1 OVERVIEW AND MOTIVATING EXAMPLE

In this section, we shall provide the motivation for our paper, an overview of our technique and illustrate the method on a motivating example.

1.1 Problem and formulation

Let us consider a set of pointer fields $\overline{p}$ and a recursive definition of a unary predicate or function $R(x)$ defined using least fixpoints over $\overline{p}$. Typical functions include properties such as "x points to a linked list segment ending at the location z", "the length of the list pointed to by x", "x points to a binary search tree", "the set of keys stored in the tree pointed to by x", "the heaplet defined by the tree pointed to by x", etc. Consider a Hoare triple of the form $@pre(\overline{x}, \overline{p}, R)\ S\ @post(\overline{x}, \overline{p}, R)$: the pre and postconditions use the recursively defined predicate/function R.

One simple approach is to capture the precondition using logic and use the *frame rule* to reason soundly (but incompletely) about the post-state— i.e., we can simply ignore the definition of R on the transformed heap and infer that $R'(x)$ holds if it held in the pre-state and the modified portion of the heap did not intersect with the underlying heaplet of $R(x)$. This is, in practice, a very *convenient and simple* reasoning that often works and is one of the foundational ideas that separation logic facilitates [?]. However, vanilla frame reasoning can be incomplete, as we shall argue through our motivating example.

The focus of this paper is on the generation of precise verification conditions for basic blocks that do not involve function calls. For basic blocks that involve function calls, our recommendation is to use frame reasoning, *à la* separation logic.

Let us now consider basic blocks that do not involve function calls. We would like to generate precise verification conditions in such cases. There are several approaches in the literature that argue for this: for example the Grasshopper suite of tools handle such blocks accurately for certain logics [?], and there is work on expressing weakest preconditions in separation logic for such blocks using the magic wand [?]; see section on related work. The goal of this paper is to accurately formulate verification conditions for very expressive logics (FO+*lfp*) that can also be reasoned with effectively, especially in the context of decidable logics.

One precise formulation of the verification condition is of the form:

$$\big(@pre(\overline{x}, \overline{p}, R) \wedge T(\overline{x}, \overline{x}', \overline{p}, \overline{p}')\big) \Rightarrow @post(\overline{x}', \overline{p}', R')$$

where T describes the effect of the program on the stack and the heap, describing how the scalar variables $\overline{x}$ and pointer-fields $\overline{p}$ have evolved to the $\overline{x}'$ and $\overline{p}'$ respectively. Most importantly, the above requrires *new* recursive definitions R' that are formulated by replacing $\overline{p}$ in the definition of R with $\overline{p}'$.

Though the above is a precise formulation of the verification condition, it has several drawbacks. First, there is a heaplet H modified by the program, and the formula will have conjuncts of the form $\forall y \notin H. p'(y) = p(y)$, for every $p \in \overline{p}$. This introduces universal quantification, which is harder to reason with automatically. However, for basic blocks that do not involve function calls, i.e, when H is finite, we can map this into a decidable quantifier free logic (modeling p as an array and p' as an update to the array). Second, the new definition of R' depends on $\overline{p}'$, which in turn depends on the various constraints introduced by the basic block. For example, pointer fields may change depending on complex properties involving the data elements stored in the heap. Reasoning with R' automatically (which involves least fixpoints), coupled with such constraints, is daunting.

Surely, there must be a simpler formulation of the verification condition! Small changes to the heap do cause global changes and can dramatically affect the valuation of recursively defined predicates/functions, which are global. But surely, the effect on the semantics of R (changing into R') must be expressible in a simpler localized fashion.

1.2 Overview of method

Delta Logics: In this paper, we describe a class of logics, called *delta-logics*, that are logics for writing verification conditions of such basic blocks and are precisely meant to address the above issues. In particular, formulations in delta-logics will avoid the need for *two* diferent recursive definitions R and R'. Instead, both R and R' will be expressed as the same recursive definition, but parameterized using different sets of first-order variables (as opposed to being parameterized over two different sets of first-order *functions*, $\vec{p}$ and $\vec{p'}$ as above).

Formulae in delta-logics are Boolean combinations of two distinct kinds of formulae: one kind, called *delta-specific* formulae, strictly talk about the modified portion of the global heap (identified by a bounded set of locations Δ) without using any recursive definitions; the other kind, called *contextual/context-logic* formulae, strictly talk about the unbounded portion excluding Δ using recursive definitions. A set of first-order interface variables are used to communicate information between Δ and the rest of the heap (its 'context'). In particular, a recursive definition R over the unbounded context is *parameterized* over a set of first-order communication variables P^R, where P^R summarizes the values of R within Δ. These variables themselves can, of course, depend on the value of R outside Δ as well, setting up mutually dependent constraints.

