\label{sec:vst}

The Verified Software Toolchain (VST) ~\cite{appel:programlogics} is a series of Coq modules which reason about those C programs that can be compiled with the CompCert compiler~\cite{leroy:compcert}. One of VST's modules, ``Floyd'', is a separation-logic based engine to help users

verify the full functional correctness of their concrete C programs. Written in Ltac and Gallina, Floyd prefers expressibility and completeness to automation. VST’s modules are machine-checked and interlocked, so there are no ``gaps'' in the end-to-end certified results; accordingly all of the rules employed by Floyd have been proved sound with respect to the underlying semantics used by CompCert. s

%\vspace\*{-0.75ex}

\subsection{Marking a graph in VST}

\label{sec:vstgraphmark}

%\vspace\*{-0.75ex}

\input{do\_scan\_listing.tex}

In Figure~\ref{fig:markgraph} we put the code and proof sketch of the classic \li{mark} algorithm that visits and colors every reachable node in a heap-represented graph. The \li{mark} algorithm is good to start with because it is complex enough to require some care to verify while being simple enough that the invariants are straightforward. In \S\ref{sec:application} we will discuss more complex examples that \emph{e.g.} add/change/remove edges and/or vertices.

The code in Figure~\ref{fig:markgraph} is written in Clight~\cite{blazy:clight}, an input language to the CompCert certified compiler~\cite{leroy:compcert}, which compiles our code exactly as written.

The paper-format verification sketch for \li{mark} in Figure~\ref{fig:markgraph} is extracted from

a Floyd proof in VST, with only minor cleanup to aid the presentation.

Accordingly, there is an unbroken certified chain from our specification of \li{mark} all the way to the assembly code.

The specification we certify (lines \ref{code:markstart} and \ref{code:markend}) is

\vspace\*{-1ex}

\[

\{\p{graph}(\li{x},\gamma)\}~\li{mark(x)}~\{\exists \gamma'.~ \p{graph}(\li{x},\gamma') /| \m{mark}(\gamma, \li{x}, \gamma')\}

\vspace\*{-1ex}

\]

The specification is for full functional correctness, stated using \emph{mathematical} graphs~$\gamma$; until \S\ref{sec:mathgraph} consider $\gamma$ to be a function that maps a vertex $v \in V$ to triples $(m,l,r)$, where $m$ is a ``mark'' bit (0 or 1) and $\{l,r\} \subseteq V \uplus \{0\}$ are the neighbors of $v$.

The \emph{spatial} \p{graph} predicate describes how the mathematical graph $\gamma$ is implemented in the heap. Until~\S\ref{sec:spacegraph} it is enough to know that \p{graph} satisfies the fold/unfold relationship in

equation~\eqref{eqn:bigraphintrofoldunfold}, located just under the code in Figure~\ref{fig:markgraph}.

This fold/unfold relationship deserves attention.

First, as we explain in~\S\ref{sec:fixpointfail}, it is probably a mistake to write~\eqref{eqn:bigraphintrofoldunfold} as a definition using $\stackrel{\Delta}{=}$ rather than as a biimplication using $<=>$. Second, \eqref{eqn:bigraphintrofoldunfold} uses the ``overlapping conjunction'' $\ocon$ of separation logic; informally $P \*\* Q$ means that $P$ and $Q$ may overlap in the heap (\emph{e.g.}, nodes in the left subgraph can also be in the right subgraph or even be the root $x$). The presence of the unspecified sharing indicated by the $\ocon$ connective is exactly why graph-manipulating algorithms are so hard to verify (\emph{e.g.}, it is hard to apply the \infrulestyle{Frame} rule). The standard semantics of the separation logic connectives used in this paper are in Figure~\ref{fig:seplogsem}.

Third, \eqref{eqn:bigraphintrofoldunfold} illustrates how industrial-strength settings complicate verification. Lines~\mbox{\ref{code:nodedefstart}--\ref{code:nodedefend}} define the data type \li{Node} used by \li{mark}. The \li{\_Alignas($n$)} directives tell CompCert to align fields on $n$-byte boundaries. As explained in~\S\ref{sec:goodgraph}, this alignment is necessary in C-like memory models to prove fold-unfold \eqref{eqn:bigraphintrofoldunfold}, which is why \eqref{eqn:bigraphintrofoldunfold} includes an alignment restriction $x~\mathsf{mod}~16 = 0$ and an existentially-quantified ``blank'' second field for the root $x \mapsto m,-,l,r$.

