Language-Parametric Static Semantic Code Completion

DANIEL A. A. PELSMAEKER, Delft University of Technology, The Netherlands
HENDRIK VAN ANTWERPEN, Delft University of Technology, The Netherlands
CASPER BACH POULSEN, Delft University of Technology, The Netherlands
EELCO VISSER, Delft University of Technology, The Netherlands

Code completion is an editor service in IDEs that proposes code fragments for the user to insert at the caret position in their code. Code completion should be sound and complete. It should be sound, such that it only proposes fragments that do not violate the syntactic and static semantic rules of the language. It should be complete, such that it proposes all valid fragments so that code completion can be used to construct all programs. To realize soundness and completeness, code completion should be informed by the language definition. In practice, the implementation of code completion is an additional effort in the implementation of a language.

In this paper, we develop a framework for language-parametric semantic code completion for statically typed programming languages based on their specification of syntax and static semantics, realizing the implementation of a code completion editor service with minimal additional effort. The framework builds on the SDF3 syntax definition formalism and the Statix static semantics specification language. The algorithm reinterprets the static semantics definition to find sound expansions of predicates and solutions to name resolution queries in scope graphs. This allows a search strategy to explore the solution space and synthesize completion proposals. The implementation of the strategy language and code completion algorithm extend the implementation of the Statix solver, and can be used for any language defined in Statix. We demonstrate soundness and completeness of the completion proposal synthesis, and evaluate its performance.

CCS Concepts: • **Software and its engineering → Semantics**.

Additional Key Words and Phrases: semantic code completion, semantics, constraint solving, name binding, editor services, reference resolution, code completion

ACM Reference Format:
Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. 2022. Language-Parametric Static Semantic Code Completion. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 85 (April 2022), 30 pages. https://doi.org/10.1145/3527329

1 INTRODUCTION

An *Integrated Development Environment (IDE)* assists programmers to produce high-quality code by means of *editor services* such as syntax highlighting, code completion, code navigation, inline error messages, and refactorings. *Code completion* is an editor service that allows the user to insert a code fragment at the current caret position, by selecting it from a list of proposals provided by the editor. This reduces typing, minimizes typing errors, helps the user find relevant code fragments, aids in discovery by proposing possible syntax and references, and avoids incorrect API usage.

Authors' addresses: Daniel A. A. Pelsmaeker, Delft University of Technology, Delft, The Netherlands, d.a.a.pelsmaeker@tudelft.nl; Hendrik van Antwerpen, Delft University of Technology, Delft, The Netherlands, h.vanantwerpen@tudelft.nl; Casper Bach Poulsen, Delft University of Technology, Delft, The Netherlands, c.b.poulsen@tudelft.nl; Eelco Visser, Delft University of Technology, Delft, The Netherlands, e.visser@tudelft.nl.

2475-1421/2022/4-ART85
https://doi.org/10.1145/3527329

Ideally, code completion should be *sound*. That is, it should take into account the syntactic and static semantic rules of the programming language, to only propose syntax and references that do not introduce additional compile-time errors at the caret position. Because code completion that proposes nothing is trivially sound, it should ideally also be *complete*. That is, it should propose all permissible syntax and references given the compile-time model of the program.

In practice, implementations of code completion generally are complete in terms of proposed names, but vary in soundness. That is, they may propose names and syntax that are not correct in the context of the caret position according to the syntactic and static semantic rules of the program. For example, proposing names whose (return) type does not match the expected type at the caret position, or proposing references that, when inserted, would inadvertently capture another name than intended. We discuss more examples below.

Many code editors provide code completion out of the box. But to support a wide range of programming languages, code completion is often either too generic or too specific and not generalizable. The simplest implementations of code completion, such as used in Notepad++, propose all the words that are present in the current document or project without any regard for their semantics or the context at the caret. This will propose keywords and identifiers equally, regardless of whether the keywords are actually correct, whether declarations are actually reachable at the caret position, and whether the proposed identifier inadvertently captures another name. While this is language-agnostic, and thus easy to implement, it is also the least useful.

More advanced code completion implementations propose only references to declarations that are reachable and visible from the caret position, according to the static semantic rules of the language, and these are the most common form of code completion in IDEs. For example, Visual Studio Code, Visual Studio, Eclipse, and IntelliJ use this kind of code completion by default for the languages they fully support. While this does not prevent the user from inserting a reference of the wrong type, it does allow the user to discover the available names. Additionally, inserting a reference of the wrong type is often useful when the reference is used as the receiver of a chain of method calls and field accesses.

Syntax-aware code completion, such as proposed by de Souza Amorim et al. [2016], propose only syntax fragments that can be inserted at the caret position according to the syntactic rules of the language. However, this approach does not take the semantics of the fragment into account, allowing, for example, a Boolean and-expression to be inserted as the initializer for an integer variable. Furthermore, syntactic code completion does not take the binding context into account and does not propose references to actual variables and functions.

Type-directed development in languages such as Agda [Norell 2007] and Idris [Brady 2013] uses programs with explicit holes to represent incompleteness. Completion commands can involve proof search to determine valid proposals. Unfortunately, such proposals are not always guaranteed to be sound. Ideally, code completion in a static context proposes only syntax and references that are valid according to both the syntactic and static semantic rules of the language. Selecting any proposal provided by this code completion will not introduce any additional syntactic or static semantic errors in the program.

Code completion is often reimplemented for each language and for each editor supporting that language. Such implementations may require significant development effort, and vary in features, soundness, and completeness. Instead, we would like to derive a sound and complete code completion editor service from the formal definition of a language automatically, to reduce development effort and increase reliability. Language workbenches [Fowler 2005] aim to reduce the effort required to implement a programming environment for a language. Language workbenches provide significant support for developing parsers and type checkers from language definitions. However, the implementation of code completion in language workbenches such as Xtext [Xtext Team 2021],

MPS [Jetbrains 2021], Rascal [Klint et al. 2009, 2010, 2019], and Spoofax [Kats and Visser 2010] either require additional work on top of a language definition, are unsound, and/or incomplete. (We discuss the state-of-the-art of code completion in language workbenches extensively in section 7.) In their vision paper, Pelsmaeker et al. [2019] speculate on the derivation of a range of editor services from the declarative, constraint-based specification of the static semantics of a language. We took their proposal to heart to automatically derive code completion from a language definition.

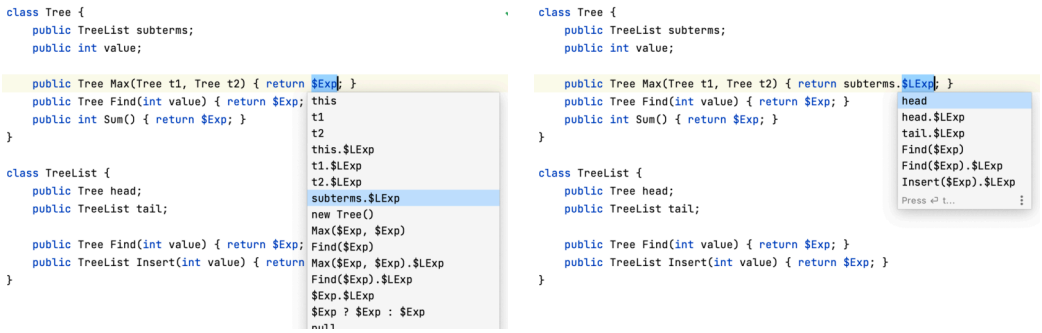
In this paper, we develop a framework for language-parametric semantic code completion for statically typed programming languages based on their syntax definition and static semantics specification, realizing the implementation of a code completion editor service with minimal additional effort. The framework builds on the SDF3 syntax definition formalism [de Souza Amorim and Visser 2020] and the Statix static semantics specification language [Rouvoet et al. 2020; van Antwerpen et al. 2016] and is integrated into the Spoofax language workbench [Kats and Visser 2010]. Our code completion algorithm reinterprets the declarative static semantics definition of a language in Statix. We use its order-independent constraints and syntax-directed predicates to drive a search strategy that uses the scope graph and Statix solver to explore the space of possible code completion solutions in a language-parametric way. Our Statix reinterpretation includes the implementation of name resolution queries with non-ground subjects. The implementation of the strategy language and code completion algorithm abstracts and extends the implementation of the Statix solver and can be used for any language defined in Statix. We evaluate the performance, soundness, and completeness of completion proposal synthesis.

