Replication of a Deductive Synthesizer for Programs with Pointers

Serge Johanns and Marieke Huisman

University of Twente, Enschede, The Netherlands
s.i.johanns@student.utwente.nl
m.huisman@utwente.nl

Abstract. Deductive synthesis aims to synthesize a program from a formal specification, without requiring the programmer to write the implementation or any additional annotations. Polikarpova and Sergey developed SuSLik, a deductive synthesizer for programs with pointers, using Synthetic Separation Logic. In this replication study we construct a similar deductive synthesizer, DeSyL, based on the methodology from the SuSLik paper while making some minor changes and resolving some irregularities. We compare the results of our implementation to the results of the SuSLik paper, and discuss the implications of our findings.

Keywords: Deductive synthesis · Separation logic · Replication

1 Introduction

Deductive synthesis is an approach inspired by Automated Theorem Proving to address the issue of the additional work for deductive verification of programs. Instead of requiring the programmer to write the implementation, the formal specification, and the annotations to guide the verification process, deductive synthesis only requires the specification and then synthesizes the implementation autonomously, requiring no further annotations. Deductive synthesis does increase the computational complexity of the proof steps, since the synthesis algorithm is required to perform all proof steps autonomously. However, a strategically designed logic system can significantly reduce this complexity by guiding the synthesis algorithm.

Seperation Logic (SL) is a formal logic for verifying programs with pointers and dynamic memory allocation that is designed to be amenable to automated reasoning [7]. Polikarpova and Sergey [6] combine these developments in a theoretical approach for deductive synthesis with Separation Logic and their implementation SuSLik. Using a synthesis-focused form of SL called Synthetic Separation Logic (SSL), SuSLik is capable of efficiently synthesizing programs with pointers, pointer arithmetic, and dynamic memory allocation from a formal specification, without requiring any additional annotations. Furthermore, SuSLik is capable of generating cyclic proofs, permitting the synthesis of structurally recursive programs. In order to better understand SuSLik, we replicated

it in DeSyL[1]. In this paper we will first outline SuSLik's methodology itself, then outline our implementation DeSyL and how it compares to SuSLik, discuss the results of our implementation on benchmark problems and discuss how they compare to SuSLik's results, and finally discuss areas for future work.

2 SuSLik methodology

At the core of SuSLik's methodology is the Synthetic Separation Logic (SSL), an extended ruleset based on Separation Logic [6]. We discuss some relevant rules in detail in Section 3, but the fundamental concept is that SSL rules transform the pre- and/or postcondition of a specification and possibly add a program statement that achieves this transformation. The rules are designed to be amenable to automated reasoning, and use the structure of the specification to guide the synthesis algorithm towards a correct implementation. The synthesis algorithm itself uses a simple depth-first search through the proof space, applying the SSL rules to the specification, backtracking when the rules cannot transform the specification further, and terminating when the specification is transformed into one that is trivially satisfied by the empty program. The program statements that were synthesized by the SSL rules to achieve the trivially satisfied specification must then form a correct implementation of the original specification.

Of the SSL rules used in SuSLik, some rely on a pure reasoning system to reason about the specification's pure constraints, which are constraints that do not involve the heap memory. In the original publication, this pure reasoning system is enabled by an external SMT solver, which is used to determine whether a pure constraint is implied by/compatible with another pure constraint.

Finally, SuSLik is capable of synthesizing programs with recursive procedure calls by generating cyclic proofs. This is achieved by using a rule that unifies a reduced version of the precondition after some rule applications with the precondition of the original function specification, and then inserts a function call to satisfy the postcondition of the original function, which can hopefully be used to complete the synthesis of the transformed specification.

3 Replication

This section discusses our replication, DeSyL, and how it compares to the original SuSLik implementation. We discuss the various aspects of the DeSyL implementation, emphasizing differences with SuSLik and ambiguities in the original paper, as well as the functionality that was not replicated in DeSyL.

3.1 Technology choice

The SuSLik paper itself only provides a theoretical framework for deductive synthesis, and so it is agnostic towards implementation choices such as the implementation language. The original SuSLik implementation is written in Scala.

[1] Pronounced "diesel".

