s_1 \\ P \longrightarrow^* Q & \equiv & \lambda s.\ \forall s'.\ s \perp s' \wedge P\ s' \longrightarrow Q\ (s +\!\!+ s') \end{array}$$

Lemma 4.2.1. *The heap merge and disjointness operators are commutative and associative:*

$$s_0 +\!\!+ s_1 +\!\!+ s_2 = s_0 +\!\!+ (s_1 +\!\!+ s_2)$$

$$s_0 \perp s_1 = s_1 \perp s_0$$

$$\frac{s_0 \perp s_1}{s_0 +\!\!+ s_1 = s_1 +\!\!+ s_0}$$

$$\frac{s_0 \perp s_1}{s_0 +\!\!+ (s_1 +\!\!+ s_2) = s_1 +\!\!+ (s_0 +\!\!+ s_2)}$$

Proof. By unfolding Defn. 4.2.3. □

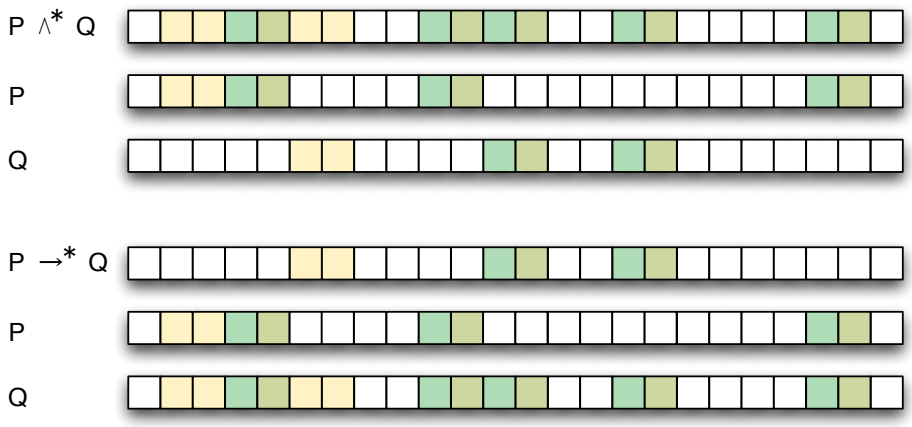

Figure 4.4: Separation connectives.

The definitions are standard, with the intuition behind separation conjunction that $(P \wedge^* Q)\ s$ asserts that s can be partitioned into two subheaps such that P holds on one subheap and Q on the other. The utility of separation implication is easiest to understand in context in §4.3. Example heap states where predicates involving these connectives may hold are shown in Fig. 4.4[1].

Definition 4.2.4. Some additional mapping assertions are common:

$$\begin{array}{lcl} \mathsf{sep\text{-}true} & \equiv & \lambda s.\ \mathsf{True} \\ p \mapsto_g - & \equiv & \lambda s.\ \exists v.\ (p \mapsto_g v)\ s \\ p \hookrightarrow_g v & \equiv & p \mapsto_g v \wedge^* \mathsf{sep\text{-}true} \\ p \hookrightarrow_g - & \equiv & \lambda s.\ \exists x.\ (p \hookrightarrow_g x)\ s \end{array}$$

4.2.2 Properties

The standard commutative, associative and distributive properties apply to the connectives, and we have formalised pure, intuitionistic, domain and strictly exact assertions and their properties [81] — see Table 4.1, Table 4.2, Table 4.3, Table 4.4 and Table 4.5 for their Isabelle/HOL formulation. This exercise was a useful sanity check on the definitions, as earlier attempts at providing a singleton heap assertion based on a weaker non-monotonic notion of validity failed to have some of the intuitionistic properties. Unlike other developments, the singleton heap assertion is not strictly exact, as there can be more than one byte encoding of the heap for which $p \mapsto_g v$ holds. Also, the existential quantification of a predicate with a free variable does not

[1] In the separation implication states in this example, either P is strictly exact or it would be necessary to examine all disjoint states P might hold in.

syntactically preserve domain exactness. The proofs for these properties are given in Isar format in Appendix D.

Some of the properties, and others derived from them, are routinely used in verification proofs and have been added to the default simplification set. Those added to the simplification set tend to be quite specific and direct, e.g. $(p \mapsto_g v)\ s \Longrightarrow (p \hookrightarrow_g v)\ s$, and are not intended to be part of any lengthy sequence of rewrites, as separation logic proofs tend to follow a more rule-oriented approach. The exception to this is the $\wedge^*$ commutative and associative rewrites, that are completed with a derived left-commutative rule to provide a permutative rewrite set for normalising expressions involving this connective. The [Sci] rule is extremely important in proofs, as it provides a means of splitting the problem based on corresponding distinct regions of the heap in the pre- and post-states of proof obligations.

Since this is a shallow embedding, HOL connectives, quantifiers, and constants can be freely mixed with the separation connectives, for example $\lambda s.\ P\ s \wedge (\exists x.\ (p \hookrightarrow \ \mathsf{fib}\ x)\ s \wedge x \in X) \wedge (Q \wedge^* \mathsf{list\text{-}sum}\ X)\ s$.

A key feature of this embedding is that it avoids the problem of *skewed sharing* [81]. This is essentially the problem of inter-type aliasing in separation logic, where for example $\lambda s.\ (p \hookrightarrow \ u)\ s \wedge (q \hookrightarrow \ v)\ s$ describes not only heaps where $p = q \wedge u = v$ or $p \neq q$ and the pointer footprints are distinct, but also the possibilities where p and q point into each other's encoding. An approach where a ghost variable like the heap type description is introduced was suggested as a future direction for separation logic by Reynolds. The embedding given in this chapter has developed this as a machine-checked formalisation.

Another notable gain from the development presented here is the harnessing of Isabelle's type inference to avoid explicit type annotations in assertions. Since language types are assigned Isabelle types and pointer types are derived from these, asserting that $p \mapsto v$, where p is a program variable, automatically constrains the type of v. The alternative of having to write $p \mapsto^{unsigned\ int} v$ is somewhat cumbersome and contributes little to the readability of specifications.

Example 4.2.1. Abstraction predicates can be defined linking algebraic datatypes, sets and functions in Isabelle/HOL with their heap representation. Consider a pre-order traversal representation of a tree, stored in a NULL-terminated linked list with a depth field[2]:

```
struct node {
    unsigned int depth;
    struct node *next;
};
```

[2] A similar data structure, utilising a doubly-linked list instead for efficient node removal, features in the mapping database of L4Ka::Pistachio [49].

$$P \wedge^* \mathsf{sep\text{-}false} = \mathsf{sep\text{-}false}$$

$$P \wedge^* \square = P$$

$$\mathsf{sep\text{-}false} \longrightarrow^* P = \mathsf{sep\text{-}true}$$

$$P \longrightarrow^* \mathsf{sep\text{-}true} = \mathsf{sep\text{-}true}$$

$$P \wedge^* Q = Q \wedge^* P \quad [\textsc{ScComm}]$$

$$(P \wedge^* Q) \wedge^* R = P \wedge^* Q \wedge^* R \quad [\textsc{ScAssoc}]$$

$$((\lambda s.\ P\ s \vee Q\ s) \wedge^* R)\ s = (P \wedge^* R)\ s \vee (Q \wedge^* R)\ s \quad [\textsc{ScdDist}]$$

$$((\lambda s.\ \exists x.\ P\ x\ s) \wedge^* Q)\ s = \exists x.\ (P\ x \wedge^* Q)\ s \quad [\textsc{SExists}]$$

$$\frac{((\lambda s.\ \forall x.\ P\ x\ s) \wedge^* Q)\ s}{(P\ x \wedge^* Q)\ s} \quad [\textsc{SUniv}]$$

$$\frac{((\lambda s.\ P\ s \wedge Q\ s) \wedge^* R)\ s}{(P \wedge^* R)\ s \wedge (Q \wedge^* R)\ s} \quad [\textsc{ScDist}]$$

$$\frac{\bigwedge s.\ P\ s \longrightarrow P'\ s \qquad \bigwedge s.\ Q\ s \longrightarrow Q'\ s}{(P \wedge^* Q)\ s \longrightarrow (P' \wedge^* Q')\ s} \quad [\textsc{Sci}]$$

$$\frac{\bigwedge s.\ (P \wedge^* Q)\ s \longrightarrow R\ s}{P\ s \longrightarrow (Q \longrightarrow^* R)\ s} \quad [\textsc{ScSi}]$$

$$\frac{\bigwedge s.\ P\ s \longrightarrow (Q \longrightarrow^* R)\ s}{(P \wedge^* Q)\ s \longrightarrow R\ s} \quad [\textsc{SiSc}]$$

$$\frac{(p \hookrightarrow_g v)\ s \qquad (p \hookrightarrow_h v')\ s}{v = v'} \quad [\textsc{Sinj}]$$

$$\frac{(P \wedge^* Q)\ s \qquad \bigwedge s.\ P\ s \longrightarrow (p \hookrightarrow_g -)\ s \qquad \bigwedge s.\ Q\ s \longrightarrow (p \hookrightarrow_g -)\ s}{\mathsf{False}} \quad [\textsc{Sinter}]$$

$$\frac{(P \wedge^* (P \longrightarrow^* Q))\ s}{Q\ s} \quad [\textsc{ScSiSame}]$$

Table 4.1: Standard and derived separation logic rules.

$$\mathsf{pure}\ P \equiv \forall\, s\ s'.\ P\ s = P\ s'$$

$$\mathsf{pure}\ P = (P = \mathsf{sep\text{-}true} \vee P = \mathsf{sep\text{-}false})$$

$$\frac{P\ s \wedge Q\ s \qquad \mathsf{pure}\ P \vee \mathsf{pure}\ Q}{(P \wedge^* Q)\ s}$$

$$\frac{(P \wedge^* Q)\ s \qquad \mathsf{pure}\ P \qquad \mathsf{pure}\ Q}{P\ s \wedge Q\ s}$$

$$\frac{\mathsf{pure}\ P}{(\lambda s.\ P\ s \wedge Q\ s) \wedge^* R = (\lambda s.\ P\ s \wedge (Q \wedge^* R)\ s)}$$

$$\frac{(P \longrightarrow^* Q)\ s \qquad \mathsf{pure}\ P}{P\ s \longrightarrow Q\ s} \qquad \frac{P\ s \longrightarrow Q\ s \qquad \mathsf{pure}\ P \qquad \mathsf{pure}\ Q}{(P \longrightarrow^* Q)\ s}$$

Table 4.2: Pure separation assertions.

A mutually recursive abstraction predicate for the tree, maintaining the list structural invariants is given:

$$\textbf{datatype}\ \mathit{tree} \quad = \quad \mathsf{Node}\ \text{“}\mathit{node\ ptr}\text{”}\ \text{“}\mathit{tree\ list}\text{”}$$

$$\begin{array}{lll} \mathsf{tree} & :: & \mathit{tree} \Rightarrow \mathit{node\ ptr} \Rightarrow (\mathit{node\ ptr} \Rightarrow \mathit{heap\text{-}assert}) \Rightarrow \mathit{nat} \Rightarrow \mathit{heap\text{-}assert} \\ \mathsf{tree\text{-}list} & :: & \mathit{tree\ list} \Rightarrow \mathit{node\ ptr} \Rightarrow (\mathit{node\ ptr} \Rightarrow \mathit{heap\text{-}assert}) \Rightarrow \mathit{nat} \Rightarrow \\ & & \mathit{heap\text{-}assert} \end{array}$$

$$\begin{array}{lll} \mathsf{tree}\ (\mathsf{Node}\ p\ ns)\ q\ c\ d & \equiv & \lambda s.\ q = p \wedge \\ & & \quad (\exists\, v.\ (p \mapsto v \wedge^* \mathsf{tree\text{-}list}\ ns\ (\mathsf{next}\ v)\ c\ (d + 1))\ s \\ & & \quad \wedge\ d = \mathbb{N}^{\Leftarrow}\ (\mathsf{depth}\ v)) \\ \mathsf{tree\text{-}list}\ []\ q\ c\ d & \equiv & c\ q \\ \mathsf{tree\text{-}list}\ (n{\cdot}ns)\ q\ c\ d & \equiv & \mathsf{tree}\ n\ q\ (\lambda r.\ \mathsf{tree\text{-}list}\ ns\ r\ c\ d)\ d \end{array}$$

The abstraction predicate traverses the tree, following its structure in the first parameter, with the second parameter providing the current point in the linked-list. The third parameter is a continuation, used when visiting a node to remember how to continue the traversal at the node's next sibling. In a NULL-terminated list this might be initially passed as $\lambda p\ s.\ p = \mathsf{NULL} \wedge \square\ s$. Each node in the abstract tree contains a pointer back to the corresponding linked-list node — if additional information was stored in nodes then this

$$\mathsf{intuitionistic}\ P \equiv \forall\, s\ s'.\ P\ s \wedge s \subseteq_m s' \longrightarrow P\ s'$$

$$\frac{\mathsf{pure}\ P}{\mathsf{intuitionistic}\ P} \qquad \mathsf{intuitionistic}\ (p \hookrightarrow_g v)$$

$$\mathsf{intuitionistic}\ (P \wedge^* \mathsf{sep\text{-}true}) \qquad \mathsf{intuitionistic}\ (\mathsf{sep\text{-}true} \longrightarrow^* P)$$

$$\frac{\mathsf{intuitionistic}\ P \qquad \mathsf{intuitionistic}\ Q}{\mathsf{intuitionistic}\ (\lambda s.\ P\ s \wedge Q\ s)}$$

$$\frac{\mathsf{intuitionistic}\ P \qquad \mathsf{intuitionistic}\ Q}{\mathsf{intuitionistic}\ (\lambda s.\ P\ s \vee Q\ s)}$$

$$\frac{\bigwedge x.\ \mathsf{intuitionistic}\ (P\ x)}{\mathsf{intuitionistic}\ (\lambda s.\ \forall\, x.\ P\ x\ s)} \qquad \frac{\bigwedge x.\ \mathsf{intuitionistic}\ (P\ x)}{\mathsf{intuitionistic}\ (\lambda s.\ \exists\, x.\ P\ x\ s)}$$

$$\frac{\mathsf{intuitionistic}\ P}{\mathsf{intuitionistic}\ (P \wedge^* Q)} \qquad \frac{\mathsf{intuitionistic}\ Q}{\mathsf{intuitionistic}\ (P \longrightarrow^* Q)}$$

$$\frac{(P \wedge^* \mathsf{sep\text{-}true})\ s \qquad \mathsf{intuitionistic}\ P}{P\ s} \qquad \frac{P\ s \qquad \mathsf{intuitionistic}\ P}{(\mathsf{sep\text{-}true} \longrightarrow^* P)\ s}$$

Table 4.3: Intuitionistic separation assertions.

$$\textsf{strictly-exact}\ P \equiv \forall\, s\ s'.\ P\ s \wedge P\ s' \longrightarrow s = s'$$

$$\frac{\textsf{strictly-exact}\ P \qquad \textsf{strictly-exact}\ Q}{\textsf{strictly-exact}\ (P \wedge^{*} Q)}$$

$$\frac{(Q \wedge^{*} \textsf{sep-true})\ s \qquad P\ s \qquad \textsf{strictly-exact}\ Q}{(Q \wedge^{*} (Q \longrightarrow^{*} P))\ s}$$

Table 4.4: Strictly exact separation assertions.

would provide indirection to it. Fig. 4.5 illustrates related tree and linked-list representation states.

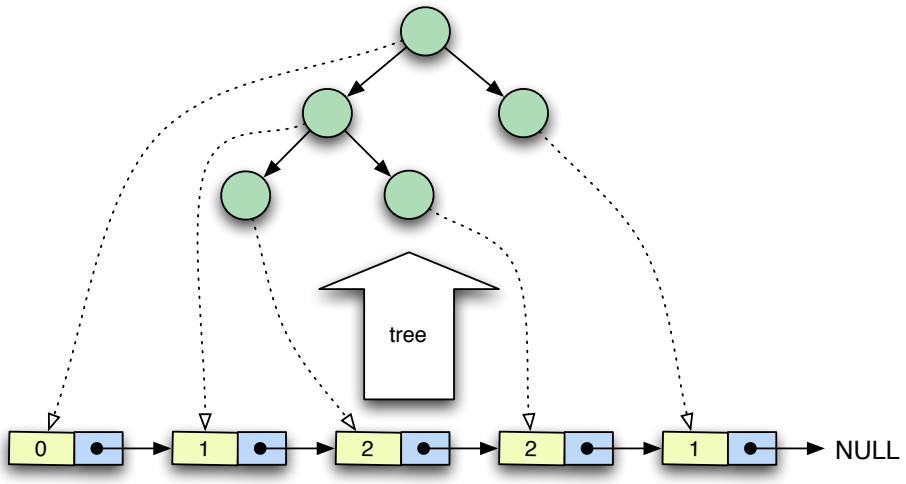

Figure 4.5: Pre-order tree traversal representation and abstraction.

Using these definitions and the rules in Table 4.1, it is possible to prove the following:

$$\frac{(q \mapsto - \wedge^{*} \textsf{tree}\ (\textsf{Node}\ p\ ns)\ p\ c\ d)\ s \qquad (p \hookrightarrow\ v)\ s \qquad d < |word32| - 1}{\textsf{insert-update}\ p\ v\ q\ (\textsf{tree}\ (\textsf{Node}\ p\ (\textsf{Node}\ q\ [] \cdot ns))\ p\ c\ d)\ s}$$

where

$$\textsf{dom-exact}\ P \equiv \forall\, s\ s'.\ P\ s \wedge P\ s' \longrightarrow \textsf{dom}\ s = \textsf{dom}\ s'$$

$$\frac{\textsf{strictly-exact}\ P}{\textsf{dom-exact}\ P} \qquad \textsf{dom-exact}\ (p \mapsto_g v)$$

$$\frac{\textsf{dom-exact}\ (P\ x)}{\textsf{dom-exact}\ (\lambda s.\ \forall x.\ P\ x\ s)}$$

$$\frac{\textsf{dom-exact}\ P \qquad \textsf{dom-exact}\ Q}{\textsf{dom-exact}\ (P \wedge^*\ Q)}$$

$$\frac{(P \wedge^*\ R)\ s \qquad (Q \wedge^*\ R)\ s \qquad \textsf{dom-exact}\ R}{((\lambda s.\ P\ s \wedge Q\ s) \wedge^*\ R)\ s}$$

$$\frac{\forall x.\ (P\ x \wedge^*\ Q)\ s \qquad \textsf{dom-exact}\ Q}{((\lambda s.\ \forall x.\ P\ x\ s) \wedge^*\ Q)\ s}$$

Table 4.5: Domain exact separation assertions.

$$\begin{array}{l}\textsf{insert-update}\ p\ v\ q\ R \equiv \\ q \mapsto - \wedge^* \\ (q \mapsto (\!|\ \textsf{depth} = \textsf{depth}\ v + 1, \textsf{next} = \textsf{next}\ v\ |\!) \longrightarrow^* \\ \ p \mapsto - \wedge^* (p \mapsto v(\!|\textsf{next} := q|\!) \longrightarrow^* R))\end{array}$$

This rule is similar to the proof obligation resulting from a series of updates inserting a new node q under the root p, as the left-most child. Note that the correctness conditions here are subtle — if the abstraction predicate did not require child nodes to be exactly one deeper than their parent it is possible that insertion could cause the new node to collect children of the root. Also, q must not already be part of the tree — this is simply expressed through separation conjunction.

The reasoning in this example was entirely at the level of separation logic and the abstracted algebraic datatype — there was no need to unfold definitions of the separation connectives. We revisit this example again later in Exmp. 4.3.2, where we use the same reasoning to discharge proof obligations resulting from C code, with a specification based on the tree

predicate, that performs the insertion list updates.

4.3 Lifting proof obligations

The verification condition generator applies weakest pre-condition rules to transform Hoare triples to HOL goals that can then be solved by applying theorem prover tactics. In §3.4.2 rewrites were given that could automatically lift the raw heap component of these proof obligations, and in this section rules are provided that allow the low-level applications of lift and heap-update in assertions to be expressed in terms of a separation predicate on the original state. This is desirable as reasoning can then use the derived rules for separation logic, whereas the alternative of unfolding the definitions and working with accesses and updates to the underlying heap state produces a massively more complex goal and proof.

The approach taken here is quite different to the usual separation Hoare logic proof technique employed in the literature, where a new Hoare logic is developed based on separation logic and individual rules are applied at the Hoare logic level. The advantage of our approach is two-fold; we avoid having to manually apply Hoare rules, a task easily automated, and we can take advantage of an existing verification framework and condition generator. On the other hand, there is the disadvantage that applying the rules in this section requires the program verifier to understand the relationship between components of the HOL goals and the original program, since this structure is lost during verification condition generation, and some additional work must be done to transform the proof obligations to the correct form.

We require some supporting lemmas to establish the first separation logic lifting rule.

Lemma 4.3.1. *Validity is preserved by right merges:*

$$\frac{s_1,g \models_s p}{s_0 \mathbin{+\!\!+} s_1,g \models_s p}$$

Proof. Follows from Defn. 3.2.2, Defn. 3.3.4 and proj-d $(s \mathbin{+\!\!+} t)$ = proj-d s $\mathbin{+\!\!+}$ proj-d t. □

Lemma 4.3.2. heap-list-s *for a valid pointer* $p::\alpha::$*c-type ptr is unchanged by right merge:*

$$\frac{s_1,g \models_s p}{\text{heap-list-s } (s_0 \mathbin{+\!\!+} s_1) \text{ (size-of TYPE}(\alpha)) \ p_\& = \text{heap-list-s } s_1 \text{ (size-of TYPE}(\alpha)) \ p_\&}$$

Proof. We first show the stronger statement:

$$\frac{s_1,g \models_s p \qquad n \leq \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)}{\begin{array}{c}\mathsf{heap\text{-}list\text{-}s}\ (s_0 +\!\!+ s_1)\ n\ (p_\& + \mathbb{N}^{\Rightarrow}\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) - n)) = \\ \mathsf{heap\text{-}list\text{-}s}\ s_1\ n\ (p_\& + \mathbb{N}^{\Rightarrow}\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) - n))\end{array}}$$

by induction on n. The base case is trivial and the inductive case can be solved with $\mathsf{proj\text{-}h}\ (s +\!\!+ t) = (\lambda x.\ \mathsf{if}\ x \in \mathsf{dom}\ t\ \mathsf{then}\ \mathsf{proj\text{-}h}\ t\ x\ \mathsf{else}\ \mathsf{proj\text{-}h}\ s\ x)$ and the inductive hypothesis. The proof is completed by instantiation of n with $\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)$. □

Lemma 4.3.3. *A valid mapping given by* lift-typ-heap *is preserved by right merge:*

$$\frac{\mathsf{lift\text{-}typ\text{-}heap}\ g\ s_1\ p = \lfloor v \rfloor}{\mathsf{lift\text{-}typ\text{-}heap}\ g\ (s_0 +\!\!+ s_1)\ p = \lfloor v \rfloor}$$

Proof. By unfolding Defn. 3.3.5, Lem. 4.3.1 and Lem. 4.3.2. □

Lemma 4.3.4. *Separation mapping assertions imply a corresponding value at the lifted underlying heap:*

$$\frac{(p \hookrightarrow_g v)\ s}{\mathsf{lift\text{-}typ\text{-}heap}\ g\ s\ p = \lfloor v \rfloor}$$

Proof. From Defn. 4.2.2, Defn. 4.2.3, Lem. 4.3.3 and Lem. 4.2.1. □

Theorem 4.3.5. lift*s and separation mapping assertions are connected by:*

$$\frac{(p \hookrightarrow_g v)\ (\mathsf{lift\text{-}state}\ (h,\ d))}{\mathsf{lift}\ h\ p = v}$$

Proof. Let $s = \mathsf{lift\text{-}state}\ (h,\ d)$ in Lem. 4.3.4 and apply Thm. 3.4.1 to the goal. Unfold the definition of lift_τ to complete the proof. □

Example 4.3.1. With the Hoare triple resulting from replacing dereferences in the code with their semantic equivalents:

$\{\!|(p \hookrightarrow\ x \wedge^*\ Q)^{sep}|\!\}$
$a = \mathsf{lift}\ (\mathsf{hrs\text{-}mem}\ \mathcal{H})\ p + \mathsf{lift}\ (\mathsf{hrs\text{-}mem}\ \mathcal{H})\ p$
$\{\!|a = 2 * x|\!\}$

the following proof obligation:

$$(p \hookrightarrow\ x \wedge^*\ Q)\ (\mathsf{lift\text{-}state}\ (h,\ d)) \Longrightarrow 2 * \mathsf{lift}\ h\ p = 2 * x$$

requires the value of x in terms of $\mathsf{lift}\ h\ p$ or vice versa. By deriving from the assumption of the goal that $(p \hookrightarrow\ x)\ (\mathsf{lift\text{-}state}\ (h,\ d))$ using the standard rules of separation logic, Thm. 4.3.5 can be applied to solve the goal.

While Thm. 4.3.5 is fine for simplifying the lifts we encounter in goals, it can branch the proof tree before we start reasoning exclusively at the separation logic level. If we want to approach the lifting from a weakest precondition perspective, as done in verification condition generators, we need to express the relationship between lift and separation mapping assertions differently.

Theorem 4.3.6. *The connection in Thm. 4.3.5 can be expressed in a similar form to the Hoare logic backwards reasoning rule for heap accesses [81]:*

$$\frac{\exists\, v.\ (p \mapsto_g v \wedge^* (p \mapsto_g v \longrightarrow^* P\ v))\ (\mathsf{lift\text{-}state}\ (h,\ d))}{P\ (\mathsf{lift}\ h\ p)\ (\mathsf{lift\text{-}state}\ (h,\ d))}$$

Proof. We can use Thm. 4.3.5 on the goal by deriving the separation mapping assertion from the assumption. The proof is completed with [ScSiSame]. □

Heap update dereferences produce proof goals of the form:

$$\begin{array}{c} P\ (\mathsf{lift\text{-}state}\ (h,\ d)) \Longrightarrow \\ Q\ (\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p_0\ v_0\ (\mathsf{heap\text{-}update}\ p_1\ v_1 \\ (\mathsf{heap\text{-}update}\ p_{\ldots}\quad v_{\ldots}\ (\mathsf{heap\text{-}update}\ p_n\ v_n\ h))),d)) \end{array}$$

Definition 4.3.1. To reduce heap-updates to a separation assertion on the original state we first introduce a new predicate for validity at the separation logic level:

$$g \vdash_s p \equiv \lambda s.\ s,g \models_s p \wedge \mathsf{dom}\ s = \{p_{\&}..{+}\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)\}$$

This is related to the idea of the singleton heap predicate $p \mapsto_g -$, but the implication only works in one direction, $(p \mapsto_g v)\ s \Longrightarrow (g \vdash_s p)\ s$, since it is possible to have both $\mathsf{lift}_\tau\ g\ s\ p = \bot$ and a valid footprint at p.

Definition 4.3.2. Also, the concept of the *singleton state* is helpful:

$$\mathsf{singleton}\ p\ v \equiv \mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p\ v\ \mathsf{arbitrary},\ \mathsf{ptr\text{-}tag}\ p\ \mathsf{empty})$$

This is the state whose only valid mapping has the footprint of p and byte encoding of v. It can be shown that $g\ p \Longrightarrow (p \mapsto_g v)\ (\mathsf{singleton}\ p\ v)$.

Lemma 4.3.7. *After the first lifting stage, a* heap-update *at valid pointer p with v is equivalent to the original lifted state right merged with a singleton heap representing v at p:*

$$\frac{d,g \models_t p}{\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d) = \mathsf{lift\text{-}state}\ (h,\ d)\ {+\!+}\ \mathsf{singleton}\ p\ v}$$

Proof. By extensionality, letting the point under consideration be called x. When $x \in \mathsf{dom}\ (\mathsf{singleton}\ p\ v)$, the goal can be simplified to $\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d)\ x = \mathsf{singleton}\ p\ v\ x$. Examining the two state components independently, $\mathsf{heap\text{-}update}\ p\ v\ h\ x = \mathsf{heap\text{-}update}\ p\ v\ \mathsf{arbitrary}\ x$ and $\exists\, t.\ \mathsf{ptr\text{-}tag}\ p\ \mathsf{empty}\ x = \lfloor t \rfloor \wedge d\ x = \lfloor t \rfloor$, based on the observation that $[\![d, g \models_t p;\ x \in \{p_{\&}..+\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)\}]\!] \Longrightarrow \mathsf{ptr\text{-}tag}\ p\ d'\ x = d\ x$ and that the domain of $\mathsf{singleton}\ p\ v$ is the same as the footprint of p.

If $x \notin \mathsf{dom}\ (\mathsf{singleton}\ p\ v)$ then $\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d)\ x = \mathsf{lift\text{-}state}\ (h,\ d)\ x$ based on $x \notin \{p..+|v|\} \Longrightarrow \mathsf{heap\text{-}update\text{-}list}\ p\ v\ h\ x = h\ x$. □

Theorem 4.3.8. *The following rules allow for the reduction of* heap-update*s:*

$$\frac{(g \vdash_s p \wedge^* (p \mapsto_g v \longrightarrow^* P))\ (\mathsf{lift\text{-}state}\ (h,\ d))}{P\ (\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d))}$$

$$\frac{(g \vdash_s p \wedge^* P)\ (\mathsf{lift\text{-}state}\ (h,\ d))}{(p \mapsto_g v \wedge^* P)\ (\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d))}$$

Proof. The first rule can be derived by unfolding the definition of separation conjunction and applying Lem. 4.3.7 to the goal. The assumption partitions the state $\mathsf{lift\text{-}state}\ (h,\ d)$ into two states s_0 and s_1 such that $(g \vdash_s p)\ s_0$ and $(p \mapsto_g v \longrightarrow^* P)\ s_1$. After unfolding the separation implication and instantiating with $\mathsf{singleton}\ p\ v$, we are left with $P\ (s_1 +\!\!+ \mathsf{singleton}\ p\ v)$. Based on the original partitioning, the goal is $P\ (s_1 +\!\!+ s_0 +\!\!+ \mathsf{singleton}\ p\ v)$ and $s_0 +\!\!+ \mathsf{singleton}\ p\ v = \mathsf{singleton}\ p\ v$ as both terms on the LHS have identical domains.

The second rule is derived from the first with [Sci] and [ScSi]. □

These rules are analogous to the backwards and global reasoning Hoare logic mutation rules [81]. The first rule provides a weakest pre-condition style rule that will match any separation assertion, while the second rule may be used on goal assertions that can be manipulated into the matching form.

When the heap type description may also be modified with ptr-tag the resulting goal has the more general form:

$$\begin{array}{c} P\ (\mathsf{lift\text{-}state}\ (h,\ d)) \Longrightarrow \\ Q\ (\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p_0\ v_0\ (\mathsf{heap\text{-}update}\ p_1\ v_1 \\ (\mathsf{heap\text{-}update}\ p_{\ldots}\ \ v_{\ldots}\ (\mathsf{heap\text{-}update}\ p_m\ v_m\ h))), \\ \mathsf{ptr\text{-}tag}\ q_0\ (\mathsf{ptr\text{-}tag}\ q_1\ (\mathsf{ptr\text{-}tag}\ q_{\ldots}\ (\mathsf{ptr\text{-}tag}\ q_n\ d))))) \end{array}$$

We provide a rule for ptr-tag reductions that establishes separation validity from sep-cut, yet another supplemental separation predicate.

Definition 4.3.3. sep-cut asserts that the locations in the *heap-state* domain are the supplied interval, i.e. ownership of a region of memory:

$$\textsf{sep-cut}\ x\ y \equiv \lambda s.\ \textsf{dom}\ s = \{x..{+}y\}$$

Lemma 4.3.9. *Inside the retyped region, lifted heaps do not depend on the original heap type description:*

$$\frac{x \in \{p_{\&}..{+}\textsf{size-of TYPE}(\alpha)\}}{\textsf{lift-state}\ (h,\ \textsf{ptr-tag}\ p\ d)\ x = \textsf{lift-state}\ (h,\ \textsf{ptr-tag}\ p\ \textsf{empty})\ x}$$

where $p{::}\alpha{::}$*mem-type ptr.*

Proof. Based on $x \in \{p_{\&}..{+}\textsf{size-of TYPE}(\alpha)\} \Longrightarrow \textsf{ptr-tag}\ p\ d\ x = \textsf{ptr-tag}\ p\ \textsf{empty}\ x$ which can be derived from Defn. 3.2.4. □

Corollary. *Lifted heaps restricted to the retyped region do not depend on the original heap type description:*

$$\textsf{lift-state}\ (h,\ \textsf{ptr-tag}\ p\ d)\restriction_{\{p_{\&}..{+}\textsf{size-of TYPE}(\alpha)\}} = \textsf{lift-state}\ (h,\ \textsf{ptr-tag}\ p\ \textsf{empty})$$

Lemma 4.3.10. *Outside the retyped region, the lifted heap remains unchanged:*

$$\frac{x \notin \{p_{\&}..{+}\textsf{size-of TYPE}(\alpha)\}}{\textsf{lift-state}\ (h,\ \textsf{ptr-tag}\ p\ d)\ x = \textsf{lift-state}\ (h,\ d)\ x}$$

where $p{::}\alpha{::}$*mem-type ptr.*

Proof. Based on $x \notin \{p_{\&}..{+}\textsf{size-of TYPE}(\alpha)\} \Longrightarrow \textsf{ptr-tag}\ p\ d\ x = d\ x$ which can be derived from Defn. 3.2.4. □

Theorem 4.3.11. *Retyping yields separation validity in a disjoint region of the heap corresponding to the target footprint:*

$$\frac{(\textsf{sep-cut}\ p_{\&}\ (\textsf{size-of TYPE}(\alpha))\ \wedge^*\ P)\ (\textsf{lift-state}\ (h,\ d)) \qquad g\ p}{(g \vdash_s p\ \wedge^*\ P)\ (\textsf{lift-state}\ (h,\ \textsf{ptr-tag}\ p\ d))}$$

$$\frac{(\textsf{sep-cut}\ p_{\&}\ (\textsf{size-of TYPE}(\alpha))\ \wedge^*\ (g \vdash_s p \longrightarrow^*\ P))\ (\textsf{lift-state}\ (h,\ d)) \qquad g\ p}{P\ (\textsf{lift-state}\ (h,\ \textsf{ptr-tag}\ p\ d))}$$

Proof. Partition $\textsf{lift-state}\ (h,\ d)$ as s_0 and s_1 where $\textsf{dom}\ s_0 = \{p_{\&}..{+}\textsf{size-of TYPE}(\alpha)\}$ and $P\ s_1$. It can be shown that the goal state $\textsf{lift-state}\ (h,\ \textsf{ptr-tag}\ p\ d)$ is equivalent to $s_1 +\!\!+ \textsf{lift-state}\ (h,\ \textsf{ptr-tag}\ p\ d)\restriction_{\textsf{dom}\ s_0}$ by considering both states at x, case splitting on $x \in \textsf{dom}\ s_0$ and Lem. 4.3.10. The corollary of Lem. 4.3.9 leads to $(g \vdash_s p)\ (\textsf{lift-state}\ (h,\ \textsf{ptr-tag}\ p\ d)\restriction_{\textsf{dom}\ s_0})$ holding, hence there is a partition for the goal state on which the conjuncts hold independently, allowing them to be combined with separation conjunction.

The weakest pre-condition style rule follows from this by [ScSiSame] instantiated with $g \vdash_s p$ and P. □

The shallow embedding gives the flexibility to add separation predicates that express information about a region of memory at different levels of abstraction, from $p \mapsto_g v$ to sep-cut. Once the above reduction rules are applied the reasoning can continue using the standard rules of separation logic without requiring additional proof goals or side-conditions.

In deriving these rules we have primarily been concerned with soundness, which was required prior to Isabelle admitting the rules. The focus in this thesis on real-world verification examples has resulted in less attention being paid to the completeness of the system, particularly as it is established that one can always drop down to the lower more concrete levels if required. However, completeness results have been explored elsewhere in the literature [45, 105].

Example 4.3.2. The insert operation in Exmp. 4.2.1 can be specified and implemented in C as in Table 4.6. The single proof obligation is first lifted to the separation logic level by applying the WP-style rules in Thm. 4.3.6, Thm. 4.3.8 and Thm. 4.3.11. The proof then essentially[3] follows from the rules in Table 4.1, $(p \mapsto_g v)\ s \Longrightarrow (g \vdash_s p)\ s$ and some bit-vector arithmetic reasoning.

$\forall\, ns\ c\ d.$
$\quad \{\!|$(sep-cut $'q_{\&}$ (size-of TYPE($node$)) $\wedge^*$ tree (Node $'p$ ns) $'p$ c d)^{sep} $\wedge$
$\quad\ $c-guard $'q \wedge d < |word32| - 1$ $|\!\}$
$\quad insert\text{-}node('q, 'p)$
$\quad \{\!|$(tree (Node $'p$ (Node $'q$ $[]{\cdot}ns$)) $'p$ c d)^{sep} $|\!\}$

```
void insert_node(struct node *q, struct node *p)
{
    /** AUXUPD: (λx. True, ptr-tag 'q) */
    q->depth = p->depth + 1;
    q->next = p->next;
    p->next = q;
}
```

Table 4.6: **insert_node** specification and definition.

[3] This example uses an earlier version of the C_{sys}-*com* translation that treats field dereferences as entire **struct** object accesses, since we do not yet have the development of Chapter 5. As a result, we also had to show the existence of a singleton mapping assertion implied by validity post-retyping, which works for **struct node**s as from-bytes is total for this type.

4.4 Frame rule

4.4.1 Globalised specifications

The separation frame rule [105] is often seen as the key to the scalability of the separation logic approach to verification. It allows for deriving a global specification from a local specification of a program's behaviour, with an arbitrary conjoined separation assertion on a part of the heap preserved by the program. That is, one can verify a function working in one region of the heap and then utilise its specification in the context of a function operating on a superset of the region.

The frame rule has the form:

$$\frac{\{\!|P^{sep}|\!\}\ c\ \{\!|Q^{sep}|\!\}}{\{\!|(P\ \wedge^*\ R)^{sep}|\!\}\ c\ \{\!|(Q\ \wedge^*\ R)^{sep}|\!\}}$$

Note that this is a Hoare logic rule, that can be applied to a specification triple prior to verification condition generation, as opposed to the rules in §4.3 that are applied after. If R is universally quantified in the globalised specification, it can be instantiated as required when solving the proof obligations of the calling function.

Unfortunately, such a general rule cannot be expressed in a shallow embedding since:

- The state-space type is program dependent. The type of programs in the verification environment is $(\sigma, \eta, \chi)\ com$, where σ is the state space. lift-state requires two variables from this as an argument in a $(\ldots)^{sep}$ expression, the heap memory and heap type description functions. The output from C translation names these *t-hrs*, but when writing this rule on a σ state space this information is unavailable.

- c is an arbitrary program in the underlying verification framework for which this rule may not be true. Since there are no restrictions on heap updates other than those imposed by the guard mechanism, the following triple holds:

  $\{\!|\ \square^{sep}\ |\!\}\ *p = 0;\ \{\!|\ \square^{sep}\ |\!\}$

  However, after applying the frame rule with $R = {'p} \mapsto v$, the triple below does not:

  $\{\!|\ (\square\ \wedge^*\ ({'p} \mapsto v))^{sep}\ |\!\}\ *p = 0;\ \{\!|\ (\square\ \wedge^*\ ({'p} \mapsto v))^{sep}\ |\!\}$

  The problem here is that the frame rule depends on a specific notion of memory safety. In other developments of separation logic, it is a requirement that heap locations can only be modified if they are described in the pre-condition of the specification. This is backed by a semantic restriction on updates, where the state includes information on which parts of the heap are acceptable to update.

> Such a restriction would be contrary to ROLM — memory safety in the framework presented in this thesis is enforced by the guard mechanism and is far coarser, e.g. a guard may only require that all updates occur within a 1MB region representing the heap in a system.

It is however possible to prove this rule for specific programs and state-spaces and below we explain how this can be automated and made generic for C_{sys}.

4.4.2 Heap-state type class

To solve the first problem, we make further use of type classes to define α::*heap-state-type*, which provides access and update functions for the heap state and heap type description.

Definition 4.4.1. The α::*heap-state-type* class provides constants hst-mem, hst-mem-update, hst-htd and hst-htd-update giving the access and update functions for the heap state and type description components of the state respectively. The class is constrained by the following axioms:

$$\begin{aligned}
&\textsf{hst-htd}\ (\textsf{hst-htd-update}\ d\ s) &&\equiv\ d\ (\textsf{hst-htd}\ s)\\
&\textsf{hst-mem}\ (\textsf{hst-mem-update}\ h\ s) &&\equiv\ h\ (\textsf{hst-mem}\ s)\\
&\textsf{hst-mem}\ (\textsf{hst-htd-update}\ d\ s) &&\equiv\ \textsf{hst-mem}\ s
\end{aligned}$$

where h::*heap-mem* $\Rightarrow$ *heap-mem* and d::*heap-typ-desc* $\Rightarrow$ *heap-typ-desc* are state transformers for their respective components of the heap state.

This is sufficient to reason over the state spaces of all target programs of interest in the following sections. A concrete program's state space is instantiated as a member of this type class by defining the access and update functions, applying the standard Isabelle/HOL tactics for class instantiation and then simplifying the resulting proof obligations using these definitions. This is possible as the state record representation already includes concrete rewrites analogous to the axioms of α::*heap-state-type*.

The $(\ldots)^{sep}$ syntax is extended to assertions on lifted α::*heap-state-type* states using the following definition:

$$\textsf{lift-hst}\ s \equiv \textsf{lift-state}\ (\textsf{hst-mem}\ s, \textsf{hst-htd}\ s)$$

4.4.3 Memory safety

The frame rule can then be expressed for programs c with a state space in α::*heap-state-type* as:

$$\frac{\forall \sigma.\ \{\!|\sigma.\ P^{sep}|\!\}\ c\ \{\!|Q^{sep}|\!\} \qquad \textsf{mem-safe}\ c\ \Gamma}{\forall \sigma.\ \{\!|\sigma.\ (P\ \wedge^*\ R)^{sep}|\!\}\ c\ \{\!|(Q\ \wedge^*\ R)^{sep}|\!\}}$$

The mem-safe c Γ[4] assumption addresses the second problem by requiring programs on which one wishes to apply the frame rule to provide a form of memory safety. Such programs generate a guard failure if either:

- The program modifies the heap state or heap type description outside of the initial domain of the heap type description.
- The program depends on the heap type description outside this domain in any expression. The program is still free to depend on the heap memory state outside the domain of the heap type description.

These conditions are not met by the normal output of the C translation stage, since guards are only generated to prevent undefined behaviour as the C_{sys} semantics understands it. Here the verifier optionally enables additional memory safety guard generation, and consequently imposes a slightly higher proof effort, to gain a property — if the frame rule is not required in a verification, the framework allows these guards to be suppressed.

Definition 4.4.2. Memory safety with respect to the heap type description is defined:

$$\textsf{mem-safe}\ C\ \Gamma \equiv \forall\, s\ t.\ \Gamma\vdash \langle C,\textsf{Normal}\ s\rangle \Rightarrow t \longrightarrow \textsf{restrict-safe}\ s\ t\ C\ \Gamma$$

This definition has some unfamiliar concepts. On the LHS of the implication, we have a big-step semantics predicate from the verification environment, that asserts that program C, in an initially normal state s, with environment Γ, leads to state t. This then implies:

$$\begin{array}{l}\textsf{restrict-safe}\ s\ t\ C\ \Gamma \equiv \\ \forall\, X.\ (\textsf{case}\ t\ \textsf{of Normal}\ t' \Rightarrow \Gamma \vdash \langle C,s\rangle \Rightarrow_X \textsf{Normal},t' \\ \quad \mid \textsf{Abrupt}\ t' \Rightarrow \Gamma \vdash \langle C,s\rangle \Rightarrow_X \textsf{Abrupt},t' \mid \text{-} \Rightarrow \textsf{False}) \vee \\ \quad \textsf{exec-fatal}\ C\ \Gamma\ (s\ {}^{d}|_X)\end{array}$$

A program's execution can end in normal or abrupt (used for the exception mechanism) termination, or in some failure state that would prohibit us from proving anything about the program. The intuition behind this definition is that, given a computation leading from state s to t, restricting the domain of the heap type description in s to X should either result in t with a similarly restricted heap type description, or a failure state. That is, the program does not modify or depend on state outside the heap type description domain without triggering a guard.

[4] Γ is a component of the verification environment's state that we have so far hidden as it has not had any direct relevance. It provides a map from procedure names to bodies, i.e. $\eta \rightharpoonup (\sigma,\, \eta,\, \chi)\ com$.

The $\Gamma \vdash \langle C,s\rangle \Rightarrow_X f,t$ predicate is similar to the big-step semantics, but restricts the heap type description domain prior to execution and then ensures the resulting state has a similarly restricted heap type description and that the state components are unchanged by execution outside of X. It is parametrised by f which can be either the constructor Normal or Abrupt:

$$\begin{array}{l}\Gamma \vdash \langle C,s\rangle \Rightarrow_X f,t \equiv \\ \Gamma\vdash \langle C,\mathsf{Normal}\ (s\ {}^{d}|_X)\rangle \Rightarrow f\ (t\ {}^{d}|_X) \wedge \\ \mathsf{hst\text{-}mem}\ t =^{-X} \mathsf{hst\text{-}mem}\ s \wedge \mathsf{hst\text{-}htd}\ t =^{-X} \mathsf{hst\text{-}htd}\ s\end{array}$$

where

$$\begin{array}{lcl} s\ {}^{d}|_X & \equiv & s(\!|\mathsf{hst\text{-}htd} := \mathsf{hst\text{-}htd}\ s\restriction_X|\!) \\ f =^{-X} g & \equiv & \forall x.\ x \notin X \longrightarrow f\ x = g\ x \end{array}$$

An execution leading to failure can be expressed in the verification environment as[5]:

$$\begin{array}{l}\mathsf{exec\text{-}fatal}\ C\ \Gamma\ s \equiv \\ (\exists f.\ \Gamma\vdash \langle C,\mathsf{Normal}\ s\rangle \Rightarrow \mathsf{Fault}\ f) \vee \Gamma\vdash \langle C,\mathsf{Normal}\ s\rangle \Rightarrow \mathsf{Stuck}\end{array}$$

This notion of memory safety is motivated by the introduced memory safety guards' effect on program semantics. The C translation inserts additional guards for memory safety; for a heap update dereference asserting that the lvalue's heap footprint is contained entirely in the heap type description's domain. *AUXUPD* annotations also contain a guard for this purpose, which should prevent heap type description updates or dependencies outside the current domain. The ptr-safe guard (Defn. 3.2.3) is used to achieve this. An example guarded expression statement is:

$$\mathsf{Guard\ MemSafe}\ \{\!|\ \mathsf{ptr\text{-}safe}\ {}^{\backprime}p\ \mathcal{D}\ |\!\}\ (\mathsf{Basic}\ (\lambda s.\ \mathsf{heap\text{-}update}\ {}^{s}p\ v\ (\mathsf{hrs\text{-}mem}\ {}^{s}\mathcal{H})))$$

It should be noted that we do not require type safety, nor do we require guards for most expressions as only heap type description updates may depend on the heap type description, i.e. the output of the C translation stage does not feature heap type description accesses except in such updates.

