Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 217 (2008) 41–59
www.elsevier.com/locate/entcs

Structured Types and Separation Logic

Harvey Tuch

Sydney Research Lab., National ICT Australia, Australia[1]
School of Computer Science and Engineering, UNSW, Sydney, Australia
harvey.tuch@nicta.com.au

Abstract

Structured types, such as C's arrays and `structs`, present additional challenges in pointer program verification. The conventional proof abstractions, multiple independent typed heaps and separation logic, which in previous work have been built on a low-level memory model for C and shown to be sound, are not directly applicable in verifications. This is 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. In this paper we present a generalisation of our earlier formal memory model that captured the low-level features of C's pointers and memory and formed the basis for an expressive implementation of separation logic, with new features providing explicit support for C's structured types. We implement this framework in the theorem prover Isabelle/HOL and all proofs are machine checked.

Keywords: Separation Logic, C, Interactive Theorem Proving

1 Introduction

Programs featuring pointers are more difficult to verify than programs without indirection, largely as a result of the *aliasing* problem [1]. For example, consider a program with two pointer variables `float * p` and `int * q` and the following triple:

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

We are unable to ascertain the value pointed to by p as it may refer to the same location as q. With type-safe languages, this form of aliasing, which we call *inter-type aliasing*, can be ignored in proofs if the abstraction of multiple-typed heaps is used, where we have a semantic model with a heap function variable for each language type, e.g. $\mathsf{float\text{-}heap} :: \mathit{float\ ptr} \rightharpoonup \mathit{float}$, $\mathsf{int\text{-}heap} :: \mathit{int\ ptr} \rightharpoonup \mathit{int}$. Unfortunately in C we do not have this luxury as language features such as pointer arithmetic and casting break any illusion of type-safety, and we are forced to adopt the programmer's

[1] National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

doi:10.1016/j.entcs.2008.06.041

model of the heap as a function *addr* ⇒ *byte*, in particular when we wish to verify systems code exploiting compiler and architecture dependent language features.

A key observation is that while C permits code that violates memory and type safety, most code does remain within a type-safe fragment, and in earlier work we reconciled the *multiple-typed heaps* proof abstraction and this low-level view of memory, providing a rewriting approach to lifting proof states from byte granularity maps to typed heaps [13]. This avoided inter-type aliasing considerations where possible and gave a unified framework for proofs that needed to consider code that violated type-safety. The framework included a C parsing tool that emitted a mixed deep-shallow embedding in Schirmer's Hoare logic verification environment [11].

There still remains the problem of *intra-type aliasing* however, where pointers of the same type may alias one another. Again, it is possible to provide explicit conditions on states stating the presence or absence of aliasing, but this becomes rather cumbersome for inductively-defined data structures [1,10]. In particular, the *frame problem* limits the scalability of verifications. A potential solution is the *separation logic* of O'Hearn, Reynolds and others [5,10], providing a language for specifications and inference rules that both concisely allows for the expression of aliasing conditions in assertions and ensures modularity of specifications. In other previous work [14], we provided a shallow embedding of separation logic in Isabelle/HOL, building on the multiple-typed heaps development, resulting in a framework capable of accommodating different proof techniques to address aliasing.

In this paper, we extend this framework to further support C's structured types:

- We provide details of a deep embedding of structure type information capable of handling C's size, alignment, and padding restrictions as well as semantics for heap dereferencing for structured types.
- Earlier rewrites and proof rules for multiple-typed heaps and separation logic are generalised in such a way that they benefit from mechanisation and are still usable in verifications with little new overhead.
- Aspects of structured types that were previously handled in our semantics through shallow translation by trusted ML code are able to be promoted to the HOL level.

2 C `structs`

In our C-HOL type encoding, each C type was given a unique type in the theorem prover. All such types belonged to an axiomatic type class α::*c-type* in Isabelle, which introduced constants that connected the low-level byte representation and the HOL values:

$$\mathsf{to\text{-}bytes} :: \alpha{::}\textit{c-type} \Rightarrow \textit{byte list} \qquad \mathsf{from\text{-}bytes} :: \textit{byte list} \rightharpoonup \alpha{::}\textit{c-type}$$
$$\mathsf{typ\text{-}tag} :: \alpha{::}\textit{c-type itself} \Rightarrow \textit{typ-tag} \qquad \mathsf{typ\text{-}info} :: \alpha{::}\textit{c-type itself} \Rightarrow \textit{typ-info}$$

The functions to-bytes and from-bytes converted between Isabelle values and lists of bytes suitable for writing to or reading from the raw heap state. The function typ-tag associated a unique *type tag* with each α::*c-type*, providing a means of treating language types as first-class values in HOL. Finally, typ-info allowed size and

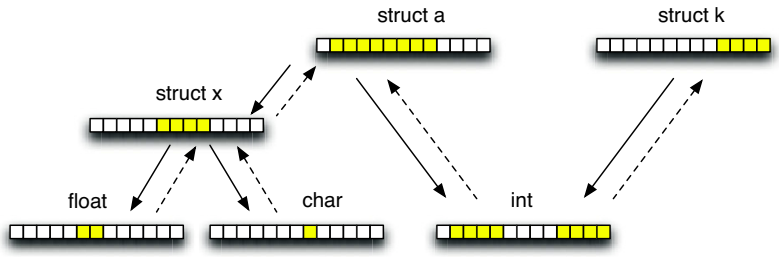

Fig. 1. Heap update dependencies.

alignment information for the type to be calculated.

A distinct Isabelle pointer type for each Isabelle type, used to model C pointer types, was defined with:

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

The phantom α on the left-hand side was used to associate the pointer type information with pointer values in Isabelle's type system.

Primitive types such as `char` and `long *` could be defined in a library for each architecture/compiler in the expected way. `struct` types could be modelled at the HOL level with Isabelle **record** types. Trusted ML code in the C parser provided the following for each structured type used in a program:

- A corresponding **record** declaration
- Definitions of functions appearing in α::*c-type*, requiring full structure information to appear shallowly at the HOL level.
- Lvalue calculations, requiring the full structure information inside the ML parser, as well as offset/size/alignment calculations.

