Separation Logic Predicates for Aggregate Data Types

Qinxiang Cao
Princeton University
caoqinxiang@gmail.com

Andrew Appel
Princeton University
appel@princeton.edu

Abstract
Many imperative languages have type systems built from a few elementary types with rules for building aggregate types. In C, integers, floating point numbers, and pointers are elementary types; struct, union and array build aggregates. When using separation logic to write assertions for such languages, one uses separating conjunction \(\ast\) to describe aggregate values in memory. These conjunctions can be large and unwieldy.

We formalize a separation logic predicate, \texttt{data-at}, to describe aggregate-typed data stored in memory. Also, we present an extension of separation logic to conveniently manipulate aggregate types and \texttt{data-at}. We use the ordinary maps-to predicate to define the semantics of our new predicate and prove all our separation-logic proof rules sound in Coq. Based on previous work in the Verified Software Toolchain project, we provide a tool enabling people to write concise specifications for C programs that use aggregate types and to formally prove them correct in a convenient way.

1. Introduction
Separation logic \cite{c:O48} has been widely used in program verification. One writes \((p \mapsto v)\) to represent that a piece of data \(v\) is stored at address \(p\) in memory. In the past decade, systems such as the Verified Software Toolchain \cite{c:O48} and Charge! \cite{c:O48} have enabled people to build Hoare-style correctness proofs for real programs. In these systems, people can easily write specifications in separation logic. However, in separation logic it can be inconvenient to handle aggregate types.

For example, in C, programmers can use struct types, union types and arrays to build aggregates from elementary types (integers, floating point numbers, pointers and etc). For example,

\begin{verbatim}
struct IntPair { int a; int b; } *x;
\end{verbatim}

A reasonable triple for a common store command can be,

\[
\begin{aligned}
\{(\llbracket x \rrbracket_r \mapsto u \ast \llbracket x \rrbracket_r + 4 \mapsto v)\} & \\
x \rightarrow b = 0
\end{aligned}
\]

An integer occupies 4 bytes, so the second field is stored at \(\llbracket x \rrbracket_r + 4\). The notation \(\llbracket e \rrbracket_r\) evaluates the \(r\)-value interpretation of \(e\) in the current environment. Similarly, \(\llbracket e \rrbracket_l\) means the \(l\)-value interpretation of \(e\), which is always a memory address.

When verifying real programs, aggregate types are often large (struct, union and array nested, many fields in a struct) and there may be several aggregates described in one single assertion. This leads to many conjuncts in the separating conjunction: this is verbose and it slows down proof-building, proof-checking, and symbolic execution. Therefore, Reynolds proposed \cite{c:O48} a more compact notation, \((p \mapsto (u, v))\).

\[
\begin{aligned}
\{(\llbracket x \rrbracket_r \mapsto (u, v))\} & \\
x \rightarrow b = 0 & \quad \{(\llbracket x \rrbracket_r \mapsto (u, 0))\}
\end{aligned}
\] (1)

In this paper we formalize Reynolds’s notation and define its semantics in Coq. We introduce a predicate

\[
p \mapsto_{t} v
\]

or \((\texttt{data\_at\ t\ v\ p})\), saying that a datum \(v\) of \(C\) type \(t\) is stored at address \(p\). Intuitively, it can be treated as a separating conjunction of ordinary maps-to predicates.

A mechanized system such as VST is not just a program logic (with soundness proof)—it also has proof automation to help users apply the logic to programs. When \texttt{data-at} is involved, our tactical program should at least be able to derive triples like the one in Fig. 1. Efficiency is another challenge in proof automation. In Coq, tactical proof scripts build proofs, which are rechecked when the \texttt{Qed} command is reached. But tactical proofs can be very slow, and can lead to complex proof terms which hinder the performance of \texttt{Qed}.
These challenges are exacerbated by dependent types. In $p \vdash v$, the type of $v$ is dependent on the value of $t$. Dependent typing programing is complicated and proving properties of dependently typed functions is hard. Heavy use of dependent types causes more difficulty in designing proof automation, and it can make Coq slow in type inference (slowing tactic execution) and in type checking (slowing Qed). In this work we introduce an extension of separation logic and adapt VST’s automation to this extension to permit concise formal proofs with short specs. All separation logic entailments and Hoare rules in our logic are proved sound.

In §7 discusses related work and §8 concludes.

Our Coq development is based on the Verified C program logic of the Verified Software Toolchain [1, Chapter 22]. Specifically, the semantics of data-at is defined by the semantics of elementary-type maps-to, based on a C resource model already formally defined in VST. Our soundness proofs of separation logic entailments are based on VST’s formalization of standard separation logic, and our Hoare-rule proofs are constructed as corollaries of primitive rules such as the consequence rule, frame rule and sequence rule, all previously proved sound w.r.t to CompCert’s Clight semantics [?].

