J Autom Reasoning (2009) 42:125–187
DOI 10.1007/s10817-009-9120-2

# Formal Verification of C Systems Code

## Structured Types, Separation Logic and Theorem Proving

**Harvey Tuch**

Received: 26 February 2009 / Accepted: 26 February 2009 / Published online: 2 April 2009


**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 paper we present a study in the mechanisation of two proof abstractions for pointer program verification in the Isabelle/HOL theorem prover, based on a low-level memory model for C. The language's type system presents challenges for the multiple independent typed heaps (Burstall-Bornat) and separation logic proof techniques. In addition to issues arising from explicit value size/alignment, padding, type-unsafe casts and pointer address arithmetic, structured types such as C's arrays and **struct**s are problematic due to the non-monotonic nature of pointer and lvalue validity in the presence of the unary &-operator. For example, type-safe updates through pointers to fields of a **struct** break the independence of updates across typed heaps or $\wedge^*$-conjuncts. We provide models and rules that are able to cope with these language features and types, eschewing common over-simplifications and utilising expressive shallow embeddings in higher-order logic. Two case studies are provided that demonstrate the applicability of the mechanised models to real-world systems code; a working of the standard in-place list reversal example and an overview of the verification of the L4 microkernel's memory allocator.

**Keywords** Separation logic · C · Interactive theorem proving

H. Tuch (✉)
Sydney Research Lab., National ICT Australia, Eveleigh, Australia
e-mail: htuch@cse.unsw.edu.au

## 1 Introduction

Systems code constitutes the lowest layer of the software stack, typically an operating system, hypervisor, language run-time, real-time executive or the like. The vast majority of systems code today is implemented in the C [39] programming language or some variant such as C++ and Objective-C. Higher-level languages, for example Java or ML, are incompatible with the goals of system implementation due to the many abstraction breaking requirements, for example zero-copy I/O, address translation manipulation and control over data structure layout [43].

We would like to have high confidence in the correctness of the implementation of this foundational aspect of any system and a rigorous approach requires formal arguments to be made about the behaviour of low-level C source code. This entails being able to precisely model the syntax of the language and specifications, as well as the language semantics, and possessing sound proof rules to manipulate these statements. In addition, as a practical necessity we need to be able to produce a proof in a timely and economical manner.

Unfortunately, C was not developed with formalisation in mind and it has a low level of type and memory abstraction—details down to the bit-layout of values, references and data structures are not opaque and it is easy to violate the C type system by its cast mechanism and through address arithmetic. C does not have garbage collection and the programmer is responsible for allocation and deallocation of memory through library calls.

Over the last decade, developments in interactive theorem proving technology [31, 35] and studies in real-world language mechanisation [20, 21, 32, 34] have paved the way to verification of actual source code, as the compiler sees it. Previously this had been only a theoretical possibility and toy problems and simplified languages were the norm. In this paper we address the challenges and details of both formalising C's memory model and types in higher-order logic with the Isabelle theorem proving system, and present developments of two commonly used proof techniques for reasoning about pointer programs. We have chosen this emphasis here due to the critical importance of reasoning about inductively-defined mutable data structures, such as linked lists and trees, in C systems code verification. For example, most contemporary operating systems feature thread/process control blocks, scheduler queues, page tables and more than one kernel memory allocator—non-trivial pointer linked data structures.

Below we provide an introduction to the aliasing and frame problems that cause much of the difficulty in reasoning about pointer programs, the Burstall-Bornat and separation logic proof techniques that tame them and the additional complications that the C language burdens the verification effort with. We then conclude the introduction with a summary of our contributions and an overview of the rest of the paper.

### 1.1 C Pointer Program Verification

#### *1.1.1 Aliasing and Frame Problems*

For an example of the aliasing problem, 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 paper 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.

### *1.1.2 Burstall-Bornat and Separation Logic*

We first consider a simplified semantic model for the heap where 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 language 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 modelled 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:

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

Since this is the same view of memory as in the previous section, the aliasing and frame problems described above are present.

Multiple independently typed heaps, also known as the *Burstall-Bornat* model, allow us to rule out the adverse effects of inter-type aliasing for type-safe languages by having a separate heap variable for each language type in the program's state space, e.g. *float-heap* :: *float ptr* $\rightharpoonup$ *float*,[1] *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 [7] 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.

Independently 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 [18, 38]. 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 allows 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.

### *1.1.3 Problematic C Language Features*

While both of the above proof abstractions are clearly inspired by low-level imperative languages, there is some additional effort necessary to apply them to C.

The Burstall-Bornat model assumes type-safety but C does not guarantee this and hence multiple typed heaps are unsound as a fundamental memory model for the

[1] *float ptr*, *int ptr* etc. are a HOL type encoding of the respective C types **float** *, **int** *, etc. We formally introduce the types in Section 3.2.2.

language. Such a model also does not support language features we require such as casts and pointer arithmetic. Finally, Bornat's restrictions do not apply to the language as **struct**s are first-class types in C. We see later in this paper how we can recover an extended Burstall-Bornat model for C in the situation in which we are operating in the type-safe fragment of the language while still supporting concrete proofs about unsafe code.

With separation logic there are similar problems with type-unsafe languages. In particular, there is the problem of *skewed sharing* [38] related to inter-type aliasing, which we discuss at the end of Section 7.3. The frame rule also requires special treatment to be applicable to C programs that use a more relaxed notion of memory safety than is usually present in other treatments in the literature. Finally, additional rules are needed to make reasoning about **struct**s feasible.

### 1.2 Contributions

This paper provides the following contributions:

- A rigorous treatment of the Burstall-Bornat 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. We treat both standard and implementation defined behaviour and 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 Burstall-Bornat 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.
- Two case studies are provided—a proof of the standard in-place list reversal implementation and a summary of a study in the application of the framework to the verification of a kernel memory allocator for the L4Ka::Pistachio [44] microkernel implementation. This contains unsafe code that exercises our framework and provides an opportunity to compare the two proof abstractions studied in the same setting.

### 1.3 Overview

We briefly introduce the non-standard notation that appears in the paper in the next section. Following this, we describe our C verification environment and approach to providing semantics for C. We then examine in some more detail the problem that C's structured types pose for proofs about pointer programs and give an in-depth treatment of the development of the multiple typed heaps and separation logic proof rules in Isabelle/HOL, including support for proofs about programs featuring structured types. Finally, the two case studies are provided and we conclude with

some discussion of the engineering effort in both the verifications and the theory development.

## 2 Notation

Our meta-language Isabelle/HOL conforms largely to everyday mathematical notation. This section introduces further non-standard notation and in particular a few basic data types along with their primitive operations.

The space of total functions is denoted by $\Rightarrow$. Type variables are written $\alpha$, $\beta$, etc. The notation $t :: \tau$ means that HOL term $t$ has HOL type $\tau$. The option type

$$\textbf{datatype}\ \alpha\ \mathit{option} = \bot \mid \mathsf{Some}\ \ \alpha$$

adjoins a new element $\bot$ to a type $\alpha$. We use $\alpha$ *option* to model partial functions, writing $\lfloor a \rfloor$ instead of $\mathsf{Some}\ a$ and $\alpha \rightharpoonup \beta$ instead of $\alpha \Rightarrow \beta$ *option*. The $\mathsf{Some}$ constructor has an underspecified inverse called $\mathsf{the}$, satisfying $\mathsf{the}\ \lfloor x \rfloor = x$. Function update is written $f(x := y)$ where $f :: \alpha \Rightarrow \beta$, $x :: \alpha$ and $y :: \beta$ and $f(x \mapsto y)$ stands for $f\ (x := \lfloor y \rfloor)$. We can also merge two partial functions with $f +\!\!+ g$, defined $\lambda x.\ \mathsf{case}\ g\ x\ \mathsf{of}\ \bot \Rightarrow f\,x \mid \lfloor y \rfloor \Rightarrow \lfloor y \rfloor$. Domain restriction is $f\restriction_A$ where $f :: \alpha \rightharpoonup \beta$ and $(f\restriction_A)\ x = (\mathsf{if}\ x \in A\ \mathsf{then}\ f\,x\ \mathsf{else}\ \bot)$.

Implication is denoted by $\Longrightarrow$ and $[\![\ A_1;\ \ldots;\ A_n\ ]\!] \Longrightarrow A$ abbreviates $A_1 \Longrightarrow (\ldots \Longrightarrow (A_n \Longrightarrow A)\ldots)$.

Finite integers are represented by the type $\alpha$ *word* where $\alpha$ determines the word length. For succinctness, we use abbreviations like *word8* and *word32*. The functions $\mathbb{N}^{\Leftarrow}$ and $\mathbb{N}^{\Rightarrow}$ convert $\alpha$ *word*s to and from natural numbers. Arithmetic operations on bit-vector values are modulo $2^n$, where $n$ is the word length.

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 $\vdash \{\!|P|\!\}\ c\ \{\!|Q|\!\}$ where $P$ and $Q$ are assertions and $c$ a program. In assertions, we use the syntax $\acute{x}$ to refer to the program variable $x$ in the current state, while ${}^{\sigma}x$ means $x$ in state $\sigma$. Program states can be bound in assertions by $\{\!|\sigma.\ P|\!\}$.

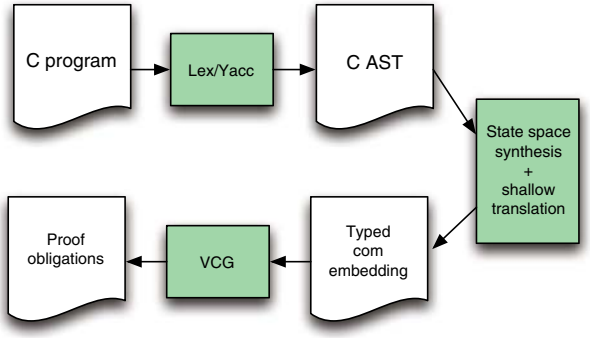
Isabelle supports axiomatic type classes [49] similar to, but more restrictive than Haskell's. The notation $\alpha$::*ring* restricts the type variable $\alpha$ to those types that support the axioms of class *ring*. Type classes can be reasoned about abstractly, with recourse just to the defining axioms. Further, a type $\tau$ can be shown to belong to a type class given a proof that the class's axioms hold in $\tau$. All abstract consequences of the class's axioms then follow for $\tau$.

For every Isabelle/HOL type $\alpha$ we can derive a type $\alpha$ *itself*, consisting of a single element denoted by $\mathsf{TYPE}(\alpha)$. This provides a convenient way to restrict the type of a term when working with polymorphic definitions.

## 3 C Verification Framework

Our verification flow begins with C source code written in a syntactic subset of ANSI C and annotated with pre/post conditions and invariants in the style of Hoare logic. This undergoes an automated parsing and translation into HOL and from there

**Fig. 1** C verification processes




a mechanical verification condition generator applies weakest pre-condition rules, leaving the verifier with a set of HOL proof obligations to discharge.

This verification approach is quite common for imperative languages and Schirmer [41] has developed a generic verification environment built on Isabelle/HOL for this purpose. This environment includes a deeply embedded language, called *com*, which provides a number of primitive constructs which the language features of an imperative language can be mapped to, for example sequential, conditional and While statements. Machine-checked HOL operational semantics, Hoare logics and a verification condition generator are provided for *com* in the environment. Our task is then to translate C syntax to the *com* language such that the *com* semantics reflect the behaviours described in the C standard [1] for a given program.

Figure 1 illustrates the verification process. Below we describe our C subset and the translation of C expressions and statements to *com*, focusing on pointer and memory related aspects that we later build upon.

### 3.1 Syntax

We base our grammar on Kernighan and Ritchie [19], section A.13, essentially ANSI C with some restrictions. The philosophy behind the subset is that the idioms and practises of systems code should be supported while it is acceptable to limit some constructs that can be replaced with slightly more verbose equivalents where they unnecessarily complicate the translation to *com*. Pre/post condition and loop invariant annotations are expressed as Isabelle/HOL expressions with inline C comments, e.g. `/** /INV: {`$\sigma\,.\,P\,\sigma$`} */`.

The most significant restriction is that all expressions are required to be side-effect free and we 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. This is similar to the approach taken in CIL [30]. Any standard C compiler will continue to recognise the expression statements below as they are indistinguishable from cases of the C expression statement. This restriction is motivated by the non-determinism in the order of expression evaluation inherent in the standard. This in

turn complicates both the translation and resulting proof obligations. The expression statement production in our grammar is:

```
expression_statement =
  unary_expression = expression ; |
  unary_expression = call_expression ; |
  call_expression ; |
  unary_expression ++ ; |
  unary_expression -- ; |
  ;
```

Other limitations of interest in the subset are:

- Programs are assumed to be preprocessed and free of directives removed at this stage, that is there are no macros present.
- Pointers to automatic storage are not supported, i.e. the & operator is disallowed for stack variables. This provides for a simple model of stack variables in the state space described in Section 3.2.1. Since it is not usual to have automatic storage linked to mutable inductively-defined data structures, we consider reasoning about issues such as dangling pointers and pass-by-reference orthogonal to the focus of this paper.
- No **float**, function pointer or nested **union** types.
- **switch** and **goto** statements are elided. These statements violate block structure and their semantics have been treated elsewhere in the literature [45].

Norrish [32, 47] has described formal YACC and LEX grammars for this subset.

## 3.2 Semantics

The translation targets a mixed deep-shallow embedding, where most statements in *com* are deeply embedded but expressions and state transformers for expression statements are HOL sets and functions. For example, we might translate p = q + 5; **if** (p) x = 2; to Seq (Basic ($\lambda s.\ s(\!| p := q\ s + 5 |\!)$)) (Cond {$s.\ p\ s \neq 0$} (Basic ($\lambda s.\ s(\!| x := 2 |\!)$)) Skip).

Schirmer [41] gives both small- and big-step operational semantics for *com*. These are parametrised by the state space and the shallowly embedded expression and state update functions that operate on the state space. We describe below first the state space for our embedding and how C's types are encoded as HOL types, and then explore aspects of the shallow embedding that relate to the memory model and reasoning about pointer programs.

### *3.2.1 State Space*

The state for C programs is modelled with a **record** in the verification environment. Since we disallow references to stack variables, they can be treated as fields of the **record**. The state type is synthesised for each program based on the variable names and types present.

The heap is also a member field with a function type. 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. $*x = 0$ xdeadbeef 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**.

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:

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

### *3.2.2 Type Encoding*

The low-level heap model corresponds to the C programmer's notion of an underlying byte map, however we do not wish to reason at this level in most proofs, preferring more abstract typed HOL values. By utilising HOL types we can harness the theorem prover's type checking and inference abilities and make use of existing libraries for the types.

Each language type is assigned a unique type in the theorem prover's logic and placed in a type class providing functions to allow heap dereferencing expression semantics to be defined. The $\alpha$::*c-type* class introduces several constants that connect the low-level byte representation and HOL values for encoded language types. A relatively simple approach to this would be to define the functions:

$$\begin{array}{ll} \mathsf{to\text{-}bytes} & :: \alpha{::}\mathit{c\text{-}type} \Rightarrow \mathit{byte\ list} \\ \mathsf{from\text{-}bytes} & :: \mathit{byte\ list} \rightharpoonup \alpha{::}\mathit{c\text{-}type} \\ \mathsf{size\text{-}of} & :: \alpha{::}\mathit{c\text{-}type\ itself} \Rightarrow \mathit{nat} \\ \mathsf{align\text{-}of} & :: \alpha{::}\mathit{c\text{-}type\ itself} \Rightarrow \mathit{nat} \end{array}$$

for each language type $\alpha$. For C's scalar types, integers can be encoded in the expected way with bit-vectors, for example the mixed-endian representation in Fig. 2.