Example 2.1 As a running example, consider the following `struct` declarations:

```
struct x {          struct a {
  float y;            int b;
  char z;             struct x c;
};                  };
```

The following triple demonstrates the most significant limitation with the earlier memory model:

$$\{\!|\ *p = (\!|\ y = 2.1,\ z = 'm'\ |\!)\ |\!\}\ p{\rightarrow}y = 1.2;\ \{\!|\ *p = ?\ |\!\}$$

The problem here is that even though the update and dereference are type-safe, and we do not need to consider aliasing, the proof rules we had developed so far considered this to be type-unsafe, as any region of memory could only have a single type, and p and $\&(p{\rightarrow}y)$ share a common address despite having different but related types. There is a similar problem for the effect of updates through `struct` references on enclosed field pointer values.

Fig. 1 demonstrates how this problem manifests itself in the multiple-typed heaps abstraction. It is no longer the case that updates to heaps can be treated independently, i.e.:

- Updating a field type's heap *may* affect typed heaps of enclosing `structs`.
- Updating a `struct` affects typed heaps of field types (fields-of-fields, etc.).
- Update effects are no longer simple function update, they involve potentially multiple field updates and accesses.

The solution we propose in this paper is to treat structured type information as a first-class value in HOL and develop generalised definitions, rewrites and rules making use of this. We treat structured types as first-class C types in the following to provide the benefits of abstraction and typing in proofs, even though at a semantic level they can be considered in terms of their members. Arrays in the heap decay to the corresponding pointer arithmetic, and inside structured values are also modelled using the definitions in §4.1. `unions` are treated differently, decaying to casts and *byte lists* as value representations.

3 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 α, β, etc. The notation $t :: \tau$ means that HOL term t has HOL type τ. The option type

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

adjoins a new element $\bot$ to a type α. We use α *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 Some constructor has an underspecified inverse called 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 $f\ (x := \mathsf{Some}\ y)$. Domain restriction is $f\restriction_A$ where $f{::}\alpha \rightharpoonup \beta$ and $(f\restriction_A)\ x = (\mathsf{if}\ x \in A\ \mathsf{then}\ f\ x\ \mathsf{else}\ \bot)$.

Finite integers are represented by the type α *word* where α determines the word length. For succinctness, we use abbreviations like *word*8 and *word*32. The functions unat and $\mathsf{of\text{-}nat}$ convert to and from natural numbers (with u for *unsigned*).

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

Isabelle supports axiomatic type classes [16] similar to, but more restrictive than Haskell's. The notation $\alpha{::}\mathit{ring}$ restricts the type variable α 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 τ can be shown to belong to a type class given a proof that the class's axioms hold in τ. All abstract consequences of the class's axioms then follow for τ.

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

type of a term when working with polymorphic definitions.

4 Memory model

4.1 Type descriptions

The solution proposed in §2 requires that type meta-data be available at the HOL level. This needs to include information about the type structure, size, and alignment. In addition, a fine grained description of the value representation encoding and decoding functions, such that it is possible to extract the functions for specific fields as well as the structure as a whole, is desirable.

At the HOL level, structure objects are represented 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 — we write $v(\!|\mathsf{b} := x|\!)$ for b-update (K x) v. 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. To facilitate this, functions derived from the **record** functions are included in the type meta-data.

Definition 4.1 We can capture *abstract* record access and update functions for fields as *field descriptions*:

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

These functions provide a connection between the structure's value as a typed HOL object and the value of a field in the structure as a *byte list*. field-access takes an additional *byte list* parameter, utilised in the semantics to provide the existing state of the *byte* sequence representing the field being described. This allows padding fields the ability to "pass through" the previous state during an update [2]. E.g. The field description for field `b` in `struct a` would be:

$$\begin{array}{c} (\!|\mathsf{field\text{-}access} = \mathsf{to\text{-}bytes} \circ \mathsf{b},\ \mathsf{field\text{-}update} = \lambda bs\ s.\ \mathsf{if}\ |bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\textit{int}) \\ \mathsf{then}\ s(\!|\mathsf{b} := \mathsf{from\text{-}bytes}\ bs|\!)\ \mathsf{else}\ s|\!) \end{array}$$

Definition 4.2 The type meta-data is captured in a *type description* with the following mutually-inductive definitions:

$$\begin{array}{lll} \textbf{datatype} \quad \alpha\ \textit{typ-desc} & = & \mathsf{TypDesc}\ \alpha\ \textit{typ-struct}\ \textit{typ-name} \\ \alpha\ \textit{typ-struct} & = & \mathsf{TypScalar}\ \textit{nat}\ \textit{nat}\ \alpha \\ & & |\ \mathsf{TypAggregate}\ (\alpha\ \textit{typ-desc} \times \textit{field-name})\ \textit{list} \end{array}$$

A type description is a tree, with structures as internal nodes, branches labeled with field names and leaves corresponding to fields with primitive types. At leaves, size, alignment and an α is provided. A type description for `struct a` is given in Fig. 2.

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

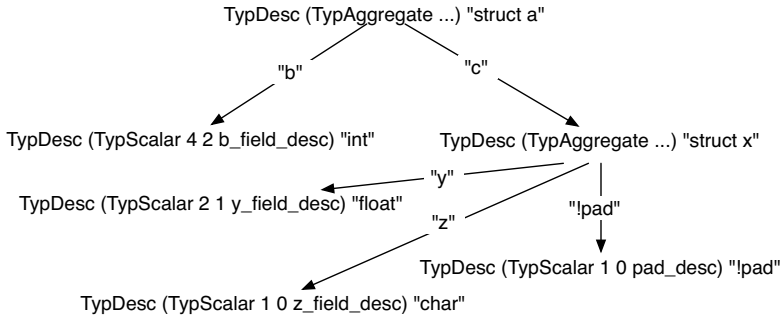

Fig. 2. Type description for `struct a`.

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. Type descriptions are specialised in two ways:

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

The type information provides the information required to describe the encoding and decoding of the representation. Type information t can be "exported", with a function export-uinfo, to remove the α dependency with export-uinfo t, where leaf field descriptions are collapsed to *byte list* normalisation functions, i.e. an α *field-desc* d at a leaf with size n is replaced with λbs. field-access d (field-update d bs arbitrary) (replicate n 0).

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. It provides us with a means to quantify over and compare C types.