We have built three systems: an industrial-strength system embedded in VST, and two simplified Coq development on tiny models for presentation in this paper. The small systems are submitted together with this paper which also contains a link to the full system in the VST github repository.

2. An Extension of Separation Logic

2.1 Background: Verify Hoare triples in VST

Verifyable C [?, Chapter 22] is a Hoare logic for C, embedded in Coq and proved sound w.r.t. the operational semantics defined by CompCert. It is expressive enough to use in proving full functional correctness of real C programs. VST offers a semi-automation system to help people build these proofs. Specifically, when the proof goal is a triple $\{ P \} c_0; c \{ Q \}$, an Ltac program forward will first analyze the first command $c_0$ and the precondition $P$ and then apply corresponding Hoare rules to simplify the proof goal. Users must write the precondition in normal form, $v \cdot \land \cdots \land \ast \ast \ast \ast \ast \ast$ in which the first part is a conjunction of pure facts and the second part is a separating conjunction of spatial facts.

The forward tactic will analyze the spatial part of $P$ only if $c_0$ is a load command, a store command or a function call. So the work we described in this paper affects the logic and automation of only these three kinds of C commands.

\[
\begin{align*}
\{ [x]_r \mapsto u \cdot [x]_r + 4 \mapsto v \} & \quad x \rightarrow b = 0 \quad \{ [x]_r \mapsto u \cdot [x]_r + 4 \mapsto 0 \} \\
\{ [x]_r \mapsto \text{IntPair} (u, v) \} & \quad x \rightarrow b = 0 \quad \{ [x]_r \mapsto \text{IntPair} (u, 0) \}
\end{align*}
\]

Figure 1: Example: Hoare triple derivation

When $c_0$ is a load command, forward will apply the sequence rule and LoadRule. Users should ensure that $(p \mapsto \_)$ appears as a separating conjunct in $P$. Similarly, when $c_0$ is a store command, forward will apply the sequence rule and StoreRule.

When $c_0$ is a function call, forward will apply sequence rule, frame rule and the specification of the function. Function specifications are written as parameterized triples, e.g.

\[
\forall v_1 v_2 \ldots v_n, \{ P_f (v_1, v_2, \ldots, v_n) \} f () \{ Q_f (v_1, v_2, \ldots, v_n) \}
\]

The precondition $P_f$ should be in normal form. Users must instantiate the parameters $v_1$ … $v_n$, and forward will automatically pick a subset of $P_f$’s conjuncts to form the spatial part of $P_f (v_1, v_2, \ldots, v_n)$; the remaining conjuncts form the frame.

2.2 Unfolding Equations

When preconditions are built from elementary maps-to predicates, forward can always find the required $(p \mapsto \_)$ clause for load/store commands and can do frame inference for function calls. But not when data-at is involved. A naive solution to the problem is to unfold data-at into conjunctions of elementary maps-tos.

The predicate $p \mapsto v$, or $(\text{data-at} \ t \ v \ p)$, is dependently typed. The type of $v$ is dependent on C-type $t$. Specifically, when $t$ is a struct, $v$ should have a product type. When $t$ is an array, $v$ should be a list. And when $t$ is a union, $v$ should have a sum type.

So, in $p \mapsto v$ has tree structure. Unfolding data-at is replacing a predicate of a tree with a separating conjunctions of maps-to predicates on tree leaves. Intuitively, every leaf corresponds a path of general fields (also called a nested
The offset of field $f_2$ is 4. The offset from the beginning of field $f_2$ to the
third array element is 8. Thus the offset of $[f_2; 2]$ is 12 in all.

\[ p \mapsto v = p \uparrow_t v \]
\[ p \mapsto \vec{F} = v = p \triangleright \vec{F} \mapsto v \quad \text{if } t . \vec{F} \text{ is an elementary type} \]
\[ p \mapsto \vec{F} = v = \bigstar_{f \in t.\vec{F}} \left( p \mapsto \vec{F}_f = v.\vec{F}_f \cdot \text{Space}(\vec{F}_f, p) \right) \quad \text{if } t.\vec{F} \text{ is a nonempty struct} \]
\[ p \mapsto \vec{F} = \{ f : v \} = p \mapsto \vec{F}_f = v \cdot \text{Space}(\vec{F}_f, p) \quad \text{if } t.\vec{F} \text{ is a union} \]
\[ p \mapsto \vec{F} = \bigstar_{0 \leq i < n} v_i \quad \text{if } t.\vec{F} \text{ is an array of positive length } n \]

Figure 2: Unfolding equations

2.3 Nested Load/Store Rule

For convenient verification of load/store commands, we add the NestedLoadRule and NestedStoreRule (Fig. 3) to handle \texttt{data.at} predicates. These rules use a projection function $u.\vec{F}$ that means: given an aggregate datum $u$ that can represent $C$ type $t$, give the datum that can represent the component of $u$ reachable by path $\vec{F}$. The update function $u[\vec{F}]$ means: given an aggregate datum $u$, give the aggregate datum obtained by replacing its $\vec{F}$ component with field value $v$.

