Natural Hoare Logic: Towards formal verification of programs from logical forms of natural language specifications

Jayaraj Poroor
jayaraj.pooroor@gmail.com
JIFFY.ai

ABSTRACT

Formal verification provides strong guarantees of correctness of software, which are especially important in safety or security critical systems. Hoare logic is a widely used formalism for rigorous verification of software against specifications in the form of pre-condition/post-condition assertions. The advancement of semantic parsing techniques and higher computational capabilities enable us to extract semantic content from natural language text as formal logical forms, with increasing accuracy and coverage.

This paper proposes a formal framework for Hoare logic-based formal verification of imperative programs using logical forms generated from compositional semantic parsing of natural language assertions. We call our reasoning approach *Natural Hoare Logic*. This enables formal verification of software directly against safety requirements specified by a domain expert in natural language.

We consider both declarative assertions of program invariants and state change as well as imperative assertions that specify commands which alter the program state. We discuss how the reasoning approach can be extended using domain knowledge and a practical approach for guarding against semantic parser errors.

1 INTRODUCTION

Formal verification techniques [50] are a collection of techniques that can be used to provide strong guarantees of correctness and security for software. They are especially desirable in safety critical software such as the control software of medical devices [24] or security sensitive portions of a software system [2]. In order to perform formal verification we must specify the correctness or security requirements in a formal language. For instance, consider the following natural language specification of a safety requirement of a medical infusion pump from [35]:

“The infusion manager shall stop infusion whenever a critical alarm occurs.” [35]

Such natural language specifications are typically given by domain experts who understand the safety requirements of the particular field. However, in order to formally verify the system against these requirements, we must specify them in a formal language, which cannot be typically done by the domain expert. The objective of this paper is to propose a general method that enables software to be formally verified directly against requirements specified by a domain expert in natural language.

Hoare logic [16]¹ and its variants are established formal techniques for proving that a program satisfies its specification in the form of pre-condition, post-condition assertions.

Hoare logic assertions are specified as triples of the form $\{P\}S\{Q\}$, where P and Q are logical assertions on program variables. The meaning of the triple is that: if program variables satisfy the condition P and the program S is executed, then the program variables satisfy the condition Q provided the program terminates. This is called the partial correctness assertion. We may extend it to total correctness with the additional assertion that the program will terminate when starting at a variable state satisfying P . In this paper, we consider only partial correctness. If we do not consider exception conditions, for loop-free programs partial correctness coincide with total correctness.

Semantic parsing [18] is a collection of techniques whose objective is to obtain a formal representation of meaning from natural language sentences. For most part, the research in semantic parsing is concerned with obtaining a representation of meaning that can be interpreted by a computer to perform the task at hand, such as querying a database to obtain results. Semantic parsing covers a spectrum of techniques from shallow to deep. Shallow semantic parsing techniques such as semantic role labelling [30] aims to identify semantic roles (such as agent or location) associated with the events specified in a sentence. In contrast, deep semantic parsing aims to extract a rich representation of meaning known as logical forms in a formal system such as the λ -calculus [5].

In this paper we concern ourselves with formal logical forms generated by semantic parsers in the form of λ -calculus expressions. λ -calculus is preferred over first order logic for logical forms since it allows us to define semantics compositionally using function composition and function application, from logic forms representing meanings of constituents. Compositional semantics is originally due to Richard Montague [34] and hence is broadly called Montague semantics [17]. Simply-typed λ -calculus [10] with pre-defined constants for logical connectives and quantifiers, known as the higher-order logics², is commonly employed as the formal language for logical forms [7]. In this paper we consider logical forms in this formal language³. Owing to the expressive power of λ -calculus we do not lose any generality. Other logical forms like λ -DCS [25] can be readily expressed in terms of λ -calculus.

¹Also called Floyd-Hoare logic since Floyd had developed a similar system for Flowchart programs earlier[14].

²Higher-order logics may also be considered as a generalization of first order logic.

³The lambda calculator available at <http://lambdacalculator.com/> is a great educational tool for natural language semantics in the λ -calculus.

We propose a framework for associating logical forms generated from assertions about program state made in natural language to pre-condition/post-condition assertions in Hoare logic. We call our framework, *Natural Hoare Logic* (NHL). Our objective is to enable formal verification of programs from natural language assertions.