The type information for a C type α is given by $\mathsf{TYPE}(\alpha)_\tau$ and we write $\mathsf{TYPE}(\alpha)_\nu$ for export-uinfo $\mathsf{TYPE}(\alpha)_\tau$.

Definition 4.3 A field name used to access and update structure fields with the C . and $\rightarrow$ 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. Arrays members are named by index, e.g. $[''\textit{--array-}37'']$.

Table 1 provides a number of functions defined over type descriptions that we make use of in this paper. Here we summarise and provide examples — all functions are backed by primitive recursive definitions in Isabelle/HOL. $t \triangleright f$ performs "lookup", following a path f from the root of t and returning a sub-tree and offset if it exists. A related concept is td-set, where all sub-trees are returned. E.g.

$$\begin{array}{lcl} \mathsf{TYPE}(a\textit{-struct})_\nu \triangleright [''c''] & = & \lfloor (\mathsf{TYPE}(x\textit{-struct})_\nu,\ 4) \rfloor \\ \mathsf{TYPE}(a\textit{-struct})_\nu \triangleright [''c'',\ ''b''] & = & \bot \\ \textsf{td-set}\ \mathsf{TYPE}(x\textit{-struct})_\nu & = & \{(\mathsf{TYPE}(x\textit{-struct})_\nu,\ 0),\ (\mathsf{TYPE}(\textit{float})_\nu,\ 0),\ (\mathsf{TYPE}(\textit{char})_\nu,\ 2), \\ & & (\textsf{pad-export}\ 1,\ 3)\} \end{array}$$

size-td and align-td are found by summing and taking the maximum of the leaf node sizes and alignments respectively. The latter is justified by the C standard's requirement that fields of aligned structures are themselves aligned. field-access-ti and field-update-ti compose their respective primitive leaf functions sequentially to provide the expected encoding and decoding functions for the aggregate type. E.g.

$$\text{field-access-ti TYPE}(a\text{-}struct)_\tau = \lambda v\ bs.\ \text{to-bytes (b } v\text{) (take (size-of TYPE}(int)\text{) } bs\text{) @ to-bytes (c } v\text{) (take (size-of TYPE}(x\text{-}struct)\text{) (drop (size-of TYPE}(int)\text{) } bs\text{))}$$

- ▷ -	:: α *typ-desc* ⇒ *qualified-field-name* ⇀ α *typ-desc* × *nat* The sub-tree and offset from the base of the structure that a valid qualified field name leads to.
td-set	:: α *typ-desc* ⇒ (α *typ-desc* × *nat*) *set* The set of all sub-trees and their offset from the base of a structure.
size-td	:: α *typ-desc* ⇒ *nat* Type size, e.g. size-td TYPE(a-*struct*)$_\tau$ = 8.
align-td	:: α *typ-desc* ⇒ *nat* Type alignment exponent, e.g. align-td TYPE(a-*struct*)$_\tau$ = 2.
field-access-ti	:: α *typ-info* ⇒ (α ⇒ *byte list* ⇒ *byte list*) Derived field access for the entire structure represented by the type information.
field-update-ti	:: α *typ-info* ⇒ (*byte list* ⇒ α ⇒ α) Derived field update for the entire structure represented by the type information.
export-uinfo	:: α *typ-info* ⇒ *typ-uinfo* Export type information.

Table 1
Type description functions.

Definition 4.4 The address corresponding to an *lvalue* designated by a structure field access or update can be found with:

$$\&(p{::}\alpha\ ptr\ {\to}f) \equiv \text{ptr-val } p + \text{of-nat (snd (the (TYPE}(\alpha)_\nu \triangleright f)))$$

Lvalues appear in the semantics and proof obligations for statements like `p->f = v;`.

Definition 4.5 Finally, the connection between the HOL typed value, type information, size, alignment and underlying byte representation can be made through the following function definitions:

$$\text{to-bytes } (v{::}\alpha) \equiv \text{field-access-ti TYPE}(\alpha)_\tau\ v \qquad \text{from-bytes } bs \equiv \text{field-update-ti TYPE}(\alpha)_\tau\ bs\ \text{arbitrary}$$
$$\text{size-of TYPE}(\alpha) \equiv \text{size-td TYPE}(\alpha)_\tau \qquad \text{align-of TYPE}(\alpha) \equiv 2\ \hat{}\ \text{align-td TYPE}(\alpha)_\tau$$

4.2 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 ensure that the functions in Defn. 4.5 and the rest of §4.1 behave as expected by the C standard and in the proofs of the update rules. They are also available to the user of the framework.

Definition 4.6 The α::*mem-type* axiomatic type class requires the following size and alignment related properties to hold on a C type α for instantiation:

$$\text{align-of TYPE}(\alpha)\ \text{dvd size-of TYPE}(\alpha) \qquad \text{size-of TYPE}(\alpha) < |addr| \qquad \text{align-of TYPE}(\alpha)\ \text{dvd}\ |addr|$$

These conditions follow mostly from requirements in the C standard, with the exception of the final alignment constraint which we add to make pointer arithmetic

better behaved, and which holds on all the C implementations we are aware of. The constant $|addr|$ represents the size of the address space, e.g. 2^{32}.

The result of an entire structure update is independent of the original value:

$$|bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) \longrightarrow \mathsf{field\text{-}update\text{-}ti}\ \mathsf{TYPE}(\alpha)_\tau\ bs\ v = \mathsf{field\text{-}update\text{-}ti}\ \mathsf{TYPE}(\alpha)_\tau\ bs\ w$$

Three well-formedness conditions on the type information ensure sensible values for field names, node sizes and field descriptions:

$$\mathsf{wf\text{-}desc}\ \mathsf{TYPE}(\alpha)_\tau \qquad \mathsf{wf\text{-}size\text{-}desc}\ \mathsf{TYPE}(\alpha)_\tau \qquad \mathsf{wf\text{-}field\text{-}desc}\ \mathsf{TYPE}(\alpha)_\tau$$

These conditions are now detailed in Defn. 4.7, Defn. 4.8 and Defn. 4.10.

Definition 4.7 We write wf-desc t when a type description t has no node with two or more branches labelled with the same field name.