We distinguish $\vec{F}$ from $\vec{F}_i$: they represent a nested field in a $C$ command and its denotation respectively. A struct/union field in a $C$ expression is the same as its denotation but an array subscript is an int-type expression while its denotation is an number. We design these rules with a format similar to LoadRule and StoreRule, so similar automation can be applied to handle load/store commands with \texttt{data.at} in pre-conditions.

In order to display nice proof goals to users, we do not want to print out $u.\vec{F}$ and $u[\vec{F}]$ all the time. Thus, the forward tactic will replace them with their computation result automatically. For example, from the proof goal of line (7), the forward tactic will leave (9) to users instead of (9).

\[ \{ p \mapsto (x, y) \} \mapsto b = 0; c \{ Q \} \]
\[ \{ p \mapsto (x, y)[0/[b]] \} c \{ Q \} \]
\[ \{ p \mapsto (x, 0) \} c \{ Q \} \]

Does our logic need nested load/store rules? No—without these rules, triples involving \texttt{data.at} and load/store commands could still be proved by the consequence rule, Load/StoreRule, and unfolding equations. We could automate such combinations in Ltac (tactic programming). However, the nested load/store rules create much smaller proof terms; Coq needs less memory, tactics execute faster, and Qed time is shorter.

\[ \{ p \mapsto (x, y) \} \mapsto b = 0; c \{ Q \} \]
\[ \{ p \mapsto (x, y)[0/[b]] \} c \{ Q \} \]
\[ \{ p \mapsto (x, 0) \} c \{ Q \} \]
\[ P \vdash \llbracket e \rrbracket_t = p \land \llbracket F \rrbracket_r = \bar{F} \land p \mapsto u \rightarrow \top \quad \text{type}(e) = t \]

\[
\{ P \} \ x = e.\bar{F} \{ \exists v'. \llbracket x \rrbracket_r = u.\bar{F} \land P[v'/x] \}
\]

\[ P \vdash \llbracket e_1 \rrbracket_t = p \land \llbracket e_2 \rrbracket_r = v \land \llbracket \bar{F} \rrbracket_r = \bar{F} \land P' \mapsto p \mapsto u \quad \text{type}(e) = t \]

\[
\{ P \} \ e_1.\bar{F} = e_2 \{ P' \mapsto p \mapsto u[v/\bar{F}] \}
\]

Figure 3: Rules of separation logic

2.4 Reroot Equation

Suppose \texttt{int\_pair\_swap()} has the following function specification.

\[
\forall x, y.
\{ p \mapsto \text{IntPair} \rightarrow (x, y) \}
\]

\[
\{ p \mapsto \text{void} \text{IntPair\_swap} (\text{IntPair} \mapsto p) \}
\]

Then Figure 4 is a common scenario when verifying triples of its function call.

\[
\{ p \mapsto \text{ToyType} \rightarrow ((x, y), l) \}
\]

\[
\{ p \mapsto \text{ToyType} \rightarrow (x, y) \mapsto p \mapsto \text{ToyType} \rightarrow l \}
\]

\[
\{ p \mapsto \text{IntPair} \rightarrow (x, y) \mapsto p \mapsto \text{ToyType} \rightarrow l \}
\]

\[
\{ p \mapsto \text{IntPair} \rightarrow (y, x) \mapsto p \mapsto \text{ToyType} \rightarrow l \}
\]

\[
\{ p \mapsto \text{ToyType} \rightarrow ((y, x), l) \}
\]

Figure 4: An example of verifying function call

In Figure 4, a combination of unfolding equation (4) and (3) can verify the transition from (11) to (12). Here, we introduce an single rule—reroot equation—to replace this combination. We call it reroot equation since its left side is a predicate on an internal node of a tree and the right side treats the internal node as a root. Generally, whenever \texttt{data\_at} is used in the specification of a function, reroot equations are useful for verifying a corresponding function call.

\[ \text{Reroot Equation: } p \mapsto \bar{F} \mapsto v = p \mapsto \bar{F} \mapsto v \]

3. Semantics: from a simple model

To illustrate the most important features of our formal semantics of \texttt{data\_at} and other predicates and functions, we will start from a small sublanguage of C, whose type system contains only integer and struct types. For simplicity, we will also ignore the spaces among struct fields, i.e. no word-alignment rules.

**Inductive type:**

\[
\text{Type} :=
\begin{align*}
&| \text{Tint: type} \\
&| \text{Tstruct: } \forall (id: \text{ident}) (fs: \text{fieldlist}), \text{type} \\
&| \text{with fieldlist: Type :=} \\
&| \text{Fnil: fieldlist} \\
&| \text{Fcons: } \forall (hd: \text{ident} \mapsto \text{type}) (tl: \text{fieldlist}), \text{fieldlist}.
\end{align*}
\]

In our formal definition, the type of \( v \) in \( p \mapsto v \) is computed from the C-type \( t \), i.e. \texttt{data\_at} should have the following type.\(^2\)

\[ \text{reptype: type } \rightarrow \text{Type}. \]

\[ \text{data\_at: } \forall t, \text{reptype } t \rightarrow \text{val } \rightarrow \text{Pred}. \]

\[ \text{reptype } t \text{ stands for the representation data type of C-type } t. \]

Figure 5 shows the definitions of \texttt{reptype} and \texttt{data\_at} as recursive functions on C-types.