Once the guards are inserted, it is then necessary to perform some automatic analysis to provide the link between the augmented program object and Defn. 4.4.2. We consider first the intra-procedural case, then explain how this can be used in the presence of function calls.

[5]The Stuck state does not model non-termination, but instead the inability for forward progression of a computation, e.g. the invocation of a non-existent procedure.

Intra-procedural analysis

A primitive recursive algorithm that performs syntactic decomposition at the statement level and checks expression properties using only rewriting is supplied in Table 4.7. By using the intra-safe conditional rewrites the *com* object output of the C translation stage can be automatically shown to posses the mem-safe property.

$$
\begin{array}{lcl}
\textsf{intra-safe Skip} & \equiv & \textsf{True} \\
\textsf{intra-safe (Basic } f\textsf{)} & \equiv & \textsf{comm-restrict-safe } \mathcal{U}\ f\ \wedge \\
 & & \textsf{point-eq-mod-safe } \mathcal{U}\ f\ \textsf{hst-mem}\ \wedge \\
 & & \textsf{point-eq-mod-safe } \mathcal{U}\ f\ \textsf{hst-htd} \\
\textsf{intra-safe (Seq } C\ D\textsf{)} & \equiv & \textsf{intra-safe } C\ \wedge\ \textsf{intra-safe } D \\
\textsf{intra-safe (Cond } P\ C\ D\textsf{)} & \equiv & \textsf{expr-htd-ind } P\ \wedge\ \textsf{intra-safe } C\ \wedge\ \textsf{intra-safe } D \\
\textsf{intra-safe (While } P\ C\textsf{)} & \equiv & \textsf{expr-htd-ind } P\ \wedge\ \textsf{intra-safe } C \\
\textsf{intra-safe (Call } p\textsf{)} & \equiv & \textsf{True} \\
\textsf{intra-safe (DynCom } f\textsf{)} & \equiv & \textsf{fun-htd-ind } f\ \wedge\ (\forall s.\ \textsf{intra-safe } (f\ s)) \\
\textsf{intra-safe (Guard } f\ G\ C\textsf{)} & \equiv & \textsf{mono-guard } G\ \wedge \\
 & & (\textsf{case } C\ \textsf{of} \\
 & & \ \textsf{Basic } g \Rightarrow \\
 & & \ \ \textsf{comm-restrict-safe } G\ g\ \wedge \\
 & & \ \ \textsf{point-eq-mod-safe } G\ g\ \textsf{hst-mem}\ \wedge \\
 & & \ \ \textsf{point-eq-mod-safe } G\ g\ \textsf{hst-htd} \\
 & & \ |\ \text{-} \Rightarrow \textsf{intra-safe } C) \\
\textsf{intra-safe Throw} & \equiv & \textsf{True} \\
\textsf{intra-safe (Catch } C\ D\textsf{)} & \equiv & \textsf{intra-safe } C\ \wedge\ \textsf{intra-safe } D
\end{array}
$$

where

$$
\begin{array}{lcl}
\textsf{point-eq-mod-safe } P\ f\ g & \equiv & \forall s\ X.\ s\ {}^{d}|_{X} \in P \longrightarrow g\ (f\ s) =^{-X} g\ s \\
\textsf{comm-restrict } f\ s\ X & \equiv & f\ (s\ {}^{d}|_{X}) = f\ s\ {}^{d}|_{X} \\
\textsf{comm-restrict-safe } P\ f & \equiv & \forall s\ X.\ s\ {}^{d}|_{X} \in P \longrightarrow \textsf{comm-restrict } f\ s\ X \\
\textsf{mono-guard } G & \equiv & \forall s\ X.\ s\ {}^{d}|_{X} \in G \longrightarrow s \in G \\
\textsf{fun-htd-ind } f & \equiv & \forall d\ s.\ f\ (\textsf{hst-htd-update } d\ s) = f\ s \\
\textsf{expr-htd-ind } P & \equiv & \forall d\ s.\ (s(\!|\textsf{hst-htd} := d|\!) \in P) = (s \in P)
\end{array}
$$

Table 4.7: Intra-procedural rewrites.

In this section we focus on the correctness of the intra-safe analysis and how it may be applied on a single procedure, assuming this check has been completed for all other procedures in the environment. In the next section we examine how this assumption can be efficiently discharged.

Theorem 4.4.1. *A procedure C is* mem-safe *if* intra-safe *holds for C and all procedures in the environment:*

$$\frac{\textsf{intra-safe}\ C \qquad \bigwedge n\ C.\ \dfrac{\Gamma\ n = \lfloor C \rfloor}{\textsf{intra-safe}\ C}}{\textsf{mem-safe}\ C\ \Gamma}$$

Proof. The proof is via induction on the big-step semantic relation given by the definition of mem-safe, i.e.:

$$\frac{\Gamma\vdash \langle C,\textsf{Normal}\ s\rangle \Rightarrow t \qquad \textsf{intra-safe}\ C \qquad \bigwedge n\ C.\ \dfrac{\Gamma\ n = \lfloor C \rfloor}{\textsf{intra-safe}\ C}}{\textsf{restrict-safe}\ s\ t\ C\ \Gamma}$$

We sketch the non-trivial base and inductive cases here[6]:

- Guard $C\ f\ G\ s\ t$ — this is the case where $s \in G$. If $\exists\, g.\ C = \textsf{Basic}\ g$ then the goal follows from considering the cases of t. Only the Normal case is possible, as primitive Basic steps do not fail in the semantics, and is handled by observing the following from definitions:

$$\textsf{exec-fatal}\ (\textsf{Guard}\ f\ G\ C)\ \Gamma\ s = (s \in G \longrightarrow \textsf{exec-fatal}\ C\ \Gamma\ s)$$

$$\frac{\textsf{comm-restrict-safe}\ G\ f \qquad s\ ^d|_X \in G}{f\ s\ ^d|_X = f\ (s\ ^d|_X)}$$

$$\frac{\textsf{point-eq-mod-safe}\ G\ f\ g \qquad s\ ^d|_X \in G \qquad x \notin X}{g\ (f\ s)\ x = g\ s\ x}$$

  The intuition behind this aspect of the intra-safe definition and proof is that changes to the heap memory state and heap type description only take place when correctly guarded.

  Otherwise, use the inductive hypothesis and the rule:

$$\frac{\textsf{restrict-safe}\ s\ t\ C\ \Gamma}{\textsf{restrict-safe}\ s\ t\ (\textsf{Guard}\ f\ G\ C)\ \Gamma}$$

- GuardFault $C\ f\ G\ s$ — here $s \notin G$ and $t = \textsf{Fault}\ f$. Guard monotonicity gives:

$$\frac{s \notin G \qquad \textsf{mono-guard}\ G}{\textsf{restrict-safe}\ s\ (\textsf{Fault}\ f)\ (\textsf{Guard}\ f\ G\ C)\ \Gamma}$$

[6] We refer the interested reader to the big-step semantics relation of *com* [85] for a complete explanation of the cases.

- Basic $f\ s$ — this is similar to the first case of Guard $C\ f\ G\ s\ t$, except that the guard does not allow the initial states considered to be restricted, hence we require the safety predicates to hold on all states. This prevents unguarded Basic statements from modifying the heap state or type description and from depending on the heap type description at all.

- Seq $C\ D\ s\ s'\ t$ — given by case splitting on s', inductive hypothesis and transitivity of $f =^{-X} g$.

- CondTrue $P\ C\ D\ s\ t$, CondFalse $P\ C\ D\ s\ t$, WhileTrue $P\ C\ s\ s'\ t$, WhileFalse $P\ C\ s$, DynCom $f\ s\ t$ — since the expression is independent of the heap type description, control flow is not influenced by restriction. Hence these cases can be discharged with the inductive hypothesis.

- Throw s, CatchMatch $C\ D\ s\ s'\ t$, CatchMiss $C\ D\ s\ t$ — no dependency on any aspect of the program state in the control flow here, so as above.

- Call $C\ p\ s\ t$ — by the environment assumption.

□

Table 4.8 provides a set of conditional rewrites for discharging the side-conditions resulting from intra-safe evaluation. They follow from definitions, with the rules relating to ptr-tags requiring some additional reasoning about footprint intervals. The rules are presented in a format designed to take advantage of the simplifier's pattern matching — a concrete heap access function will not immediately match with hst-mem for example, but the equality can be established during rewriting of the assumption using the definition of hst-mem which is placed in the default simplifier set.

Inter-procedural analysis

Thm. 4.4.1 expresses sufficient syntactic conditions for the frame rule to hold. It requires all procedures in Γ to have the intra-safe property. This is not as practical as we would like, as it should not be the case that anything needs to be proved about procedures that are not reachable from C in the call graph. Also, the verification environment provides definitions for Γ at each procedure present using Isabelle's locale mechanism, but does not provide a full definition for Γ. In this section we remedy this problem by providing a means to restrict the intra-safe proof obligation to only the procedures reachable from C in Γ.

Table 4.9 presents an inductive definition proc-deps $C\ \Gamma$ that gives the names of reachable procedures.

$$\frac{\bigwedge s\ X.\ (s(\!|\mathsf{hst\text{-}htd} := X|\!) \in G) = (s \in G)}{\mathsf{mono\text{-}guard}\ G}$$

$$\frac{\bigwedge s.\ d\ s = \mathsf{hst\text{-}htd}\ s \qquad \mathsf{fun\text{-}htd\text{-}ind}\ p}{\mathsf{mono\text{-}guard}\ \{s \mid \mathsf{ptr\text{-}safe}\ (p\ s)\ (d\ s)\}}$$

$$\frac{\bigwedge s.\ g\ (f\ s) = g\ s}{\mathsf{point\text{-}eq\text{-}mod\text{-}safe}\ P\ f\ g}$$

$$\frac{\begin{array}{c} d = \mathsf{hst\text{-}htd} \qquad f = (\lambda s.\ \mathsf{hst\text{-}mem\text{-}update}\ (\mathsf{heap\text{-}update}\ (p\ s)\ (v\ s))\ s) \\ h = \mathsf{hst\text{-}mem} \qquad \mathsf{fun\text{-}htd\text{-}ind}\ p \end{array}}{\mathsf{point\text{-}eq\text{-}mod\text{-}safe}\ \{s \mid \mathsf{ptr\text{-}safe}\ (p\ s)\ (d\ s)\}\ f\ h}$$

$$\frac{\begin{array}{c} d = \mathsf{hst\text{-}htd} \\ d' = \mathsf{hst\text{-}htd} \qquad f = (\lambda s.\ \mathsf{hst\text{-}htd\text{-}update}\ (\mathsf{ptr\text{-}tag}\ (p\ s))\ s) \qquad \mathsf{fun\text{-}htd\text{-}ind}\ p \end{array}}{\mathsf{point\text{-}eq\text{-}mod\text{-}safe}\ \{s \mid \mathsf{ptr\text{-}safe}\ (p\ s)\ (d\ s)\}\ f\ d'}$$

$$\frac{\bigwedge s\ X.\ f\ (s(\!|\mathsf{hst\text{-}htd} := \mathsf{hst\text{-}htd}\ s\restriction_X|\!)) = f\ s(\!|\mathsf{hst\text{-}htd} := \mathsf{hst\text{-}htd}\ (f\ s)\restriction_X|\!)}{\mathsf{comm\text{-}restrict\text{-}safe}\ P\ f}$$

$$\frac{\begin{array}{c} d = \mathsf{hst\text{-}htd} \qquad f = (\lambda s.\ \mathsf{hst\text{-}htd\text{-}update}\ (\mathsf{ptr\text{-}tag}\ (p\ s))\ s) \qquad \mathsf{fun\text{-}htd\text{-}ind}\ p \\ \bigwedge d\ d'\ s.\ \mathsf{hst\text{-}htd\text{-}update}\ (d\ s)\ (\mathsf{hst\text{-}htd\text{-}update}\ (d'\ s)\ s) \\ = \mathsf{hst\text{-}htd\text{-}update}\ ((d\ s) \circ (d'\ s))\ s \end{array}}{\mathsf{comm\text{-}restrict\text{-}safe}\ \{s \mid \mathsf{ptr\text{-}safe}\ (p\ s)\ (d\ s)\}\ f}$$

Table 4.8: Intra-procedural side-condition conditional rewrites.

Lemma 4.4.2. *A procedure p may be safely removed from the environment during calculation of* proc-deps *providing its reachable procedures are considered independently and merged with the result.*

$$\mathsf{proc\text{-}deps}\ C\ \Gamma \subseteq \mathsf{proc\text{-}deps}\ C\ (\Gamma(p := \bot)) \cup \mathsf{proc\text{-}deps}\ (\mathsf{Call}\ p)\ \Gamma$$

Proof. The proof is by induction on the proc-deps definition from the LHS. The base case is straightforward. In the inductive case we require that for all x and y where $x \in \mathsf{proc\text{-}deps}\ C\ \Gamma$, $\Gamma\ x = \lfloor D \rfloor$ and $y \in \mathsf{intra\text{-}deps}\ D$, that $\forall\, p.\ y \in \mathsf{proc\text{-}deps}\ C\ (\Gamma(p := \bot)) \cup \mathsf{proc\text{-}deps}\ (\mathsf{Call}\ p)\ \Gamma$. If we fix p and assume $y \notin \mathsf{proc\text{-}deps}\ (\mathsf{Call}\ p)\ \Gamma$ then all that needs to be shown is that $y \in \mathsf{proc\text{-}deps}\ C\ (\Gamma(p := \bot))$. This can be achieved by considering the cases for $x = p$, application of the proc-deps introduction rules and the inductive hypothesis. □

$$\frac{x \in \textsf{intra-deps}\ C}{x \in \textsf{proc-deps}\ C\ \Gamma}$$

$$\frac{x \in \textsf{proc-deps}\ C\ \Gamma \qquad \Gamma\ x = \lfloor D \rfloor \qquad y \in \textsf{intra-deps}\ D}{y \in \textsf{proc-deps}\ C\ \Gamma}$$

where

$$\begin{array}{ll}
\textsf{intra-deps Skip} & \equiv\ \emptyset \\
\textsf{intra-deps (Basic } f\textsf{)} & \equiv\ \emptyset \\
\textsf{intra-deps (Seq } C\ D\textsf{)} & \equiv\ \textsf{intra-deps } C \cup \textsf{intra-deps } D \\
\textsf{intra-deps (Cond } P\ C\ D\textsf{)} & \equiv\ \textsf{intra-deps } C \cup \textsf{intra-deps } D \\
\textsf{intra-deps (While } P\ C\textsf{)} & \equiv\ \textsf{intra-deps } C \\
\textsf{intra-deps (Call } p\textsf{)} & \equiv\ \{p\} \\
\textsf{intra-deps (DynCom } f\textsf{)} & \equiv\ \bigcup\{\textsf{intra-deps } (f\ s) \mid \textsf{True}\} \\
\textsf{intra-deps (Guard } f\ G\ C\textsf{)} & \equiv\ \textsf{intra-deps } C \\
\textsf{intra-deps Throw} & \equiv\ \emptyset \\
\textsf{intra-deps (Catch } C\ D\textsf{)} & \equiv\ \textsf{intra-deps } C \cup \textsf{intra-deps } D
\end{array}$$

Table 4.9: Inter-procedural dependency definition.

Theorem 4.4.3. *It is valid to restrict the environment to only the reachable procedures as given by* proc-deps *when establishing memory safety:*

$$\textsf{mem-safe}\ C\ \Gamma = \textsf{mem-safe}\ C\ (\Gamma\restriction_{\textsf{proc-deps}\ C\ \Gamma})$$

Proof. Induct on the big-step semantics relation to give

$$\frac{\Gamma\restriction_X \vdash \langle C,s\rangle \Rightarrow t \qquad \textsf{proc-deps}\ C\ \Gamma \subseteq X}{\Gamma \vdash \langle C,s\rangle \Rightarrow t}$$

and

$$\frac{\Gamma \vdash \langle C,s\rangle \Rightarrow t \qquad \textsf{proc-deps}\ C\ \Gamma \subseteq X}{\Gamma\restriction_X \vdash \langle C,s\rangle \Rightarrow t}$$

hence

$$\Gamma\restriction_{\textsf{proc-deps}\ C\ \Gamma} \vdash \langle C,s\rangle \Rightarrow t = \Gamma \vdash \langle C,s\rangle \Rightarrow t$$

Most of the cases are trivial and can be solved with the appropriate introduction rules, however $\textsf{Call}\ C\ p\ s\ t$ requires Lem. 4.4.2.

The proof then follows from Defn. 4.4.2. □

So far it is possible to restrict the environment to the proc-deps set. To use this in a proof requires calculation of this inductively-defined set, which cannot be automated directly from such a definition. Instead rewrites are derived, which look mostly like those for intra-deps in Table 4.9 with the exception of:

$$\begin{array}{c}\textsf{proc-deps}\ (\textsf{Call}\ p)\ \Gamma = \\ \{p\} \cup (\textsf{case}\ \Gamma\ p\ \textsf{of}\ \bot \Rightarrow \emptyset \mid \lfloor C \rfloor \Rightarrow \textsf{proc-deps}\ C\ (\Gamma(p := \bot)))\end{array}$$

It is evident that the analysis in this section is conservative. A more restricted set of procedures could be obtained at the expense of having to consider program semantics and perform manual proofs. This is not desirable as we would like the frame rule to be cheap to apply. In cases where the frame rule is needed, it seems reasonable to require that all reachable procedures in the call graph are proven safe.

4.4.4 Soundness

The above presentation of the frame rule is somewhat simplified in that P, Q and R are not functions of the local variable state. Here we derive the more general rule where they are and R does not depend on variables modified by C. In the below presentation of the frame rule, a separation assertion takes as a parameter a function applied to the current state, representing the extraction of local variables. A fun-htd-ind[7] side-condition is then introduced for the function to prevent it depending on the global heap type description state.

Theorem 4.4.4. *The frame rule is sound:*

$$\frac{\begin{array}{c}\forall \sigma.\ \{s \mid \sigma = s \wedge (P\ (f\ s))^{sep}\}\ C\ \{s \mid (Q\ (g\ \sigma\ s))^{sep}\} \\ \textsf{fun-htd-ind}\ f \qquad \textsf{fun-htd-ind}\ g \qquad \forall s.\ \textsf{fun-htd-ind}\ (g\ s) \qquad \textsf{mem-safe}\ C\ \Gamma\end{array}}{\forall \sigma.\ \{s \mid \sigma = s \wedge (P\ (f\ s) \wedge^*\ R\ (h\ s))^{sep}\}\ C\ \{s \mid (Q\ (g\ \sigma\ s) \wedge^*\ R\ (h\ \sigma))^{sep}\}}$$

Proof. By using the Hoare logic completeness result, the goal can be shifted to the semantic level:

$$\frac{\begin{array}{c}\Gamma \vdash \langle C, \textsf{Normal}\ s\rangle \Rightarrow t \\ (P\ (f\ s) \wedge^*\ R\ (h\ s))\ (\textsf{lift-hst}\ s) \qquad t \notin \textsf{Fault}\ `\ \emptyset \qquad t \notin \textsf{Abrupt}\ `\ \emptyset\end{array}}{t \in \textsf{Normal}\ `\ \{s' \mid (Q\ (g\ s\ s') \wedge^*\ R\ (h\ s))\ (\textsf{lift-hst}\ s')\}}$$

where s is the pre-state and t the post-state under consideration.

From the separation conjunction in the assumptions, we can obtain s_0 and s_1 where $P\ (f\ s)\ s_0$, $R\ (h\ s)\ s_1$, $s_0\ \bot\ s_1$ and $\textsf{lift-hst}\ s = s_1\ +\!\!+\ s_0$.

The proof is completed by case splitting on t:

- $\textsf{Normal}\ t'$ — Using the Hoare soundness result and original specification assumption, it is easy to see that $\neg\ \textsf{exec-fatal}\ C\ \Gamma\ (s\ {}^{d}|_{\textsf{dom}}\ s_0)$, as if the program does terminate, the Hoare triple semantics gives that it is in a non-fatal post-state. Using this, the mem-safe assumption and definition and the execution assumption in the above intermediate goal we can conclude the following:

[7] See Table 4.7 for definition.

$$\Gamma \vdash \langle C, \mathsf{Normal}\ (s\ ^{d}|_{\mathsf{dom}\ s_0})\rangle \Rightarrow \mathsf{Normal}\ (t'\ ^{d}|_{\mathsf{dom}\ s_0})$$
$$\mathsf{hst\text{-}mem}\ t' =^{-\mathsf{dom}\ s_0} \mathsf{hst\text{-}mem}\ s$$
$$\mathsf{hst\text{-}htd}\ t' =^{-\mathsf{dom}\ s_0} \mathsf{hst\text{-}htd}\ s$$

From soundness, $Q\ (g\ s\ (t'\ ^{d}|_{\mathsf{dom}\ s_0}))\ (\mathsf{lift\text{-}hst}\ (t'\ ^{d}|_{\mathsf{dom}\ s_0}))$ and since $s_1 = \mathsf{lift\text{-}hst}\ t'{\uparrow}_{(\mathcal{U}\ -\ \mathsf{dom}\ s_0)}$ we also have $R\ (h\ s)\ (\mathsf{lift\text{-}hst}\ (t'\ ^{d}|_{\mathsf{dom}\ s_1}))$. Hence the two disjoint heap states can be merged to introduce the goal separation conjunction.

- Abrupt t' — the theorem does not mention abrupt termination post-states, as these are not relevant across function calls in the output of the C translation. As a result, the above reasoning can be applied with an empty post-condition to achieve the goal.
- Fault f, Stuck — follow from the $\neg$ exec-fatal $C\ \Gamma\ (s\ ^{d}|_{\mathsf{dom}\ s_0})$ result.

□

4.4.5 Instantiation

Using Thm. 4.4.4, a local specification can be globalised. The program verifier will typically do this as an explicit step in the proof script, universally quantifying R, h and other aspects of the local specification in addition to the pre-state before supplying the globalised specification to the verification condition generator in a client proof. In the resultant proof obligation, the specification is then available and can be instantiated during proof.

Example 4.4.1. Table 4.10 gives a simple swap operation in C and a separation logic specification. After verification condition generation, we can derive $p \hookrightarrow x$ and $q \hookrightarrow y$ from the pre-state. The single proof obligation can then be lifted to the separation logic level with the rules in §4.3 and the proof completed in a single step. The guards are also dischargeable in a single simplification step during the proof, with $(p \hookrightarrow_g v)\ (\mathsf{lift\text{-}state}\ (h,\ d)) \Longrightarrow \mathsf{ptr\text{-}safe}\ p\ d$ handling the additional ptr-safe p and ptr-safe q guards we require for memory safety.

With the new guards in place, we can use Isabelle's automatic tactics to apply the intra-procedural rewrites and establish mem-safe $(swap(\acute{}p, \acute{}q))\ \Gamma$. The frame rule can then be applied to yield the globalised specification:

$$\begin{array}{l}\forall \sigma\ x\ y\ R\ h.\\ \quad \{\!|\sigma.\ (\acute{}p \mapsto x \wedge^{*}\ \acute{}q \mapsto y \wedge^{*} R\ (h\ \sigma))^{sep}|\!\}\\ \quad swap(\acute{}p, \acute{}q)\\ \quad \{\!|(^{\sigma}p \mapsto y \wedge^{*}\ ^{\sigma}q \mapsto x \wedge^{*} R\ (h\ \sigma))^{sep}|\!\}\end{array}$$

$\forall \sigma\ x\ y.$
$\quad \{\!|\sigma.\ (\acute{}p \mapsto x \wedge^*\ \acute{}q \mapsto y)^{sep}|\!\}$
$\quad swap(\acute{}p, \acute{}q)$
$\quad \{\!|(^{\sigma}p \mapsto y \wedge^*\ ^{\sigma}q \mapsto x)^{sep}|\!\}$

```
void swap(unsigned int *p, unsigned int *q)
{
    unsigned int x;

    x = *p;
    *p = *q;
    *q = x;
}
```

Table 4.10: **swap** specification and definition.

Table 4.11 contains a client function that invokes **swap** twice. The proof here is also very simple, although rather verbose at 47 LoP for such a simple function. The breakdown includes 20 lines related to guards, which just involved the extraction of the appropriate $\hookrightarrow$ assertions to discharge, 13 lines to apply the lifting rules and 10 LoP to instantiate the quantifiers in the globalised specification for **swap** (5 LoP per instantiation). The discharge of guards and application of lifting rules should be easy to automate through Isabelle tactics, as only explicit mappings appearing in the assumption were involved.

$\forall \sigma\ x\ y.$
$\quad \{\!|\sigma.\ (\acute{}a \mapsto x \wedge^*\ \acute{}b \mapsto y \wedge^*\ \acute{}c \mapsto -)^{sep}|\!\}$
$\quad test\text{-}swap(\acute{}a, \acute{}b, \acute{}c)$
$\quad \{\!|(^{\sigma}a \mapsto (x + y) \wedge^*\ ^{\sigma}b \mapsto x \wedge^*\ ^{\sigma}c \mapsto y)^{sep}|\!\}$

```
void test_swap(unsigned int *a, unsigned int *b, unsigned int *c)
{
    swap(a,b);
    *c = *a + *b;
    swap(c,a);
}
```

Table 4.11: **test_swap** specification and definition.

4.5 Examples

4.5.1 In-place list reversal revisited

We now revisit the list reversal example in §3.6. The implementation of **reverse** remains the same as in Table 3.6, and the specification in separation logic is similar on the pre/post level, although without the need to explicitly state the typed heap frame condition:

$$\begin{array}{l}\forall\, zs.\ \{\!|(\mathsf{list}\ zs\ \acute{}i)^{sep}|\!\}\\ \quad \acute{}\mathit{reverse\text{-}ret} :== \mathit{reverse}(\acute{}i)\\ \quad \{\!|(\mathsf{list}\ (\mathsf{rev}\ zs)\ (\mathsf{Ptr}\ \acute{}\mathit{reverse\text{-}ret}))^{sep}|\!\}\end{array}$$

The abstraction predicate is defined differently:

$$\begin{array}{lcl}\mathsf{list}\ []\ i & \equiv & \lambda s.\ i = \mathsf{NULL} \wedge \square\ s\\ \mathsf{list}\ (x{\cdot}xs)\ i & \equiv & \lambda s.\ i = \mathsf{Ptr}\ x \wedge x \neq 0 \wedge (\exists j.\ (i \mapsto j \wedge^{*} \mathsf{list}\ xs\ (\mathsf{Ptr}\ j))\ s)\end{array}$$

The invariant is a bit shorter, because the separating conjunction takes care of distinctness:

$$\{\!|\exists\, xs\ ys.\ (\mathsf{list}\ xs\ \acute{}i \wedge^{*} \mathsf{list}\ ys\ (\mathsf{Ptr}\ \acute{}j))^{sep} \wedge \mathsf{rev}\ zs = \mathsf{rev}\ xs\ @\ ys|\!\}$$

The proof remains easy, but does not have the same degree of automation any more, requiring some manual application of rules for separating conjunction. This is not surprising, since separating conjunction is an existential statement which can lead to manual intervention. The proof of the 3 verification conditions comes to a total of 32 lines, which we might be able to improve with specialised tactics for separation logic connectives, as again almost half of this is related to $\hookrightarrow$ derivation from assertions.

In this separation logic verification, there was no specific automation setup for the list data structure apart from the derivation of the rewrite lemma $\mathsf{list}\ xs\ \mathsf{NULL} = (\lambda s.\ xs = [] \wedge \square\ s)$, which brings the total proof effort for this example to 62 lines, as opposed to 90 lines in the UMM setting. Most of those 90 lines could be reused for programs on the same data structure, though, which is not the case for the separation logic proof.

4.5.2 Factorial

The largest verification example we have seen so far appears in Table 4.12. This is an artificial example, intended to demonstrate recursion, linked structures and some aspects of memory management, not the simplest means of computing the given function. The **factorial** function takes a single parameter n and returns a linked list of objects containing the values $n!$, $(n{-}1)!$, …, 1 if there is enough free memory to represent this, otherwise it

leaves the system with the same amount of memory as when it was invoked, i.e. **factorial** does not leak memory. A significant aspect of this specification is that we can be very precise with our characterisation of the memory usage, allocation and deallocation behaviour of a program.

The data structure used to represent the list is simply two adjacent **unsigned int**s, with the second punned as a pointer. This seems somewhat puzzling as we have developed the type encoding to gain some representation abstraction. The explanation for this is that even though Exmp. 4.2.1 features a **struct**, our **struct** semantics and proof tools are still underdeveloped, and it is simpler to perform proofs using only primitive types until we have the rules in Chapter 5.

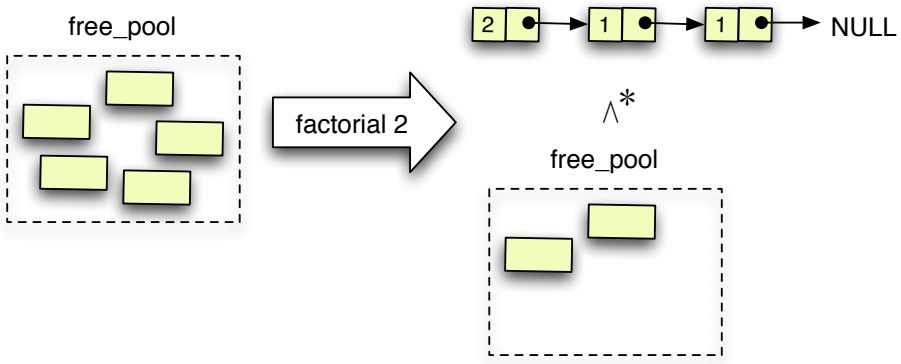

Figure 4.6: **factorial** data structure transformation.

Fig. 4.6 illustrates the operation of **factorial**. We start with a free pool of memory and, assuming it has sufficient size, end up with the intended list and a reduced free pool. The function is divided into two parts; the base and recursive cases. The base case is when $n = 0$ and the allocated storage is filled as expected. In the recursive case $n > 0$, first the function attempts to gain a pointer to the remaining list by a self-call on $n - 1$. Then an attempt is made to allocate storage for the current node. If this fails, **factorial** has cleanup code that traverses the currently allocated list and frees all nodes. The entire list is allocated before the return path begins and the contents of the allocated storage are then filled as intended.

Formally, we characterise the free pool with an underspecified separation assertion free-pool::*nat* ⇒ *heap-assert*, taking the number of free blocks as a parameter. Specifications can then be given for **alloc** and **free**:

$$\begin{array}{l}\forall\,\sigma\ k.\ \{\!|\sigma.\ (\textsf{free-pool}\ k)^{sep}|\!\}\\ \quad\ \acute{}alloc\text{-}ret :== alloc()\\ \quad\ \{\!|(\lambda s.\ \textsf{if}\ 0 < k\\ \qquad\ \textsf{then}\ (\vdash_s \acute{}alloc\text{-}ret \wedge^* \vdash_s (\acute{}alloc\text{-}ret +_p 1) \wedge^* \textsf{free-pool}\ (k - 1))\ s\\ \qquad\ \textsf{else free-pool}\ 0\ s \wedge \acute{}alloc\text{-}ret = \textsf{NULL})^{sep}|\!\}\end{array}$$

$$\begin{array}{l}\forall\,\sigma\ k.\ \{\!|\sigma.\ (\textsf{sep-cut}\ \acute{}p_{\&}\ (2 * \textsf{size-of TYPE}(word32)) \wedge^* \textsf{free-pool}\ k)^{sep}|\!\}\\ \quad\ free(\acute{}p)\\ \quad\ \{\!|(\textsf{free-pool}\ (k + 1))^{sep}|\!\}\end{array}$$

For the specification of **factorial**, in the post-condition we can describe the list similarly to the previous section:

$$\begin{array}{lcl}\textsf{list}\ []\ p & \equiv & \lambda s.\ p = \textsf{NULL} \wedge \square\ s\\ \textsf{list}\ (x{\cdot}xs)\ p & \equiv & \lambda s.\ \exists j.\ (p \mapsto x \wedge^* (p +_p 1) \mapsto j \wedge^* \textsf{list}\ xs\ (\textsf{Ptr}\ j))\ s\end{array}$$

where the supplied list $n!$, $(n{-}1)!$, ..., 1 is given by:

$$\begin{array}{lcl}\textsf{fac-list}\ 0 & \equiv & [1]\\ \textsf{fac-list}\ (Suc\ n) & \equiv & \textsf{fac}\ (Suc\ n){\cdot}\textsf{fac-list}\ n\\ \textsf{fac}\ 0 & \equiv & 1\\ \textsf{fac}\ (Suc\ n) & \equiv & \mathbb{N}^{\Rightarrow}\ (Suc\ n) * \textsf{fac}\ n\end{array}$$

These definitions are combined in the separation assertion:

$$\textsf{sep-fac-list}\ n\ p \equiv \textsf{list}\ (\textsf{fac-list}\ (\mathbb{N}^{\Leftarrow}\ n))\ p$$

The proof follows the now familiar structure of globalisation of the **alloc** and **free** functions, additional lemmas for reasoning about the new separation assertions, verification condition generation and then the proof obligation discharging, involving lifting to the separation logic level and proofs at this level. The cleanup loop invariant is:

$$\{\!|\exists\,xs.\ (\textsf{list}\ xs\ \acute{}q \wedge^* \textsf{free-pool}\ (k - |xs|))^{sep} \wedge |xs| \leq k|\!\}$$

Many of the steps are essentially manipulating assertions into the desired form, to either apply some rule such as the lifting rule for heap-update or a lemma from earlier in the proof about the list data structure, and to match up components in the assumption and goal. The other significant parts of the main proof include extracting information to discharge guards, mostly from conjuncts where the information is available explicitly and some arithmetic reasoning. The entire proof script is 386 LoP for 39 LoC.

$\forall\, \sigma\ k.\ \{\!|\sigma.\ (\mathsf{free\text{-}pool}\ k)^{sep}|\!\}$
$\quad \acute{}factorial\text{-}ret :== factorial(\acute{}n)$
$\quad \{\!|\mathsf{if}\ \acute{}factorial\text{-}ret \neq \mathsf{NULL}$
$\quad\ \mathsf{then}\ (\mathsf{sep\text{-}fac\text{-}list}\ ^{\sigma}n\ \acute{}factorial\text{-}ret \wedge^{*}$
$\quad\quad \mathsf{free\text{-}pool}\ (k - (\mathbb{N}^{\Leftarrow}\ ^{\sigma}n + 1)))^{sep} \wedge$
$\quad\quad \mathbb{N}^{\Leftarrow}\ ^{\sigma}n + 1 \leq k$
$\quad\ \mathsf{else}\ (\mathsf{free\text{-}pool}\ k)^{sep}|\!\}$

```
unsigned int *factorial(unsigned int n)
{
    unsigned int *p, *q;

    if (n == 0) {
        p = alloc();

        if (!p)
            return NULL;

        *p = 1;
        *(p + 1) = 0;

        return p;
    }

    q = factorial(n - 1);

    if (!q)
        return NULL;

    p = alloc();

    if (!p) {
        while (q) {
            unsigned int *r = (unsigned int *)*(q + 1);

            free(q);
            q = r;
        }

        return NULL;
    }

    *p = n * *q;
    *(p + 1) = (unsigned int)q;

    return p;
}
```

Table 4.12: **factorial** specification and definition.

Chapter 5

Structured types

5.1 C's **struct**, **union** and array types

In the C-HOL type encoding of Chapter 2, each C type was given a unique type in the theorem prover. All such types belonged to an axiomatic type class α::*c-type* in Isabelle, which introduced a number of constants that connected the low-level byte representation and the HOL values. Primitive types such as **char** and **long *** could be defined in a library for each architecture/compiler in the expected way.

C's aggregate, or *structured*, types could also be modeled inside the theorem prover, e.g. **struct** types as Isabelle **record** types[1]. Structured types warrant further investigation, however, as they require additional work behind the scenes to instantiate and expose limitations in the existing notion of independently typed heaps which the developments of Chapter 3 and Chapter 4 are based upon. Structured types are also crucial in the implementation of systems code — while, in principle, programs could be implemented without them, this is impractical, as witnessed by Ritchie [82]:

> The language and compiler were strong enough to permit us to rewrite the Unix kernel for the PDP-11 in C during the summer of that year. (Thompson had made a brief attempt to produce a system coded in an early version of C — before structures — in 1972, but gave up the effort.)

A verification framework should therefore provide convenient rules for common usage cases with these types. We treat structured types as first-class C types in the following to provide the benefits of abstraction and typing in proofs, while still allowing direct references to members.

[1] In our implementation of this work, a **record** package substitute based on Isabelle/HOL's **datatype**s was used to allow for structured C types that introduce a circular type dependency, e.g. a **struct x** with a field of type **struct x ***.

Each structured type appearing in a program required the C_{sys} translator, implemented as an Isabelle tactic in ML, to perform an α::*mem-type* instantiation, e.g. for **struct** types:

- A corresponding **record** declaration.
- Definitions of functions appearing in α::*c-type*, requiring full structure information to appear shallowly at the HOL level.
- Lvalue calculations, requiring the full structure information inside the ML parser, as well as offset/size/alignment calculations.

Since the translation stage is trusted it is highly desirable to minimise and simplify it. In our framework the first two steps need to be in the ML, since Isabelle/HOL does not reflect these aspects of the theorem prover's runtime. The last step introduces redundant information that can be mostly pushed to the HOL level and we endeavour to achieve this reduction in the trusted code in §5.2.4.

Example 5.1.1. As a running example, consider the following **struct** declarations:

```
struct x {
    short y;
    char z;
};

struct a {
    int b;
    struct x c;
};
```

The following triple demonstrates the most significant limitation with the earlier memory model:

$$\{\!|\ *p = (\!|\ y = 2,\ z = 'm'\ |\!)\ |\!\}\ p{\rightarrow}y = 1;\ \{\!|\ *p = ?\ |\!\}$$

The problem here is that even though the update and dereference are type-safe, and we do not need to consider aliasing, the proof rules we have developed so far consider this update to be type-unsafe, as any region of memory can only have a single type, and p and $\&(p{\rightarrow}y)$ share a common address despite having different but related types. There is a similar problem for the effect of updates through **struct** references on enclosed field pointer values.

Fig. 5.1 demonstrates how this problem manifests itself in the multiple typed heaps abstraction. Typed heaps now have locations in common, and there is an update dependency given by the arrows between the heaps:

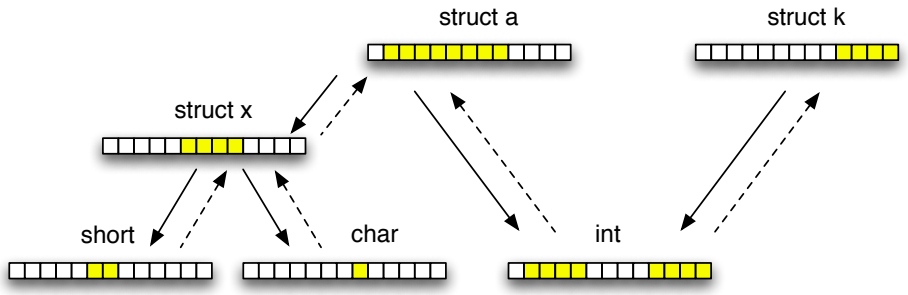

Figure 5.1: Heap update dependencies.

- Updating a field type's heap *may* affect typed heaps of enclosing **struct**s, indicated in the figure by a dashed arrow.

- Updating a **struct** affects typed heaps of field types (fields-of-fields, etc.), indicated in the figure by a solid arrow.

- Update effects are no longer simple function update, they involve potentially multiple field updates and accesses.

The solution we propose in this chapter is to treat structured type information as first-class HOL values and generalise the definitions, rewrites and rules of the previous development making use of this.

The above discussion focused on **struct**s. We maintain the approach to modeling arrays and **union**s as HOL types described in §2.4.3. In the following discussion, arrays inside **struct**s can be considered **struct**s with fields of the same type and with names derived from the index. Since we have not yet implemented **union**s in the C_{sys}-*com* translation, we do not treat them below, but refer the interested reader to the discussion in §7.3 on how they may be handled.

5.2 Structured type encoding

The solution proposed in §5.1 requires that structured type meta-data be available at the HOL level. This needs to include the same information as in §2.4.1 — type structure, size, and alignment. In addition, a fine grained description of the value representation encoding and decoding functions, such that it is possible to extract the functions for specific fields as well as the structure as a whole, is desirable.

5.2.1 Field descriptions

At the HOL level, we represent structure objects using potentially nested Isabelle/HOL **record**s. Each field has access and update functions defined by the **record** package, e.g. for **struct a** represented as HOL record type *a-struct*, the functions b::*a-struct* ⇒ *int* and b-update::(*int* ⇒ *int*) ⇒ *a-struct* ⇒ *a-struct* are supplied. Where possible, it is helpful to use these **record** functions when reasoning about field accesses and updates, rather than the more detailed, lower-level view of fields as a subsequence of the byte-level value representation — the connection between these two views is explored in §5.3.4. To facilitate this, functions derived from the corresponding **record** functions are included in the type meta-data.

Definition 5.2.1. We can capture *abstract* record access and update functions for fields as *field descriptions*:

$$\begin{array}{lcl} \textbf{record}\ \alpha\ \textit{field-desc} & = & \mathsf{field\text{-}access} :: \alpha \Rightarrow \textit{byte list} \Rightarrow \textit{byte list} \\ & & \mathsf{field\text{-}update} :: \textit{byte list} \Rightarrow \alpha \Rightarrow \alpha \end{array}$$

These functions provide a connection between the structure's value as a typed HOL object and the value of a field in the structure as a *byte list*. field-access takes an additional *byte list* parameter, utilised in the semantics to provide the existing state of the *byte* sequence representing the field being described. This allows padding fields the ability to "pass through" the previous state during an update[2].

Example 5.2.1. The field description for field b in **struct a** is:

$$\begin{array}{l} (\!|\mathsf{field\text{-}access} = \mathsf{to\text{-}bytes} \circ \mathsf{b}, \\ \quad \mathsf{field\text{-}update} = \\ \quad\quad \lambda bs\ v. \\ \quad\quad\quad \mathsf{if}\ |bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\textit{word32})\ \mathsf{then}\ v(\!|\mathsf{b} := \mathsf{from\text{-}bytes}\ bs|\!)\ \mathsf{else}\ v|\!) \end{array}$$

The update function only has an effect on *byte list*s of the correct length, a constraint that runs through later definitions and properties.

5.2.2 Extended type tags

Definition 5.2.2. The type meta-data is captured in a *type description* with the following mutually-inductive definitions:

$$\begin{array}{lcll} \textbf{datatype} & \alpha\ \textit{typ-desc} & = & \mathsf{TypDesc}\ \text{“}\alpha\ \textit{typ-struct}\text{”}\ \textit{typ-name} \\ & \alpha\ \textit{typ-struct} & = & \mathsf{TypScalar}\ \textit{nat}\ \textit{nat}\ \alpha\ | \\ & & & \mathsf{TypAggregate}\ (\alpha\ \textit{typ-desc} \times \textit{field-name})\ \textit{list} \end{array}$$

[2] A more conservative, standard compliant approach, would be to use non-determinism or an oracle here.

A type description is a tree, with structures as internal nodes, branches labeled with field names and leaves corresponding to fields with primitive types. At leaves, size, alignment and an α is provided. The α is free and can be used to carry primitive type encoding and decoding functions. Alignment is now considered to be an exponent, enforcing the power-of-two restriction in §2.3.1 structurally. An example type description for **struct a** is given in Fig. 5.2.

As in §2.4.1, there is not a one-to-one correspondence between fields in this structure and those in a C **struct**, as fields in this definition are also intended to explicitly represent the padding inserted by the compiler to ensure alignment restrictions are met.

The previous *typ-info* and *typ-tag* types are now instances of α *typ-desc*, with field descriptions included in the type information:

$$\begin{array}{lrcl} \textbf{types} & \alpha\ \textit{typ-info} & = & \alpha\ \textit{field-desc typ-desc} \\ & \textit{typ-tag} & = & \textit{unit typ-desc} \end{array}$$

The type information provides the information required to describe the encoding and decoding of the representation. $\mathsf{TYPE}(\alpha)_\tau$ gives the type information for an α::*c-type* and $\mathsf{TYPE}(\alpha)_\nu$ provides a type description similar to a *typ-tag*, explained in §5.2.7. Here the subscript operators are functions from α::*c-type itself* to type information and descriptions.

Definition 5.2.3. A field name used to access and update structure fields with the C . and −> operators can be viewed as a *field-name list* of .-separated fields leading to a sub-structure, which we refer to as a *qualified field name*. A qualified field name may lead to a field with a primitive or structure type, e.g. [] is the structure itself. Array members are named by index, e.g. [''*--array-37*''].

Example 5.2.2. [], [''*b*''], and [''*c*'', ''*z*''] are valid qualified field names in Exmp. 5.1.1 for **struct a**, corresponding to the entire structure, `b` field and nested `z` field respectively.

A number of functions can be defined on type descriptions which allow the lifting and update rules of §5.3.4 and §5.4.5 to be expressed and proven. We summarise all these and the other key functions defined over type descriptions introduced in this chapter in Table 5.1. All functions are backed by primitive recursive definitions in Isabelle/HOL, however in some definitions below we replace what constitutes a lengthy and verbose but somewhat trivial HOL term with explanation and examples. In such cases a reference to a table in Appendix B is provided next to the definition.

Definition 5.2.4. [B.1] `map-td` applies the given function f at leaf nodes, modifying the contents of a type description's leaves while not affecting the structure. f is a function of the size and alignment at a leaf node but does not modify these values.