Definition 4.8 We write wf-size-desc t when every node of the type description t has a non-zero size.

Definition 4.9 Type information t is *consistent* if the following properties hold:

$$\begin{gathered}
\forall v\ bs\ bs'.\ |bs| = |bs'| \longrightarrow \mathsf{field\text{-}update\text{-}ti}\ t\ bs\ (\mathsf{field\text{-}update\text{-}ti}\ t\ bs'\ v) = \mathsf{field\text{-}update\text{-}ti}\ t\ bs\ v \\
\forall v\ bs.\ |bs| = n \longrightarrow \mathsf{field\text{-}update\text{-}ti}\ t\ (\mathsf{field\text{-}access\text{-}ti}\ t\ v\ bs)\ v = v \\
\forall bs.\ |bs| = n \longrightarrow (\forall bs'.\ |bs'| = n \longrightarrow (\forall v\ v'.\ \mathsf{field\text{-}access\text{-}ti}\ t\ (\mathsf{field\text{-}update\text{-}ti}\ t\ bs\ v)\ bs' = \\
\mathsf{field\text{-}access\text{-}ti}\ t\ (\mathsf{field\text{-}update\text{-}ti}\ t\ bs\ v')\ bs')) \\
\forall v\ bs.\ |bs| = n \longrightarrow |\mathsf{field\text{-}access\text{-}ti}\ t\ v\ bs| = n
\end{gathered}$$

where $n = \mathsf{size\text{-}td}\ t$. The properties are similar to those already provided by Isabelle's **record** package at the HOL level and can be established automatically.

Definition 4.10 Type information is well-formed w.r.t. field descriptions if all leaf fields are consistent, and for every pair of distinct leaf fields, s and t, the following properties hold:

$$\begin{gathered}
\forall v\ bs\ bs'.\ \mathsf{field\text{-}update\text{-}ti}\ s\ bs\ (\mathsf{field\text{-}update\text{-}ti}\ t\ bs'\ v) = \mathsf{field\text{-}update\text{-}ti}\ t\ bs'\ (\mathsf{field\text{-}update\text{-}ti}\ s\ bs\ v) \\
\forall v\ bs\ bs'.\ |bs| = \mathsf{size\text{-}td}\ t \longrightarrow |bs'| = \mathsf{size\text{-}td}\ s \longrightarrow \mathsf{field\text{-}access\text{-}ti}\ s\ (\mathsf{field\text{-}update\text{-}ti}\ t\ bs\ v)\ bs' = \\
\mathsf{field\text{-}access\text{-}ti}\ s\ v\ bs'
\end{gathered}$$

Again, these are standard commutativity and non-interference properties that we have at the HOL level and wish to preserve in field descriptions.

Theorem 4.11 *The α::mem-type axioms imply the following properties:*

$$\frac{|bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)}{\mathsf{from\text{-}bytes}\ (\mathsf{to\text{-}bytes}\ v\ bs) = v} \qquad \frac{|bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)}{|\mathsf{to\text{-}bytes}\ v\ bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)} \qquad 0 < \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)$$

4.3 Type combinators

The constraints of the previous section require both the construction of suitable type information and a corresponding α::*mem-type* instantiation proof for each type appearing in programs we wish to verify. This can be done entirely at the ML level, 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. We use this approach and combinators and corresponding proof rules have been derived, but we elide for brevity.

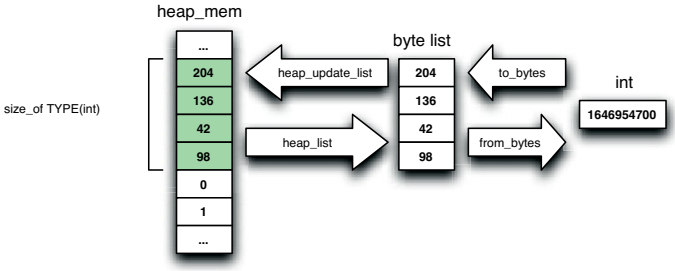

Fig. 3. *int* heap representation.

4.4 Semantics

The C translation has a shallow HOL embedding as its target for expressions. Tuch et al [14] provide details of how side-effects and other aspects of the C semantics are translated, here we provide simply the definitions for the terms used to model heap accesses and updates.

Definition 4.12 Heap dereferences in expressions, e.g. $*p + 1$ are given a semantics by first lifting the raw heap state with the polymorphic lift function, e.g. lift $s\ p + 1$ where s is the current state.

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

$$\begin{aligned}
&\textsf{lift} :: (addr \Rightarrow byte) \Rightarrow \alpha{::}c\text{-}type\ ptr \Rightarrow \alpha\\
&\textsf{lift}\ h \equiv \lambda p.\ \textsf{from-bytes}\ (\textsf{heap-list}\ h\ (\textsf{size-of TYPE}(\alpha))\ (\textsf{ptr-val}\ p))
\end{aligned}$$

heap-update providing semantics for update dereferences:

$$\begin{aligned}
&\textsf{heap-update-list} :: addr \Rightarrow byte\ list \Rightarrow (addr \Rightarrow byte) \Rightarrow (addr \Rightarrow byte)\\
&\textsf{heap-update-list}\ p\ []\ h \equiv h\\
&\textsf{heap-update-list}\ p\ (x{\cdot}xs)\ h \equiv \textsf{heap-update-list}\ (p+1)\ xs\ (h(p := x))\\
&\textsf{heap-update}\ p\ (v{::}\alpha)\ h \equiv \textsf{heap-update-list}\ (\textsf{ptr-val}\ p)\ (\textsf{to-bytes}\ v\ (\textsf{heap-list}\ h\ (\textsf{size-of TYPE}(\alpha))\\
&\qquad (\textsf{ptr-val}\ p)))\ h
\end{aligned}$$

For example, $*p = *q + 5$ translates to the state transformer $\lambda s.$ heap-update p (lift $s\ q + 5$) s. Fig. 3 illustrates the above functions' value transformations.

4.5 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. In earlier work [14], we introduced this mapping as an additional state component, and referred to it as the *heap type description*:

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

The heap type description is a *history* 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

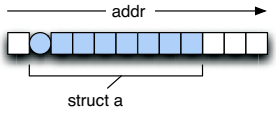

Fig. 4. Previous heap type description with a valid `struct a`. pointer

description. 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. 4.