%{\color{magenta}(In our Floyd proofs the alignment restriction and blank second field are nicely hidden ``behind the scenes''.)}

Notice that the postcondition of \li{mark} is specified \emph{relationally}, \emph{i.e.} $\{\exists \gamma'.~ \p{graph}(\li{x},\gamma') /| \m{mark}(\gamma, \li{x}, \gamma')\}$ instead of \emph{functionally}, \emph{i.e.} $\{\p{graph}\big(\li{x},\m{mark}(\gamma, \li{x})\big)\}$. In the first case $\m{mark}$ is a relation that specifies that~$\gamma'$ is the result of correctly marking~$\gamma$ from~\li{x}, whereas in the second $\m{mark}$ is a function that \textbf{computes} the result of marking~$\gamma$ from~\li{x}. A relational approach is preferable for both theoretical and practical reasons.

Theoretically, relations are preferable because they are more general. For example, relations allow ``inputs'' to have no ``outputs'' (\emph{i.e.} be partial) or alternatively have many outputs (\emph{i.e.} be nondeterministic). Our graph \li{copy} algorithm is specified nondeterministically to avoid specifying how \li{malloc} allocates fresh blocks of memory. Relations are also more compositional than functions.

We take advantage of this compositionality by using $\m{mark}(\gamma,x,\gamma') /| \ldots$ to specify both our ``spanning tree'' and ``graph copy'' algorithms in~\S\ref{sec:application}, which also mark nodes while carrying out their primary tasks.

\begin{figure}

\[

\begin{array}{lcl}

\sigma |= P \* Q & \defeq & \exists \sigma\_1, \sigma\_2.~ \sigma\_1 \oplus \sigma\_2 = \sigma /| \null \\ && ~~ (\sigma\_1 |= P) /| (\sigma\_2 |= 2)\\

[-2pt]

\sigma |= P \*\* Q & \defeq & \exists \sigma\_1, \sigma\_2, \sigma\_3.~ \sigma\_1 \oplus \sigma\_2 \oplus \sigma\_3 = \sigma /| \null \\ && ~~ (\sigma\_1 \oplus \sigma\_2 |= P) /| (\sigma\_2 \oplus \sigma\_3 |= Q) \\

[-2pt]

\sigma |= P --\* Q & \defeq & \forall \sigma\_1, \sigma\_2.~ \sigma\_1 \oplus \sigma = \sigma\_2 /| \null \\ && ~~

(\sigma\_1 |= P) => (\sigma\_2 |= Q) \\

[-2pt]

\sigma |= P --o Q & \defeq & \exists \sigma\_1, \sigma\_2.~ \sigma\_1 \oplus \sigma = \sigma\_2 /| \null \\ && ~~

(\sigma\_1 |= P) /| (\sigma\_2 |= Q)

\end{array}

\]

\vspace\*{-1.5em}

\caption{Separation logic connectives; $\oplus$ is the join operation on states, usually some kind of disjoint union on heaps}

\label{fig:seplogsem}

\vspace\*{-1em}

\end{figure}

Practically, defining computational functions over graphs is painful and sometimes overcomplicated in a proof assistant like Coq. For example, Coq requires that all functions terminate – a nontrivial proof obligation over cyclic structures like graphs – but our verification of \li{mark} is only for partial correctness. Defining relations is much easier because \emph{e.g.} one can use quantifiers and avoid proving termination.

The $\m{mark}$ and $\m{mark1}$ relations we use are defined straightforwardly at the bottom of Figure~\ref{fig:markgraph}.