We could let \texttt{data\_at} be defined as \texttt{data\_at\_rec} directly. But instead, we first define \texttt{field\_at} as an instance of \texttt{data\_at\_rec} and \texttt{data\_at} is then defined as \texttt{field\_at} with empty field path. The semantics of \((\text{field\_at } t \ F v)\) is also dependently typed. The type of \( v \) is \texttt{reptype (nested\_field\_type } t \ F \text{)}, i.e. \texttt{reptype}(\texttt{t} \ F)\).

**Discussion 1.** Our definition of \texttt{data\_at\_rec} is rather verbose. Ideally, we would hope that Coq allowed the definition on the right column of Figure 5.

This succinct definition is illegal because after doing pattern matching on \( t \), the type of \( v \) which is dependent on \( t \) is not recomputed by the pattern matching result. To force Coq to do that type recalculation, we have to use \texttt{match ... as ... return ... with}.

**Discussion 2.** We choose to use product type to be the representation type of struct. In our full development, we choose sum type and list to be the representation type of union and array. An alternative approach, which we choose not to do, is to use one single inductive type in Coq to define \texttt{reptype}. Here, we make this design decision to allow users

\(^2\)The \texttt{data\_at} predicate takes another argument in our industrial strength development that we omit in this paper and toy demos: a permission-share indicating read-only, read-write, or various other levels of access to the data.
Fixpoint reptype (t: type): Type :=
match t with
| Tint ⇒ val
| Tstruct .fs ⇒ reptype.fl fs
end
with reptype.fl (fs: fieldlist) : Type :=
match fs with
| Fnil ⇒ unit
| Fcons (fieldname, t) fs_tl ⇒ prod (reptype t) (reptype.fl fs_tl)
end.

(* legal definition *)
Fixpoint data.at.rec (t: type) (v: reptype t) : val → Pred :=
match t as t-PAT return reptype t-PAT → val → Pred with
| Tint ⇒ fun v p ⇒
  mapsto p v
| Tstruct .fs ⇒ fun v p ⇒
  data.at.rec.fl fs v fs p
end v
with data.at.rec.fl (fs: fieldlist) (v: reptype.fl fs) (full-fs: fieldlist): val → Pred :=
match fs as fs-PAT return reptype.fl fs-PAT → val → Pred with
| Fnil ⇒ fun v p ⇒
  emp
| Fcons (id, t) fs_tl ⇒ fun v p ⇒
  data.at.rec.t (fst v)
  (offset.val (field.offset id full.fs) p) *
  data.at.rec.fl fs_tl (snd v) full.fs p
end v.

Fixpoint nested.field.type (t: type) (nf: list ident) : type := ...
Fixpoint nested.field.offset (t: type) (nf: list ident) : Z := ...

Definition field.at (t: type) (nf: list ident) (v: reptype (nested.field.type t nf)) (p: val): Pred :=
data.at.rec (nested.field.type t nf) v (offset.val (nested.field.offset t nf) p).

Definition data.at (t: type) (v: reptype t) (p: val) : Pred := field.at t nil v p.

Figure 5: Reptype and data_at

a more natural and idiomatic set of types for manipulating the mathematical values that serve as the meanings of their C data structures.

Projection and updating. We define \( \tilde{F}, v, F \) and \( u[v/F] \) by recursion on the length of field path. For simplicity of computation, these functions are defined as total functions. They will just give meaningless results when the field path is illegal; another predicate legal.nested.field tells whether the field path is legal or not. For the sake of space, we omit their definitions here.

4. Challenges from nonstructural recursion

In CompCert 2.4 and earlier versions, the Clight type definition is similar to our mutual inductive definition in the previous section. However, from CompCert 2.5, struct and union types are represented by name instead of by structure. Specifically, every Clight program is associated with a composite.env. A composite.env is a dictionary mapping every struct/union name to a list of all its fields. The following is a simplified version of this type system, in which PTree.t(α) implements an efficient computational map from ident to α.
4.1 A ranking system

One problem with this definition is that we cannot do induction on Clight types or write functions recursive on Clight types. All functions in Coq must be terminating. But in this definition, it is impossible to prevent infinitely circular types.

Because of this consideration, the CompCert developers accepted our suggestion that every type should be tagged with a rank, which is a natural number. The ranking system ensures that the rank of a struct type is the max rank of its fields plus one; the rank of a union type is the max rank of its fields plus one; the rank of an array type is the rank of its element type plus one. The rank of elementary types (including pointers) is zero. As a result, every type has finite rank.