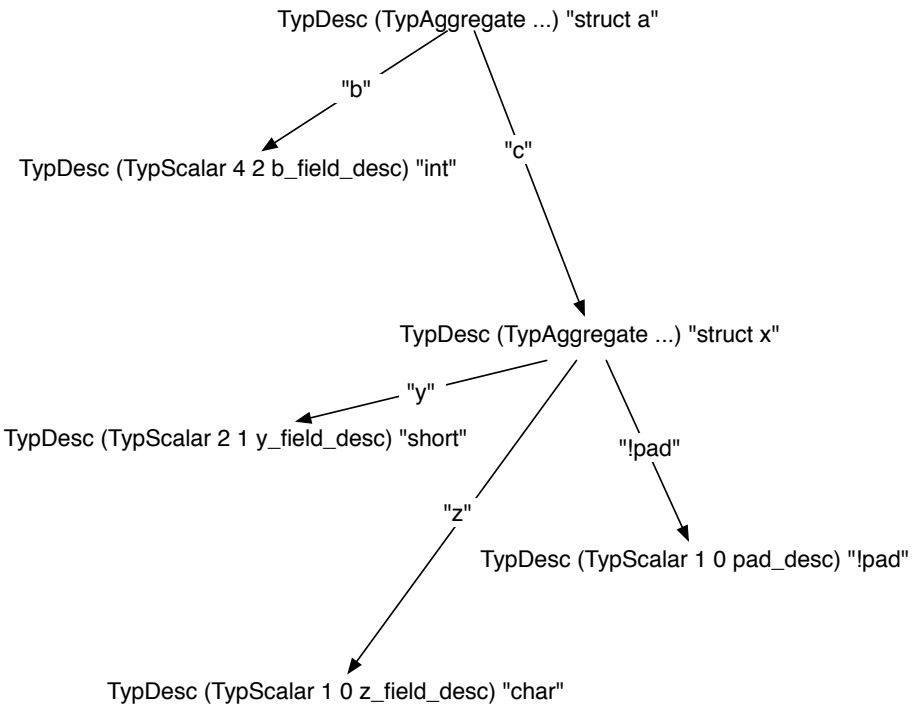

Figure 5.2: Type description for **struct a**.

Definition 5.2.5. [B.2, B.3] Type size size-td and alignment align-td are found by summing and taking the maximum of the leaf node sizes and alignments respectively, similar to the calculations in Defn. 2.4.4 and Defn. 2.4.5.

Definition 5.2.6. [B.4] lookup :: α *typ-desc* $\Rightarrow$ *qualified-field-name* $\Rightarrow$ *nat* $\rightharpoonup$ α *typ-desc* $\times$ *nat* follows a path f from the root of a type description t and returns a sub-tree and offset if it exists. We write $t \triangleright f$ as an abbreviation for lookup t f *0*.

Example 5.2.3. A lookup on the field c in **struct a** yields:

$$\mathsf{TYPE}(a\text{-}struct)_\nu \triangleright [''c''] \;=\; \lfloor (\mathsf{TYPE}(x\text{-}struct)_\nu,\ 4) \rfloor$$

A lookup on an invalid field name fails:

$$\mathsf{TYPE}(a\text{-}struct)_\nu \triangleright [''c'',\ ''b''] \;=\; \bot$$

Lemma 5.2.1. *The size of a type description is no smaller than the sum of the size of any field's type description and offset:*

$$\frac{\mathsf{TYPE}(\alpha)_\tau \triangleright f \;=\; \lfloor (t,\ n) \rfloor}{\mathsf{size\text{-}td}\ t \;+\; n \;\leq\; \mathsf{size\text{-}td}\ \mathsf{TYPE}(\alpha)_\tau}$$

map-td	:: $(nat \Rightarrow nat \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha$ *typ-desc* $\Rightarrow \beta$ *typ-desc*
Transforms leaf α values to β values.	
size-td	:: α *typ-desc* $\Rightarrow$ *nat*
Type size, e.g. size-td $\mathsf{TYPE}(a\text{-}struct)_\tau = 8$.	
align-td	:: α *typ-desc* $\Rightarrow$ *nat*
Type alignment exponent, e.g. align-td $\mathsf{TYPE}(a\text{-}struct)_\tau = 2$.	
- $\triangleright$ -	:: α *typ-desc* $\Rightarrow$ *qualified-field-name* $\rightharpoonup \alpha$ *typ-desc* $\times$ *nat*
The sub-tree and offset from the base of the structure that a valid qualified field name leads to.	
td-set	:: α *typ-desc* $\Rightarrow (\alpha$ *typ-desc* $\times$ *nat*$)$ *set*
The set of all sub-trees and their offset from the base of a structure.	
access-ti	:: α *typ-info* $\Rightarrow (\alpha \Rightarrow$ *byte list* $\Rightarrow$ *byte list*$)$
Derived field access for the entire structure represented by the type information.	
update-ti	:: α *typ-info* $\Rightarrow$ (*byte list* $\Rightarrow \alpha \Rightarrow \alpha$)
Derived field update for the entire structure represented by the type information.	
export-uinfo	:: α *typ-info* $\Rightarrow$ *typ-uinfo*
Export type information (see §5.2.7 for *typ-uinfo*).	
norm-tu	:: *typ-uinfo* $\Rightarrow$ (*byte list* $\Rightarrow$ *byte list*)
Derived normalisation for the entire structure represented by the exported type information.	
- $\leq$ -	:: α *typ-desc* $\Rightarrow \alpha$ *typ-desc* $\Rightarrow$ *bool*
Update dependency order, e.g. $\mathsf{TYPE}(x\text{-}struct)_\nu \leq \mathsf{TYPE}(a\text{-}struct)_\nu$.	

Table 5.1: Type description functions.

Proof. By structural induction on the type description. □

Definition 5.2.7. [B.5] A related concept is the *type description set*, td-set t, of a type description t where all sub-trees and their offsets are returned.

Example 5.2.4. The type description set for **struct x** is:

$$\mathsf{td\text{-}set}\ \mathsf{TYPE}(x\text{-}struct)_\nu = \{(\mathsf{TYPE}(x\text{-}struct)_\nu,\ 0),\ (\mathsf{TYPE}(word16)_\nu,\ 0),\ (\mathsf{TYPE}(word8)_\nu,\ 2),\ (\mathsf{pad\text{-}export}\ 1,\ 3)\}$$

Definition 5.2.8. The address corresponding to an *lvalue* expression containing a structure field access or update can be found with:

$$\&(p{\rightarrow}f) \equiv p_\& + \mathbb{N}^\Rightarrow\ (\mathsf{snd}\ (\mathsf{the}\ (\mathsf{TYPE}(\alpha)_\nu \triangleright f)))$$

Lvalue terms appear in the semantics and proof obligations for statements like p−>f = v; as described in §2.5.6 and §2.5.7.

Example 5.2.5. The lvalue address for an *a-struct ptr* dereference on the c field is given by:

$$\&(p{\rightarrow}[''c''\,]) = p_\& + \mathbb{N}^\Rightarrow\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(word32))$$

Lemma 5.2.2. *The heap interval footprint of a field is a subset of that of an enclosing structure:*

$$\frac{\mathsf{TYPE}(\alpha)_\tau \triangleright f = \lfloor (t,\ n) \rfloor}{\{\&(p \rightarrow f)..+\mathsf{size\text{-}td}\ t\} \subseteq \{p_\&..+\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)\}}$$

Proof. By interval reasoning and Lem. 5.2.1. □

Definition 5.2.9. [B.6, B.7] Access access-ti and update update-ti functions compose their respective primitive leaf functions (from the field descriptions) sequentially to provide the expected encoding and decoding for an aggregate type. Since a given type information may represent an entire structure type or just a field, the access and update functions generalise the earlier notion of to-bytes and from-bytes for a C type.

Example 5.2.6. The access function for **struct a** is given by:

$$\begin{aligned}
\mathsf{access\text{-}ti}\ \mathsf{TYPE}(a\text{-}struct)_\tau \ =\ & \lambda v\ bs. \\
& \mathsf{to\text{-}bytes}\ (\mathsf{b}\ v) \\
& \ (\mathsf{take}\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(word32))\ bs)\ @ \\
& \mathsf{to\text{-}bytes}\ (\mathsf{c}\ v) \\
& \ (\mathsf{take}\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(x\text{-}struct)) \\
& \quad (\mathsf{drop}\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(word32))\ bs))
\end{aligned}$$

Definition 5.2.10. The connection between the HOL typed value, type information, size, alignment and underlying byte representation can be made through the following function definitions:

$$\begin{aligned}
\mathsf{to\text{-}bytes} &\equiv \mathsf{access\text{-}ti}\ \mathsf{TYPE}(\alpha)_\tau \\
\mathsf{from\text{-}bytes}\ bs &\equiv \mathsf{update\text{-}ti}\ \mathsf{TYPE}(\alpha)_\tau\ bs\ \mathsf{arbitrary} \\
\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) &\equiv \mathsf{size\text{-}td}\ \mathsf{TYPE}(\alpha)_\tau \\
\mathsf{align\text{-}of}\ \mathsf{TYPE}(\alpha) &\equiv 2^{\mathsf{align\text{-}td}\ \mathsf{TYPE}(\alpha)_\tau}
\end{aligned}$$

to-bytes is now also a function of the previous *byte list* representation, with the same rationale as field-access in Defn. 5.2.1. We write $\mathsf{access\text{-}ti}_0$ and $\mathsf{to\text{-}bytes}_0$ when a list of zero bytes with length equal to that of the type's size is to be supplied for the padding state. We generalise the constraints on and properties of α::*mem-type*s in the next section.

5.2.3 Type constraints

In this section we describe the fundamental properties that need to hold for each Isabelle/HOL type we use to model a C type. These generalise Defn. 2.4.2, and we show the earlier properties to follow in Thm. 5.2.3 at the end of the section.

Definition 5.2.11. Table 5.2 gives the constraints on an α::*c-type* for instantiation in the α::*mem-type* axiomatic type class. [MaxSize], [AlignDvdSize] and [AlignField] give some basic size and alignment related properties.

$$\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) < |\mathsf{addr}| \qquad [\textsc{MaxSize}]$$

$$\mathsf{align\text{-}of}\ \mathsf{TYPE}(\alpha)\ \mathsf{dvd}\ \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) \qquad [\textsc{AlignDvdSize}]$$

$$\mathsf{TYPE}(\alpha)_\tau \triangleright f = \lfloor (s,\ n) \rfloor \longrightarrow 2^{\mathsf{align\text{-}td}\ s\ \mathsf{dvd}\ n} \qquad [\textsc{AlignField}]$$

$$\frac{|bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)}{\mathsf{update\text{-}ti}\ \mathsf{TYPE}(\alpha)_\tau\ bs\ v = \mathsf{update\text{-}ti}\ \mathsf{TYPE}(\alpha)_\tau\ bs\ w} \qquad [\textsc{Upd}]$$

$$\mathsf{wf\text{-}desc}\ \mathsf{TYPE}(\alpha)_\tau \qquad [\textsc{WFDesc}]$$

$$\mathsf{wf\text{-}size\text{-}desc}\ \mathsf{TYPE}(\alpha)_\tau \qquad [\textsc{WFSizeDesc}]$$

$$\mathsf{wf\text{-}field\text{-}desc}\ \mathsf{TYPE}(\alpha)_\tau \qquad [\textsc{WFFD}]$$

Table 5.2: α::*mem-type* axioms.

The [MaxSize] and [AlignDvdSize] conditions are taken directly from Defn. 2.4.2 and [AlignField] is implied by the C standard's requirement that derived field pointers posses the alignment of their type [1, 6.7.2.1–12].

[Upd] states that the result of an update to the entire structure is independent of the original value.

Finally, three well-formedness conditions on the type information ensure sensible values for field names, node sizes and field descriptions. These conditions are detailed below in Defn. 5.2.12, Defn. 5.2.13 and Defn. 5.2.15.

Definition 5.2.12. [B.8] A type description t is well-formed w.r.t. field names, wf-desc t, when no node has two or more branches labelled with the same field name.

Definition 5.2.13. [B.9] A type description t is well-formed w.r.t. size, wf-size-desc t, when every node has a non-zero size.

Definition 5.2.14. A field description d and size n are considered *consistent*, fd-cons-desc d n, when the following properties hold:

$$\forall\, v\ bs\ bs'.\ |bs| = |bs'| \longrightarrow \mathsf{field\text{-}update}\ d\ bs\ (\mathsf{field\text{-}update}\ d\ bs'\ v) = \mathsf{field\text{-}update}\ d\ bs\ v \qquad [\textsc{FuFu}]$$

$$\forall\, v\ bs.\ |bs| = n \longrightarrow \mathsf{field\text{-}update}\ d\ (\mathsf{field\text{-}access}\ d\ v\ bs)\ v = v \qquad [\textsc{FuFaId}]$$

$$\forall\, bs\ bs'\ v\ v'.\ |bs| = n \longrightarrow |bs'| = n \longrightarrow \mathsf{field\text{-}access}\ d\ (\mathsf{field\text{-}update}\ d\ bs\ v)\ bs' = \mathsf{field\text{-}access}\ d\ (\mathsf{field\text{-}update}\ d\ bs\ v')\ bs' \qquad [\textsc{FaFu}]$$

$$\forall\, v\ bs.\ |bs| = n \longrightarrow |\mathsf{field\text{-}access}\ d\ v\ bs| = n \qquad \text{[FaLen]}$$

The properties are similar to those already provided by Isabelle's **record** package at the HOL level and can be established automatically.

Definition 5.2.15. [B.10] Type information t is well-formed w.r.t. field descriptions, wf-field-desc t, if the field descriptions of all leaf fields are consistent, and for every pair of distinct leaf fields, s and t, the following properties hold:

$$\begin{aligned}&\forall\, v\ bs\ bs'.\\ &\quad \mathsf{update\text{-}ti}\ s\ bs\ (\mathsf{update\text{-}ti}\ t\ bs'\ v) =\\ &\quad \mathsf{update\text{-}ti}\ t\ bs'\ (\mathsf{update\text{-}ti}\ s\ bs\ v)\end{aligned} \qquad \text{[FuCom]}$$

$$\begin{aligned}&\forall\, v\ bs\ bs'.\\ &\quad |bs| = \mathsf{size\text{-}td}\ t \longrightarrow\\ &\quad |bs'| = \mathsf{size\text{-}td}\ s \longrightarrow\\ &\quad \mathsf{access\text{-}ti}\ s\ (\mathsf{update\text{-}ti}\ t\ bs\ v)\ bs' =\\ &\quad \mathsf{access\text{-}ti}\ s\ v\ bs'\end{aligned} \qquad \text{[FaFuInd]}$$

Again, these are standard commutativity and non-interference properties that we have at the HOL level and wish to preserve in field descriptions.

We now show that the earlier axioms follow from the generalised axioms presented in this section, allowing the reuse of many of the further derived properties and program verification proofs.

Theorem 5.2.3. *The* α*::mem-type axioms from this section imply the remaining axioms in Defn. 2.4.2:*

$$\frac{|bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)}{\mathsf{from\text{-}bytes}\ (\mathsf{to\text{-}bytes}\ x\ bs) = x} \qquad \text{[Inv]}$$

$$\frac{|bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)}{|\mathsf{to\text{-}bytes}\ x\ bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)} \qquad \text{[Len]}$$

$$0 < \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) \qquad \text{[SzNZero]}$$

$$\mathsf{align\text{-}of}\ \mathsf{TYPE}(\alpha)\ \mathsf{dvd}\ |\mathsf{addr}| \qquad \text{[Align]}$$

Proof. For [Inv], we unfold Defn. 5.2.10 and transform the arbitrary to a v using [Upd]. [WFFD] gives that field descriptions at leaves are consistent, which inductively provides this property for derived field descriptions at internal nodes. The proof is completed by using [FuFaId] at the root. [Len] follows from Defn. 5.2.10 and [FaLen]. [SzNZero] is implied by [WFSizeDesc]. Finally, for [Align], observe align-of $\mathsf{TYPE}(\alpha) <$ |addr| from [AlignDvdSize], hence the alignment as a power-of-two divides |addr|, a larger power-of-two. □

5.2.4 Type combinators

The constraints of the previous section require both the construction of suitable type information and a corresponding α::*mem-type* instantiation proof for each type appearing in programs we wish to verify. This can be done entirely at the ML level during C-HOL translation, by synthesising both the intended HOL term for the type information directly and a proof on the unfolded definition, but this is fragile and does not scale well.

An improved approach to type information construction is to do so using constructor combinators for which generic proof rules can be given. This then reduces the proof effort at the ML level to discharging simple side-conditions resulting from applying the proof rules from the library, greatly reducing the complexity of the ML instantiation code and improving the performance of this step. In this section we detail the combinators and proof rules.

The construction of type information occurs field-wise, in the order of the fields as declared. The importance of this becomes apparent in Defn. 5.2.20, where the correct calculation of padding fields requires this ordering. We now give 5 type information combinators — the empty type information, field extension, padding extension, field-with-padding extension and type information finalisation.

Definition 5.2.16. The empty type information for a type name tn is given by:

$$\mathsf{empty\text{-}typ\text{-}info}\ tn \equiv \mathsf{TypDesc}\ (\mathsf{TypAggregate}\ [])\ tn$$

Definition 5.2.17. Type information ti::α *typ-info* can be extended with a given additional field's type t::β::*c-type itself*, access f_a::$\alpha \Rightarrow \beta$ and update f_u::$\beta \Rightarrow \alpha \Rightarrow \alpha$ functions and name:

$$\mathsf{ti\text{-}typ\text{-}combine}\ t\ f_a\ f_u\ fn\ ti \equiv \mathsf{extend\text{-}ti}\ ti\ (\mathsf{adjust\text{-}ti}\ \mathsf{TYPE}(\beta)_\tau\ f_a\ f_u)\ fn$$

where the functions adjust-ti and extend-ti adapt the existing field descriptions in the new field's type information to work as a field in the combined type information and extend the existing type information with the field, respectively:

$$\begin{array}{lcl}
\mathsf{extend\text{-}ti}\ (\mathsf{TypDesc}\ st\ nm)\ t\ fn & \equiv & \mathsf{TypDesc}\ (\mathsf{extend\text{-}ti\text{-}struct}\ st\ t\ fn)\ nm \\
\mathsf{extend\text{-}ti\text{-}struct}\ (\mathsf{TypAggregate}\ ts)\ t\ fn & \equiv & \mathsf{TypAggregate}\ (ts\ @\ [\langle t,fn\rangle])
\end{array}$$

$$\begin{array}{lcl}
\mathsf{update\text{-}desc}\ f_a\ f_u\ d & \equiv & (\!|\mathsf{field\text{-}access} = \mathsf{field\text{-}access}\ d \circ f_a, \\
 & & \quad \mathsf{field\text{-}update} = \lambda bs\ v.\ f_u\ (\mathsf{field\text{-}update}\ d\ bs\ (f_a\ v))\ v|\!) \\
\mathsf{adjust\text{-}ti}\ t\ f_a\ f_u & \equiv & \mathsf{map\text{-}td}\ (\lambda n\ algn.\ \mathsf{update\text{-}desc}\ f_a\ f_u)\ t
\end{array}$$

Definition 5.2.18. Type information can be extended with a padding field n bytes wide:

```
ti-pad-combine n ti ≡
let fn = foldl op @ ''!pad-'' (field-names-list ti);
    td = (|field-access = λv. id, field-update = λbs. id|);
    nf = TypDesc (TypScalar n 0 td) ''!pad-typ''
in extend-ti ti nf fn
```

where field-names-list provides a list of the field names in a type description. In this way, a unique field name for the padding field is generated to preserve [WFDESC]. Padding fields are anonymous with the access functions preserving existing state and update functions not affecting the value of the structure.

This treatment of padding is somewhat idealised. In reality, the C standard allows the implementation to treat padding fields in an arbitrary fashion during update. Hence the definition of the ti-pad-combine combinator invokes implementation-dependent behaviour, as do the padding calculations in Defn. 5.2.20 and Defn. 5.2.21 below. These combinators may require modification if the target implementation behaves differently[3]. An alternative to the reliance on this behaviour would be to explicitly supply padding fields inside structure declarations in C programs such that the compiler has no need to introduce its own padding.

Definition 5.2.19. The padding needed to meet the given alignment *align* for a structure of size n is:

```
padup align n ≡ (align − n mod align) mod align
```

The next definition builds on ti-typ-combine and ti-pad-combine to provide a combinator we use to add a new field while meeting C's alignment requirements.

Definition 5.2.20. Type information $ti{::}\alpha$ *typ-info* can be extended with a given additional field's type $t{::}\beta{::}$*c-type itself*, access $f_a{::}\alpha \Rightarrow \beta$ and update $f_u{::}\beta \Rightarrow \alpha \Rightarrow \alpha$ functions, name and any necessary padding to meet alignment requirements by:

```
ti-typ-pad-combine t f_a f_u fn ti ≡
let pad = padup (align-of TYPE(β)) (size-td ti)
in ti-typ-combine t f_a f_u fn (if 0 < pad then ti-pad-combine pad ti else ti)
```

[3] In less ideal situations it may be necessary to construct an underspecified *oracle* function to provide the correct padding behaviours.

A padding field is inserted before the intended field if necessary — an empty padding field would violate [WFSizeDesc].

After each field has been added, padding has been introduced to ensure alignment requirements are met for the fields, but it may still be necessary to place an additional padding field to meet [AlignDvdSize].

Definition 5.2.21. Termination of type information construction is achieved by appending a padding field to meet the alignment requirements of the entire structure:

```
final-pad ti ≡
let n = padup 2^(align-td ti) (size-td ti)
in if 0 < n then ti-pad-combine n ti else ti
```

Example 5.2.7. Type information for *a-struct* can be created with:

```
final-pad
 (ti-typ-pad-combine TYPE(x-struct) c (c-update ∘ (λx -. x)) ''c''
   (ti-typ-pad-combine TYPE(word32) b (b-update ∘ (λx -. x)) ''b''
     (empty-typ-info ''a-struct'')))
```

So far we have combinators that allow the construction of the type information, but we still require a proof to place a type with the constructed type information in α::*mem-type*. The rules to do this are rather detailed, and the proofs are involved, so we relegate this to Appendix C. It is sufficient to note here that they are a complete set of rules, in the sense that they cover all the α::*mem-type* axioms and above combinators, and can be applied as introduction rules, with the aid of the simplifier for some side-conditions.

5.2.5 Type installation

During the state space synthesis of Fig. 2.4, the Isabelle/HOL type for a structured type is created and the combinators are used to construct the type information. The translation process then applies the proof rules and the new type is available as an α::*mem-type*.

At the same time, some additional rewrites are shown by the system, and placed in the default simplification set, to allow efficient rewriting of lookup terms for the new type and to improve the scalability of the instantiation process. The lookup rewrites are of great import when applying the later UMM or separation logic rules in this chapter, as $\triangleright$ terms appear frequently as side-conditions.

Example 5.2.8. The following rule for resolving field names beginning with $''z''$ is installed for *x-struct*:

lookup TYPE(x-*struct*)$_\tau$ ($''z''$·*fs*) m =
lookup (adjust-ti TYPE(*word8*)$_\tau$ z (z-update ∘ (λx -. x))) *fs*
(m + size-of TYPE(*word16*))

Simplifications for size-td and align-td on the entire structure are also installed. This has the advantage that when structure s occurs as a field of t, the α::*mem-type* type instantiation of t does not need to unfold the type information for s.

5.2.6 Heap semantics

The translation of §2.5 remains mostly unchanged with the new type encoding, with the exception of heap-update in Defn. 2.5.4, which now supplies to-bytes with the existing heap state underneath the target footprint to facilitate padding field semantics:

heap-update p v h ≡
heap-update-list $p_\&$ (to-bytes v (heap-list h (size-of TYPE(α)) $p_\&$)) h

Structured types introduce a new initialisation concern, where an object may be partially initialised. This is not directly relevant to the type encoding, as any potential exception conditions or other undefined behaviour that may result can be treated separately with guards in §2.5.5. We do not supply initialisation guards but believe that the framework is sufficient to accommodate these if required.

5.2.7 Representation normalisation

In later sections of this chapter, we make frequent use of the concepts of exported type information and normalisation, which we introduce in this section.

Type information may be "exported" to remove the α dependency by collapsing leaf field descriptions to *byte list* normalisation functions, resulting in a *typ-uinfo*:

types *typ-uinfo* = (*byte list* ⇒ *byte list*) *typ-desc*

Normalisation is motivated by the observation that padding fields are ignored when reading structured values from their byte representation. Also, there may exist more than one byte representation for a value in C, even for primitive types. Export of the type information also provides us with a means to quantify over and compare C types.

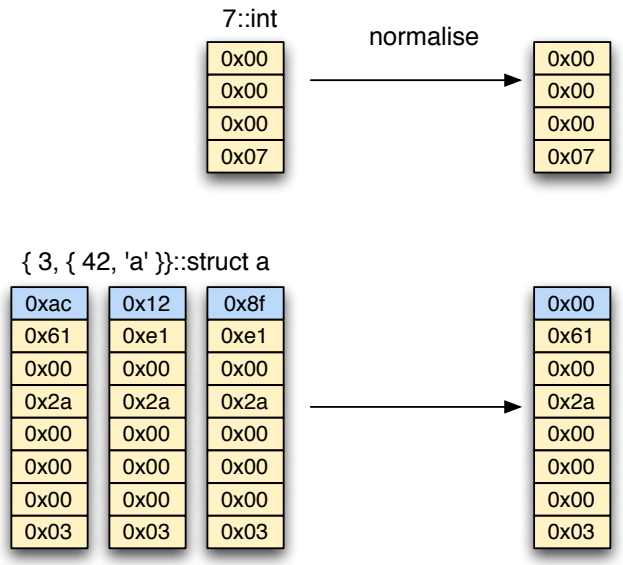

Figure 5.3: Normalisation mapping to *byte list* equivalence classes.

Example 5.2.9. Fig. 5.3 demonstrates two example normalisations. The *byte list*s are arranged with the least-significant byte at the bottom and the shaded *byte*s indicate padding. In the **struct a** case, the padding field is transformed to zero and the MSB in the **char** field is ignored.

Definition 5.2.22. Type information is exported with export-uinfo:

export-uinfo $ti \equiv$
map-td
($\lambda n\ algn\ d\ bs.$
  if $|bs| = n$
  then field-access d (field-update $d\ bs$ arbitrary) (replicate $n\ 0$)
  else [])
ti

We write $\mathsf{TYPE}(\alpha)_\nu$ for export-uinfo $\mathsf{TYPE}(\alpha)_\tau$.

We can no longer obtain derived access and update functions using access-ti and update-ti from exported type information. Instead, we can derive a normalisation function for the entire structure from the leaf normalisation functions.

Definition 5.2.23. [B.11] Normalisation norm-tu for type information is derived by the sequential composition of leaf normalisation functions. We write norm-bytes $\mathsf{TYPE}(\alpha)$ for norm-tu (export-uinfo $\mathsf{TYPE}(\alpha)_\tau$).

Theorem 5.2.4. norm-tu *applied to exported type information is equivalent to normalisation with the access and update functions derived from the type information:*

$$\frac{\text{wf-field-desc } ti \qquad \text{wf-desc } ti \qquad |bs| = \text{size-td } ti}{\text{norm-tu } (\text{export-uinfo } ti)\ bs = \text{access-ti}_0\ ti\ (\text{update-ti } ti\ bs\ \text{arbitrary})}$$

Proof. By structural induction on the type information *ti*. The base case occurs at the leaves and matches the definitions. For internal nodes, we use the inductive hypothesis and the commutativity and non-interference properties derivable from [WFFD]. □

Theorem 5.2.5. *Normalisation does not affect the HOL value of a byte list:*

$$\frac{|bs| = \text{size-of TYPE}(\alpha)}{\text{from-bytes } (\text{norm-bytes TYPE}(\alpha)\ bs) = \text{from-bytes } bs}$$

Proof. From definitions, Thm. 5.2.4 and Defn. 5.2.14 properties. □

Theorem 5.2.6. *Field access is equivalent to normalisation of the corresponding fragment of the underlying byte list representation:*

$$\frac{\text{TYPE}(\alpha)_\tau \triangleright f = \lfloor (t,\ n) \rfloor \qquad |bs| = \text{size-of TYPE}(\alpha)}{\begin{array}{c}\text{access-ti}_0\ t\ (\text{from-bytes } bs) = \\ \text{norm-tu } (\text{export-uinfo } t)\ (\text{take } (\text{size-td } t)\ (\text{drop } n\ bs))\end{array}}$$

Proof. By Thm. 5.2.4 and:

$$\frac{\begin{array}{c} s \triangleright f = \lfloor (t,\ n) \rfloor \\ |bs| = \text{size-td } s \qquad |bs'| = \text{size-td } t \qquad \text{wf-lf } (\text{lf-set } s\ []) \qquad \text{wf-desc } s \end{array}}{\begin{array}{c}\text{access-ti } t\ (\text{update-ti } s\ bs\ v)\ bs' = \\ \text{access-ti } t\ (\text{update-ti } t\ (\text{take } (\text{size-td } t)\ (\text{drop } n\ bs))\ \text{arbitrary})\ bs'\end{array}}$$

This auxiliary rule can be shown by structural induction on the type information. □

5.3 Structured UMM

While the unified memory model (UMM) of Chapter 3 provides a basis for the construction of the typed heaps and separation logic proof abstractions for all α::*mem-type*s, there are some shortcomings when using this in proofs about structured types. These were introduced at the end of §5.1 and in this section we give an extended and generalised UMM that overcomes these limitations.

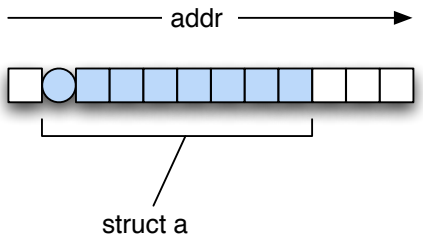

Figure 5.4: Previous heap type description with a valid **struct a** pointer.

5.3.1 Extended heap type description

The heap type description of §3.2 provided a ghost variable, capturing the implicit mapping between memory locations and types. Fig. 5.4 depicts an example valid pointer in the heap type description for **struct a**.

The problem with this notion of the heap type description is that only a single pointer may be valid at any location. This gives rise to the inability to abstractly reason about updates through pointers to fields. With structured types, we would like that at the base address a pointer for the structure type and that of the first field's type be valid. In general, for valid qualified field names f, we desire a *field monotonicity* property, i.e. $d,g \models_t p \implies d,g \models_t \mathsf{Ptr}\ \&(p{\rightarrow}f)$.

To accomplish this, we introduce a new definition for the heap type description:

$$\begin{array}{lll} \textbf{types} & \textit{typ-base} & = \textit{bool} \\ & \textit{typ-slice} & = \textit{nat} \rightharpoonup \textit{typ-uinfo} \times \textit{typ-base} \\ & \textit{heap-typ-desc} & = \textit{addr} \Rightarrow \textit{bool} \times \textit{typ-slice} \end{array}$$

Each location maps to a tuple, with the first component a *bool* indicating whether there is a value located at the address[4]. The second component is a *typ-slice*, providing an indexed map[5] to the *typ-uinfo*s that may reside at a particular address. The index for the exported type information of a field type at a particular offset is calculated from the depth of the tree at the offset, where zero corresponds to the deepest field type and the highest index to the root type. The *typ-base* value indicates whether the location is the base or some other part of a value's footprint[6].

[4] This approach is taken in preference to a partial function to aid in partitioning state in §5.4.

[5] A list could also have been used, however the map allows for separation logic predicates in §5.4.6 that cannot have states modeled with finite lists.

[6] This is for the same reason as in §3.2.1, i.e. allowing consideration of the potential overlap of values of the same type to be eliminated for valid pointers, thus overcoming the problem of skewed sharing.

An example of the extended heap type description is provided in Fig. 5.5. Presence or absence of a value is not indicated. Each point is a *typ-uinfo* × *bool* pair, with the colour determined by the first component and shape by the second. Here a **struct a**, from Exmp. 5.1.1, footprint extends on the horizontal axis above the footprints of its members. The vertical axis indicates a position in the *typ-slice* at the address. The second half of the *a-struct* is higher than the first, as the tree is deeper due to the *x-struct* changing the depth past this offset. That is, at $(p,1)$, $(p+1,1)$,...,$(p+3,1)$ we have an entry for the exported type information of *a-struct*, as the tree has only a depth of 2 at offsets 0–3, but at offsets 4–7, we have *a-struct* at $(p+4,2)$,...,$(p+7,2)$, as the tree is one deeper. An observation about the intuition behind pointer validity that can be taken from this figure is that it is independent of the presence or absence of type information from enclosing structured types in the ghost variable. The validity of the **short** entry at q requires only the entries at $(q,0)$ and $(q+1,0)$, identical in the situations where q is a field of a structure or an independent object.

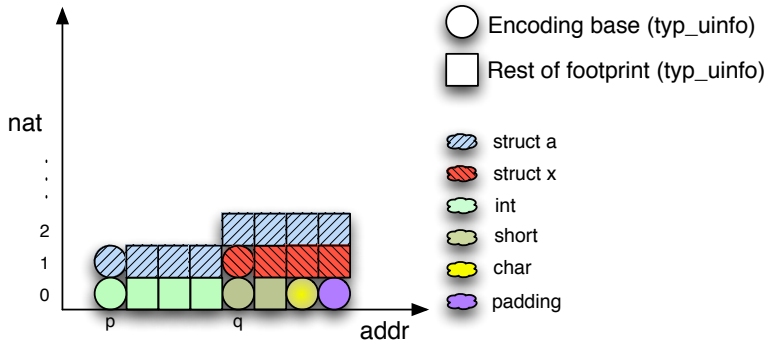

Figure 5.5: Extended heap type description with a valid **struct a** pointer.

Validity

Definition 5.3.1. [B.12] Pointer validity is defined for the heap type description as:

$$\begin{array}{l}\mathsf{valid\text{-}footprint}\ d\ x\ t \equiv \\ \mathsf{let}\ n = \mathsf{size\text{-}td}\ t \\ \mathsf{in}\ 0 < n \wedge \\ \quad (\forall\, y{<}n.\ \mathsf{list\text{-}map}\ (\mathsf{typ\text{-}slice}\ t\ y) \subseteq_m \mathsf{snd}\ (d\ (x + \mathbb{N}^{\Rightarrow}\ y)) \wedge \\ \qquad\quad \mathsf{fst}\ (d\ (x + \mathbb{N}^{\Rightarrow}\ y)))\end{array}$$

$$d,g \models_t (p::\alpha::c\text{-}type\ ptr) \equiv \mathsf{valid\text{-}footprint}\ d\ p_{\&}\ \mathsf{TYPE}(\alpha)_\nu \wedge g\ p$$

where list-map::α *list* $\Rightarrow$ (*nat* $\rightharpoonup$ α) converts a list to the expected map and typ-slice::*typ-uinfo* $\Rightarrow$ *nat* $\Rightarrow$ (*typ-uinfo* $\times$ *typ-base*) *list* takes a vertical slice of the intended heap footprint from the type description at an offset, e.g.:

$$\text{typ-slice TYPE}(a\text{-}struct)_\nu\ 4 \quad = \quad [(\text{TYPE}(word16)_\nu,\ \text{True}),\ (\text{TYPE}(x\text{-}struct)_\nu,\ \text{True}),\ (\text{TYPE}(a\text{-}struct)_\nu,\ \text{False})]$$

corresponding to $(q,0)$, $(q,1)$ and $(q,2)$ in Fig. 5.5 respectively.

The use of the map subset operator $\subseteq_m$ provides monotonicity.

Definition 5.3.2. Field monotonicity for a guard is defined as:

$$\begin{array}{l}\text{guard-mono } (g::\alpha\ ptr \Rightarrow bool)\ (g'::\beta\ ptr \Rightarrow bool) \equiv \\ \forall\, n\ f\ p.\ g\ p \wedge \text{TYPE}(\alpha)_\nu \triangleright f = \lfloor(\text{TYPE}(\beta)_\nu,\ n)\rfloor \longrightarrow g'\ (\text{Ptr}\ (p_\& + \mathbb{N}^\Rightarrow\ n))\end{array}$$

In normal usage, both arguments are the same polymorphic function, e.g. guard-mono ptr-aligned ptr-aligned. This allows us to indirectly quantify over types when using this definition in theorems below.

Theorem 5.3.1. *Validity has field monotonicity:*

$$\frac{\text{TYPE}(\alpha)_\tau \triangleright f = \lfloor(s,\ n)\rfloor \qquad \text{export-uinfo}\ s = \text{TYPE}(\beta)_\nu \qquad \text{guard-mono}\ g\ g'}{d,g \models_t p \longrightarrow d,g' \models_t \text{Ptr}\ \&(p \rightarrow f)}$$

where $p::\alpha\ ptr$ *and* Ptr $\&(p \rightarrow f)::\beta\ ptr$.

Proof. Unfold definitions and consider the typ-slice at some offset y in the field. Since $n + y <$ size-of TYPE(α), from Lem. 5.2.1, we can infer from p's validity that the first component of the tuple at $p_\& + \mathbb{N}^\Rightarrow\ n + \mathbb{N}^\Rightarrow\ y$ is true and that list-map (typ-slice TYPE(α)$_\nu$ $(y + n)$) $\subseteq_m$ snd $(d\ (p_\& + \mathbb{N}^\Rightarrow\ n + \mathbb{N}^\Rightarrow\ y))$. The proof is completed with $\subseteq_m$ transitivity and:

$$\frac{(s,\ n) \in \text{td-set}\ t \qquad k < \text{size-td}\ s}{\text{typ-slice}\ s\ k \leq \text{typ-slice}\ t\ (n + k)}$$

which can be shown with structural induction on the type description. □

Theorem 5.3.2. *The guards in §2.5.5 introduced by the VCG for pointer dereferences are field monotonic:*

$$\text{guard-mono c-guard c-guard}$$

Proof. guard-mono ptr-aligned ptr-aligned follows from [AlignField] and:

$$\frac{t \triangleright f = \lfloor(s,\ n)\rfloor}{\text{align-td}\ s \leq \text{align-td}\ t}$$

which can be seen by structural induction on the type description. c-null-guard is monotonic by Lem. 5.2.2.

□

Retyping

As in Defn. 3.2.4, we have a retyping function ptr-retyp::α::*c-type ptr* $\Rightarrow$ *heap-typ-desc* $\Rightarrow$ *heap-typ-desc* that updates the heap type description such that the given pointer is valid and locations outside the pointer's footprint remain untouched.

Definition 5.3.3. A region of memory may be retyped such that p::α::*c-type ptr* is valid:

$$\begin{array}{l}\textsf{htd-update-list}\ p\ []\ d \equiv d\\ \textsf{htd-update-list}\ p\ (x{\cdot}xs)\ d \equiv \textsf{htd-update-list}\ (p+1)\ xs\\ \qquad\qquad (d(p := (\textsf{True}, \textsf{snd}\ (d\ p)\ +\!\!+\ x)))\end{array}$$

$$\begin{array}{l}\textsf{typ-slices TYPE}(\alpha) \equiv\\ \textsf{map}\ (\lambda n.\ \textsf{list-map}\ (\textsf{typ-slice TYPE}(\alpha)_\nu\ n))\ [0..{<}\textsf{size-of TYPE}(\alpha)]\end{array}$$

$$\textsf{ptr-retyp}\ p \equiv \textsf{htd-update-list}\ p_\& \ (\textsf{typ-slices TYPE}(\alpha))$$

htd-update-list is similar to heap-update-list, but transforms the heap type description instead of heap memory. The typ-slices gives the slices of the type description that occur at the offsets corresponding to list indices.

ptr-retyp is a little different to ptr-tag in that it does not clear the locations being retyped, but instead merges the new map with the existing contents at each updated location. Additional entries in the indexed map at a location in the heap type description do not affect the validity of the target pointer, and hence do not require removal. In §5.4.6 we exploit this to provide separation logic predicates that use dummy type entries above a value's footprint to reserve space for later retyping.

Lemma 5.3.3. *Inside the retyped region,* ptr-retyp (p::α::*mem-type ptr*) *provides the expected heap type description value:*

$$\frac{x \in \{p_\&..{+}\textsf{size-of TYPE}(\alpha)\}}{\begin{array}{l}\textsf{ptr-retyp}\ p\ d\ x =\\ (\textsf{True}, \textsf{snd}\ (d\ x)\ +\!\!+\ \textsf{list-map}\ (\textsf{typ-slice TYPE}(\alpha)_\nu\ (\mathbb{N}^{\Leftarrow}\ (x - p_\&))))\end{array}}$$

Proof. By induction on the list. □

Using Defn. 5.3.3 we can restate and demonstrate Thm. 3.2.4 and Thm. 3.2.5.

Theorem 5.3.4. *Following retyping, a target pointer* p::α::*mem-type ptr is valid:*

$$\frac{g\ p}{\textsf{ptr-retyp}\ p\ d, g \models_t p}$$

Proof. By unfolding definitions, considering a point in the footprint and Lem. 5.3.3. □

Theorem 5.3.5. *A previously valid pointer q::β::mem-type ptr remains valid across a retype as long as its footprint and p::α::mem-type ptr's are disjoint:*

$$\frac{d,g \models_t q \qquad \{p_\&..+\text{size-of TYPE}(\alpha)\} \cap \{q_\&..+\text{size-of TYPE}(\beta)\} = \emptyset}{\text{ptr-retyp } p\ d,g \models_t q}$$

Proof. We have $x \notin \{p_\&..+\text{size-of TYPE}(\alpha)\} \implies \text{ptr-retyp } p\ d\ x = d\ x$ by list induction. The rule then follows by unfolding and application of this fact to show each point in the footprint remains unchanged. □

As before, application-specific rules can be developed to provide a convenient interface to retyping that avoids interval reasoning.

5.3.2 Lifting

Two lifting stages are again used to provide an abstract heap view for proofs. The stages differ from those in §3.3 in the underlying state space for the heap type description and the intermediate heap state.

Definition 5.3.4. The first stage, lift-state, results in a new intermediate *heap-state*:

datatype	*s-heap-index*	=	SIndexVal | SIndexTyp *nat*
datatype	*s-heap-value*	=	SValue *byte* | STyp *typ-uinfo* × *typ-base*
types	*s-addr*	=	*addr* × *s-heap-index*
	heap-state	=	*s-addr* ⇀ *s-heap-value*

An example of this state is provided in Fig. 5.6, with an *x-struct* footprint. This should be read as with Fig. 5.5, the vertical axis now the second component of *s-addr* rather than an index. The rationale for this model is based on the requirements of the separation logic embedding and is provided in §5.4.1.

The function lift-state filters out locations that are False or ⊥ in the heap type description, depending on the index, removing values that should not affect the final lifted typed heaps. Equality between lifted heaps is then modulo the heap type description locations of interest for valid pointers.

```
lift-state ≡
λ(h, d) (x, y).
  case y of SIndexVal ⇒ if fst (d x) then ⌊SValue (h x)⌋ else ⊥
  | SIndexTyp n ⇒ option-case ⊥ (Some ∘ STyp) (snd (d x) n)
```

The second lifting stage results in typed heaps again, defined as in Defn. 3.3.5. The two stages, shown in Fig. 5.7, are combined with lift_τ:

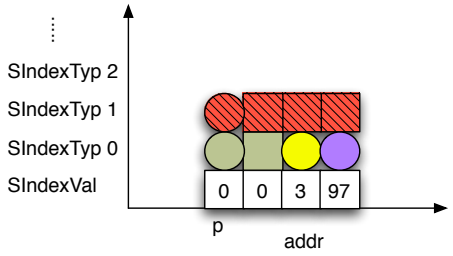

Figure 5.6: Example *heap-state*.

$$\mathsf{lift}_\tau\ g \equiv \mathsf{lift\text{-}typ\text{-}heap}\ g \circ \mathsf{lift\text{-}state}$$

Theorem 5.3.6. *A mapping in a heap lifted from the intermediate heap state implies the existence of a mapping for all valid fields at the corresponding offset in the field type's lifted heap, with a value derived using the field access function:*

$$\frac{\mathsf{TYPE}(\alpha)_\tau \triangleright f = \lfloor (t,\ n) \rfloor \qquad \mathsf{lift\text{-}typ\text{-}heap}\ g\ s\ p = \lfloor v \rfloor \qquad \mathsf{export\text{-}uinfo}\ t = \mathsf{TYPE}(\beta)_\nu \qquad \mathsf{guard\text{-}mono}\ g\ g'}{\mathsf{lift\text{-}typ\text{-}heap}\ g'\ s\ (\mathsf{Ptr}\ \&(p \rightarrow f)) = \lfloor \mathsf{from\text{-}bytes}\ (\mathsf{access\text{-}ti}_0\ t\ v) \rfloor}$$

where $p::\alpha::$*mem-type ptr and* $\mathsf{Ptr}\ \&(p \rightarrow f)::\beta::$*mem-type ptr.*

Proof. Thm. 5.3.1 provides field monotonicity for validity. It is left for us to show $\mathsf{from\text{-}bytes}\ (\mathsf{heap\text{-}list\text{-}s}\ s\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\beta))\ \&(p \rightarrow f)) = \mathsf{from\text{-}bytes}\ (\mathsf{access\text{-}ti}_0\ t\ (\mathsf{from\text{-}bytes}\ (\mathsf{heap\text{-}list\text{-}s}\ s\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha))\ p_\&)))$. This can be achieved with Thm. 5.2.6 and Thm. 5.2.5. □

Corollary. *The property in Thm. 5.3.6 also applies with* lift_τ*:*

$$\frac{\mathsf{TYPE}(\alpha)_\tau \triangleright f = \lfloor (t,\ n) \rfloor \qquad \mathsf{lift}_\tau\ g\ s\ p = \lfloor v \rfloor \qquad \mathsf{export\text{-}uinfo}\ t = \mathsf{TYPE}(\beta)_\nu \qquad \mathsf{guard\text{-}mono}\ g\ g'}{\mathsf{lift}_\tau\ g'\ s\ (\mathsf{Ptr}\ \&(p \rightarrow f)) = \lfloor \mathsf{from\text{-}bytes}\ (\mathsf{access\text{-}ti}_0\ t\ v) \rfloor}$$

5.3.3 Update dependency order

At the end of §5.1 it was clear that the effects of heap updates on typed heaps depended on the structural relationship between types. In this section we formalise this notion, allowing update rules in the next section to distinguish between cases of this relation.