Variable	$x, y ::=$ Alpha-numeric ident.
Value	$d ::=$ Theory-specific atoms
Offset	$\iota ::=$ Non-negative integer
Expression	$e ::= d \mid x \mid e = e \mid e \wedge e \mid \neg e \mid \ldots$
Command	$c ::= \texttt{let}\ x = *(x + \iota) \mid *(x + \iota) = e \mid \texttt{skip} \mid \texttt{error} \mid f(\overline{e_i}) \mid \texttt{if}\ (e)\ \{c\}\ \texttt{else}\ \{c\} \mid c; c$
Fun. dict.	$\Delta ::= \epsilon \mid \Delta, f\ (\overline{x_i})\ \{c\}$

Pure term	$\phi, \psi, \chi ::= e \mid \ldots$
Symb. heap	$P, Q, R ::= \texttt{emp} \mid \langle e, \iota \rangle \mapsto e \mid [x, n] \mid p\ (\overline{x_i}) \mid P * Q$
Assertion	$\mathcal{P}, \mathcal{Q} ::= \{\phi, P\}$
Heap predicate	$\mathcal{D} ::= p\ (\overline{x_i})\ \overline{\langle e_j, \{\chi_j, R_j\} \rangle}$
Function spec	$\mathcal{F} ::= f\ (\overline{x_i}) : \{\mathcal{P}\}\{\mathcal{Q}\}$
Environment	$\Gamma ::= \epsilon \mid \Gamma, x$
Context	$\Sigma ::= \epsilon \mid \Sigma, \mathcal{D} \mid \Sigma, \mathcal{F}$

Fig. 1. Grammar of program language (left) and SSL specifications (right) as per [6].

Our implementation DeSyL is written in C++, a language that is known for its strong support for systems programming and performance, a choice that is a matter of personal preference. Although the synthesis algorithm in the SuSLik paper is described in a functional style, it is easily translated to an imperative style since it describes a simple depth-first search. Furthermore, SuSLik uses Z3 as an external SMT solver to enable the pure reasoning system, and ScalaSMT to interface with Z3. DeSyL does not use an external SMT solver, as it is primarily designed to be a replication of the SuSLik methodology and integration with an external SMT solver was not a priority. Because of the lack of an explicit SMT solver, DeSyL uses modified versions of many of the SSL rules to enable the pure reasoning system, and emulates a simpler pure reasoning system with dedicated rules to rewrite and simplify the pure specification. Note that these rules are chosen specifically to include the capabilities needed to synthesize our benchmark examples, so the successful synthesis of the benchmarks in question does not imply that this system approximates the capabilities of an SMT solver.

3.2 Syntax

The most basic aspect of the implementation is the syntax definition for the program language and the goal specification language. The SuSLik paper gives formal definitions of the syntax of the program language and the goal specification language, shown in Figure 1. These grammars are relatively representative, but not entirely complete or congruent with the actual SuSLik implementation. The main ambiguities arise from symbolic parts of the syntax, such as the use of f indicating a function name and the separating comma between Δ and f not being a literal element of the syntax, but merely indicating a concatenation of functions. However, there are also cases where the implementation differs from the grammar, and we have to choose an approach for DeSyL. For the most part, these are minor and essentially arbitrary syntactic differences, such as the use of `**` for the separating conjunction $*$ since `*` is the multiplication operator, or adding the `predicate` keyword before predicate definitions and the `void` keyword before function names in the specification. The most remarkable

incongruity is that the specification grammar states that the arguments to a predicate invocation in a heap specification only take identifiers x as parameters, and not general expressions. This is a strange limitation, as it would not change the semantics of the specifications significantly if general expressions could be used. It would simply require substituting those expressions when unfolding the predicate definition. Furthermore, although there is no example for the SuSLik implementation that violates this limitation, there is actually an example in the SuSLik paper that does, using the predicate invocation $lseg^0(x, 0, S)$ in Section 2.5.2.[2] DeSyL resolves this ambiguity by following the grammar and only allowing identifiers as parameters, but this is an arbitrary choice.