Turning to the body of the verification (lines~\ref{code:inmark}--\ref{code:outmark}), readers may already have noticed our new notation: blocks of proof sketch bracketed by the symbols $\searrow$ and $\swarrow$, such as lines~\ref{code:beforerootmark}--\ref{code:afterrootmark}. We call a bracketed set of lines like this a ``localization block''; localization blocks were inspired by our new \li{localize} $\searrow$ and \li{unlocalize} $\swarrow$ tactics in Floyd (\S\ref{sec:vst}).

The intuitive idea is that we zoom in from a larger ``global'' context to a smaller ``local'' one. After verifying some commands locally to arrive at a local postcondition, we zoom back out to the global context. Although we do not see this in Figure~\ref{fig:markgraph}, localization blocks can safely nest.

In lines~\ref{code:beforerootmark}--\ref{code:afterrootmark}, imagine unfolding the \p{graph} predicate in line~\ref{code:globalbeforerootmark} using equation \eqref{eqn:bigraphintrofoldunfold} and then zooming in to the root node \li{x} for lines~\ref{code:beforerootmark}--\ref{code:afterrootmark}, before zooming back out in line~\ref{code:globalafterrootmark}.

To define localization blocks formally we need to first understand the \infrulestyle{Frame} and \infrulestyle{Ramify} rules.

\subsection{Frames and ramifications are localizations}

%\label{sec:localizations}

The key rule of separation logic is \infrulestyle{Frame}~\cite{rey02}:

\[

\infrule{Frame}

{\{ P \} ~ c ~ \{Q \}}

{\{P \* F \} ~ c ~ \{ Q \* F \}}

{\begin{array}{c}F \textrm{ ignores } \MV(c) \end{array}} \qquad \qquad

\vspace{-0.75ex}

\]

\infrulestyle{Frame} is important because it enables local verifications. That is, a verifier can focus on the portions of the heap that are relevant to command $c$ and ``frame away'' the rest. The side condition ``$F \textrm{ ignores } \MV(c)$'' relates to modified program variables and will be discussed in \S\ref{sec:freevars}.

Hobor and Villard observed that \infrulestyle{Frame} is restrictive because it forces verifiers to split program assertions into syntactically $\*$-separated parts~\cite{hobor:ramification}. In particular, this is hard to ensure when verifying programs that manipulate data structures with intrinsic unspecified sharing, as seen in DAGs and graphs. Hobor and Villard proposed the \infrulestyle{Ramify} rule to circumvent this rigidity:

\vspace{-1.5ex}

\[

\infrule{Ramify}

{\{L\_1\} ~ c ~ \{L\_2\} \\ G\_1 |- L\_1 \* (L\_2--\* G\_2)}

{\{G\_1\} ~ c ~ \{G\_2\}}

{\begin{array}{c}(L\_2 --\* G\_2) \\ \textrm{ignores} \\ \MV(c) \end{array}} \qquad \qquad \qquad

%{$\begin{array}{l}\m{fv}(Q --\* R') \cap \null \\ \m{modif}(c) = \emptyset\end{array}$} \qquad \qquad \qquad

\vspace{-1.5ex}

\]

That is, we can verify a ``global'' specification $\{G\_1\}~c~\{G\_2\}$ by combining a ``local'' specification $\{L\_1\}~c~\{L\_2\}$ with a \emph{ramification entailment} $G\_1 |- L\_1 \* (L\_2--\* G\_2)$. This entailment uses the ``magic wand'' operator $--\*$ of separation logic\footnote{$--\*$ is the adjunct of $\*$, \emph{i.e.} $(P \* Q |- R) <=> (P |- Q --\* R)$.} to express a notion of ``substate update'': inside $G\_1$, replace $L\_1$ with $L\_2$ to reach $G\_2$. Essentially the ramification entailment ensures that a change in state specified locally fits properly into its global context. In exchange for proving the ramification entailment, a verifier can use \infrulestyle{Ramify} at any time, \emph{i.e.} they need not worry about syntactically matching their assertions with the $\*$ in the \infrulestyle{Frame} rule. Although the ramification entailments can appear difficult, Hobor and Villard observed that in many practical cases they can be handled easily using a ``ramification library''.