2 THE PROPOSED FRAMEWORK

We introduce the proposed framework with the help of simple examples. We use the notation $\llbracket \text{text} \rrbracket$ to denote the semantics of *text* in λ -calculus. We use **identifier** to denote the semantic function or constant corresponding to *identifier* appearing in the sentence.

Consider the statement:

All balances must be greater than zero.

where,

$\llbracket \text{balances} \rrbracket = \lambda x. \text{balance}(x)$, a function that returns the balance value, given an account. For the sake of simplicity, in our examples, we restrict our semantic domain (and hence our domain of discourse) to accounts.

$\llbracket \text{greater than} \rrbracket = \lambda x, y. x > y$

$\llbracket \text{zero} \rrbracket = \lambda x. 0$, which we may simply write as 0 .

$\llbracket \text{all} \rrbracket = \forall. \forall$ is simply a higher-order function constant in our logic that takes a truth-valued function f and evaluates to true if f evaluates to true for all values.

The semantics of the entire sentence may be composed from the constituent semantics to get:

$\llbracket \text{All balances must be greater than zero} \rrbracket =$

$$\forall x. \text{balance}(x) > 0 \quad (1)$$

Please note that, $\forall x. P$ is a short-hand for $\forall \lambda x. P$. We omit explicit type annotations in the logical forms used in our examples.

Now, consider the following program fragment that simply increments a variable representing balance:

$$_balance := _balance + 1 \quad (2)$$

To avoid name confusion we prefix the names of program variables with underscore (e.g., $_balance$).

2.1 Invariant declaratives

In order to prove that the program holds the invariant specified by the logical form in Equation 1, we must relate the variables occurring in the logical form to the variables occurring in the program.

We specify the relation between the logical form and program variables via a formal relation. We call this the *logical form - program logic* relation (LFPL relation). This is similar in spirit to the state relations [49] from the refinement theory [11] used to prove formal correspondence (forward/backward simulations) between an abstract specification and its concrete implementation.

We may use the semantic expressions corresponding to the entities present in the natural language specification to construct the relation, as below.

Figure 1: Invariant proofs in Hoare logic from logical forms.

$$\mathcal{R} \triangleq \exists x. \text{balance}(x) = _balance \quad (3)$$

It may be noted that $\exists x. P = \exists \lambda x. P$.

Given a logical form $\mathcal{L}$ specifying a program invariant⁴ and an LFPL relation $\mathcal{R}$, we may prove that the program S satisfies the invariant by establishing the Hoare triple $\{I\} S \{I\}$, where I is the strongest condition that satisfies:

$$\mathcal{L} \wedge \mathcal{R} \implies I \not\Rightarrow \mathcal{R} \quad (4)$$

The condition, $I \not\Rightarrow \mathcal{R}$ is required to remove the correspondence relation from the Hoare logic assertions. In practice, I should be an assertion on program variables without any correspondence to logical forms. Presence of the correspondence relation in Hoare logic assertions may cause the proof to fail owing to the predicate transformer nature of Hoare logic rules. The Hoare logic rules will transform the program variables in the assertions while the variables in the logical forms remain unchanged. This results in the LFPL relation specifications in the assertions to become inconsistent as Hoare logic rules get applied.

The requirement that I is the strongest condition that satisfies the above constraint ensures that a weakened form of the user requirement is not take up for proof.

Figure 1 visualizes how $\mathcal{R}$ relates the invariant specified in the logical form $\mathcal{L}$ to pre-condition P and post-condition Q . The arrow $\{Q\} \rightarrow S \rightarrow \{P\}$ shows the direction of the proof. Program execution and the state transformation occurs in the opposite direction.

Using the Hoare triple in Equation 21 for proving logical form 1 on program 2:

Since:

$$\{\forall x. \text{balance}(x) > 0\} \wedge \exists x. \text{balance}(x) = _balance \quad (5)$$

$$_balance > 0 \quad (6)$$

We need to prove:

$$\{_balance > 0\} \quad (7)$$

$$_balance := _balance + 1 \quad (8)$$

$$\{_balance > 0\} \quad (9)$$

We can prove this by considering a stronger post-condition $_balance > 1$ and applying the assignment rule, and then applying the consequence rule to weaken the post-condition to get back the invariant.

⁴We consider the invariant to hold true only before and after the program execution, not in the intermediate states.

The LFPL relation simply specifies the formal association between the logical form and the program variables. We may make a general assertion in natural language and then use the LFPL relation to establish the association. For example, consider the natural language assertion:

All values must be greater than zero.