3.3 Synthesis algorithm

As mentioned in Section 2, the synthesis algorithm is a simple depth-first search through the proof space, applying the SSL rules to the specification, backtracking when the rules cannot transform the specification further, and terminating when the specification is transformed into one that is trivially satisfied by the empty program. The SuSLik paper also outlines three major optimizations: phase distinction, rule invertibility, and commutativity optimization. Of these only rule invertibility is implemented in DeSyL. In both implementations some rules are known to be "invertible", meaning their application does not affect the solvability of the synthesis goal, so if one of these rules is applied and synthesis backtracks, other rules do not have to be attempted since the goal is not solvable by the synthesis algorithm. The SuSLik paper specifically indicates four invertible rules, Read, StarPartial, NullNotLVal, and SubsLeft. Of these, DeSyL does not include the StarPartial rule due to the different way it handles pure reasoning, and for this pure reasoning includes some other invertible rules, but the other three are implemented and invertible in DeSyL. The other two optimizations are completely absent in DeSyL. The first, phase distinction, involves dividing of the rule applications into "phases" where the rules are grouped. The synthesis algorithm is always in some phase corresponding to a specific group. The other optimization is commutativity optimization, which ensures that commutative rules are only applied in a canonical order to avoid redundant rule applications. One ambiguous aspect of the synthesis algorithm is that while the depth-first search attempts the different rules in order at every step (barring optimizations), the paper does not specify the order in which the rules are attempted in SuSLik. In DeSyL's development the order changed several times to optimize the synthesis process, with one notable concern being that the Read rule, which generates code and is often applicable without being necessary, is at the bottom of the priority list.

[2] Here the superscript 0 is a label used to indicate nesting for the recursion mechanism, and is not literally part of the syntax.

$$\frac{\mathrm{EV}(\Gamma, \mathcal{P}, \mathcal{Q}) = \varnothing \qquad \vdash \phi \Rightarrow \psi}{\Gamma; \{\phi; \mathtt{emp}\} \rightsquigarrow \{\psi; \mathtt{emp}\} \mid \mathtt{skip}} \textsc{Emp}_\mathrm{S} \qquad \frac{\mathrm{EV}(\Gamma, \mathcal{P}, \mathcal{Q}) = \varnothing}{\Gamma; \{\phi; \mathtt{emp}\} \rightsquigarrow \{true; \mathtt{emp}\} \mid \mathtt{skip}} \textsc{Emp}_\mathrm{D}$$

$$\frac{\psi \neq \bot \qquad \vdash \phi \wedge \psi \Rightarrow \bot \qquad \Gamma; \{\mathtt{emp}\} \rightsquigarrow \{\bot; \mathtt{emp}\} \mid c}{\Gamma; \{\phi; P\} \rightsquigarrow \{\psi; Q\} \mid c} \textsc{PostInconsistent}_\mathrm{S} \qquad \frac{\exists \phi' \in \phi \cup \psi : \neg\phi' \in \phi \cup \psi \qquad \Gamma; \{\mathtt{emp}\} \rightsquigarrow \{\bot; \mathtt{emp}\} \mid c}{\Gamma; \{\phi; P\} \rightsquigarrow \{\psi; Q\} \mid c} \textsc{PostInconsistent}_\mathrm{D}$$

Fig. 2. Comparison of the Emp and PostInconsistent rules in SuSLik (left) and DeSyL (right).

3.4 Synthesis rules

The specific SSL rules are the most important part of the SuSLik methodology, and the bulk of the implementation work in DeSyL is in the implementation of these rules. Although most of these rules remain the same in DeSyL, there are some notable differences. Figure 2 shows a comparison of the implementations of the Emp and PostInconsistent rules, where Emp is used to complete synthesis once the specification is trivially satisfied by the empty program, and PostInconsistent is used to terminate early from a synthesis where the postcondition is inconsistent with the precondition. Both of these rules rely on SuSLik's pure reasoning system through an external SMT solver to determine implication and inconsistency respectively. Since DeSyL does not use an external SMT solver, it relies on simpler pure reasoning rules to simplify the postcondition and strengthen the precondition. Then the Emp rule can be applied once the postcondition is simply true, and the PostInconsistent rule can be applied when there is a pure clause that is present in the specification while its literal negation is also present in the specification. DeSyL also adds a completely new early termination rule, the SetSize rule shown in Figure 3, which is used to terminate early from a synthesis where the postcondition and precondition specify inconsistent constraints on the size of a set variable. This is achieved by the PostInconsistent rule in SuSLik, but has to be handled separately in DeSyL due to the simpler pure reasoning system.