In summary, our contributions are:

- a language-parametric code completion editor service that is derived automatically from the static semantics of a language;
- code completion proposals that are sound and complete by construction;
- a definition of code completion as a language-parametric constraint resolution problem on incomplete programs with holes/placeholders using syntax-directed semantic rules in Statix, see section 4;
- extensions to the Statix solver interface to expand predicate constraints into their syntax-directed alternatives, and to expand query constraints into their possible resolutions;
- a set of strategy combinators to compose these primitives, and the composition of a search strategy for sound and complete code completion, as presented in section 5;
- the integration of our language-parametric semantic code completion in the IntelliJ and Eclipse editors generated by the Spoofax language workbench; and
- the evaluation of the algorithm by applying it to language definitions for Tiger [Appel 2002] and ChocoPy [Padhye et al. 2019] in section 6. Using test suites of 135 Tiger and 136 ChocoPy programs (60 624 test cases of up to 1 091 lines of code) we determined soundness and completeness, and measured performance. The median performance of code completions is 422 ms, but has outliers exceeding 4 seconds for certain large test files with many declarations in scope.

We submitted the related source code and benchmark files as an artifact [Pelsmaeker et al. 2022].

Outline. We proceed as follows. In the next section, we analyze the requirements for sound and complete code completion. In section 3 we discuss the prerequisites for our approach. In section 4 we show the code completion algorithm on a running example, and its implementation using search strategies is detailed in section 5. In section 6 we evaluate the soundness, completeness, and performance of our implementation. We discuss related work in section 7 and future work in section 8. We conclude in section 9.



(a) Completion on placeholder of type `Tree`.

(b) Completion on the inserted proposal.

Fig. 1. Code completion in action: the user invokes code completion on the selected placeholder and gets presented a complete list of completion proposals that are syntactically and semantically sound.

2 CODE COMPLETION

In this section, we discuss the requirements for sound and complete semantic code completion. Throughout the paper, we use Micro-C♯, a small subset of the C♯ language, as running example object language. Figure 1a shows how code completion displays a list of completion proposals when the user performs code completion on the selected syntactic placeholder `$Exp`. The list includes both syntactic proposals, such as the ternary operator `$Exp ? $Exp : $Exp`, as well as references, such as to variable `t1` and method call `Max($Exp, $Exp)`.

Representing Incompleteness. Code completion is often, but not necessarily, invoked on an incomplete program, where one of the proposals is used to get one step closer to a complete program. For the purpose of this paper, we assume that the program has a syntactic placeholder near the caret position to represent a hole in an incomplete program. In the Spoofax language workbench, a placeholder is typically named after the syntactic sort of the missing phrase. For example, `$Exp` is a placeholder representing an expression, which ensures that incomplete programs are syntactically correct. Placeholders can be inserted using placeholder inference [de Souza Amorim et al. 2016], which can infer placeholders for both complete and incomplete programs, or as part of the parsing process of a syntactically incomplete program.

Syntactic Code Completion. As a first step, code completion should propose only syntactically correct fragments. de Souza Amorim et al. [2016] describe the synthesis of a complete list of syntactically correct proposals following the grammar of the language, by proposing phrases corresponding to the productions of the syntactic sort of the placeholder. Incomplete sub-phrases are represented by new placeholders. For example, a syntactically valid proposal for `$Exp` is `$Exp + $Exp`.

Filtering on Types. When only considering the syntactic sort of the placeholder, not all syntactically correct proposals will produce fragments that are also type correct. For example, completing an expression in a context where an integer typed expression is expected will also propose expressions that produce Boolean typed expressions. Therefore, code completion should only propose syntactically correct fragments that are also type correct in the completion context. Depending on the language, there might be additional static semantic constraints on proposals, such as subtype and supertype constraints, or shape constraints on vectors in dependently typed systems. Completion proposals should adhere to such constraints.

Completing Names. Just proposing the syntax of an unspecified reference or function call (e.g., `$ID` or `$ID($Args)`) is often not very useful. Ideally, code completion should propose the actual names of variables and calls to functions that are available in the scope of the completion context. We could even improve upon this by proposing function calls with placeholders for the correct number of arguments, or record initialization syntax with initializers for each of the fields. This requires computing the names that are available at the completion context, taking into account name binding rules such as scoping, shadowing, importing, and overriding.

Soundness and Completeness. Finally, code completion should be sound and complete. Soundness ensures that inserting the proposal does not introduce additional syntactic or static semantic errors. This is achieved by taking into account the guidelines above to only propose phrases that are syntactically and semantically correct at the point of insertion with respect to the static semantics of the language. Completeness ensures that *all* syntactic constructs and reachable identifiers (modulo shadowing) that are permitted according to the syntax and static semantics of the language are proposed, up to one level deep, modulo literals and the identifiers of declarations. With a complete code completion algorithm, one can (re)create any valid program through repeated application of code completion, but repeated application of code completion does not necessarily result in a complete program. In this paper, we determine whether a program is valid by checking it against the static semantics specification of the language. Therefore, our notion of completeness is limited to what is possible to express and check with the static semantic rules.

3 BACKGROUND: LANGUAGE DEFINITION IN SDF3 AND STATIX

Our code completion editor service is derived from the formal definition of the syntax and static semantics of a language. We build on the SDF3 syntax definition formalism [de Souza Amorim and Visser 2020] and Statix static semantics specification language [Rouvoet et al. 2020; van Antwerpen et al. 2018] and their implementations in the Spoofax language workbench [Kats and Visser 2010], but note that SDF3 and Statix are not contributions of this paper. Figures 2 to 4 show excerpts of the SDF3 and Statix specifications of our Micro-C♯ language (the full specifications are in appendix A and appendix B, respectively). The rest of this section provides a high-level introduction to SDF3 and Statix to illustrate key features for this paper. We refer to the cited literature for more details.

3.1 Syntax Definition in SDF3

SDF3 is a language for the formal definition of the syntax of (domain-specific) programming languages [de Souza Amorim and Visser 2020]. In addition to a basic syntactic description based on context-free grammars, the language provides several features to aid the definition of language-aware editors. The syntactic rules relevant for this subsection are in fig. 2.

Concrete and Abstract Syntax. The syntax of language constructs is defined through context-free productions of the form `A.C = `α with `A` a non-terminal, `C` a constructor symbol, and α a list of symbols. The constructor symbol creates a direct correspondence between parse tree and abstract syntax tree. For example, the expression `1 + x` has abstract syntax tree `Add(Int("1"), Var("x"))` according to the productions in fig. 2. Keywords and layout (whitespace and comments) are omitted from the abstract syntax.