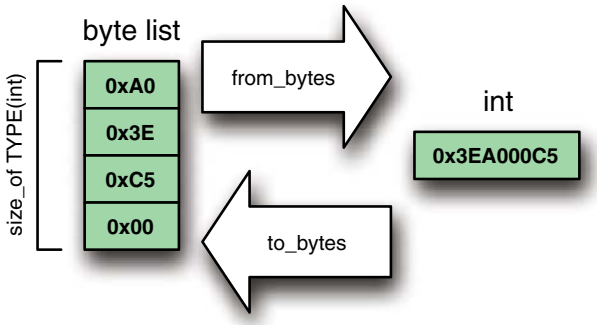



**Fig. 2** Mixed-endian integer encoding

For pointers a distinct Isabelle pointer type for each C Isabelle type can be obtained with:

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

and instantiated as an $\alpha$::*c-type*. The $\alpha$ on the left-hand side is a phantom type that allows the base type to be carried in Isabelle/HOL's type system, used to constrain the action of various pointer operators by making use of the type information associated with the $\alpha$ value. The destructor for the $\alpha$ *ptr* type is written $p_{\&}$. The *footprint* for a pointer $p$::$\alpha$::*c-type ptr* is the set of addressable locations that contain the byte representation for a value of type $\alpha$ with base $p_{\&}$. In this section, with respect to the *heap-mem* state, it is simply an *addr set*, and can be obtained with $\{p_{\&}..+\mathsf{size\text{-}of\ TYPE}(\alpha)\}$. Later we consider footprints with heap states that have a type index component in the address.

Trap representations are caught in the semantics by wrapping the statement containing a heap dereference expression with a guard predicate checking for a valid encoding and leading to failure semantics if this is not the case.

C standard and other properties can also be captured with an $\alpha$::*mem-type* axiomatic type class that further constrains $\alpha$::*c-type*:

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

where the constant |addr| represents the size of the address space, e.g. $2^{32}$. 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. For tiling in an array, [AlignDvdSize] is needed. Finally, the alignment constraint

[ALIGN] is added to make pointer arithmetic better behaved and it holds with a power-of-two alignment restriction which we assume.

By only requiring the above axioms of types representations, instantiations can include behaviours the C standard considers implementation defined or unspecified. Proofs can rely on the basic properties independent of the specific instantiations.

The above type encoding does not cope with features of C's structured types such as padding bytes and is not suitable as a basis for abstractly reasoning about the effects of updates through pointers to fields. For this we require a more sophisticated type encoding which we explore in Section 5.

### *3.2.3 Expressions*

Translation of arithmetic, bit-wise, logical and relational expressions is straight forward, utilising the operators provided in the theorem prover's bit-vector library with the occasional guard predicate added to prevent illegal behaviour. Below we discuss the translation for heap dereferences, pointer arithmetic, **sizeof** and casts, deferring **struct** and array operators to Section 5.

Heap dereferences in expressions, e.g. $*\mathrm{p} + 1$ 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.

$$
\begin{aligned}
&\mathsf{heap\text{-}list} :: \mathit{heap\text{-}mem} \Rightarrow \mathit{nat} \Rightarrow \mathit{addr} \Rightarrow \mathit{byte\ list}\\
&\mathsf{heap\text{-}list}\ h\ 0\ p \equiv [\,]\\
&\mathsf{heap\text{-}list}\ h\ (\mathit{Suc}\ n)\ p \equiv h\ p \cdot \mathsf{heap\text{-}list}\ h\ n\ (p+1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{h\text{-}val} :: \mathit{heap\text{-}mem} \Rightarrow \alpha :: \mathit{c\text{-}type}\ \ \mathit{ptr} \rightharpoonup \alpha\\
&\mathsf{h\text{-}val}\ h\ p \equiv \mathsf{from\text{-}bytes}\ (\mathsf{heap\text{-}list}\ h\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha))\ p_{\&})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{lift} :: \mathit{heap\text{-}mem} \Rightarrow \alpha :: \mathit{c\text{-}type}\ \mathit{ptr} \Rightarrow \alpha\\
&\mathsf{lift}\ h \equiv \lambda p.\ \mathsf{the}\ (\mathsf{h\text{-}val}\ h\ p)
\end{aligned}
$$

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.

Pointer addition is defined for $p{::}\alpha{::}$*c-type ptr* as:

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

Here the left-hand $\alpha$ supplies the size information.

**sizeof** is expressed as an application of size-of. When a value is cast between two non-pointer scalars we use the expected bit-vector conversions subject to the restrictions and promotion rules in the standard. When casting to or from a pointer, the underlying address represented in the pointer is exposed. For example, casting a **char** * $p$ to a **int** * translates to $\lambda s.\ \mathsf{Ptr}\ p_{\&}$ (also written $\lambda s.$ ptr-coerce $p$ in this paper). This relies on implementation behaviour, as the C standard has only weak requirements for pointer representation, but systems code is typically dependent on this anyway.

*3.2.4 Guards*

*com* has Guard statements that lead to failure semantics if guard expressions do not hold. We use these to wrap other statements featuring heap expressions that require checks for NULL pointer dereferences, trap representations, alignment, etc. We refer to these expressions as *guard predicates* routinely in the paper.

Pointer guard predicates creep into the resultant proof obligations, so we have additional support later for reasoning about them. There is a continuum on the strength of the predicates possible in our framework, from essentially no checks at all to enforcing strict standard compliant behaviour. We opt for the following guards on pointer expressions here:

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

*3.2.5 Statements*

We now give the semantics of statements producing side-effects on the heap.[2] This is achieved with the heap-update state transformer:

$$\begin{array}{l}\text{heap-update-list} :: addr \Rightarrow byte\ list \Rightarrow heap\text{-}mem \Rightarrow heap\text{-}mem \\ \text{heap-update-list } p\ [\ ]\ h \equiv h \\ \text{heap-update-list } p\ (x \cdot xs)\ h \equiv \text{heap-update-list } (p+1)\ xs\ (h(p := x))\end{array}$$

$$\begin{array}{l}\text{heap-update} :: \alpha :: c\text{-}type\ ptr \Rightarrow \alpha \Rightarrow heap\text{-}mem \Rightarrow heap\text{-}mem \\ \text{heap-update } p\ v\ h \equiv \text{heap-update-list } p_{\&} \text{ (to-bytes } v)\ h\end{array}$$

As an example of how the statement and expression translations interact, ∗p = ∗q + 5 is translated to the state transformer $\lambda s.$ heap-update $p$ (lift $s\ q + 5$) $s$.

**Lemma 3.1** *Heap updates do not affect heap reads providing the regions do not overlap:*

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

*Proof* By induction on $k$. □

## 4 C's Struct, Union and Array Types

In the C-HOL type encoding of Section 3.2.2, each C type was given a unique type in the theorem prover and all such types belonged to the $\alpha$::*c-type* type class.

[2]For full details of the expression and statement translation we refer the reader to the author's thesis [46].

C's aggregate, or *structured*, types can also be modelled inside the theorem prover in this manner, e.g. **struct** types as Isabelle **record** types.[3] Structured types warrant further investigation, however, as they require additional work behind the scenes to instantiate and expose limitations in the notion of independently typed heaps.

As an initial technical consideration, each structured type appearing in a program requires the C translator, implemented as an Isabelle tactic in ML, to perform an $\alpha$::*mem-type* instantiation during state space synthesis, e.g. for **struct** types:

- A corresponding **record** declaration.
- Definitions of functions appearing in $\alpha$::*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 an unpleasant redundancy. One of the aims of the following development is to eliminate this.

Leaving aside this implementation aspect, we now turn to the fundamental problem that structured types pose.

*Example 4.1* 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 existing pointer proof techniques:

$$\{\!|\, *p = (\!|\, y = 2,\ z = 'm' \,|\!) \,|\!\}\ p{\rightarrow}y = 1;\ \{\!|\, *p = ? \,|\!\}$$

The problem here is that even though the update and dereference are type-safe, and we do not need to consider aliasing, proof rules based on independently typed heaps 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.

[3] 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** *.

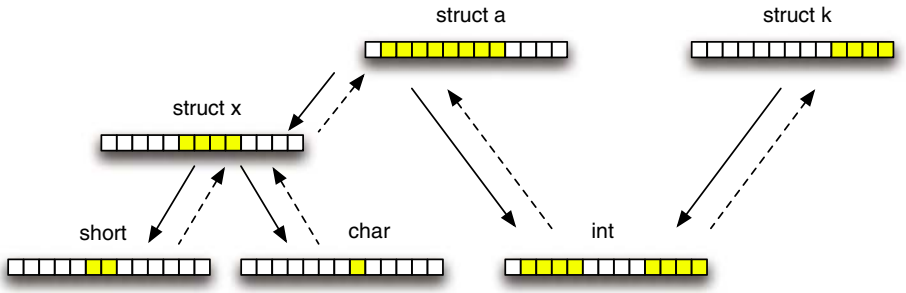



**Fig. 3** Heap update dependencies

Figure 3 demonstrates how this problem manifests itself in the multiple typed heaps abstraction. Typed heaps have locations in common, and there is an update dependency given by the arrows between the heaps:

- 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.

In the next three sections we describe a generalised development of the type encoding, multiple typed heaps and separation logic based on a deep embedding of structured type information. By treating type information as a first-class HOL value we are able to both avoid the instantiation redundancy described above and provide generic definitions and rules for structured types.

The above discussion focused on **struct**s. There are two distinct situations in which arrays require treatment. First, when appearing as objects in the heap. Here, array expressions are given pointer semantics as required by the standard [1, 6.3.2.1–3]. Where arrays appear as members of other objects or as automatic variables, they must have a constant size[4] and are represented using the techniques of Harrison [15]. In this case, array $\alpha$::*c-type* definitions and $\alpha$::*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. **union**s are more difficult, however if we forbid their nesting it is conceivable they could be handled through the type encoding. Cock [12] has developed such support for tagged unions and bitfields.

## 5 Structured Type Encoding

The solution proposed in Section 4 requires that structured type meta-data be available at the HOL level. This needs to include the same information as in

[4]We elide the special case of flexible array members [1, 6.7.2.1–16].

Section 3.2.2, for example 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 in the development of the rules in Sections 6 and 7.

### 5.1 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 Section 6.4. To facilitate this, functions derived from the corresponding **record** functions are included in the type meta-data.

**Definition 5.1** We can capture *abstract* record access and update functions for fields as *field descriptions*:

$$\begin{aligned} \textbf{record}\ \alpha\ \textit{field-desc} \quad = \quad & \mathsf{field\text{-}access} :: \alpha \Rightarrow \textit{byte list} \Rightarrow \textit{byte list} \\ & \mathsf{field\text{-}update} :: \textit{byte list} \Rightarrow \alpha \Rightarrow \alpha \end{aligned}$$

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.[5] We similarly extend the to-bytes type signature in the rest of the paper to support padding fields:

$$\mathsf{to\text{-}bytes} :: \alpha :: \textit{c-type} \Rightarrow \textit{byte list} \Rightarrow \textit{byte list}$$

*Example 5.1* The field description for field b in **struct a** is:

$$\begin{aligned} &(\!|\mathsf{field\text{-}access} = \mathsf{to\text{-}bytes} \circ \mathsf{b}, \\ &\quad \mathsf{field\text{-}update} = \\ &\qquad \lambda \textit{bs}\ v. \\ &\qquad\quad \mathsf{if}\ |\textit{bs}| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\textit{word32})\ \mathsf{then}\ v(\!|\mathsf{b} := \mathsf{from\text{-}bytes}\ \textit{bs}|\!)\ \mathsf{else}\ v|\!) \end{aligned}$$

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] A more conservative, standard compliant approach, would be to use non-determinism or an oracle here.

5.2 Type Descriptions

**Definition 5.2** The type meta-data is captured in a *type description* with the following mutually-inductive definitions:

$$\begin{array}{lllll} \textbf{datatype} & \alpha\ \textit{typ-desc} & = & \mathsf{TypDesc}\ \text{“}\alpha\ \textit{typ-struct}\text{”}\ \textit{typ-name} \\ & \alpha\ \textit{typ-struct} & = & \mathsf{TypScalar}\ \textit{nat}\ \textit{nat}\ \alpha\ \mid \\ & & & \mathsf{TypAggregate}\ (\alpha\ \textit{typ-desc} \times \textit{field-name})\ \textit{list} \end{array}$$

A type description is a tree, with structures as internal nodes, branches labelled with field names and leaves corresponding to fields with primitive types. At leaves, size, alignment and an $\alpha$ are provided. The $\alpha$ is free and can be used to carry primitive type encoding and decoding functions. Alignment is an exponent, enforcing a power-of-two restriction structurally. An example type description for **struct a** is given in Fig. 4.

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.

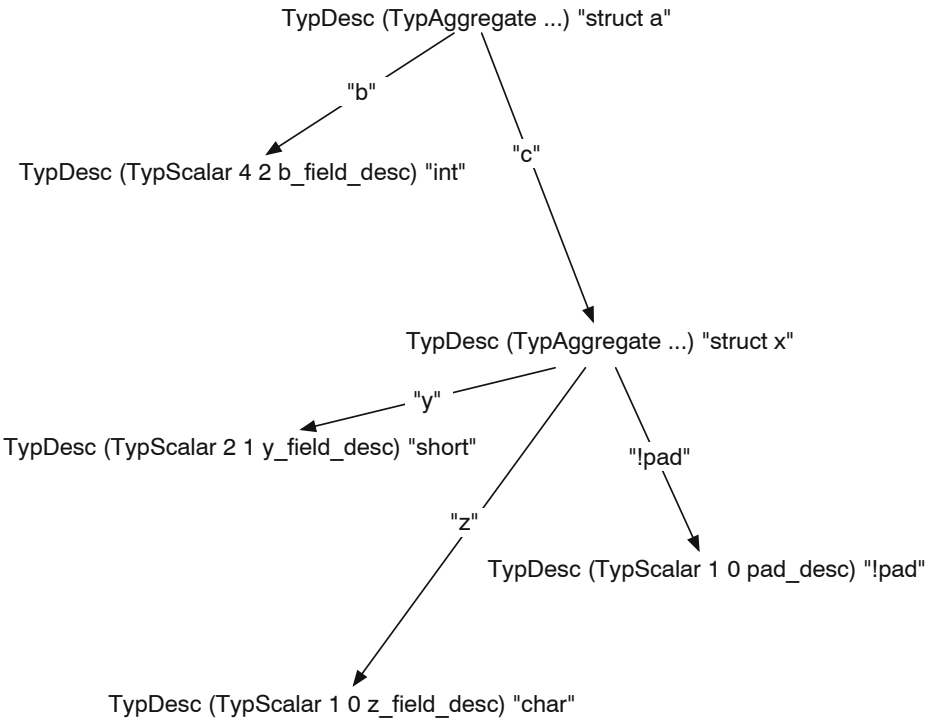



**Fig. 4** Type description for **struct a**

Type descriptions can be specialised in various ways, for example:

$$\begin{array}{ll} \textbf{types} & \alpha\ \textit{typ-info} = \alpha\ \textit{field-desc typ-desc} \\ & \textit{typ-uinfo} = (\textit{byte list} \Rightarrow \textit{byte list})\ \textit{typ-desc} \end{array}$$

The *type information*, $\alpha$ *typ-info*, provides the information required to describe the encoding and decoding of the representation. *Exported type information*, *typ-uinfo*, provide a means of treating language types as first-class values in HOL, divorced from the HOL type to allow direct comparison, and normalising value representations—this is further explained in Section 5.6. $\mathsf{TYPE}(\alpha)_\tau$ gives the type information for an $\alpha$::*c-type* and $\mathsf{TYPE}(\alpha)_\nu$ provides the exported type information. Here the subscript operators are functions from $\alpha$::*c-type itself*.

**Definition 5.3** 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* [], [''*b*''], and [''*c*'', ''*z*''] are valid qualified field names in Example 4.1 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 Section 6.4 and Section 7.6 to be expressed and proven. We summarise all these and the other key functions defined over type descriptions introduced in this chapter in Table 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.

**Definition 5.4** 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.

**Definition 5.5** Type size size-td and alignment align-td are found by summing and taking the maximum of the leaf node sizes and alignments[6] respectively.

**Definition 5.6** lookup :: $\alpha$ *typ-desc* $\Rightarrow$ *qualified-field-name* $\Rightarrow$ *nat* $\rightharpoonup$ $\alpha$ *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 5.3* A lookup on the field c in **struct a** yields:

$$\mathsf{TYPE}(\textit{a-struct})_\nu \triangleright [''c''] = \lfloor(\mathsf{TYPE}(\textit{x-struct})_\nu, 4)\rfloor$$

[6]This is implied by the C standard with power-of-two alignments.