Advantages of reasoning with delta logics: Expressing verification conditions in delta-logics has a distinct advantage in automated reasoning. We formulate verification conditions in delta logics in the following manner (see Figure **??**) on the right. We can view the program's transformation of pre-heap to the post-heap a static model consisting of three different submodels: one is the context heap, the second is the pre-heap restricted to Δ, and the third is the post-heap restricted to Δ. Note that there is only one context heap as it does not change, and this context heap is infinite, in general. However, the other two heaps are finite. The recursive properties of the heap are then defined on the context-heap, parameterized with the communication variables P_R with the second heap for expressing properties of the pre-heap, and parameterized over the communication variables P'_R with the third heap for expressing properies of the post-heap.

The key advantage of the above model is that two of the submodels (both over Δ) are finite and reasoning about them and the data-fields accessible from them can be automated using standard SMT solvers. The context heap is the single unbounded submodel that poses automation challenges. In this paper, we exploit this simplicity of dealing only with one infinite submodel (as opposed to the naive formulation, which would have two infinite submodels that are given as being related at finitely many locations, with no other structural advantage to exploit) to build new powerful classes of decidable logics over lists and list-measures. A decidable contextual logic for lists is one of the main contributions of this paper (see below).

Expressing verification conditions in Delta logics. The key idea here is to set up parameterized sets of recursive definitions with two sets of parameters P_R and P'_R, as described above, expressing the precondition using one and the postcondition using the other. The change to the heaplet can be described in quantifier-free FOL as usual.

The main challenge here is to express the precondition and postcondition using Delta logics. In order to do this, we prove a general technical result for Delta logics, called the *separability theorem* in Section **??**. This theorem shows that any quantifier-free FO formula with recursive definitions (with *lfp* semantics) can be expressed in delta-logics, i.e., as a boolean combination of delta-specific formulae and contextual formulae. We prove this by setting up communication between the two portions: the Δ portion of the heap and its context. As explained earlier, the context is handled/described using (now parameterized) recursive functions/predicates and the Δ portion, being finite, ensures the preservation of the semantics of the original recursive definition by 'unfolding' it across Δ. This unfolding would stop at the 'boundary' of Δ and the context, which is where we use the interface variables to communicate the valuations across these disjoint portions. These are the mutual constraints that were referred to earlier in this section. However, it turns out that a new set of recursive definitions and communication variables involving a notion of *rank* for each (parameterised) recursive definition is needed to accurately capture least fixpoints, for the paramterized recursive functions are now merely defined on the context and the *lfp* semantics cannot merely preserved by a naive unfolding of the definition, even if the portion on which the unfolding is done is finite (for instance, the familiar definition of lists as either being *nil* or pointing to a location that points to a list is incorrect with just fixpoint semantics, as it would then be acceptable to call a cycle a list as well). These ranks can, however, be constrained to be *bounded integers* as opposed to ordinals (though the heap is infinite).

Thus, we have argued that delta-logics are a better alternative in which to generate VCs, and to this end we contribute a powerful decidable logic of lists and list-measures that arises as a consequence of the investigation of the contextual logic of lists and list-measures. We argue that this is because delta-logics explicitly delineate the changed portion of the heap by only allowing recursive definitions to be parameterized over first-order communication variables and avoid introducing quantification, making them easier to reason with.

1.3 Motivating Example

We shall now broadly illustrate our method on a motivating example. Let us consider the following Hoare triple with pre/post conditions written in FO+*lfp*:

$$\{list(x) \wedge list(w) \wedge y \notin hlist(w)\}\ \texttt{y.next := w}\ \{list(x)\}$$

where $list(x)$ means x points to a list, and in which case, $hlist(x)$ stands for the set of locations in that list.

It is easy to see that this Hoare Triple is valid. First notice that frame reasoning argues that w continues to point to a list. There are then two cases: if y does not belong to the list pointed to by x, then since y is not in the heaplet, frame reasoning would prove that x continues to point to a list. However, in the case when y does belong to the list pointed to by x, we cannot use frame reasoning to immediately conclude that x will continue to point to a list. However, notice that x will point to a list in this case as well, since y will point to w which points to a list. Therefore, frame reasoning *alone* will not be sufficient to conclude the postcondition from the precondition and the program transformation at this point. The solution in separation logic would be to identify the frame that does not change, separate the global heap into the correct frames, and then apply frame reasoning. Our solution handles this problem (automatically) by writing the VC in delta-logic, the formative ideas for which have been explained above.