We are now ready to give a formal meaning to the ``localization'' pattern employed in Figure~\ref{fig:markgraph}. When we write:

\vspace{-4ex}

\begin{lstlisting}

// $\label{code:prelocal}\{ G\_1 \}$

// $\label{code:inlocal}\searrow \{ L\_1 \}$

$\ramify(i)$ $c\_1$; ... ; $c\_n$;

// $\label{code:outlocal}\swarrow \{ L\_2 \}$

// $\label{code:postlocal}\{ G\_2 \}$

\end{lstlisting}

\vspace{-1.5ex}

we mean apply \infrulestyle{Ramify} with $G\_1 |- L\_1 \* (L\_2 --\* G\_2)$.

An advantage of this notation is crystal clarity on the predicates used in the ramification entailment. For convenience, the optional $\ramify(i)$ specification can reference an equation or lemma number that solves the ramification entailment. For example, in Figure \ref{fig:markgraph} line \ref{code:markram2} references Equation \eqref{lem:updategraphnode} whereas we omit $\ramify$ around line \ref{code:markram1} since the heap is unchanged and so the entailment is straightforward. To save vertical space we can compress the line pairs \ref{code:prelocal}--\ref{code:inlocal} and \ref{code:outlocal}--\ref{code:postlocal}

to the single lines $\{ G\_1 \} \searrow \{ L\_1 \}$ and $\{ G\_2 \} \swarrow \{ L\_2 \}$ without sacrificing clarity.

Hobor and Villard pointed out that \infrulestyle{Ramify} implies \infrulestyle{Frame} (modulo the modified program variables issue we fix in \S\ref{sec:freevars}), meaning that our notation can clarify uses of \infrulestyle{Frame} as well. This is particularly useful in multi-line contexts with nontrivial $F$, for which the current popular notation to express \infrulestyle{Frame} involves a liberal use of ``\ldots'', \emph{e.g.}:

\vspace{5pt}

\begin{minipage}{.19\textwidth}

Old notation:

\begin{lstlisting}

// $\{ P\_1 \* F\_1 \* F\_2 \* F\_3 \}$

$c\_1$;

// $\{ P\_2 \* \ldots \}$

$c\_2$;

// $\{ P\_3 \* \ldots \}$

$c\_3$;

// $\{ P\_4 \* F\_1 \* F\_2 \* F\_3 \}$

\end{lstlisting}

\end{minipage} \vline ~~~

\begin{minipage}{.2\textwidth}

New notation:

\begin{lstlisting}[numbers=none]

// $\{ P\_1 \* F\_1 \* F\_2 \* F\_3 \} \searrow \{ P\_1 \}$

$c\_1$;

// $\{ P\_2 \}$

$c\_2$;

// $\{ P\_3 \}$

$c\_3$;

// $\{ P\_4 \* F\_1 \* F\_2 \* F\_3 \} \swarrow \{ P\_4 \}$

\end{lstlisting}

\end{minipage}

\vspace{-0.75ex}

\subsection{Localizations in VST with \li{localize} and \li{unlocalize}}

\label{sec:vstlocalunlocal}

\vspace{-0.75ex}

Floyd presents users with a ``decorated program'' visualization for Hoare proofs, in which users work from the top of the program to the bottom even though the formal proof is maintained as applications of inference rules. For example, suppose the proof goal is $\triple{P\_1}{c\_1\li{;}c\_2}{P\_5}$ and and the user applies a Hoare rule for $c\_1$, \emph{e.g.}~$\triple{P\_1}{c\_1}{P\_2}$. Floyd will then automatically apply the \infrulestyle{Sequence} rule and show the user $\triple{P\_2}{c\_2}{P\_5}$ as the remaining goal.

When the user is in the middle of a verification, the decorated program is partially done (\emph{i.e.} the proof is finished from the top to ``the current program point'') and the inference tree is also partially done (\emph{i.e.} with holes that are represented by the remaining proof goals in Coq).