Definition 5.3.5. An order can be defined on type descriptions that expresses the update dependency between heaps::

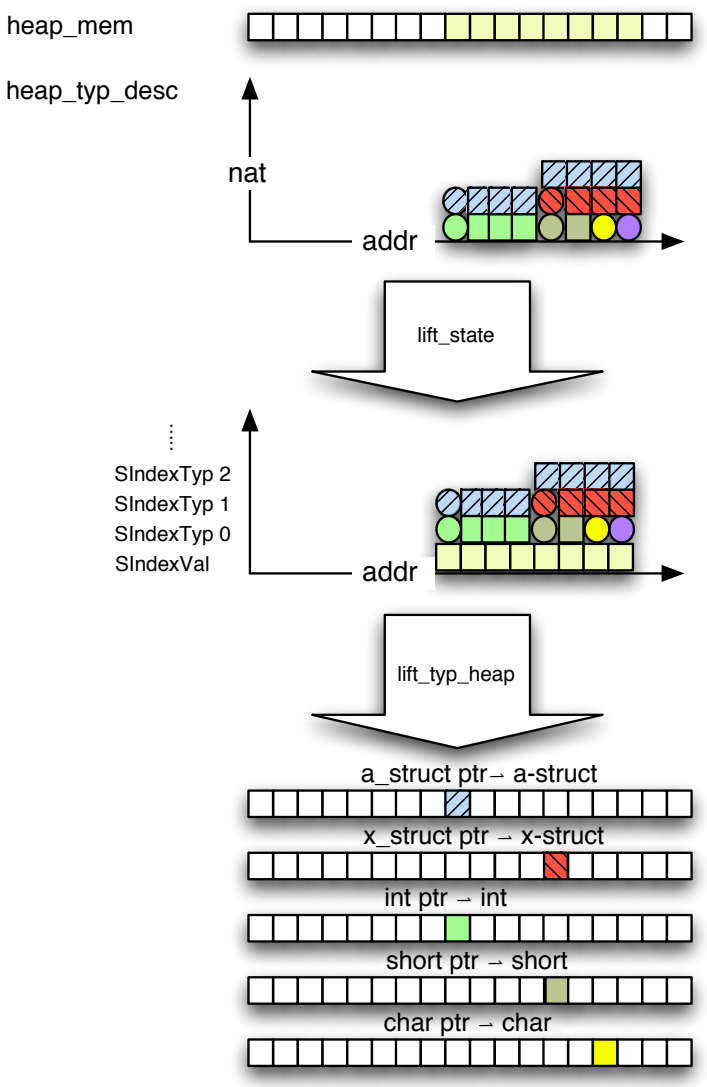

Figure 5.7: Two-stage lifting.

$$s \leq t \equiv \exists\, n.\ (s,\ n) \in \mathsf{td\text{-}set}\ t$$

This can be lifted to a predicate on α::*c-type itself* and β::*c-type itself*s:

$$\mathsf{TYPE}(\alpha) \leq_\tau \mathsf{TYPE}(\beta) \equiv \mathsf{TYPE}(\alpha)_\nu \leq \mathsf{TYPE}(\beta)_\nu$$

Example 5.3.1. Using the running example, it can be easily observed that $\mathsf{TYPE}(\textit{x-struct}) <_\tau \mathsf{TYPE}(\textit{a-struct})$ and $\mathsf{TYPE}(\textit{word32}) <_\tau \mathsf{TYPE}(\textit{a-struct})$. An update to an *a-struct* will always affect the lifted *int* heap, but an update of an *x-struct* will only sometimes affect the lifted *a-struct* heap.

Theorem 5.3.7. $\leq$ *is a partial order:*

$$s \leq s \qquad \frac{s \leq t \qquad t \leq s}{s = t} \qquad \frac{s \leq t \qquad t \leq u}{s \leq u}$$

Proof. Reflexivity is trivial. Antisymmetry can be shown with $(s,\ n) \in \mathsf{td\text{-}set\text{-}offset}\ t\ m \Longrightarrow \textit{size}\ s = \textit{size}\ t \wedge s = t \wedge n = m \vee \textit{size}\ s < \textit{size}\ t$, by structural induction. Transitivity is given by $(s,\ n) \in \mathsf{td\text{-}set\text{-}offset}\ t\ m \Longrightarrow \mathsf{td\text{-}set\text{-}offset}\ s\ n \subseteq \mathsf{td\text{-}set\text{-}offset}\ t\ m$, also derivable by structural induction. □

5.3.4 Generalised rewrites

In this section we develop rewrites that allow the effects of updates on lifted typed heaps to be evaluated, generalising the results in §3.4. First we present some auxiliary definitions and then the key theorems, Thm. 5.3.8 and Thm. 5.3.10. These theorems have the form of conditional rewrites, but require some additional support to be efficiently applicable, so are followed by this detail.

Definition 5.3.6. [B.13] A list of names of all fields matching an exported type information can be obtained with $\mathsf{field\text{-}names} :: \alpha\ \textit{typ-info} \Rightarrow \textit{typ-uinfo} \Rightarrow \textit{qualified-field-name list}$. E.g. $\mathsf{field\text{-}names}\ \mathsf{TYPE}(\textit{a-struct})_\tau\ \mathsf{TYPE}(\textit{word16})_\nu = [[''c'',\ ''y'']]$.

Definition 5.3.7. From td-set, a predicate may be derived that checks whether a given pointer p::α *ptr* is to a field of a structured type with base q::β *ptr*:

$$\mathsf{field\text{-}of}\ p\ q \equiv (\mathsf{TYPE}(\alpha)_\nu,\ \mathbb{N}^{\Leftarrow}\ (p_\& - q_\&)) \in \mathsf{td\text{-}set}\ \mathsf{TYPE}(\beta)_\nu$$

Definition 5.3.8. From lookup, functions may be derived that provide the first and second components of the result for a valid qualified field name:

$$\begin{array}{lcl} \mathsf{field\text{-}typ}\ \mathsf{TYPE}(\alpha)\ n & \equiv & \mathsf{fst}\ (\mathsf{the}\ (\mathsf{TYPE}(\alpha)_\tau \rhd n)) \\ \mathsf{field\text{-}offset}\ \mathsf{TYPE}(\alpha)\ n & \equiv & \mathsf{snd}\ (\mathsf{the}\ (\mathsf{TYPE}(\alpha)_\nu \rhd n)) \end{array}$$

We now give the rule for β heaps when an α update occurs and $\mathsf{TYPE}(\alpha) \leq_\tau \mathsf{TYPE}(\beta)$. The intuition here is that locations that are not the base of valid β pointers or where the α *ptr* does not correspond to a field of β are unaffected. When the update pointer does correspond to a field of β, we traverse the α fields of the enclosing β, looking for a field offset that matches the difference between the enclosing pointer base and p. When found, the field's update function is applied.

Theorem 5.3.8. *The lifted β heap following an update of a valid α ptr p, where α is a sub-type of β is given by:*

$$\frac{d, g' \models_t p \qquad \mathsf{TYPE}(\alpha) \leq_\tau \mathsf{TYPE}(\beta)}{\mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d) = \mathsf{super\text{-}field\text{-}update}\ p\ v\ (\mathsf{lift}_\tau\ g\ (h,\ d))}$$

where

$$\begin{array}{l} \mathsf{super\text{-}field\text{-}update}\ p\ v\ s \equiv \\ \lambda q.\ \mathsf{if}\ \mathsf{field\text{-}of}\ p\ q \\ \quad \mathsf{then}\ \mathsf{case}\ s\ q\ \mathsf{of}\ \bot \Rightarrow \bot \\ \qquad \mid \lfloor w \rfloor \Rightarrow \\ \qquad\quad \lfloor \mathsf{update\text{-}value}\ (\mathsf{field\text{-}names}\ \mathsf{TYPE}(\beta)_\tau\ \mathsf{TYPE}(\alpha)_\nu)\ v\ w \\ \qquad\quad\ (\mathbb{N}^{\Leftarrow}\ (p_\& - q_\&)) \rfloor \\ \quad \mathsf{else}\ s\ q \end{array}$$

$$\begin{array}{lcl} \mathsf{update\text{-}value}\ []\ v\ w\ x & \equiv & w \\ \mathsf{update\text{-}value}\ (f{\cdot}fs)\ v\ w\ x & \equiv & \mathsf{if}\ x = \mathsf{field\text{-}offset}\ \mathsf{TYPE}(\beta)\ f \\ & & \mathsf{then}\ \mathsf{update\text{-}ti}\ (\mathsf{field\text{-}typ}\ \mathsf{TYPE}(\beta)\ f)\ (\mathsf{to\text{-}bytes}_0\ v)\ w \\ & & \mathsf{else}\ \mathsf{update\text{-}value}\ fs\ v\ w\ x \end{array}$$

Proof. Equality of the two heaps can be shown with extensionality and unfolding of super-field-update, letting the pointer be called q. Expand lift_τ terms with Thm. 3.3.4 and it is easy to see that locations without valid β *ptr*s remain unchanged as $\bot$. Locations corresponding to valid pointers can be shown to contain values that are equivalent by case splitting on whether the update pointer p is a field of the value at q.

When field-of p q, on the LHS the representation of the raw updated heap value at q may be considered to consist of 3 parts, those bytes before $p_\& - q_\&$, those following, corresponding to the representation of v, and those remaining. The from-bytes inside the lift_τ gives rise to an update-ti term on the LHS, which can then be seen to be the same as the original value, obtained from the byte representation with from-bytes/update-ti, after an update-ti with v's representation, using the rule:

$$\frac{\begin{array}{c}t{\triangleright}f = \lfloor(s,\ n)\rfloor \\ \text{wf-field-desc } t \qquad \text{wf-desc } t \qquad |bs| = \text{size-td } t \qquad |v| = \text{size-td } s\end{array}}{\begin{array}{c}\text{update-ti } t\ (\text{take } n\ bs\ @\ v\ @\ \text{drop } (n + |v|)\ bs)\ w = \\ \text{update-ti } s\ v\ (\text{update-ti } t\ bs\ w)\end{array}}$$

obtained by structural induction and list fragment reasoning. On the RHS, the update-value can be transformed to the same form — this can be shown by induction on the list of field names supplied to update-value.

If $\neg$ field-of $p\ q$ then there is a further case split on $\mathsf{TYPE}(\alpha)_\nu = \mathsf{TYPE}(\beta)_\nu$. If the types are the same then the treatment is similar to Lem. 3.4.4. Otherwise, we can generalise Thm. 3.2.2 as:

$$\frac{d,g \models_t p \qquad d,g' \models_t q \qquad \neg\ \mathsf{TYPE}(\beta) <_\tau \mathsf{TYPE}(\alpha) \qquad \neg\ \text{field-of } p\ q}{\{p_{\&}..+\text{size-of } \mathsf{TYPE}(\alpha)\} \cap \{q_{\&}..+\text{size-of } \mathsf{TYPE}(\beta)\} = \emptyset}$$

using the following rules:

$$\frac{\text{valid-footprint } d\ p\ s \qquad \text{valid-footprint } d\ q\ t \qquad \neg\ t < s}{p \in \{q..+\text{size-td } t\} \longrightarrow (s,\ \mathbb{N}^{\Leftarrow}\ (p - q)) \in \text{td-set } t}$$

$$\frac{\text{valid-footprint } d\ p\ s \qquad \text{valid-footprint } d\ q\ t \qquad \neg\ t < s}{q \notin \{p..+\text{size-td } s\} \vee p = q}$$

and Lem. 3.4.3. □

While Thm. 5.3.8 gives a conditional rewrite that allows an update to be lifted to the typed heap level of §5.3.2, making use of the updated typed heap could involve unfolding this complex definition in general. However, additional rewrites can be given for well-behaved updates.

Theorem 5.3.9. *For a valid qualified field name f, a* super-field-update *for a pointer* Ptr $(\&(p{\rightarrow}f))::\alpha::$*mem-type ptr, where $p::\beta::$mem-type ptr can be reduced to the field update obtained from the type information:*

$$\frac{\mathsf{TYPE}(\beta)_\tau{\triangleright}f = \lfloor(s,\ n)\rfloor \qquad \text{lift}_\tau\ g\ h\ p = \lfloor w\rfloor \qquad \mathsf{TYPE}(\alpha)_\nu = \text{export-uinfo } s}{\begin{array}{c}\text{super-field-update } (\text{Ptr } \&(p{\rightarrow}f))\ v\ (\text{lift}_\tau\ g\ h) = \\ \text{lift}_\tau\ g\ h(p \mapsto \text{update-ti } s\ (\text{to-bytes}_0\ v)\ w)\end{array}}$$

Proof. field-of (Ptr $\&(p{\rightarrow}f)$) p holds from the assumption that f is a valid qualified field name. Again, applying extensionality and unfolding the definition of super-field-update, letting the pointer be called q, gives two cases to consider when the pointers are valid.

When $p = q$, the LHS can be reduced to the intended field update as in the proof of Thm. 5.3.8. In the case when $p \neq q$, then $\neg$ field-of (Ptr $\&(p{\rightarrow}f)$) q, since p and q are valid pointers of the same type and hence may not overlap. This then leaves both the LHS and RHS heaps at q unchanged. □

As detailed in §5.2.5, the lookup side-condition can be resolved without having to unfold the type information definition using field specific rewrites installed during type information construction at the ML level. The update-ti is also rewritten to an Isabelle/HOL **record** field update function.

Example 5.3.2. For a safe update at the next field for a **struct**:

$$\frac{\mathsf{lift}_\tau\ g\ s\ p = \lfloor w \rfloor}{\begin{array}{c}\mathsf{super\text{-}field\text{-}update}\ (\mathsf{Ptr}\ \&(p{\rightarrow}[''next''])\ v\ (\mathsf{lift}_\tau\ g\ s)\\ = \mathsf{lift}_\tau\ g\ s(p \mapsto w(\!|\mathsf{next} := v|\!))\end{array}}$$

A rewrite can also be given for the two remaining cases, where $\mathsf{TYPE}(\beta) <_\tau \mathsf{TYPE}(\alpha)$ or $\mathsf{TYPE}(\alpha) \perp_\tau \mathsf{TYPE}(\beta)$. This may involve no updates if the types are disjoint, or several updates of the β heap when α has multiple fields of type β. The heap update function sub-field-update takes a list of all such fields and applies an update at each.

Theorem 5.3.10. *The lifted β heap following an update of a valid α ptr p, where α is not a sub-type of β is given by:*

$$\frac{d,g' \models_t p \qquad \neg\ \mathsf{TYPE}(\alpha) <_\tau \mathsf{TYPE}(\beta)}{\begin{array}{c}\mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d) =\\ \mathsf{sub\text{-}field\text{-}update}\ (\mathsf{field\text{-}names}\ \mathsf{TYPE}(\alpha)_\tau\ \mathsf{TYPE}(\beta)_\nu)\ p\ v\ (\mathsf{lift}_\tau\ g\ (h,\ d))\end{array}}$$

where

$$\begin{array}{lcl}\mathsf{sub\text{-}field\text{-}update}\ []\ p\ v\ s & \equiv & s\\ \mathsf{sub\text{-}field\text{-}update}\ (f{\cdot}fs)\ p\ v\ s & \equiv & (\mathsf{let}\ s' = \mathsf{sub\text{-}field\text{-}update}\ fs\ p\ v\ s\\ & & \ \mathsf{in}\ s'(\mathsf{Ptr}\ \&(p{\rightarrow}f) \mapsto\\ & & \quad \mathsf{from\text{-}bytes}\\ & & \quad (\mathsf{access\text{-}ti}_0\\ & & \quad\ (\mathsf{field\text{-}typ}\ \mathsf{TYPE}(\alpha)\ f)\ v)))\restriction_{\mathsf{dom}\ s}\end{array}$$

Proof. This can be proven by induction on the list of field names. To do this we first strengthen the induction hypothesis:

$$\frac{\begin{array}{cc}d,g' \models_t p & \\ \neg\ \mathsf{TYPE}(\alpha) <_\tau \mathsf{TYPE}(\beta) & \mathsf{set}\ fs \subseteq \mathsf{set}\ (\mathsf{field\text{-}names}\ \mathsf{TYPE}(\alpha)_\tau\ \mathsf{TYPE}(\beta)_\nu)\\ K = \mathcal{U} - (\mathsf{field\text{-}ptrs}\ p\ (\mathsf{field\text{-}names}\ \mathsf{TYPE}(\alpha)_\tau\ \mathsf{TYPE}(\beta)_\nu) - \mathsf{field\text{-}ptrs}\ p\ fs) & \end{array}}{\mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d)\restriction_K = \mathsf{sub\text{-}field\text{-}update}\ fs\ p\ v\ (\mathsf{lift}_\tau\ g\ (h,\ d))\restriction_K}$$

where

$$\mathsf{field\text{-}ptrs}\ p\ fs \equiv \{\mathsf{Ptr}\ \&(p{\rightarrow}f) \mid f \in \mathsf{set}\ fs\}$$

The heaps are again compared pointwise, but with the mask K hiding those β *ptr*s affected by the update yet not present in the field names supplied to sub-field-update.

In the base case, where $fs = []$, we can use a generalised Lem. 3.4.4:

$$\frac{d,g \models_t p \qquad d,g' \models_t q \qquad \neg\ \mathsf{TYPE}(\alpha) <_\tau \mathsf{TYPE}(\beta) \qquad \neg\ \mathsf{field\text{-}of}\ q\ p}{\mathsf{h\text{-}val}\ (\mathsf{heap\text{-}update}\ p\ v\ h)\ q = \mathsf{h\text{-}val}\ h\ q}$$

to give equality of the values at unmasked locations with valid pointers. The field-of condition is discharged by K covering all relevant fields.

In the inductive case, where $fs = y{\cdot}ys$, we can perform a case split on both $\mathsf{Ptr}\ \&(p{\rightarrow}y) \in \mathsf{dom}\ (\mathsf{lift}_\tau\ g\ (h,\ d))$ and $q = \mathsf{Ptr}\ \&(p{\rightarrow}y)$, where q is the point being considered with extensionality:

- When $\mathsf{Ptr}\ \&(p{\rightarrow}y) \in \mathsf{dom}\ (\mathsf{lift}_\tau\ g\ (h,\ d)) \wedge q = \mathsf{Ptr}\ \&(p{\rightarrow}y)$ — the update at this location needs to be shown to be equivalent to that given by sub-field-update. This is done by simplifying the RHS update for y with:

$$\frac{\begin{array}{c}\mathsf{TYPE}(\alpha)_\tau{\triangleright}f = \lfloor(s,\ n)\rfloor \\ bs = \mathsf{to\text{-}bytes}\ v\ (\mathsf{heap\text{-}list}\ h\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha))\ p_{\&})\end{array}}{\mathsf{access\text{-}ti}_0\ s\ v = \mathsf{norm\text{-}tu}\ (\mathsf{export\text{-}uinfo}\ s)\ (\mathsf{take}\ (\mathsf{size\text{-}td}\ s)\ (\mathsf{drop}\ n\ bs))}$$

  and the LHS with:

$$\frac{n + x \leq |v| \wedge |v| < |\mathsf{addr}|}{\mathsf{heap\text{-}list}\ (\mathsf{heap\text{-}update\text{-}list}\ p\ v\ h)\ n\ (p +_{\mathbb{N}^\Rightarrow} x) = \mathsf{take}\ n\ (\mathsf{drop}\ x\ v)}$$

  Since the comparison after unfolding the lift_τ is at the typed level, after applying from-bytes, Thm. 5.2.5 can be used to complete this case.

- When $\mathsf{Ptr}\ \&(p{\rightarrow}y) \in \mathsf{dom}\ (\mathsf{lift}_\tau\ g\ (h,\ d)) \wedge q \neq \mathsf{Ptr}\ \&(p{\rightarrow}y)$ — we can use the inductive hypothesis.

- When $\mathsf{Ptr}\ \&(p{\rightarrow}y) \notin \mathsf{dom}\ (\mathsf{lift}_\tau\ g\ (h,\ d)) \wedge q = \mathsf{Ptr}\ \&(p{\rightarrow}y)$ — then q is not in the domain of the LHS and the domain restriction in the inductive case of sub-field-update removes this from the domain of the RHS.

- When $\mathsf{Ptr}\ \&(p{\rightarrow}y) \notin \mathsf{dom}\ (\mathsf{lift}_\tau\ g\ (h,\ d)) \wedge q \neq \mathsf{Ptr}\ \&(p{\rightarrow}y)$ — we can use the inductive hypothesis.

□

A sub-field-update version of Thm. 5.3.9 is not as easy to state, as the β heap will be updated at multiple locations. Inter-type framing is hence not as reasonable to handle as in §3.5. This motivates strongly the use of separation logic when reasoning about programs with structured types.

5.3.5 Non-interference

Theorem 5.3.11. *The rewrites for an update to a lifted typed heap through a valid pointer of the same type, or a disjoint type are the same as in Thm. 3.4.5 and Thm. 3.4.6:*

$$\frac{d,g \models_t p}{\mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d) = \mathsf{lift}_\tau\ g\ (h,\ d)(p \mapsto v)}$$

$$\frac{d,g' \models_t p \qquad \mathsf{TYPE}(\alpha)_\nu \perp_t \mathsf{TYPE}(\beta)_\nu}{\mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d) = \mathsf{lift}_\tau\ g\ (h,\ d)}$$

Proof. By Thm. 5.3.10 and reducing the field-names term with field-names ti (export-uinfo ti) = [[]] and $\mathsf{TYPE}(\alpha)_\nu \perp_t \mathsf{TYPE}(\beta)_\nu \Longrightarrow$ field-names $\mathsf{TYPE}(\beta)_\tau$ $\mathsf{TYPE}(\alpha)_\nu$ = [], respectively. □

Bornat [14] describes multiple independent heaps based on distinct field names. Updates through a pointer dereference to a specific field only affect that heap. This does not work directly in the presence of the $\&(p{\rightarrow}f)$ operator and address arithmetic. However, the following can be shown:

Theorem 5.3.12. *When the base pointers are of the same type β, and neither of the field names is a prefix of the other, updates through an α pointer derived from one field do not affect a value in the γ lifted heap at the other:*

$$\frac{\begin{array}{ccc} d,g' \models_t p & d,g'' \models_t q & \mathsf{TYPE}(\beta)_\tau \triangleright f = \lfloor (s,\ m) \rfloor \\ \mathsf{TYPE}(\beta)_\tau \triangleright f' = \lfloor (t,\ n) \rfloor & & \mathsf{size\text{-}td}\ s = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) \\ \mathsf{size\text{-}td}\ t = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\gamma) & \neg f \leq f' & \neg f' \leq f \end{array}}{\begin{array}{c} \mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ (\mathsf{Ptr}\ \&(p{\rightarrow}f))\ v\ h,\ d)\ (\mathsf{Ptr}\ \&(q{\rightarrow}f')) = \\ \mathsf{lift}_\tau\ g\ (h,\ d)\ (\mathsf{Ptr}\ \&(q{\rightarrow}f')) \end{array}}$$

Proof. Unfold definitions and then use Lem. 3.4.3. Disjointness of the two field heap footprints can be found by case splitting on $p_\& = q_\&$. If they match then field disjointness is given by structural induction on the common type description. Otherwise, the two valid pointers have disjoint footprints, field footprints derived from the pointers will be subsets of the base pointer footprints and hence disjoint. □

5.4 Structured separation logic

We concluded §5.3.4 with the observation that the use of separation logic when reasoning about structured types is well motivated, since even inter-type aliasing can be difficult to reason about in the multiple typed heaps abstraction with first-class structured types.

In this section we describe how the shallow embedding of separation logic in Chapter 4 can be extended to structured types. We first describe the heap state model and shallow embedding, where the focus is on the singleton heap assertion $p \mapsto_g v$, and several variants, as other definitions and properties remain mostly unchanged. The singleton heap assertion has new properties of interest for structured types. In particular, we are able to decompose singleton mapping assertions to reason independently about field mapping assertions.

Following this, generalisation of proof obligation lifting is given, and in the next section we again revisit the separation logic in-place list reversal example, armed with the development of this section.

5.4.1 Domain

We maintain the model of separation assertions as predicates on *heap-state*s, applied in assertions of the verification environment to the result of the first lifting stage of §5.3.2. Here, *heap-state* predicates can also be seen as the specialisation (*addr* × *s-heap-index*, *s-heap-value*) *map-assert* of the (α, β) *map-assert* state in §4.2.1.

The rationale for this choice of domain is that it allows for more expressive separation assertions than are possible with simpler models. From the earlier intermediate state, *addr* ⇀ *typ-tag option* × *byte* for unstructured types, a naive extension might be something like *addr* ⇀ *typ-uinfo list* × *byte*. Unfortunately, this does not allow for two assertions separated by $\wedge^*$ to refer to distinct type information levels at the same address, necessary to provide flexible rules for retyping and unfolding. Hence we have a two-dimensional address space in *heap-state*s, with the first component providing the physical address and the second the type index.

Example 5.4.1. Ignoring padding, we would expect that $(p \mapsto (\!|\, \mathsf{y} = 3, \mathsf{z} = 'r' \,|\!)) = (\mathsf{Ptr}\ (\&(p{\rightarrow}[''y'']))) \mapsto 3) \wedge^* (\mathsf{Ptr}\ (\&(p{\rightarrow}[''z'']))) \mapsto 'r') \wedge^* \mathsf{typ\text{-}outline}\ p$, where $\mathsf{typ\text{-}outline}\ p$ contains the root type information for the enclosing structure. By adding a type level index to the domain of the *heap-state* we are able to write $\mathsf{typ\text{-}outline}\ p$ separate to $(\mathsf{Ptr}\ (\&(p{\rightarrow}[''y'']))) \mapsto 3)$ and hence reason about the y field independently.

5.4.2 Shallow embedding

Definition 5.4.1. The s-footprint::α::*c-type ptr* ⇒ *s-addr set* gives a set of addresses inside a pointer's *heap-state* footprint:

$$\mathsf{s\text{-}footprint\text{-}untyped}\ p\ t \equiv \{(p + \mathbb{N}^{\Rightarrow}\ x, \mathsf{SIndexVal}) \mid x < \mathsf{size\text{-}td}\ t\} \cup \{(p + \mathbb{N}^{\Rightarrow}\ x, \mathsf{SIndexTyp}\ n) \mid x < \mathsf{size\text{-}td}\ t \wedge n < |\mathsf{typ\text{-}slice}\ t\ x|\}$$

$$\mathsf{s\text{-}footprint}\ (p{::}\alpha\ ptr) \equiv \mathsf{s\text{-}footprint\text{-}untyped}\ p_{\&}\ \mathsf{TYPE}(\alpha)_{\nu}$$

Definition 5.4.2. $p \mapsto_g v$ asserts that the heap contains exactly one mapping matching the guard g, at the location given by pointer p to value v:

$$p \mapsto_g v \equiv$$
$$\lambda s.\ \mathsf{lift\text{-}typ\text{-}heap}\ g\ s\ p = \lfloor v \rfloor \wedge \mathsf{dom}\ s = \mathsf{s\text{-}footprint}\ p \wedge \mathsf{wf\text{-}heap\text{-}val}\ s$$

wf-heap-val states that the type, SValue or STyp, of a value in the *heap-state*, if present, matches the type of the index, SIndexVal or SIndexTyp respectively.

Definition 5.4.3. Defn. 4.3.1 can be similarly extended to the new definition of *heap-state*:

$$g \vdash_s p \equiv \lambda s.\ s, g \models_s p \wedge \mathsf{dom}\ s = \mathsf{s\text{-}footprint}\ p$$

The rest of the definitions, Defn. 4.2.1, Defn. 4.2.3 and Defn. 4.2.4, in §4.2.1 remain unchanged.

5.4.3 Properties

The properties in §4.2.2 continue to hold. Proofs of properties not involving the singleton mapping assertions are the same, and those that do involve the assertion can be generalised in a straightforward manner.

The frame rule of §4.4 also still applies in this development. The approach to showing this is the same, however we have to change ptr-safe to reflect the extended heap type description, i.e.:

$$\mathsf{ptr\text{-}safe}\ p\ d \equiv$$
$$\mathsf{s\text{-}footprint}\ p$$
$$\subseteq \{(x, \mathsf{SIndexVal}) \mid \mathsf{fst}\ (d\ x)\} \cup \{(x, \mathsf{SIndexTyp}\ n) \mid \mathsf{snd}\ (d\ x)\ n \neq \bot\}$$

The structure of and other definitions used in the proof are similar, with the main change being domain restriction now operating on a tuple space.

5.4.4 Unfolding

Inside a proof it may be necessary or helpful to extract the mapping assertions of individual fields from a mapping assertion for a structured value. For example, a function call that has field references as parameters, with a specification unaware of the enclosing structure, will have a proof obligation demanding this. The field monotonicity given in Thm. 5.3.6 hints at this being possible with Defn. 5.4.2, and in this section we provide the rules to accomplish this.

Exmp. 5.4.1 gives a "complete" unfolding of the outer structure for a value. This is generally not all that useful, for two reasons. First, when the structure contains padding fields they need to also be expanded as mapping assertions, since padding has no special treatment other than at type information construction time. These clutter the proof state and do not aid in advancing towards the goal. The same applies to fields that do not need to be unfolded for a proof. The second problem with this approach is that later in a proof one might want to take fields that have been updated and independently reasoned about after unfolding, and fold them back together to resume reasoning at the granularity of the structured value. While it is not too difficult to do a complete unfolding with rewriting, this is harder in the opposite direction.

To avoid these problems, instead of complete unfolding we give rules to unfold and fold individual, potentially nested, fields. To do so, we make use of a new separation predicate, called a *masked mapping assertion*, that allows us to express the existence of a structured mapping assertion sans a set of fields that have been extracted through unfolding. Defining masked mapping requires first several auxiliary definitions.

Definition 5.4.4. The singleton state in Defn. 4.3.2 is revised for the extended state space as:

$$\begin{array}{l}\mathsf{singleton}\ p\ v \equiv \\ \mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p\ v\ (\lambda x.\ 0),\ \mathsf{ptr\text{-}retyp}\ p\ (\lambda x.\ (\mathsf{False}, \mathsf{empty})))\end{array}$$

Lemma 5.4.1. *The domain of a singleton state is given by:*

$$\mathsf{dom}\ (\mathsf{singleton}\ p\ v) = \mathsf{s\text{-}footprint}\ p$$

Proof. Examining the lift-state definition, the domain is determined by the heap type description. Lem. 5.3.3 gives the existence of mappings inside the footprint, and the absence of mappings outside can be shown with:

$$\frac{x \notin \{p_{\&}..{+}\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)\}}{\mathsf{ptr\text{-}retyp}\ p\ (\lambda x.\ (\mathsf{False}, \mathsf{empty}))\ x = (\mathsf{False}, \mathsf{empty})}$$

which follows from Defn. 5.3.3 and induction on the list.

□

Definition 5.4.5. The set of all valid qualified field names for a type is given by:

$$\mathsf{fields}\ \mathsf{TYPE}(\alpha) \equiv \{f \mid \mathsf{TYPE}(\alpha)_\tau \triangleright f \neq \bot\}$$

Definition 5.4.6. The footprint for a set of qualified field names for a type α, with respect to a base pointer $p{::}\alpha\ ptr$, is given by:

$$\begin{array}{l}\mathsf{fs\text{-}footprint}\ p\ F \equiv \\ \bigcup \{\mathsf{s\text{-}footprint\text{-}untyped}\ (p_{\&} + \mathbb{N}^{\Rightarrow}\ (\mathsf{field\text{-}offset}\ \mathsf{TYPE}(\alpha)\ f)) \\ \quad (\mathsf{export\text{-}uinfo}\ (\mathsf{field\text{-}typ}\ \mathsf{TYPE}(\alpha)\ f)) \mid f \in F\}\end{array}$$

Lemma 5.4.2. *The footprint for a subset of valid qualified field names for a type α is a subset of a base pointer $p{::}\alpha$ ptr's footprint:*

$$\frac{F \subseteq \mathsf{fields}\ \mathsf{TYPE}(\alpha)}{\mathsf{fs\text{-}footprint}\ p\ F \subseteq \mathsf{s\text{-}footprint}\ p}$$

Proof. Each field can be shown to be contained by s-footprint p with the following, derivable directly from definitions:

$$\frac{\mathsf{TYPE}(\alpha)_{\tau} \triangleright f = \lfloor (s,\ n) \rfloor}{\mathsf{s\text{-}footprint\text{-}untyped}\ \&(p \rightarrow f)\ (\mathsf{export\text{-}uinfo}\ s) \subseteq \mathsf{s\text{-}footprint}\ p}$$

□

Definition 5.4.7. $p \mapsto_g{}^F\ v$ asserts that the heap contains exactly one mapping matching the guard g, at the location given by pointer $p{::}\alpha\ ptr$ to value v, with the set of valid fields F masked:

$$\begin{array}{l}p \mapsto_g{}^F\ v \equiv \\ \lambda s.\ \mathsf{lift\text{-}typ\text{-}heap}\ g\ (\mathsf{singleton}\ p\ v\ +\!\!+\ s)\ p = \lfloor v \rfloor \wedge \\ \quad F \subseteq \mathsf{fields}\ \mathsf{TYPE}(\alpha) \wedge \\ \quad \mathsf{dom}\ s = \mathsf{s\text{-}footprint}\ p - \mathsf{fs\text{-}footprint}\ p\ F \wedge \mathsf{wf\text{-}heap\text{-}val}\ s\end{array}$$

The footprint of this assertion excludes the masked fields, and the lifted value has the expected value for the masked fields supplied by singleton.

Example 5.4.2. Fig. 5.8 gives a state where $p \mapsto_g{}^{\{[''c'',\ ''y'']\}}\ v$ holds.

Theorem 5.4.3. *A masked mapping assertion with an empty set of fields is equivalent to a singleton mapping assertion:*

$$p \mapsto_g v = p \mapsto_g{}^{\emptyset}\ v$$

Proof. From the definitions, $F = \emptyset$ hence fs-footprint $p\ F = \emptyset$. s then covers the domain of singleton $p\ v$ as a result of Lem. 5.4.1, giving singleton $p\ v\ +\!\!+ s = s$. □

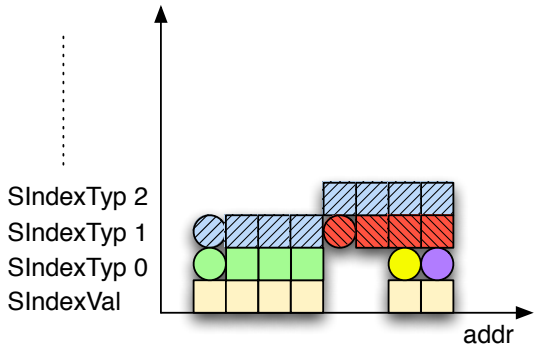

Figure 5.8: Example *heap-state* for a masked mapping assertion.

From a masked mapping assertion, a valid field may be extracted, providing the qualified field name is not in a prefix relation with any member of F. Intuitively this is reasonable, as if the field is inside another that has already been extracted or covers the same footprint then it will not be possible to partition the state of the masked mapping assertion as required for unfolding.

Example 5.4.3. The $[''c'', ''y'']$ field can be independently extracted when $[''b'']$ is masked, but not if $[''c'']$ is masked.

Definition 5.4.8. A qualified field name is said to be disjoint from a set of qualified field names with:

$$\mathsf{disjoint\text{-}fn}\ f\ F \equiv \forall f' \in F.\ \neg\ f \leq f' \wedge \neg\ f' \leq f$$

Lemma 5.4.4. *A field disjoint from a set of valid qualified field names has a footprint disjoint from the set's footprint:*

$$\frac{\mathsf{disjoint\text{-}fn}\ f\ F \qquad F \subseteq \mathsf{fields}\ \mathsf{TYPE}(\alpha) \qquad \mathsf{TYPE}(\alpha)_\tau \triangleright f = \lfloor (t,\ n) \rfloor}{\mathsf{fs\text{-}footprint}\ p\ F \cap \mathsf{fs\text{-}footprint}\ p\ \{f\} = \emptyset}$$

Proof. For each field in the set, the following rule derivable by structural induction on the type information gives disjointness:

$$\frac{\begin{array}{c}\mathsf{lookup}\ t\ f\ m = \lfloor (d,\ n) \rfloor \qquad \mathsf{lookup}\ t\ f'\ m = \lfloor (d',\ n') \rfloor \\ \neg\ f \leq f' \wedge \neg\ f' \leq f \qquad \mathsf{wf\text{-}field\text{-}desc}\ t \qquad \mathsf{wf\text{-}desc}\ t \qquad \mathsf{size\text{-}td}\ t < |\mathsf{addr}|\end{array}}{\{\mathbb{N}^{\Rightarrow}\ n..{+}\mathsf{size\text{-}td}\ d\} \cap \{\mathbb{N}^{\Rightarrow}\ n'..{+}\mathsf{size\text{-}td}\ d'\} = \emptyset}$$

□

The unfolding and folding rules can now be given.

Theorem 5.4.5. *A valid disjoint field may be unfolded from a masked mapping assertion with:*

$$\frac{\mathsf{TYPE}(\alpha)_\tau \triangleright f = \lfloor (t,\ n) \rfloor \qquad \mathsf{disjoint\text{-}fn}\ f\ F \qquad \mathsf{guard\text{-}mono}\ g\ g' \qquad \mathsf{export\text{-}uinfo}\ t = \mathsf{TYPE}(\beta)_\nu}{p \mapsto_g{}^{F}\ v = p \mapsto_g{}^{(\{f\}\ \cup\ F)}\ v\ \wedge^*\ \mathsf{Ptr}\ \&(p{\rightarrow}f) \mapsto_g{}'\ \mathsf{from\text{-}bytes}\ (\mathsf{access\text{-}ti}_0\ t\ v)}$$

where p*::*α*::mem-type ptr and* $\mathsf{Ptr}\ \&(p{\rightarrow}f)$*::*$\beta$*::mem-type ptr.*

Proof. Equality of the LHS and RHS assertions can be shown with extensionality, letting the *heap-state* be s.

First we show that if the LHS holds on s then the RHS also holds on s. The state can be partitioned as two states $s\restriction_{(\mathsf{dom}\ s\ -\ \mathsf{fs\text{-}footprint}\ p\ \{f\})}$ and $s\restriction_{\mathsf{fs\text{-}footprint}\ p\ \{f\}}$. The singleton mapping assertion can then be shown to hold on its partitioned state with:

$$\frac{\mathsf{TYPE}(\alpha)_\tau \triangleright f = \lfloor (t,\ n) \rfloor \qquad \mathsf{disjoint\text{-}fn}\ f\ F \qquad \mathsf{guard\text{-}mono}\ g\ g' \qquad \mathsf{export\text{-}uinfo}\ t = \mathsf{TYPE}(\beta)_\nu \qquad (p \mapsto_g{}^{F}\ v)\ s}{(\mathsf{Ptr}\ \&(p{\rightarrow}f) \mapsto_g{}'\ \mathsf{from\text{-}bytes}\ (\mathsf{access\text{-}ti}_0\ t\ v))\ (s\restriction_{\mathsf{fs\text{-}footprint}\ p\ \{f\}})}$$

which is a result of Thm. 5.3.6 and $\mathsf{fs\text{-}footprint}\ p\ \{f\} \subseteq \mathsf{dom}\ s$, from Lem. 5.4.2 and Lem. 5.4.4. The masked mapping assertion on the RHS, with f now included in the set of masked fields, then holds on the remaining state:

$$\frac{(p \mapsto_g{}^{F}\ v)\ s \qquad \mathsf{TYPE}(\alpha)_\tau \triangleright f = \lfloor (t,\ n) \rfloor}{(p \mapsto_g{}^{(\{f\}\ \cup\ F)}\ v)\ (s\restriction_{(\mathsf{dom}\ s\ -\ \mathsf{fs\text{-}footprint}\ p\ \{f\})})}$$

This can be seen by observing that validity at p is preserved by the domain restriction, since p is valid in the $\mathsf{singleton}\ p\ v\ {+\!\!+}\ s$ from the assumption, where $\mathsf{dom}\ s = \mathsf{s\text{-}footprint}\ p\ -\ \mathsf{fs\text{-}footprint}\ p\ F$, with:

$$\begin{aligned}&\mathsf{singleton}\ p\ v\ {+\!\!+}\ s\restriction_{(\mathsf{s\text{-}footprint}\ p\ -\ \mathsf{fs\text{-}footprint}\ p\ F\ -\ \mathsf{fs\text{-}footprint}\ p\ \{f\})}\\ &\quad = \mathsf{singleton}\ p\ v\ {+\!\!+}\ s\ {+\!\!+}\ \mathsf{singleton}\ p\ v\restriction_{\mathsf{fs\text{-}footprint}\ p\ \{f\}}\end{aligned}$$

and $[\![s,g \models_s p;\ t,g' \models_s p]\!] \implies s\ {+\!\!+}\ t\restriction_X, g \models_s p$. The contents of the heap may change as a result of the domain restriction though, as the singleton state supplies normalised value representations for removed fields in the map addition. To show the lifted value remains v, we utilise the approach in the proof of Thm. 5.3.8, where the underlying representation is split into three components, the field f's representation and the segments before and after. The map addition of the heap state covering the field's footprint, $\mathsf{singleton}\ p\ v\restriction_{\mathsf{fs\text{-}footprint}\ p\ \{f\}}$ is then an $\mathsf{update\text{-}ti}\ t$ on v with f's byte representation in $\mathsf{singleton}\ p\ v\restriction_{\mathsf{fs\text{-}footprint}\ p\ \{f\}}$. That this does not modify v can be seen with [FuFaId] and:

$$\frac{\mathsf{TYPE}(\alpha)_\tau \triangleright f = \lfloor (t,\ n) \rfloor}{\mathsf{heap\text{-}list\text{-}s}\ (\mathsf{singleton}\ p\ v\restriction_{\mathsf{fs\text{-}footprint}\ p\ \{f\}})\ (\mathsf{size\text{-}td}\ t)\ \&(p{\rightarrow}f) = \mathsf{access\text{-}ti}_0\ t\ v}$$

In the other direction we demonstrate that the LHS holds on s, given this for the RHS. Now the separation conjunction gives the partitioning of the heaps. The non-trivial part of the proof is again to show equivalence of lifted values at p. To do so we split the heap representation into three segments as before and hence have the singleton map assertion for f providing a field update on the lifted value. The proof is then completed with the aid of the field description consistency conditions. □

Corollary. *A mapping assertion for a valid qualified field name can be derived from a singleton heap assertion with:*

$$\frac{\mathsf{TYPE}(\alpha)_\tau \triangleright f = \lfloor (d,\ n) \rfloor \qquad (p \mapsto_g v)\ s \qquad \mathsf{export\text{-}uinfo}\ d = \mathsf{TYPE}(\beta)_\nu \qquad \mathsf{guard\text{-}mono}\ g\ g'}{(\mathsf{Ptr}\ \&(p{\rightarrow}f) \hookrightarrow_g{}'\ \mathsf{from\text{-}bytes}\ (\mathsf{access\text{-}ti}_0\ d\ v))\ s}$$

where p::α::*mem-type ptr and* Ptr $\&(p{\rightarrow}f)$::β::*mem-type ptr.*

Theorem 5.4.6. *A valid disjoint field may be folded into a masked mapping assertion with:*

$$\frac{\mathsf{TYPE}(\alpha)_\tau \triangleright f = \lfloor (t,\ n) \rfloor \qquad f \in F \qquad \mathsf{disjoint\text{-}fn}\ f\ (F - \{f\}) \qquad \mathsf{guard\text{-}mono}\ g\ g' \qquad \mathsf{export\text{-}uinfo}\ t = \mathsf{TYPE}(\beta)_\nu}{p \mapsto_g{}^F v \wedge^* \mathsf{Ptr}\ \&(p{\rightarrow}f) \mapsto_g{}' w = p \mapsto_g{}^{(F - \{f\})} \mathsf{update\text{-}ti}\ t\ (\mathsf{to\text{-}bytes}_0\ w)\ v}$$

where p::α::*mem-type ptr and* Ptr $\&(p{\rightarrow}f)$::β::*mem-type ptr.*

Proof. Thm. 5.4.5 can be applied to the RHS. The two singleton mapping assertions for f then cancel out, leaving us to establish:

$$\frac{\mathsf{TYPE}(\alpha)_\tau \triangleright f = \lfloor (t,\ n) \rfloor \qquad f \in F \qquad \mathsf{disjoint\text{-}fn}\ f\ (F - \{f\}) \qquad \mathsf{guard\text{-}mono}\ g\ g' \qquad \mathsf{export\text{-}uinfo}\ t = \mathsf{TYPE}(\beta)_\nu}{p \mapsto_g{}^F v = p \mapsto_g{}^F \mathsf{update\text{-}ti}\ t\ (\mathsf{to\text{-}bytes}_0\ w)\ v}$$

The v value on the RHS can be expanded as update-ti t (to-bytes$_0$ (from-bytes (access-ti$_0$ t v))) v with the field consistency conditions. The proof is completed by expanding definitions, the field consistency conditions and on the RHS:

$$\begin{array}{l}\bigwedge s.\ \mathsf{from\text{-}bytes} \\ \quad (\mathsf{heap\text{-}list\text{-}s}\ (\mathsf{singleton}\ p\ (\mathsf{update\text{-}ti}\ t\ (\mathsf{to\text{-}bytes}_0\ w)\ v)\ +\!\!+\ s) \\ \quad\ \ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha))\ p_{\&}) = \\ \quad \mathsf{update\text{-}ti}\ t\ (\mathsf{to\text{-}bytes}_0\ w) \\ \quad (\mathsf{from\text{-}bytes}\ (\mathsf{heap\text{-}list\text{-}s}\ (\mathsf{singleton}\ p\ v\ +\!\!+\ s)\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha))\ p_{\&}))\end{array}$$

This can be seen by reducing the map addition to a field update as in the unfolding proof. □

Example 5.4.4. The y field of an *x-struct* can be unfolded as:

$$p \mapsto_{\text{ptr-aligned}} (\!|\mathsf{y} = 3, \mathsf{z} = 65|\!) \;=\; \mathsf{Ptr}\ \&(p{\rightarrow}[''y'']) \mapsto_{\text{ptr-aligned}} 3 \wedge^* p \mapsto_{\text{ptr-aligned}}^{\{[''y'']\}} (\!|\mathsf{y} = 3, \mathsf{z} = 65|\!)$$

Later, after an update to y setting it to the value 1, it can be folded back to the structured value to give the expected update:

$$\mathsf{Ptr}\ \&(p{\rightarrow}[''y'']) \mapsto_{\text{ptr-aligned}} 1 \wedge^* p \mapsto_{\text{ptr-aligned}}^{\{[''y'']\}} (\!|\mathsf{y} = 3, \mathsf{z} = 65|\!) \;=\; p \mapsto_{\text{ptr-aligned}} (\!|\mathsf{y} = 1, \mathsf{z} = 65|\!)$$

The masked mapping assertion is a constructive approach to unfolding. It may seem that separation implication could offer a simpler approach, where masked fields could be placed in the premise. Unfortunately it has not been our experience that this is the case. Separation implication leaves us with a non-domain exact predicate, and even if a dedicated predicate for non-constructive masking is used, problems arise due to singleton mapping not being strictly exact, which leads us to rely on properties of the singleton mapping assertion and with proofs no simpler than the above.