The problem with this notion of the heap type description is that only a single pointer may be valid at any location. 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 validity monotonicity property, i.e. $d,g \models_t p \implies d,g \models_t \mathsf{Ptr}\ \&(p{\rightarrow}f)$.

To achieve this, we introduce a new definition for the heap type description:

$$\begin{array}{lcl} \mathit{typ\text{-}slice} & = & \mathit{nat} \rightharpoonup \mathit{typ\text{-}uinfo} \times \mathit{bool} \\ \mathit{heap\text{-}typ\text{-}desc} & = & \mathit{addr} \Rightarrow \mathit{bool} \times \mathit{typ\text{-}slice} \end{array}$$

Each location maps to a tuple, with the first component a *bool* indicating whether there is a value located at the address[3]. The second component is a *typ-slice*, providing an indexed map to the *typ-uinfo*s that may reside at a particular address. The index is calculated from the depth of the tree at an offset. The *bool* value indicates whether the location is the base or some other part of a value's footprint[4].

An example of the new heap type description is provided in Fig. 5. Each point is a *typ-uinfo* $\times$ *bool* pair, with the colour determined by the first component and shape by the second. Here an *a-struct* 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. 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 history variable.

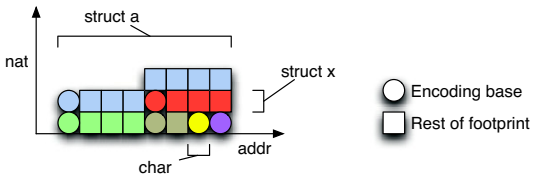

Fig. 5. New heap type description with a valid `struct a` pointer.

Definition 4.13 Pointer validity is defined for the heap type description as:

[3] This approach is taken in preference to a partial function to aid in partitioning state in §6.

[4] This is for same reason as in the previous approach to the heap type description, allowing consideration of the potential overlap of values of the same type to be eliminated for valid pointers.

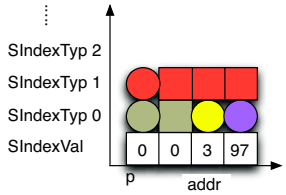

Fig. 6. Example *heap-state*.

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

$$d,g \models_t (p::\alpha\ ptr) \equiv \text{valid-footprint } d\ (\text{ptr-val } p)\ \text{TYPE}(\alpha)_\nu \wedge g\ p$$

where list-map::$\alpha\ list \Rightarrow (nat \rightharpoonup \alpha)$ converts a list to the expected map and typ-slice takes a vertical slice of the intended heap footprint from the exported type information at a given offset, e.g.:

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

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

As before, we have a retyping function ptr-retyp that updates the heap type description to make a given pointer valid. The definitions, properties and rules for this function are omitted for brevity.

5 Typed heaps

5.1 Lifting

The following two-stage lifting process provides an abstract heap view for proofs.

Definition 5.1 The first stage, lift-state, results in an intermediate *heap-state*:

$$\begin{array}{lcl}\textbf{datatype } s\text{-}heap\text{-}index & = & \text{SIndexVal} \mid \text{SIndexTyp } nat\\ \textbf{datatype } s\text{-}heap\text{-}value & = & \text{SValue } byte \mid \text{STyp } typ\text{-}uinfo \times bool\\ s\text{-}addr & = & addr \times s\text{-}heap\text{-}index\\ heap\text{-}state & = & s\text{-}addr \rightharpoonup s\text{-}heap\text{-}value\end{array}$$

An example of this state is provided in Fig. 6, with a *x-struct* footprint. The explanation for this model is provided in §6.1.

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.

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

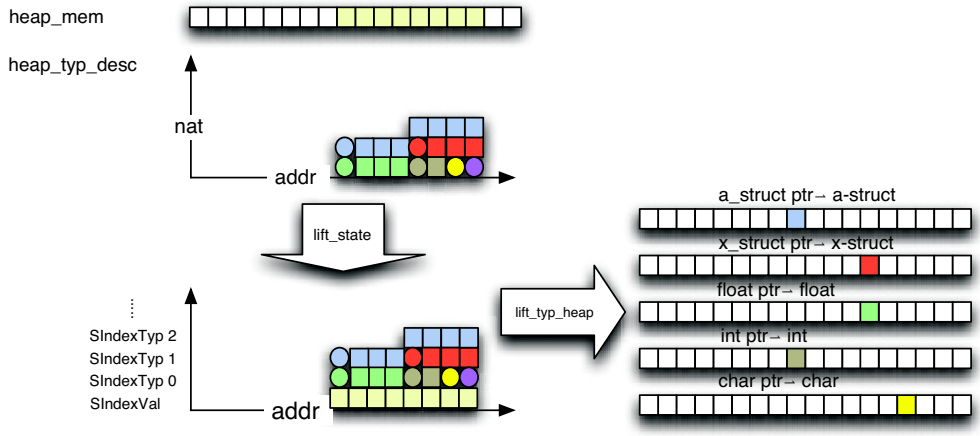

Fig. 7. Two-stage lifting.

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.

Definition 5.2 The second lifting stage results in typed lifted heaps again. The lift-typ-heap function restricts the heap domain so that the only locations affecting the resultant $\alpha\ ptr \rightharpoonup \alpha$ heap are valid pointer values. Equality is now modulo pointer validity.

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

The two stages, shown in Fig. 7, are combined with lift_τ:

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

Like lift, lift_τ is polymorphic and returns an α typed heap. The program embedding continues to use the functions lift and heap-update, while pre/post conditions and invariants use the stronger lift_τ to make more precise statements.

5.2 Update dependency order

Definition 5.3 A partial order can be defined on type descriptions that expresses the update dependency between heaps, formalising the relation described in §2:

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

This can be lifted to a predicate on α::*c-type itself* and β::*c-type itself*s:

$$s \leq_\tau t \equiv \text{export-uinfo}\ \text{TYPE}(\alpha)_\tau \leq \text{export-uinfo}\ \text{TYPE}(\beta)_\tau$$

Example 5.4 Using the running example, $\text{TYPE}(x\text{-}struct) <_\tau \text{TYPE}(a\text{-}struct)$ and $\text{TYPE}(int) <_\tau \text{TYPE}(a\text{-}struct)$. An update to an *a-struct* will always affect the lifted *int* heap, but an update of a *x-struct* will only sometimes affect the *a-struct* heap.