Its corresponding logical form may be:

$$\forall x \cdot \text{valueof}(x) > 0 \quad (10)$$

In this case, the LFPL relation would specify that the program variable `_balance` is the value of an account, as follows:

$$\exists x \cdot \text{valueof}(x) = \text{_balance} \quad (11)$$

2.2 Imperatives

Consider the statement:

Increment the balance.

This is an imperative sentence, specifying a command to be performed. Semantics of imperatives are much more challenging than that of declarative statements [21]. A common approach for specifying semantics of imperative sentences is by introducing modal operators [20, 38]. In this paper, we do not consider imperatives in a general setting, instead restrict ourselves to considering only those imperative statements that specify operations on program state.

As a result, we explicitly model a restricted form of modality as a logical relation between the current state of the world (before-state) and the state after the command has been carried out (after-state)⁵. We define a function *post* that maps elements to their after-state. Therefore, $\lambda x \cdot \text{balance}(\text{post}(x))$ gives the balance of an account after the specified command has been carried out. The semantics of ‘increment’ can be defined as a higher order function as follows:

$$\llbracket \text{increment} \rrbracket = \lambda f \cdot \forall x \cdot f(\text{post}(x)) = f(x) + 1$$

This allows such imperative commands to be composed with the semantics of its objects, enabling compositional semantics, e.g.,

$$\llbracket \text{increment the balance} \rrbracket = \forall x \cdot \text{balance}(\text{post}(x)) = \text{balance}(x) + 1$$

Of course we are unable to model general imperative statements such as: *you may increment the balance*. However, we believe our approach is useful enough for specifying pre-condition/post-conditions assertions on sequential imperative programs.

In addition to the $\mathcal{R}$ relation defined previously, we also define an $\mathcal{R}'$ relation as:

$$\mathcal{R}' \triangleq \exists x \cdot \text{balance}(\text{post}(x)) = \text{_balance} \quad (12)$$

$\mathcal{R}'$ is obtained by replacing the variable x by $\text{post}(x)$ in the body of $\mathcal{R}$, assuming the $\mathcal{R}$ relation is on a single logical variable x . If there are multiple logical variables, then the *post* function must be applied on all of them.

⁵We may consider this as a modality with two possible worlds represented by the before-state and after-state.

Figure 2: Hoare logic proofs from imperative logical forms.

Given a $\mathcal{L}$ which is a logical form of the imperative assertion, we may prove that a program S satisfies the specified imperative command, by proving the following:

$$\{\mathcal{L} \wedge \mathcal{R}\} S \{\mathcal{L} \wedge \mathcal{R}'\} \quad (13)$$

$$(14)$$

This is simpler than that of the invariant because post-condition and pre-condition use different LFPL relations, $\mathcal{R}$ and $\mathcal{R}'$.

Figure 2 visualizes how LFPL relations $\mathcal{R}$ and $\mathcal{R}'$ are used for Hoare logic proofs from imperative logical forms.

For the sentence, *increment the balance* the Hoare triple to prove becomes:

We need to prove:

$$\{\mathcal{L} \wedge \exists x \cdot \text{balance}(x) = \text{_balance}\} \quad (15)$$

$$\text{_balance} := \text{_balance} + 1 \quad (16)$$

$$\{\mathcal{L} \wedge \exists x \cdot \text{balance}(\text{post}(x)) = \text{_balance}\} \quad (17)$$

where $\mathcal{L} = \forall x \cdot \text{balance}(\text{post}(x)) = \text{balance}(x) + 1$.

Applying the assignment rule to the post-condition, we get the following assertion, from which the pre-condition follows:

$$\mathcal{L} \wedge \exists x \cdot \text{balance}(\text{post}(x)) = \text{_balance} + 1 \quad (18)$$

2.3 Conditional imperatives

We consider conditional imperatives where the conditional and imperative parts are specified separately. An example of a conditional imperative statement is:

IF: *the balance is greater than 0* THEN: *increment the balance*.

The proof approach for imperatives may be extended to conditional imperatives in a straightforward manner.

If $\mathcal{E}$ is the logical form of the conditional part and $\mathcal{L}$, then the proof rule is:

$$\{\mathcal{L} \wedge \mathcal{R} \wedge \mathcal{E}\} S \{\mathcal{L} \wedge \mathcal{R}'\} \quad (19)$$

$$(20)$$

The proof rule is visually illustrated in Figure 3.