**Table 1** Type description functions

| | | |
|---|---|---|
| map-td | :: | $(nat \Rightarrow \text{nat} \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha\ typ\text{-}desc \Rightarrow \beta\ typ\text{-}desc$ |
| Transforms leaf $\alpha$ values to $\beta$ values. | | |
| size-td | :: | $\alpha\ typ\text{-}desc \Rightarrow nat$ |
| Type size, e.g. size-td $\mathsf{TYPE}(a\text{-}struct)_\tau = 8$. | | |
| align-td | :: | $\alpha\ typ\text{-}desc \Rightarrow nat$ |
| Type alignment exponent, e.g. align-td $\mathsf{TYPE}(a\text{-}struct)_\tau = 2$. | | |
| -▷- | :: | $\alpha\ typ\text{-}desc \Rightarrow qualified\text{-}field\text{-}name \rightharpoonup \alpha\ typ\text{-}desc \times nat$ |
| The sub-tree and offset from the base of the structure that a valid qualified field name leads to. | | |
| td-set | :: | $\alpha\ typ\text{-}desc \Rightarrow (\alpha\ typ\text{-}desc \times nat)\ set$ |
| The set of all sub-trees and their offset from the base of a structure. | | |
| access-ti | :: | $\alpha\ typ\text{-}info \Rightarrow (\alpha \Rightarrow byte\ list \Rightarrow byte\ list)$ |
| Derived field access for the entire structure represented by the type information. | | |
| update-ti | :: | $\alpha\ typ\text{-}info \Rightarrow (byte\ list \Rightarrow \alpha \Rightarrow \alpha)$ |
| Derived field update for the entire structure represented by the type information. | | |
| export-uinfo | :: | $\alpha\ typ\text{-}info \Rightarrow typ\text{-}uinfo$ |
| Export type information (see Section 5.6 for *typ-uinfo*). | | |
| norm-tu | :: | $typ\text{-}uinfo \Rightarrow (byte\ list \Rightarrow byte\ list)$ |
| Derived normalisation for the entire structure represented by the exported type information. | | |
| -≤- | :: | $\alpha\ typ\text{-}desc \Rightarrow \alpha\ typ\text{-}desc \Rightarrow bool$ |
| Update dependency order, e.g. $\mathsf{TYPE}(x\text{-}struct)_v \leq \mathsf{TYPE}(a\text{-}struct)_v$. | | |

A lookup on an invalid field name fails:

$$\mathsf{TYPE}(a\text{-}struct)_v \triangleright [''c'', ''b''] = \bot$$

**Lemma 5.1** *The size of a type description is no smaller than the sum of the size of any field's type description and offset:*

$$\frac{\mathsf{TYPE}(\alpha)_\tau \triangleright f = \lfloor (t, n) \rfloor}{\mathsf{size\text{-}td}\ t + n \leq \mathsf{size\text{-}td}\ \mathsf{TYPE}(\alpha)_\tau}$$

*Proof* By structural induction on the type description. □

**Definition 5.7** A related concept is the *type description set*, td-set $t$, of a type description $t$ where all sub-trees and their offset are returned.

*Example 5.4* The type description set for **struct x** is:

$$\mathsf{td\text{-}set}\ \mathsf{TYPE}(x\text{-}struct)_v = \{(\mathsf{TYPE}(x\text{-}struct)_v, 0), (\mathsf{TYPE}(word16)_v, 0), (\mathsf{TYPE}(word8)_v, 2), (\mathsf{pad\text{-}export}\ 1, 3)\}$$

**Definition 5.8** The address corresponding to an *lvalue* expression containing a structure field access or update can be found with:

$$\&(p \rightarrow f) \equiv p_\& + \mathbb{N}^{\Rightarrow}\ (\mathsf{snd}\ (\mathsf{the}\ (\mathsf{TYPE}(\alpha)_v \triangleright f)))$$

Lvalue terms appear in the semantics and proof obligations for statements like $p->f = v;$. This definition provides the translation for the . and –> operators for heap structures.

*Example 5.5* The lvalue address for an *a-struct ptr* dereference on the c field is given by:

$$\&(p\rightarrow[''c''])= p_{\&} + \mathbb{N}^{\Rightarrow}\ (\text{size-of TYPE}(\mathit{word32}))$$

**Lemma 5.2** *The heap interval footprint of a field is a subset of that of an enclosing structure:*

$$\frac{\text{TYPE}(\alpha)_\tau \triangleright f = \lfloor(t,\ n)\rfloor}{\{\&(p\rightarrow f)..+\text{size-td } t\} \subseteq \{p_{\&}..+\text{size-of TYPE}(\alpha)\}}$$

*Proof* By interval reasoning and Lemma 5.1. □

**Definition 5.9** 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 5.6* The access function for **struct a** is given by:

$$\begin{aligned}\text{access-ti TYPE}(\mathit{a\text{-}struct})_\tau = \lambda v\ bs.\\ &\text{to-bytes (b } v)\\ &\text{(take (size-of TYPE}(\mathit{word32}))\ bs) \ @\\ &\text{to-bytes (c } v)\\ &\text{(take (size-of TYPE}(\mathit{x\text{-}struct}))\\ &\text{(drop (size-of TYPE}(\mathit{word32}))\ bs))\end{aligned}$$

**Definition 5.10** The connection between the HOL typed value, type information, size, alignment and underlying byte representation can be made through the following function definitions:

$$\begin{aligned}&\text{to-bytes} && \equiv \text{access-ti TYPE}(\alpha)_\tau\\ &\text{from-bytes } bs && \equiv \text{update-ti TYPE}(\alpha)_\tau\ bs\ \text{arbitrary}\\ &\text{size-of TYPE}(\alpha) && \equiv \text{size-td TYPE}(\alpha)_\tau\\ &\text{align-of TYPE}(\alpha) && \equiv 2^{\text{align-td TYPE}_{(\alpha)_\tau}}\end{aligned}$$

We write $\text{access-ti}_0$ and $\text{to-bytes}_0$ 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 $\alpha$::*mem-type*s in the next section.

**Table 2** $\alpha$::*mem-type* axioms

| | |
|---|---|
| size-of TYPE($\alpha$) $<$ \|addr\| | [MaxSize] |
| align-of TYPE($\alpha$) dvd size-of TYPE($\alpha$) | [AlignDvdSize] |
| $\text{TYPE}(\alpha)_\tau \triangleright f = \lfloor (s, n) \rfloor \rightarrow 2^{\text{align-td } s}$ dvd $n$ | [AlignField] |
| $\frac{|bs| = \text{size-of TYPE}(\alpha)}{\text{update-ti TYPE}(\alpha)_\tau\ bs\ v = \text{update-ti TYPE}(\alpha)_\tau\ bs\ w}$ | [Upd] |
| wf-desc TYPE($\alpha$)$_\tau$ | [WFDesc] |
| wf-size-desc TYPE($\alpha$)$_\tau$ | [WFSizeDesc] |
| wf-field-desc TYPE($\alpha$)$_\tau$ | [WFFD] |

## 5.3 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 the $\alpha$::*mem-type* axioms in Section 3.2.2, and we show the earlier properties to follow in Theorem 5.3 at the end of the section.

**Definition 5.11** Table 2 gives the constraints on an $\alpha$::*c-type* for instantiation in the $\alpha$::*mem-type* axiomatic type class. [MaxSize], [AlignDvdSize] and [AlignField] give some basic size and alignment related properties. The [MaxSize] and [AlignDvdSize] conditions are taken directly from Section 3.2.2 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 Definitions 5.12, 5.13, and 5.15.

**Definition 5.12** 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 5.13** A type description $t$ is well-formed w.r.t. size, wf-size-desc $t$, when every node has a non-zero size.

**Definition 5.14** A field description $d$ and size $n$ are considered *consistent* when the following properties hold:

$$\forall\, v\ bs\ bs'. \quad |bs| = |bs'| \longrightarrow \text{field-update } d\ bs\ (\text{field-update } d\ bs'\ v) = \text{field-update } d\ bs\ v \qquad [\text{FuFu}]$$

$$\forall\, v\ bs.\ |bs| = n \longrightarrow \text{field-update } d\ (\text{field-access } d\ v\ bs)\ v = v \qquad [\text{FuFaId}]$$

$$\forall\, bs\ bs'\ v\ v'. \quad |bs| = n \longrightarrow |bs'| = n \longrightarrow \text{field-access } d\ (\text{field-update } d\ bs\ v)\ bs' = \text{field-access } d\ (\text{field-update } d\ bs\ v')\ bs \qquad [\text{FaFu}]$$

$$\forall\, v\ bs.\ |bs| = n \longrightarrow |\text{field-access } d\ v\ bs| = n \qquad [\text{FaLen}]$$

The properties are similar to those already provided by Isabelle's **record** package at the HOL level and can be established automatically. The restriction to *byte list*s of size $n$ in these conditions is required as field-access and field-update may only be well-defined for certain list lengths.

**Definition 5.15** 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:

$$\begin{aligned}
&\forall\ v\ bs\ bs'.\\
&\quad \mathsf{update\text{-}ti}\ s\ bs\ (\mathsf{update\text{-}ti}\ t\ bs'\ v) = \qquad [\textsc{FuCom}]\\
&\quad \mathsf{update\text{-}ti}\ t\ bs'\ (\mathsf{update\text{-}ti}\ s\ bs\ v)\\
&\forall\ v\ bs\ bs'.\\
&\quad |bs| = \mathsf{size\text{-}td}\ t \longrightarrow \qquad [\textsc{FaFuInd}]\\
&\quad |bs'| = \mathsf{size\text{-}td}\ s \longrightarrow\\
&\quad \mathsf{access\text{-}ti}\ s\ (\mathsf{update\text{-}ti}\ t\ bs\ v)\ bs' =\\
&\quad \mathsf{access\text{-}ti}\ s\ v\ bs'
\end{aligned}$$

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.

**Theorem 5.3** *The $\alpha$::mem-type axioms from this section imply the remaining axioms in Section 3.2.2:*

$$\frac{|bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)}{\mathsf{from\text{-}bytes}\ (\mathsf{to\text{-}bytes}\ x\ bs) = x} \qquad [\textsc{Inv}]$$

$$\frac{|bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)}{|\mathsf{to\text{-}bytes}\ x\ bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)} \qquad [\textsc{Len}]$$

$$0 < \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) \qquad [\textsc{SzNZero}]$$

$$\mathsf{align\text{-}of}\ \mathsf{TYPE}(\alpha)\ \mathsf{dvd}\ |\mathsf{addr}| \qquad [\textsc{Align}]$$

*Proof* For [Inv], we unfold Definition 5.10 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 Definition 5.10 and [FaLen]. [SzNZero] is implied by [WFSizeDesc]. Finally, for [Align], observe align-of TYPE($\alpha$) $<$ |addr| from [AlignDvdSize], hence the alignment as a power-of-two divides |addr|, a larger power-of-two. □

### 5.4 Type Installation

The constraints of the previous section require both the construction of suitable type information and a corresponding $\alpha$::*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 combinators that allow the structure to be built up field-wise and 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. We use this approach and combinators and corresponding proof rules have been derived, but we elide for brevity. Details may be found in [46].

At the same time as type class instantiation, 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 5.7* The following rule for resolving field names beginning with $''z''$ is installed for *x-struct*:

$$\begin{aligned}&\mathsf{lookup}\ \mathsf{TYPE}(\textit{x-struct})_\tau\ (''z''.fs)\ m =\\ &\mathsf{lookup}\ (\mathsf{adjust\text{-}ti}\ \mathsf{TYPE}(\textit{word8})_\tau\ \mathsf{z}\ (\mathsf{z\text{-}update} \circ (\lambda x\ \text{-}.\ x)))\ fs\\ &(m + \mathsf{size\text{-}of}\ \mathsf{TYPE}(\textit{word16}))\end{aligned}$$

Simplifications for size-td and align-td on the entire structure are also installed.

### 5.5 Heap Semantics

The translation of Section 3 remains mostly unchanged with the new type encoding, with the exception of heap-update, which now supplies to-bytes with the existing heap state underneath the target footprint to facilitate padding field semantics:

$$\begin{aligned}&\mathsf{heap\text{-}update}\ p\ v\ h \equiv\\ &\mathsf{heap\text{-}update\text{-}list}\ p_\&\ (\mathsf{to\text{-}bytes}\ v\ (\mathsf{heap\text{-}list}\ h\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha))\ p_\&))\ h\end{aligned}$$

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 guard predicates.

### 5.6 Representation Normalisation

In later sections, 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 $\alpha$ dependency by collapsing leaf field descriptions to *byte list* normalisation functions, resulting in a *typ-uinfo*. 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.

*Example 5.8* Figure 5 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 5.16** Type information is exported with export-uinfo:

$$
\begin{array}{l}
\text{export-uinfo } ti \equiv \\
\text{map-td} \\
\ (\lambda n\ algn\ d\ bs. \\
\quad \text{if } |bs| = n \\
\quad \text{then field-access } d \text{ (field-update } d\ bs \text{ arbitrary) (replicate } n\ 0) \\
\quad \text{else } []) \\
\ ti
\end{array}
$$

We write $\text{TYPE}(\alpha)_v$ for export-uinfo $\text{TYPE}(\alpha)_\tau$.

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.

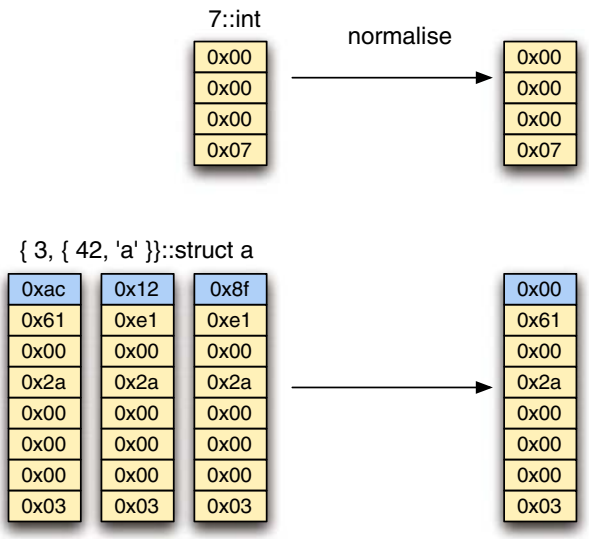



**Fig. 5** Normalisation mapping to *byte list* equivalence classes

**Definition 5.17** Normalisation norm-tu for type information is derived by the sequential composition of leaf normalisation functions. We write norm-bytes TYPE($\alpha$) for norm-tu (export-uinfo TYPE($\alpha$)$_\tau$).

**Theorem 5.4** norm-tu *applied to exported type information is equivalent to normalisation with the access and update functions derived from the type information:*

$$\frac{\text{wf-field-desc } ti \qquad \text{wf-desc } ti \qquad |bs| = \text{size-td } ti}{\text{norm-tu (export-uinfo } ti) \; bs = \text{access-ti}_0 \; ti \; (\text{update-ti } ti \; bs \; \text{arbitrary})}$$

*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 5.5** *Normalisation does not affect the HOL value of a byte list:*

$$\frac{|bs| = \text{size-of TYPE}(\alpha)}{\text{from-bytes (norm-bytes TYPE}(\alpha) \; bs) = \text{from-bytes } bs}$$

*Proof* From definitions, Theorem 5.4 and Definition 5.14 properties. □

**Theorem 5.6** *Field access is equivalent to normalisation of the corresponding fragment of the underlying byte list representation:*

$$\frac{\text{TYPE}(\alpha)_\tau \rhd f = \lfloor (t, n) \rfloor \qquad |bs| = \text{size-of TYPE}(\alpha)}{\begin{array}{l}\text{access-ti}_0 \; t \; (\text{from-bytes } bs) = \\ \text{norm-tu (export-uinfo } t) \; (\text{take (size-td } t) \; (\text{drop } n \; bs))\end{array}}$$

*Proof* By Theorem 5.4 and:

$$\frac{s \rhd f = \lfloor (t, n) \rfloor \qquad |bs| = \text{size-td } s \qquad |bs'| = \text{size-td } t \qquad \text{wf-field-desc } s \qquad \text{wf-desc } s}{\begin{array}{l}\text{access-ti } t \; (\text{update-ti } s \; bs \; v) \; bs' = \\ \text{access-ti } t \; (\text{update-ti } t \; (\text{take (size-td } t) \; (\text{drop } n \; bs)) \; \text{arbitrary}) \; bs'\end{array}}$$

This auxiliary rule can be shown by structural induction on the type information. □

## 6 Unified Memory Model

We now turn to unifying the semantic model in Section 3.2.1 with the abstract view of memory provided by multiple typed heaps. This yields the ability to express the semantics of programs that exhibit inter-type aliasing where needed and have multiple typed heaps as a proof abstraction where the program remains within a type-safe fragment of C, thus avoiding proof obligations arising from inter-type aliasing.