5.4.5 Lifting proof obligations

The proof obligations output by the verification condition generator still have the same form as in §4.3. To solve these, we can make use of the existing rules for reasoning about lifts and heap-updates, as well as new rules that can be derived for structured types.

Theorem 5.4.7. *Thm. 4.3.5, Thm. 4.3.6 and Thm. 4.3.8 continue to hold:*

$$\frac{(p \hookrightarrow_g v)\ (\text{lift-state}\ (h,\ d))}{\text{lift}\ h\ p = v}$$

$$\frac{\exists v.\ (p \mapsto_g v \wedge^* (p \mapsto_g v \longrightarrow^* P\ v))\ (\text{lift-state}\ (h,\ d))}{P\ (\text{lift}\ h\ p)\ (\text{lift-state}\ (h,\ d))}$$

$$\frac{(g \vdash_s p \wedge^* (p \mapsto_g v \longrightarrow^* P))\ (\text{lift-state}\ (h,\ d))}{P\ (\text{lift-state}\ (\text{heap-update}\ p\ v\ h,\ d))}$$

$$\frac{(g \vdash_s p \wedge^* R)\ (\text{lift-state}\ (h,\ d))}{(p \mapsto_g v \wedge^* R)\ (\text{lift-state}\ (\text{heap-update}\ p\ v\ h,\ d))}$$

Proof. As before, but for the heap-update rules we use a slightly different notion of the singleton state to Defn. 4.3.2 and Defn. 5.4.4:

$$\text{singleton}\ p\ v\ h\ d \equiv \text{lift-state}\ (\text{heap-update}\ p\ v\ h,\ d)\restriction_{\text{s-footprint}\ p}$$

This avoids the normalising behaviour of Defn. 5.4.4 when revising Lem. 4.3.7. □

Theorem 5.4.8. *For updates of a* Ptr $\&(p{\rightarrow}f)::\alpha::$*mem-type ptr corresponding to a field of* $p::\beta::$*mem-type ptr, where we have a singleton mapping assertion for* p*, we can use:*

$$\frac{\begin{array}{l}(p \mapsto_g u \wedge^* R)\ (\mathsf{lift\text{-}state}\ (h,\ d)) \qquad \mathsf{TYPE}(\beta)_\tau \triangleright f = \lfloor (t,\ n) \rfloor \\ \mathsf{export\text{-}uinfo}\ t = \mathsf{TYPE}(\alpha)_\nu \qquad w = \mathsf{update\text{-}ti}\ t\ (\mathsf{to\text{-}bytes}_0\ v)\ u\end{array}}{(p \mapsto_g w \wedge^* R)\ (\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ (\mathsf{Ptr}\ \&(p{\rightarrow}f))\ v\ h,\ d))}$$

Proof. Convert to a masked mapping assertion with Thm. 5.4.3. Unfold f with Thm. 5.4.5. Apply the heap-update rule of Thm. 5.4.7 and fold f back in with Thm. 5.4.6. The proof is complete by returning to a singleton mapping assertion with Thm. 5.4.3 again. At various points it is necessary to simplify with the field description consistency conditions. □

Thm. 5.4.8 can be applied in goals in similar situations to Thm. 5.3.8 and Thm. 5.3.9.

5.4.6 Retyping

To be able to express backwards compatible ptr-retyp rules, we need to consider how the type information space is managed. Suppose we know at p that an *int* pointer is valid, and wish to retype it to an *x-struct*. Since they are the same size, this should be perfectly reasonable and Thm. 4.3.11 provides rules at the separation logic level for this.

With the *heap-state* in Defn. 5.3.4 we encounter a problem, as a validity or singleton mapping assertion for *int* restricts the *s-heap-index* component of the domain to SIndexTyp *0*, and the retyping to *x-struct* affects SIndexTyp *1* as well. The problem is that retyping can affect an arbitrary amount of *s-heap-index* space in an *addr* interval, and the singleton mapping and validity assertions are domain exact. A solution is to carry around an additional predicate stating ownership of all the type information space in the heap interval concerned, in a stronger singleton mapping assertion that needs to be available to code performing retyping. Code that does not need to retype can continue to use Defn. 5.4.2 and Defn. 5.4.3. We now give the definitions for these stronger assertions.

Definition 5.4.9. The *inverse footprint assertion* for a $p::\alpha::$*c-type ptr* is given by:

$$\begin{array}{l}\mathsf{inv\text{-}footprint}\ p \equiv \\ \lambda s.\ \mathsf{dom}\ s = \{(x,\ y) \mid x \in \{p_{\&}..{+}\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)\}\} - \mathsf{s\text{-}footprint}\ p\end{array}$$

This asserts ownership on the type information footprint of the interval of a pointer that is not covered by the singleton mapping assertion.

Definition 5.4.10. The singleton mapping and validity assertions can be strengthened with the inverse footprint assertion:

$$p \mapsto^i_g v \equiv p \mapsto_g v \wedge^* \mathsf{inv\text{-}footprint}\ p$$
$$g \vdash_s{}^i\ p \equiv g \vdash_s p \wedge^* \mathsf{inv\text{-}footprint}\ p$$

The other definitions in §4.2.1 have similar counterparts. sep-cut can also be stated on the new *heap-state*:

$$\mathsf{sep\text{-}cut}\ p\ n \equiv \lambda s.\ \mathsf{dom}\ s = \{(x,\ y) \mid x \in \{p..{+}n\}\}$$

Theorem 5.4.9. *Thm. 4.3.11 can be shown for* ptr-retyp *with the strengthened validity assertion:*

$$\frac{(\mathsf{sep\text{-}cut}\ p_{\&}\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)) \wedge^*\ P)\ (\mathsf{lift\text{-}state}\ (h,\ d)) \qquad g\ p}{(g \vdash_s{}^i\ p \wedge^*\ P)\ (\mathsf{lift\text{-}state}\ (h,\ \mathsf{ptr\text{-}retyp}\ p\ d))}$$

$$\frac{(\mathsf{sep\text{-}cut}\ p_{\&}\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)) \wedge^*\ (g \vdash_s{}^i\ p \longrightarrow^*\ P))\ (\mathsf{lift\text{-}state}\ (h,\ d)) \qquad g\ p}{P\ (\mathsf{lift\text{-}state}\ (h,\ \mathsf{ptr\text{-}retyp}\ p\ d))}$$

where $p{::}\alpha{::}$*mem-type ptr.*

Proof. Similar to the proof of Thm. 4.3.11. □

We use the non-strengthened assertions in the next section's in-place list reversal example, but in Chapter 6 we use the developments of this section to cope with retyping annotations.

5.5 Example: In-place list reversal revisited once more

We continue the in-place list reversal example of §3.6 and §4.5.1 in Table 5.3 using a **struct** type to represent nodes. This time there is no casting and the updates are safe. The list abstraction predicate is now defined as:

$$\begin{aligned}\mathsf{list}\ []\ i &\equiv \lambda s.\ i = \mathsf{NULL} \wedge \square\ s\\ \mathsf{list}\ (x{\cdot}xs)\ i &\equiv \lambda s.\ i \neq \mathsf{NULL} \wedge\\ &\qquad (\exists j.\ \mathsf{item}\ j = x \wedge (i \mapsto_g j \wedge^*\ \mathsf{list}\ xs\ (\mathsf{next}\ j))\ s)\end{aligned}$$

Theorem 5.5.1. ***reverse_struct*** *implements its specification.*

Proof. After running the verification condition generation, we are left with the 3 resulting proof obligations arising from the while Hoare logic rule with the invariant:

$\forall\, zs.\ \Gamma\vdash \{\!|(\mathsf{list}\ zs\ \acute{}ptr)^{sep}|\!\}$
$\quad \acute{}reverse\text{-}struct\text{-}ret :== reverse\text{-}struct(\acute{}ptr)$
$\quad \{\!|(\mathsf{list}\ (\mathsf{rev}\ zs)\ \acute{}reverse\text{-}struct\text{-}ret)^{sep}|\!\}$

```
struct node {
    int item;
    struct node *next;
};

struct node *reverse_struct (struct node *ptr)
{
    struct node *last = NULL;

    while (ptr) {
        struct node *temp = ptr->next;

        ptr->next = last;
        last  = ptr;
        ptr  = temp;
    }

    return last;
}
```

Table 5.3: **reverse_struct** specification and definition.

$\{\!|\exists\, xs\ ys.\ (\mathsf{list}\ xs\ \acute{}ptr \wedge^*\ \mathsf{list}\ ys\ \acute{}last)^{sep} \wedge\ \mathsf{rev}\ zs = \mathsf{rev}\ xs\ @\ ys|\!\}$

The $Pre \implies Inv$ and $Inv \Rightarrow Post$ conditions are trivial. The loop invariant preservation proof requires we show:

1. $\bigwedge zs\ a\ b\ last\ ptr\ ys\ list\ j.$
$\quad [\![ptr \neq \mathsf{NULL};\ \mathsf{rev}\ zs = \mathsf{rev}\ list\ @\ \mathsf{item}\ j{\cdot}ys;$
$\quad (ptr \mapsto_g j \wedge^*\ \mathsf{list}\ list\ (\mathsf{next}\ j) \wedge^*\ \mathsf{list}\ ys\ last)$
$\quad (\mathsf{lift\text{-}state}\ (a,\ b))]\!]$
$\quad \implies (ptr \mapsto_g j(|\mathsf{next} := last|) \wedge^*$
$\quad\quad \mathsf{list}\ ys\ last \wedge^*\ \mathsf{list}\ list\ (\mathsf{lift}\ a\ (\mathsf{Ptr}\ \&(ptr{\rightarrow}[''next''])))$
$\quad\quad (\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ (\mathsf{Ptr}\ \&(ptr{\rightarrow}[''next'']))\ last\ a,\ b))$

This follows from Thm. 5.4.8. The first side-condition may be discharged with Thm. 4.3.5 and Thm. 5.4.5, eliminating the lift. The other side-conditions are discharged by rewriting, using the rules of §5.2.5.

□

An interesting point in the proof is when we have to show:

```
1. ⋀zs a b last ptr ys list j.
     ⟦ptr ≠ NULL; rev zs = rev list @ item j·ys;
      (ptr ↦g j ∧* list list (next j) ∧* list ys last)
       (lift-state (a, b))⟧
     ⟹ j(|next := last|) = update-ti
                             (adjust-ti TYPE(node ptr)τ next
                               (next-update ∘ (λx -. x)))
                             (to-bytes0 last) j
```

Here, applying the reverse definition of from-bytes and the α::*mem-type* axioms lifts the RHS to the HOL **record** level to simplify for the goal.

Compared to the earlier in-place list reversal examples, the proof script was about the same structure and size, 67 lines. We can then see that for this example, the sophisticated machinery of this chapter does not unduly burden a verification that remains in the type-safe fragment of C.

Chapter 6

Case study: L4 kernel memory allocator

So far, all examples have been toy ones, in some cases positively contrived. In this chapter we present a case study in the application of our models to the verification of real-world C systems code derived from an implementation of the L4 [57] microkernel. Not only does this provide an opportunity to validate the models against realistic code, but it also allows us to compare and contrast the multiple typed heaps and separation logic abstractions in practice. While separation logic enjoys clear superiority in its ability to cope with intra-type aliasing and the frame problem, on smaller, specialised, verifications we may benefit from the easier automation of multiple typed heaps verification, so the winner is not clear *a priori*.

The microkernel concept is simple — only place what needs to execute in the CPU's privileged modes in the kernel and execute the rest of the system above this level. This provides the benefits of hardware enforced process isolation to the rest of the OS, as well as applications, and massively reduces the trusted computing base. Early microkernels were not performant or entirely minimal and developed a reputation as being impractical for real systems. L4 is a second-generation microkernel, with only a small number of primitives concerned with three abstractions provided by the kernel — threads, address spaces and inter-process communication (IPC). Liedtke [57] demonstrated through L4 that paying close attention to IPC overheads and cache footprint is key to performance.

The implementation we target is L4Ka::Pistachio [90]. Pistachio is a mostly C++ implementation, with a small amount of architecture and platform dependent assembler, of the L4 X.2 API [52] and has been ported to many architectures (x86, ARM, Alpha, MIPS, Itanium, PowerPC) without sacrificing high performance. We have chosen this to study as Pistachio has been used in industry as well as academia [41], has a relatively mature code base, was the basis of early exploratory work on the L4.verified project [94]

and also as a result of the author's familiarity with the kernel [103].

While Pistachio is implemented in C++, no essential use of C++ features is made apart from using classes to structure the code. For the sample of the kernel under investigation, we configure the kernel for the x86 architecture, preprocess the source code, strip debugging statements and irrelevant coarse-grained locks for multiprocessor implementations and end up with something that is essentially C, albeit with function signatures including classes which we also drop as they are of no import inside the memory allocator itself. The result of this process is then valid C_{sys} code which can be input to the translation stage of the verification flow.

We now examine the subsystem we have selected for the case study — the internal kernel memory allocator. The following sections detail the role of the kernel memory allocator, its implementation data structures, code, specifications and proofs in both models.

6.1 Kernel memory management

To support L4's abstractions, Pistachio requires heap-allocated storage for dynamic kernel data structures like page tables and thread control blocks. At the kernel level, the usual C library functions for this task, **malloc** and **free**, are not available yet and have to be provided internally. This presents an ideal target for the memory models we have developed so far as the implementation of a memory allocator will have both safe and unsafe C expressions.

Three functions define the interface of the kernel memory allocator:

```
void init(void *start, void *end);
void *alloc(word_t size);
void free(void *address, word_t size);
```

init takes a contiguous region of memory and sets this as the free pool. This should be aligned and sized a multiple of the allocator's "chunk" size, which we take as 1KB. **alloc** returns an aligned pointer to a block of memory of the requested size if available, otherwise NULL. If the size is less than 1KB, it is rounded up to the kilobyte. The alignment is that of the request size if it is a power-of-two. The final function, **free**, allows allocated memory to be returned to the free pool. The given size should be that of the original request size. This is different to the standard C library's allocator which tracks the size of allocated memory for each block.

The granularity of allocation is quite coarse — other kernel subsystems like the mapping database implement their own second-level memory management that is ultimately backed by the kernel memory allocator.

6.2 Data structures

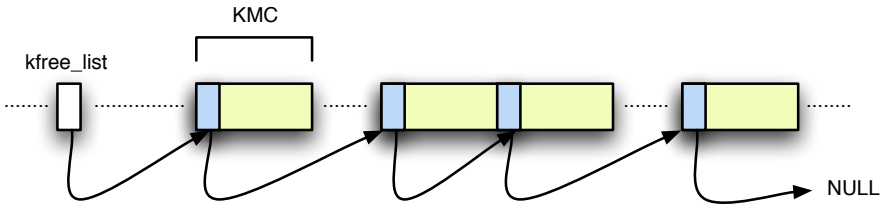

Figure 6.1: Management data structure of the L4 memory allocator.

Fig. 6.1 depicts the internal data structure that is used to manage memory. It is a NULL-terminated, singly-linked list of chunks of memory of a fixed 1KB size. A single global variable word_t * kfree_list provides a pointer to the start of the free list. Rather than storing the meta-information apart from the free memory, for efficiency, the first 4 bytes of each free memory block are used to point to the next one. The blocks are often, but not always, adjacent in memory, and are ordered by base address. This has the effect of reducing fragmentation.

In Fig. 6.2 some free list states and transformations are given. The initial list state is just the original region divided into 1KB chunks linked to their next neighbour. After allocation, links are adjusted and when a block is later returned during deallocation it is inserted at the correct point in the list.

6.3 Implementation code

The C_{sys} source code for **alloc**, **init** and **free** is given in Table 6.1, Table 6.2 and Table 6.3 respectively. Annotations are omitted as they differ in the two verifications and instead are supplied in §6.5. Since **init** is simply a call to **free** we do not need to provide it any special treatment in the verification and focus instead on **alloc** and **free**. The source code is mostly platform independent, with the platform dependency in the parameters supplied to **init**. However, the code is also not strictly conforming either, with assumptions made about the interchangeability of pointers and integers in the casting and pointer arithmetic and a ROLM-like treatment of memory objects.

alloc performs any necessary rounding on the supplied size to bring it to at least 1KB, which we use the constant KMC to represent, and then executes its outer loop, traversing the free list. If a chunk is found with the intended alignment, an inner loop is entered that traverses the list from the found block, attempting to establish that a contiguous region of memory

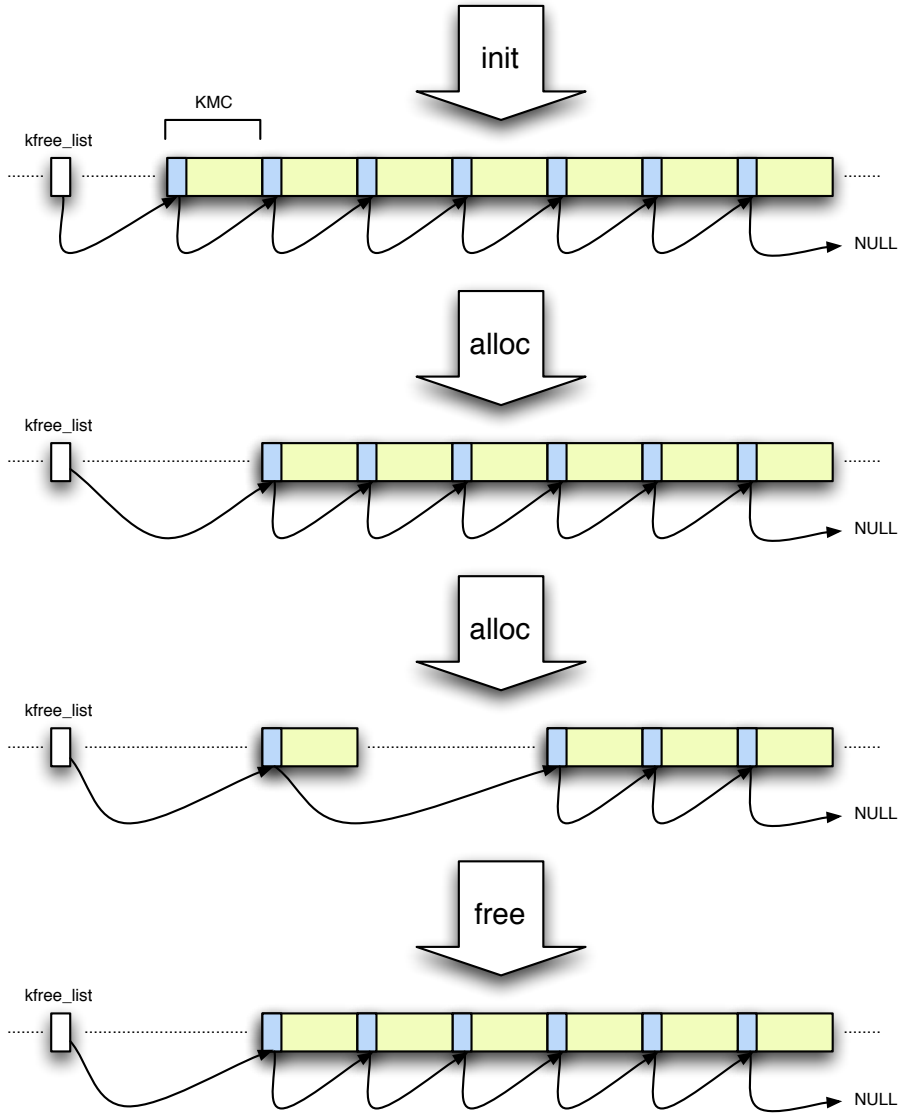

Figure 6.2: Allocator states across operations.

of the desired size exists in the free list there. If found, the link from the previous chunk's next pointer is adjusted to point to the chunk following the contiguous region in the list, the region is zeroed and a pointer to its base is returned to the caller. Otherwise, traversal continues in the outer loop. If no suitable contiguous region is found a NULL value is returned.

The **free** function is shorter and also begins with an adjustment of size to a minimum of 1KB. The returning region is divided into 1KB chunks and a list is threaded through the chunks in the first loop. The free list is then traversed in the second loop to find the correct location to maintain the base address ordering invariant and the appropriate pointers are adjusted to perform the insertion.

6.4 Specifications

We now give in detail the multiple typed heaps and separation logic specifications. At the specification stage the separation logic description of behaviours is clearer and stronger as we can globalise with the frame rule.

Multiple typed heaps

The data abstraction predicate for the free list is very similar to the simple linked list of §3.6:

$$\begin{array}{lcl} \mathsf{list}\ s\ p\ q\ [] & \equiv & p = q \\ \mathsf{list}\ s\ p\ q\ (x{\cdot}xs) & \equiv & p = \mathsf{Ptr}\ x \wedge p \neq q \wedge (\exists\, y.\ s\ p = \lfloor y \rfloor \wedge \mathsf{list}\ s\ (\mathsf{Ptr}\ y)\ q\ xs) \end{array}$$

We supply an additional parameter for the tailing pointer as we require the ability to reason about list fragments in the proofs, which are not in general NULL terminated.

For the specification of **alloc**, we first need to define the abstract behaviour. For this, we are not interested in the list structure itself, but only in the set of chunks in the free pool:

$$\begin{array}{lcl} \mathsf{free\text{-}set}\ s\ p\ q\ F & \equiv & \exists\, xs.\ \mathsf{list}\ s\ p\ q\ xs \wedge \mathsf{distinct}\ xs \wedge F = \mathsf{set}\ xs \\ \mathsf{alloc}\ p\ n\ F & \equiv & F - \mathsf{chunks}\ p\ (p_{\&} + (n - \mathsf{KMC})) \end{array}$$

The function $\mathsf{chunks}\ p\ q$ in the definition above refers to a set of locations starting with pointer p, ending with address q, that consists of base addresses of adjacent memory chunks:

$$\begin{array}{l} \mathsf{chunks}\ p\ y \equiv \\ \{x \mid p_{\&} \leq x \wedge x \leq y \wedge (\exists\, n{\geq}0.\ \mathbb{Z}^{\Leftarrow}\ x = \mathbb{Z}^{\Leftarrow}\ p_{\&} + n * \mathbb{Z}^{\Leftarrow}\ \mathsf{KMC})\} \end{array}$$

To make subtraction better behaved we map our bit-vectors to the integers rather than natural numbers in this definition and extend the $\mathbb{N}$ syntax to $\mathbb{Z}$.

```
void *alloc(word_t size)
{
    word_t i;
    word_t *prev, *curr, *tmp;

    size = size >= KMC ? size : KMC;

    for (prev = (word_t *)&kfree_list, curr = kfree_list;
         curr;
         prev = curr, curr = (word_t *)*curr) {

        if (!((word_t)curr & (size - 1))) {
            tmp = (word_t *)*curr;

            for (i = 1; tmp && (i < (size / KMC)); i++) {

                if ((word_t)tmp != ((word_t)curr + KMC * i)) {
                    tmp = 0;
                    break;
                }

                tmp = (word_t *)*tmp;
            }

            if (tmp) {
                *prev = (word_t)tmp;

                for (i = 0; i < (size / sizeof(word_t)); i++)
                    curr[i] = 0;

                return curr;
            }
        }
    }

    return 0;
}
```

Table 6.1: **alloc** definition.

```
void init(void *start, void *end)
{
    kfree_list = 0;

    free(start, (word_t)end - (word_t)start);
}
```

Table 6.2: **init** definition.

```
void free(void *address, word_t size)
{
    word_t *p, *prev, *curr;

    size = size >= KMC ? size : KMC;

    for (p = (word_t *)address;
         p < ((word_t *)(((word_t)address) + size - KMC));
         p = (word_t *)*p)
        *p = (word_t)p + KMC;

    for (prev = (word_t *)&kfree_list, curr = kfree_list ;
         curr && (address > (void *)curr);
         prev = curr, curr = (word_t *)*curr)
        ;

    *prev = (word_t)address;
    *p = (word_t)curr;
}
```

Table 6.3: **free** definition.

One final definition is required. free-set allows us to talk about fragments of the free list, but the data structure also has a header sentinel node, *kfree-list*, that does not contribute a chunk. This node must also be disjoint from the free pool. As it is not necessarily KMC aligned, we need to show disjointness not just with the base addresses of chunks but the entire area of memory described. We also require that for all true free list members that the base addresses are aligned. An abstraction predicate that takes this into account is:

$$\textsf{free-set-h}\ s_w\ s_p\ p\ r\ F \equiv$$
$$\exists\, q.\ s_p\ p = \lfloor q \rfloor \wedge \textsf{free-set}\ s_w\ q\ r\ F \wedge \textsf{disjoint-chunks}\ p_{\&}\ F \wedge \textsf{aligned}\ F$$

where

$$\begin{array}{lcl}
\mathsf{disjoint\text{-}chunks}\ p\ F & \equiv & \forall\, q{\in}F.\ p \notin \{q..{+}\mathbb{N}^{\Leftarrow}\ \mathsf{KMC}\} \\
x\ \mathsf{udvd}\ y & \equiv & \mathbb{N}^{\Leftarrow}\ x\ \mathsf{dvd}\ \mathbb{N}^{\Leftarrow}\ y \\
\mathsf{aligned}\ P & \equiv & P \cap \{p \mid \mathsf{KMC}\ \mathsf{udvd}\ p\} = P
\end{array}$$

In the free-set-h definition, the s_w and s_p parameters correspond to the lifted word_t and word_t * heaps respectively.

With these definitions, the Hoare logic specification of **alloc** is given by the triple:

$$\begin{array}{l}
\forall F\ \sigma.\ \Gamma{\vdash}\ \{\!|\sigma.\ \mathsf{free\text{-}set\text{-}h}\ \Phi\ \Phi\ \mathsf{kfree\text{-}list\text{-}addr}\ \mathsf{NULL}\ F \wedge \\
\qquad\qquad \mathsf{KMC}\ \mathsf{udvd}\ \mathsf{max}\ \acute{}size\ \mathsf{KMC}|\!\} \\
\qquad \acute{}alloc\text{-}ret :== alloc(\acute{}size) \\
\qquad \{\!|(\acute{}alloc\text{-}ret \neq \mathsf{NULL} \longrightarrow \\
\qquad\quad \mathsf{size\text{-}aligned}\ \acute{}alloc\text{-}ret\ (\mathsf{max}\ {}^{\sigma}size\ \mathsf{KMC}) \wedge \\
\qquad\quad \acute{}alloc\text{-}ret_{\&} \leq \acute{}alloc\text{-}ret_{\&} + \mathsf{max}\ {}^{\sigma}size\ \mathsf{KMC} - \mathsf{KMC} \wedge \\
\qquad\quad \mathsf{chunks}\ \acute{}alloc\text{-}ret \\
\qquad\quad (\acute{}alloc\text{-}ret_{\&} + (\mathsf{max}\ {}^{\sigma}size\ \mathsf{KMC} - \mathsf{KMC})) \\
\qquad\quad \subseteq F \wedge \\
\qquad\quad \mathsf{free\text{-}set\text{-}h}\ \Phi\ \Phi\ \mathsf{kfree\text{-}list\text{-}addr}\ \mathsf{NULL} \\
\qquad\quad (\mathsf{alloc}\ \acute{}alloc\text{-}ret\ (\mathsf{max}\ {}^{\sigma}size\ \mathsf{KMC})\ F)) \wedge \\
\qquad (\acute{}alloc\text{-}ret = \mathsf{NULL} \longrightarrow \mathcal{H} = {}^{\sigma}\mathcal{H})|\!\}
\end{array}$$

where the returned memory is guaranteed to be aligned to the effective request size, if a power-of-two, by:

$$\mathsf{size\text{-}aligned}\ p\ n \equiv (\exists\, k.\ n = 2^k) \longrightarrow n\ \mathsf{udvd}\ p_{\&}$$

Here we supply the lifted heaps with Φ, which was defined back in Exmp. 3.4.1 as the polymorphic lift_τ c-guard $\mathcal{H}$.

The pre-condition requires that the free list rooted at kfree-list-addr describe some set of free memory chunks F and that the effective requested size be aligned with KMC. Alignment is expressed using the non-overflowing version of divisibility on finite integers with udvd. The post-condition refers to the pre-state σ for *size* and the heap.

In the post-condition there are two cases. Either some memory was returned, $\acute{}alloc\text{-}ret \neq \mathsf{NULL}$, or the kernel ran out of memory, $\acute{}alloc\text{-}ret = \mathsf{NULL}$. In the latter case, we claim that nothing changes in the heap. In the former case, the new set of free memory chunks is equivalent to the set obtained by evaluation of the abstract alloc on the returned pointer. If the request size is a power-of-two then the returned pointer will have this alignment. We rule out the possibility of the returned region wrapping around the address space with $\acute{}alloc\text{-}ret_{\&} \leq \acute{}alloc\text{-}ret_{\&} + \mathsf{max}\ {}^{\sigma}size\ \mathsf{KMC} - \mathsf{KMC}$ and state that the returned blocks are a subset of F.

The abstract specification of **free** is:

$$\mathsf{free}\ p\ n\ F \equiv F \cup \mathsf{chunks}\ p\ (p_{\&} + (n - \mathsf{KMC}))$$

Unfortunately, the triple for **free** is more complicated:

$$\begin{array}{l}\forall F\ \sigma.\ \Gamma\vdash \{\!|\sigma.\ \mathsf{free\text{-}set\text{-}h}\ \Phi\ \Phi\ \mathsf{kfree\text{-}list\text{-}addr}\ \mathsf{NULL}\ F \wedge \\ \qquad\qquad F \cap \{\,'address_{\&}..+\mathbb{N}^{\Leftarrow}\ (\mathsf{max}\ 'size\ \mathsf{KMC})\} = \emptyset \wedge \\ \qquad\qquad 'address \neq \mathsf{NULL} \wedge \\ \qquad\qquad \mathsf{KMC}\ \mathsf{udvd}\ 'address_{\&} \wedge \\ \qquad\qquad \mathsf{KMC}\ \mathsf{udvd}\ \mathsf{max}\ 'size\ \mathsf{KMC} \wedge \\ \qquad\qquad 'address_{\&} \leq 'address_{\&} + \mathsf{max}\ 'size\ \mathsf{KMC} - \mathsf{KMC} \wedge \\ \qquad\qquad \mathsf{disjoint\text{-}chunks}\ \mathsf{kfree\text{-}list\text{-}addr}_{\&} \\ \qquad\qquad\ (\mathsf{chunks}\ 'address\ ('address_{\&} + (\mathsf{max}\ 'size\ \mathsf{KMC} - \mathsf{KMC})))\}\!| \\ \qquad free('address, 'size) \\ \qquad \{\!|\mathsf{free\text{-}set\text{-}h}\ \Phi\ \Phi\ \mathsf{kfree\text{-}list\text{-}addr}\ \mathsf{NULL} \\ \qquad\ (\mathsf{free}\ {}^{\sigma}address\ (\mathsf{max}\ {}^{\sigma}size\ \mathsf{KMC})\ F)\}\!|\end{array}$$

The additional complexity is entirely in the pre-condition and is related to the anti-aliasing conditions that need to be expressed. This is particularly a problem here as we do not know anything about the type of returned memory and hence cannot rely on pointer validity to help. The **free** pre-condition can be established from the post-condition of **alloc** and with these specifications we can prove the correctness of client code such as **kmalloc_test** in Table 6.4.

$$\begin{array}{l}\forall F\ \sigma.\ \Gamma\vdash \{\!|\sigma.\ \mathsf{free\text{-}set\text{-}h}\ \Phi\ \Phi\ \mathsf{kfree\text{-}list\text{-}addr}\ \mathsf{NULL}\ F \wedge \\ \qquad\qquad \mathsf{KMC}\ \mathsf{udvd}\ \mathsf{max}\ 'size\ \mathsf{KMC}\}\!| \\ \qquad kmalloc\text{-}test('size) \\ \qquad \{\!|\mathsf{free\text{-}set\text{-}h}\ \Phi\ \Phi\ \mathsf{kfree\text{-}list\text{-}addr}\ \mathsf{NULL}\ F\}\!|\end{array}$$

```
void kmalloc_test(word_t size)
{
    void *p;

    p = alloc(size);

    if (!p) return;

    free(p, size);
}
```

Table 6.4: **kmalloc_test** specification and definition.

These specifications are not completely satisfying. We would ideally like to know that nothing else in the heap changes. This "nothing else" is hard to nail down formally. The set F as used in the specification above is too loose

as it would miss the heap changes caused by zeroing out the freshly allocated memory in **alloc**. Separation logic handles this more naturally below.

Separation logic

The separation logic version of the abstraction predicate is the following:

$$
\begin{array}{lcl}
\mathsf{list}\ p\ r\ [] & \equiv & \lambda s.\ p = r \wedge \square\ s \\
\mathsf{list}\ p\ r\ (x{\cdot}xs) & \equiv & \lambda s.\ (p = \mathsf{Ptr}\ x \wedge p \neq r) \wedge \\
 & & \quad (\exists\, q.\ (\mathsf{block}\ p\ q \wedge^{*} \mathsf{list}\ q\ r\ xs)\ s) \\
\mathsf{block}\ p\ q & \equiv & \lambda s.\ \mathsf{KMC\ udvd}\ p_{\&} \wedge (p \hookrightarrow\ q_{\&})\ s \wedge \mathsf{sep\text{-}cut}\ p_{\&}\ \mathsf{KMC}\ s \\
\mathsf{free\text{-}set}\ p\ q\ F & \equiv & \lambda s.\ \exists\, xs.\ \mathsf{list}\ p\ q\ xs\ s \wedge F = \mathsf{set}\ xs \\
\mathsf{free\text{-}set\text{-}h}\ p\ r\ F & \equiv & \lambda s.\ \exists\, q.\ (\mathsf{ptr\text{-}coerce}\ p \mapsto q_{\&} \wedge^{*} \mathsf{free\text{-}set}\ q\ r\ F)\ s
\end{array}
$$

In addition to performing data abstraction, free-set now asserts ownership over the entire footprint of each chunk through the block predicate.

The separation logic specification of **alloc** is then given by:

$$
\begin{array}{l}
\forall F\ \sigma.\ \Gamma\vdash\ \{\!|\sigma.\ (\mathsf{free\text{-}set\text{-}h\ kfree\text{-}list\text{-}addr\ NULL}\ F)^{sep} \wedge \\
\qquad\qquad \mathsf{KMC\ udvd\ max}\ \acute{}size\ \mathsf{KMC}|\!\} \\
\qquad \acute{}alloc\text{-}ret :== alloc(\acute{}size) \\
\qquad \{\!|(\acute{}alloc\text{-}ret \neq \mathsf{NULL} \longrightarrow \\
\qquad\ \mathsf{size\text{-}aligned}\ \acute{}alloc\text{-}ret\ (\mathsf{max}\ {}^{\sigma}size\ \mathsf{KMC}) \wedge \\
\qquad\ \acute{}alloc\text{-}ret_{\&} \leq \acute{}alloc\text{-}ret_{\&} + \mathsf{max}\ {}^{\sigma}size\ \mathsf{KMC} - \mathsf{KMC} \wedge \\
\qquad\ \mathsf{chunks}\ \acute{}alloc\text{-}ret \\
\qquad\quad (\acute{}alloc\text{-}ret_{\&} + (\mathsf{max}\ {}^{\sigma}size\ \mathsf{KMC} - \mathsf{KMC})) \\
\qquad\ \subseteq F \wedge \\
\qquad\ (\mathsf{free\text{-}set\text{-}h\ kfree\text{-}list\text{-}addr\ NULL} \\
\qquad\quad (\mathsf{alloc}\ \acute{}alloc\text{-}ret\ (\mathsf{max}\ {}^{\sigma}size\ \mathsf{KMC})\ F) \wedge^{*} \\
\qquad\quad \mathsf{zero}\ \acute{}alloc\text{-}ret\ (\mathsf{max}\ {}^{\sigma}size\ \mathsf{KMC}))^{sep}) \wedge \\
\qquad (\acute{}alloc\text{-}ret = \mathsf{NULL} \longrightarrow \\
\qquad\ (\mathsf{free\text{-}set\text{-}h\ kfree\text{-}list\text{-}addr\ NULL}\ F)^{sep})|\!\}
\end{array}
$$

The pre-condition here is as before and the post-condition has the same two cases. If we have run out of memory, $\acute{}alloc\text{-}ret = \mathsf{NULL}$, we now only say that F does not change and by the frame rule it can be derived that nothing else in the heap changes either.

In the success case, $\acute{}alloc\text{-}ret \neq \mathsf{NULL}$, we still state that the new set of free memory chunks is the same as that given by evaluating the abstract function alloc. Additionally, we now explicitly say that the memory returned is a separate, contiguous block of the right size, filled with zero words:

$$
\begin{array}{lcl}
\mathsf{zero}\ p\ n & \equiv & \mathsf{zero\text{-}block}\ (\mathsf{ptr\text{-}coerce}\ p)\ (\mathbb{N}^{\Leftarrow}\ (n\ \mathsf{div}\ 4)) \\
\mathsf{zero\text{-}block}\ p\ 0 & \equiv & \square \\
\mathsf{zero\text{-}block}\ p\ (Suc\ n) & \equiv & (p +_p \mathbb{N}^{\Rightarrow}\ n) \mapsto 0 \wedge^{*} \mathsf{zero\text{-}block}\ p\ n
\end{array}
$$

The zero conjunct can be directly used by client code operating on the freshly allocated memory. All other memory is implicitly left unchanged by **alloc**.

The **free** triple is much clearer than in the previous section:

$$
\begin{array}{l}
\forall\, F\ \sigma.\ \Gamma\vdash \{\!|\sigma.\ (\textsf{free-set-h kfree-list-addr NULL}\ F\ \wedge^{*} \\
\qquad\qquad \textsf{sep-cut}\ \acute{}address_{\&}\ (\textsf{max}\ \acute{}size\ \textsf{KMC}))^{sep}\ \wedge \\
\qquad\qquad \acute{}address \neq \textsf{NULL}\ \wedge \\
\qquad\qquad \textsf{KMC udvd}\ \acute{}address_{\&}\ \wedge \\
\qquad\qquad \textsf{KMC udvd max}\ \acute{}size\ \textsf{KMC}\ \wedge \\
\qquad\qquad \acute{}address_{\&} \leq \acute{}address_{\&} + \textsf{max}\ \acute{}size\ \textsf{KMC} - \textsf{KMC}|\!\} \\
\qquad free(\acute{}address, \acute{}size) \\
\qquad \{\!|(\textsf{free-set-h kfree-list-addr NULL} \\
\qquad\quad (\textsf{free}\ {}^{\sigma}address\ (\textsf{max}\ {}^{\sigma}size\ \textsf{KMC})\ F))^{sep}|\!\}
\end{array}
$$

The returned memory is transferred with the sep-cut assertion and there is no need to state explicit anti-aliasing conditions. The advantages of separation logic assertions are even greater when we consider the invariants which are more detailed than the outer specifications.

6.5 Invariants

In this section we describe the loop invariants that were used to structure the verifications. These provide the key proof steps and insight into how the allocator works. We focus our attention on **alloc** here and present the separation logic invariants first as they are simpler.

Separation logic

The outer loop invariant is:

$$
\begin{array}{l}
\{\!|(\exists\, G\ H.\ (\acute{}prev = \textsf{ptr-coerce kfree-list-addr} \vee\ \acute{}prev_{\&} \in G)\ \wedge \\
\qquad\qquad (\textsf{free-set-h kfree-list-addr}\ \acute{}curr\ G\ \wedge^{*} \\
\qquad\qquad\ \textsf{free-set}\ \acute{}curr\ \textsf{NULL}\ H)^{sep}\ \wedge \\
\qquad\qquad F = G \cup H)\ \wedge \\
\quad (\textsf{free-set-h kfree-list-addr NULL}\ F)^{sep}\ \wedge \\
\quad \mathcal{H} = {}^{\sigma}\mathcal{H}\ \wedge \\
\quad \acute{}size = \textsf{max}\ {}^{\sigma}size\ \textsf{KMC}\ \wedge\ \textsf{KMC udvd}\ \acute{}size\ \wedge\ (\acute{}prev \hookrightarrow\ \ \acute{}curr_{\&})^{sep}|\!\}
\end{array}
$$

The pointer *curr* partitions the free list during the traversal. While the heap may be modified inside the loop body, if this occurs a **return** is always performed, so at the point where the invariant must hold, the heap state is never modified. The rest of the invariant mostly just carries information from the pre-condition.

Inside the outer loop, in the first inner loop, the situation is more tricky:

$$
\begin{array}{l}
\{\!|\ 'curr_{\&} \leq 'curr_{\&} + ('i - 1) * \mathsf{KMC} \wedge \\
\quad (\exists\, G\ H.\ ('prev = \mathsf{ptr\text{-}coerce\ kfree\text{-}list\text{-}addr} \vee 'prev_{\&} \in G) \wedge \\
\qquad\quad (\mathsf{free\text{-}set\text{-}h\ kfree\text{-}list\text{-}addr}\ 'curr\ G \wedge^{*} \\
\qquad\qquad \mathsf{free\text{-}set}\ 'curr\ 'tmp \\
\qquad\qquad\quad (\mathsf{chunks}\ 'curr\ ('curr_{\&} + ('i - 1) * \mathsf{KMC})) \wedge^{*} \\
\qquad\qquad \mathsf{free\text{-}set}\ 'tmp\ \mathsf{NULL}\ H)^{sep} \wedge \\
\qquad\quad F = G \cup \mathsf{chunks}\ 'curr\ ('curr_{\&} + ('i - 1) * \mathsf{KMC}) \cup H) \wedge \\
\quad (\mathsf{free\text{-}set\text{-}h\ kfree\text{-}list\text{-}addr\ NULL}\ F)^{sep} \wedge \\
\quad \mathcal{H} = {}^{\sigma}\mathcal{H} \wedge \\
\quad 'size = \mathsf{max}\ {}^{\sigma}size\ \mathsf{KMC} \wedge \\
\quad \mathsf{KMC\ udvd}\ 'size \wedge \\
\quad 1 \leq 'i \wedge \\
\quad 'i \leq 'size\ \mathsf{div\ KMC} \wedge \\
\quad 'curr \neq \mathsf{NULL} \wedge \mathsf{size\text{-}aligned}\ 'curr\ 'size \wedge ('prev \hookrightarrow\ 'curr_{\&})^{sep}\ |\!\}
\end{array}
$$

Now the free list is partitioned three ways — the fragment up to *curr*, the candidate allocation block between *curr* and *tmp* and the rest of the list. This is illustrated in Fig. 6.3. We also ensure the candidate block does not wrap around the address space, carry some information from the pre-condition and establish size-aligned as we have passed the alignment test prior to entering the loop. Still, the heap remains unmodified at this point.

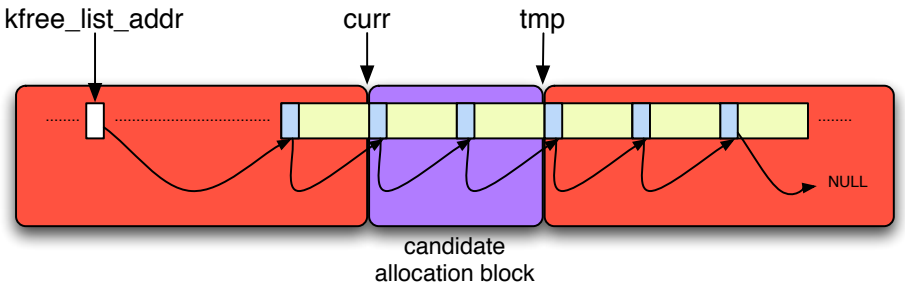

Figure 6.3: Partition of free list.

If the test for a suitably sized block in this first inner loop fails then we can show the outer loop invariant is implied by the inner loop invariant. If it succeeds then the block will be de-linked from the list structure, zeroed, and returned to the user. The second inner loop witnesses the zeroing and its invariant is:

$$\{\!|\ 'curr_{\&} \leq\ 'curr_{\&} + ('size - \mathsf{KMC}) \wedge$$
$$(\mathsf{free\text{-}set\text{-}h}\ \mathsf{kfree\text{-}list\text{-}addr}\ \mathsf{NULL}$$
$$(\mathsf{alloc}\ (\mathsf{ptr\text{-}coerce}\ 'curr)\ (\mathsf{max}\ ^{\sigma}size\ \mathsf{KMC})\ F) \wedge^{*}$$
$$\mathsf{sep\text{-}cut}\ ('curr_{\&} + 'i * 4)\ ('size - 'i * 4) \wedge^{*}$$
$$\mathsf{zero\text{-}block}\ 'curr\ (\mathbb{N}^{\Leftarrow}\ 'i))^{sep} \wedge$$
$$\mathsf{chunks}\ 'curr\ ('curr_{\&} + (\mathsf{max}\ ^{\sigma}size\ \mathsf{KMC} - \mathsf{KMC})) \subseteq F \wedge$$
$$'size = \mathsf{max}\ ^{\sigma}size\ \mathsf{KMC} \wedge$$
$$\mathsf{KMC\ udvd}\ 'size \wedge$$
$$\mathsf{KMC\ udvd}\ 'curr_{\&} \wedge$$
$$'i \leq\ 'size\ \mathsf{div}\ 4 \wedge \mathsf{size\text{-}aligned}\ 'curr\ 'size \wedge\ 'curr \neq \mathsf{NULL}\ |\!\}$$

When we enter the loop, the heap is partitioned such that the free-set-h conjunct describes the post-allocation free list state. The remainder of the heap is partitioned by the offset into the allocated block given by the loop counter i. The loop gradually retypes the allocated block as it zeroes it with a ptr-retyp $('curr +_p 'i)$ annotation in the body. When the loop condition fails, i.e. $\neg$ i < (size / **sizeof**(word_t)), the post-condition is implied.

Multiple typed heaps

We give the multiple typed heaps invariants in Table 6.5. They are quite similar in many respects to those in the previous section, but more verbose. Some parts have the appearance of an unfolded separation logic assertion. The main thing to observe is the additional number of anti-aliasing conditions — the structure of the proof itself does not differ significantly. It is clear that separation logic for specifications and invariants is far more concise in this case study and naturally suits the problem domain.

6.6 Results

With the invariants in place, the proof obligations post-VCG can be discharged. For this study we wrote mostly tactic-style proofs that we now discuss.