2.4 Pre/Post-condition declaratives

Another form of natural language specification we consider is the declarative specification of the pre-condition and post-condition separately. For instance:

IF: *the balance is greater than 0* THEN AFTER: *balance must be remain non-negative*

Figure 3: Hoare logic proofs from logical forms of conditional imperatives.

Figure 4: Hoare logic proofs from pre/post-condition logical forms.

This case is simpler than the imperative specification. The Hoare logic assertion would be:

$$\{\mathcal{L} \wedge \mathcal{R}\} S \{\mathcal{L}' \wedge \mathcal{R}'\} \quad (21)$$

$$(22)$$

where $\mathcal{L}'$ is the logical form of the post-condition and $\mathcal{L}$ is the logical form of the pre-condition.

Figure 4 visualizes how LFPL relations $\mathcal{R}$ and $\mathcal{R}'$ are used for Hoare logic froms for logical forms of pre/post-condition assertions.

2.5 Extending with domain knowledge

In this section we discuss how our reasoning technique can be extended using domain knowledge specified in the form of logical expressions. For example, consider the sentence and its logical form below:

All values must be greater than zero

Logical form:

$$\forall x \cdot \text{valueof}(x) > 0 \quad (23)$$

Using the LFPL relation below, assume that we have already established proof for this invariant for some program S .

$$\exists x \cdot \text{valueof}(x) = \text{_balance} \quad (24)$$

Now consider a new sentence and its logical form:

All balances must be greater than zero

$$\forall x \cdot \text{balance}(x) > 0 \quad (25)$$

Consider the domain knowledge that the balance of an account is its value, which we may express logically as:

$$\forall x \cdot \text{balance}(x) = \text{valueof}(x) \quad (26)$$

We may use this and our previous proof to now infer, that program S satisfies new invariant in Equation 25, without having to go through the proof again.

In many domains, formalized domain knowledge exists in the form of domain ontologies in languages such as the web ontology language [33] (which has a logical basis in description logics [3]). For instance the ontological assertion that A is a B can be expressed as:

$$\forall x \cdot A(x) \implies B(x) \quad (27)$$

Bringing such knowledge to aid in our reasoning is likely to be quite beneficial.

2.6 Guarding against semantic parser errors

Owing to the variety and ambiguity in natural language sentences, semantic parsers are probabilistic parsers. The parse (and the resultant logical form) with the highest probability is taken as the selected parse (logical form). This need not always be the correct parse. Since we are targeting formal verification of programs, we need to guard against these errors. In this section, we consider one practical approach.

Given a language of logical forms generated by a semantic parser, we define a function from logical forms to natural language, let's call this function $\mathcal{N}$. We may use this function to paraphrase the user input in the manner understood by the system. Given a natural language sentence S input by a user and if its logical form generated by a semantic parser is $\llbracket S \rrbracket$, then we could ask back the user:

Did you mean: $\mathcal{N}(\llbracket S \rrbracket)$?

If the answer is *yes* we proceed, otherwise we could show the $\mathcal{N}$ -mappings of other probable logical forms (if available) for user to choose from. User's selection may also be used as a form of labelling for retraining the model. If the user is unable to choose, then the system may prompt the user to rephrase the specification (perhaps as simpler sentences).

Of course it is important for the $\mathcal{N}$ -function to generate a consistent and understandable natural language representation. This is usually a simpler problem than the semantic parsing itself.

Semantic parsing based on synchronous context free grammars (SCFG) [48] is especially interesting to address this problem. This approach recasts semantic parsing as a machine translation problem - that of translating from natural language to the language of logical forms. This enables the parsing to be inverted to generate natural language statements from logical forms, automatically giving us the function $\mathcal{N}$ from the learnt grammar.

3 BRIEF REVIEW OF RELATED FIELDS

In this section, we briefly review Hoare logics and semantic parsing since our framework attempts to connect logical forms that are outcomes of semantic parsing with Hoare logic assertions on programs.

Hoare logic

We briefly review the basic Hoare logic rules. The simplest rule is the axiom of assignment:

$$\frac{}{\{Q[E/x]\} x := E \{Q\}} \quad (28)$$

The notation $Q[E/x]$ stands for the expression obtained after all instances of the variable x occurring in Q is substituted with expression E .