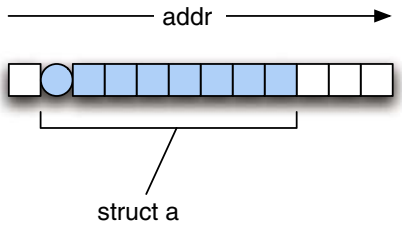



**Fig. 6** Previous heap type description with a valid **struct a** pointer

### 6.1 Heap Type Description

Inside the type-safe fragment of C, where the majority of code remains, there is an implicit mapping between memory locations and types, and heap dereferences respect this mapping. We introduce this mapping as an additional state component, and refer to it as the heap type description. The heap type description is a ghost variable, and as such does not influence the semantics of our programs. Since in C this mapping cannot be extracted from the source code, the program verifier adds proof annotations that update the heap type description.

The heap type description we describe below is more complicated than one would naively expect, so we first give a simpler model from earlier work [47] and observe its limitations as motivation. There we introduced the mapping as:

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

The type *typ-tag* can be thought of as being equivalent to *unit typ-desc*. We wrote $d,g \models_t p$ to mean that the pointer $p$ is valid in heap type description $d$ with guard $g$. The guard $g$ restricts the validity assertion based on the language's pointer dereferencing rules. This is depicted in Fig. 6. In the figure it can be observed that each location in a valid pointer's footprint is mapped to the type's *typ-tag* and the base is distinguished, hence the nested *option* type. This has been shown to be sufficient to recover multiple typed heaps via rewriting for scalar types.

The problem with this notion of the heap type description for structured types 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. $d,g \models_t p \Longrightarrow d,g \models_t \mathsf{Ptr}\ \&(p \rightarrow f)$.

To accomplish this, we introduce an extended 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.[7] The second component is a *typ-slice*, providing an indexed map to the *typ-uinfo*s that may reside at a particular address.

[7] This approach is taken in preference to a partial function to aid in partitioning state in Section 7.

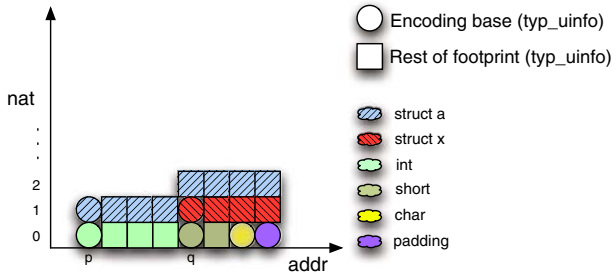



**Fig. 7** Extended heap type description with a valid **struct a** pointer

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.[8]

An example of the extended heap type description is provided in Fig. 7. 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 Example 4.1, 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 $(p,1)$, $(p+1,1),\ldots,(p+3,1)$ 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 $(p+4,2),\ldots,(p+7,2)$, 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.

### *6.1.1 Validity*

Validity for a pointer with respect to the heap type description is a core concept in the following development. So far an informal description of this has been given, here we define it for the heap type description and explore some related properties.

**Definition 6.1** Pointer validity is defined for the heap type description as:

$$
\begin{aligned}
&\mathsf{valid\text{-}footprint}\ d\ x\ t \equiv \\
&\mathsf{let}\ n = \mathsf{size\text{-}td}\ t \\
&\mathsf{in}\ 0 < n \wedge \\
&\quad (\forall\, y{<}n.\ \mathsf{list\text{-}map}\ (\mathsf{typ\text{-}slice}\ t\ y) \subseteq_m \mathsf{snd}\ (d\ (x + \mathbb{N}^{\Rightarrow}\ y)) \wedge \\
&\qquad\quad \mathsf{fst}\ (d\ (x + \mathbb{N}^{\Rightarrow}\ y)))
\end{aligned}
$$

$$
d,g \models_t (p::\alpha::\textit{c-type ptr}) \equiv \mathsf{valid\text{-}footprint}\ d\ p_{\&}\ \mathsf{TYPE}(\alpha)_v \wedge g\ p
$$

[8]This allows consideration of the potential overlap of values of the same type to be eliminated for valid pointers.

where the function list-map::$\alpha$ *list* $\Rightarrow$ (*nat* $\rightharpoonup$ $\alpha$) 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.:

$$\begin{aligned}\text{typ-slice TYPE}(a\text{-}struct)_v\ 4 = [&(\text{TYPE}(word16)_v, \quad \text{True}), \quad (\text{TYPE}(x\text{-}struct)_v, \text{True}),\\ &(\text{TYPE}(a\text{-}struct)_v, \text{False})]\end{aligned}$$

corresponding to (*q*,*0*), (*q*,*1*) and (*q*,*2*) in Fig. 7 respectively.

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.

The use of the map subset operator $\subseteq_m$ in Definition 6.1 provides monotonicity.

**Definition 6.2** Field monotonicity for a guard is defined as:

$$\begin{aligned}&\text{guard-mono } (g::\alpha\ ptr \Rightarrow bool)\ (g'::\beta\ ptr \Rightarrow bool) \equiv\\ &\forall n\ f\ p.\ g\ p \wedge \text{TYPE}(\alpha)_v \rhd f = \lfloor(\text{TYPE}(\beta)_v, n)\rfloor \longrightarrow g'\ (\text{Ptr}\ (p_{\&} + \mathbb{N}^{\Rightarrow}\ n))\end{aligned}$$

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 6.1** *Validity has field monotonicity:*

$$\frac{\text{TYPE}(\alpha)_\tau \rhd f = \lfloor(s, n)\rfloor \qquad \text{export-uinfo } s = \text{TYPE}(\beta)_v \qquad \text{guard-mono } g\ g'}{d,g \models_t p \longrightarrow d,g' \models_t \text{Ptr } \&(p \rightarrow f)}$$

*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($\alpha$), from Lemma 5.1, 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($\alpha$)$_v$ $(y + n)$) $\subseteq_m$ 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. □

*6.1.2 Retyping*

We now give a retyping function ptr-retyp::$\alpha$::*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. This is typically used in a programmer supplied heap type description update annotation. Syntactically, they are C comments of the form:

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

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, providing a form of memory safety on retypes.

**Definition 6.3** A region of memory may be retyped such that $p$::$\alpha$::*c-type ptr* is valid:

$$\begin{aligned}
&\textsf{htd-update-list}\ p\ []\ d \equiv d\\
&\textsf{htd-update-list}\ p\ (x \cdot xs)\ d \equiv \textsf{htd-update-list}\ (p+1)\ xs\\
&\qquad\qquad\qquad\qquad (d(p := (\textsf{True}, \textsf{snd}\ (d\ p)) + + x)))\\
&\textsf{typ-slices}\ \ \textsf{TYPE}(\alpha) \equiv\\
&\textsf{map}(\lambda n.\ \textsf{list-map}\ (\textsf{typ-slice}\ \textsf{TYPE}(\alpha)_v\ n))\ [0..<\textsf{size-of}\ \textsf{TYPE}(\alpha)]\\
&\\
&\textsf{ptr-retyp}\ p \equiv \textsf{htd-update-list}\ p_\& \ (\textsf{typ-slices}\ \textsf{TYPE}(\alpha))
\end{aligned}$$

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 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.

**Lemma 6.2** *Inside the retyped region,* ptr-retyp ($p$::$\alpha$::*mem-type ptr*) *provides the expected heap type description value:*

$$\frac{x \in \{p_\&..+\textsf{size-of}\ \textsf{TYPE}(\alpha)\}}{\begin{aligned}&\textsf{ptr-retyp}\ p\ d\ x =\\ &(\textsf{True}, \textsf{snd}\ (d\ x) + + \textsf{list-map}\ (\textsf{typ-slice}\ \textsf{TYPE}(\alpha)_v\ (\mathbb{N}^{\Leftarrow}\ (x - p_\&))))\end{aligned}}$$

*Proof* By induction on the list. □

**Theorem 6.3** *Following retyping, a target pointer $p$::$\alpha$::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 Lemma 6.2. □

**Theorem 6.4** *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:*

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

*Proof* We have $x \notin \{p_\&..+\text{size-of TYPE}(\alpha)\} \Longrightarrow \text{ptr-retyp } p\ d\ x = d\ x$ by list induction. The rule then follows by unfolding and application of this fact to show each point in the footprint remains unchanged. □

## 6.2 Lifting

So far, the effect of updates on the lifted heap can only be expressed point-wise; 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 typed 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.

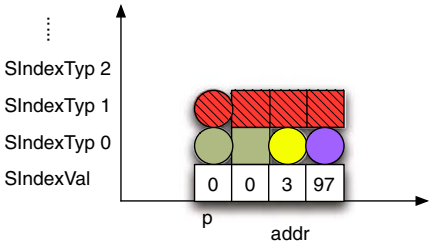
1. The two heap related components are combined into a single function.

**Definition 6.4** The first stage, lift-state, results in an 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. 8, with an *x-struct* footprint. This should be read as with Fig. 7, the vertical axis now the second component of *s-addr* rather

**Fig. 8** Example *heap-state*



than an index. The rationale for this model is based on the requirements of the separation logic embedding and is provided in Section 7.1.

**Definition 6.5** The function lift-state filters out locations that are False or $\bot$ 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.

$$
\begin{array}{l}
\text{lift-state} \equiv \\
\lambda(h,\, d)\ (x,\, y). \\
\quad \text{case } y \text{ of SIndexVal} \Rightarrow \text{if fst } (d\ x) \text{ then } \lfloor \text{SValue } (h\ x) \rfloor \text{ else } \bot \\
\quad \mid \text{SIndexTyp } n \Rightarrow \text{option-case } \bot\ (\text{Some} \circ \text{STyp})\ (\text{snd } (d\ x)\ n)
\end{array}
$$

Lifted validity and heap-list are expressed on *heap-state*s with $d,g \models_s p$ and heap-list-s respectively in the obvious way.

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 type.

**Definition 6.6** The lift-typ-heap function, with the type signature $\alpha$ *ptr-guard* $\Rightarrow$ *heap-state* $\Rightarrow$ ($\alpha$::*c-type ptr* $\rightharpoonup$ $\alpha$), 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:

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

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

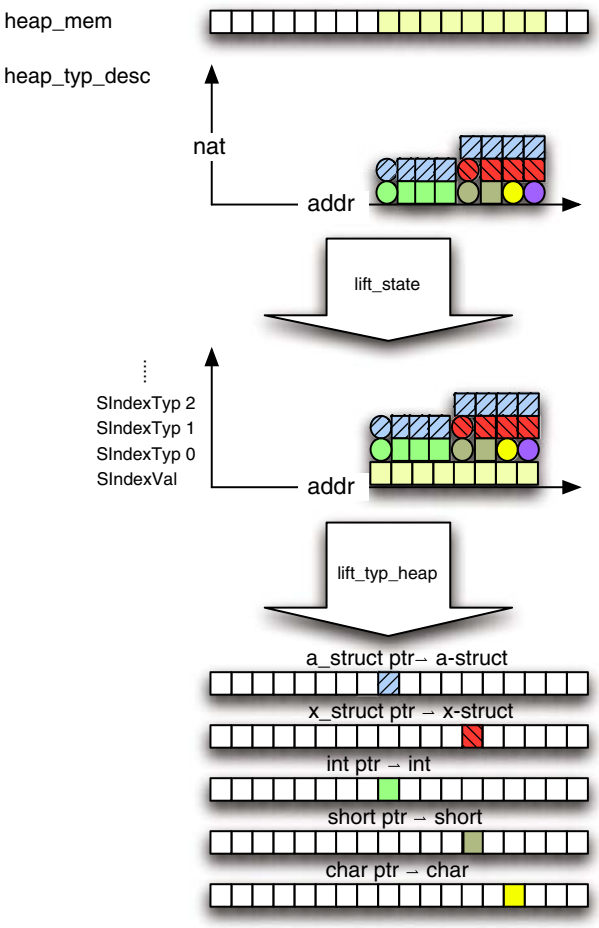
$$
\begin{array}{l}
\text{lift-typ-heap } g\ s \equiv \\
\lambda p.\ \text{if } s,g \models_s p \text{ then } \lfloor \text{from-bytes } (\text{heap-list-s } s\ (\text{size-of TYPE}(\alpha))\ p_\&) \rfloor \\
\quad \text{else } \bot
\end{array}
$$

**Definition 6.7** The two stages, shown in Fig. 9, are combined with $\text{lift}_\tau$:

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

Like lift, $\text{lift}_\tau$ is polymorphic and returns a heap abstraction of type $\alpha$ *typ-heap* = $\alpha$ *ptr* $\rightharpoonup$ $\alpha$. The program text itself can continue to use the functions lift and

**Fig. 9** Two-stage lifting




heap-update, while pre/post conditions and invariants use the stronger $\mathsf{lift}_\tau$ to make more precise statements.

**Theorem 6.5** *An alternative definition of* $\mathsf{lift}_\tau$ *that provides a connection with* lift *is:*

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

*Proof* Expand $\mathsf{lift}_\tau$ and the alternate definition for lift-typ-heap, letting $s =$ lift-state $(h,\ d)$. □

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

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

**Theorem 6.6** *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:*

$$\frac{\mathsf{TYPE}(\alpha)_\tau \rhd f = \lfloor (t, n) \rfloor \quad \mathsf{lift\text{-}typ\text{-}heap}\ g\ s\ p = \lfloor v \rfloor \quad \mathsf{export\text{-}uinfo}\ t = \mathsf{TYPE}(\beta)_\nu \quad \mathsf{guard\text{-}mono}\ g\ g'}{\mathsf{lift\text{-}typ\text{-}heap}\ g'\ s\ (\mathsf{Ptr}\ \&(p \rightarrow f)) = \lfloor \mathsf{from\text{-}bytes}\ (\mathsf{access\text{-}ti}_0\ t\ v) \rfloor}$$

*where* $p$*::*$\alpha$*::mem-type ptr and* $\mathsf{Ptr}\ \&(p \rightarrow f)$*::*$\beta$*::mem-type ptr.*

*Proof* Theorem 6.1 provides field monotonicity for validity. It is left for us to show that $\mathsf{from\text{-}bytes}\ (\mathsf{heap\text{-}list\text{-}s}\ s\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\beta))\ \&(p \rightarrow f)) = \mathsf{from\text{-}bytes}\ (\mathsf{access\text{-}ti}_0\ t\ (\mathsf{from\text{-}bytes}\ (\mathsf{heap\text{-}list\text{-}s}\ s\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha))\ p_\&)))$. This can be achieved with Theorems 5.6 and 5.5. □

**Corollary** *The property in Theorem* 6.6 *also applies with* $\mathsf{lift}_\tau$*:*

$$\frac{\mathsf{TYPE}(\alpha)_\tau \rhd f = \lfloor (t, n) \rfloor \quad \mathsf{lift}_\tau\ g\ s\ p = \lfloor v \rfloor \quad \mathsf{export\text{-}uinfo}\ t = \mathsf{TYPE}(\beta)_\nu \quad \mathsf{guard\text{-}mono}\ g\ g'}{\mathsf{lift}_\tau\ g'\ s\ (\mathsf{Ptr}\ \&(p \rightarrow f)) = \lfloor \mathsf{from\text{-}bytes}\ (\mathsf{access\text{-}ti}_0\ t\ v) \rfloor}$$

## 6.3 Update Dependency Order

At the end of Section 4 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 6.8** An order can be defined on type descriptions that expresses the update dependency between heaps::

$$s \leq t \equiv \exists n.\ (s, n) \in \mathsf{td\text{-}set}\ t$$

This can be lifted to a predicate on $\alpha$*::c-type itself* and $\beta$*::c-type itself*:

$$\mathsf{TYPE}(\alpha) \leq_\tau \mathsf{TYPE}(\beta) \equiv \mathsf{TYPE}(\alpha)_\nu \leq \mathsf{TYPE}(\beta)_\nu$$

*Example 6.1* Using the running example, it can be easily observed that $\mathsf{TYPE}$(*x-struct*) $<_\tau$ $\mathsf{TYPE}$(*a-struct*) and $\mathsf{TYPE}$(*word32*) $<_\tau$ $\mathsf{TYPE}$(*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 6.7** $\leq$ *is a partial order:*

$$s \leq s \qquad \frac{s \leq t \quad t \leq s}{s = t} \qquad \frac{s \leq t \quad t \leq u}{s \leq u}$$

*Proof* Reflexivity is trivial. Antisymmetry can be shown with $(s, n) \in \mathsf{td\text{-}set\text{-}offset}\ t\ m \Longrightarrow size\ s = size\ t \wedge s = t \wedge n = m \vee size\ s < size\ t$, 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. □

## 6.4 Generalised Rewrites

In this section we develop the key rewrites that allow lift terms to be translated to $\mathsf{lift}_\tau$ terms and the effects of updates on lifted typed heaps to be evaluated. First we deal with lift terms with Theorem 6.8 and then present some auxiliary definitions and then the key theorems for heap-updates, Theorems 6.10 and 6.12. These theorems have the form of conditional rewrites, but require some additional support to be efficiently applicable, so are followed by this detail.