Record composite : Type :=
  { co_members : members; co_rank : nat }.
Definition composite.env : Type := PTree.t composite.
Fixpoint rank.type (env : composite.env) (t : type) : nat :=
  composite-env-consistent
  + (map rank.type (map snd (co_members (get.co id))))

4.3 Challenge from nonconvertible equality between Coq types

When $a = b$ can be proved in Coq, we say they are provably equal. When $a$ is $\beta\delta\iota\eta$-convertible with $b$, we say they are convertible equal. Some equalities are provably equal but not convertible equal. For example, $a + (b + c)$ is not convertible but only provably equal to $(a + b) + c$ for general situations.

The left and right side of `reptype_eq` are provably but not convertible equal. For example, suppose $t_1$ and $t_2$ are two Clight types and $\text{rank.type}(t_1) < \text{rank.type}(t_2)$. Then, let $t$ be a struct type with two fields, which have type $t_1$ and $t_2$ respectively. Now, if we compute `reptype` of these Clight types, we will get the following result.

\[
\text{reptype}(t) = \text{reptype}(t_1) \times \text{reptype}(t_2)
\]
However, (17) can only be proved by induction in general situation (e.g. when \( t_1 \) and \( t_2 \) are not constant types but contain some Coq variables inside). Thus it cannot be a convertible equality. As a result, (16) is only provably true as well.

Generally, proving \texttt{reptype.eq} correct is based on the fact that for any Clight type \( t \) and any natural number \( n_1 \) and \( n_2 \), if \( n_1, n_2 \geq \text{rank}_\text{type}(t) \), then

\[
\text{reptype_rec}(t, n_1) = \text{reptype_rec}(t, n_2)
\]

Since it is not a convertible equality but can only be proved by induction, \texttt{reptype.eq} is not a convertible equality either.

Worse yet, because dependent types are involved in the whole development and \texttt{reptype} is widely used to compute the type of other functions (for example \texttt{proj.reptype}), \texttt{reptype.eq} is not only a nonconvertible equality about the results of a specific function but also a nonconvertible equality between Coq types. As a result, Coq does not know that a variable with type (\texttt{reptype (Tstruct i)}) is a tuple. So, when defining functions like \texttt{proj.reptype} in §3, the dependently typed pattern matching does not type-check.

To overcome this problem, we use \texttt{eq.rect} to type-cast in the definition of \texttt{proj.reptype} and apply \texttt{reptype.eq} to fill the type equality requirement of \texttt{eq.rect}.

\textbf{Definition} unfold \texttt{reptype} \{ t : type \}
\[(v : \text{reptype}\ t) : \text{REPTYPE} \ t := @\text{eq.rect} \ 	ext{Type} \ (	ext{reptype}\ t) \ (\text{fun} x : \text{Type} \Rightarrow x) \ v \ \text{(REPTYPE} \ t) \ (\text{reptype.eq} \ t)\].

\textbf{Definition} \texttt{proj.struct} (f : ident) (m : members)
\[(v : \text{reptype.members} m) : \text{reptype} \ (\text{field.type} f m) := \text{proj.list.prod} (f, \text{field.type} f m) m v \ \text{(default.val .)} \ \text{member.dec}\].

\textbf{Definition} \texttt{proj.gfield.reptype} (t : type)(f : ident)
\[(v : \text{reptype} t) : \text{reptype} \ (\text{field.type} t f) := \text{match} t \text{ as} \text{t.PAT return} \ \text{(REPTYPE} \ t, \text{PAT} \rightarrow \text{reptype} \ (\text{field.type} t, \text{PAT} f)) \text{ with} \]
| Tstruct id \Rightarrow \text{fun} v \Rightarrow \text{proj.struct} f \ (\text{co.members} \ (\text{get.co id})) v |
| \_ \Rightarrow \text{fun} \_ \Rightarrow \text{Vundef} |
\text{end} \ (\text{unfold}\ \text{reptype} \ v)\].

\textbf{Fixpoint} \texttt{proj.reptype} (t : type)(nf : list ident)
\[(v : \text{reptype} t) : \text{reptype} \ (\text{nested.field.type} t \ nf) := \text{match} nf \text{ as} \text{nf.PAT return} \ \text{reptype} \ (\text{nested.field.type} t \ nf.PAT) \text{ with} \]
| nil \Rightarrow v |
| f :: nf0 \Rightarrow \text{proj.gfield.reptype} f (\text{proj.reptype} t \ nf0 v) |
\text{end}\].