Template Productions. SDF3 extends plain context-free grammar productions with *template* productions to support pretty-printing abstract syntax trees to text. A template quotes a text template with anti-quotation for non-terminal symbols. For example, the template `MemAcc.Call = <<ID>(<{Exp ", "}*>)>` specifies that no spaces should be used around the parentheses of the call, but that spaces *should* be inserted after the commas between arguments. This entails that code

completion can generate an abstract syntax tree as a completion proposal, which can then be translated automatically to a pretty-printed concrete syntax fragment.

Placeholders. SDF3 already provides language-parametric support for *syntactic* code completion by generating an editor service that synthesizes completion proposals based on a syntax definition [de Souza Amorim et al. 2016]. Incompleteness in a program is represented explicitly through placeholders, syntax which is added automatically to the syntax definition. That is, for each syntactic sort `S`, a production `S.S-Plhdr = "$S"` is automatically generated. For example, `2 + $Exp` represents an expression with a hole. Code completion is a transformation on the abstract syntax tree, transforming placeholders to AST representations of completion proposals. For example, an *addition* proposal is synthesized by rewriting `Exp-Plhdr()` to `Add(Exp-Plhdr(), Exp-Plhdr())`. Textual completions are produced by pretty-printing the AST proposals.

Disambiguation. Instead of encoding the disambiguation rules of a language in the productions of the syntax definition, they are encoded in separate associativity and priority declarations. For example, the sentence `a * b + c` is parsed as `Add(Mul(Var("a"), Var("b")), Var("c"))` since multiplication has higher priority than addition, as specified in fig. 2. These declarative disambiguation rules are not just used for parsing, but also during pretty-printing to ensure that brackets are inserted where necessary to preserve the abstract syntax tree structure. For example, when the placeholder in `a * $Exp` is expanded into an addition, the result is pretty-printed as `a * ($Exp + $Exp)`.

Syntax API. In summary, SDF3 provides a representation and API for syntax definition, parsing, and pretty-printing, including support for the representation of incomplete programs and syntactic completion proposals.

3.2 Static Semantics in Statix

Statix is a declarative language for describing the static semantics of programming languages in terms of equality and scope graph constraints [van Antwerpen et al. 2018]. The Statix implementation automatically derives a type checker from a specification. The implementation is based on an operational semantics that is sound with respect to the declarative semantics [Rouvoet et al. 2020]. In the rest of this paper, we will show how we automatically derive a code completion editor service from a Statix specification. In this section, we give a brief overview of the relevant features of Statix. Excerpts from the Statix specification of Micro-C♯ in appendix B are shown in fig. 3 and fig. 4.

Equality Constraints. Statix is a constraint-based specification language. The basic constraints are equality constraints `t1 == t2` and inequality constraints `t1 != t2` on terms. An equality constraint is satisfiable if its term arguments are equal or can be unified. An inequality constraint is satisfiable if its term arguments are not equal. The quantification of existentials `{T}` enables specification of type inference. The constraints are order-independent, which we use in our code completion algorithm to be able to explore different alternatives in parallel by expanding upon unsolved constraints.

Specifying Name Binding with Scope Graphs. Scope graph constraints [van Antwerpen et al. 2016] are a key feature of Statix. A scope graph is an abstraction of a program to its name binding facts. A scope is a vertex in the graph that corresponds to a region of a program that behaves uniformly with respect to name binding. Declarations of names are data in scopes. Edges between scopes encode reachability. A declaration in a scope is reachable from all scopes with paths (sequences of edges) to that scope.

```
context-free sorts
  Exp LValue LExp MemAcc

context-free syntax
  Exp.New       = <new <Type>(<>)>
  Exp.Int       = <<INT>>
  Exp.Mul       = <<Exp> * <Exp>>    {left}
  Exp.Add       = <<Exp> + <Exp>>    {left}
  Exp.Cond      = <<Exp> ? <Exp> : <Exp>>
  Exp           = <(<Exp>)>       {bracket}
  Exp           = LValue
  LValue.Member = <<Exp>.<LExp>>    {right}
  LValue        = LExp
  LExp.Project  = <<MemAcc>.<LExp>>{right}
  LExp          = MemAcc
  MemAcc.Var    = <<ID>>
  MemAcc.Call   = <<ID>(<{Exp ", "}*>)>

context-free priorities
  { Exp.New LExp.Project
    MemAcc.Var MemAcc.Call
  } > Exp.Mul > Exp.Add > Exp.Cond,

  LValue.Member <0> .> Exp = LValue,

  LValue.Member <0> .> { Exp.New Exp.Mul
    Exp.Add Exp.Cond }
```

Fig. 2. Excerpt of Micro-C♯ syntax definition in SDF3

```
declareVar : scope * ID * TYPE
declareVar(s, decl, T) :-
  !type[Var{decl@decl}, T] in s.

typeOfVar : scope * ID → TYPE
typeOfVar(s, name) = T :- {decl}
  resolveVar(s, name) == [(_,(decl,T))|_].

resolveVar : scope * ID → list(DECL)
resolveVar(s, name) = ps :-
  query type
    filter P*
    and { d' :- d' == Var{name@_} }
    min $ < P
    in s ⟼ ps.
```

Fig. 3. Excerpt of Micro-C♯ name resolution in Statix

```
typeOfExp(s, New(t)) = T :- {cs}
  typeOfType(s, t) == T@CLASS(_, cs).

typeOfExp(s, Add(e1, e2)) = INT() :-
  subtypeOf(typeOfExp(s, e1), INT()),
  subtypeOf(typeOfExp(s, e2), INT()).

typeOfExp(s, Cond(ec, e1, e2)) = T :-
  {T1 T2}
  subtypeOf(typeOfExp(s, ec), BOOL()),
  typeOfExp(s, e1) == T1,
  typeOfExp(s, e2) == T2,
  lub(T1, T2) == T.

typeOfExp(s, LValue2Exp(lvalue)) =
  typeOfLValue(s, lvalue).

typeOfLValue(s,
  LExp2LValue(MemAcc2LExp(Var(x))))
  = typeOfVarOrMember(s, x).

typeOfLValue(s, LExp2LValue(le)) =
  typeOfLExp(s, s, le).

typeOfLValue(s, Member(e, le)) = T :- {cs}
  typeOfExp(s, e) == CLASS(_,cs),
  typeOfLExp(s, cs, le) == T.

typeOfLExp(s, cs, Project(ma, le)) = T :-
  {T1 cs2}
  typeOfMemAcc(s, cs, ma) == CLASS(_,cs2),
  typeOfLExp(s, cs2, le) == T.

typeOfLExp(s, cs, MemAcc2LExp(ma)) =
  typeOfMemAcc(s, cs, ma).

typeOfMemAcc(s, cs, Var(name)) = T :-
  typeOfMember(cs, name) == T.

typeOfMemAcc(s, cs, Call(x, es)) = TR :-
  {TA}
  typeOfMember(cs, x) == METHOD(TA, TR),
  typesOfExps(s, es) == TA.

typeOfExp(_, Exp-Plhdr()) = BOTTOM().
```

Fig. 4. Excerpt of Micro-C♯ type rules in Statix

The language agnostic representation allows high-level reasoning about name resolution [Néron et al. 2015]. A named reference is bound to its corresponding declaration through a name resolution query constraint. A name resolution query for a reference starts in a scope to find a path along edges to a matching declaration. References are ambiguous if multiple matching declarations can be reached. Disambiguation can be expressed through path well-formedness and specificity ordering on paths. For example, the query in fig. 3 defines lexical shadowing of variables by finding paths along zero or more `P` edges (`P*`), preferring shorter paths to longer paths (`$ < P`), where `P` edges in the scope graph represent the lexical parent relation between scopes. For code completion we use the name resolution and scope graph of Statix to find all reachable declarations for query constraints where the subject is not ground.