**Theorem 6.8** *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 Theorem 6.5 and the definition of lift. □

**Definition 6.9** A list of names of all fields matching an exported type information can be obtained with field-names :: $\alpha$ *typ-info* $\Rightarrow$ *typ-uinfo* $\Rightarrow$ *qualified-field-name list*. E.g. $\mathsf{field\text{-}names}\ \mathsf{TYPE}(a\text{-}struct)_\tau\ \mathsf{TYPE}(word16)_v = [[''c'', ''y'']]$.

**Definition 6.10** From td-set, a predicate may be derived that checks whether a given pointer $p$::$\alpha$ *ptr* is to a field of a structured type with base $q$::$\beta$ *ptr*:

$$\mathsf{field\text{-}of}\ p\ q \equiv (\mathsf{TYPE}(\alpha)_v, \mathbb{N}^{\Leftarrow}\ (p_\& - q_\&)) \in \mathsf{td\text{-}set}\ \mathsf{TYPE}(\beta)_v$$

**Definition 6.11** From lookup, functions may be derived that provide the first and second components of the result for a valid qualified field name:

$$\begin{aligned}\mathsf{field\text{-}typ}\ \mathsf{TYPE}(\alpha)\ n &\equiv \mathsf{fst}\ (\mathsf{the}\ (\mathsf{TYPE}(\alpha)_\tau \triangleright n))\\ \mathsf{field\text{-}offset}\ \mathsf{TYPE}(\alpha)\ n &\equiv \mathsf{snd}\ (\mathsf{the}\ (\mathsf{TYPE}(\alpha)_v \triangleright n))\end{aligned}$$

**Lemma 6.9** *Updates to the heap function at a valid* $\alpha$ *ptr p do not affect* h-val *at distinct valid* $\beta$ *ptr locations q:*

$$\frac{d,g \models_t p \qquad d,g' \models_t q \qquad \neg\ \mathsf{TYPE}(\alpha) <_\tau \mathsf{TYPE}(\beta) \qquad \neg\ \mathsf{field\text{-}of}\ q\ p}{\mathsf{h\text{-}val}\ (\mathsf{heap\text{-}update}\ p\ v\ h)\ q = \mathsf{h\text{-}val}\ h\ q}$$

*Proof* Starting with Lemma 3.1, we can then apply the h-val and heap-update definitions as well as

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

to give a complete proof. □

We now give the rule for $\beta$ heaps when an $\alpha$ update occurs and $\mathsf{TYPE}(\alpha) \leq_\tau \mathsf{TYPE}(\beta)$. The intuition here is that locations that are not the base of valid $\beta$ pointers or where the $\alpha$ *ptr* does not correspond to a field of $\beta$ are unaffected. When the update pointer does correspond to a field of $\beta$, we traverse the $\alpha$ fields of the enclosing $\beta$, 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 6.10** *The lifted $\beta$ heap following an update of a valid $\alpha$ ptr $p$, where $\alpha$ is a sub-type of $\beta$ 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*

$$\begin{array}{l}
\mathsf{super\text{-}field\text{-}update}\ p\ v\ s \equiv \\
\lambda q.\ \mathsf{if}\ \mathsf{field\text{-}of}\ p\ q \\
\quad \mathsf{then\ case}\ s\ q\ \mathsf{of}\ \bot \Rightarrow \bot \\
\qquad \mid \lfloor w \rfloor \Rightarrow \\
\qquad\quad \lfloor \mathsf{update\text{-}value}\ (\mathsf{field\text{-}names}\ \mathsf{TYPE}(\beta)_\tau\ \mathsf{TYPE}(\alpha)_v)\ v\ w \\
\qquad\quad\ (\mathbb{N}^{\Leftarrow}\ (p_\& - q_\&))\rfloor \\
\quad \mathsf{else}\ s\ q
\end{array}$$

$$\begin{array}{lll}
\mathsf{update\text{-}value}\ []\ v\ w\ x & \equiv & w \\
\mathsf{update\text{-}value}\ (f \cdot fs)\ v\ w\ x & \equiv & \mathsf{if}\ x = \mathsf{field\text{-}offset}\ \mathsf{TYPE}(\beta)\ f \\
 & & \quad \mathsf{then\ update\text{-}ti}\ (\mathsf{field\text{-}typ}\ \mathsf{TYPE}(\beta)\ f)(\mathsf{to\text{-}bytes}_0\ v)\ w \\
 & & \quad \mathsf{else\ update\text{-}value}\ fs\ v\ w\ x
\end{array}$$

*Proof* Equality of the two heaps can be shown with extensionality and unfolding of super-field-update, letting the pointer be called $q$. Expand $\mathsf{lift}_\tau$ terms with Theorem 6.5 and it is easy to see that locations without valid $\beta$ *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 $\mathsf{lift}_\tau$ 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:

$$\frac{t \triangleright f = \lfloor (s, n) \rfloor \quad \text{wf-field-desc } t \quad \text{wf-desc } t \quad |bs| = \text{size-td } t \quad |v| = \text{size-td } s}{\begin{array}{c}\text{update-ti } t \ (\text{take } n \ bs \ @ \ v \ @ \ \text{drop } (n + |v|) \ bs) \ w = \\ \text{update-ti } s \ v \ (\text{update-ti } t \ bs \ w)\end{array}}$$

obtained by structural induction and list fragment reasoning. On the RHS, the term 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 $\text{TYPE}(\alpha)_v = \text{TYPE}(\beta)_v$. If the types are the same then the treatment is similar to Lemma 6.9. Otherwise, we can show

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

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 Lemma 3.1. □

While Theorem 6.10 gives a conditional rewrite that allows an update to be lifted to the typed heap level of Section 6.2, 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 6.11** *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:*

$$\frac{\text{TYPE}(\beta)_\tau \triangleright f = \lfloor (s, n) \rfloor \qquad \text{lift}_\tau\ g\ h\ p = \lfloor w \rfloor \qquad \text{TYPE}(\alpha)_v = \text{export-uinfo } s}{\begin{array}{c}\text{super-field-update } (\text{Ptr } \&(p\rightarrow f))\ v\ (\text{lift}_\tau\ g\ h) = \\ \text{lift}_\tau\ g\ h(p \mapsto \text{update-ti } s\ (\text{to-bytes}_0\ v)\ w)\end{array}}$$

*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 Theorem 6.10. 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 Section 5.4, 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 6.2* 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 $\beta$ heap when $\alpha$ has multiple fields of type $\beta$. The heap update function sub-field-update takes a list of all such fields and applies an update at each.

**Theorem 6.12** *The lifted $\beta$ heap following an update of a valid $\alpha$ ptr $p$, where $\alpha$ is not a sub-type of $\beta$ is given by:*

$$\frac{d,g' \models_t p \qquad \neg\ \mathsf{TYPE}(\alpha) <_\tau \mathsf{TYPE}(\beta)}{\begin{array}{c}\mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d) =\\ \mathsf{sub\text{-}field\text{-}update}\ (\mathsf{field\text{-}names}\ \mathsf{TYPE}(\alpha)_\tau\ \mathsf{TYPE}(\beta)_v)\ p\ v\ (\mathsf{lift}_\tau\ g\ (h,\ d))\end{array}}$$

*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:

$$\frac{\begin{array}{c}d,g' \models_t p\\ \neg\ \mathsf{TYPE}(\alpha) <_\tau \mathsf{TYPE}(\beta) \qquad \mathsf{set}\ fs \subseteq \mathsf{set}\ (\mathsf{field\text{-}names}\ \mathsf{TYPE}(\alpha)_\tau\ \mathsf{TYPE}(\beta)_v)\\ K = \mathcal{U} - (\mathsf{field\text{-}ptrs}\ p\ (\mathsf{field\text{-}names}\ \mathsf{TYPE}(\alpha)_\tau\ \mathsf{TYPE}(\beta)_v) - \mathsf{field\text{-}ptrs}\ p\ fs)\end{array}}{\mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d)\restriction_K = \mathsf{sub\text{-}field\text{-}update}\ fs\ p\ v\ (\mathsf{lift}_\tau\ g\ (h,\ d))\restriction_K}$$

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 $\beta$ *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 Lemma 6.9 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:

$$\frac{\mathsf{TYPE}(\alpha)_\tau \rhd f = \lfloor (s, n) \rfloor \qquad bs = \mathsf{to\text{-}bytes}\ v\ (\mathsf{heap\text{-}list}\ h\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha))\ p_\&)}{\mathsf{access\text{-}ti}_0\ s\ v = \mathsf{norm\text{-}tu}\ (\mathsf{export\text{-}uinfo}\ s)\ (\mathsf{take}\ (\mathsf{size\text{-}td}\ s)\ (\mathsf{drop}\ n\ bs))}$$

  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 $\mathsf{lift}_\tau$ is at the typed level, after applying from-bytes, Theorem 5.5 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 Theorem 6.11 is not as easy to state, as the $\beta$ heap will be updated at multiple locations.

## 6.5 Non-interference

The results of the previous section can be specialised to obtain a set of conditional rewrites that provide independently typed heaps when the update and heap types are not related by the update dependency order. When both types are the same, an update can be lifted to a simple function update.

**Theorem 6.13** *The rewrites for an update to a lifted typed heap through a valid pointer of the same type, or a disjoint type are:*

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

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

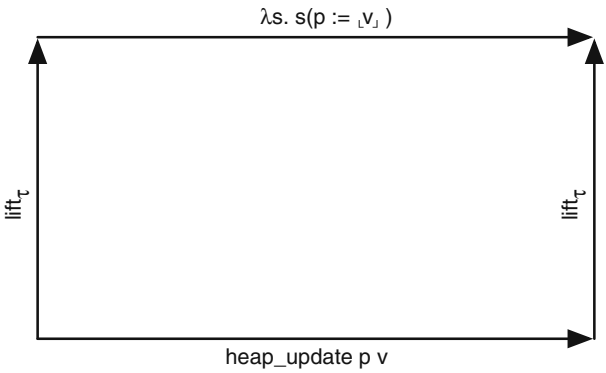



**Fig. 10** Lifted heap updates when the heap is of the same type

*Proof* By Theorem 6.12 and reducing the field-names term with field-names *ti* (export-uinfo *ti*) = [[]] and TYPE($\alpha$)$_\nu$ $\perp_t$ TYPE($\beta$)$_\nu$ $\Longrightarrow$ field-names TYPE($\beta$)$_\tau$ TYPE($\alpha$)$_\nu$ = [], respectively. □

The connection between levels and updates is illustrated in Figs. 10 and 11.

Bornat [7] 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 &($p$→$f$) operator, first-class **struct** types, pointer casts and address arithmetic. However, the following can be shown:

**Theorem 6.14** *When the base pointers are of the same type $\beta$, and neither of the field names is a prefix of the other, updates through an $\alpha$ pointer derived from one field do not affect a value in the $\gamma$ lifted heap at the other:*

$$\frac{\begin{array}{c} d,g' \models_t p \quad d,g'' \models_t q \qquad \mathsf{TYPE}(\beta)_\tau \rhd f = \lfloor (s, m) \rfloor \qquad \mathsf{TYPE}(\beta)_\tau \rhd f' = \lfloor (t, n) \rfloor \\ \mathsf{size\text{-}td}\ s = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) \quad \mathsf{size\text{-}td}\ t = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\gamma) \quad \neg f \leq f' \qquad \neg f' \leq f \end{array}}{\begin{array}{c} \mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ (\mathsf{Ptr}\ \&(p{\rightarrow}f))\ v\ h,\ d)\ (\mathsf{Ptr}\ \&(q{\rightarrow}f')) = \\ \mathsf{lift}_\tau\ g\ (h,\ d)\ (\mathsf{Ptr}\ \&(q{\rightarrow}f')) \end{array}}$$

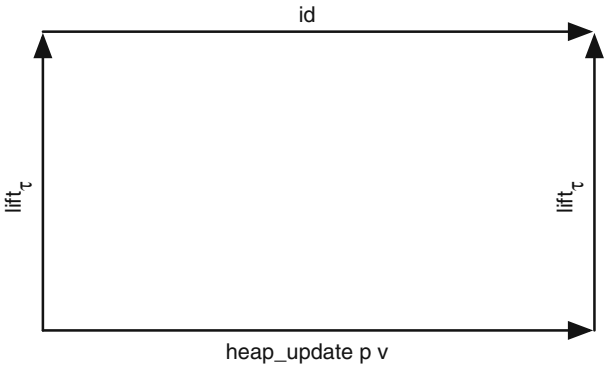



**Fig. 11** Lifted heap updates when the heap is of a disjoint type

*Proof* Unfold definitions and then use Lemma 3.1. 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. □

## 7 Structured Separation Logic

We now have a proof technique that tames inter-type aliasing for C pointer programs that feature structured types. However, the problem of *intra-type aliasing* remains. Separation logic provides an approach in which anti-aliasing information, needed to reason about intra-type aliasing, 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 section we describe a shallow embedding of separation logic based on the unified memory model with specific support for structured types. 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.

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 are mostly standard. 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. The section concludes with an approach to proof obligation lifting to the separation logic level, analogous to the connection made in Sections 6.4 and 6.5.

### 7.1 Domain

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 Section 6.2. For example, a loop invariant with the separation assertion $P$ is written $\{\!|\ P\ (\mathsf{lift\text{-}state}\ \mathcal{H})\ |\!\}$, which we abbreviate as $\{\!|P^{sep}|\!\}$.

The rationale for the specific choice of domain is that it allows for more expressive separation assertions than are possible with simpler models. A first attempt at defining the domain might herald $addr \rightharpoonup typ\text{-}uinfo\ list \times byte$. 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. Spatial separation can be based on a more sophisticated domain than just memory addresses, in our case we can conceive of also owning type information levels at memory locations. 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 7.1* 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

typ-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 typ-outline $p$ separate to (Ptr $(\&(p\rightarrow[''y''])) \mapsto 3$) and hence reason about the y field independently.

## 7.2 Shallow Embedding

We give the definitions of the core separation logic assertions and connectives in our embedding below.

**Definition 7.1** As in the development of Reynolds [38] there is an empty *heap-state* predicate:

$$\square = (\lambda s.\ s = \mathsf{empty})$$

The definition of the singleton heap assertion is more involved in our embedding as we need to consider the pointer's footprint in the intermediate heap state in order to restrict the domain.

**Definition 7.2** The s-footprint::$\alpha$::*c-type ptr* $\Rightarrow$ *s-addr set* gives a set of addresses inside a pointer's *heap-state* footprint:

$$\begin{array}{l}\mathsf{s\text{-}footprint\text{-}untyped}\ p\ t \equiv \\ \{(p + \mathbb{N}^{\Rightarrow}\ x,\ \mathsf{SIndexVal}) \mid x < \mathsf{size\text{-}td}\ t\} \cup \\ \{(p + \mathbb{N}^{\Rightarrow}\ x,\ \mathsf{SIndexTyp}\ n) \mid x < \mathsf{size\text{-}td}\ t \wedge n < |\mathsf{typ\text{-}slice}\ t\ x|\}\end{array}$$

$$\mathsf{s\text{-}footprint}\ (p::\alpha\ ptr) \equiv \mathsf{s\text{-}footprint\text{-}untyped}\ p_{\&}\ \mathsf{TYPE}(\alpha)_v$$

**Definition 7.3** $p \mapsto_g v$ asserts that the heap contains exactly one mapping matching the guard $g$, at the location given by pointer $p$ to value $v$:

$$\begin{array}{l}p \mapsto_g v \equiv \\ \lambda s.\ \mathsf{lift\text{-}typ\text{-}heap}\ g\ s\ p = \lfloor v \rfloor \wedge \mathsf{dom}\ s = \mathsf{s\text{-}footprint}\ p \wedge \mathsf{wf\text{-}heap\text{-}val}\ s\end{array}$$

The guard is an addition to the usual $p \mapsto v$ and serves the same purpose as in Definition 6.1, i.e. strengthening the assertion to aid in discharging guard proof obligations and thereby making the treatment of guards in the framework generic. 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 7.4** We introduce a new predicate that captures validity at the separation logic level:

$$g \vdash_s p \equiv \lambda s.\ s,g \models_s p \wedge \mathsf{dom}\ s = \mathsf{s\text{-}footprint}\ p$$

**Definition 7.5** There are two significant separation connectives, conjunction and implication:

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

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 Section 7.6.