We wish to preserve this ``decorated program'' view while extending Floyd to support ramification. Our task therefore is to construct a proof in Coq's underlying logic that allows a localization block to be constructed in this manner---that is, we wish to enter a localization block without requiring the user to specify the ``exit point'' in advance. The engineering is tricky because the proof Floyd is constructing (\emph{i.e.} applications of inference rules) has holes in places where the user's ``top to bottom'' view of things has not yet arrived.

\begin{figure}[t]

\begin{minipage}{.1\textwidth}

\begin{lstlisting}

$\{\ \ P\_1 \ \ \}$

c1

$\{\ \ P\_2 \ \ \}\label{code:localglobalin}$

$\searrow \{\ \ P\_3 \ \ \}\label{code:locallocalin}$

c2;

$~~\,\{\ \ P\_4 \ \ \}\label{code:localsndcmd}$

...

\end{lstlisting}

\end{minipage} \vline ~

\begin{minipage}{.11\textwidth}

\begin{lstlisting}[numbers=none]

$\{\ \ P\_1 \ \ \}$

c1

$\{\ \ P\_2 \ \ \}$

$\{\ \ ?F \* P\_3 \ \}$

c2;

$\{\ \ ?F \* P\_4 \ \}$

...

\end{lstlisting}

\end{minipage} \vline ~

\begin{minipage}{.11\textwidth}

\begin{lstlisting}[numbers=none]

$\{\ \ P\_1 \ \ \}$

c1

$\{\ \ P\_2 \ \ \}$

$\searrow \{\ \ P\_3 \ \ \}$

c2;

$\,\{\ \ P\_4 \ \ \}$

c3;

$\swarrow\{\ \ P\_5 \ \ \}$

$\{\ \ P\_6 \ \ \}$

...

\end{lstlisting}

\end{minipage} \vline ~

\begin{minipage}{.11\textwidth}

\begin{lstlisting}[numbers=none]

$\{\ \ P\_1 \ \ \}$

c1

$\{\ \ P\_2 \ \ \}$

$\{\ \ ?F \* P\_3 \ \}$

c2;

$\{\ \ ?F \* P\_4 \ \}$

c3;

$\{\ \ ?F \* P\_5 \ \}$

$\{\ \ P\_6 \ \ \}$

...

\end{lstlisting}

\end{minipage}

%\vspace{-ex}

\caption{Front and back ends of \li{localize} and \li{unlocalize}}

\label{figure:backend}

\end{figure}

Figure~\ref{figure:backend} has four partially-decorated ``proofs in progress'', from both the user's (front end) and Floyd's (back end) points of view. In the first column, from the user's point of view, they saw the assertion $P\_2$ (line~\ref{code:localglobalin}) and decided to use the \li{localize} tactic to zoom into $P\_3$ (line~\ref{code:locallocalin}). They then applied some proof rules to move past $c\_2$ to reach the assertion $P\_4$ (line~\ref{code:localsndcmd}). At this point, Floyd does not know when the corresponding \li{unlocalize} tactic will execute, so it does not know which commands will be inside the block or what the final local and global postconditions will be.

Accordingly, the \li{localize} tactic builds an incremental proof in the underlying program logic by applying \infrulestyle{Frame} with an uninstantiated metavariable.

The second column of Figure~\ref{figure:backend} shows the back end with the unknown frame $?F$, which will eventually be instantiated by \li{unlocalize}.

In the third column, the user has advanced past $c\_3$ to reach the local postcondition $P\_5$ and now wishes to \li{unlocalize} to~$P\_6$. Afterwards, the internal state looks like the fourth column, and so to a first approximation, \li{unlocalize} can instantiate $?F$ with $P\_5 \wand P\_6$. In truth, $?F$ is chosen more subtly to properly handle both existential variables and modified program variables; \li{unlocalize} then automatically simplifies the goals to present a cleaner interface to the user. These transformations require some additional theory given in \S\ref{sec:localizations}.

\subsection{Additional verified examples}

\label{sec:application}

In addition to \li{mark} for graphs, we have also verified the same program for DAGs.

More interestingly, we have verified two algorithms that modify the link structure:

\li{spanning} for graphs, which prunes a graph into its spanning tree, and \li{copy}

for graphs, which makes a structure-preserving copy. For space reasons we put

\li{spanning} and \li{copy} in Appendix~A\hide{\ref{apx:spanning}}.