Figure 4 shows a comparison of the Call rule in SuSLik and DeSyL. The Call rule is used to synthesize a recursive function call, and is the only rule that

$$\frac{\begin{array}{c} s \text{ is a set variable} \\ minsize(s, \phi) > maxsize(s, \psi) \\ \vee\ maxsize(s, \phi) < minsize(s, \psi) \\ \Gamma; \{\mathtt{emp}\} \rightsquigarrow \{\bot; \mathtt{emp}\} \mid c \end{array}}{\Gamma; \{\phi; P\} \rightsquigarrow \{\psi; Q\} \mid c} \textsc{SetSize}_\mathrm{D}$$

Fig. 3. The SetSize rule in DeSyL.

$$\frac{\begin{array}{c}\mathcal{F} \triangleq f(\overline{x_i}) : \{\phi_f; P_f\}\ \{\psi_f : Q_f\} \in \Sigma \\ R =^l [\sigma]P_f \qquad \vdash \phi \Rightarrow [\sigma]\phi_f \\ \phi' \triangleq [\sigma]\psi_f \quad R' \triangleq [\sigma]Q_f \quad \overline{e_i} = [\sigma]\overline{x_i} \\ \textsc{Vars}(\{\overline{e_i}\}) \subseteq \Gamma \quad \Sigma; \{\phi \wedge \phi'; P * R'\} \rightsquigarrow \{Q\} \mid c\end{array}}{\Sigma; \{\phi; P * R\} \rightsquigarrow \{Q\} \mid f(\overline{e_i}); c} \qquad \frac{\begin{array}{c}\mathcal{F} \triangleq f(\overline{x_i}) : \{\phi_f; P_f\}\ \{\psi_f : Q_f\} \in \Sigma \\ R >^l [\sigma]P_f \\ \phi' \triangleq [\sigma]\psi_f \quad R' \triangleq [\sigma]Q_f \quad \overline{e_i} = [\sigma]\overline{x_i} \\ \textsc{Vars}(\{\overline{e_i}\}) \subseteq \Gamma \quad \Sigma; \{\phi'; P * R'\} \rightsquigarrow \{Q\} \mid c\end{array}}{\Sigma; \{\texttt{true}; P * R\} \rightsquigarrow \{Q\} \mid f(\overline{e_i}); c}$$

Fig. 4. Comparison of the CALL rule in SUSLIK (left) and DESYL (right).

is used to generate cyclic proofs in SUSLIK. In order to be able to synthesize recursive function calls, SUSLIK checks that the current precondition can imply a unified version of the precondition of the function specification, but DESYL uses a weaker CALL rule that can only call functions with a trivially true pure precondition. SUSLIK also uses an additional rule, the ABDUCECALL rule, to determine what parts of the precondition are still required to prepare for a recursive call and derive this preparation. This rule is not present in DESYL at all, meaning there are some cases where DESYL cannot synthesize a recursive call that SUSLIK can because it cannot prepare for the call.

3.5 Pure reasoning system

DESYL's pure reasoning system consists of a set of rules that are used to rewrite and simplify the pure specification, shown in Figure 5. The PUREFRAME rule is used to eliminate pure constraints from the postcondition when they are implied by the precondition, the REFLEXIVITY rule is used to eliminate a pure constraint from the postcondition when it is trivially true because it is a reflexive binary boolean operator applied to the same expression, the TRUEELISION rule simply eliminates any pure constraint that consists of the boolean literal `true` from the postcondition, and the EXPANDIMPLIED rule is used to strengthen the precondition by adding every constraint that is in the closure C of some constraint in the precondition. These rules are used to simplify the pure specification as much as possible, after which rules like EMP and POSTINCONSISTENT can be applied.

$$\frac{\Gamma; \{\phi \wedge \phi'; P\} \rightsquigarrow \{\psi; Q\} \mid c}{\Gamma; \{\phi \wedge \phi'; P\} \rightsquigarrow \{\psi \wedge \phi'; Q\} \mid c}\ \textsc{PureFrame}_{\text{D}} \qquad \frac{\Gamma; \{P\} \rightsquigarrow \{\psi; Q\} \mid c}{\Gamma; \{P\} \rightsquigarrow \{\psi \wedge \texttt{true}; Q\} \mid c}\ \textsc{TrueElision}_{\text{D}}$$

$$\frac{\begin{array}{c}\diamond \text{ is a reflexive binary operator} \\ \Gamma; \{P\} \rightsquigarrow \{\psi; Q\} \mid c\end{array}}{\Gamma; \{P\} \rightsquigarrow \{\psi \wedge e \diamond e\ Q\} \mid c}\ \textsc{Reflexivity}_{\text{D}} \qquad \frac{\begin{array}{c}C' := \{\phi^* \in C(\phi') \mid \phi^* \notin \phi\} \qquad C' \neq \varnothing \\ \Gamma; \{\phi \wedge \phi' \wedge \bigwedge_{\phi^* \in C'} \phi^*; P\} \rightsquigarrow \{Q\} \mid c\end{array}}{\Gamma; \{\phi \wedge \phi'; P\} \rightsquigarrow \{Q\} \mid c}\ \textsc{ExpandImplied}_{\text{D}}$$

Fig. 5. The pure reasoning rules in DESYL.