Multiple typed heaps

Table 6.6 lists the size of each proof script. For the specific code verification theories, in parenthesis we give the size of the main proof. The proof of the verification conditions for **alloc** takes about 400 LoP with an additional 190 LoP of specific supporting lemmas, a further 1400 LoP of lemmas shared[1]

[1]Shared lemmas are considered those lemmas found in theories imported by more than a single verification. They contain lemmas either used in more than one code proof or that are general properties of the data abstraction predicates.

Outer loop invariant:

$$\{\!|(\exists\, G\ H.\ (\text{´}prev = \text{ptr-coerce kfree-list-addr} \vee \text{´}prev_{\&} \in G) \wedge$$
$$\text{free-set-h}\ \Phi\ \Phi\ \text{kfree-list-addr}\ \text{´}curr\ G \wedge$$
$$\text{free-set}\ \Phi\ \text{´}curr\ \text{NULL}\ H \wedge G \cap H = \emptyset \wedge F = G \cup H) \wedge$$
$$\mathcal{H} = {}^{\sigma}\mathcal{H} \wedge$$
$$\text{´}size = \text{max}\ {}^{\sigma}size\ \text{KMC} \wedge$$
$$\text{KMC udvd}\ \text{´}size \wedge$$
$$\text{lift}\ (\text{hrs-mem}\ \text{´}t\text{-}hrs)\ \text{´}prev = \text{´}curr_{\&} \wedge$$
$$\text{disjoint-chunks kfree-list-addr}_{\&}\ F \wedge \text{aligned}\ F|\!\}$$

First inner loop invariant:

$$\{\!|\text{´}curr_{\&} \leq \text{´}curr_{\&} + (\text{´}i - 1) * \text{KMC} \wedge$$
$$(\exists\, G\ H.\ (\text{´}prev = \text{ptr-coerce kfree-list-addr} \vee \text{´}prev_{\&} \in G) \wedge$$
$$\text{free-set-h}\ \Phi\ \Phi\ \text{kfree-list-addr}\ \text{´}curr\ G \wedge$$
$$\text{free-set}\ \Phi\ \text{´}curr\ \text{´}tmp$$
$$(\text{chunks}\ \text{´}curr\ (\text{´}curr_{\&} + (\text{´}i - 1) * \text{KMC})) \wedge$$
$$\text{free-set}\ \Phi\ \text{´}tmp\ \text{NULL}\ H \wedge$$
$$G \cap \text{chunks}\ \text{´}curr\ (\text{´}curr_{\&} + (\text{´}i - 1) * \text{KMC}) = \emptyset \wedge$$
$$H \cap \text{chunks}\ \text{´}curr\ (\text{´}curr_{\&} + (\text{´}i - 1) * \text{KMC}) = \emptyset \wedge$$
$$G \cap H = \emptyset \wedge$$
$$F = G \cup \text{chunks}\ \text{´}curr\ (\text{´}curr_{\&} + (\text{´}i - 1) * \text{KMC}) \cup H) \wedge$$
$$\mathcal{H} = {}^{\sigma}\mathcal{H} \wedge$$
$$\text{´}size = \text{max}\ {}^{\sigma}size\ \text{KMC} \wedge$$
$$\text{KMC udvd}\ \text{´}size \wedge$$
$$1 \leq \text{´}i \wedge$$
$$\text{´}i \leq \text{´}size\ \text{div KMC} \wedge$$
$$\text{´}curr \neq \text{NULL} \wedge$$
$$\text{lift}\ (\text{hrs-mem}\ \text{´}t\text{-}hrs)\ \text{´}prev = \text{´}curr_{\&} \wedge$$
$$\text{size-aligned}\ \text{´}curr\ \text{´}size \wedge$$
$$\text{disjoint-chunks kfree-list-addr}_{\&}\ F \wedge \text{aligned}\ F|\!\}$$

Second inner loop invariant:

$$\{\!|(\text{´}curr_{\&} \leq \text{´}curr_{\&} + (\text{´}size - \text{KMC}) \wedge$$
$$\text{free-set-h}\ \Phi\ \Phi\ \text{kfree-list-addr NULL}$$
$$(\text{alloc}\ (\text{ptr-coerce}\ \text{´}curr)\ \text{´}size\ F)) \wedge$$
$$\text{chunks}\ \text{´}curr\ (\text{´}curr_{\&} + (\text{max}\ {}^{\sigma}size\ \text{KMC} - \text{KMC})) \subseteq F \wedge$$
$$\text{´}size = \text{max}\ {}^{\sigma}size\ \text{KMC} \wedge$$
$$\text{KMC udvd}\ \text{´}size \wedge$$
$$\text{KMC udvd}\ \text{´}curr_{\&} \wedge$$
$$\text{´}curr \neq \text{NULL} \wedge$$
$$\text{size-aligned}\ \text{´}curr\ \text{´}size \wedge$$
$$\text{disjoint-chunks kfree-list-addr}_{\&}$$
$$(\text{chunks}\ \text{´}curr\ (\text{´}curr_{\&} + (\text{´}size - \text{KMC})))|\!\}$$

Table 6.5: Multiple typed heaps allocator invariants.

Theory	LoP
Shared (all)	1400
Shared (multiple typed heaps)	330
Shared (separation logic)	688
alloc (multiple typed heaps)	581 (387)
alloc (separation logic)	975 (660)
free (multiple typed heaps)	924 (403)
free (separation logic)	736 (550)

Table 6.6: Proof script sizes.

between **alloc** and **free** in multiple typed heaps and separation logic settings and 330 LoP shared for only the multiple typed heaps. The **free** main proof is also about 400 LoP but with 520 LoP of specific supporting lemmas. The greater length of the latter appears to be a result of **free** retyping memory prior to its first loop (not shown in the earlier source code) where it threads the list structure through the returned memory. This requires a set of rules to cope with this unsafe operation. The reported sizes are based on a specific organisation of lemmas into theories and a particular proof style, so should be read only as an approximation. We should take from these results the order-of-magnitude as a guide and that **free**, despite having shorter source code than **alloc**, had a more difficult proof.

Separation logic

The proof of the verification conditions induced by the **alloc** specification takes 660 LoP with around an additional 320 LoP of specific supporting lemmas, 690 LoP shared with **free** and the 1400 LoP shared with the other parts of the verification. **free** this time has 550 LoP with 190 LoP supporting. This is more in line with **alloc** and may be shorter than above as retyping was not a special case, since we asserted ownership of returning memory in the pre-condition. While the effort appears to be a little higher for **alloc**, we have a stronger post-condition as a result. Much of the additional length in the **alloc** case can be attributed to verbose but not difficult proof steps, where perhaps we could in the future benefit from some automation.

Discussion

This case study corresponds to 136 lines of code in the original Pistachio source and 62 lines of code after configuring and preprocessing[2]. The functions are not very large, but the fact that the originals contain close to 40% tracing and debugging code indicates that they were not easy to get right. We did

[2]We only count function body sizes here.

not find any clear bugs in the code during verification, which is encouraging for a system with several years of deployment. There are however some subtleties that the specifications expose and would be useful for a kernel developer to be aware of. For example, size-aligned gives us alignment with the effective requested size, but only if this is a power-of-two. If a non-power-of-two is given, not only may we not get back a correctly aligned block, but the allocator may fail to allocate despite having sufficient resources as the alignment test in the source code will be incorrect. The full proof document for this case study, including code and invariants is available online [95].

The proofs were developed over several person months, simultaneously with the original separation logic embedding. It is hard to get a direct measure of how much time was spent as a result. The proofs are definitely not optimal with respect to length and there is the occasional redundancy or copy-and-paste evidence. This is not surprising — while it is reasonable to expect one to make an effort at providing clean theories for the memory models or other theory developments, the specific nature of a code verification proof makes this an unattractive expenditure of time.

Many of the proof steps were somewhat mechanical. At the leaves of the proofs we typically had some problem involving bit-vector arithmetic and intervals, reasoning about chunks, extracting mapping assertions from under a separation conjunction, massaging a separation assertion into the desired form or some property of the data abstraction predicates.

Even though the proofs were time consuming, we managed to prove some strong functional properties of some tricky low-level code. It should be noted that we chose **alloc** and **free** because they constitute a challenging case for this framework. While there are some type-safe accesses, e.g. the free list traversal and update, there are many pointer accesses that are unsafe and require additional reasoning, such as the first loop in **free** or the alignment test. Our framework provides both a means of coping with the unsafe parts and abstraction inside the safe fragment. In other pointer program verification developments in the literature this is impossible or leads to unsoundness if applied naïvely, here it is merely more work than usual. Once the allocator verification is completed, client code does not need to go to the same level of detail to use the pre/post conditions provided. The complexity is hence hidden.

Even small verifications such as this benefit from separation logic. While we did not use the frame rule, the stronger data abstraction predicates and specifications that are natural in this approach proved beneficial.

Looking ahead, with further experience we should be able to build up a set of libraries that amortise some of the proof effort, in particular on large verification projects like L4.verified. Even in this case study we managed to share a number of lemmas across verifications. Automation improvements would be welcomed, in particular tactics for manipulating separation logic assertions.

It would be fair to conclude that we have demonstrated the feasibility of verifying real systems code with the models presented in this thesis and interactive theorem proving. While proofs are lengthy and not cheap, we can show functional correctness and have a high degree of trust in the soundness of our tools, methods and models, going far beyond what is possible today with more automated algorithmic techniques.

Acknowledgments

The **free** specification and proof in the multiple typed heaps abstraction was joint work with Gerwin Klein.

Chapter 7

Conclusion

7.1 Discussion

We have seen that providing mechanised proof techniques for C, built on a low-level view of underlying memory and capable of managing the aliasing and frame problems, is feasible, although a technically involved process. The exercise has enhanced our understanding of the relationship between the models and how they can be extended to cope with language features such as unrestricted pointer arithmetic, casts, first-class structure types and relaxation of type- and memory-safety.

The L4 memory allocator case study and other earlier examples give us confidence that these developments are useful and not merely well-intentioned impractical explorations. We would go as far as to say that these code verifications would have been impossible or at least far more tedious and verbose without multiple typed heaps and separation logic, and we would likely have ended up reinventing the techniques in an ad-hoc manner anyway.

While we consider the effort to be a success, we should acknowledge limitations. Completeness has been mostly ignored due to the use of shallow embeddings of both C and the assertion languages. The case studies focus on a restricted subset of systems code, which we hope is representative. We have chosen to compromise on features such as pointers to stack-allocated storage, function pointers, virtual memory and concurrency. We assume some compiler behaviour that has not yet been independently verified. Nonetheless, the models developed in this thesis are being used in the L4.verified project for a complete microkernel verification. We anticipate that the only major extension needed will be the addition of virtual addressing to the model and even this will not be necessary in the vast majority of the code base. We believe the limitations can be addressed in future complementary work without invalidating the results of this thesis and that our contributions provide one of the necessary steps towards a body of knowledge that we require to be able to effectively verify C systems code.

C does not appear to be going away in the near future as a systems implementation language. While today it is becoming more common to use algorithmic techniques to analyse C systems code for some classes of bugs, we envision that the routine verification of functional properties of C systems code, albeit at relatively higher cost, is attainable and that a theorem prover like Isabelle/HOL provides the correct infrastructure for this task. The next two sections provide some more details of the mechanisation experience and suggested future directions to achieve this state of affairs.

7.2 Implementation experience

The core chapters of this thesis provide results and proofs, but do not contain a discussion of the effort required to attain them. The work occurred in two phases, where first the unified memory model (UMM) and separation logic abstractions were developed. At this point we carried out the memory allocator case study and followed this with the structured types extension. This involved a significant change to the underlying theory. We present in Table 7.1 the number of definitions, theorems and the size of the Isabelle/HOL theory files that are behind each model in the respective developments, measured in lines of proof (LoP) script.

	Initial development			**Structured types**		
	Defs	Thms	LoP	Defs	Thms	LoP
Semantic	41	60	846	152	566	6945
UMM	19	143	1190	45	220	3072
Separation logic	36	246	2010	48	396	4665
Total	96	449	4046	245	1182	14682

Table 7.1: Isabelle/HOL model implementation metrics.

These are simple metrics to obtain, but do not give a true picture of complexity on their own — the reader is referred to the relevant chapters to assess this. The structured types extension is not as mature as the initial development, with only a small number of the overall proofs being written in the high-level Isar proof language, the rest being tactic style. In the initial development, almost all proofs were in Isar.

The combined total of the initial development and structured types work is 18,728 LoP. The semantic model is the most significant increase in the generalised development, an order of magnitude larger than the original, compared to the UMM and separation logic models that are a factor of 2–3 larger. The reason for this substantial increase is that the deep embedding of structure information and its use requires many supporting lemmas to reason about. The proofs of these lemmas are through mutual induction,

which can be quite verbose. Often there is a need to derive a fairly intuitive well-formedness result from the generalised axioms. A further reason for the growth is the type information combinators, which required 1,415 LoP, and array treatment that had an additional 610 LoP in the semantic model.

Several iterations of the structured types extension took place, where we reduced the complexity of the underlying axioms and generalised them. Properties like Thm. 5.2.6 were axiomatised initially, but it became apparent later that these followed from the more fundamental set in §5.2.3. This style of development seems reasonable, albeit time consuming when many thousand LoP undergo revision.

The technical details of the models in Isabelle/HOL are lengthy, and some concepts seem intuitively obvious and unnecessary in their formal treatment. However, the mechanisation has the advantage of giving a high degree of trust in the soundness of the system that is unattainable in pen-and-paper formalisations and the user of the system is shielded from much of the implementation detail. It is easy to miss side-conditions such as alignment when reasoning about pointer programs and it is precisely this kind of detail that makes pointer program correctness in type-unsafe languages, even informally, a more difficult problem than general software correctness.

We benefited greatly from the maturity of tools and libraries in our work. Schirmer's verification environment clocks in at 27,400 LoP [85], the bit-vector libraries at 8,300 LoP and basic HOL libraries at around 35,000 LoP. Clearly if we had to engage in the development of these components it would not have been practical to carry out our developments and case study. The implementation in this thesis will in turn provide a basis for further layering and allow later research to reap the benefits of these models.

The case study in Chapter 6 was a success. As discussed in §6.6 we had many tedious bit-vector arithmetic proofs to carry out manually and a large number of the proofs were specialised to the free-list data structure in use, making it unlikely the proofs will be reusable for significantly different memory allocators. The loop invariants, with copious amounts of alignment detail, were quite large, although the specifications were compact enough to hide the details from clients. In both the multiple typed heaps and separation logic proofs, the effort required to apply the appropriate rewrites or inference rules for safe updates and accesses was relatively small compared to the overall proof burden, although retyping was non-trivial. From this we can conclude an achievement of our goals in the proof abstraction implementation and interface to the user.

7.3 Future work

There are many potential directions to take in further research. These can be clustered as language features, automation, experience and trustworthiness

improvements. Some additions should be fairly straightforward, such as dealing with physical sub-typing. Others, like integration with algorithmic techniques, will require more than just improvements to our models but also significant changes to the underlying verification infrastructure and theory. The significance of each possible direction will depend on the specifics of the verifications one wishes to perform. We make the general observation that carrying out what some may consider boring and laborious mechanical verification proofs, of actual real-world systems code, provides a learning experience that allows one to prioritise and understand the relative significance of various theories and technologies.

Language features

Chapter 5 deals with **struct**s and arrays, but does not discuss **union**s. As we comment in §2.4.3, these types are not so easy to even encode in HOL in the first place, if a general treatment is the goal. Given that the use of a different **union** member indicates effectively a retype operation, it does not make sense to treat them as first-class types in our setting. As second-class types, they could be handled as with casts. Moy [65] observes that tagged unions and physical sub-typing [88] casts in C are much better behaved than the general case. When two structured types share a common prefix, the prefix fields of these types remain valid across a retype of the heap type description. Relaxing the suggested restrictions in §2.4.3 could be necessary in some verifications. In this case retyping will have non-local effects, and we may require fundamental changes in the way structure information is represented in the embedding to avoid this.

The C_{sys}-*com* translation could be extended to handle the missing standard C control structures, such as **switch** and **goto**, possibly adapting the techniques of Tews [91]. *com* can support function pointers, and in standard compliant use we do not have to worry about adding functions to the memory model. In the case of self-modifying code, which does occur in some bootstrap systems code, one may have to reconsider this. Pointers to stack-allocated storage are more troublesome, as they can violate the validity of modeling local state as record fields, and could force all state to be moved into the memory model. As we move further from the C standard, other features that appear in systems code become interesting possibilities in future work. An integration with work like Myreen et al [66, 67], where a sound Hoare logic is developed to reason about ARM assembler, could make it possible to verify code with in-line assembler blocks. XCAP's treatment of non-standard control structures [71] appears useful and we could also stand to benefit from interfacing this type of reasoning with our C model. One other advanced feature is concurrency, which would require some fundamental changes to the way we treat the semantics of heap accesses and updates, as they could no longer be considered atomic. Depending on the level of realism required,

one may have to consider the details of memory ordering on contemporary processors [58].

Systems code may execute in the presence of address translation provided by the processor's memory management unit (MMU). This introduces new aliasing problems, as even pointers with distinct footprints in the virtual address space may alias. In addition, systems code executes in privileged mode and may modify virtual-physical mappings through page table updates. This could be an exciting target for separation logic, and Kolanski [50] has begun explorations that may pave the way here.

An extension of C with data refinement [25], where only vanilla C makes it to the compiler at the lowest level of refinement, could make an interesting project. Through data hiding and invariant isolation the verifier would gain abstraction, and as well as C's typed heaps, states could also contain ADT abstract state. Related work here includes the hypothetical frame rule [76] that facilitates hiding module invariants from client code, thus avoiding, for example, threading allocator invariants from Chapter 6 through the rest of the kernel proofs.

Automation

Our separation logic code proofs have many manual steps that could benefit from some simple tactic support, for example extracting existential quantifiers from inside separation conjuncts and also tactics that better support reordering conjuncts to fit a particular form. Appel [2] has done some recent work on this topic which we could build on.

Using Isabelle/HOL as the core verification platform and adopting a proof oriented approach does not prohibit us from benefiting from algorithmic techniques like software model checking and shape analysis. There are some problems though in how to structure verifications to take advantage of these methods. First, we usually only get a yes/no/don't-know response, possibly with counter-example, from these tools and after proof obligation generation it may be too late to apply them. The interactive proof paradigm requires that the proof state be advanced and then returned to the user, unless we are at the leaf of a proof. Bit-vector decision procedures are the exception, and we would benefit greatly from having these available to tackle the leaves of some of our proofs. Daum et al [24] make use of the verification environment's ability to discharge guard proof obligations separately. Not all of C's language features are always supported by these tools. This situation is improving, e.g. having structural invariants as given by shape analysis available to the verifier could be a benefit, and some more recent research has relaxed constraints that disallowed pointer arithmetic [17]. Model checkers and static analysers have their own memory models for C [23], and we would need to reconcile these views with those of this thesis when importing proof certificates.

How useful any of this is in practice depends on the code undergoing verification, the effort to interface the technique with Isabelle, the desired degree of maintainability and how hard the property is to prove without the technique. The last point is important — if memory safety is going to be implied by code invariants anyway, why bother with a tool that checks this?

Experience

We present a framework for verifying pointer programs and there are several parameters that could be varied. It could be seen how reasonable it is to implement different compiler and architecture assumptions, and how much work is involved in updating code verification proofs with these dependencies. The guards generated during translation could be enriched. ROLM could be strengthened and we could develop memory models that incorporate allocation primitives for code that does not need to implement its own.

More code verifications would help address any concerns over completeness, enrich the libraries for reasoning about unsafe code, expose limitations, help in understanding the scalability of Hoare logic verification of C systems code and provide us with greater experience in determining where next to focus attention. The L4.verified project should be very fruitful from this perspective.

Trust

While the proofs in this thesis are entirely machine-checked and sit above an LCF-style proof kernel, we still have various places where our confidence in soundness can be even further improved. The most obvious is the guards generated — we only provide a minimal set of guards in §2.5.5 to exercise the framework. At the very least there should be guards added to ensure the C run-time, i.e. code and stack, is protected from heap updates. This omission is not as serious as it may appear though in our case study, as our specifications and invariants, at least in the separation logic case, assert ownership over any memory that is modified.

The various compiler assumptions need to be validated. This could take the form of a code inspection, or even better a formally specified and verified compiler. Leroy [55] and Blazy et al [11] have developed such a compiler for a C subset Clight, targeting PowerPC, in Coq. Leinenbach et al [53, 54] also have a verified compiler for C0 in Isabelle/HOL.

The C_{sys}-*com* translation is about 5,200 LoC in ML, and we could reduce the trusted base in a few ways. First we could move to a deep embedding, as long as we could discard the additional structure during verification condition generation and have the translation executable. We could embed the translator itself inside an executable subset of Isabelle/HOL and prove correctness properties of it. There will probably always be some

trusted code involved for generating state spaces and the type encoding, since Isabelle/HOL does not reflect these aspects. This is another way in which the development of Chapter 5 is useful, as it allows at least the lvalue calculations to be pushed from trusted ML to the HOL level.

7.4 Concluding remarks

Verification of real-world systems code is hard and the devil is in the detail. We want to model systems faithfully, but at the same time can be easily overwhelmed by the lack of abstraction inherent in our systems languages. Through providing sound abstractions built on low-level system representations we can bridge the gap, and we have achieved this for memory models in C verification.

Appendix A

C_{sys} syntax

A semi-formal description of the C subset targeted in this thesis is presented below. Since the purpose of the grammar is to highlight differences between the C standard and C_{sys}, this level of detail is preferred. The following is largely based on Kernighan and Ritchie [47], section A.13 and the formal YACC and LEX grammars of Norrish [74, 96].

constant =
  integer-constant
struct-or-union =
  `struct` | `union`
unary-operator =
  `&` | `*` | `+` | `-` | `~` | `!`
type-qualifier =
  `const` | `volatile`
type-qualifier-list =
  type-qualifier |
  type-qualifier-list type-qualifier
pointer =
  `*` *type-qualifier-list option* |
  `*` *type-qualifier-list option pointer*
storage-class-specifier =
  `auto` |
  `register` |
  `static` |
  `extern` |
  `typedef`
struct-declarator =
  declarator |
  declarator option `:` *constant*
struct-declarator-list =
  struct-declarator |
  struct-declarator-list `,` *struct-declarator*
struct-declaration =
  specifier-qualifier-list struct-declarator-list `;`

struct-declaration-list =
  struct-declaration |
  struct-declaration-list struct-declaration
parameter-declaration =
  declaration-specifiers declarator |
  declaration-specifiers abstract-declarator option
parameter-list =
  parameter-declaration |
  parameter-list `,` *parameter-declaration*
direct-declarator =
  identifier |
  `(` *declarator* `)` |
  direct-declarator `[` *constant option* `]` |
  direct-declarator `(` *parameter-list* `)`
declarator =
  pointer option direct-declarator
declaration-specifiers =
  storage-class-specifier declaration-specifiers option |
  type-specifier declaration-specifiers option |
  type-qualifier declaration-specifiers option
struct-or-union-specifier =
  struct-or-union identifier option `{` *struct-declaration-list* `}` |
  struct-or-union identifier

type-specifier =
  `void` | `char` | `short` | `int` | `long` | `signed` | `unsigned` |
  struct-or-union-specifier
specifier-qualifier-list =
  type-specifier specifier-qualifier-list option |
  type-qualifier specifier-qualifier-list option
direct-abstract-declarator =
  `(` *abstract-declarator* `)` |
  direct-abstract-declarator option `[` *constant option* `]` |
  direct-abstract-declarator option `(` *parameter-list option* `)`
abstract-declarator =
  pointer |
  pointer option direct-abstract-declarator
type-name =
  specifier-qualifier-list abstract-declarator option
primary-expression =
  identifier |
  constant |
  `(` *expression* `)`
postfix-expression =
  primary-expression |
  postfix-expression `[` *expression* `]` |
  postfix-expression `.` *identifier*|
  postfix-expression `->` *identifier*
unary-expression =

postfix-expression |
unary-operator cast-expression |
`sizeof` *unary-expression* |
`sizeof (` *type-name* `)`
cast-expression =
unary-expression |
`(` *type-name* `)` *cast-expression*
multiplicative-expression =
cast-expression |
multiplicative-expression `*` *cast-expression* |
multiplicative-expression `/` *cast-expression* |
multiplicative-expression `%` *cast-expression*
additive-expression =
multiplicative-expression |
additive-expression `+` *multiplicative-expression* |
additive-expression `-` *multiplicative-expression*
shift-expression =
additive-expression |
shift-expression `<<` *additive-expression* |
shift-expression `>>` *additive-expression*
relational-expression =
shift-expression |
relational-expression `<` *shift-expression* |
relational-expression `>` *shift-expression* |
relational-expression `<=` *shift-expression* |
relational-expression `>=` *shift-expression*
equality-expression =
relational-expression |
equality-expression `==` *relational-expression* |
equality-expression `!=` *relational-expression*
and-expression =
equality-expression |
and-expression `&` *equality-expression*
exclusive-or-expression =
and-expression |
exclusive-or-expression `^` *and-expression*
inclusive-or-expression =
exclusive-or-expression |
inclusive-or-expression `|` *exclusive-or-expression*
logical-and-expression =
inclusive-or-expression |
logical-and-expression `&&` *inclusive-or-expression*
logical-or-expression =
logical-and-expression |
logical-or-expression `||` *logical-and-expression*
constant-expression =
conditional-expression
conditional-expression =
logical-or-expression |

logical-or-expression `?` *conditional-expression* `:` *conditional-expression*
expression =
Expression conditional-expression
init-declarator =
declarator |
declarator `=` *expression*
init-declarator-list =
init-declarator |
init-declarator-list `,` *init-declarator*
c-declaration =
declaration-specifiers init-declarator-list option
declaration-list =
c-declaration |
declaration-list c-declaration
argument-expression-list =
expression |
argument-expression-list `,` *expression*
jump-statement =
`continue ;` |
`break ;` |
`return` *expression option* `;`
call-expression =
identifier `(` *argument-expression-list option* `)`
expression-statement =
unary-expression `=` *expression* `;` |
unary-expression `=` *call-expression* `;` |
call-expression `;` |
unary-expression `++ ;` |
unary-expression `-- ;` |
`;`
for-expression-list =
expression-statement auxupd option |
expression-statement auxupd option `,` *for-expression-list*
iteration-statement =
`while (` *expression* `)` *statement* |
`while (` *expression* `)` *annotation statement* |
`for (` *for-expression-list option* `;` *expression option* `;` *for-expression-list option* `)`
statement
selection-statement =
`if (` *expression* `)` *statement* |
`if (` *expression* `)` *statement* `else` *statement*
statement-list =
statement |
statement-list statement
compound-statement =
`{` *declaration-list option statement-list option* `}`
statement =
expression-statement |
compound-statement |

selection-statement |
iteration-statement |
jump-statement |
auxupd

function-definition =
declaration-specifiers option declarator declaration-list option compound-statement

external-declaration =
function-definition |
c-declaration

translation-unit =
external-declaration |
translation-unit external-declaration

Appendix B

Type description functions

In this appendix, we give the full Isabelle/HOL definitions for the type description functions in §5.2, where they were omitted. Each definition appears in a table with a reference to the original definition in which it was introduced. Many of these functions can also be expressed in a concise fashion using higher-order map and fold functions for type descriptions, however the explicit primitive recursive definitions are somewhat clearer and easier to understand.

map-td f (TypDesc st nm)	$\equiv$	TypDesc (map-td-struct f st) nm
map-td-struct f (TypScalar n $algn$ d)	$\equiv$	TypScalar n $algn$ (f n $algn$ d)
map-td-struct f (TypAggregate xs)	$\equiv$	TypAggregate (map-td-list f xs)
map-td-list f []	$\equiv$	[]
map-td-list f (x·xs)	$\equiv$	map-td-pair f x·map-td-list f xs
map-td-pair f $\langle t,n\rangle$	$\equiv$	$\langle$map-td f t,$n\rangle$

Table B.1: map-td definition (5.2.4).

size-td (TypDesc st nm)	$\equiv$	size-td-struct st
size-td-struct (TypScalar n $algn$ d)	$\equiv$	n
size-td-struct (TypAggregate xs)	$\equiv$	size-td-list xs
size-td-list []	$\equiv$	0
size-td-list (x·xs)	$\equiv$	size-td-pair x + size-td-list xs
size-td-pair $\langle t,n\rangle$	$\equiv$	size-td t

Table B.2: size-td definition (5.2.5).

align-td (TypDesc st nm)	$\equiv$	align-td-struct st
align-td-struct (TypScalar n $algn$ d)	$\equiv$	$algn$
align-td-struct (TypAggregate xs)	$\equiv$	align-td-list xs
align-td-list []	$\equiv$	0
align-td-list (x·xs)	$\equiv$	max (align-td-pair x) (align-td-list xs)
align-td-pair $\langle t,n \rangle$	$\equiv$	align-td t

Table B.3: align-td definition (5.2.5).

lookup (TypDesc st nm) f m	$\equiv$	if f = [] then $\lfloor$(TypDesc st nm, m)$\rfloor$ else lookup-struct st f m
lookup-struct (TypScalar n $algn$ d) f m	$\equiv$	$\bot$
lookup-struct (TypAggregate ts) (f·fs) m	$\equiv$	lookup-list ts (f·fs) m
lookup-list [] f m	$\equiv$	$\bot$
lookup-list (x·xs) (f·fs) m	$\equiv$	case lookup-pair x (f·fs) m of $\bot \Rightarrow$ lookup-list xs (f·fs) (m + size-td (dt-fst x)) | $\lfloor y \rfloor \Rightarrow \lfloor y \rfloor$
lookup-pair $\langle t,nm \rangle$ (f·fs) m	$\equiv$	if $nm = f$ then lookup t fs m else $\bot$

Table B.4: lookup definition (5.2.6).

td-set t	$\equiv$	td-set-offset t 0
td-set-offset (TypDesc st nm) m	$\equiv$	{(TypDesc st nm, m)} $\cup$ td-set-struct st m
td-set-struct (TypScalar n $algn$ d) m	$\equiv$	$\emptyset$
td-set-struct (TypAggregate xs) m	$\equiv$	td-set-list xs m
td-set-list [] m	$\equiv$	$\emptyset$
td-set-list (x·xs) m	$\equiv$	td-set-pair x m $\cup$ td-set-list xs (m + size-td (dt-fst x))
td-set-pair $\langle t,nm \rangle$ m	$\equiv$	td-set-offset t m

Table B.5: td-set definition (5.2.7).

access-ti (TypDesc st nm)	$\equiv$	access-ti-struct st
access-ti-struct (TypScalar n $algn$ d)	$\equiv$	field-access d
access-ti-struct (TypAggregate xs)	$\equiv$	access-ti-list xs
access-ti-list []	$\equiv$	$\lambda v\ bs.$ []
access-ti-list (x·xs)	$\equiv$	$\lambda v\ bs.$ access-ti-pair x v (take (size-td (dt-fst x)) bs) @ access-ti-list xs v (drop (size-td (dt-fst x)) bs)
access-ti-pair $\langle t,nm \rangle$	$\equiv$	access-ti t

Table B.6: access-ti definition (5.2.9).

update-ti t	$\equiv$	λbs. if $\lvert bs\rvert$ = size-td t then update-ti-val t bs else id
update-ti-val (TypDesc st nm)	$\equiv$	update-ti-struct st
update-ti-struct (TypScalar n $algn$ d)	$\equiv$	field-update d
update-ti-struct (TypAggregate xs)	$\equiv$	update-ti-list xs
update-ti-list []	$\equiv$	λbs. id
update-ti-list (x·xs)	$\equiv$	$\lambda bs\ v$. update-ti-pair x (take (size-td (dt-fst x)) bs) (update-ti-list xs (drop (size-td (dt-fst x)) bs) v)
update-ti-pair $\langle t,nm\rangle$	$\equiv$	update-ti-val t

Table B.7: update-ti definition (5.2.9).

wf-desc (TypDesc ts n)	$\equiv$	wf-desc-struct ts
wf-desc-struct (TypScalar n $algn$ d)	$\equiv$	True
wf-desc-struct (TypAggregate ts)	$\equiv$	wf-desc-list ts
wf-desc-list []	$\equiv$	True
wf-desc-list (x·xs)	$\equiv$	wf-desc-pair x $\wedge$ dt-snd x $\notin$ dt-snd ‘ set xs $\wedge$ wf-desc-list xs
wf-desc-pair $\langle x,n\rangle$	$\equiv$	wf-desc x

Table B.8: wf-desc definition (5.2.12).

wf-size-desc (TypDesc ts n)	$\equiv$	wf-size-desc-struct ts
wf-size-desc-struct (TypScalar n $algn$ d)	$\equiv$	$0 < n$
wf-size-desc-struct (TypAggregate ts)	$\equiv$	$ts \neq$ [] $\wedge$ wf-size-desc-list ts
wf-size-desc-list []	$\equiv$	True
wf-size-desc-list (x·xs)	$\equiv$	wf-size-desc-pair x $\wedge$ wf-size-desc-list xs
wf-size-desc-pair $\langle x,n\rangle$	$\equiv$	wf-size-desc x

Table B.9: wf-size-desc definition (5.2.13).

```
wf-field-desc t ≡ wf-lf (lf-set t [])

record α leaf-desc  =  lf-fd :: α field-desc
                       lf-sz :: nat
                       lf-fn :: qualified-field-name

lf-set (TypDesc st nm) fn              ≡ lf-set-struct st fn
lf-set-struct (TypScalar n algn d) fn  ≡ {(|lf-fd = d, lf-sz = n, lf-fn = fn|)}
lf-set-struct (TypAggregate xs) fn     ≡ lf-set-list xs fn
lf-set-list [] fn                      ≡ ∅
lf-set-list (x·xs) fn                  ≡ lf-set-pair x fn ∪ lf-set-list xs fn
lf-set-pair ⟨t,n⟩ fn                   ≡ lf-set t (fn @ [n])

wf-lf D ≡
∀ x. x ∈ D ⟶
    fd-cons-desc (lf-fd x) (lf-sz x) ∧
    (∀ y. y ∈ D ⟶
        lf-fn y ≠ lf-fn x ⟶
        fu-commutes (field-update (lf-fd x)) (field-update (lf-fd y)) ∧
        fa-fu-ind (lf-fd x) (lf-fd y) (lf-sz y) (lf-sz x))

fu-commutes f g ≡ ∀ v bs bs'. f bs (g bs' v) = g bs' (f bs v)

fa-fu-ind d d' n n' ≡
∀ v bs bs'.
  |bs| = n ⟶
  |bs'| = n' ⟶
  field-access d (field-update d' bs v) bs' = field-access d v bs'
```

Table B.10: wf-field-desc definition (5.2.15).

```
norm-tu (TypDesc st nm)            ≡ norm-tu-struct st
norm-tu-struct (TypScalar n aln f) ≡ f
norm-tu-struct (TypAggregate xs)   ≡ norm-tu-list xs
norm-tu-list []                    ≡ λbs. []
norm-tu-list (x·xs)                ≡ λbs. norm-tu-pair x (take (size-td-pair x) bs) @
                                          norm-tu-list xs (drop (size-td-pair x) bs)
norm-tu-pair ⟨t,n⟩                 ≡ norm-tu t
```

Table B.11: norm-tu definition (5.2.23).

typ-slice (TypDesc st nm) m	$\equiv$	typ-slice-struct st m @ [if $m = 0$ then (TypDesc st nm, True) else (TypDesc st nm, False)]
typ-slice-struct (TypScalar n $algn$ d) m	$\equiv$	[]
typ-slice-struct (TypAggregate xs) m	$\equiv$	typ-slice-list xs m
typ-slice-list [] m	$\equiv$	[]
typ-slice-list (x·xs) m	$\equiv$	if $m <$ size-td (dt-fst x) $\vee$ $xs = []$ then typ-slice-pair x m else typ-slice-list xs ($m -$ size-td (dt-fst x))
typ-slice-pair $\langle t,n \rangle$ m	$\equiv$	typ-slice t m

Table B.12: typ-slice definition (5.3.1).

field-names (TypDesc st nm) t	$\equiv$	if $t =$ export-uinfo (TypDesc st nm) then [[]] else field-names-struct st t
field-names-struct (TypScalar m $algn$ d) t	$\equiv$	[]
field-names-struct (TypAggregate xs) t	$\equiv$	field-names-list xs t
field-names-list [] t	$\equiv$	[]
field-names-list (x·xs) t	$\equiv$	field-names-pair x t @ field-names-list xs t
field-names-pair $\langle s,f \rangle$ t	$\equiv$	map (op # f) (field-names s t)

Table B.13: field-names definition (5.3.6).

Appendix C

Type combinator proof rules

The rules for α::*mem-type* instantiation of type information built with the combinators of §5.2.4 are given here, together with a summary of the corresponding Isabelle/HOL proofs. Where definitions are required, this appendix should be read in conjunction with Appendix B.

[MaxSize]

Theorem C.0.1. *The* [MaxSize] *axiom can be shown for the combinators by the simplifier with the following rewrite rules, Defn. 5.2.19 and Defn. 5.2.10*[1]*:*

$$\mathsf{aggregate}\ (\mathsf{empty\text{-}typ\text{-}info}\ tn) \qquad [\textsc{AgEmpty}]$$

$$\mathsf{aggregate}\ (\mathsf{ti\text{-}typ\text{-}combine}\ t\ f\ g\ fn\ ti) \qquad [\textsc{AgTyp}]$$

$$\mathsf{aggregate}\ (\mathsf{ti\text{-}pad\text{-}combine}\ n\ ti) \qquad [\textsc{AgPad}]$$

$$\mathsf{aggregate}\ (\mathsf{ti\text{-}typ\text{-}pad\text{-}combine}\ t\ f\ g\ fn\ ti) \qquad [\textsc{AgTypPad}]$$

$$\mathsf{align\text{-}td}\ (\mathsf{empty\text{-}typ\text{-}info}\ tn) = 0 \qquad [\textsc{AlgnEmpty}]$$

$$\frac{\mathsf{aggregate}\ ti}{\begin{array}{l}\mathsf{align\text{-}td}\ (\mathsf{ti\text{-}typ\text{-}combine}\ t\ f\ g\ fn\ ti) \\ = \mathsf{max}\ (\mathsf{align\text{-}td}\ ti)\ (\mathsf{align\text{-}td}\ \mathsf{TYPE}(\alpha)_\tau)\end{array}} \qquad [\textsc{AlgnTyp}]$$

$$\frac{\mathsf{aggregate}\ ti}{\mathsf{align\text{-}td}\ (\mathsf{ti\text{-}pad\text{-}combine}\ n\ ti) = \mathsf{align\text{-}td}\ ti} \qquad [\textsc{AlgnPad}]$$

[1] The aggregate predicate asserts that the type description has a TypAggregate constructor at the root.

$$\frac{\mathsf{aggregate}\ ti}{\begin{array}{l}\mathsf{align\text{-}td}\ (\mathsf{ti\text{-}typ\text{-}pad\text{-}combine}\ t\ f\ g\ fn\ ti)\\ =\ \mathsf{max}\ (\mathsf{align\text{-}td}\ ti)\ (\mathsf{align\text{-}td}\ \mathsf{TYPE}(\alpha)_\tau)\end{array}} \quad [\textsc{AlgnTypPad}]$$

$$\frac{\mathsf{aggregate}\ ti}{\mathsf{align\text{-}td}\ (\mathsf{final\text{-}pad}\ ti) = \mathsf{align\text{-}td}\ ti} \quad [\textsc{AlgnFinal}]$$

$$\mathsf{size\text{-}td}\ (\mathsf{empty\text{-}typ\text{-}info}\ tn) = 0 \quad [\textsc{SzEmpty}]$$

$$\frac{\mathsf{aggregate}\ ti}{\begin{array}{l}\mathsf{size\text{-}td}\ (\mathsf{ti\text{-}typ\text{-}combine}\ t\ f\ g\ fn\ ti)\\ =\ \mathsf{size\text{-}td}\ ti\ +\ \mathsf{size\text{-}td}\ \mathsf{TYPE}(\beta)_\tau\end{array}} \quad [\textsc{SzTyp}]$$

$$\frac{\mathsf{aggregate}\ ti}{\mathsf{size\text{-}td}\ (\mathsf{ti\text{-}pad\text{-}combine}\ n\ ti) = \mathsf{size\text{-}td}\ ti\ +\ n} \quad [\textsc{SzPad}]$$

$$\frac{\mathsf{aggregate}\ ti}{\begin{array}{l}\mathsf{size\text{-}td}\ (\mathsf{ti\text{-}typ\text{-}pad\text{-}combine}\ t\ f\ g\ fn\ ti) = \mathsf{let}\ k = \mathsf{size\text{-}td}\ ti\ \mathsf{in}\\ k\ +\ \mathsf{size\text{-}td}\ \mathsf{TYPE}(\beta)_\tau\ +\ \mathsf{padup}\ 2^{\mathsf{align\text{-}td}\ \mathsf{TYPE}(\beta)_\tau}\ k\end{array}} \quad [\textsc{SzTypPad}]$$

$$\frac{\mathsf{aggregate}\ ti}{\begin{array}{l}\mathsf{size\text{-}td}\ (\mathsf{final\text{-}pad}\ ti) = \mathsf{let}\ k = \mathsf{size\text{-}td}\ ti\ \mathsf{in}\\ k\ +\ \mathsf{padup}\ 2^{\mathsf{align\text{-}td}\ ti}\ k\end{array}} \quad [\textsc{SzFinal}]$$

Proof. The size and alignment values are evaluated by rewriting and then Isabelle performs arithmetic evaluation for the constants.

[AgEmpty], [AgTyp], [AgPad], [AgTypPad], [SzEmpty], [SzTyp], [SzPad], [SzTypPad], [SzFinal], [AlgnEmpty], [AlgnTyp], [AlgnPad], [AlgnTypPad] and [AlgnFinal] follow trivially from definitions and simplification.

□

[AlignDvdSize]

Theorem C.0.2. [AlignDvdSize] *is given by Defn. 5.2.10 and:*

$$\frac{\mathsf{aggregate}\ ti}{2^{\mathsf{align\text{-}td}\ (\mathsf{final\text{-}pad}\ ti)}\ \mathsf{dvd}\ \mathsf{size\text{-}td}\ (\mathsf{final\text{-}pad}\ ti)} \quad [\text{ADSFinal}]$$

Proof. First unfold final-pad. In the case where there is a padding field at the end of the structure, simplifying with [SzPad] and observing $0 < x \Longrightarrow x\ \mathsf{dvd}\ y\ +\ \mathsf{padup}\ x\ y$ gives the rule. When there is no padding, it is sufficient to observe $0 < x \Longrightarrow (\mathsf{padup}\ x\ y = 0) = (x\ \mathsf{dvd}\ y)$, since $\mathsf{padup}\ 2^{\mathsf{align\text{-}td}\ ti}\ (\mathsf{size\text{-}td}\ ti) = 0$. □

[AlignField]

Theorem C.0.3. *The combinator proofs rule for* [AlignField] *are:*

$$\frac{}{\mathsf{align\text{-}field}\ (\mathsf{empty\text{-}typ\text{-}info}\ tn)} \quad [\text{AFEmpty}]$$

$$\frac{\mathsf{align\text{-}field}\ ti}{\mathsf{align\text{-}field}\ (\mathsf{ti\text{-}pad\text{-}combine}\ n\ ti)} \quad [\text{AFPad}]$$

$$\frac{\mathsf{align\text{-}field}\ ti \qquad 2^{\mathsf{align\text{-}td}\ \mathsf{TYPE}(\alpha)_\tau}\ \mathsf{dvd}\ \mathsf{size\text{-}td}\ ti}{\mathsf{align\text{-}field}\ (\mathsf{ti\text{-}typ\text{-}combine}\ t\ f\ g\ fn\ ti)} \quad [\text{AFTyp}]$$

$$\frac{\mathsf{align\text{-}field}\ ti \qquad \mathsf{aggregate}\ ti}{\mathsf{align\text{-}field}\ (\mathsf{ti\text{-}typ\text{-}pad\text{-}combine}\ t\ f\ g\ fn\ ti)} \quad [\text{AFTypPad}]$$

$$\frac{\mathsf{align\text{-}field}\ ti}{\mathsf{align\text{-}field}\ (\mathsf{final\text{-}pad}\ ti)} \quad [\text{AFFinal}]$$

where

$$\mathsf{align\text{-}field}\ ti \equiv \forall f\ s\ n.\ ti{\triangleright}f = \lfloor(s,\ n)\rfloor \longrightarrow 2^{\mathsf{align\text{-}td}\ s}\ \mathsf{dvd}\ n$$

Proof. [AFEmpty] is trivially shown by unfolding and simplification. [AFPad] is a consequence of the following result, with the alignment of padding fields being defined as 1 allowing the dvd side-condition to be discharged:

$$\frac{\mathsf{align\text{-}field}\ ti \qquad \mathsf{align\text{-}field}\ t \qquad 2^{\mathsf{align\text{-}td}\ t}\ \mathsf{dvd}\ \mathsf{size\text{-}td}\ ti}{\mathsf{align\text{-}field}\ (\mathsf{extend\text{-}ti}\ ti\ t\ fn)} \quad [\text{AFExtend}]$$

There are three cases for $\mathsf{extend\text{-}ti}\ ti\ t\ fn{\triangleright}f = \lfloor(s,\ n)\rfloor$. $fn = []$ is trivial as there is no offset. If the field is in ti, the alignment result is obtained from $\mathsf{align\text{-}field}\ ti$. Otherwise, it is arrived at by the extended field fn, and we need to show $2^{\mathsf{align\text{-}td}\ s}\ \mathsf{dvd}\ \mathsf{size\text{-}td}\ ti$ to use $\mathsf{align\text{-}field}\ t$. Since alignments are powers-of-two, and $\mathsf{align\text{-}td}\ s \leq \mathsf{align\text{-}td}\ t$, it is evident that $2^{\mathsf{align\text{-}td}\ s}\ \mathsf{dvd}\ 2^{\mathsf{align\text{-}td}\ t}$ and the desired result is given by dvd transitivity.

[AFFinal] is a direct result of [AFPad]. [AFTyp] can be seen from [AFExtend] and adjust-ti not affecting alignment or offset. [AFTypPad] follows the definition simplification with [AFTyp] and [AFPad], and the dvd side-condition handled as in Thm. C.0.2. □

[Upd]

Definition C.0.1. The proof rules for [Upd] make use of an additional constant:

$$\begin{array}{l}\mathsf{fu\text{-}eq\text{-}mask}\ ti\ f \equiv \\ \forall\, bs\ v\ v'. \\ \quad |bs| = \mathsf{size\text{-}td}\ ti \longrightarrow \\ \quad \mathsf{update\text{-}ti}\ ti\ bs\ (f\ v) = \mathsf{update\text{-}ti}\ ti\ bs\ (f\ v')\end{array}$$

This is [Upd] modulo a "mask" f. The intuition behind the mask is that [Upd] only holds for the structure as a whole. A field-wise decomposition proof involves progressively masking out fields as their combinators are reduced.