One consequence of using this explicit type casting is from the requirement that \( v.F \) should be convertible equal to its computational result (see §2.3). The computation of \( v.[F_1 F_2 \ldots F_n] \) now involves computing the following proof terms: (\texttt{reptype.eq} \ t), (\texttt{reptype.eq} \ t [F_1]), (\texttt{reptype.eq} \ t [F_1 F_2]) \ldots (\texttt{reptype.eq} \ t [F_1 F_2 \ldots F_{n-1}]). So, we make the proof of \texttt{reptype.eq} fully transparent in Coq. Specifically, \texttt{reptype.eq} is now type checked by Defined instead of Qed. Those standard-library theorems used by \texttt{reptype.eq} we reprove transparently in Coq.

\textbf{Discussion.} Coq supports nonstructural recursive functions in a limited way. It offers a well-founded recursive function fix and some syntactic sugar like \texttt{Program Fixpoint} and \texttt{Function}. However, in all these approaches to defining nonstructural recursive functions, (A) Internal function expressions are proof-involved and complicated. (B) Internal function expressions are structurally recursive on a proof term related to well-foundedness. (C) Function applications are not convertible equally to the intermediate result, e.g. \texttt{reptype t[10]} is not convertible to (\texttt{list} \ (\texttt{reptype} \ t)) when \( t \) is an abstract variable.

5. Other subtle issues in C semantics

In §3 and §4, we use a simplified model to illustrate some important features of \texttt{data.at}’s semantics. Real C semantics has more subtle setting. Also we made some design choices such that nicer subgoals can be displayed but formal definitions are harder to write. We list the major ones in this section.

(A) The CompCert memory model assumes that memory contains a set of blocks. Every block has at most 2^32 bytes. Thus, \( p \mapsto v \) should require \( p \) not to exceed the \( (2^{32} - \text{sizeof}(t)) \)-th byte in its block.

(B) In the Clight operational semantics, a load or store command requires the address to be properly aligned w.r.t. the C type. For example, 32-bit integers can only be loaded from 4-aligned addresses. We put size compatibility, alignment compatibility and nested field legality into the definition of \texttt{data.at}.

(C) In C, spaces are put among struct fields such that every field is ensured to be properly aligned. Also, spaces are put at the end of union fields such that an union occupies a fixed length of memory no matter which field is valid. We put these spaces in the definition of \texttt{data.at.rec}.

(D) It is better to write \( p \mapsto\text{InPair} \ (x, y) \) instead of \( p \mapsto\text{InPair} \ (x, y, t) \). We define \texttt{compact.prod} to be the retype of struct types; \texttt{compact.prod}\[X_1; X_2; \ldots ; X_n] is defined as \((X_1 \times X_2 \times \ldots \times X_n)\) instead of \((X_1 \times X_2 \times \ldots \times X_n \times \text{unit})\).

(E) C has many more elementary types.
6. Soundness Proofs

Unfolding equations and reroot equation can be proved straightforwardly from the recursive definition of data.at.rec and the definitions of field.at and data.at. In this section we focus on how to prove NestedLoadRule and NestedStoreRule sound.

As shown in Figure 6, NestedLoadRule can be proved by applying LoadRule. The only gaps are the following lemmas. (We will prove them in §6.2.)

Field-mapsto : For any legal nested field $\vec{F}$ of $t$, $p \mapsto u \vdash p \triangleright \vec{F} \mapsto u.\vec{F} * \top$.

Field-lvalue : If $[[e]]_t = p$ and $[[\vec{F}]]_r = \vec{F}$ then $P \vdash [e.\vec{F}]_t = p \triangleright \vec{F}$.

In contrast, NestedStoreRule is hard to prove. One natural approach is to split $p \mapsto u$ into a separating conjunction of $p \triangleright \vec{F} \mapsto u.\vec{F}$ and a residue. Then, $u.\vec{F}$ will be replaced with another value $v_0$ by a store command and we can get a data.at predicate again by putting the residue back, i.e.

$$p \mapsto u \vdash p \triangleright \vec{F} \mapsto u.\vec{F} * R \quad (18)$$

$$p \triangleright \vec{F} \mapsto v_0 * R \vdash p \mapsto u[v_0/\vec{F}]$$

Though this approach seems not complicated as a pen-and-paper proof, a proof script of it in a formal language like Coq cannot be as succinct. The main difference is that we have to define the residue $R$ in a formal way. In detail, it should be the result of digging a small hole inside a field.at predicate; although this could be defined by recursion on C-type structure, it would be much more complicated than data.at.rec.

We choose another approach to avoid this complicated dependently typed definition. We prove the nested store rule by ramification.

6.1 Background: Proof theory of ramification

The "Ramify" rule is proposed by Hobor and Villard [21]. Essentially, it is a special frame rule with an anonymous frame. A ramification premise (RP), $G \vdash L * (L' \rightarrow G')$, ensures that there exists a frame $F$ such that $G \vdash L * F$ and $L' * F' \vdash G'$.