The rule for sequential composition can be used to infer the pre-condition/post-condition asserts of a program from its sequential components:

$$\frac{\{P\} S_1 \{Q\} \quad \{Q\} S_2 \{R\}}{\{P\} S_1; S_2 \{R\}} \quad (29)$$

The conditional rule may be applied to reason about if-else statements.

$$\frac{\{E \wedge P\} S_1 \{Q\} \quad \{\neg E \wedge P\} S_2 \{Q\}}{\{P\} \text{if } E \text{ then } S_1 \text{ else } S_2 \{Q\}} \quad (30)$$

The consequence rules allows us to strengthen the pre-condition and weaken the post-condition:

$$\frac{P \implies P' \quad \{P'\} S \{Q'\} \quad Q' \implies Q}{\{P\} S \{Q\}} \quad (31)$$

Reasoning about loops is done using the *while* loop rule:

$$\frac{\{I \wedge E\} S \{I\}}{\{I\} \text{while } E \text{ do } S \{\neg E \wedge I\}} \quad (32)$$

Here I is the *loop invariant*. One of the primary difficulties in automating Hoare logic proofs is the construction of loop invariants. In general, the code must be manually annotated with loop invariants for Hoare logic proofs to go through.

The Hoare logic axioms and inference rules provide an axiomatic semantics for an imperative programming language. Hoare logic rules may be recast as rules for obtaining weakest pre-conditions from post-condition assertions (or equivalently, strongest post-conditions from pre-condition assertions). The weakest pre-condition or predicate transformer semantics is originally due to Dijkstra [12].

In recent years, the basic Hoare logic [1] has been extended to support formal verification of complex and real-world code. For instance, separation logic [42] is one of its variants used for proving correctness properties of programs with pointer data structures. Crash Hoare Logic [9] extends Hoare logic with special *crash* assertions and recovery procedures that enables formal reasoning about crash recovery. The paper establishes formal proofs using this extended logic for FSCQ file system.

A number of researchers have worked on defining Hoare logics for real-world programming languages, both high level and low level. A Hoare Logic for Java embedded in the Isabelle/HOL theorem prover [37] is given in [46]. A Hoare logic for realistic machine code is presented in [36].

The principle of forward (backward) simulations from the refinement theory [11] may be used to prove formal correspondence between an abstract specification and concrete

implementation. This enables us to prove formal properties on the abstract specification using Hoare logic (or equivalent methods) which then carries over to the concrete implementation owing to the simulation proofs. Formal verification of complex programs such as the seL4 microkernel takes this approach [22].

Mechanization of Hoare logic proofs involve first generating verification conditions by a verification condition generator (VCG) that apply Hoare logic rules on the annotated code [32]. The verification conditions are then typically mechanically checked by an automated theorem prover.

Semantic parsing

Combinatory categorical grammars (CCGs) [45] is a well studied and powerful method to achieve compositional semantics. CCGs employ a few general purpose combinatory rules along with a highly lexicalized grammar that maps tokens/phrases to categories. CCG2Lambda [31] is a recent work that compositionally maps CCG trees to λ -terms.

Compositional semantics can also be achieved by attaching logical forms to context-free syntactic parsing rules (usually parsing is done on tokens/phrases tagged with semantic annotations)⁶. Whenever a context-free syntactic rule is applied, the attached logical form is composed with the logical forms of the constituents to generate a resultant logical form.

Semantic parsing based on synchronous context free grammars (SCFG) is proposed in [48]. This approach recasts semantic parsing as a machine translation problem - that of translating from natural language to the language of logical forms. This also enables the parsing to be inverted to generate natural language statements from logical forms. The ability to invert logical forms to natural language statements is especially interesting for the problem being addressed in this paper. This enables us to give a natural language feedback of what the system has understood regarding the natural language specification made by the domain expert. This acts as a check against any potential errors introduced by the semantic parser.

Compositional semantics using dependency-based parse structures is discussed in [27]. More recently, in [41], a compositional semantics is defined over universal dependency parse trees. Here the λ -calculus logical forms capture semantics in Neo-Davidsonian or event style [8].

Since natural language parsing involves ambiguity, a learnt scoring model (usually a linear or a log-linear model) is applied during the semantic parsing to keep track of most probable parses [28]. Learning weights in the scoring model is done in a supervised setting. Recently techniques for weak supervision wherein the sentences are labelled with the denotations (i.e., the results of executing the logical forms) are being investigated [6, 26, 47] replacing the need for strong supervision using parse tree annotations or even annotating sentences with the logical forms. This brings in the problem

⁶e.g., SippyCup semantic parser available at <https://github.com/wcmac/sippycup>

of program induction, i.e., inducing logical forms from denotations, which is a key challenge that must be solved when labelling with denotations.