Theorem C.0.4. *The combinator proof rules for* [Upd] *can then be stated as:*

$$\frac{\exists\, k.\ \forall\, v.\ f\ v = k}{\mathsf{fu\text{-}eq\text{-}mask}\ t\ f} \qquad [\textsc{FuEqConst}]$$

$$\frac{\mathsf{fu\text{-}eq\text{-}mask}\ ti\ f \qquad \mathsf{aggregate}\ ti}{\mathsf{fu\text{-}eq\text{-}mask}\ (\mathsf{ti\text{-}pad\text{-}combine}\ n\ ti)\ f} \qquad [\textsc{FuEqPad}]$$

$$\frac{\begin{array}{c}\mathsf{fu\text{-}eq\text{-}mask}\ ti\ (\lambda v.\ g\ (f\ \mathsf{arbitrary})\ (h\ v)) \\ \bigwedge u\ v.\ f\ (g\ u\ v) = u \\ \bigwedge u\ u'\ v.\ g\ u\ (g\ u'\ v) = g\ u\ v \qquad \bigwedge v.\ g\ (f\ v)\ v = v \\ \mathsf{fu\text{-}commutes}\ (\mathsf{update\text{-}ti}\ ti)\ g \qquad \mathsf{aggregate}\ ti\end{array}}{\mathsf{fu\text{-}eq\text{-}mask}\ (\mathsf{ti\text{-}typ\text{-}combine}\ t\ f\ g\ fn\ ti)\ h} \qquad [\textsc{FuEqTyp}]$$

$$\frac{\begin{array}{c}\mathsf{fu\text{-}eq\text{-}mask}\ ti\ (\lambda v.\ g\ (f\ \mathsf{arbitrary})\ (h\ v)) \\ \bigwedge u\ v.\ f\ (g\ u\ v) = u \\ \bigwedge u\ u'\ v.\ g\ u\ (g\ u'\ v) = g\ u\ v \qquad \bigwedge v.\ g\ (f\ v)\ v = v \\ \mathsf{fu\text{-}commutes}\ (\mathsf{update\text{-}ti}\ ti)\ g \qquad \mathsf{aggregate}\ ti\end{array}}{\mathsf{fu\text{-}eq\text{-}mask}\ (\mathsf{ti\text{-}typ\text{-}pad\text{-}combine}\ t\ f\ g\ fn\ ti)\ h} \qquad [\textsc{FuEqTypPad}]$$

$$\frac{\mathsf{fu\text{-}eq\text{-}mask}\ ti\ f \qquad \mathsf{aggregate}\ ti}{\mathsf{fu\text{-}eq\text{-}mask}\ (\mathsf{final\text{-}pad}\ ti)\ f} \qquad [\textsc{FuEqFinal}]$$

$$\frac{|bs| = \mathsf{size\text{-}td}\ ti \qquad \mathsf{fu\text{-}eq\text{-}mask}\ ti\ \mathsf{id}}{\mathsf{update\text{-}ti}\ ti\ bs\ v = \mathsf{update\text{-}ti}\ ti\ bs\ w} \qquad [\textsc{FuEqUpd}]$$

with additional proof rules:

$$\frac{}{\textsf{fu-commutes}\ (\textsf{update-ti}\ (\textsf{empty-typ-info}\ tn))\ f} \quad [\textsc{FCEmpty}]$$

$$\frac{\textsf{fu-commutes}\ (\textsf{update-ti}\ ti)\ f}{\textsf{fu-commutes}\ (\textsf{update-ti}\ (\textsf{ti-pad-combine}\ n\ ti))\ f} \quad [\textsc{FCPad}]$$

$$\frac{\begin{array}{c}\textsf{fu-commutes}\ (\textsf{update-ti}\ ti)\ h \\ \bigwedge v\ u\ u'.\ g\ u\ (h\ u'\ v) = h\ u'\ (g\ u\ v) \\ \bigwedge u\ v.\ f\ (h\ u\ v) = f\ v \qquad \bigwedge u\ v.\ f\ (g\ u\ v) = u \\ \bigwedge u\ u'\ v.\ g\ u\ (g\ u'\ v) = g\ u\ v \qquad \bigwedge v.\ g\ (f\ v)\ v = v\end{array}}{\textsf{fu-commutes}\ (\textsf{update-ti}\ (\textsf{ti-typ-combine}\ t\ f\ g\ fn\ ti))\ h} \quad [\textsc{FCTyp}]$$

$$\frac{\begin{array}{c}\textsf{fu-commutes}\ (\textsf{update-ti}\ ti)\ h \\ \bigwedge v\ u\ u'.\ g\ u\ (h\ u'\ v) = h\ u'\ (g\ u\ v) \\ \bigwedge u\ v.\ f\ (h\ u\ v) = f\ v \qquad \bigwedge u\ v.\ f\ (g\ u\ v) = u \\ \bigwedge u\ u'\ v.\ g\ u\ (g\ u'\ v) = g\ u\ v \qquad \bigwedge v.\ g\ (f\ v)\ v = v\end{array}}{\textsf{fu-commutes}\ (\textsf{update-ti}\ (\textsf{ti-typ-pad-combine}\ t\ f\ g\ fn\ ti))\ h} \quad [\textsc{FCTypPad}]$$

Proof. The [Upd] proof obligation is discharged by first applying [FuEqUpd], and reducing the type information with [FuEqFinal] and [FuEqTypPad] until there is only the empty type information remaining. [FuEqConst] completes a proof, with the existential instantiated with arbitrary. The intuition here is that the mask is built up from id to $\lambda v.$ arbitrary as each field is considered. The fu-commutes side-conditions are discharged by applying [FCTypPad] and [FCEmpty]. Other assumptions are handled automatically by rewriting using the **record** generated simplification set. The entire process should run in $O(n^2)$, as each field is compared against those preceding it in the fu-commutes side-condition proof branch.

[FuEqUpd], [FuEqConst] and [FuEqPad] can be shown from definitions. [FuEqFinal] is a result of [FuEqPad]. For [FuEqTyp], in the assumption we have a simple equality of update-ti terms for *ti*, but in the goal the extended field results in a comparison of terms consisting of the sequential composition of update-ti, for the new field, followed by update-ti for *ti*. For the new field, the assumed access/update properties and adjust-ti definition give that update-ti (adjust-ti $t\ f\ g$) $bs\ v = g$ (update-ti $t\ bs$ ($f\ v$)) v. Then the goal and assumption have a similar form, differing in the applied argument to g in the mask (the assumption has f arbitrary while goal has an update-ti term. This can be reconciled by using [Upd] for α where t::α::*mem-type itself* and fu-commutes to swap the *ti* and new field updates to allow the assumption to be used. [FuEqTypPad] then follows from [FuEqTyp] and [FuEqPad].

[FCEmpty] is trivial from the definitions. For [FCPadEmpty], we can show:

$$\frac{\begin{array}{c}\mathsf{fu\text{-}commutes}\ (\mathsf{update\text{-}ti}\ s)\ h \\ \mathsf{fu\text{-}commutes}\ (\mathsf{update\text{-}ti}\ t)\ h\end{array}}{\mathsf{fu\text{-}commutes}\ (\mathsf{update\text{-}ti}\ (\mathsf{extend\text{-}ti}\ s\ t\ fn))\ h} \quad [\text{FCExtend}]$$

and fu-commutes holds for the identity padding field access and update functions. [FCTyp] is derived from [FCExtend] and update-ti (adjust-ti $t\ f\ g$) $bs\ v = g$ (update-ti $t\ bs$ ($f\ v$)) v (as above). Here the additional assumptions relating f, g and h provide commutativity for the extension field. The proof rules are completed with [FCTypPad], following from [FCPad] and [FCTyp].

□

[WFDesc]

Theorem C.0.5. *The combinator proof rules for* [WFDesc] *are:*

$$\mathsf{wf\text{-}desc}\ (\mathsf{empty\text{-}typ\text{-}info}\ tn) \quad [\text{WFDescEmpty}]$$

$$\frac{\mathsf{wf\text{-}desc}\ ti}{\mathsf{wf\text{-}desc}\ (\mathsf{ti\text{-}pad\text{-}combine}\ n\ ti)} \quad [\text{WFDescPad}]$$

$$\frac{\mathsf{wf\text{-}desc}\ ti \qquad fn \notin \mathsf{set}\ (\mathsf{field\text{-}names\text{-}list}\ ti)}{\mathsf{wf\text{-}desc}\ (\mathsf{ti\text{-}typ\text{-}combine}\ t\ f\ g\ fn\ ti)} \quad [\text{WFDescTyp}]$$

$$\frac{\begin{array}{c}\mathsf{wf\text{-}desc}\ ti \\ fn \notin \mathsf{set}\ (\mathsf{field\text{-}names\text{-}list}\ ti) \qquad \mathsf{hd}\ fn \neq \mathsf{CHR}\ ''!''\end{array}}{\mathsf{wf\text{-}desc}\ (\mathsf{ti\text{-}typ\text{-}pad\text{-}combine}\ t\ f\ g\ fn\ ti)} \quad [\text{WFDescTypPad}]$$

$$\frac{\mathsf{wf\text{-}desc}\ ti}{\mathsf{wf\text{-}desc}\ (\mathsf{final\text{-}pad}\ ti)} \quad [\text{WFDescFinal}]$$

Proof. By unfolding definitions, wf-desc (map-td $f\ t$) = wf-desc t for adjust-ti and $s \neq [] \Longrightarrow$ foldl op @ $s\ xs \notin$ set xs giving that padding field names do not clash in ti-pad-combine. The hd $fn \neq$ CHR $''!''$ assumption prevents the new field name in ti-typ-pad-combine from conflicting with the name given to any potential padding field. □

[WFSizeDesc]

Theorem C.0.6. *The combinator proof rules for* [WFSizeDesc] *are:*

$$\frac{\text{wf-size-desc } ti}{\text{wf-size-desc (ti-typ-combine } t\ f\ g\ fn\ ti)} \quad [\text{WFSzTyp}]$$

$$\frac{\text{wf-size-desc } ti \qquad 0 < n}{\text{wf-size-desc (ti-pad-combine } n\ ti)} \quad [\text{WFSzPad}]$$

$$\frac{\text{wf-size-desc } ti}{\text{wf-size-desc (ti-typ-pad-combine } t\ f\ g\ fn\ ti)} \quad [\text{WFSzTypPad}]$$

$$\text{wf-size-desc (ti-typ-pad-combine } t\ f\ g\ fn\ (\text{empty-typ-info } tn)) \quad [\text{WFSzEmpty}]$$

$$\frac{\text{wf-size-desc } ti}{\text{wf-size-desc (final-pad } ti)} \quad [\text{WFSzFinal}]$$

Proof. There is no rule for empty-typ-info on its own, as this has a zero size. Instead [WFSzEmpty] is used when only a single field extension remains after rule application.

The proofs are again by simplifying with definitions and wf-size-desc (map-td $f\ t$) = wf-size-desc t for adjust-ti. □

[WFFD]

Theorem C.0.7. *The combinator proof rules for* [WFFD] *are:*

$$\text{wf-lf } \emptyset \quad [\text{WFLFEmpty}]$$

$$\frac{\text{wf-lf (lf-set } ti\ [])}{\text{wf-lf (lf-set (ti-pad-combine } n\ ti)\ [])} \quad [\text{WFLFPad}]$$

$$\frac{\begin{array}{c}\text{wf-lf (lf-set } ti\ []) \qquad fn \notin \text{set (field-names-list } ti) \\ \bigwedge v.\ g\ (f\ v)\ v = v \qquad \bigwedge w\ u\ v.\ g\ w\ (g\ u\ v) = g\ w\ v \\ \bigwedge w\ v.\ f\ (g\ w\ v) = w \\ \text{g-ind (lf-set } ti\ []) \ g \qquad \text{f-ind } f\ (\text{lf-fd `lf-set } ti\ []) \\ \text{fa-ind (lf-fd `lf-set } ti\ []) \ g\end{array}}{\text{wf-lf (lf-set (ti-typ-combine } t\ f\ g\ fn\ ti)\ [])} \quad [\text{WFLFTyp}]$$

$$\frac{\begin{array}{c}\text{wf-lf (lf-set } ti\ []) \\ fn \notin \text{set (field-names-list } ti) \qquad \text{hd } fn \neq \text{CHR ''!''} \\ \bigwedge v.\ g\ (f\ v)\ v = v \qquad \bigwedge w\ u\ v.\ g\ w\ (g\ u\ v) = g\ w\ v \\ \bigwedge w\ v.\ f\ (g\ w\ v) = w \\ \text{g-ind (lf-set } ti\ []) \ g \qquad \text{f-ind } f\ (\text{lf-fd `lf-set } ti\ []) \\ \text{fa-ind (lf-fd `lf-set } ti\ []) \ g\end{array}}{\text{wf-lf (lf-set (ti-typ-pad-combine } t\ f\ g\ fn\ ti)\ [])} \quad [\text{WFLFTypPad}]$$

$$\frac{\text{wf-lf (lf-set } ti\ [])}{\text{wf-lf (lf-set (final-pad } ti)\ [])} \quad [\text{WFLFFinal}]$$

where

$$\begin{array}{lcl}\text{g-ind } X\ g & \equiv & \forall x.\ x \in \text{field-update `lf-fd `} X \longrightarrow \text{fu-commutes } x\ g \\ \text{f-ind } f\ X & \equiv & \forall x\ bs\ v.\ x \in X \longrightarrow f\ (\text{field-update } x\ bs\ v) = f\ v \\ \text{fa-ind } X\ g & \equiv & \forall x\ bs\ v.\ x \in X \longrightarrow \text{field-access } x\ (g\ bs\ v) = \text{field-access } x\ v\end{array}$$

with proof rules:

$$\text{g-ind } \emptyset\ g \quad [\text{GIndEmpty}]$$

$$\frac{\text{g-ind (lf-set } ti\ []) \ g}{\text{g-ind (lf-set (ti-pad-combine } n\ ti)\ []) \ g} \quad [\text{GIndPad}]$$

$$\frac{\begin{array}{c}\text{g-ind (lf-set } ti\ []) \ h \\ \bigwedge w\ u\ v.\ g\ w\ (h\ u\ v) = h\ u\ (g\ w\ v) \\ \bigwedge w\ v.\ f\ (h\ w\ v) = f\ v \qquad \bigwedge v.\ g\ (f\ v)\ v = v\end{array}}{\text{g-ind (lf-set (ti-typ-combine } t\ f\ g\ fn\ ti)\ []) \ h} \quad [\text{GIndTyp}]$$

$$\frac{\begin{array}{c}\text{g-ind (lf-set } ti\ []) \ h \\ \bigwedge w\ u\ v.\ g\ w\ (h\ u\ v) = h\ u\ (g\ w\ v) \\ \bigwedge w\ v.\ f\ (h\ w\ v) = f\ v \qquad \bigwedge v.\ g\ (f\ v)\ v = v\end{array}}{\text{g-ind (lf-set (ti-typ-pad-combine } t\ f\ g\ fn\ ti)\ []) \ h} \quad [\text{GIndTypPad}]$$

$$\text{f-ind } f\ \emptyset \quad [\text{FIndEmpty}]$$

$$\frac{\text{f-ind } f\ (\text{lf-fd `lf-set } t\ [])}{\text{f-ind } f\ (\text{lf-fd `lf-set (ti-pad-combine } n\ t)\ [])} \quad [\text{FIndPad}]$$

$$\frac{\textsf{f-ind}\ h\ (\textsf{lf-fd}\ `\ \textsf{lf-set}\ ti\ [])\quad \bigwedge v\ w.\ h\ (g\ w\ v) = h\ v\quad \bigwedge v.\ g\ (f\ v)\ v = v}{\textsf{f-ind}\ h\ (\textsf{lf-fd}\ `\ \textsf{lf-set}\ (\textsf{ti-typ-combine}\ t\ f\ g\ fn\ ti)\ [])}\ [\textsc{FIndTyp}]$$

$$\frac{\textsf{f-ind}\ h\ (\textsf{lf-fd}\ `\ \textsf{lf-set}\ ti\ [])\quad \bigwedge v\ w.\ h\ (g\ w\ v) = h\ v\quad \bigwedge v.\ g\ (f\ v)\ v = v}{\textsf{f-ind}\ h\ (\textsf{lf-fd}\ `\ \textsf{lf-set}\ (\textsf{ti-typ-pad-combine}\ t\ f\ g\ fn\ ti)\ [])}\ [\textsc{FIndTypPad}]$$

$$\textsf{fa-ind}\ \emptyset\ g\ [\textsc{FAIndEmpty}]$$

$$\frac{\textsf{fa-ind}\ (\textsf{lf-fd}\ `\ \textsf{lf-set}\ ti\ [])\ g}{\textsf{fa-ind}\ (\textsf{lf-fd}\ `\ \textsf{lf-set}\ (\textsf{ti-pad-combine}\ n\ ti)\ [])\ g}\ [\textsc{FAIndPad}]$$

$$\frac{\textsf{fa-ind}\ (\textsf{lf-fd}\ `\ \textsf{lf-set}\ ti\ [])\ h\quad \bigwedge v\ w.\ f\ (h\ w\ v) = f\ v\quad \bigwedge v.\ g\ (f\ v)\ v = v}{\textsf{fa-ind}\ (\textsf{lf-fd}\ `\ \textsf{lf-set}\ (\textsf{ti-typ-combine}\ t\ f\ g\ fn\ ti)\ [])\ h}\ [\textsc{FAIndTyp}]$$

$$\frac{\textsf{fa-ind}\ (\textsf{lf-fd}\ `\ \textsf{lf-set}\ ti\ [])\ h\quad \bigwedge v\ w.\ f\ (h\ w\ v) = f\ v\quad \bigwedge v.\ g\ (f\ v)\ v = v}{\textsf{fa-ind}\ (\textsf{lf-fd}\ `\ \textsf{lf-set}\ (\textsf{ti-typ-pad-combine}\ t\ f\ g\ fn\ ti)\ [])\ h}\ [\textsc{FAIndTypPad}]$$

Proof. [WFLFEmpty], [GIndEmpty], [FIndEmpty] and [FAIndEmpty] are trivially true. As in earlier proofs in this appendix, the final-pad and ti-typ-pad-combine derivations build on those for ti-typ-combine and ti-pad-combine, which are the cases we now give.

[WFLFTyp] and [WFLFPad] essentially follow from the assumptions giving local counterparts of wf-lf and the intermediate results:

$$\frac{\textsf{wf-lf}\ (\textsf{lf-set}\ t\ [])\quad \textsf{wf-lf}\ (\textsf{lf-set}\ ti\ [])\quad \textsf{wf-desc}\ t\quad fn \notin \textsf{set}\ (\textsf{field-names-list}\ ti)\quad \textsf{ti-ind}\ (\textsf{lf-set}\ ti\ [])\ (\textsf{lf-set}\ t\ [])}{\textsf{wf-lf}\ (\textsf{lf-set}\ (\textsf{extend-ti}\ ti\ t\ fn)\ [])}\ [\textsc{WFLFExtend}]$$

$$\frac{\textsf{wf-lf}\ (\textsf{lf-set}\ t\ [])\quad \bigwedge v.\ g\ (f\ v)\ v = v\quad \bigwedge bs\ bs'\ v.\ g\ bs\ (g\ bs'\ v) = g\ bs\ v\quad \bigwedge bs\ v.\ f\ (g\ bs\ v) = bs}{\textsf{wf-lf}\ (\textsf{lf-set}\ (\textsf{adjust-ti}\ t\ f\ g)\ [])}\ [\textsc{WFLFUpdate}]$$

where

$$
\begin{array}{l}
\mathsf{ti\text{-}ind}\ X\ Y \equiv \\
\forall\, x\ y.\ x \in X \wedge y \in Y \longrightarrow \\
\qquad \mathsf{fu\text{-}commutes}\ (\mathsf{field\text{-}update}\ (\mathsf{lf\text{-}fd}\ x))\ (\mathsf{field\text{-}update}\ (\mathsf{lf\text{-}fd}\ y)) \wedge \\
\qquad \mathsf{fa\text{-}fu\text{-}ind}\ (\mathsf{lf\text{-}fd}\ x)\ (\mathsf{lf\text{-}fd}\ y)\ (\mathsf{lf\text{-}sz}\ y)\ (\mathsf{lf\text{-}sz}\ x) \wedge \\
\qquad \mathsf{fa\text{-}fu\text{-}ind}\ (\mathsf{lf\text{-}fd}\ y)\ (\mathsf{lf\text{-}fd}\ x)\ (\mathsf{lf\text{-}sz}\ x)\ (\mathsf{lf\text{-}sz}\ y)
\end{array}
$$

[WFLFExtend] can be demonstrated by unfolding definitions and then:

$$\frac{\mathsf{lf\text{-}fn}\ `\ X \cap \mathsf{lf\text{-}fn}\ `\ Y = \emptyset}{\mathsf{wf\text{-}lf}\ (X \cup Y) = (\mathsf{wf\text{-}lf}\ X \wedge \mathsf{wf\text{-}lf}\ Y \wedge \mathsf{ti\text{-}ind}\ X\ Y)}$$

with simplification using the assumed properties. [WFLFUpdate] is a consequence of:

$$\frac{t \in \mathsf{lf\text{-}set}\ (\mathsf{adjust\text{-}ti}\ ti\ f\ g)\ [] \qquad \bigwedge v.\ g\ (f\ v)\ v = v}{\exists\, s.\ s \in \mathsf{lf\text{-}set}\ ti\ [] \wedge \mathsf{lf\text{-}fd}\ t = \mathsf{update\text{-}desc}\ f\ g\ (\mathsf{lf\text{-}fd}\ s)} \qquad [\text{LFUpdate}]$$

[FIndPad], [FIndTyp], [GIndPad], [GIndTyp], [FAIndPad] and [FAIndTyp] are arrived at by a similar process using [LFUpdate].

□

Appendix D

Separation property proofs

theory *Separation* **imports** *TypHeap* **begin**

types $('a,'b)$ *map-assert* $= ('a \rightharpoonup 'b) \Rightarrow$ *bool*
types *heap-assert* $=$ (*addr*,*typ-tag option* $\times$ *byte*) *map-assert*

constdefs *sep-emp* :: $('a,'b)$ *map-assert* $(\square)$
$\square \equiv (op =)$ *empty*

lemma *sep-empD*:
$\square$ *s* $\Longrightarrow$ *s* $=$ *empty*
by (*simp add*: *sep-emp-def*)

lemma *sep-emp-empty* [*simp*]:
$\square$ *empty*
by (*simp add*: *sep-emp-def*)

constdefs *sep-true* :: $('a,'b)$ *map-assert*
sep-true $\equiv \lambda s.$ *True*

lemma *sep-true* [*simp*]:
sep-true s
by (*simp add*: *sep-true-def*)

constdefs *sep-false* :: $('a,'b)$ *map-assert*
sep-false $\equiv \lambda s.$ *False*

lemma *sep-false* [*simp*]:
$\neg$ *sep-false s*
by (*simp add*: *sep-false-def*)

declare *sep-false-def* [*symmetric*, *simp add*]

constdefs *singleton* :: *'a::c-type ptr* ⇒ *'a* ⇒ *heap-state*
singleton p v ≡ *lift-state* (*heap-update p v arbitrary*,*ptr-tag p empty*)

lemma *singleton-dom*:
dom (*singleton p* (*v::'a::mem-type*)) = {*ptr-val p..+size-of TYPE*('*a*)}
by (*force simp*: *singleton-def ptr-tag-dom-lift-state*)

lemma *singleton-s-valid*:
g p ⟹ *singleton p* (*v::'a::mem-type*),*g* ⊨$_s$ *p*
by (*simp add*: *singleton-def ptr-tag-s-valid*)

lemma *singleton-lift-typ-heap-Some*:
g p ⟹ *lift-typ-heap g* (*singleton p v*) *p* = *Some* (*v::'a::mem-type*)
by (*simp add*: *singleton-def lift*$_\tau$ *lift*$_\tau$*-heap-update ptr-tag-h-t-valid*)

constdefs *sep-map* :: *'a::c-type ptr* ⇒ *'a ptr-guard* ⇒ *'a* ⇒ *heap-assert*
(- ↦$_-$ - [*150*,*150*,*150*] *150*)
p ↦$_g$ *v* ≡ λ*s*. *lift-typ-heap g s p* = *Some v* ∧
dom s = {*ptr-val* (*p::'a ptr*)*..+size-of TYPE*('*a*)}

lemma *sep-map-g*:
(*p* ↦$_g$ *v*) *s* ⟹ *g p*
by (*force simp*: *sep-map-def dest*: *lift-typ-heap-g*)

lemma *sep-map-singleton*:
g p ⟹ ((*p::'a::mem-type ptr*) ↦$_g$ *v*) (*singleton p v*)
by (*simp add*: *sep-map-def singleton-lift-typ-heap-Some singleton-dom*)

lemma *sep-mapD*:
(*p* ↦$_g$ *v*) *s* ⟹ *lift-typ-heap g s p* = *Some v* ∧
dom s = {*ptr-val* (*p::'a::c-type ptr*)*..+size-of TYPE*('*a*)}
by (*simp add*: *sep-map-def*)

lemma *sep-map-lift-typ-heapD*:
(*p* ↦$_g$ *v*) *s* ⟹ *lift-typ-heap g s p* = *Some* (*v::'a::c-type*)
by (*simp add*: *sep-map-def*)

lemma *sep-map-dom*:
(*p* ↦$_g$ (*v::'a::c-type*)) *s* ⟹ *dom s* = {*ptr-val p..+size-of TYPE*('*a*)}
by (*simp add*: *sep-map-def*)

lemma *sep-map-inj*:
⟦ (*p* ↦$_g$ (*v::'a::c-type*)) *s*; (*p* ↦$_h$ *v'*) *s* ⟧ ⟹ *v* = *v'*
by (*clarsimp simp*: *sep-map-def lift-typ-heap-if split*: *split-if-asm*)

constdefs *sep-map-any* :: *'a ::c-type ptr* ⇒ *'a ptr-guard* ⇒ *heap-assert*
(- ↦$_-$ − [*150*,*150*] *150*)
p ↦$_g$ − ≡ λ*s*. ∃ *v*. (*p* ↦$_g$ *v*) *s*

lemma *sep-map-anyI* [*simp*]:
(*p* ↦$_g$ *v*) *s* ⟹ (*p* ↦$_g$ −) *s*
by (*force simp*: *sep-map-any-def*)

lemma *sep-map-anyD*:
(*p* ↦$_g$ −) *s* ⟹ ∃ *v*. (*p* ↦$_g$ *v*) *s*
by (*force simp*: *sep-map-any-def*)

lemma *sep-map-any-singleton*:
g i ⟹ (*i* ↦$_g$ −) (*singleton i* (*v*::'*a*::*mem-type*))
by (*unfold sep-map-any-def*, *rule-tac x*=*v* **in** *exI*, *erule sep-map-singleton*)

constdefs *heap-disj* :: ('*a* ⇀ '*b*) ⇒ ('*a* ⇀ '*b*) ⇒ *bool* (- ⊥ - [*90*,*90*] *980*)
s$_0$ ⊥ *s*$_1$ ≡ *dom s*$_0$ ∩ *dom s*$_1$ = {}

lemma *heap-disj-empty-right* [*simp*]:
s ⊥ *empty*
by (*simp add*: *heap-disj-def*)

lemma *heap-disj-com*:
s$_0$ ⊥ *s*$_1$ = *s*$_1$ ⊥ *s*$_0$
by (*simp add*: *heap-disj-def*, *fast*)

lemma *heap-disj-dom*:
s$_0$ ⊥ *s*$_1$ ⟹ *dom s*$_0$ ∩ *dom s*$_1$ = {}
by (*simp add*: *heap-disj-def*)

lemma *proj-h-heap-merge*:
proj-h (*s* ++ *t*) = (λ*x*. *if x* ∈ *dom t then proj-h t x else proj-h s x*)
by (*force simp*: *proj-h-def intro*: *ext split*: *option.splits*)

lemma *proj-d-heap-merge*:
proj-d (*s* ++ *t*) = *proj-d s* ++ *proj-d t*
by (*force simp*: *proj-d-def map-add-def intro*: *ext split*: *option.splits*)

lemma *s-valid-heap-merge-right*:
s$_1$,*g* ⊨$_s$ *p* ⟹ *s*$_0$ ++ *s*$_1$,*g* ⊨$_s$ *p*
by (*simp add*: *s-valid-def h-t-valid-def valid-footprint-def*
proj-d-heap-merge)

lemma *heap-list-s-heap-merge-right'*:
⟦ *s*$_1$,*g* ⊨$_s$ (*p*::'*a*::*c-type ptr*); *n* ≤ *size-of TYPE*('*a*) ⟧ ⟹
heap-list-s (*s*$_0$ ++ *s*$_1$) *n* (*ptr-val p* + *of-nat* (*size-of TYPE*('*a*) − *n*))
= *heap-list-s s*$_1$ *n* (*ptr-val p* + *of-nat* (*size-of TYPE*('*a*) − *n*))
proof (*induct n*)
case *0* **thus** *?case* **by** (*simp add*: *heap-list-s-def*)
next
case (*Suc n*)

hence *ptr-val p + (of-nat (size-of TYPE('a) − Suc n)) ∈ dom s₁*
by − (*drule-tac x=size-of TYPE('a) − Suc n* **in** *s-valid-Some*, *auto*)
with *Suc* **show** *?case*
by (*simp add*: *heap-list-s-def proj-h-heap-merge compare-rls*)
qed

lemma *heap-list-s-heap-merge-right*:
s₁,g ⊨ₛ p ⟹ heap-list-s (s₀ ++ s₁) (size-of TYPE('a)) (ptr-val p) =
heap-list-s s₁ (size-of TYPE('a)) (ptr-val (p::'a::c-type ptr))
by (*force dest*: *heap-list-s-heap-merge-right'*)

lemma *lift-typ-heap-heap-merge-right*:
lift-typ-heap g s₁ p = Some v ⟹
lift-typ-heap g (s₀ ++ s₁) (p::'a::c-type ptr) = Some v
by (*force simp*: *lift-typ-heap-if s-valid-heap-merge-right*
heap-list-s-heap-merge-right split: *split-if-asm*)

lemma *lift-typ-heap-heap-merge-sep-map*:
(p ↦_g v) s₁ ⟹ lift-typ-heap g (s₀ ++ s₁) p = Some (v::'a::c-type)
by − (*drule sep-map-lift-typ-heapD*, *erule lift-typ-heap-heap-merge-right*)

lemma *heap-merge-com*:
s₀ ⊥ s₁ ⟹ s₀ ++ s₁ = s₁ ++ s₀
by (*drule heap-disj-dom*, *rule map-add-comm*, *force*)

declare *map-add-assoc* [*simp del*]

lemma *heap-merge-ac*:
s₀ ⊥ s₁ ⟹ s₀ ++ (s₁ ++ s₂) = s₁ ++ (s₀ ++ s₂)
by (*simp add*: *heap-merge-com heap-disj-com map-add-assoc*)

lemmas *heap-merge = map-add-assoc* [*symmetric*] *heap-merge-com heap-disj-com heap-merge-ac*

lemma *heap-merge-disj*:
s₀ ⊥ s₁ ++ s₂ = (s₀ ⊥ s₁ ∧ s₀ ⊥ s₂)
by (*simp add*: *heap-disj-def*, *fast*)

constdefs *sep-conj* :: *('a,'b) map-assert ⇒ ('a,'b) map-assert ⇒ ('a,'b) map-assert*

(**infixr** *∧* 90*)
P ∧ Q ≡ λs. ∃ s₀ s₁. s₀ ⊥ s₁ ∧ s = s₁ ++ s₀ ∧ P s₀ ∧ Q s₁*

lemma *sep-conjI*:
⟦ P s₀; Q s₁; s₀ ⊥ s₁; s = s₁ ++ s₀ ⟧ ⟹ (P ∧ Q) s*
by (*force simp*: *sep-conj-def*)

lemma *sep-conjD*:
(P ∧ Q) s ⟹ ∃ s₀ s₁. s₀ ⊥ s₁ ∧ s = s₁ ++ s₀ ∧ P s₀ ∧ Q s₁*

by (*force simp*: *sep-conj-def*)

constdefs *sep-map'* :: *'a*::*c-type ptr* ⇒ *'a ptr-guard* ⇒ *'a* ⇒ *heap-assert*
(- ↪- - [*150*,*150*,*150*] *150*)
p ↪$_g$ *v* ≡ (*p* ↦$_g$ *v*) ∧* *sep-true*

lemma *sep-map'I*:
((*p* ↦$_g$ *v*) ∧* *sep-true*) *s* ⟹ (*p* ↪$_g$ *v*) *s*
by (*simp add*: *sep-map'-def*)

lemma *sep-map'D*:
(*p* ↪$_g$ *v*) *s* ⟹ ((*p* ↦$_g$ *v*) ∧* *sep-true*) *s*
by (*simp add*: *sep-map'-def*)

lemma *sep-map'-g*:
(*p* ↪$_g$ *v*) *s* ⟹ *g p*
by (*force simp add*: *sep-map'-def dest*: *sep-conjD sep-map-g*)

lemma *sep-conj-sep-true*:
P s ⟹ (*P* ∧* *sep-true*) *s*
by (*erule-tac* s_1=*empty* **in** *sep-conjI*, *simp*+)

lemma *sep-map-sep-map'* [*simp*]:
(*p* ↦$_g$ *v*) *s* ⟹ (*p* ↪$_g$ *v*) *s*
by (*unfold sep-map'-def*, *erule sep-conj-sep-true*)

lemma *sep-conj-true* [*simp*]:
sep-true ∧* *sep-true* = *sep-true*
by (*rule ext*, *simp*, *rule-tac* s_0=*x* **and** s_1=*empty* **in** *sep-conjI*, *auto*)

lemma *sep-conj-assocD*:
assumes *l*: ((*P* ∧* *Q*) ∧* *R*) *s*
shows (*P* ∧* (*Q* ∧* *R*)) *s*
proof −
from *l* **obtain** *s'* s_2 **where** *disj-o*: *s'* ⊥ s_2 **and** *merge-o*: *s* = s_2 ++ *s'* **and**
l-o: (*P* ∧* *Q*) *s'* **and** *r-o*: *R* s_2 **by** (*force dest*: *sep-conjD*)
then obtain s_0 s_1 **where** *disj-i*: s_0 ⊥ s_1 **and** *merge-i*: *s'* = s_1 ++ s_0 **and**
l-i: *P* s_0 **and** *r-i*: *Q* s_1 **by** (*force dest*: *sep-conjD*)
from *disj-o disj-i merge-i* **have** *disj-i'*: s_1 ⊥ s_2
by (*force simp*: *heap-merge heap-merge-disj*)
with *r-i* **and** *r-o* **have** *r-o'*: (*Q* ∧* *R*) (s_2 ++ s_1) **by** (*fast intro*: *sep-conjI*)
from *disj-o merge-i disj-i disj-i'* **have** s_0 ⊥ s_2 ++ s_1
by (*force simp*: *heap-merge heap-merge-disj*)
with *r-o' l-i* **have** (*P* ∧* (*Q* ∧* *R*)) ((s_2 ++ s_1) ++ s_0)
by (*force intro*: *sep-conjI*)
moreover from *merge-o merge-i disj-i disj-i'* **have** *s* = ((s_2 ++ s_1) ++ s_0)
by (*simp add*: *map-add-assoc* [*symmetric*])
ultimately show *?thesis* **by** *simp*
qed

lemma *sep-conj-com* [*simp*]:
  $P \wedge^* Q = Q \wedge^* P$
  by (*force simp*: *heap-merge intro*: *ext sep-conjI dest*: *sep-conjD*)

lemma *sep-conj-false-right* [*simp*]:
  $P \wedge^* \mathit{sep\text{-}false} = \mathit{sep\text{-}false}$
  by (*force dest*: *sep-conjD intro*: *ext*)

lemma *sep-conj-false-left* [*simp*]:
  $\mathit{sep\text{-}false} \wedge^* P = \mathit{sep\text{-}false}$
  by *simp*

lemma *sep-conj-comD*:
  $(P \wedge^* Q)\ s \Longrightarrow (Q \wedge^* P)\ s$
  by *simp*

lemma *exists-left*:
  $(Q \wedge^* (\lambda s.\ \exists x.\ P\ x\ s)) = ((\lambda s.\ \exists x.\ P\ x\ s) \wedge^* Q)$ **by** *simp*

lemma *sep-conj-assoc* [*simp*]:
  $(P \wedge^* Q) \wedge^* R = P \wedge^* (Q \wedge^* R)$ (**is** *?x* = *?y*)
proof (*rule ext*, *rule*)
  fix *s*
  assume *?x s*
  thus *?y s* **by** − (*erule sep-conj-assocD*)
next
  fix *s*
  assume *?y s*
  hence $((R \wedge^* Q) \wedge^* P)\ s$ **by** *simp*
  hence $(R \wedge^* (Q \wedge^* P))\ s$ **by** − (*erule sep-conj-assocD*)
  thus *?x s* **by** *simp*
qed

lemma *sep-conj-left-com* [*simp*]:
  $P \wedge^* (Q \wedge^* R) = Q \wedge^* (P \wedge^* R)$ (**is** *?x* = *?y*)
proof −
  have $?x = (Q \wedge^* R) \wedge^* P$ **by** *simp*
  also have $\ldots = Q \wedge^* (R \wedge^* P)$ **by** (*subst sep-conj-assoc*, *simp*)
  finally show *?thesis* **by** *simp*
qed

lemma *sep-conj-empty*:
  $P \wedge^* \square = P$
proof (*rule ext*, *rule*)

fix x
assume $(P \wedge^* \square)\ x$
thus $P\ x$ **by** (*force simp*: *sep-emp-def dest*: *sep-conjD*)
next
fix x
assume $P\ x$
moreover have $\square$ *empty* **by** *simp*
ultimately show $(P \wedge^* \square)\ x$ **by** $-$ (*erule* (*1*) *sep-conjI*, *auto*)
qed

lemma *sep-conj-empty'* [*simp*]:
$\square \wedge^* P = P$
by (*simp add*: *sep-conj-empty*)

lemma *sep-conj-true-P* [*simp*]:
sep-true $\wedge^*$ (*sep-true* $\wedge^*$ P) = (*sep-true* $\wedge^*$ P)
by *simp*

lemma *sep-map'-unfold*:
$(p \hookrightarrow_g v) = ((p \hookrightarrow_g v) \wedge^*$ *sep-true*$)$
by (*simp add*: *sep-map'-def*)

lemma *sep-conj-disj*:
$((\lambda s.\ P\ s \vee Q\ s) \wedge^* R)\ s = ((P \wedge^* R)\ s \vee (Q \wedge^* R)\ s)$ (**is** $?x = (?y \vee ?z)$)
proof *rule*
assume *?x*
then obtain $s_0\ s_1$ **where** $s_0 \perp s_1$ **and** $s = s_1 +\!\!+ s_0$ **and** $P\ s_0 \vee Q\ s_0$ **and**
$R\ s_1$
by $-$ (*drule sep-conjD*, *auto*)
moreover hence $\neg\ ?z \Longrightarrow \neg\ Q\ s_0$
by $-$ (*clarsimp*, *erule notE*, *erule* (*2*) *sep-conjI*, *simp*)
ultimately show *?y* $\vee$ *?z* **by** (*force intro*: *sep-conjI*)
next
have *?y* $\Longrightarrow$ *?x*
by (*force simp*: *heap-merge intro*: *sep-conjI dest*: *sep-conjD*)
moreover have *?z* $\Longrightarrow$ *?x*
by (*force simp*: *heap-merge intro*: *sep-conjI dest*: *sep-conjD*)
moreover assume *?y* $\vee$ *?z*
ultimately show *?x* **by** *fast*
qed

lemma *sep-conj-conj*:
$((\lambda s.\ P\ s \wedge Q\ s) \wedge^* R)\ s \Longrightarrow (P \wedge^* R)\ s \wedge (Q \wedge^* R)\ s$
by (*force intro*: *sep-conjI dest*!: *sep-conjD*)

lemma *sep-conj-exists*:
$((\lambda s.\ \exists x.\ P\ x\ s) \wedge^* Q)\ s = (\exists x.\ (P\ x \wedge^* Q)\ s)$
by (*force intro*: *sep-conjI dest*: *sep-conjD*)

lemma *sep-conj-forall*:
$((\lambda s.\ \forall x.\ P\ x\ s)\ \wedge^*\ Q)\ s \Longrightarrow (P\ x\ \wedge^*\ Q)\ s$
by (*force intro*: *sep-conjI dest*: *sep-conjD*)

lemma *sep-conj-impl*:
$[\![\ (P\ \wedge^*\ Q)\ s;\ \bigwedge s.\ P\ s \Longrightarrow P'\ s;\ \bigwedge s.\ Q\ s \Longrightarrow Q'\ s\ ]\!] \Longrightarrow (P'\ \wedge^*\ Q')\ s$
by (*force intro*: *sep-conjI dest*: *sep-conjD*)

lemma *sep-conj-sep-true-left*:
$(P\ \wedge^*\ Q)\ s \Longrightarrow (sep\text{-}true\ \wedge^*\ Q)\ s$
by (*erule sep-conj-impl*, *simp*+)

lemma *sep-conj-sep-true-right*:
$(P\ \wedge^*\ Q)\ s \Longrightarrow (P\ \wedge^*\ sep\text{-}true)\ s$
by (*subst* (*asm*) *sep-conj-com*, *drule sep-conj-sep-true-left*, *simp*)

lemma *sep-globalise*:
$[\![\ (P\ \wedge^*\ R)\ s;\ (\bigwedge s.\ P\ s \Longrightarrow Q\ s)\ ]\!] \Longrightarrow (Q\ \wedge^*\ R)\ s$
by (*fast elim*: *sep-conj-impl*)

constdefs *sep-impl* :: $('a,'b)$ *map-assert* $\Rightarrow$ $('a,'b)$ *map-assert* $\Rightarrow$
$('a,'b)$ *map-assert*
(**infixr** $\longrightarrow^*$ *85*)
$x \longrightarrow^* y \equiv \lambda s.\ \forall s'.\ s \perp s' \wedge x\ s' \longrightarrow y\ (s +\!\!+ s')$

lemma *sep-implI*:
$\forall s'.\ s \perp s' \wedge x\ s' \longrightarrow y\ (s +\!\!+ s') \Longrightarrow (x \longrightarrow^* y)\ s$
by (*force simp*: *sep-impl-def*)

lemma *sep-implD*:
$(x \longrightarrow^* y)\ s \Longrightarrow \forall s'.\ s \perp s' \wedge x\ s' \longrightarrow y\ (s +\!\!+ s')$
by (*force simp*: *sep-impl-def*)

lemma *sep-impl-sep-true* [*simp*]:
$P \longrightarrow^* sep\text{-}true = sep\text{-}true$
by (*force intro*: *sep-implI ext*)

lemma *sep-impl-sep-false* [*simp*]:
$sep\text{-}false \longrightarrow^* P = sep\text{-}true$
by (*force intro*: *sep-implI ext*)

lemma *sep-impl-sep-true-P*:
$(sep\text{-}true \longrightarrow^* P)\ s \Longrightarrow P\ s$
by (*auto dest*!: *sep-implD*, *drule-tac x*=*empty* **in** *spec*, *simp*)

lemma *sep-impl-sep-true-false* [*simp*]:
$sep\text{-}true \longrightarrow^* sep\text{-}false = sep\text{-}false$
by (*force intro*: *ext dest*: *sep-impl-sep-true-P*)

lemma *sep-conj-sep-impl*:
⟦ $P\ s$; $\bigwedge s.\ (P \wedge^*\ Q)\ s \Longrightarrow R\ s$ ⟧ $\Longrightarrow (Q \longrightarrow^*\ R)\ s$
proof (*rule sep-implI*, *clarsimp*)
fix s'
assume $P\ s$ **and** $s \perp s'$ **and** $Q\ s'$
hence $(P \wedge^*\ Q)\ (s +\!\!+ s')$ **by** (*force simp*: *heap-merge intro*: *sep-conjI*)
moreover assume $\bigwedge s.\ (P \wedge^*\ Q)\ s \Longrightarrow R\ s$
ultimately show $R\ (s +\!\!+ s')$ **by** *simp*
qed

lemma *sep-conj-sep-impl2*:
⟦ $(P \wedge^*\ Q)\ s$; $\bigwedge s.\ P\ s \Longrightarrow (Q \longrightarrow^*\ R)\ s$ ⟧ $\Longrightarrow R\ s$
by (*force simp*: *heap-merge dest*: *sep-implD sep-conjD*)

constdefs *sep-map'-any* :: *'a ::c-type ptr* ⇒ *'a ptr-guard* ⇒ *heap-assert*
(- ↪- - [*150*] *150*)
$p \hookrightarrow_g - \equiv \lambda s.\ \exists x.\ (p \hookrightarrow_g x)\ s$

lemma *sep-map'-anyI* [*simp*]:
$(p \hookrightarrow_g v)\ s \Longrightarrow (p \hookrightarrow_g -)\ s$
by (*force simp*: *sep-map'-any-def*)

lemma *sep-map'-anyD*:
$(p \hookrightarrow_g -)\ s \Longrightarrow \exists v.\ (p \hookrightarrow_g v)\ s$
by (*force simp*: *sep-map'-any-def*)

lemma *sep-map'-any-unfold*:
$(i \hookrightarrow_g -) = ((i \hookrightarrow_g -) \wedge^*\ sep\text{-}true)$
by (*rule ext*, *simp add*: *sep-map'-any-def*)
(*subst sep-map'-unfold*, *subst sep-conj-com*, *subst sep-conj-exists*, *simp*)

lemma *sep-map-sep-map'-any* [*simp*]:
$(p \mapsto_g v)\ s \Longrightarrow (p \hookrightarrow_g -)\ s$
by (*rule-tac v=v* **in** *sep-map'-anyI*, *simp*)

lemma *map-add-right-dom-eq*:
⟦ $x +\!\!+ y = x' +\!\!+ y'$; $dom\ y = dom\ y'$ ⟧ $\Longrightarrow y = y'$
by (*unfold map-add-def*, *rule ext*, *rule ccontr*,
drule-tac x=xa **in** *fun-cong*, *clarsimp split*: *option.splits*,
drule sym, *drule sym*, *force+*)