6.2 Proof of nested load/store rule

In previous discussion, we use Field-mapsto and Field-lvalue to prove NestedLoadRule. Here, we use the following auxiliary Hoare rule, AuxStoreRule, to prove NestedStoreRule.

$$G \vdash p \mapsto * (p \triangleright v \rightarrow G')$$

$$P \vdash [e_1]_t = p \wedge [e_2]_r = v \wedge P' * G$$

$$\{ P \} e_1 = e_2 \{ P' * G' \}$$

This auxiliary rule can be proved by a combination of RamifyRule and StoreRule. And NestedStoreRule can be proved by instantiating $G$ with $p \mapsto u$ and instantiating $G'$ with $p \mapsto u[v/\vec{F}]$ (see Figure 7). The only gaps that remain are RP-mapsto and Field-lvalue.

$$R \mapsto p \mapsto u \vdash p \triangleright \vec{F} \mapsto u.\vec{F} *$$

$$(p \triangleright \vec{F} \mapsto v \rightarrow p \mapsto u[v/\vec{F}])$$

$$R \mapsto p \mapsto u \vdash p \mapsto \vec{F} *$$

$$(p \triangleright \vec{F} \mapsto v \rightarrow p \mapsto u[v/\vec{F}])$$
Field-mapsto is a direct corollary of RP-mapsto; Field-lvalue is straightforward from the definition of C-expressions’ lvalue; RP-mapsto can be proved by unfolding equation (3) and RP-field. Now, RP-field is all we need to prove NestedLoadRule and NestedStoreRule sound.

We prove RP-field by induction on the length of field path \( F \). The base case is trivial. The induction step is proved by RP-trans (see Figure 8).

In summary, the following diagram shows our process of proving NestedLoadRule and NestedStoreRule sound.

As in §4.3, we face some type mismatches in dependently typed proofs. For example, when we say \( u, F, [f] = u, F, f \), the types of both sides are provably equal but not convertibly equal. We use John Major equality (\( \text{JMeq} \)) to capture these equalities with type mismatch in our proofs. \( \text{JMeq} \{ A \ a \ (B \ b) \) says the types of \( a \) and \( b \), which are \( A \) and \( B \) (implicit arguments), are the same, and their values are the same as well.

6.3 Comparison between proof by ramification and proof by defining residue

Proving the Ramification Premise (RP-mapsto) is just proving (18) while filling the residue \( R \) with the separation logic formula \( (p, F) \rightarrow v \rightarrow p \rightarrow u[v/F] \). In fact, this is why we call RamifyRule a frame rule with anonymous frame. In our soundness proof, we do not need to explicitly construct the frame. Proving a RP with form \( G \vdash L \ast (L' \rightarrow G') \) implies the existence of a legal frame.

6.4 Use ramification to improve function call verification

Ramification is not only useful for soundness proofs. In fact, it is also helpful for function call verification. In §2.4, users still have to apply unfolding equations to verify the triple of a function call, though reroot equations can eliminate some of them.

Is it possible to verify triples of function calls without using unfolding equation, like verifying triples of load/store commands? The answer for the most general situations is no. A function in C may free a piece of memory. It may allocate a new piece of memory. Or, the same piece of memory may be tagged with different types, like \( \text{IntPair} \) and \( \text{int}[2] \), in pre and postcondition. As a consequence, it is impractical to write a single Hoare rule to handle all these different situations.

However, in some cases we can concisely prove a triple for function call by ramification. For example, Figure 9 uses RamifyRule, RP-field and reroot equations to prove the same sample triple in §2.4. No unfolding equation is used.

Although the proof in Figure 9 looks intimidating, the premises are easy to calculate from the conclusion and the function specification. This proof style can be used in many (but not all) practical cases. For example, ramification can be applied on function calls of the following function specification. This proof style can be used in many (but not all) practical cases. For example, ramification can be applied on function calls of the following function specification.
IH: \( p \xrightarrow{t} u \vdash p \xrightarrow{t} u, F \ast (p \xrightarrow{t} u, F[\nu]/[f]) \Rightarrow p \xrightarrow{t} u[u[F[\nu]/[f]]/\bar{F}] \)

Single Layer RP: \( p \xrightarrow{t} u, \bar{F} \vdash p \xrightarrow{t} u, \bar{F} \ast \left( p \xrightarrow{t} \nu \Rightarrow p \xrightarrow{t} u, \bar{F}[\nu]/[f]\right) \)

\[
\begin{align*}
\text{Single Layer RP: } & p \xrightarrow{t} u, \bar{F} \vdash p \xrightarrow{t} u, \bar{F}\ast \left( p \xrightarrow{t} \nu \Rightarrow p \xrightarrow{t} u[\bar{F}[\nu]/[f]]\right) \\
& \Rightarrow u, \bar{F}[\nu]/[f] = u[\bar{F}\nu]/[\bar{f}] = u[\bar{F}\nu] \\
& p \xrightarrow{t} u \vdash p \xrightarrow{t} \nu \Rightarrow p \xrightarrow{t} u[\bar{F}\nu] \\
\end{align*}
\]

\[\text{[RP-trans]}\]