5.3 Rewrites

In this section we develop rewrites that allow the effects of updates on lifted typed heaps to be evaluated. First we present some auxiliary definitions and the key theorems, Thm. 5.7 and Thm. 5.9. These theorems have the form of conditional rewrites, but require some additional support to be efficiently applicable, so are followed by this detail.

Definition 5.5 A list of names of all fields matching an exported type information can be obtained with field-names :: α *typ-info* $\Rightarrow$ *typ-uinfo* $\Rightarrow$ *qualified-field-name list*. E.g. field-names TYPE(a-*struct*)$_\tau$ TYPE(*float*)$_\nu$ = [[$''c''$, $''y''$]].

Definition 5.6 From td-set, a predicate may be derived that checks whether a given pointer p::α *ptr* is to a field of a structured type with base q::β *ptr*:

$$\text{field-of } p\ q \equiv (\text{TYPE}(\alpha)_\nu, \text{unat } (\text{ptr-val } p - \text{ptr-val } q)) \in \text{td-set TYPE}(\beta)_\nu\ 0$$

From $\triangleright$, functions may be derived that provide the first and second components of the result for a valid qualified field name:

$$\text{field-typ TYPE}(\alpha)\ f \equiv \text{fst } (\text{the } (\text{TYPE}(\alpha)_\tau \triangleright f)) \qquad \text{field-offset TYPE}(\alpha)\ f \equiv \text{snd } (\text{the } (\text{TYPE}(\alpha)_\nu \triangleright f))$$

Theorem 5.7 *The lifted β heap following an update of a valid α ptr p, where α is a sub-type of β is given by:*

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

where

$$\begin{aligned}\text{super-field-update } p\ v\ h \equiv \lambda q.\ &\text{if field-of } p\ q\\ &\text{then case } h\ q \text{ of } \bot \Rightarrow \bot\\ &\quad \mid \lfloor w \rfloor \Rightarrow \lfloor \text{update-value } (\text{field-names TYPE}(\beta)_\tau\ \text{TYPE}(\alpha)_\nu)\ v\ w\\ &\qquad\qquad (\text{unat } (\text{ptr-val } p - \text{ptr-val } q)) \rfloor\\ &\text{else } h\ q\end{aligned}$$

Locations that do not enclose or are not valid β pointers are unaffected. The update is given by update-value:

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

This traverses the relevant fields of the enclosing structured type, looking for a field offset that matches the difference between the enclosing pointer base and p. If a match is found, update-value performs the update with the field's updator. We write to-bytes$_0$ and field-access-ti$_0$ when the supplied *byte list* is all zero.

While Thm. 5.7 gives a conditional rewrite that allows an update to be lifted to the typed heap level of §5.1, making use of the updated typed heap could involve unfolding this complex definition in general. However, additional rewrites can be given for well-behaved updates.

Theorem 5.8 *For a valid qualified field name, a* super-field-update *for a pointer $\&(p \rightarrow f)$ can be reduced to the field update obtained from the type information:*

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

The $\rhd$ side-condition can be resolved without having to unfold the type information using rewrites installed during construction with combinators at the ML level. The field-update-ti is also rewritten to a **record** field updator. E.g.:

$$\mathsf{lift}_\tau\ g\ s\ p = \lfloor w \rfloor \Longrightarrow \mathsf{super\text{-}field\text{-}update}\ (\mathsf{Ptr}\ \&(p{\rightarrow}[''next''])\ v\ (\mathsf{lift}_\tau\ g\ s) = \mathsf{lift}_\tau\ g\ s(p \mapsto w(\!|\mathsf{next} := v|\!))$$

A rewrite can also be given for the two remaining cases, where $\mathsf{TYPE}(\beta) <_\tau \mathsf{TYPE}(\alpha)$ or $\mathsf{TYPE}(\alpha) \perp_\tau \mathsf{TYPE}(\beta)$.

Theorem 5.9 *The lifted β heap following an update of a valid α ptr p, where α is not a strict sub-type of β is given by:*

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

where

$$\begin{array}{lcl}\mathsf{sub\text{-}field\text{-}update}\ []\ p\ (v{::}\alpha)\ s & \equiv & s{::}\beta\ ptr \rightharpoonup \beta \\ \mathsf{sub\text{-}field\text{-}update}\ (f{\cdot}fs)\ p\ v\ s & \equiv & (\mathsf{let}\ s' = \mathsf{sub\text{-}field\text{-}update}\ fs\ p\ v\ s\ \mathsf{in}\ s'(\mathsf{Ptr}\ \&(p{\rightarrow}f) \mapsto \mathsf{from\text{-}bytes} \\ & & (\mathsf{field\text{-}access\text{-}ti}_0\ (\mathsf{field\text{-}typ}\ \mathsf{TYPE}(\alpha)\ f)\ v)))\restriction_{\mathsf{dom}\ s}\end{array}$$

5.4 Non-interference

Theorem 5.10 *The rewrites for an update to a lifted typed heap through a valid pointer of the same type, or a disjoint type are the same as before [14]:*

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

Bornat [1] describes multiple independent heaps based on distinct field names. Updates through a pointer dereference to a specific field only affect that heap. This does not work directly in the presence of the $\&(p{\rightarrow}f)$ operator and address arithmetic. However, the following can be shown:

Theorem 5.11 *When the base pointers are of the same type β, and neither of the field names are a prefix of the other, updates through an α pointer derived from one field do not affect the value in the γ lifted heap at the other:*

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

6 Separation logic

In this section we describe how the shallow embedding of separation logic [5,10] in Tuch et al [14] can be extended to structured types. The focus is on the singleton heap assertion $p \mapsto_g v$ as most of the other definitions and properties are standard.

6.1 Domain

We model separation assertions as predicates on *heap-state*s, applied in assertions of the verification environment to the result of the first lifting stage of §5.1. For example, a loop invariant with the separation assertion P and heap memory and type description state in the variables h and d respectively is written $\{\!|\ P\ (\mathsf{lift\text{-}state}\ (`h, `d))\ |\!\}$, which we abbreviate as $\{\!|\ P^{sep}\ |\!\}$.