Predicates. Predicate symbols are used to introduce user-defined constraints that abstract from basic unification and scope graph constraints. In general, a predicate defines a relation on terms and scopes. Rules define whether a predicate holds for a combination of terms. A rule has the form $p(t_1, \ldots, t_n)$ `:-` $c_1, \ldots, c_n$ and defines a predicate p through the conjunction of a number of constraints $c_1, \ldots, c_n$. A predicate can be defined by means of multiple rules, as long as the heads of the rules are mutually exclusive (including specificity, see below). (Or in operational terms, rule selection in Statix is non-backtracking.) Additionally, rule selection is 'syntax-directed', which we use in our algorithm to be able to propose syntax fragments.

A functional predicate defines a one-to-one relationship between input terms and output term. For example, the predicate `typeOfExp(s, e) = T` associates a single type `T` with the combination of a scope `s` and an expression `e`. From the perspective of specifying static semantics, a predicate is interpreted as an assertion about the well-boundness and well-typedness of abstract syntax terms. For example, this rule defines that a method call has type `TR` if the method has type `METHOD(TA, TR)` and the arguments have types `TA`:

```
typeOfMemAcc(s, cs, Call(x, es)) = TR :- {TA}
  typeOfMember(cs, x) == METHOD(TA, TR),
  typesOfExps(s, es) == TA.
```

Rules with overlapping patterns are ordered by means of a specificity ordering. That is, rules with a more specific pattern in its head have priority over (and exclude) rules with more general patterns. Furthermore, the binding of patterns in rule heads is done through pattern matching rather than unification, thus the (input) arguments of predicates needs to be ground (enough).

From Declarative to Operational Semantics. Statix has a declarative semantics that defines when a solution satisfies a set of constraints. A language engineer should only have to consider the declarative semantics. The Statix solver implements the operational semantics of Statix, which schedules the solution of query constraints such that their results are guaranteed to be stable, i.e., provide the same results as they would in the final scope graph [Rouvoet et al. 2020].

In section 5, we extend the interface of the solver in order to give alternative interpretations to Statix specifications. In particular, we use the solver to get all possible alternatives of a predicate and all possible resolutions of a query and use these in the code completion algorithm to explore the solution space and synthesize proposals. Finally, we use the solver to reject unsound proposals.

4 CONSTRAINT-BASED CODE COMPLETION

Given an incomplete program, the goal of code completion is to synthesize a list of completion proposal phrases that can be substituted for the placeholder being completed, such that the resulting program is well-bound and well-typed. That is, the program after substituting a completion proposal for a placeholder should not violate any constraints when type checked (that were not

violated before invoking code completion). The key idea of our approach is to use the constraint-based definition of the type checker to synthesize the completion proposals, in order to guarantee that the proposals are well-bound and well-typed by construction.

The core of our language-parametric code completion algorithm is a search strategy that explores the solution space through the static semantics specification of the object language. In this section we explore that search strategy by tracing the three steps of the synthesis of the list of completion proposals for the example program in fig. 1a, *predicate expansion*, *injection expansion*, and *query expansion*, using the rules from figs. 3 and 4. In the next section, we give a formal definition of the algorithm.

4.1 Preliminary Analysis

The preliminary step in the code completion algorithm is the analysis of the incomplete program, in which we replace the placeholder to be completed (the first `$Exp` in fig. 1a) with a constraint variable. We will refer to this constraint variable as the *completion variable*. Synthesizing completion proposals entails finding all terms that can be substituted for the completion variable without violating any constraints.

Constraint solving in Statix first solves basic constraints and simplifies predicates via rule application. Statix only simplifies predicates if the input arguments are ground enough. That is, the arguments are at least as ground as the patterns in the heads of the predicate rules. Constraints that are not ground enough cannot be solved, which is typically the case for constraints involving the completion variable. For the example program in fig. 1a, we have the following residual constraint after the preliminary analysis

```
typeOfExp(s, e) == CLASS("Tree", cs).
```

with `e` being the completion variable corresponding to placeholder `$Exp`, `s` the scope of the completion variable, and `cs` the scope of the class definition. These unsolved constraints are the starting point for the code completion algorithm.

4.2 Predicate Expansion

In the first step of the code completion algorithm, the completion variable is expanded by unifying the residual constraint with the heads of predicate rules, leading to its instantiations. The syntax-directed nature of the rules ensures that all terms that are type correct will be considered. Each rule leads to a possible alternative. We will examine several representative rules.

```
typeOfExp(s, Cond(ec, e1, e2)) = T :- {T1 T2}
  typeOfExp(s, e1) == T1,    typeOfExp(s, e2) == T2,
  subtypeOf(typeOfExp(s, ec), BOOL()),  lub(T1, T2) == T.
```

The above rule for the *conditional* expression declares that a *conditional* expression with condition `ec`, then-expression `e1` and else-expression `e2` is well-typed when the type of the condition is a subtype of Boolean, and the types of the branches `e1` and `e2` share a least-upper-bound type `T`, which has to equal the type of the *conditional* expression as a whole. Unifying the residual constraint to the head of this rule leads to the constraints:

```
e == Cond(ec, e1, e2),  CLASS("Tree", cs) == T,  subtypeOf(typeOfExp(s, ec), BOOL()),
typeOfExp(s, e1) == T1, typeOfExp(s, e2) == T2,  lub(T1, T2) == T.
```

Since the unification constraint `CLASS("Tree", cs) == T` is trivially satisfiable and the other constraints are non-ground, this leads to the completion proposal `Cond(ec, e1, e2)`, or `$Exp ? $Exp : $Exp` in concrete syntax. Note that the predicate expansion step is *only applied once* to a non-ground constraint. Recursive application of this strategy to the residual, non-ground constraints

synthesized by this step could lead to infinitely many proposals, even though each expansion only introduces a finite number of expansions. However, not all rules lead to a completion proposal. For example, consider the rule for *addition*:

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  subtypeOf(typeOfExp(s, e1), INT()),  subtypeOf(typeOfExp(s, e2), INT()).
```

Unifying the residual constraint to the head of this rule leads to the constraints

```
e == Add(e1, e2),                    INT() == CLASS("Tree", cs),
subtypeOf(typeOfExp(s, e1), INT()),  subtypeOf(typeOfExp(s, e2), INT()).
```

Since the unification constraint `INT() == CLASS("Tree", cs)` is not satisfiable, this alternative is discarded and does not lead to a completion proposal. And it should not, since `Add(e1, e2)` would lead to an ill-typed term when substituted for the placeholder. Finally, consider the rule for *new*:

```
typeOfExp(s, New(t)) = T :- {cs}
  typeOfType(s, t) == T@CLASS(_, cs).
```

Unifying the residual constraint to the head of this rule leads to the constraints

```
e == New(t),   T == CLASS("Tree", cs),
typeOfType(s, t) == T@CLASS(_, cs).
```

Since the unification with `T` is satisfiable, this leads to a completion proposal `New(t)` (or `new($Type)` in concrete syntax), requiring the type `t` to be `Tree`. Following this procedure also for the rule `typeOfExp(s, LValue2Exp(lvalue))`, we arrive at the following list of proposals:

```
$Exp ? $Exp : $Exp,    new $Type(),    $LValue
```

This list only includes well-typed (syntactic) completion proposals, but includes another placeholder `$LValue`. We will improve this in the next step.

4.3 Injection Expansion

An injection is a grammar production of the form `S1 = S2`, that includes all `S2` constructs in `S1`. For example, in the Micro-C♯ syntax definition we have defined the `LValue` sort to be used as expressions and as the left-hand side of assignments. The inclusion of `LValue` in `Exp` is an injection since no additional syntax is added. In fact, there is a chain of injections `Exp = LValue = LExp = MemAcc` to include variables and calls as expressions:

```
Exp.Add       = <<Exp> + <Exp>>   {left}
Exp.Cond      = <<Exp> ? <Exp> : <Exp>>
Exp           = <(<Exp>)>      {bracket}
Exp           = LValue
LValue.Member = <<Exp>.<LExp>>   {right}
LValue        = LExp
LExp.Project  = <<MemAcc>.<LExp>>{right}
```

In Statix, injections get an explicit constructor. For example, the `Exp = LValue` production becomes `Exp.LValue2Exp = LValue` with typing rule:

```
typeOfExp(s, LValue2Exp(lvalue)) == typeOfLValue(s, lvalue).
```

Thus, the expansion procedure above expands the completion variable to

```
e == LValue2Exp(lvalue), typeOfLValue(s, lvalue).
```

leading to the completion proposal `LValue2Exp(lvalue)` (`$LValue` in concrete syntax); not a very useful proposal.

Recognizing the special status of injections, the second step of the algorithm expands injections, which includes the constructs of the sub-sort in the set of proposals. That is, recognizing

`LValue2Exp(lvalue)` as an injection, marks `lvalue` as a completion variable to be considered for expansion, following the procedure of predicate expansion above. Considering the full set of typing rules for LValues in fig. 4, code completion now replaces the `$LValue` proposal with `Member(e, le)`, `Project(m, le)`, `Var(x)`, and `Call(x, es)` (or `$Exp.$LExp`, `$MemAcc.$LExp`, `$ID`, and `$ID($Exps)` in concrete syntax), leading to the following list of proposals:

```
$ID,            $ID($Exps),            $Exp.$LExp,
$Exp ? $Exp : $Exp,   new $Type(),   $MemAcc.$LExp
```

The list contains only well-typed syntactic completion proposals. This includes placeholders for referencing variables and calling methods, but not yet the actual identifiers of those that are in scope. We will address this next.

4.4 Query Expansion

So far we have been concerned with synthesizing completion proposals that are syntactically correct and type correct. However, to get truly useful proposals, we also want to propose those identifiers of the right type that are in scope. As the proposals above have constraint variables instead of concrete names, this requires performing name resolution queries with non-ground subjects.

To understand the issues involved, we examine the case of variables. One rule for `typeOfLValue` uses the `typeOfVarOrMember(s, name)` predicate to assert that the name refers to a valid local variable or class member. For the sake of this presentation, we simplify that to just asserting that the name refers to a valid local variable, defined by the following rules:

```
typeOfExp(s, LValue2Exp(lvalue)) =
  typeOfLValue(s, lvalue).
typeOfLValue(s, LExp2LValue(MemAcc2LExp(Var(x)))) =
  typeOfVar(s, x).
typeOfVar(s, name) = T :-
  resolveVar(s, name) == [(_, (_, T))].
resolveVar(s, name) = ps :-
  query type filter P* and { d' :- d' == Var{name@_} } min $ < P in s ⟼ ps.
```

The predicate `typeOfVar` is just an abstraction for a pattern match on the result of the name resolution query defined by `resolveVar`. But it does illustrate that after injection expansion, regular constraint resolution by the Statix solver unfolds the predicate `typeOfExp(s, e)` into the constraints:

```
e == LExp2LValue(MemAcc2LExp(Var(x))), T == CLASS("Tree", cs), ps == [(_, (_, T))],
query type filter P* and { d' :- d' == Var{x@_} } min $ < P in s ⟼ ps.
```

In other words, we are looking for a concrete name for the `x` constraint variable, such that the query results in a path to a single declaration with the type `CLASS("Tree", cs)`.

The query expansion step of the algorithm interprets a non-ground query and produces all possible alternatives. That is, it finds all names of declarations such that executing the query with that name makes the query succeed. We describe the technical details of query expansion in section 5. For the example program of fig. 1a, expanding the query finds the arguments `t1` and `t2` of type `CLASS("Tree", _)` and the instance variable `subterms` of type `CLASS("TreeList", _)`. Since we are looking for an expression of type `Tree`, only `t1` and `t2` end up in the list of proposals. Similarly, we find completion proposals for `Calls` to methods `Max` and `Find`, since they are in scope and return `Tree` objects.

```
t1,     t2,     $ID,     Max($Exps),     Find($Exps),     $ID($Exps)
$Exp ? $Exp : $Exp,     new $Type(),    $Exp.$LExp,    $MemAcc.$LExp,
```

4.5 Rule Specialization

Consider again the syntax of method calls and variables in section 4.3. As we saw above, we do get proposals for variables and method calls, but by default, the algorithm will propose quite shallow syntax, only one level deep. However, to create a chain of dot expressions, we need to first complete `$Exp` to `$MemAcc.$LExp` and then complete the `$MemAcc` placeholder to a call. This would have to be done by the user, as the code completion algorithm only synthesizes one level of expansion in order to avoid infinite recursion.

While this approach is sound and complete, in cases like this we would like code completion to also expand nested constraint variables, whereas in other cases the current behavior is fine, such as in `$Exp + $Exp`. Resolution queries are a possible indication that it is desirable to further expand a constraint. However, queries may be hidden behind a series of predicate abstractions as we saw in the case of variables above. We have not succeeded in identifying a reliable heuristic to selectively expand constraints in these particular cases.

Fortunately, we have identified *rule specialization* as a simple approach for the language engineer to direct the code completion algorithm to provide more concrete and useful proposals for these particular cases, without code duplication and without affecting the generality of the definition or the soundness of the approach. For example, consider the following rules for member access projection (dot expressions):

```
// Rules for method calls
typeOfLExp(s, cs, lv@Project(_, _))           = typeOfProjectExp(s, cs, lv).
typeOfLExp(s, cs, lv@Project(Var(_), _))      = typeOfProjectExp(s, cs, lv).
typeOfLExp(s, cs, lv@Project(MCall(_, _), _)) = typeOfProjectExp(s, cs, lv).

typeOfProjectExp(s, cs, Project(ma, le)) = T :- {T1 cs2}
  typeOfMemAcc(s, cs, ma) == CLASS(_, cs2), typeOfLExp(s, cs2, le) == T.
```

Here we have captured the rule for dot expressions in a new predicate `typeOfProjectExp`. We define the `typeOfLExp` predicate for the general case of a dot expression and we extend it with a rule for the special cases `Project(Var(_), _)` and `Project(MCall(_, _), _)`. This instructs the predicate expansion of section 4.2 to also synthesize completion proposals for these cases. Applying such rule specializations to `New` as well, leads to a list of more specific completion proposals in addition to the generic syntactic proposals:

```
t1,     t2,     $ID,     Max($Exps),     Find($Exps),     $ID($Exps)
t1.$LExp,     t2.$LExp,     subterms.$LExp,     this,     this.$Exp,
new Tree(),       new $Type(),       $MemAcc.$LExp,      $Exp.$LExp,
Max($Exps).$LExp,       Find($Exps).$LExp,       $Exp ? $Exp : $Exp.
```

Thus, by a slight refactoring of the specification, we can guide the code completion algorithm to produce more useful completion proposals. A rule specialization adds only a single additional rule to be expanded by predicate expansion, and therefore does not incur more performance overhead than the other rules in the specification.

4.6 Filtering and Ranking

The completion proposals often include some placeholders for unspecified literals (e.g., string literals, integers, and identifiers), such as the `$ID` placeholder for an identifier. In general, it is impractical to expand on these placeholders and list all possible literal values, but for completeness and to aid in language discovery, we decided not to filter these placeholders from the list of proposals.

Ranking the completion proposals aims to offer the most likely choices to the user first. However, as ranking the proposals is out of the scope of this paper, we chose to simply rank them by

specificity, putting the proposal with the least number of placeholders first. Thus, we present to the user the final list of proposals for the completion of `$Exp` in example program of fig. 1a:

```
t1,       t2,       this,       new Tree(),       subterms.$LExp,
t1.$LExp,    t2.$LExp,    this.$Exp,    Find($Exps),    Max($Exps),
$ID,     new $Type(),     Find($Exps).$LExp,     Max($Exps).$LExp,
$ID($Exps),    $MemAcc.$LExp,    $Exp.$LExp,    $Exp ? $Exp : $Exp,
```

4.7 Applying Code Completion

A primary motivation of this work is that enabling language-parametric semantic code completion for a Spoofax language takes very little additional effort. Any Spoofax language which has its static semantics defined using Statix requires only adding rules that permit placeholders and changing rules that catch all remaining constraints, to enable sound and complete static semantic code completion.

Rules for Placeholders. Syntactic placeholders in Spoofax are just constructor application terms of the form `Exp-Plhdr()`. As placeholders are inserted explicitly when choosing a completion proposal, the language developer has to add some extra rules to the Statix specification to allow it to accept these placeholder terms, as shown in fig. 5. In the future, these rules could be generated automatically based on the syntactic and semantic specifications.

```
rules // placeholders
  programOk(Module-Plhdr()).
  declOk(_, Decl-Plhdr()).
  typeOfExp(_, Exp-Plhdr()) = _.
```

Fig. 5. Example static semantic rules for accepting placeholders.

Catch-All Rules. Some language developers write *catch-all* Statix rules that, for example, generate a warning about unsupported syntax. As Statix rules are tried in order of specificity, such catch-all rules are tried last and therefore work as expected during the normal code analysis. However, such rules consume the remaining constraints that apply to the placeholder being completed. This prevents code completion from finding completions, as there will be no constraints left for the code completion algorithm to expand upon and this impacts completeness.

```
typeOfExp(_, _) = _ :- try { false } | warning $[Not implemented].
```

Instead, the language developer should be more specific about the syntax. For example:

```
typeOfExp(_, Plus(_, _)) = _ :- try { false } | warning $[Not implemented].
```

5 CODE COMPLETION ALGORITHM

In the previous section, we illustrated the main steps of the code completion algorithm. In this section, we formally define the algorithm as a composition of basic solver functions using a strategy language to search the solution space. In the actual implementation, we use Java, the implementation language of the Spoofax language workbench [Kats and Visser 2010], to implement the strategies. However, we use Haskell notation to present the algorithm in this paper because Java is quite verbose.

5.1 Constraint Solver

The code completion algorithm builds on the Statix constraint solver, which performs syntax-directed constraint solving and unification based on the operational semantics of Statix [Rouvoet et al. 2020]. The solver takes a declarative specification and a solver state (shown in fig. 6), which

consists of a set of unsolved constraints, a unifier, a scope graph, and a set of error messages, and some other data that is only used by the Statix solver internally.

When analyzing a file during normal error-checking operation, the solver gets an initial solver state with only the program's root predicate constraint (e.g., `programOk()`) applied to the AST of the program, represented as a *term*. For example:

```
programOk(Program([ Class("TreeList", ...), Class("Tree", ...) ])).
```

The solver expands predicate constraints that are sufficiently ground into their constituent constraints, and then performs unification to remove as many solved (or unsolvable) equality constraints from the solver state as possible. Equality constraints whose variables the solver cannot unify contribute errors to the set of errors. The remaining unsolved constraints, those that are not ground enough, remain in the resulting solver state.

Code completion is performed on an incomplete program, where syntactic placeholders represent (typed) holes in the program. These holes are represented by constraint variables in the solver state, which inevitably results in some unsolved constraints due to not being ground enough. The core idea of the code completion algorithm is to use the solver to try different expansions of the solver state's unsolved constraints to explore the space of possible solutions for a particular *completion variable*, the constraint variable that corresponds to the syntactic placeholder being completed.

```
data SolverState = SolverState {
  constraints :: Set Constraint
, unifier :: Unifier
, scopeGraph :: ScopeGraph
, messages :: Set Message
, spec :: Spec
} deriving Eq;
```

Fig. 6. Solver state structure

5.2 Search Strategies

To explore the space of possible code completion solutions, we employ a language-parametric *search strategy* that is itself a composition of other *strategies*. A strategy is a function of type `a → [b]`, which defines a computation that takes an input and optional arguments, and produces a list of alternatives. If the list is empty, the strategy is said to have failed. In particular, many of these strategies are *search strategies* of type `SolverState → [SolverState]`, which take a solver state and produce a (possibly empty) list of solver states. We compose the strategies using *strategy combinators*, which are basic primitive strategies that take other strategies as parameters.

Search strategies give us a functional way to compose more basic strategies, and the flexibility to experiment with different strategy compositions to find the most optimal implementation. Additionally, we expect that other editor services that use the static analysis of a program could be implemented language-parametrically using different compositions of strategies.

5.3 Built-in Search Strategies

The code completion algorithm uses the existing Statix solver, persistent solver state structure, name resolution algorithm, and the static semantics specification of the object language. Additionally, the algorithm requires access to the pretty-printer of the object language, which is automatically generated from the language's syntax specification.

Our contribution is an abstraction of these details of the Statix solver and Spoofax internals into a handful of search strategies, shown in fig. 7, providing a well-defined interface for their interaction with the code completion algorithm.