Figure 8: Induction Step of RP-field’s Proof

\[
\begin{align*}
p \triangleright [f1] & \xrightarrow{\text{IntPair}} (y, x) = p \xrightarrow{\text{IntPair}} (y, x) \\
p \triangleright [f1] & \xrightarrow{\text{IntPair}} (x, y) = p \xrightarrow{\text{IntPair}} (x, y) \\
\{p \triangleright [f1] \xrightarrow{\text{IntPair}} (x, y)\} & \triangleright \text{int_pair_swap}(\&(p \rightarrow f1)) \{p \triangleright [f1] \xrightarrow{\text{IntPair}} (y, x)\} \\
p \xrightarrow{\text{ToyType}} ((x, y), l) & \vdash p \xrightarrow{\text{ToyType}} ((y, x), l) \ast \left( p \xrightarrow{\text{ToyType}} ((y, x), l) \Rightarrow p \xrightarrow{\text{ToyType}} ((y, x), l)\right) \\
\{p \xrightarrow{\text{ToyType}} ((x, y), l)\} & \triangleright \text{int_pair_swap}(\&(p \rightarrow f1)) \{p \xrightarrow{\text{ToyType}} ((y, x), l)\}
\end{align*}
\]

\[\text{[RamifyRule]}\]

Figure 9: Prove a triple for function call using ramification

\[
\forall a \ b \ c \ d, \\
\{p \xrightarrow{\text{IntPair}} (a, b) \ast q \xrightarrow{\text{IntPair}} (c, d)\} \\
\text{void int_pair_swap2(IntPair} \ast p, \text{IntPair} \ast q) \\
\{p \xrightarrow{\text{IntPair}} (b, a) \ast q \xrightarrow{\text{IntPair}} (d, c)\}
\]

7. Related Work and Discussion

Since separation logic was introduced, people have developed several systems using this program logic to reason about real world programming languages, like C and Java.

Affeldt et al. [?] , Krebbers [?] and Tuch et al. [?] formalize aggregate types into their separation logic predicates.

In these systems, Affeldt and Krebbers use Coq inductive types to define their representation type. Although inductive types are easier to reason (because of this, they do not need an explicit ranking to ensure acyclic property of C types), we choose not to do that but to use product type, sum type and list in Coq standard library for our users’ convenience.

None of these projects has NestedLoadRule or NestedStoreRule in their logic. As a result, when their users want to prove a triple like (1), they have to use unfold equations first.

Hip/Sleek [?] , JStar [?] and VeriFast [?] are automatic tools which can verify C or Java programs. In these tools, separation logic is used as the assertion language (practical usage of aggregate types are covered in these projects).

These tools do not have soundness proofs in languages like Coq, HOL or agda. Moreover, automatic tools cannot be complete in general cases, since even the proof theory of natural number arithmetics is incomplete. In comparison, we prove our extension of separation logic sound in Coq. We offer some automation but users can always prove and use their own domain specific theorems during verification.

8. Further Research and Conclusion

John Reynolds’s informal notation \( p \xrightarrow{t} (x, y) \) is very useful to anyone doing pencil-and-latex proofs in separation logic. Along with that notation, Reynolds assumes implicitly that one can load and store directly to individual fields, without expanding to a separating conjunction of primitive mapstos.

And Reynolds uses the standard tuple notation of mathematics.

But this notation, and these derived proof rules (NestedLoadRule, NestedStoreRule and Reroot equation), had not been formalized for use in a proof assistant, in part because it requires dependent types and in part because the syntax and operational semantics of aggregate types in real programming languages is rather complex. In this work we have shown how to do it — using a recursive function to define representation types and using ramification to prove related Hoare rules — and we demonstrate that it scales to the real C language.

Therefore, in the VST system (or in Charge! or Bedrock [?] if our construction were to be ported there), one can more easily construct highly reliable verifications of real programs with our extension.
Besides the specific problem we solved—handling aggregate types in separation logic—our techniques are relevant in other contexts.

First, Coq is not only a reliable proof system, but also a programming language. Coq users should care about how much time and memory it takes to build their proofs. Our design philosophy is that our automation system should provide convenient tactics for users, should display nice proof goals for better interaction with users, and should also be efficient enough to execute.

Second, our work offers an example of using Coq’s dependent types. The difficulties we faced and the solutions we provide are instructive for future Coq users. Coq’s use of (mutually) structurally recursive functions to define terms led to many convertibility problems that we had to work around. Our work can also serve as a useful test case in the design of other Coq-like proof assistants or programming languages.

Finally, our work demonstrates a practical application of ramification. The theory of of ramification looks scary because magic wand is involved in the ramification premise. However, ramification theory uses magic wand in a very limited way. RamifyRule turns out to be a special frame rule such that the frame can be anonymous. The transitivity, split, and frame rules of ramification enable the anonymous frame to be implicitly constructed combinatorially. When an explicit frame is hard to define, using ramification becomes a better choice.

References