Some additional mapping assertions are common:

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

## 7.3 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 [38]. 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.

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.

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* [38]. 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.

### 7.4 Frame Rule

The separation frame rule [50] 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:

$$\frac{\vdash \{\!|P^{sep}|\!\}\ c\ \{\!|Q^{sep}|\!\}}{\vdash \{\!|(P \wedge^* R)^{sep}|\!\}\ c\ \{\!|(Q \wedge^* R)^{sep}|\!\}}$$

Unfortunately, such a general rule cannot be expressed in a shallow embedding since:

- The state-space type is program dependent. This is a less serious problem as we can make further use of type classes to capture all state spaces.
- $c$ 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 $R = \acute{p} \mapsto v$, the triple below does not:

  $$\{\!|\ (\square \wedge^* (\acute{p} \mapsto v))^{sep}\ |\!\}\ *p = 0;\ \{\!|\ (\square \wedge^* (\acute{p} \mapsto v))^{sep}\ |\!\}$$

  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. In C we have a much weaker notion of memory safety and only require that heap accesses occur in some region of memory, e.g. the heap or stack.

It is however possible to prove this rule for specific programs and state-spaces. To achieve this, when the frame rule needs to be applied, we requires programs to provide a stronger form of memory safety based on the heap type description. 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 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.

We use the following predicate to capture the above restrictions in guards on heap updates and in /∗∗ *AUXUPD: ...* ∗/ annotations.

**Definition 7.6** A pointer $p$::$\alpha$::*c-type ptr* is *safe* w.r.t. a heap type description $d$ if its footprint is a subset of the domain of $d$:

$$\begin{aligned}&\mathsf{ptr\text{-}safe}\ p\ d \equiv\\ &\mathsf{s\text{-}footprint}\ p\\ &\subseteq \{(x, \mathsf{SIndexVal}) \mid \mathsf{fst}\ (d\ x)\} \cup \{(x, \mathsf{SIndexTyp}\ n) \mid \mathsf{snd}\ (d\ x)\ n \neq \bot\}\end{aligned}$$

Subject to some involved technical considerations detailed in [46], the frame rule can then be expressed for programs $c$ with a state space in $\alpha$::*heap-state-type* with a rule similar to:

$$\frac{\vdash \{\!|P^{sep}|\!\}\ c\ \{\!|Q^{sep}|\!\} \qquad \mathsf{mem\text{-}safe}\ c}{\vdash \{\!|(P \wedge^{*} R)^{sep}|\!\}\ c\ \{\!|(Q \wedge^{*} R)^{sep}|\!\}}$$

The mem-safe $c$ assumption asserts the above memory safety requirement is met for the whole program, and can be discharged automatically through rewriting for programs featuring the appropriate ptr-safe guards on heap and heap type description updates.

## 7.5 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 Theorem 6.6 hints at this being possible with Definition 7.3, and in this section we provide the rules to accomplish this.

Example 7.1 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 7.7** The concept of the singleton state is useful in the following:

$$\begin{array}{l}\mathsf{singleton}\ p\ v \equiv \\ \mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p\ v\ (\lambda x.\ 0),\ \mathsf{ptr\text{-}retyp}\ p\ (\lambda x.\ (\mathsf{False}, \mathsf{empty})))\end{array}$$

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

**Lemma 7.1** *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. Lemma 6.2 gives the existence of mappings inside the footprint, and the absence of mappings outside can be shown with a complementary lemma derivable from definitions. □

**Definition 7.8** 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 7.9** The footprint for a set of qualified field names for a type $\alpha$, 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 7.2** *The footprint for a subset of valid qualified field names for a type $\alpha$ 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 7.10** $p \mapsto_g^F v$ 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:

$$
\begin{aligned}
&p \mapsto_g^F v \equiv \\
&\lambda s.\ \mathsf{lift\text{-}typ\text{-}heap}\ g\ (\mathsf{singleton}\ p\ v\ +\!\!+\ s)\ p = \lfloor v \rfloor \wedge \\
&\quad F \subseteq \mathsf{fields}\ \mathsf{TYPE}(\alpha) \wedge \\
&\quad \mathsf{dom}\ s = \mathsf{s\text{-}footprint}\ p - \mathsf{fs\text{-}footprint}\ p\ F \wedge \mathsf{wf\text{-}heap\text{-}val}\ s
\end{aligned}
$$

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 7.2* Figure 12 gives a state where $p \mapsto_g^{\{[''c'',\ ''y'']\}} v$ holds.

**Theorem 7.3** *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 $\mathsf{fs\text{-}footprint}\ p\ F = \emptyset$. $s$ then covers the domain of $\mathsf{singleton}\ p\ v$ as a result of Lemma 7.1, giving $\mathsf{singleton}\ p\ v\ +\!\!+\ s = s$. □

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 7.3* The $[''c'',\ ''y'']$ field can be independently extracted when $[''b'']$ is masked, but not if $[''c'']$ is masked.

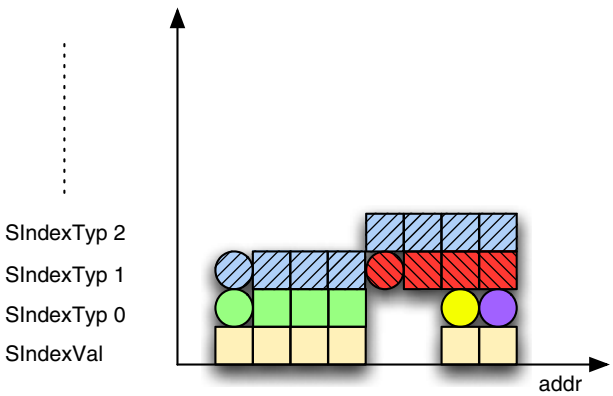



**Fig. 12** Example *heap-state* for a masked mapping assertion

**Definition 7.11** A qualified field name is said to be disjoint from a set of qualified field names with:

$$\mathsf{disjoint\text{-}fn}\ f\ F \equiv \forall f' \in F.\ \neg f \leq f' \wedge \neg f' \leq f$$

where $\leq$ is the list prefix order.

**Lemma 7.4** *A field disjoint from a set of valid qualified field names has a footprint disjoint from the set's footprint:*

$$\frac{\mathsf{disjoint\text{-}fn}\ f\ F \qquad F \subseteq \mathsf{fields}\ \mathsf{TYPE}(\alpha) \qquad \mathsf{TYPE}(\alpha)_\tau \rhd f = \lfloor (t, n) \rfloor}{\mathsf{fs\text{-}footprint}\ p\ F \cap \mathsf{fs\text{-}footprint}\ p\ \{f\} = \emptyset}$$

*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 7.5** *A valid disjoint field may be unfolded from a masked mapping assertion with:*

$$\frac{\begin{array}{c}\mathsf{TYPE}(\alpha)_\tau \rhd f = \lfloor (t, n) \rfloor \\ \mathsf{disjoint\text{-}fn}\ f\ F \qquad \mathsf{guard\text{-}mono}\ g\ g' \qquad \mathsf{export\text{-}uinfo}\ t = \mathsf{TYPE}(\beta)_\nu\end{array}}{p \mapsto_g{}^{F}\ v = p \mapsto_g{}^{(\{f\} \cup F)}\ v \wedge^* \mathsf{Ptr}\ \&(p{\rightarrow}f) \mapsto_g{}'\ \mathsf{from\text{-}bytes}\ (\mathsf{access\text{-}ti}_0\ t\ v)}$$

*where* $p$*::*$\alpha$*::mem-type ptr and* 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:

$$\frac{\begin{array}{c}\mathsf{TYPE}(\alpha)_\tau \rhd f = \lfloor (t, n) \rfloor \\ \mathsf{disjoint\text{-}fn}\ f\ F \qquad \mathsf{guard\text{-}mono}\ g\ g' \qquad \mathsf{export\text{-}uinfo}\ t = \mathsf{TYPE}(\beta)_\nu \qquad (p \mapsto_g{}^{F}\ v)\ s\end{array}}{(\mathsf{Ptr}\ \&(p{\rightarrow}f) \mapsto_g{}'\ \mathsf{from\text{-}bytes}\ (\mathsf{access\text{-}ti}_0\ t\ v))\ (s\restriction_{\mathsf{fs\text{-}footprint}\ p\ \{f\}})}$$

which is a result of Theorem 6.6 and $\mathsf{fs\text{-}footprint}\ p\ \{f\} \subseteq \mathsf{dom}\ s$, from Lemma 7.2 and Lemma 7.4. The masked mapping assertion on the RHS, with $f$ now included in the set of masked fields, then holds on the remaining state:

$$\frac{(p \mapsto_g{}^{F}\ v)\ s \qquad \mathsf{TYPE}(\alpha)_\tau \rhd f = \lfloor (t, n) \rfloor}{(p \mapsto_g{}^{(\{f\} \cup F)}\ v)\ (s\restriction_{(\mathsf{dom}\ s\ -\ \mathsf{fs\text{-}footprint}\ p\ \{f\})})}$$

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 $[\![s,g \models_s p;\ t,g' \models_s p]\!] \Longrightarrow s\ +\!\!+\ t\restriction_X, g \models_s p$. 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 Theorem 6.10, 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:

$$\frac{\mathsf{TYPE}(\alpha)_\tau \rhd f = \lfloor (t, n) \rfloor}{\mathsf{heap\text{-}list\text{-}s}\ (\mathsf{singleton}\ p\ v\restriction_{\mathsf{fs\text{-}footprint}\ p\ \{f\}})\ (\mathsf{size\text{-}td}\ t)\ \&(p \rightarrow f) = \mathsf{access\text{-}ti}_0\ t\ v}$$

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:*

$$\frac{\mathsf{TYPE}(\alpha)_\tau \rhd f = \lfloor (d, n) \rfloor \qquad \mathsf{export\text{-}uinfo}\ d = \mathsf{TYPE}(\beta)_v \qquad \mathsf{guard\text{-}mono}\ g\ g' \qquad (p \mapsto_g v)\ s}{(\mathsf{Ptr}\ \&(p \rightarrow f) \hookrightarrow_{g'} \mathsf{from\text{-}bytes}\ (\mathsf{access\text{-}ti}_0\ d\ v))\ s}$$

*where* $p$::$\alpha$::*mem-type ptr and* $\mathsf{Ptr}\ \&(p \rightarrow f)$::$\beta$::*mem-type ptr.*

**Theorem 7.6** *A valid disjoint field may be folded into a masked mapping assertion with:*

$$\frac{\begin{array}{c}\mathsf{TYPE}(\alpha)_\tau \rhd f = \lfloor (t,\, n) \rfloor \\ f \in F \quad \mathsf{disjoint\text{-}fn}\ f\ (F - \{f\}) \quad \mathsf{guard\text{-}mono}\ g\ g' \quad \mathsf{export\text{-}uinfo}\ t = \mathsf{TYPE}(\beta)_v\end{array}}{p \mapsto_g{}^F\, v \wedge^* \mathsf{Ptr}\ \&(p \rightarrow f) \mapsto_{g'} w = p \mapsto_g{}^{(F - \{f\})}\, \mathsf{update\text{-}ti}\ t\ (\mathsf{to\text{-}bytes}_0\ w)\ v}$$

*where* $p$::$\alpha$::*mem-type ptr and* $\mathsf{Ptr}\ \&(p \rightarrow f)$::$\beta$::*mem-type ptr.*

*Proof* Theorem 7.5 can be applied to the RHS. The two singleton mapping assertions for $f$ then cancel out, leaving us to establish:

$$\frac{\begin{array}{c}\mathsf{TYPE}(\alpha)_\tau \rhd f = \lfloor (t,\, n) \rfloor \\ f \in F \quad \mathsf{disjoint\text{-}fn}\ f\ (F - \{f\}) \quad \mathsf{guard\text{-}mono}\ g\ g' \quad \mathsf{export\text{-}uinfo}\ t = \mathsf{TYPE}(\beta)_v\end{array}}{p \mapsto_g{}^F\, v = p \mapsto_g{}^F\, \mathsf{update\text{-}ti}\ t\ (\mathsf{to\text{-}bytes}_0\ w)\ v}$$

The $v$ value on the RHS can be expanded as $\mathsf{update\text{-}ti}\ t\ (\mathsf{to\text{-}bytes}_0\ (\mathsf{from\text{-}bytes}\ (\mathsf{access\text{-}ti}_0\ t\ v)))\ v$ 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 7.4* The $\mathsf{y}$ field of an *x-struct* can be unfolded as:

$$\begin{array}{r}p \mapsto_{\mathsf{ptr\text{-}aligned}}\ (|y = 3,\ \mathsf{z} = 65|) = \mathsf{Ptr}\ \&\ (p \rightarrow [''y'']) \mapsto_{\mathsf{ptr\text{-}aligned}}\ 3 \wedge^* \\ p \mapsto_{\mathsf{ptr\text{-}aligned}}{}^{\{[''y'']\}} (|y = 3,\ \mathsf{z} = 65|)\end{array}$$

Later, after an update to $\mathsf{y}$ setting it to the value 1, it can be folded back to the structured value to give the expected update:

$$\begin{array}{l}\mathsf{Ptr}\ \&\ (p \rightarrow [''y'']) \mapsto_{\mathsf{ptr\text{-}aligned}}\ 1 \wedge^* = p \mapsto_{\mathsf{ptr\text{-}aligned}} (|y = 1,\ \mathsf{z} = 65|) \\ p \mapsto_{\mathsf{ptr\text{-}aligned}}{}^{\{[''y'']\}} (|y = 3,\ \mathsf{z} = 65|)\end{array}$$

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.

## 7.6 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 Section 6.4 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.

**Theorem 7.7** lift *and* heap-update *are connected to separation mapping assertions through the following rules:*

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

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

$$\frac{(g \vdash_s p \wedge^* (p \mapsto_g v \longrightarrow^* P))\ (\textsf{lift-state}\ (h, d))}{P\ (\textsf{lift-state}\ (\textsf{heap-update}\ p\ v\ h, d))}$$

$$\frac{(g \vdash_s p \wedge^* R)\ (\textsf{lift-state}\ (h, d))}{(p \mapsto_g v \wedge^* R)\ (\textsf{lift-state}\ (\textsf{heap-update}\ p\ v\ h, d))}$$

*Proof* The lift rules can essentially be derived using definitions and Theorem 6.8.

For the heap-update rules, we partition the heap into two disjoint subheaps with the separation conjunction—the subheap with footprint matching $p$ and the rest of the heap. The latter is preserved by the heap-update and the former component is transformed such that $p \mapsto_g v$ holds. □

These rules are analogous to the backwards and global reasoning Hoare logic mutation rules [38]. The latter provides a weakest pre-condition style rule that will

match any separation assertion, while the former may be used on goal assertions that can be manipulated into the matching form.

**Theorem 7.8** *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:*

$$\frac{(p \mapsto_g u \wedge^* R)\ (\textsf{lift-state}\ (h, d)) \qquad \textsf{TYPE}(\beta)_\tau \rhd f = \lfloor (t, n) \rfloor \quad \textsf{export-uinfo}\ t = \textsf{TYPE}(\alpha)_v \qquad w = \textsf{update-ti}\ t\ (\textsf{to-bytes}_0\ v)\ u}{(p \mapsto_g w \wedge^* R)\ (\textsf{lift-state}\ (\textsf{heap-update}\ (\textsf{Ptr}\ \&(p\rightarrow f))\ v\ h, d))}$$

*Proof* Convert to a masked mapping assertion with Theorem 7.3. Unfold $f$ with Theorem 7.5. Apply the heap-update rule of Theorem 7.7 and fold $f$ back in with Theorem 7.6. The proof is complete by returning to a singleton mapping assertion with Theorem 7.3 again. At various points it is necessary to simplify with the field description consistency conditions. □

Theorem 7.8 can be applied in goals in similar situations to Theorem 6.10 and Theorem 6.11.