The rationale for this choice of domain is that it allows for more expressive separation assertions than are possible with simpler models. From the earlier intermediate state, *addr* $\rightharpoonup$ *typ-tag option* $\times$ *byte* for unstructured types, a naive extension might be *addr* $\rightharpoonup$ *typ-uinfo list* $\times$ *byte*. Unfortunately, this does not allow for two assertions separated by $\wedge^*$ to refer to distinct type levels at the same address, necessary to provide flexible rules for retyping and unfolding, e.g. ignoring padding, we would expect that $(p \mapsto (\!|\ y = 3,\ z = 'r'\ |\!)) = (\mathsf{Ptr}\ (\&(p{\rightarrow}[''y'']))) \mapsto 3) \wedge^*\ (\mathsf{Ptr}\ (\&(p{\rightarrow}[''z'']))) \mapsto 'r') \wedge^*\ \mathsf{typ\text{-}outline}\ p$, where $\mathsf{typ\text{-}outline}\ p$ contains the outer level type information for the enclosing structure. Adding a type level index to the domain of the *heap-state* provides this facility.

6.2 Shallow embedding

Definition 6.1 The $\mathsf{s\text{-}footprint}::\alpha::$*c-type ptr* $\Rightarrow$ *s-addr set* gives the set of addresses inside a pointer's *heap-state* footprint:

$$\begin{aligned}\mathsf{s\text{-}footprint}\ p \equiv \{&(\mathsf{ptr\text{-}val}\ p\ +\ \mathsf{of\text{-}nat}\ x,\\ &y)\ |\ x < \mathsf{size\text{-}td}\ \mathsf{TYPE}(\alpha)_\nu\ \wedge\\ &(y = \mathsf{SIndexVal} \vee (\exists n.\ y = \mathsf{SIndexTyp}\ n \wedge n < |\mathsf{typ\text{-}slice}\ \mathsf{TYPE}(\alpha)_\nu\ x|))\}\end{aligned}$$

Definition 6.2 $p \mapsto_g v$ asserts that the heap contains exactly one mapping matching the guard g, at the location given by pointer p to value v:

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

wf-heap-val asserts 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 6.3 The standard definitions [10] for connectives can then be used, for the empty heap predicate, separation conjunction and implication these are:

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

Since this is a shallow embedding, standard HOL connectives and quantifiers can be freely mixed with the separation connectives, e.g. $\lambda s.\ P\ s \wedge (Q \wedge^* R)\ s$.

The standard commutative, associative, and distributive properties apply to the connectives, and we have formalised pure, intuitionistic, domain, and strictly exact assertions and properties [10]. The frame rule also still applies in this development.

6.3 Lifting proof obligations

Our verification condition generator applies weakest precondition rules to transform Hoare triples to HOL goals that can then be solved by applying theorem prover tactics. In §5.3, rewrites were given that could lift the raw heap component of these proof obligations, and in this section we provide rules that allow the low-level applications of lift and heap-update to be expressed in terms of separation assertions. This is desirable as reasoning can then use the derived rules for these assertions at the separation logic level.

Theorem 6.4 *The following rule connects* lift *and separation mapping assertions:*

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

Heap update dereferences produce proof goals of the form:

$$P\ (\text{lift-state}\ (h,\ d)) \Longrightarrow Q\ (\text{lift-state}\ (\text{heap-update}\ p_0\ v_0\ (\text{heap-update}\ p_1\ v_1\ (\text{heap-update}\ p\ldots\ \ v\ldots\ (\text{heap-update}\ p_n\ v_n\ h))),d))$$

Theorem 6.5 *To reduce* heap-update*s to the pre-state we can use:*

$$\frac{(p \mapsto_g w \wedge^* R)\ (\text{lift-state}\ (h,\ d)) \qquad \text{TYPE}(\beta)_\tau \triangleright f = \lfloor (s,\ n) \rfloor \qquad \text{export-uinfo}\ s = \text{TYPE}(\alpha)_\nu}{(p \mapsto_g \text{field-update-ti}\ s\ (\text{to-bytes}_0\ v)\ w \wedge^* R)\ (\text{lift-state}\ (\text{heap-update}\ (\text{Ptr}\ \&(p \rightarrow f))\ v\ h,\ d))}$$

Thm. 6.5 can be applied in goals in similar situations to Thm. 5.8.

Theorem 6.6 *The earlier* heap-update *rules [14] still apply:*

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

6.4 Unfolding

Additional rules can be given that allow one to dive inside a singleton heap assertion for a structured type value. This may be needed in extracting points-to information to aid in discharging guard proof obligations or side-conditions of some of the rules such as Thm. 6.4 and is useful in allowing the granularity of an assertion to be changed.

Theorem 6.7 *A points-to mapping assertion for a valid qualified field name can be derived from a singleton heap assertion with:*

$$\frac{(p \mapsto_g v)\ s \qquad \text{TYPE}(\beta)_\tau \triangleright f = \lfloor (t,\ n) \rfloor \qquad \text{export-uinfo}\ t = \text{TYPE}(\alpha)_\nu \qquad \text{guard-mono}\ g\ g}{(\text{Ptr}\ \&(p \rightarrow f) \hookrightarrow_g \text{from-bytes}\ (\text{field-access-ti}_0\ t\ v))\ s}$$

We have also developed a rewrite approach that unfolds fields for structured values — one can "zoom" in and out of structured values with this.

7 Example: In-place list reversal

Fig. 8 provides an example type-safe C program that performs in-place list reversal on a singly-linked list using a struct type to represent nodes.

```
struct node {
  int item;
  struct node *next;
};
```

```
struct node *reverse (struct node *ptr)
{
  struct node *last = NULL;

  while (ptr)
  {
    struct node *temp = ptr->next;

    ptr->next = last;
    last = ptr;
    ptr = temp;
  }
  return last;
}
```

Fig. 8. In-place list reversal C source code.

This is a standard example in the separation logic and pointer program verification literature [10,7,14] and the pre/post specification and loop invariant are provided in Thm. 7.2.