```
type State = SolverState
type Constr = Constraint

selectPredicate     :: ((Constr, State) → Bool) → State → [(Constr, State)]
selectQuery         :: ((Constr, State) → Bool) → State → [(Constr, State)]
expandPredicate     :: (Constr, State) → [State]
expandQuery         :: (Constr, State) → [State]
infer               :: State → [State]
assertNoErrors      :: Set Message → State → [State]

containsVar         :: [Var] → Constr → Bool
findInjection       :: Var → State → [(Var, State)]
projectVar          :: Var → State → [Term]
prettyPrint         :: Term → [String]
```

Fig. 7. Built-in search strategies.

Selecting Constraints. The `selectPredicate` and `selectQuery` strategies inspect the solver state, select a number of predicate or query constraints, respectively, and return these as constraint-state pairs. The function that is passed as the first argument determines whether a constraint is selected, such as the `containsVar` strategy which is used to check whether the constraint contains the completion variable. These constraint-state pairs are the input to one of the built-in `expandPredicate` and `expandQuery` strategies.

Expanding a Predicate. The `expandPredicate` strategy takes a constraint-state pair and determines all possible rules that the predicate constraint could expand to. For each rule, the strategy produces a new solver state with the rule's constraints added to it. In Statix, as rules are tried in order of specificity, a more general rule for a predicate implies that all more specific rules were already tried and rejected. Because our algorithm has to produce order-independent expansions, we make these rejections of more specific rules explicit in the expansion of each rule. Section 4.2 gives examples of the application of this strategy. The `expandPredicate` strategy itself does not verify whether the resulting solver states are sound, this is done in a separate step.

Expanding a Query. The `expandQuery` strategy expands the selected query constraint into all possible sets of declarations the query could resolve to, while taking shadowing into account. This strategy calls the name resolution algorithm that is also used by normal Statix inference. Normally, Statix only performs name resolution when the query arguments are sufficiently ground—that is, the scope and name of the query are known—but as we are trying to find all declarations, we do not have a name to resolve. We first query for all reachable names without applying the query's shadowing rules, to avoid applying shadowing across *all* names in the program. This gives us all names reachable from the current scope, for each of which we invoke the name resolution again but this time we apply the normal name and shadowing rules to get all reachable declarations with correct shadowing applied. Most query constraints match the result to a singleton set, asserting that exactly one declaration is expected, but it is entirely possible in Statix for a query to match to a larger set. Therefore, this strategy constructs all *sets* of reachable declarations while taking shadowing into account. See section 4.4 for an example.

```
id     :: a → [a]  -- Identity
seq    :: (a → [b]) → (b → [c]) → a → [c] -- Sequential composition
or     :: (a → [b]) → (a → [b]) → a → [b] -- Disjunction
try    :: (a → [a]) → a → [a]  -- Apply strategy if possible
repeat :: (a → [a]) → a → [a]  -- Repeat strategy until failure
fixSet :: Eq a ⇒ (a → [a]) → a → [a]  -- Repeat strategy until nothing changes
limit  :: Int → (a → [b]) → a → [b]  -- Limit sequence resulting from strategy
(.>)    = seq
```

Fig. 8. Strategy combinators

Inference. The `infer` strategy takes a solver state with unsolved constraints and invokes the Statix constraint solver to perform inference and solve as many constraints as possible. The strategy does not fail if the resulting solver state contains error messages. We deal with erroneous solver states explicitly in a separate step, which we will detail next.

Asserting Well-Typedness. The `assertNoErrors` strategy inspects a solver state and returns a singleton list of this state if it contains no errors messages other than those that were present when code completion was invoked (which are passed as a set to the strategy). However, if the solver state contains new errors, the strategy returns an empty list, indicating failure. This ensures that code completion will not generate proposals that are unsound with respect to the static semantics of the language.

Finding an Injection. The projection of the completion variable in the solver state might contain constraint variables for injections, such as `lv` in `LValue2Exp(lv)`. The `findInjection` strategy finds all these constraint variables and returns a list of tuples of these variables and the input solver state. This allows these variables to be expanded by another step in the algorithm.

Projecting a Variable. The syntax-directed semantic rules use *terms* to represent fragments of the AST of a program. The solver state does not directly contain the values of variables but instead has a unifier that associates a partial term with each variable, where holes in the term are represented by other variables that may or may not be present in the unifier. The `projectVar` strategy projects the term for a given variable out of the solver state's unifier, by finding the partial term for a variable and recursively repeating this for each variable in the term until only un-unified variables remain.

Pretty-Printing a Term. The completion proposals should be represented to the user using the syntax of the object language. *Pretty-printing* converts a term into its corresponding syntax using the syntax rules of the language. In the Spoofax language workbench, this is achieved by calling the `pp` pretty-printing Stratego term-rewriting strategy that is generated for each language from its syntax definition. The `prettyPrint` search strategy takes a term and invokes the `pp` term-rewriting strategy to get its string representation.Any constraint variables that were left in the term are pretty-printed as syntactic placeholders in the object language. For example, the term `Add(e, Int("3"))` would be pretty-printed as `$Exp + 3`.

5.4 Combining Strategies

The built-in strategies above provided core operations on solver states. We built the code completion algorithm by combining these built-in strategies using strategy combinators inspired by the Stratego transformation language [Visser et al. 1998]. Figure 8 defines the type signatures of the strategy combinators. We include the definitions of the combinators in appendix E.

5.5 Code Completion Algorithm

Our code completion algorithm composes the built-in strategies using the strategy combinators, to perform a directed search of the space of all possible completion proposals for a given completion variable. The list of proposals must be sound and complete. That is, it must include all reachable declarations (modulo shadowing) and all syntactic constructs up to one level deep (modulo literals and declaration names), as defined in the static semantics specification of the object language. We present the algorithm in fig. 9.

The core of the code completion algorithm is the `complete` strategy, which performs three steps to gather all valid proposals: expand predicates, expand injections, and expand queries. These correspond to the descriptions in section 4.2, section 4.3, and section 4.4.

Expanding Predicates. The first step is to `expandAllPredicates`, which takes a solver state and selects exactly one predicate constraint that contains the completion variable. It removes this constraint and expands it into its possible alternatives through the built-in `expandPredicate` strategy. Unsound solver states are rejected through `infer` and `assertNoErrors`. This process is repeated until all predicate constraints that contain the completion variable have been expanded.

Note that this only expands the completion variable for one level, as the newly added constraints in the expanded solver states describe the expansion of the completion variable, and therefore no longer contain the completion variable itself. This is desirable, since it prevents the algorithm from infinitely recursing to expand, say, `$Exp` into `$Exp + $Exp` into `($Exp + $Exp) + $Exp` and so forth.

Expanding Injections. As the previous step only expands predicate constraints for one level deep, in some solver states the completion variable will have been expanded into the explicit constructor `B2A` of an injection `A = B`. The `expandAllInjections` strategy expands these injections by applying `findInjection` to find the states in which the completion variable has been bound to an injection constructor, and then invokes `complete` recursively to expand the injections into their alternatives. This is repeated, using `fixSet`, until either the strategy fails because there are no injection constructors to expand, or the set of solver states no longer changes. Since `complete` only returns sound solver states, there is no need to check whether the states are sound.

If injections are (indirectly) recursive, the algorithm could end up in an infinite loop. To avoid this, we track the names of the injections that have been expanded so far during this step and avoid expanding them again.

Expanding Queries. Any unsolved query constraints that remain in the solver state will not be ground enough for the normal solver. The `expandAllQueries` strategy selects and expands one of the query constraints that contains the completion variable using `expandQuery`. Ill-typed states are again rejected by applying `infer` and `assertNoErrors`, and this is repeated until there are no more query constraints to expand. The `or` strategy is used to include the original input state in the resulting set of solver states, such that the original state with the placeholder is retained regardless of whether are query constraints.

Presenting to the User. The algorithm starts and finishes at the `getCompletionProposals` strategy, which takes the completion variable, set of pre-existing error messages, and the solver state constructed from the initial analysis of the program. This is a solver state with some unsolved constraints, due to the completion variable. Once the strategy gathers the completion proposals through calling `complete`, it projects the terms of the completion variables out of the unifier and pretty-prints them in the syntax of the object language. These are the completion proposals that will be shown to the user.