Another approach to semantic parsing [44] takes advantage of semantic lexicons based on specific linguistic theories such as VerbNet[43] which is based on Levin’s theory of verb alternation classes[23] and FrameNet[4] which is based on frame semantics [13].

A yet another approach is to target a narrow domain specific language such as the structured query language (SQL). In such a case the semantic parser may directly generate the SQL (DSL) or may first generate an intermediate logical form, which is then automatically translated to the DSL (SQL). Natural language to SQL is a well studied problem [51]. The current state of the art supports SQL queries without joins using deep learning models. Such deep learning models have also become state-of-the-art in semantic role labelling.

A semantic parser for parsing natural language sentences to if-this-then-that recipes is given in [40], targeting the IFTTT service [39]. Their natural language specifications are somewhat similar to the conditional imperative forms discussed in this paper. However, their semantic parser output is in the form of IFTTT recipes rather than λ -calculus logical forms. Their parsing approach is based on KRISP [19] that uses a classifier with a string subsequence kernel [29] for mapping the natural language sentences to most probable production rules.

4 CONCLUSION AND FUTURE WORK

Hoare logic belongs to a general class of formal systems used for reasoning about programs. The results presented here may also be employed to reason with other program logics such as the dynamic logic[15]. We intend to further validate the NHL framework with more elaborate examples and case studies. Exploring the use of domain knowledge to aid reasoning, investigating approaches for a formal soundness proof, and mechanization of the framework are other lines of future work.

REFERENCES

- [1] Krzysztof R Apt and Ernst-Rüdiger Olderog. 2019. Fifty years of Hoare’s logic. *Formal Aspects of Computing* 31, 6 (2019), 751–807.
- [2] Matteo Avale, Alfredo Pironti, and Riccardo Sisto. 2014. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing* 26, 1 (2014), 99–123.
- [3] Franz Baader, Ian Horrocks, and Ulrike Sattler. 2004. Description logics. In *Handbook on ontologies*. Springer, 3–28.
- [4] Collin F Baker, Charles J Fillmore, and John B Lowe. 1998. The berkeley framenet project. In *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 1*. 86–90.
- [5] Hendrik P Barendregt et al. 1984. *The lambda calculus*. Vol. 3. North-Holland Amsterdam.
- [6] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 conference on empirical methods in natural language processing*. 1533–1544.
- [7] Bob Carpenter. 1997. *Type-logical semantics*. MIT press.
- [8] Lucas Champollion. 2015. The interaction of compositional semantics and event semantics. *Linguistics and Philosophy* 38, 1 (2015), 31–66.
- [9] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 18–37.
- [10] Alonzo Church. 1940. A formulation of the simple theory of types. *The journal of symbolic logic* 5, 2 (1940), 56–68.
- [11] Willem-Paul De Roeper, Kai Engelhardt, and Karl-Heinz Buth. 1998. *Data refinement: model-oriented proof methods and their comparison*. Number 47. Cambridge University Press.
- [12] Edsger W Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457.
- [13] Charles J Fillmore et al. 2006. Frame semantics. *Cognitive linguistics: Basic readings* 34 (2006), 373–400.
- [14] RM FL00. 1967. Assigning Meanings to Programs. In *Proceedings of Symposium in Applied Mathematics*.
- [15] David Harel, Dexter Kozen, and Jerzy Tiuryn. 2001. Dynamic logic. In *Handbook of philosophical logic*. Springer, 99–217.
- [16] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.
- [17] Theo Janssen. 2011. Montague semantics. (2011).
- [18] Aishwarya Kamath and Rajarshi Das. 2018. A survey on semantic parsing. *arXiv preprint arXiv:1812.00978* (2018).
- [19] Rohit Kate and Raymond Mooney. 2007. Semi-supervised learning for semantic parsing using support vector machines. In *Human Language Technologies 2007: the Conference of the North American Chapter of the Association for Computational Linguistics; Companion Volume, Short Papers*. 81–84.
- [20] Magdalena Kaufmann. 2019. Fine-tuning natural language imperatives. *Journal of Logic and Computation* 29, 3 (2019), 321–348.
- [21] Stefan Kaufmann and Magdalena Schwager. 2009. A unified analysis of conditional imperatives. In *Semantics and linguistic theory*, Vol. 19. 239–256.
- [22] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 207–220.
- [23] Beth Levin. 1993. *English verb classes and alternations: A preliminary investigation*. University of Chicago press.
- [24] Chunxiao Li, Anand Raghunathan, and Niraj K Jha. 2013. Improving the trustworthiness of medical device software with formal verification methods. *IEEE Embedded Systems Letters* 5, 3 (2013), 50–53.
- [25] Percy Liang. 2013. Lambda dependency-based compositional semantics. *arXiv preprint arXiv:1309.4408* (2013).
- [26] Percy Liang. 2016. Learning executable semantic parsers for natural language understanding. *Commun. ACM* 59, 9 (2016), 68–76.
- [27] Percy Liang, Michael I Jordan, and Dan Klein. 2013. Learning dependency-based compositional semantics. *Computational Linguistics* 39, 2 (2013), 389–446.
- [28] Percy Liang and Christopher Potts. 2015. Bringing machine learning and compositional semantics together. *Annu. Rev. Linguist.* 1, 1 (2015), 355–376.
- [29] Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. 2002. Text classification using string kernels. *Journal of Machine Learning Research* 2, Feb (2002), 419–444.
- [30] Lluís Màrquez, Xavier Carreras, Kenneth C Litkowski, and Suzanne Stevenson. 2008. Semantic role labeling: an introduction to the special issue.
- [31] Pascual Martínez-Gómez, Koji Mineshima, Yusuke Miyao, and Daisuke Bekki. 2016. ccg2lambda: A compositional semantics system. In *Proceedings of ACL-2016 System Demonstrations*. 85–90.
- [32] John Matthews, J Strother Moore, Sandip Ray, and Daron Vroon. 2006. Verification condition generation via theorem proving. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 362–376.
- [33] Deborah L McGuinness, Frank Van Harmelen, et al. 2004. OWL web ontology language overview. *W3C recommendation* 10, 10 (2004), 2004.
- [34] Richard Montague. 1973. The proper treatment of quantification in ordinary English. In *Approaches to natural language*. Springer, 221–242.
- [35] Anitha Murugesan, Michael W Whalen, Sanjai Rayadurgam, and Mats PE Heimdahl. 2013. Compositional verification of a medical