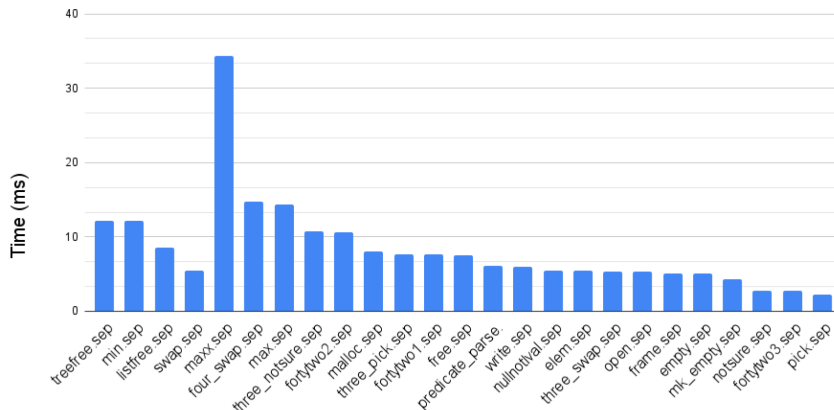

Fig. 6. DeSyL synthesis times.

4 Results

Next we compare the outputs of DeSyL and SuSLik, both in terms of efficiency and in terms of utility and versatility. We can see that many of the features of the initial SuSLik implementation are also included in DeSyL, although some are implemented in a more simplistic way. Furthermore, the lack of the Abduce-Call rule and the less powerful pure reasoning mechanism impact the ability to synthesize more complex programs, limiting the synthesis of arbitrary recursive programs. However, in general terms the two implementations are comparable, and the simpler examples are easily synthesized in both tools.

Unfortunately, due to the limitations of DeSyL and the SuSLik paper's emphasis on more complex synthesis examples many of the benchmarks reported in the SuSLik paper are not feasible in DeSyL at all. Of the 20 benchmarks presented in Section 6.1 of the SuSLik paper, only 4 have been achieved in DeSyL at all, namely `swap`, `min`, `listfree`, and `treefree`. Additionally, three more examples given in the body of the SuSLik paper to clarify the methodology; `elem`, `max`, and `notsure`; have been achieved in DeSyL as well but are not used as performance benchmarks in the SuSLik paper, so we cannot compare performance. The remaining 14 benchmarks are not feasible in DeSyL at all, and so we cannot compare performance on these benchmarks either.

4.1 Performance

Figure 6 shows the time required for synthesis for each example. Similarly to the SuSLik paper, we report the time as obtained on a commodity laptop (2.1 GHz AMD Ryzen Acer Aspire 5 with 16 GB of RAM, as opposed to SuSLik's 2.7 GHz Intel Core i7 Lenovo Thinkpad with 16GB RAM), and we compare to the SuSLik time with all optimizations enabled since this time is equivalent to the times without the optimizations that are absent in DeSyL for all examples.

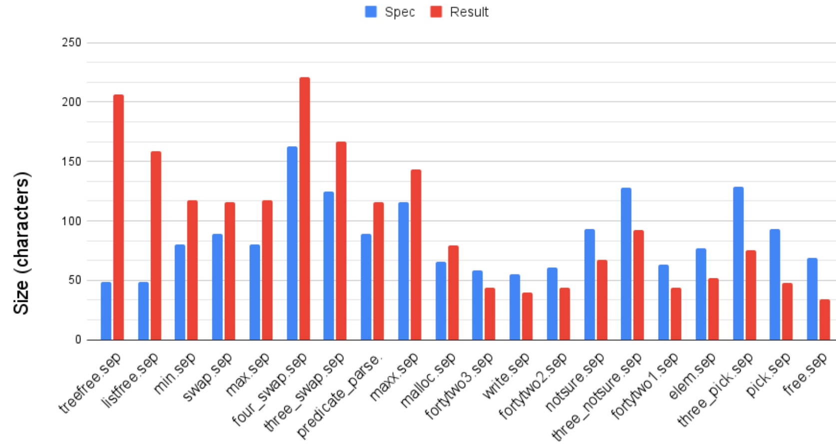

Fig. 7. DeSyL specification and synthesized program sizes.