## 8 Case Study: In-Place List Reversal

We now look at a case study in the application of the proof rules for separation logic. This is the standard in-place list reversal example from the literature, which features a linked inductively-defined data structure, abstraction predicate in specifications,

**Table 3** **reverse_struct** specification and definition

$$\forall zs.\ \vdash\ \{\!|(\textsf{list}\ zs\ \acute{}ptr)^{sep}|\!\}\ \ \acute{}reverse\text{-}struct\text{-}ret :== reverse\text{-}struct(\acute{}ptr)\ \ \{\!|(\textsf{list}\ (\textsf{rev}\ zs)\ \acute{}reverse\text{-}struct\text{-}ret)^{sep}|\!\}$$

```
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;
}
```

iteration, and pointer updates, i.e. the features that create non-trivial aliasing conditions. Mehta and Nipkow [26] mechanise the same example in their more abstract setting.

The source code and specification are provided in Table 3. Here a **struct** type is used to represent nodes. The specification contains a list abstraction predicate, defined:

$$\begin{aligned}
&\mathsf{list}\ []\ i \quad\ \equiv \lambda s.\ i = \mathsf{NULL}\ \wedge\ \square\, s\\
&\mathsf{list}(x \cdot xs)\ i \equiv \lambda s.i \neq \mathsf{NULL}\ \wedge\\
&\qquad\qquad (\exists j.\ \mathsf{item}\ j = x\ \wedge\ (i \mapsto_g j \wedge^*\ \mathsf{list}\ xs\ (\mathsf{next}\ j))\ s)
\end{aligned}$$

**Theorem 8.1** ***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:

$$\{\!|\exists xs\ ys.\ (\mathsf{list}\ xs\ \acute{p}tr \wedge^*\ \mathsf{list}\ ys\ \acute{l}ast)^{sep} \wedge \mathsf{rev}\ zs = \mathsf{rev}\ xs\ @\ ys|\!\}$$

The $Pre \Longrightarrow Inv$ and $Inv \Rightarrow Post$ conditions are trivial. After existential instantiation and simplification, the loop invariant preservation proof requires we show:

$$\begin{aligned}
&1.\ \bigwedge zs\ a\ b\ last\ ptr\ ys\ list\ j.\\
&\quad [\![ptr \neq \mathsf{NULL};\ \mathsf{rev}\ zs = \mathsf{rev}\ list\ @\ \mathsf{item}\ j{\cdot}ys;\\
&\quad (ptr \mapsto_g j \wedge^*\ \mathsf{list}\ list\ (\mathsf{next}\ j) \wedge^*\ \mathsf{list}\ ys\ last)\\
&\quad (\mathsf{lift\text{-}state}\ (a, b))]\!]\\
&\quad \Longrightarrow (ptr \mapsto_g j(\!|\mathsf{next} := last|\!) \wedge^*\\
&\qquad \mathsf{list}\ ys\ last \wedge^*\ \mathsf{list}\ list\ (\mathsf{lift}\ a\ (\mathsf{Ptr}\ \&(ptr{\rightarrow}[''next''])))\\
&\qquad (\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ (\mathsf{Ptr}\ \&(ptr{\rightarrow}[''next'']))\ last\ a,\ b))
\end{aligned}$$

This follows from Theorem 7.8. The first side-condition may be discharged with Theorem 7.7 and Theorem 7.5, eliminating the lift. The other side-conditions are discharged by rewriting, using the rules of Section 5.4. □

An interesting point in the proof is when we have to show:

$$\begin{aligned}
&1.\ \bigwedge zs\ a\ b\ last\ ptr\ ys\ list\ j.\\
&\quad [\![ptr \neq \mathsf{NULL};\ \mathsf{rev}\ zs = \mathsf{rev}\ list\ @\ \mathsf{item}\ j{\cdot}ys;\\
&\quad (ptr \mapsto_g j \wedge^*\ \mathsf{list}\ list\ (\mathsf{next}\ j) \wedge^*\ \mathsf{list}\ ys\ last)\\
&\quad (\mathsf{lift\text{-}state}\ (a, b))]\!]\\
&\quad \Longrightarrow j(\!|\mathsf{next} := last|\!) = \mathsf{update\text{-}ti}\\
&\qquad\qquad (\mathsf{adjust\text{-}ti}\ \mathsf{TYPE}(node\ ptr)_\tau\ \mathsf{next}\\
&\qquad\qquad\ (\mathsf{next\text{-}update} \circ (\lambda x\ \text{-}.\ x)))\\
&\qquad\qquad (\mathsf{to\text{-}bytes}_0\ last)\ j
\end{aligned}$$

Here, applying the reverse definition of from-bytes and the $\alpha$::*mem-type* axioms lifts the RHS to the HOL **record** level to simplify for the goal.

Compared to an earlier in-place list reversal example [47], the proof script was about the same structure and size, 67 lines. We can then see that for this example, the sophisticated machinery of the previous sections does not unduly burden a verification that remains in the type-safe fragment of C.

## 9 Case Study: L4 Kernel Memory Allocator

The previous case study provided a taste for the proof process, but the verification was relatively simple. In this section 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 [24] 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.

The kernel implementation we target is L4Ka::Pistachio [44]. Pistachio is a mostly C++ implementation, with a small amount of architecture and platform dependent assembler, of the L4 X.2 API [22] and has been ported to many architectures (x86, ARM, Alpha, MIPS, Itanium, PowerPC) without sacrificing high performance. While Pistachio is implemented in C++, no essential use of C++ features is made apart from using classes to structure the code and we are able to extract out what amounts to C for this study.

A brief overview of this case study has previously appeared in [47]. In the following we provide a more detailed treatment, built on the developments in this paper, which, while being similar in use to the earlier model for this example, have the additional support for structured types.

### 9.1 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 KMC, 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.

### 9.2 Implementation Data Structure and Code

Figure 13 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 size KMC. A single global variable word_t *kfree_list provides a pointer to the start of the

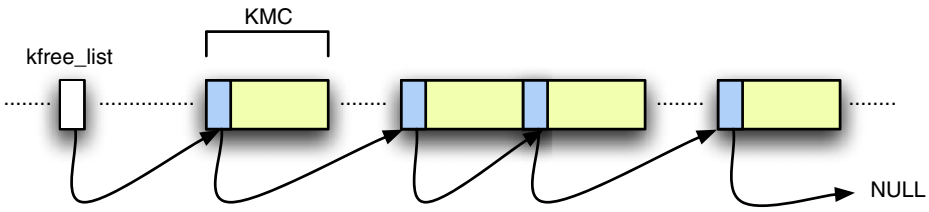



**Fig. 13** Management data structure of the L4 memory allocator

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.

**Table 4** **alloc** definition

```
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 5** **init** definition

```
void init(void *start, void *end)
{
    kfree_list = 0;

    free(start, (word_t)end - (word_t)start);
}
```

The C source code for **alloc**, **init** and **free** is given in Tables 4, 5 and 6 respectively. 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. No structured types feature in the source, the aspects of the models related to bridging the gap between semantics for unsafe code and abstract models for safe code are instead stressed by the verification.

## 9.3 Specifications

We now give in detail the separation logic specification for **alloc**. We refer the reader to [46] for the typed heaps treatment and other function specifications, skipping them here for brevity. The main difference between the two is in the amount of detail related to aliasing invariants and frame properties, with the separation logic specification superior on both counts.

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:

$$\mathsf{alloc}\ p\ n\ F \quad \equiv\ F - \mathsf{chunks}\ p\ (p_{\&} + (n - \mathsf{KMC}))$$

**Table 6** **free** 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;
}
```

The function chunks $q\ y$ in the definition above refers to a set of locations starting with pointer $q$, ending with address $y$, that consists of base addresses of adjacent memory chunks:

$$\begin{array}{l}\mathsf{chunks}\ q\ y \equiv \\ \{x \mid q_{\&} \leq x \wedge x \leq y \wedge (\exists n{\geq}0.\ \mathbb{Z}^{\Leftarrow}\ x = \mathbb{Z}^{\Leftarrow}\ q_{\&} + n * \mathbb{Z}^{\Leftarrow}\ \mathsf{KMC})\}\end{array}$$

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 $\mathbb{Z}$.

The separation logic version of the abstraction predicate is the following:

$$\begin{array}{lcl}\mathsf{list}\ p\ r\ [] & \equiv & \lambda s.\ p = r\ \wedge\ \square\ s \\ \mathsf{list}\ p\ r\ (x \cdot xs) & \equiv & \lambda s.\ (p\ =\ \mathsf{Ptr}\ x\ \wedge\ p\ \neq\ r)\ \wedge \\ & & \quad (\exists q.\ (\mathsf{block}\ p\ q\ \wedge^{*}\ \mathsf{list}\ q\ r\ xs)\ s) \\ \mathsf{sep\text{-}cut}\ x\ y & \equiv & \lambda s.\ \mathsf{dom}\ s\ =\ \{(x, y) \mid x \in \{p..+n\}\} \\ \mathsf{block}\ p\ q & \equiv & \lambda s.\ \mathsf{KMC\ udvd}\ p_{\&} \wedge (p \hookrightarrow q_{\&})\ s\ \wedge \mathsf{sep\text{-}cut}\ p_{\&}\ \mathsf{KMC}\ s \\ \mathsf{free\text{-}set}\ p\ q\ F & \equiv & \lambda s.\ \exists xs.\ \mathsf{list}\ p\ q\ xs\ s\ \wedge\ F = \mathsf{set}\ xs \\ \mathsf{free\text{-}set\text{-}h}\ p\ r\ F & \equiv & \lambda s.\ \exists q.\ (\mathsf{ptr\text{-}coerce}\ p\ \mapsto\ q_{\&}\ \wedge^{*}\ \mathsf{free\text{-}set}\ q\ r\ F)\ s\end{array}$$

In addition to performing data abstraction, free-set asserts ownership over the entire footprint of each chunk through the block predicate.

The separation logic specification of **alloc** is then given by:

$$\begin{array}{l}\forall F\ \sigma.\ \vdash \{\!|\sigma.\ (\mathsf{free\text{-}set\text{-}h\ kfree\text{-}list\text{-}addr\ NULL}\ F)^{sep} \wedge \\ \qquad\qquad \mathsf{KMC\ udvd\ max}\ \acute{s}ize\ \mathsf{KMC}|\!\} \\ \qquad \acute{a}lloc\text{-}ret :== alloc(\acute{s}ize) \\ \qquad \{\!|(\acute{a}lloc\text{-}ret \neq \mathsf{NULL} \longrightarrow \\ \qquad\quad \mathsf{size\text{-}aligned}\ \acute{a}lloc\text{-}ret\ (\mathsf{max}\ ^{\sigma}size\ \mathsf{KMC}) \wedge \\ \qquad\quad \acute{a}lloc\text{-}ret_{\&} \leq \acute{a}lloc\text{-}ret_{\&} + \mathsf{max}\ ^{\sigma}size\ \mathsf{KMC} - \mathsf{KMC} \wedge \\ \qquad\quad \mathsf{chunks}\ \acute{a}lloc\text{-}ret \\ \qquad\quad (\acute{a}lloc\text{-}ret_{\&} + (\mathsf{max}\ ^{\sigma}size\ \mathsf{KMC} - \mathsf{KMC})) \\ \qquad\quad \subseteq F \wedge \\ \qquad\quad (\mathsf{free\text{-}set\text{-}h\ kfree\text{-}list\text{-}addr\ NULL} \\ \qquad\qquad (\mathsf{alloc}\ \acute{a}lloc\text{-}ret\ (\mathsf{max}\ ^{\sigma}size\ \mathsf{KMC})\ F) \wedge^{*} \\ \qquad\qquad \mathsf{zero}\ \acute{a}lloc\text{-}ret\ (\mathsf{max}\ ^{\sigma}size\ \mathsf{KMC}))^{sep}) \wedge \\ \qquad\quad (\acute{a}lloc\text{-}ret = \mathsf{NULL} \longrightarrow \\ \qquad\quad (\mathsf{free\text{-}set\text{-}h\ kfree\text{-}list\text{-}addr\ NULL}\ F)^{sep})|\!\}\end{array}$$

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.

In the post-condition there are two cases. If we have run out of memory, $\acute{a}lloc\text{-}ret = \mathsf{NULL}$, we 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, $\acute{a}lloc\text{-}ret \neq \mathsf{NULL}$, we state that the new set of free memory chunks is the same as that given by evaluating

the abstract function alloc. Additionally, we explicitly say that the memory returned is a separate, contiguous block of the right size, filled with zero words:

$$\begin{array}{lcl}
\mathsf{zero}\ p & \equiv & \mathsf{zero\text{-}block}\ (\mathsf{ptr\text{-}coerce}\ p)\ (\mathbb{N}^{\Leftarrow}\ (n\ \mathsf{div}\ 4)) \\
\mathsf{zero\text{-}block}\ p\ 0 & \equiv & \square \\
\mathsf{zero\text{-}block}\ p\ (\mathit{Suc}\ n) & \equiv & (p +_p\ \mathbb{N}^{\Rightarrow})\ n)\ \mapsto\ 0\ \wedge^*\ \mathsf{zero\text{-}block}\ p\ n
\end{array}$$

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 returned memory is guaranteed to be aligned to the effective request size, if a power-of-two, by:

$$\mathsf{size\text{-}aligned}\ p\ n \equiv (\exists k.\ n = 2^k) \longrightarrow n\ \mathsf{udvd}\ p_{\&}$$

## 9.4 Invariants

In this section we describe the loop invariants that were used to structure the verification. These provide the key proof steps and insight into how the allocator works. We again focus our attention on **alloc** here and present the separation logic invariants.

The outer loop invariant is:

$$\begin{array}{l}
\{\!|(\exists G\ H.\ (\acute{p}rev = \mathsf{ptr\text{-}coerce\ kfree\text{-}list\text{-}addr} \vee \acute{p}rev_{\&} \in G) \wedge \\
\qquad (\mathsf{free\text{-}set\text{-}h\ kfree\text{-}list\text{-}addr}\ \acute{c}urr\ G \wedge^* \\
\qquad\ \mathsf{free\text{-}set}\ \acute{c}urr\ \mathsf{NULL}\ H)^{sep} \wedge \\
\qquad F = G \cup H) \wedge \\
(\mathsf{free\text{-}set\text{-}h\ kfree\text{-}list\text{-}addr\ NULL}\ F)^{sep} \wedge \\
\mathcal{H} = {}^{\sigma}\mathcal{H} \wedge \\
\acute{s}ize = \mathsf{max}\ {}^{\sigma}size\ \mathsf{KMC} \wedge \mathsf{KMC\ udvd}\ \acute{s}ize \wedge (\acute{p}rev \hookrightarrow\ \acute{c}urr_{\&})^{sep}|\!\}
\end{array}$$

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:

$$\begin{array}{l}
\{\!|\acute{c}urr_{\&} \leq \acute{c}urr_{\&} + (\acute{i} - 1) * \mathsf{KMC} \wedge \\
(\exists G\ H.\ (\acute{p}rev = \mathsf{ptr\text{-}coerce\ kfree\text{-}list\text{-}addr} \vee \acute{p}rev_{\&} \in G) \wedge \\
\qquad (\mathsf{free\text{-}set\text{-}h\ kfree\text{-}list\text{-}addr}\ \acute{c}urr\ G \wedge^* \\
\qquad\ \mathsf{free\text{-}set}\ \acute{c}urr\ \acute{t}mp \\
\qquad\quad (\mathsf{chunks}\ \acute{c}urr\ (\acute{c}urr_{\&} + (\acute{i} - 1) * \mathsf{KMC})) \wedge^* \\
\qquad\ \mathsf{free\text{-}set}\ \acute{t}mp\ \mathsf{NULL}\ H)^{sep} \wedge \\
\qquad F = G \cup \mathsf{chunks}\ \acute{c}urr\ (\acute{c}urr_{\&} + (\acute{i} - 1) * \mathsf{KMC}) \cup H) \wedge \\
(\mathsf{free\text{-}set\text{-}h\ kfree\text{-}list\text{-}addr\ NULL}\ F)^{sep} \wedge \\
\mathcal{H} = {}^{\sigma}\mathcal{H} \wedge \\
\acute{s}ize = \mathsf{max}\ {}^{\sigma}size\ \mathsf{KMC} \wedge \\
\mathsf{KMC\ udvd}\ \acute{s}ize \wedge \\
1 \leq \acute{i} \wedge \\
\acute{i} \leq \acute{s}ize\ \mathsf{div\ KMC} \wedge \\
\acute{c}urr \neq \mathsf{NULL} \wedge \mathsf{size\text{-}aligned}\ \acute{c}urr\ \acute{s}ize \wedge (\acute{p}rev \hookrightarrow\ \acute{c}urr_{\&})^{sep}|\!\}
\end{array}$$

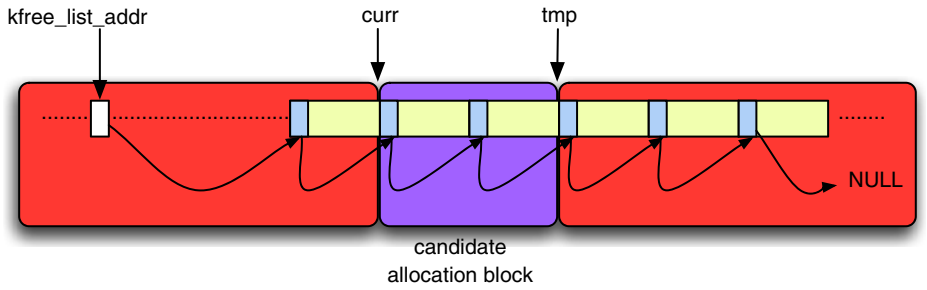



**Fig. 14** Partition of free list

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. 14. 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.