- device system. In *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology*. 51–64.
- [36] Magnus O Myreen and Michael JC Gordon. 2007. Hoare logic for realistically modelled machine code. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 568–582.
- [37] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.
- [38] Despina Oikonomou. 2016. Imperatives are existential modals: Deriving the strong reading as an implicature. In *Semantics and linguistic theory*, Vol. 26. 1043–1062.
- [39] Steven Ovidia. 2014. Automate the internet with “if this then that”(IFTTT). *Behavioral & social sciences librarian* 33, 4 (2014), 208–211.
- [40] Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 878–888.
- [41] Siva Reddy, Oscar Täckström, Slav Petrov, Mark Steedman, and Mirella Lapata. 2017. Universal semantic parsing. *arXiv preprint arXiv:1702.03196* (2017).
- [42] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 55–74.
- [43] Karin Kipper Schuler. 2005. VerbNet: A broad-coverage, comprehensive verb lexicon. (2005).
- [44] Lei Shi and Rada Mihalcea. 2005. Putting pieces together: Combining FrameNet, VerbNet and WordNet for robust semantic parsing. In *International conference on intelligent text processing and computational linguistics*. Springer, 100–111.
- [45] Mark Steedman and Jason Baldridge. 2011. Combinatory categorial grammar. *Non-Transformational Syntax: Formal and explicit models of grammar* (2011), 181–224.
- [46] David von Oheimb. 2001. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience* 13, 13 (2001), 1173–1214.
- [47] Bailin Wang, Ivan Titov, and Mirella Lapata. 2019. Learning semantic parsers from denotations with latent structured alignments and abstract programs. *arXiv preprint arXiv:1909.04165* (2019).
- [48] Yuk Wah Wong and Raymond Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*. 960–967.
- [49] Jim Woodcock and Jim Davies. 1996. Using Z: Specification ,Refinement ,and Proof. (1996).
- [50] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. 2009. Formal methods: Practice and experience. *ACM computing surveys (CSUR)* 41, 4 (2009), 1–36.
- [51] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103* (2017).