In general, given a basic block without function calls we will identify at least the set of locations whose pointer fields are accessed to be Δ. In this case we do the same, identifying $\Delta = \{y\}$; x and w are then named locations in the context. As discussed above. to write the VC in delta-logic we shall first use the separability theorem (Section ??) to write the pre- and and post-conditions in delta-logic. For the precondition, this yields:

$$\left(list^P(x) \wedge list^P(w) \wedge y \notin hlist^P(w)\right) \wedge \beta$$

where $list^P()$ and $hlist^P()$ are the parameterized versions with the parameter set P, and β contains the contraints that 'unroll' the definition of these recursive predicates across the delta, constraints that describe the relationship between the parameters and the recursive definitions and finally the rank constraints (with one definition of a rank for each recursive predicate, also parameterized, say $Rank^P_{list}$ and $Rank^P_{hlist}$)— all of which have been explained above. The parameter set consists mostly of one parameter per recursive definition per variable in Δ and so for this example we would have parameters such as $LIST^y$ and $HLIST^y$ which are first-order variables that are meant to represent the valuations of the corresponding predicates at that location in Δ. Since the ranks are also recursive definitions, we would also have parameters such as $LIST\_RANK^y$ and $HLIST\_RANK^y$.

Note: As we shall explain later, this means that our parameterised predicates essentially encode the structure of the context in that they provide a valuation on all locations but Δ given a valuation on Δ—after which, of course, we ask the same question with the post-state and attempt to reason about the relationship between the two.

The constraints that are part of β might look like:

$$list^P(y.next) \Rightarrow LIST^y \wedge \Big(LIST\_RANK^y = Rank^P_{list}(y.next) + 1\Big)$$

which says that if $y.next$ is a true list (which the *lfp* definition $list^P$ ensures), then so is y and morover, its 'rank' corresponding to the $list()$ predicate is greater than that of $y.next$. This, as we have explained earlier, is to ensure that cycles do not pose a problem to assigning true valuations: so if there were further locations upstream (on the pointer) of y, they would also be lists and their ranks would be greater than the locations after them. These are the constraints in β from 'unfolding' the definitions across Δ:

$$(a = b.next \wedge LIST^a) \Rightarrow \big(LIST^b \wedge \big(LIST\_RANK^b = LIST\_RANK^a + 1\big)\big) \qquad (a, b \in \Delta)$$

Of course, we only have one location in Δ here so these constraints would not be present for this particular example. This gives us a ranking across Δ, and avoids wrong markings on cycles within it, and the *lfp* avoids this problem outside Δ. But we must also avoid it for cycles that are shared between these disjoint portions. These are the parameter-predicate relationship constraints, and they take the form of the parameterized predicates themselves, specifically, the ranks.

Notice that in all of this the next pointer is only used on Δ and the recursive predicates are only applied to locations in the context (indeed, the domain of the parameterized recursive definitions is the complement of Δ, and does not contain Δ at all), and we shall later do the same for what we might call Δ', which is the same set of locations but whose model has the post-state's next pointers. However, these are not exactly a boolean combination of delta-specific and contextual formulae due to the prescence of terms such as $list^P(y.next)$ which inextricably applies a recursive predicate to an arbitrary location (possibly) addressed only via a location in Δ, i.e, the exact location might depend on the valuation of the next pointer on Δ, and consequently so would the valuation of $list^P()$ on it. That would make it a term that is neither in contextual logic due to this, and not a delta-specific term either due to the prescence of a recursive predicate. In our complete formalisation, we avoid this by introducing a set of parameters for so-called 'boundary' locations, which are of the form $y.next \notin \Delta$ where $y \in \Delta$. This then completely separates these terms, making the parameters the only shared variables (these are the total set of communication/interface variables) between Δ and the context, and also makes the translation a delta-logic formula.

Therefore, we get the following VC for the given Hoare Triple in delta-logic:

$$\begin{aligned}
&\Big(\Big(list^P(x) \wedge list^P(w) \wedge y \notin hlist^P(w)\Big) \wedge \beta(P)\Big) \\
&\wedge \left(\bigwedge_{d \in \Delta \setminus \{y\}} (next'(d) = next(d)) \wedge (next'(y) = w) \wedge (y' = y) \wedge (x' = x) \wedge (w' = w) \right) \\
&\Rightarrow \Big(list^{P'}(x) \wedge \beta(P')\Big)
\end{aligned}$$

where, again, not all terms are nontrivial because Δ is singleton in this case and we have used the same name *next* and *next'* to model the next pointers. Notice that we're using the same recursive predicates to describe the pre- and post-states, but two *different* sets of parameters.

The problem now reduces to solving for the validity of this VC. In the decidability section (Section **??**), we show exactly how to build an effective procedure to decide VC validity of lists with various measures. The problem really is of finding a decision procedure for the solution of contexual formulae. We do this by the ‘summarization’ mentioned earlier. We observe that the context can be abstracted by a finite set of locations, and the list segments between them, if they exist, summarized in a way that can be encoded in SMT. We shall describe this in further detail in Section **??**