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:

$$
\begin{aligned}
&\{\acute{curr}_{\&} \leq \acute{curr}_{\&} + (\acute{size} - \mathsf{KMC}) \wedge \\
&\ (\text{free-set-h kfree-list-addr NULL} \\
&\quad (\text{alloc (ptr-coerce } \acute{curr}) \text{ (max } {}^{\sigma}\acute{size}\ \mathsf{KMC})\ F) \wedge^{*} \\
&\quad \text{sep-cut } (\acute{curr}_{\&} + \acute{i} * 4)\ (\acute{size} - \acute{i} * 4) \wedge^{*} \\
&\quad \text{zero-block } \acute{curr}\ (\mathbb{N}^{\Leftarrow}\ \acute{i}))^{sep} \wedge \\
&\ \text{chunks } \acute{curr}\ (\acute{curr}_{\&} + (\text{max } {}^{\sigma}\acute{size}\ \mathsf{KMC} - \mathsf{KMC})) \subseteq F \wedge \\
&\ \acute{size} = \text{max } {}^{\sigma}\acute{size}\ \mathsf{KMC} \wedge \\
&\ \mathsf{KMC\ udvd}\ \acute{size} \wedge \\
&\ \mathsf{KMC\ udvd}\ \acute{curr}_{\&} \wedge \\
&\ \acute{i} \leq \acute{size}\ \text{div}\ 4 \wedge \text{size-aligned}\ \acute{curr}\ \acute{size} \wedge \acute{curr} \neq \mathsf{NULL}\}
\end{aligned}
$$

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 $(\acute{curr} +_p \hat{\imath})$ annotation in the body. When the loop condition fails, i.e. $\neg$ i < (size / **sizeof**(word_t)), the post-condition is implied.

## 9.5 Results

With the invariants in place, the proof obligations post-VCG can be discharged. For this study we wrote mostly tactic-style proofs. Table 7 lists the size of each proof

**Table 7** Proof script sizes

| Theory | LoP |
|---|---|
| Shared (all) | 1,400 |
| 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) |

script. For the specific code verification theories, in parenthesis we give the size of the main proof.

This case study corresponds to 136 lines of code in the original Pistachio source and 62 lines of code after configuring and preprocessing.[9] 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 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.

The separation logic proofs tended to be lengthier and more verbose, but not more difficult. 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, requiring several person months simultaneously with the original separation logic embedding, 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.

[9] We only count function body sizes here.

## 10 Related Work

The earliest work in pointer program verification focused on type-safe or typeless languages and most work at this time was through pen-and-paper formalisation. Cartwright and Oppen [11], Morris [28], Bijlsma [5] and Burstall [8] are examples of this. More recently, Bornat [7] 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 [26] in Isabelle/HOL. The Caduceus tool [13] also uses the Burstall-Bornat model and Moy [29] extends this to cope with some well-behaved cases of unions and type casts. Cock [12] has recently developed support for tagged unions and bitfields with predictable layout in our framework, utilising a generative approach for both the C code and lifting theorems.

Leroy and Blazy [23] 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 paper, including the modelling 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 [32] and Hohmuth et al. [17] provide mechanised C/C++ semantics in HOL and PVS respectively, which include low-level memory models, and provide the basis for our approach in Section 3.2.2. Our HOL type encoding has similarities to Blume's [6] 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 [37, 38], O'Hearn [33], Yang [50], Ishtiaq [18] and Calcagno [10]. This has since been mechanised for simple languages in Isabelle/HOL by Weber [48], Preoteasa [36] in PVS based on a predicate transformer semantics and Marti et al. [25] in Coq for a version of C without dealing with its types. Tuch et al. [47] gave the first treatment of separation logic that unified the byte-level and logical views of memory in Isabelle/HOL. Appel and Blazy [2] later gave a mechanised separation logic for a C intermediate language in Coq with the strict standard's memory view.

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 [42] include SLAM [3] and BLAST [16], which have had success in checking safety properties such as correct API use in device drivers. Similarly, Hallem et al. [14] use static analyses to find bugs in system code. More sophisticated abstract domains are used in shape analyses [27, 40], which can show some structural invariants, such as the absence of loops in linked lists. Separation logic decision procedures [4] 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 [9].

Most closely related to our case study in Section 9 is the successful verification of the kernel memory allocator from the teaching-oriented Topsy operating system by

Marti et al. [25] 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.

## 11 Conclusion

In this paper we developed mechanised proof techniques for C, built on a low-level view of underlying memory and capable of managing the aliasing and frame problems. We have extended the existing models for reasoning about pointer programs to cope fully with C's type system, including support for structured types, and provided solutions to the technical issues that arise when reasoning about C programs in higher-order logic. In addition, two case studies were given that demonstrate the utility of the derived proof rules.

In addition to the verification proofs discussed in the case studies, the aspects of the C verification framework related to the memory model and proof abstractions came to a total of 18,728 lines of proof (including some initial explorations without support for structured types).

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 [41], 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. It is hoped that the implementation in this paper will in turn provide a basis for further layering and allow later research to reap the benefits of these models.

Future work includes providing support for well-behaved unsafe operations, e.g. struct pointer casting in the case of physical subtyping, development of Isabelle tactics for separation logic proofs and integration with automated tools and decision procedures. While it has been shown that verification proofs for C systems code are practical in our framework, routine application will demand that the more mundane aspects of the verification proofs have greater automation. An interesting use of the framework presented in this paper, along these lines, might be to develop a sound theory for algorithmic techniques, even when not used directly in a theorem prover environment—presently most papers in this area either make liberal use of axiomatised theories and/or "pen and paper" proofs.

**Acknowledgements** We thank Gerwin Klein for discussions and for reading drafts of this paper. The design and implementation of the C subset and translation represents joint work with Michael Norrish and Gerwin Klein.

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

## References

1. Programming languages—C. Technical report 9899:TC2, ISO/IEC JTC1/SC22/WG14 (2005)
2. Appel, A.W., Blazy, S.: Separation logic for small-step Cminor. In: Schneider, K., Brandt, J. (eds.) Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2007). Lecture Notes in Computer Science, vol. 4732, pp. 5–21. Springer, New York (2007)
3. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) Proceedings of the 8th International SPIN Workshop on Model Checking Software. Lecture Notes in Computer Science, vol. 2057, pp. 103–122. Springer, New York (2001)
4. Berdine, J., Calcagno, C., O'Hearn, P.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004). Lecture Notes in Computer Science, vol. 3328, pp. 97–109. Springer, New York (2004)
5. Bijlsma, A.: Calculating with pointers. Sci. Comput. Program **12**(3), 191–205 (1989)
6. Blume, M.: No-longer-foreign: teaching an ML compiler to speak C "natively". Electron. Notes Theoret. Comput. Sci. **59**(1), 36–52 (2001)
7. Bornat, R.: Proving pointer programs in Hoare logic. In: Backhouse, R.C., Oliveira, J.N. (eds.) Proceedings of the 5th International Conference on Mathematics of Program Construction (MPC 2000). Lecture Notes in Computer Science, vol. 1837, pp. 102–126. Springer, New York (2000)
8. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 7, pp. 23–50. Edinburgh University Press, Edinburgh (1972)
9. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Beyond reachability: shape abstraction in the presence of pointer arithmetic. In: Yi, K. (ed.) Proceedings of the 13th International Symposium on Static Analysis (SAS 2006). Lecture Notes in Computer Science, vol. 4134, pp. 182–203. Springer, New York (2006)
10. Calcagno, C., Yang, H., O'Hearn, P.W.: Computability and complexity results for a spatial assertion language for data structures. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2001). Lecture Notes in Computer Science, vol. 2245, pp. 108–119. Springer, New York (2001)
11. Cartwright, R., Oppen, D.C.: The logic of aliasing. Technical report STAN-CS-79-740, Stanford University, Stanford (1979)
12. Cock, D.: Bitfields and tagged unions in C: verification through automatic generation. In: Beckert, B., Klein, G. (eds.) Proceedings of the 5th International Verification Workshop in connection with IJCAR 2008, vol. 372 of CEUR Workshop Proceedings (2008). CEUR-WS.org
13. Filliâtre, J.-C., Marché, C.: Multi-prover verification of C programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) Proceedings of the 6th International Conference on Formal Methods and Software Engineering (ICFEM 2004). Lecture Notes in Computer Science, vol. 3308, pp. 15–29. Springer, New York (2004)
14. Hallem, S., Chelf, B., Xie, Y., Engler, D.R.: A system and language for building system-specific, static analyses. In: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation. SIGPLAN Notices, vol. 37, pp. 69–82. ACM, New York (2002)
15. Harrison, J.: A HOL theory of Euclidean space. In: Hurd, J., Melham, T.F. (eds.) Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005). Lecture Notes in Computer Science, vol. 3603, pp. 114–129. Springer, New York (2005)
16. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) Proceedings of the 10th International SPIN Workshop on Model Checking Software (SPIN 2003). Lecture Notes in Computer Science, vol. 2648, pp. 235–239. Springer, New York (2003)

17. Hohmuth, M., Tews, H., Stephens, S.G.: Applying source-code verification to a microkernel—the VFiasco project. Technical report TUD-FI02-03-März, TU Dresden (2002)
18. Ishtiaq, S.S., O'Hearn, P.W.: 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). SIGPLAN Notices, vol. 36, pp. 14–26. ACM, New York (2001)
19. Kernighan, B.W., Ritchie, D.M.: The C Programming Language, 2nd edn. Prentice-Hall, Englewood Cliffs, New Jersey (1988)
20. Klein, G.: Verified java bytecode verification. Ph.D. thesis, Institut für Informatik, Technische Universität München (2003)
21. Kroening, D.: Application specific higher order logic theorem proving. In: Autexier, S., Mantel, H. (eds.) Proceedings of the 2nd Verification Workshop (VERIFY 2002), pp. 5–15, Technical Report no. 2002/07. DIKU (2002)
22. L4Ka Team. L4 eXperimental kernel reference manual version X.2. University of Karlsruhe. http://l4ka.org/projects/version4/l4-x2.pdf (2001). Oct 2001
23. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. J. Autom. Reason. **41**(1), 1–31 (2008)
24. Liedtke, J.: On $\mu$-kernel construction. In: Proceedings of the Fifteenth ACM Symposium on Operating System Principles (SOSP 95). Operating System Review, vol. 29, pp. 267–284. ACM, New York (1995)
25. Marti, N., Affeldt, R., Yonezawa, A.: 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), pp. 61–72. Charleston, South Carolina (2006)
26. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Inf. Comput. **199**(1–2), 200–227 (2005)
27. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation. SIGPLAN Notices, vol. 37, pp. 221–231. ACM, New York (2001)
28. Morris, J.M.: A general axiom of assignment. In: Broy, M., Schmidt, G. (eds.) Theoretical Foundations of Programming Methodology (Proceedings of the 1981 Maktoberdorf Summer School), pp. 25–51 (1982)
29. Moy, Y.: Union and cast in deductive verification. In: C/C++ Verification Workshop, pp. 1–16. Technical report ICIS-R07015. Radboud University Nijmegen (2007)
30. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (eds.) Proceedings of the 11th International Conference on Compiler Construction (CC 2002). Lecture Notes in Computer Science, vol. 2304, pp. 213–228. Springer, New York (2002)
31. Nipkow, T.: Term rewriting and beyond—theorem proving in Isabelle. Form. Asp. Comput. **1**(4), 320–338 (1989)
32. Norrish, M.: C formalised in HOL. Ph.D. thesis, Computer Laboratory, University of Cambridge (1998)
33. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) Proceedings of the 15th International Workshop on Computer Science Logic (CSL 2001). Lecture Notes in Computer Science, vol. 2142, pages 1–19. Springer, New York (2001)
34. Oheimb: D.v.: Information flow control revisited: noninfluence = noninterference + nonleakage. In: Samarati, P., Ryan, P.Y.A., Gollmann, D., Molva, R. (eds.) Proceedings of the 9th European Symposium on Research in Computer Security (ESORICS 2004). Lecture Notes in Computer Science, vol. 3193, pages 225–243. Springer, New York (2004)
35. Paulson, L.C.: The foundation of a generic theorem prover. J. Autom. Reason. **5**(3), 363–397 (1989)
36. Preoteasa, V.: Mechanical verification of recursive procedures manipulating pointers using separation logic. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) Proceedings of the 14th International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 4085, pp. 508–523. Springer, New York (2006)
37. Reynolds, J.C.: Intuitionistic reasoning about shared mutable data structures. In: Davies, J., Roscoe, B., Woodcock, J. (eds.) Millenial Perspectives in Computer Science, pp. 303–321. Palgrave, Houndsmill, Hampshire (2000)

38. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002), pp. 55–74. IEEE Computer Society (2002)
39. Ritchie, D.M.: The development of the C language. In: Proceedings of the ACM History of Programming Languages Conference (HOPL-II). SIGPLAN Notices, vol. 28, pp. 201–208. ACM, New York (1993)
40. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1999), pp. 105–118. ACM, New York (1999)
41. Schirmer, N.: Verification of sequential imperative programs in Isabelle/HOL. Ph.D. thesis, Technische Universität München (2006)
42. Schlich, B., Kowalewski, S.: Model checking C source code for embedded systems. In: Margaria, T., Steffen, B., Hinchey, M.G. (eds.) Proceedings of the IEEE/NASA Workshop Leveraging Applications of Formal Methods, Verification, and Validation (IEEE/NASA ISoLA 2005), pp. 65–77. NASA, Maryland, USA (2005). NASA/CP-2005-212788
43. Shapiro, J.: Programming language challenges in systems codes: why systems programmers still use C, and what to do about it. In: Probst, C.W. (ed.) Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems (PLOS 2006), pp. 9. ACM, New York (2006)
44. System Architecture Group. The L4Ka::Pistachio microkernel. White paper, University of Karlsruhe, May 2003. http://l4ka.org/projects/pistachio/pistachio-whitepaper.pdf
45. Tews, H.: Verifying Duff's device: a simple compositional denotational semantics for goto and computed jumps (2004). http://www.cs.ru.nl/~tews/Goto/goto.pdf
46. Tuch, H.: Formal memory models for verifying C systems code. Ph.D. thesis, University of NSW (2008)
47. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: Hofmann, M., Felleisen, M. (eds.) Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of
Programming Languages (POPL 2007), pp. 97–108. ACM, New York (2007)
48. Weber, T.: Towards mechanized program verification with separation logic. In: Marcinkowski, J., Tarlecki, A. (eds.) Proceedings of the 18th International Workshop on Computer Science Logic (CSL 2004). Lecture Notes in Computer Science, vol. 3210, pp. 250–264. Springer, New York (2004)
49. Wenzel, M.: Type classes and overloading in higher-order logic. In: Gunter, E.L., Felty, A.P. (eds.) Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 97). Lecture Notes in Computer Science, vol. 1275, pp. 307–322. Springer, New York (1997)
50. Yang, H., O'Hearn, P.W.: A semantic basis for local reasoning. In: Nielsen, M., Engberg, U. (eds.) Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2002). Lecture Notes in Computer Science, vol. 2303, pp. 402–416. Springer, New York (2002)