lemma *sep-map'-inj*:
assumes *pv*: $(p \hookrightarrow_g (v::'a::c\text{-}type))\ s$ **and** *pv'*: $(p \hookrightarrow_h v')\ s$
shows $v = v'$
proof −
from *pv pv'* **obtain** $s_0\ s_1\ s_0'\ s_1'$ **where** *pv-m*: $(p \mapsto_g v)\ s_1$ **and**
pv'-m: $(p \mapsto_h v')\ s_1'$ **and** $s_0 +\!\!+ s_1 = s_0' +\!\!+ s_1'$
by (*force simp*: *sep-map'-def heap-merge dest*!: *sep-conjD*)

hence $s_1 = s_1'$ **by** (*force dest*!: *map-add-right-dom-eq sep-map-dom*)
with *pv-m pv'-m* **show** *?thesis* **by** (*force dest*: *sep-map-inj*)
qed

lemma *sep-map'-any-dom*:
$((p::'a::mem\text{-}type\ ptr) \hookrightarrow_g -)\ s \Longrightarrow ptr\text{-}val\ p \in dom\ s$
by (*clarsimp simp*: *sep-map'-def sep-map'-any-def dest*!: *sep-conjD*)
(*subgoal-tac* s_1 $(ptr\text{-}val\ p) \neq None$, *force simp*: *heap-merge*,
force dest: *sep-map-dom*)

lemma *sep-map'-dom*:
$(p \hookrightarrow_g (v::'a::mem\text{-}type))\ s \Longrightarrow ptr\text{-}val\ p \in dom\ s$
by (*drule sep-map'-anyI*, *erule sep-map'-any-dom*)

lemma *sep-map'-lift-typ-heapD*:
$(p \hookrightarrow_g v)\ s \Longrightarrow lift\text{-}typ\text{-}heap\ g\ s\ p = Some\ (v::'a::c\text{-}type)$
by (*force simp*: *sep-map'-def heap-merge dest*: *sep-conjD*
lift-typ-heap-heap-merge-sep-map)

lemma *sep-map'-merge*:
assumes *map'-v*: $(p \hookrightarrow_g v)\ s_0 \vee (p \hookrightarrow_g v)\ s_1$ **and** *disj*: $s_0 \perp s_1$
shows $(p \hookrightarrow_g v)\ (s_0 +\!\!+ s_1)$ (**is** *?x*)
proof *cases*
assume $(p \hookrightarrow_g v)\ s_0$
with *disj* **show** *?x*
by (*clarsimp simp*: *sep-map'-def dest*!: *sep-conjD*)
(*rule-tac* $s_1{=}s_1'$ **and** $s_0{=}s_0 +\!\!+ s_1$ **in** *sep-conjI*,
auto simp: *heap-merge-disj heap-merge*)
next
assume $\neg\ (p \hookrightarrow_g v)\ s_0$
with *map'-v* **have** $(p \hookrightarrow_g v)\ s_1$ **by** *simp*
with *disj* **show** *?x*
by (*clarsimp simp*: *sep-map'-def dest*!: *sep-conjD*)
(*rule-tac* $s_1{=}s_1$ **and** $s_0{=}s_0 +\!\!+ s_0'$ **in** *sep-conjI*,
auto simp: *heap-merge-disj heap-merge*)
qed

lemma *sep-conj-overlapD*:
$[\![\ (P \wedge^*\ Q)\ s;\ \bigwedge s.\ P\ s \Longrightarrow ((p::'a::mem\text{-}type\ ptr) \hookrightarrow_g -)\ s;$
$\bigwedge s.\ Q\ s \Longrightarrow (p \hookrightarrow_h -)\ s\ ]\!] \Longrightarrow False$
by (*drule sep-conjD*, *clarsimp simp*: *heap-disj-def*)
(*subgoal-tac ptr-val* $p \in dom\ s_0 \wedge ptr\text{-}val\ p \in dom\ s_1$,
auto intro!: *sep-map'-any-dom*)

lemma *sep-no-skew*:
$[\![\ (p \hookrightarrow_g v)\ s;\ (q \hookrightarrow_h w)\ s\ ]\!] \Longrightarrow$
$p{=}q \vee \{ptr\text{-}val\ (p::'a::c\text{-}type\ ptr)..+size\text{-}of\ TYPE('a)\} \cap$
$\{ptr\text{-}val\ q..+size\text{-}of\ TYPE('a)\} = \{\}$
by (*force simp*: *lift-typ-heap-if s-valid-def*

dest: *h-t-valid-neq-disjoint sep-map′-lift-typ-heapD*
split: *split-if-asm*)

lemma *sep-no-skew2*:
⟦ (*p* ↪$_g$ *v*) *s*; (*q* ↪$_h$ *w*) *s*; *typ-tag-t TYPE*('*a*) ≠ *typ-tag-t TYPE*('*b*) ⟧
⟹ {*ptr-val* (*p*::'*a*::*c-type ptr*)..+*size-of TYPE*('*a*)} ∩
{*ptr-val* (*q*::'*b*::*c-type ptr*)..+*size-of TYPE*('*b*)} = {}
by (*drule sep-map′-lift-typ-heapD*)+
(*clarsimp simp*: *lift-typ-heap-if s-valid-def split*: *split-if-asm*,
frule (*1*) *h-t-valid-neq-disjoint*,
auto simp: *h-t-valid-def valid-footprint-def*)

lemma *sep-conj-impl-same*:
(*P* ∧* (*P* −→* *Q*)) *s* ⟹ *Q s*
by (*drule sep-conjD*, *auto simp*: *heap-disj-com dest*: *sep-implD*)

constdefs *pure* :: ('*a*,'*b*) *map-assert* ⇒ *bool*
pure P ≡ ∀ *s s′*. *P s* = *P s′*

lemma *pure-sep-true*:
pure sep-true
by (*simp add*: *pure-def*)

lemma *pure-sep-false*:
pure sep-false
by (*simp add*: *pure-def*)

lemma *pure-split*:
pure P = (*P* = *sep-true* ∨ *P* = *sep-false*)
by (*force simp*: *pure-def intro*!: *ext*)

lemma *pure-sep-conj*:
⟦ *pure P*; *pure Q* ⟧ ⟹ *pure* (*P* ∧* *Q*)
by (*force simp*: *pure-split*)

lemma *pure-sep-impl*:
⟦ *pure P*; *pure Q* ⟧ ⟹ *pure* (*P* −→* *Q*)
by (*force simp*: *pure-split*)

lemma *pure-for-all*:
(⋀*x*. *pure* (*P x*)) ⟹ *pure* (λ*s*. ∀ *x*. *P x s*)
by (*auto simp*: *pure-def*)

lemma *pure-exists*:
(⋀*x*. *pure* (*P x*)) ⟹ *pure* (λ*s*. ∃ *x*. *P x s*)
by (*auto simp*: *pure-def*)

lemma *pure-conj-sep-conj*:
⟦ (λs. P s ∧ Q s) s; pure P ∨ pure Q ⟧ ⟹ (P ∧* Q) s
by (*force simp*: *pure-split intro*: *sep-conj-sep-true*)

lemma *pure-sep-conj-conj*:
⟦ (P ∧* Q) s; pure P; pure Q ⟧ ⟹ (λs. P s ∧ Q s) s
by (*force simp*: *pure-split*)

lemma *pure-conj-sep-conj-assoc*:
pure P ⟹ (λs. P s ∧ Q s) ∧* R = (λs. P s ∧ (Q ∧* R) s)
by (*force simp*: *pure-split*)

lemma *pure-sep-impl-impl*:
⟦ (P −→* Q) s; pure P ⟧ ⟹ P s −→ Q s
by (*force simp*: *pure-split dest*: *sep-impl-sep-true-P*)

lemma *pure-impl-sep-impl*:
⟦ P s −→ Q s; pure P; pure Q ⟧ ⟹ (P −→* Q) s
by (*force simp*: *pure-split*)

constdefs *intuitionistic* :: ('a,'b) *map-assert* ⇒ *bool*
intuitionistic P ≡ ∀ s s'. P s ∧ s ⊆m s' −→ P s'

lemma *intuitionisticI*:
(⋀s s'. ⟦ P s; s ⊆m s' ⟧ ⟹ P s') ⟹ *intuitionistic P*
by (*unfold intuitionistic-def*, *fast*)

lemma *intuitionisticD*:
⟦ *intuitionistic P*; P s; s ⊆m s' ⟧ ⟹ P s'
by (*unfold intuitionistic-def*, *fast*)

lemma *pure-intuitionistic*:
pure P ⟹ *intuitionistic P*
by (*clarsimp simp*: *intuitionistic-def pure-def*, *fast*)

lemma *None-com*:
(*None* = x) = (x = *None*)
by *fast*

lemma *Some-com*:
(*Some y* = x) = (x = *Some y*)
by *fast*

lemma *map-le-restrict*:
s ⊆m s' ⟹ s = s' |` *dom s*
by (*force simp*: *map-le-def restrict-map-def None-com intro*: *ext*)

lemma *map-add-restrict-comp-right* [*simp*]:
(*s* |‘ *P* ++ *s* |‘ (*UNIV* − *P*)) = *s*
by (*force simp*: *map-add-def restrict-map-def split*: *option.splits intro*: *ext*)

lemma *map-add-restrict-comp-left* [*simp*]:
(*s* |‘ (*UNIV* − *P*) ++ *s* |‘ *P*) = *s*
by (*subst map-add-comm*, *auto*)

lemma *heap-disj-comp* [*simp*]:
s ⊥ *s′* |‘ (*UNIV* − *dom s*)
by (*force simp*: *heap-disj-def*)

lemma *map-le-dom-restrict-add*:
s′ ⊆$_m$ *s* ⟹ *s* |‘ (*dom s* − *dom s′*) ++ *s′* = *s*
by (*auto simp*: *None-com map-add-def restrict-map-def map-le-def*
split: *option.splits*
intro!: *ext*)
(*force simp*: *Some-com*)+

lemma *intuitionistic-sep-conj-sep-true*:
intuitionistic (*P* ∧* *sep-true*)
by (*rule intuitionisticI*, *drule sep-conjD*, *clarsimp*)
(*erule-tac s*$_1$*=s′*|‘(*dom s′* − *dom s*$_0$) **in** *sep-conjI*, *simp*,
force simp: *heap-disj-def*,
force intro: *map-le-dom-restrict-add map-add-le-mapE sym*)

lemma *intuitionistic-sep-map′*:
intuitionistic (*p* ↪$_g$ *v*)
by (*unfold sep-map′-def*)
(*rule intuitionistic-sep-conj-sep-true*)

lemma *heap-disj-map-le*:
⟦ *s* ⊆$_m$ *s′*; *s′* ⊥ *s′a* ⟧ ⟹ *s* ⊥ *s′a*
by (*force simp*: *heap-disj-def map-le-def*)

lemma *intuitionistic-sep-impl-sep-true*:
intuitionistic (*sep-true* ⟶* *P*)
proof (*rule intuitionisticI*, *rule sep-implI*, *clarsimp*)
fix *s s′ s′a*
assume (*sep-true* ⟶* *P*) *s* **and** *le*: *s* ⊆$_m$ *s′* **and** *s′* ⊥ *s′a*
moreover hence *P* (*s* ++ (*s′* |‘ (*dom s′* − *dom s*) ++ *s′a*))
by − (*drule sep-implD*,
drule-tac x=s′|‘(*dom s′* − *dom s*) ++ *s′a* **in** *spec*,
force simp: *heap-disj-def dest*: *heap-disj-map-le*)
moreover have *dom s* ∩ *dom* (*s′* |‘ (*dom s′* − *dom s*)) = {} **by** *force*
ultimately show *P* (*s′* ++ *s′a*)
by (*force simp*: *map-le-dom-restrict-add map-add-assoc dest*!: *map-add-comm*)
qed

lemma *intuitionistic-conj*:
⟦ *intuitionistic P*; *intuitionistic Q* ⟧ ⟹ *intuitionistic* (λ*s*. *P s* ∧ *Q s*)
by (*force intro*: *intuitionisticI dest*: *intuitionisticD*)

lemma *intuitionistic-disj*:
⟦ *intuitionistic P*; *intuitionistic Q* ⟧ ⟹ *intuitionistic* (λ*s*. *P s* ∨ *Q s*)
by (*force intro*: *intuitionisticI dest*: *intuitionisticD*)

lemma *intuitionistic-forall*:
(⋀*x*. *intuitionistic* (*P x*)) ⟹ *intuitionistic* (λ*s*. ∀ *x*. *P x s*)
by (*force intro*: *intuitionisticI dest*: *intuitionisticD*)

lemma *intuitionistic-exists*:
(⋀*x*. *intuitionistic* (*P x*)) ⟹ *intuitionistic* (λ*s*. ∃ *x*. *P x s*)
by (*force intro*: *intuitionisticI dest*: *intuitionisticD*)

lemma *map-le-dom-subset-restrict*:
⟦ *s'* ⊆$_m$ *s*; *dom s'* ⊆ *P* ⟧ ⟹ *s'* ⊆$_m$ (*s* |` *P*)
by (*force simp*: *restrict-map-def map-le-def*)

lemma *intuitionistic-sep-conj*:
intuitionistic (*P*::('*a*,'*b*) *map-assert*) ⟹ *intuitionistic* (*P* ∧* *Q*)
proof (*rule intuitionisticI*, *drule sep-conjD*, *clarsimp*)
fix *s' s*$_0$ *s*$_1$
assume *le*: *s*$_1$ ++ *s*$_0$ ⊆$_m$ *s'* **and** *disj*: *s*$_0$ ⊥ (*s*$_1$::'*a* ⇀ '*b*)
hence *le-restrict*: *s*$_0$ ⊆$_m$ *s'* |` (*dom s'* − *dom s*$_1$)
by − (*rule map-le-dom-subset-restrict*, *erule map-add-le-mapE*,
force simp: *heap-disj-def dest*: *map-le-implies-dom-le*
map-add-le-mapE)
moreover assume *intuitionistic P* **and** *P s*$_0$
ultimately have *P* (*s'* |` (*dom s'* − *dom s*$_1$))
by − (*erule* (*2*) *intuitionisticD*)
moreover from *le-restrict* **have** *s'* |` (*dom s'* − *dom s*$_1$) ⊥ *s*$_1$
by (*force simp*: *heap-disj-def dest*: *map-le-implies-dom-le*)
moreover from *le disj* **have** *s*$_1$ ⊆$_m$ *s'*
by (*subst* (*asm*) *map-add-comm*, *force simp*: *heap-disj-def*)
(*erule map-add-le-mapE*)
hence *s'* = *s*$_1$ ++ *s'* |` (*dom s'* − *dom s*$_1$)
by (*subst map-add-comm*, *force simp*: *heap-disj-def*,
force simp: *map-le-dom-restrict-add map-add-comm heap-disj-def*)
moreover assume *Q s*$_1$
ultimately show (*P* ∧* *Q*) *s'*
by − (*erule* (*3*) *sep-conjI*)
qed

lemma *intuitionistic-sep-impl*:
intuitionistic (*Q*::('*a*,'*b*) *map-assert*) ⟹ *intuitionistic* (*P* −→* *Q*)
proof (*rule intuitionisticI*, *rule sep-implI*, *clarsimp*)
fix *s s' s'a*

assume *le*: $(s::'a \rightharpoonup 'b) \subseteq_m s'$ **and** *disj*: $s' \perp s'a$
moreover hence $s +\!\!+ s'a \subseteq_m s' +\!\!+ s'a$
proof −
from *le disj* **have** $s \subseteq_m s +\!\!+ s'a$
by (*subst heap-merge-com*)
(*force simp*: *heap-disj-def dest*: *map-le-implies-dom-le*, *simp*)
with *le* **show** *?thesis*
by − (*rule map-add-le-mapI*, *subst heap-merge-com*,
auto elim: *map-le-trans*)
qed
moreover assume $(P \longrightarrow^* Q)\ s$ **and** *intuitionistic* Q **and** $P\ s'a$
ultimately show $Q\ (s' +\!\!+ s'a)$
by (*fast elim*: *heap-disj-map-le intuitionisticD dest*: *sep-implD*)
qed

lemma *strongest-intuitionistic*:
$\neg\ (\exists\, Q.\ (\forall\, s.\ (Q\ s \longrightarrow (P \wedge^*\ sep\text{-}true)\ s)) \wedge\ intuitionistic\ Q \wedge$
$Q \neq (P \wedge^*\ sep\text{-}true) \wedge (\forall\, s.\ P\ s \longrightarrow Q\ s))$
by (*force intro*!: *ext dest*!: *sep-conjD intuitionisticD*)

lemma *weakest-intuitionistic*:
$\neg\ (\exists\, Q.\ (\forall\, s.\ ((sep\text{-}true \longrightarrow^*\ P)\ s \longrightarrow Q\ s)) \wedge\ intuitionistic\ Q \wedge$
$Q \neq (sep\text{-}true \longrightarrow^*\ P) \wedge (\forall\, s.\ Q\ s \longrightarrow P\ s))$
by (*auto intro*!: *ext sep-implI*)
(*drule-tac* $s{=}x$ **and** $s'{=}x +\!\!+ s'$ **in** *intuitionisticD*, *auto*,
subst heap-merge-com, *simp*+)

lemma *intuitionistic-sep-conj-sep-true-P*:
$[\![\ (P \wedge^*\ sep\text{-}true)\ s;\ intuitionistic\ P\ ]\!] \Longrightarrow P\ s$
by (*force dest*: *intuitionisticD sep-conjD*)

lemma *intuitionistic-sep-conj-sep-true-simp*:
$intuitionistic\ P \Longrightarrow P \wedge^*\ sep\text{-}true = P$
by (*fast intro*: *sep-conj-sep-true ext elim*: *intuitionistic-sep-conj-sep-true-P*)

lemma *intuitionistic-sep-impl-sep-true-P*:
$[\![\ P\ s;\ intuitionistic\ P\ ]\!] \Longrightarrow (sep\text{-}true \longrightarrow^*\ P)\ s$
by (*force simp*: *heap-merge-com intro*: *sep-implI dest*: *intuitionisticD*)

lemma *intuitionistic-sep-impl-sep-true-simp*:
$intuitionistic\ P \Longrightarrow (sep\text{-}true \longrightarrow^*\ P) = P$
by (*fast intro*: *ext*
elim: *sep-impl-sep-true-P intuitionistic-sep-impl-sep-true-P*)

constdefs *dom-exact* :: $('a,'b)\ map\text{-}assert \Rightarrow bool$
$dom\text{-}exact\ P \equiv \forall\, s\ s'.\ P\ s \wedge P\ s' \longrightarrow dom\ s = dom\ s'$

lemma *dom-exactI*:
(⋀*s s'*. ⟦ *P s*; *P s'* ⟧ ⟹ *dom s* = *dom s'*) ⟹ *dom-exact P*
by (*unfold dom-exact-def*, *fast*)

lemma *dom-exactD*:
⟦ *dom-exact P*; *P* s_0; *P* s_1 ⟧ ⟹ *dom* s_0 = *dom* s_1
by (*unfold dom-exact-def*, *fast*)

lemma *dom-exact-sep-conj*:
⟦ *dom-exact P*; *dom-exact Q* ⟧ ⟹ *dom-exact* (*P* ∧* *Q*)
by (*rule dom-exactI*, (*drule sep-conjD*)+, *clarsimp*)
(*drule* (*2*) *dom-exactD*, *drule* (*2*) *dom-exactD*, *simp*)

lemma *dom-exact-forall*:
dom-exact (*P x*) ⟹ *dom-exact* (λ*s*. ∀ *x*. *P x s*)
by (*rule dom-exactI*, (*drule-tac x*=*x* **in** *spec*)+)
(*force dest*: *dom-exactD*)

lemma *heap-merge-dom-exact*:
assumes *merge*: *a* ++ *b* = *c* ++ *d* **and** *d*: *dom a* = *dom c* **and**
ab-disj: *a* ⊥ *b* **and** *cd-disj*: *c* ⊥ *d*
shows *b* = *d*
proof (*rule ext*)
fix *x*
from *merge* **have** *merge-x*: (*a* ++ *b*) *x* = (*c* ++ *d*) *x* **by** *simp*
with *d ab-disj cd-disj* **show** *b x* = *d x*
by − (*case-tac b x*, *case-tac d x*, *simp*, *fastsimp simp*: *heap-disj-def*,
case-tac d x, *clarsimp*, *simp add*: *Some-com*,
force simp: *heap-disj-def*, *simp*)
qed

lemma *dom-exact-sep-conj-conj*:
⟦ (*P* ∧* *R*) *s*; (*Q* ∧* *R*) *s*; *dom-exact R* ⟧ ⟹ ((λ*s*. *P s* ∧ *Q s*) ∧* *R*) *s*
by ((*drule sep-conjD*)+, *clarsimp*, *rule sep-conjI*, *fast*, *rule conjI*)
(*fast*, *drule* (*2*) *dom-exactD*, *drule* (*1*) *heap-merge-dom-exact*,
auto simp: *heap-merge*)

lemma *sep-conj-conj-simp*:
dom-exact R ⟹ ((λ*s*. *P s* ∧ *Q s*) ∧* *R*) = (λ*s*. (*P* ∧* *R*) *s* ∧ (*Q* ∧* *R*) *s*)
by (*fast intro*!: *sep-conj-conj dom-exact-sep-conj-conj ext*)

constdefs *dom-eps* :: ('*a*,'*b*) *map-assert* ⇒ '*a set* (ι -)
ι *P* ≡ *THE x*. ∀ *s*. *P s* ⟶ *x* = *dom s*

syntax
restrict-map :: ('*a* ⇀ '*b*) ⇒ '*a set* ⇒ ('*a* ⇀ '*b*) (-|- [*90*,*91*] *90*)

lemma *map-add-restrict*:
(*s*$_0$ ++ *s*$_1$)|$_P$ = ((*s*$_0$|$_P$) ++ (*s*$_1$|$_P$))

by (*force simp*: *map-add-def restrict-map-def intro*: *ext*)

lemma *dom-epsI*:
⟦ *dom-exact P*; *P s*; *x* ∈ *dom s* ⟧ ⟹ *x* ∈ ι *P*
by (*unfold dom-eps-def*, *rule-tac a*=*dom s* **in** *theI2*)
(*fastsimp simp*: *dom-exact-def*, *clarsimp*+)

lemma *dom-epsD* [*rule-format*]:
⟦ *dom-exact P*; *P s* ⟧ ⟹ *x* ∈ ι *P* ⟶ *x* ∈ *dom s*
by (*unfold dom-eps-def*, *rule-tac a*=*dom s* **in** *theI2*)
(*fastsimp simp*: *dom-exact-def*, *clarsimp*+)

lemma *dom-eps*:
⟦ *dom-exact P*; *P s* ⟧ ⟹ *dom s* = ι *P*
by (*force intro*: *dom-epsI dest*: *dom-epsD*)

lemma *map-restrict-dom-exact*:
⟦ *dom-exact P*; *P s* ⟧ ⟹ $s|_{\iota\ P} = s$
by (*force simp*: *restrict-map-def None-com intro*: *dom-epsI ext*)

lemma *map-restrict-dom-exact2*:
⟦ *dom-exact P*; *P* s_0; s_0 ⊥ s_1 ⟧ ⟹ $(s_1|_{\iota\ P}) = empty$
by (*force simp*: *restrict-map-def heap-disj-def intro*: *ext dest*: *dom-epsD*)

lemma *map-restrict-dom-exact3*:
⟦ *dom-exact P*; *P s* ⟧ ⟹ $s|_{(UNIV\ -\ \iota\ P)} = empty$
by (*force simp*: *restrict-map-def intro*: *ext dest*: *dom-epsI*)

lemma [*simp*]:
option-case None Some P = *P*
by (*simp split*: *option.splits*)

lemma *restrict-map-dom*:
dom s ⊆ *P* ⟹ $s|_P = s$
by (*fastsimp simp*: *restrict-map-def None-com intro*: *ext*)

lemma *map-add-restrict-dom-exact*:
⟦ *dom-exact P*; s_0 ⊥ s_1; *P* s_1 ⟧ ⟹ $(s_1$ ++ $s_0)$ |‘ (ι *P*) = s_1
by (*simp add*: *map-add-restrict map-restrict-dom-exact*)
(*subst map-restrict-dom-exact2*, *auto simp*: *heap-disj-def*)

lemma *map-add-restrict-dom-exact2*:
⟦ *dom-exact P*; s_0 ⊥ s_1; *P* s_0 ⟧ ⟹ $(s_1$ ++ $s_0)$ |‘ $(UNIV$ − ι *P*) = s_1
by (*force simp*: *map-add-restrict heap-disj-def dom-eps*
intro: *restrict-map-dom dest*: *map-restrict-dom-exact3*)

lemma *dom-exact-sep-conj-forall*:
assumes *sc*: ∀ *x*. (*P x* ∧* *Q*) *s* **and** *de*: *dom-exact Q*
shows ((λ*s*. ∀ *x*. *P x s*) ∧* *Q*) *s*

proof (*rule-tac* $s_0{=}s \mid`` (UNIV - \iota\ Q)$ **and** $s_1{=}s \mid`` \iota\ Q$ **in** *sep-conjI*)
from *sc de* **show** $\forall x.\ P\ x\ (s \mid`` (UNIV - \iota\ Q))$
by (*force simp*: *map-add-restrict-dom-exact2 dest*: *sep-conjD*)
next
from *sc de* **show** $Q\ (s \mid`` \iota\ Q)$
by (*force simp*: *map-add-restrict-dom-exact dest*!: *sep-conjD spec*)
next
from *sc de* **show** $s \mid`` (UNIV - \iota\ Q) \perp s \mid`` \iota\ Q$
by (*force simp*: *map-add-restrict-dom-exact2 heap-merge*
dest: *map-add-restrict-dom-exact dest*!: *sep-conjD spec*)
next
show $s = s \mid`` \iota\ Q +\!\!+ s \mid`` (UNIV - \iota\ Q)$ **by** *simp*
qed

lemma *sep-conj-forall-simp*:
$dom\text{-}exact\ Q \Longrightarrow ((\lambda s.\ \forall x.\ P\ x\ s) \wedge^*\ Q) = (\lambda s.\ \forall x.\ (P\ x \wedge^*\ Q)\ s)$
by (*fast dest*: *sep-conj-forall dom-exact-sep-conj-forall intro*: *ext*)

lemma *dom-exact-sep-map*:
$dom\text{-}exact\ (i \mapsto_g x)$
by (*clarsimp simp*: *dom-exact-def sep-map-def*)

constdefs *strictly-exact* :: $('a,'b)\ map\text{-}assert \Rightarrow bool$
$strictly\text{-}exact\ P \equiv \forall s\ s'.\ P\ s \wedge P\ s' \longrightarrow s = s'$

lemma *strictly-exactD*:
$[\![\ strictly\text{-}exact\ P;\ P\ s_0;\ P\ s_1\ ]\!] \Longrightarrow s_0 = s_1$
by (*unfold strictly-exact-def*, *fast*)

lemma *strictly-exactI*:
$(\bigwedge s\ s'.\ [\![\ P\ s;\ P\ s'\ ]\!] \Longrightarrow s = s') \Longrightarrow strictly\text{-}exact\ P$
by (*unfold strictly-exact-def*, *fast*)

lemma *strictly-exact-dom-exact*:
$strictly\text{-}exact\ P \Longrightarrow dom\text{-}exact\ P$
by (*force simp*: *strictly-exact-def dom-exact-def*)

lemma *strictly-exact-sep-conj*:
$[\![\ strictly\text{-}exact\ P;\ strictly\text{-}exact\ Q\ ]\!] \Longrightarrow strictly\text{-}exact\ (P \wedge^*\ Q)$
by (*force intro*!: *strictly-exactI dest*: *sep-conjD strictly-exactD*)

lemma *strictly-exact-conj-impl*:
$[\![\ (Q \wedge^*\ sep\text{-}true)\ s;\ P\ s;\ strictly\text{-}exact\ Q\ ]\!] \Longrightarrow (Q \wedge^*\ (Q \longrightarrow^*\ P))\ s$
by (*force intro*: *sep-conjI sep-implI dest*: *strictly-exactD dest*!: *sep-conjD*)

lemma *pure-conj-right*: $Q \wedge^* (\lambda s.\ P' \wedge Q'\ s) = (\lambda s.\ P' \wedge (Q \wedge^* Q')\ s)$
by (*rule ext*, *rule*, *rule*, *clarsimp dest*!: *sep-conjD*)
(*erule sep-conj-impl*, *auto*)

lemma *pure-conj-right'*: $Q \wedge^* (\lambda s.\ P'\ s \wedge Q') = (\lambda s.\ Q' \wedge (Q \wedge^* P')\ s)$
by (*simp add*: *conj-comms pure-conj-right*)

lemma *pure-conj-left*: $(\lambda s.\ P' \wedge Q'\ s) \wedge^* Q = (\lambda s.\ P' \wedge (Q' \wedge^* Q)\ s)$
by (*simp add*: *pure-conj-right*)

lemma *pure-conj-left'*: $(\lambda s.\ P'\ s \wedge Q') \wedge^* Q = (\lambda s.\ Q' \wedge (P' \wedge^* Q)\ s)$
by (*subst conj-comms*, *subst pure-conj-left*, *simp*)

lemmas *pure-conj* = *pure-conj-right pure-conj-right' pure-conj-left*
pure-conj-left'

declare *pure-conj* [*simp add*]

lemma *sep-conj-sep-conj-sep-impl-sep-conj*:
$(P \wedge^* R)\ s \Longrightarrow (P \wedge^* (Q \longrightarrow^* (Q \wedge^* R)))\ s$
by (*erule* (*1*) *sep-conj-impl*, *erule sep-conj-sep-impl*, *simp*)

lemma *sep-map'-conjE1*:
$[\![\ (P \wedge^* Q)\ s;\ \bigwedge s.\ P\ s \Longrightarrow (i \hookrightarrow_g v)\ s\ ]\!] \Longrightarrow (i \hookrightarrow_g v)\ s$
by (*subst sep-map'-unfold*, *erule sep-conj-impl*, *simp+*)

lemma *sep-map'-conjE2*:
$[\![\ (P \wedge^* Q)\ s;\ \bigwedge s.\ Q\ s \Longrightarrow (i \hookrightarrow_g v)\ s\ ]\!] \Longrightarrow (i \hookrightarrow_g v)\ s$
by (*subst* (*asm*) *sep-conj-com*, *erule sep-map'-conjE1*, *simp*)

lemma *sep-map'-any-conjE1*:
$[\![\ (P \wedge^* Q)\ s;\ \bigwedge s.\ P\ s \Longrightarrow (i \hookrightarrow_g -)\ s\ ]\!] \Longrightarrow (i \hookrightarrow_g -)\ s$
by (*subst sep-map'-any-unfold*, *erule sep-conj-impl*, *simp+*)

lemma *sep-map'-any-conjE2*:
$[\![\ (P \wedge^* Q)\ s;\ \bigwedge s.\ Q\ s \Longrightarrow (i \hookrightarrow_g -)\ s\ ]\!] \Longrightarrow (i \hookrightarrow_g -)\ s$
by (*subst* (*asm*) *sep-conj-com*, *erule sep-map'-any-conjE1*, *simp*)

lemma *sep-conj-mapD*:
$((i \mapsto_g v) \wedge^* P)\ s \Longrightarrow (i \hookrightarrow_g v)\ s \wedge ((i \mapsto_g -) \wedge^* P)\ s$
by (*force intro*: *sep-conj-impl sep-map'-conjE2*)

syntax
-sep-assert :: *bool* ⇒ *heap-state* ⇒ *bool* $((\text{-})^{sep}\ [1000]\ 100)$

end

Bibliography

[1] Programming languages — C. Technical Report 9899:TC2, ISO/IEC JTC1/SC22/WG14, May 2005.

[2] A. W. Appel. Tactics for separation logic, Jan 2006. http://www.cs.princeton.edu/~appel/papers/septacs.pdf.

[3] A. W. Appel and S. Blazy. Separation logic for small-step Cminor. In K. Schneider and J. Brandt, editors, *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2007)*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. Springer, Sep 2007.

[4] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In M. B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer, May 2001.

[5] C. Ballarin. Locales and locale expressions in Isabelle/Isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *Revised Selected Papers of the International Workshop on Types for Proofs and Programs (TYPES 2003)*, volume 3085 of *Lecture Notes in Computer Science*. Springer, Apr 2003.

[6] J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In K. Lodaya and M. Mahajan, editors, *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004)*, volume 3328 of *Lecture Notes in Computer Science*, pages 97–109. Springer, Dec 2004.

[7] S. Berghofer and M. Strecker. Extracting a formally verified, fully executable compiler from a proof assistant. In *Proceedings of the 2nd International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2003)*, volume 82 of *Electronic Notes in Theoretical Computer Science*, pages 33–50. Elsevier, Apr 2003.

[8] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles (SOSP 95)*, volume 29 of *Operating System Review*, pages 267–284. ACM, Dec 1995.

[9] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[10] A. Bijlsma. Calculating with pointers. *Science of Computer Programming*, 12(3):191–205, 1989.

[11] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Proceedings of the 14th International Symposium on Formal Methods (FM 2006)*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, Aug 2006.

[12] J. Bloch. Nearly all binary searches and mergesorts are broken. http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html, Jun 2006.

[13] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[14] R. Bornat. Proving pointer programs in Hoare Logic. In R. C. Backhouse and J. N. Oliveira, editors, *Proceedings of the 5th International Conference on Mathematics of Program Construction (MPC 2000)*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer, Jul 2000.

[15] B. C. Brock, W. A. Hunt, Jr., and M. Kaufmann. The FM9001 microprocessor proof. Technical Report 86, Computational Logic, Inc., 1994.

[16] R. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, 1972.

[17] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In K. Yi, editor, *Proceedings of the 13th International Symposium on Static Analysis (SAS 2006)*, volume 4134 of *Lecture Notes in Computer Science*, pages 182–203. Springer, Aug 2006.

[18] C. Calcagno, H. Yang, and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In R. Hariharan, M. Mukund, and V. Vinay, editors, *Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2001)*, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119. Springer, Dec 2001.

[19] R. Cartwright and D. C. Oppen. The logic of aliasing. Technical Report STAN-CS-79-740, Stanford University, Sep 1979.

[20] T. Cattel. Modelization and verification of a multiprocessor realtime OS kernel. In D. Hogrefe and S. Leue, editors, *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques (FORTE 94)*, volume 6 of *IFIP Conference Proceedings*, pages 55–70. Chapman & Hall, 1994.

[21] A. Cohn. A proof of correctness of the Viper microprocessor: the first level. Technical Report 104, University of Cambridge Computer Laboratory, Jan 1987.

[22] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In R. D. Nicola, editor, *Proceedings of the 16th European Symposium on Programming on Programming Languages and Systems (ESOP 2007)*, volume 4421 of *Lecture Notes in Computer Science*, pages 520–535. Springer, 2007.

[23] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In K. Etessami and S. K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005)*, volume 3576 of *Lecture Notes in Computer Science*, pages 296–300. Springer, Jul 2005.

[24] M. Daum, S. Maus, N. Schirmer, and M. N. Seghir. Integration of a software model checker into Isabelle. In G. Sutcliffe and A. Voronkov, editors, *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, volume 3835 of *Lecture Notes in Computer Science*, pages 381–395. Springer, Dec 2005.

[25] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[26] P. Derrin, D. Elkaduwe, and K. Elphinstone. *seL4 Reference Manual*. National ICT Australia, Sep 2006. http://www.ertos.nicta.com.au/research/sel4/sel4-refman.pdf.

[27] G. Duval and J. Julliand. Modelling and verification of the RUBIS μ-kernel with SPIN. In *Proceedings of the 1st International SPIN Workshop on Model Checking Software (SPIN 95)*, INRS-Télécommunications, Montréal, Quebec, Oct 1995.

[28] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS XI)*, page 6, San Diego, CA, USA, May 2007. Online proceedings at http://www.usenix.org/events/hotos07/tech/.

[29] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In Y. Berbers and W. Zwaenepoel, editors, *Proceedings of the 2006 EuroSys Conference*, pages 177–190. ACM, 2006.

[30] C. Feather. A formal model of sequence points and related issues: Working draft. Technical Report N925, ISO/IEC JTC1/SC22/WG14, Sep 2000.

[31] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In J. Davies, W. Schulte, and M. Barnett, editors, *Proceedings of the 6th International Conference on Formal Methods and Software Engineering (ICFEM 2004)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, Nov 2004.

[32] A. Fox. Formal verification of the ARM6 micro-architecture. Technical Report 548, University of Cambridge Computer Laboratory, Nov 2002.

[33] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In J. Hurd and T. F. Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *Lecture Notes in Computer Science*, pages 1–16. Springer, Aug 2005.

[34] M. Gordon. From LCF to HOL: a short history. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 169–185, Cambridge, MA, USA, 2000. MIT Press.

[35] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[36] S. Hallem, B. Chelf, Y. Xie, and D. R. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and*

Implementation, volume 37 of *SIGPLAN Notices*, pages 69–82. ACM, 2002.

[37] T. Hallgren, M. P. Jones, R. Leslie, and A. P. Tolmach. A principled approach to operating system construction in Haskell. In O. Danvy and B. C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, volume 40 of *SIGPLAN Notices*, pages 116–128. ACM, Sep 2005.

[38] J. Harrison. *Theorem Proving with the Real Numbers*. Springer, 1998.

[39] J. Harrison. A HOL theory of Euclidean space. In J. Hurd and T. F. Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *Lecture Notes in Computer Science*, pages 114–129. Springer, Aug 2005.

[40] R. A. Heinlein. *The moon is a harsh mistress*. Tom Doherty Associates, 1966.

[41] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *Operating Systems Review*, 41(3), Jul 2007.

[42] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In T. Ball and S. K. Rajamani, editors, *Proceedings of the 10th International SPIN Workshop on Model Checking Software (SPIN 2003)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, May 2003.

[43] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[44] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-03-März, TU Dresden, 2002.

[45] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2001)*, volume 36 of *SIGPLAN Notices*, pages 14–26. ACM, 2001.

[46] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In C. S. Ellis, editor, *Proceedings of the 2002 USENIX Annual Technical Conference, General Track*, pages 275–288. USENIX, Jun 2002.

[47] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[48] G. Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[49] G. Klein and H. Tuch. Towards verified virtual memory in L4. In K. Slind, editor, *Emerging Trends: Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLS 2004)*, Technical Report UUCS-05-004, School of Computing, University of Utah, Sep 2004.

[50] R. Kolanski. A logic for virtual memory. In *Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008) — to appear*, Sydney, Australia, Feb 2008.

[51] D. Kroening. Application specific higher order logic theorem proving. In S. Autexier and H. Mantel, editors, *Proceedings of the 2nd Verification Workshop (VERIFY 2002)*, Technical Report no. 2002/07, pages 5–15, DIKU, July 2002.

[52] L4Ka Team. *L4* e*X*perimental *Kernel Reference Manual Version X.2*. University of Karlsruhe, Oct 2001. http://l4ka.org/projects/version4/l4-x2.pdf.

[53] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctnes. In B. K. Aichernig and B. Beckert, editors, *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, pages 2–12. IEEE Computer Society, 2005.

[54] D. Leinenbach and E. Petrova. Pervasive compiler verification — from verified programs to verified systems. In *Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008) — to appear*, Sydney, Australia, Feb 2008.

[55] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 42–54. ACM, Jan 2006.

[56] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. Accepted to the Journal of Automated Reasoning, Oct 2007.

[57] J. Liedtke. On μ-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles (SOSP 95)*, volume 29 of *Operating System Review*, pages 267–284. ACM, Dec 1995.

[58] P. N. Loewenstein, S. Chaudhry, R. Cypher, and C. Manovit. Multiprocessor memory model verification. In *Automated Formal Methods*, Seattle, USA, Aug 2006.

[59] N. Marti, R. Affeldt, and A. Yonezawa. Verification of the heap manager of an operating system using separation logic. In *Third workshop on Semantics, Program Analysis, and Computing Environments For Memory Management (SPACE 2006)*, pages 61–72, Charleston, South Carolina, Jan 2006.

[60] J. McCarthy. Towards a mathematical science of computation. In *Proceedings of the IFIPS Congress 1962*, pages 21–28. North-Holland Publishing Company, 1963.

[61] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In D. Michie and B. Meltzer, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[62] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199(1-2):200–227, 2005.

[63] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 37 of *SIGPLAN Notices*, pages 221–231. ACM, 2001.

[64] J. M. Morris. A general axiom of assignment. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology (Proceedings of the 1981 Maktoberdorf Summer School)*, pages 25–51, 1982.

[65] Y. Moy. Union and cast in deductive verification. In *C/C++ Verification Workshop*, Technical Report ICIS-R07015, pages 1–16, Radboud University Nijmegen, Jul 2007.

[66] M. O. Myreen, A. C. Fox, and M. J. Gordon. A Hoare logic for ARM machine code. In F. Arbab and M. Sirjani, editors, *Proceedings of the International Symposium on Fundamentals of Software Engineering (FSEN 2007)*, volume 4767 of *Lecture Notes in Computer Science*, pages 272–286. Springer, Apr 2007.

[67] M. O. Myreen and M. J. Gordon. A Hoare logic for realistically modelled machine code. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, volume 4424 of *Lecture Notes in Computer Science*, pages 568–582. Springer, 2007.

[68] G. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[69] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. N. Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction (CC 2002)*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[70] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, SRI International, 1980.

[71] Z. Ni, D. Yu, and Z. Shao. Using XCAP to certify realistic systems code: Machine context management. In K. Schneider and J. Brandt, editors, *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2007)*, volume 4732 of *Lecture Notes in Computer Science*, pages 189–206. Springer, Sep 2007.

[72] T. Nipkow. Term rewriting and beyond — theorem proving in Isabelle. *Formal Aspects of Computing*, 1(4):320–338, 1989.

[73] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[74] M. Norrish. *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998.

[75] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Proceedings of the 15th International Workshop on Computer Science Logic (CSL 2001)*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, Sep 2001.

[76] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 268–280. ACM, 2004.

[77] D. v. Oheimb. Information flow control revisited: Noninfluence = Noninterference + Nonleakage. In P. Samarati, P. Y. A. Ryan, D. Gollmann, and R. Molva, editors, *Proceedings of the 9th European Symposium on Research in Computer Security (ESORICS 2004)*, volume 3193 of *Lecture Notes in Computer Science*, pages 225–243. Springer, Sep 2004.

[78] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[79] V. Preoteasa. Mechanical verification of recursive procedures manipulating pointers using separation logic. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Proceedings of the 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 508–523. Springer, 2006.

[80] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structures. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millenial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.

[81] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, Jul 2002.

[82] D. M. Ritchie. The development of the C language. In *Proceedings of the ACM History of Programming Languages Conference (HOPL-II)*, volume 28 of *SIGPLAN Notices*, pages 201–208. ACM, April 1993.

[83] J. Rushby. A trusted computing base for embedded systems. In *Proceedings of the 7th DoD/NBS Computer Security Initiative Conference*, pages 294–311, Gaithersburg, MD, Sep 1984.

[84] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1999)*, pages 105–118. ACM, 1999.

[85] N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[86] B. Schlich and S. Kowalewski. Model checking C source code for embedded systems. In T. Margaria, B. Steffen, and M. G. Hinchey, editors, *Proceedings of the IEEE/NASA Workshop Leveraging Applications of Formal Methods, Verification, and Validation (IEEE/NASA ISoLA 2005)*, pages 65–77. NASA, Maryland, USA, 2005. NASA/CP-2005-212788.

[87] J. Shapiro. Programming language challenges in systems codes: why systems programmers still use C, and what to do about it. In C. W. Probst, editor, *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems (PLOS 2006)*, page 9. ACM, 2006.

[88] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In O. Nierstrasz and M. Lemoine, editors, *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 99)*, volume 1687 of *Lecture Notes in Computer Science*, pages 180–198. Springer, Sep 1999.

[89] S. Sridhar and J. S. Shapiro. Type inference for unboxed types and first class mutability. In C. W. Probst, editor, *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems (PLOS 2006)*, page 7. ACM, 2006.

[90] System Architecture Group. The L4Ka::Pistachio microkernel. White paper, University of Karlsruhe, May 2003. http://l4ka.org/projects/pistachio/pistachio-whitepaper.pdf.

[91] H. Tews. Verifying Duff's device: A simple compositional denotational semantics for goto and computed jumps, 2004. http://www.cs.ru.nl/~tews/Goto/goto.pdf.

[92] H. Tews. Well-behaved memory on top of virtual memory, Aug 2006. Talk at 2nd NICTA International Workshop on Operating Systems Verification — http://www.cs.ru.nl/~tews/Talks/wellbehavedmem.ps.gz.

[93] H. Tuch and G. Klein. A unified memory model for pointers. In G. Sutcliffe and A. Voronkov, editors, *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, volume 3835 of *Lecture Notes in Computer Science*, pages 474–488. Springer, Dec 2005.

[94] H. Tuch, G. Klein, and G. Heiser. OS verification — now! In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, pages 7–12, Santa Fe, NM, USA, Jun 2005. USENIX.

[95] H. Tuch, G. Klein, and M. Norrish. Verification of the L4 kernel memory allocator. Formal proof document. http://www.ertos.nicta.com.au/research/l4.verified/kmalloc.pml, Jul 2006.

[96] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*, pages 97–108. ACM, Jan 2007.

[97] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: a practical tool for OS implementors. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS VI)*, pages 20–25, Cape Cod, MA, USA, May 1997. USENIX.

[98] VerifiCard project. http://verificard.org, 2005.

[99] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.

[100] T. Weber. Towards mechanized program verification with separation logic. In J. Marcinkowski and A. Tarlecki, editors, *Proceedings of the 18th International Workshop on Computer Science Logic (CSL 2004)*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2004.

[101] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. P. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer, Aug 1997.

[102] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, TU München, 2002.

[103] A. Wiggins, H. Tuch, V. Uhlig, and G. Heiser. Implementation of fast address-space switching and TLB sharing on the StrongARM processor. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference (ACSAC 2003)*, volume 2823 of *Lecture Notes in Computer Science*, pages 352–364. Springer, Sep 2003.

[104] M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2004.

[105] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In M. Nielsen and U. Engberg, editors, *Proceedings of the 5th International*

Conference on Foundations of Software Science and Computation Structures (FOSSACS 2002), volume 2303 of *Lecture Notes in Computer Science*, pages 402–416. Springer, Apr 2002.

Index