Definition 7.1 The specification and invariant make reference to a list abstraction predicate, which lifts from a pointer-linked data structure in the heap to the corresponding algebraic data-type for a *node list* in Isabelle/HOL:

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

Theorem 7.2 *The* `reverse` *function implements the following specification:*

$$\begin{array}{l} \forall\, zs.\ \{\!|(\mathsf{list}\ zs\ \acute{}ptr)^{sep}|\!\} \\ \quad \acute{}reverse\text{-}ret :== PROC\ reverse(\acute{}ptr) \\ \quad \{\!|(\mathsf{list}\ (\mathsf{rev}\ zs)\ \acute{}reverse\text{-}ret)^{sep}|\!\} \end{array}$$

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:

$$\{\!|\exists\, xs\ ys.\ (\mathsf{list}\ xs\ \acute{}ptr \wedge^*\ \mathsf{list}\ ys\ \acute{}last)^{sep} \wedge \mathsf{rev}\ zs = \mathsf{rev}\ xs\ @\ ys|\!\}$$

The $Pre \Longrightarrow Inv$ and $Inv \Rightarrow Post$ conditions are trivial. Loop invariant preservation proof requires we show:

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

This follows from Thm. 6.5. The first side-condition may be discharged with Thm. 6.4 and Thm. 6.7, eliminating the lift. The other side-conditions are discharged by rewrites installed during C translation for evaluating $\triangleright$.

□

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

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

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

Compared to our earlier in-place list reversal example [14], the proof script was about the same structure and size, 67 lines. In our experience, lifts and heap-updates can be reduced as above for type-safe C, freeing the user from this level of detail. However, a completeness result is not possible in this shallow treatment.

8 Related work

The idea to use separate heaps for separate pointer types and structure fields in Hoare logic goes back to Burstall [2]. On the abstract level, our multiple typed heaps formalisation is most closely related to Bornat [1] and Mehta and Nipkow's [7] work in Isabelle, although we exploit Isabelle's type inference in a different way. We ground this abstract and efficient reasoning in a detailed C semantics that is directly applicable to concrete programs, and extend support to C's structured types. Moy [8] has also developed a memory model for C structured types and a type hierarchy. This differs from ours as it is based on physical sub-typing [12] and the focus of the work is on translating well-behaved unions and casts to sub-typing instances. The Caduceus tool [3] supports Hoare logic verification of C programs, including the type-safe part of pointer arithmetic at this level. We increase the applicability of program verification drastically by supporting the unsafe part as well. Separation logic [5,10] has been mechanised in theorem proving systems previously [15,6]. Again, we provide soundness for program verification by grounding these abstract, idealised models in a concrete semantics. We are able to support abstract separation logic notation and unsafe, low-level pointer manipulations at the same time.

On the semantics front Norrish [9] presents a very thorough and detailed memory model of C and our formalisation has similarities to exploratory work on C++ [4]. Our model unifies these low-level semantics with the proof abstractions of the previous paragraph.

9 Conclusion

In this paper we continued earlier work on pointer program verification in higher-order logic for C programs by providing extensions and generalisations resulting in a framework capable of fully exploiting C's structured types. We presented a development that deeply embeds type structure information in the theorem prover and generic rules to describe type-safe updates in two common interactive proof abstractions — multiple-typed heaps and separation logic. With the former, we extended the earlier notion of heap independence to take into account a partial ordering of heap update dependency, and with the latter based the development on a heap state that allows for expressive assertions. Type-unsafe operations continue to be supported albeit at a proof cost.

Future work includes providing support for C's `union` types when they are well behaved, e.g. tagged unions, `struct` pointer casting in the case of physical sub-

typing, development of Isabelle tactics for separation logic proofs and integration with automated tools and decision procedures.

Acknowledgement

We thank Gerwin Klein for discussions and for reading drafts of this paper.

References

[1] Bornat, R., *Proving pointer programs in Hoare Logic*, in: R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction (MPC 2000)*, LNCS **1837** (2000), pp. 102–126.

[2] Burstall, R., *Some techniques for proving correctness of programs which alter data structures*, in: B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, Edinburgh University Press, 1972 pp. 23–50.

[3] Filliâtre, J.-C. and C. Marché, *Multi-prover verification of C programs*, in: *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, USA*, LNCS **3308** (2004), pp. 15–29.

[4] Hohmuth, M., H. Tews and S. G. Stephens, *Applying source-code verification to a microkernel — the VFiasco project*, Technical Report TUD-FI02-03-März, TU Dresden (2002).

[5] Ishtiaq, S. S. and P. W. O'Hearn, *BI as an assertion language for mutable data structures*, in: *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2001), pp. 14–26.

[6] Marti, N., R. Affeldt and A. Yonezawa, *Verification of the heap manager of an operating system using separation logic*, in: *Third workshop on Semantics, Program Analysis, and Computing Environments For Memory Management (SPACE 2006)*, 2006, pp. 61–72.

[7] Mehta, F. and T. Nipkow, *Proving pointer programs in higher-order logic*, Information and Computation (2005), to appear.

[8] Moy, Y., *Union and cast in deductive verification*, in: *C/C++ Verification Workshop*, Oxford, UK, 2007.

[9] Norrish, M., "C formalised in HOL," Ph.D. thesis, Computer Laboratory, University of Cambridge (1998).

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

[11] Schirmer, N., "Verification of Sequential Imperative Programs in Isabelle/HOL," Ph.D. thesis, Technische Universität München (2006).

[12] Siff, M., S. Chandra, T. Ball, K. Kunchithapadam and T. Reps, *Coping with type casts in C*, in: *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering* (1999), pp. 180–198.

[13] Tuch, H. and G. Klein, *A unified memory model for pointers*, in: G. Sutcliffe and A. Voronkov, editors, *12th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-12)*, LNCS **3835**, 2005, pp. 474–488.

[14] Tuch, H., G. Klein and M. Norrish, *Types, bytes, and separation logic*, in: M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Nice, France, 2007, p. 12.

[15] Weber, T., *Towards mechanized program verification with separation logic*, in: J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004*, Lecture Notes in Computer Science **3210** (2004), pp. 250–264.

[16] Wenzel, M., *Type classes and overloading in higher-order logic*, in: E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics'97*, LNCS **1275** (1997), pp. 307–322.