Unfortunately, we cannot meaningfully graph the SuSLik times since they are all either 0.1 seconds or "< 0.1 seconds" which is not granular enough to plot, so we consider these separately. Of the four examples that can be synthesized by both implementations and are used as benchmarks in the SuSLik paper, which are listed on the left, three have a listed time with optimizations of < 0.1 seconds, which is the lowest possible value in the SuSLik table and gives us little information about the actual time. However, the listed time for `min` is 0.1 seconds. It is not clear from the paper whether this is rounded up or down, but it at least indicates that the time is roughly in the order of magnitude of 100 milliseconds. In contrast, DeSyL synthesizes the same example in only approximately 12 milliseconds. It is not exactly clear why DeSyL synthesizes this example faster, but one possible explanation is the branch abduction mechanism. SuSLik has to enumerate all atomic boolean expressions over program variables, and then defer to its external SMT solver to determine whether this expression is sufficient to make the implication valid, while DeSyL simply checks whether the pure postcondition consists of only one clause and then uses this clause. This means that DeSyL may be able to establish the candidate branch guard much faster than SuSLik, but at the cost of having a less powerful branch abduction mechanism.

4.2 Program size

Next we show a comparison between the amount of code in the specification and the amount of code in the synthesized program in Figure 7, omitting empty programs. Again the first four examples are the ones that are feasible in both SuSLik and DeSyL. We see that DeSyL is able to synthesize most examples from a specification with a size comparable to that of the program, ranging from 2 to less than 0.25 times the size of the program. Especially longer programs seem

to consistently be longer than the specification. These results are not compared to those of SuSLik because SuSLik uses a different metric for size, namely the number of AST nodes, but manual inspection did not reveal any difference.

5 Conclusion

In summary, we have seen that the SuSLik methodology is a viable approach to efficient deductive program synthesis, and that our implementation DeSyL successfully incorporates much of the outlined functionality. Furthermore, we have seen that DeSyL is able to synthesize many of the examples presented in the paper, although fewer of the 20 actual benchmarks, and at least one even faster than SuSLik. However, there are still some key features missing from DeSyL, which impact the synthesis of more complex recursive programs. Thus, while our implementation is a modestly successful replication, there are still aspects that we were not able to reproduce within the scope of the project.

The most obvious directions for future work are the integration of an external SMT solver, allowing for a more powerful pure reasoning system as in the paper, and the AbduceCall rule to enhance recursive synthesis. However, there are also some other interesting directions, especially those outlined in a later review paper on the status of SuSLik [4]. One interesting example is supporting the synthesis of concurrent programs. To address the complexity of shared memory in concurrent programs O'Hearn, an expert on Separation Logic [3,2,5], proposed Concurrent Separation Logic [1]. Combining the SuSLik methodology with CSL could allow for efficient synthesis of concurrent programs with shared memory.

References

1. Brookes, S., O'Hearn, P.W.: Concurrent separation logic. ACM SIGLOG News **3**(3), 47–65 (aug 2016). https://doi.org/10.1145/2984450.2984457, https://doi.org/10.1145/2984450.2984457
2. Calcagno, C., Ishtiaq, S., O'Hearn, P.: Semantic analysis of pointer aliasing, allocation and disposal in hoare logic. In: Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. p. 190–201. PPDP '00, Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/351268.351291, https://doi-org.ezproxy2.utwente.nl/10.1145/351268.351291
3. Ishtiaq, S., O'Hearn, P.: Bi as an assertion language for mutable data structures. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 14–26. POPL '01, Association for Computing Machinery, New York, NY, USA (2001). https://doi.org/10.1145/360204.375719, https://doi-org.ezproxy2.utwente.nl/10.1145/360204.375719
4. Itzhaky, S., Peleg, H., Polikarpova, N., Rowe, R., Sergey, I.: Deductive synthesis of programs with pointers: Techniques, challenges, opportunities. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 110–134. Springer International Publishing, Cham (2021)

5. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) Computer Science Logic. pp. 1–19. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
6. Polikarpova, N., Sergey, I.: Structuring the synthesis of heap-manipulating programs. Proc. ACM Program. Lang. **3**(POPL) (1 2019). `https://doi.org/10.1145/3290385`, `https://doi-org.ezproxy2.utwente.nl/10.1145/3290385`
7. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74 (2002). `https://doi.org/10.1109/LICS.2002.1029817`