```
-- Core completion algorithm: find the completion proposals
complete :: Var → Set Message → SolverState → [SolverState]
complete v ms
  = (expandAllPredicates v ms) .>
    (expandAllInjections v ms) .>
    (expandAllQueries    v ms)

-- Expand predicate constraints that contain the completion variable
expandAllPredicates :: Var → Set Message → SolverState → [SolverState]
expandAllPredicates v ms =
  repeat (
    (limit 1 (selectPredicate (\(c, s) → containsVar [v] c))) .>
    (expandPredicate) .>
    (infer) .>
    (assertNoErrors ms)
  )

-- Expand predicate constraints on injections that contain the completion variable
expandAllInjections :: Var → Set Message → SolverState → [SolverState]
expandAllInjections v ms
  = fixSet (try (
      (findInjection v) .>
      (\(vInj, s) → (complete vInj ms s))
    ))

-- Expand all query constraints that contain the specified variable
expandAllQueries :: Var → Set Message → SolverState → [SolverState]
expandAllQueries v ms
  = or id (
    repeat (
      (limit 1 (selectQuery (\(c, s) → (containsVar [v] c)))) .>
      (expandQuery) .>
      (infer) .>
      (assertNoErrors ms)
    ))

-- Computes the completion proposals and filters, orders, and pretty-prints them
getCompletionProposals :: Var → Set Message → SolverState → [String]
getCompletionProposals v ms
  = (complete v ms) .>
    (projectVar v) .>
    prettyPrint
```

Fig. 9. The code completion strategy

5.6 Strategy Implementation

We implemented all strategies in Java, abstracted boilerplate code for strategies into a runtime, and integrated code completion in the Spoofax language workbench and the Eclipse and IntelliJ editors. Spoofax with its meta-languages is about 2 million lines of Java code, including almost 400 000 lines of code for the Statix language and implementation. Our code completion algorithm uses the existing Statix structures, and calls its solver and name resolution algorithm with no changes except some bug fixes. Table 1 shows the number of lines of Java code (excluding comments and blank lines) we contributed to enable language-parametric static semantic code completion in Spoofax.

To implement search strategies efficiently, there were some considerations we needed to address.

Persistent Solver State. While exploring the solution space we create many slightly different instances of the solver state, its unifier, scope graph, and other data. This can have a significant impact on the performance of the algorithm due to copying lots of data, and associated garbage collector pressure. To mitigate this, we use the efficient persistent (immutable) data structures of Steindorfer [2017], which are also used by Statix internally. This means that it is cheap to fork a solver state to explore alternatives.

Table 1. Code completion lines of Java code

Strategies Runtime	5 921 loc
Built-in Strategies (5.3)	1 333 loc
Strategy Combinators (5.4)	1 816 loc
Code Completion (5.5)	401 loc
Spoofax Integration	2 068 loc
Eclipse UI Integration	133 loc
IntelliJ UI Integration	100 loc
Total	11 772 loc

Lazy Lists. While developing our algorithm, we observed cases in which the strategies performed much more work than necessary due to strict evaluation. For example, if the strategy `foo` returns a hundred solver results, then the strategy `(limit 1 foo)` would return only the first and discard the other 99. To avoid this extra work, we implemented the strategies such that they are not immediately evaluated to return a list, but instead return a pull-based *lazy list*, whose next value is only computed when requested.

6 EVALUATION

While we used the smaller Micro-C♯ language as an illustration language for this paper, we chose to use the Tiger and ChocoPy programming languages for our evaluation, as they include some more interesting type system and name binding constructs and already have extensive test suites.

Tiger is a small Algol-based programming language described in Appel [2002] used to demonstrate both functional and object-oriented compiler concepts, and is used as a standard example language for testing the Spoofax language workbench. Its specification in the Spoofax language workbench consists of 114 syntactic productions and 32 Statix predicates defined with 75 rules in 513 lines of Statix code (excluding generated files for the constructor declarations), covering the entire language. We include the full specification of Tiger in appendix C and appendix D.

The ChocoPy language [Padhye et al. 2019] is a statically-typed subset of Python used to teach programming languages and compiler construction courses. Among other features, it demonstrates inheritance, method overloading, and nested functions. It is specified in Spoofax using 125 syntactic productions and 49 Statix predicates defines with 165 rules in 698 lines of Statix code (again, excluding generated files for the constructor declarations).

Tiger and ChocoPy have extensive test suites consisting of 166 Tiger files[1] and 182 ChocoPy files[2], that test various aspects of the languages. We selected all test files that were not testing syntactic or semantic errors, and ended up with a test suite of 114 Tiger and 129 ChocoPy files. We synthetically enlarged 21 of the Tiger files and 7 of the ChocoPy files by adding and copying additional declarations, and included these in the test sets. We used these 135 Tiger and 136 ChocoPy files to evaluate the code completion algorithm in three ways:

(1) are the proposed completions correct according to the syntax and static semantics of the language (soundness);
(2) is it always possible to select a completion that would reconstruct (the start of) the elided subtree (completeness); and
(3) how long does it take for code completion to provide the proposals (performance).

6.1 Soundness

The proposals generated by the code completion algorithm are sound by construction with respect to the static semantics specification of the object language, as they call upon the Statix constraint solver to verify whether a proposal is acceptable. Unsound proposals (i.e., those that introduce errors that were not initially present in the program) would be found by the Statix solver (through the `infer` strategy), removed from the results (through the `assertNoErrors` strategy), and not be shown to the user.

6.2 Completeness

Our notion of completeness of the algorithm implies that it will propose all syntactic constructs up to one level deep for which a static semantic rule has been defined, and all identifiers that are reachable from the current scope in the scope graph (modulo shadowing), provided that these proposals are sound. Repeated invocation of code completion will allow the user to construct any semantically valid program, modulo declaration names and literals (integers, strings).

To show with high confidence the completeness of the code completion algorithm, we constructed 23 424 Tiger and 37 200 ChocoPy test cases from the selected 135 Tiger files and 136 ChocoPy files. Each test case is a variant of a full program, but with one abstract syntax subtree replaced by a placeholder. This mimics the most common case where the user invokes completion on the only placeholder (or inferred placeholder) of the program. The test asserts that invoking code completion on this placeholder results in a set of completion proposals from which one can reconstruct some of the elided code, showing that we can use repeated invocation of code completion to reconstruct each of the test files.

All tests succeeded, showing both that the static semantics specification of Tiger and ChocoPy covers all the syntax in the test files, and that the algorithm always produced a proposal that would match the expected syntax or reference. In other words, the language-parametric code completion algorithm is complete with respect to the static semantics specification.

6.3 Performance

To measure the performance of the code completion algorithm, we use the same 60 624 constructed test cases, where we measure the time from invoking code completion on the placeholder until it produces its proposals. To avoid local user processes interfering with the tests, we ran them on an otherwise unused server with two 32-core AMD Epyc 7452 processors and 256 GB of memory, but note that both our algorithm and the Statix solver only support single-threaded operation and

[1] https://github.com/MetaBorgCube/metaborg-tiger/tree/master/org.metaborg.lang.tiger.example
[2] https://github.com/cs164berkeley

do not utilize the large amount of memory. We used Java 17 and limited the process to 4 GB of memory and 4 processors to align with the system that a typical user might have available.

In fig. 10 we show a scatter plot of the time, in milliseconds, that each test case took from being invoked until presenting the list of proposals. Tiger tests (up to 2 368 AST nodes) are in blue, whereas ChocoPy tests (up to 8 060 nodes) are in green. The tests are horizontally plotted by the number of nodes in the abstract syntax tree in which completion is being invoked, and vertically by the time they took, in milliseconds.

The performance results are outlined in table 2, split into three performance aspects: the time for the initial analysis of the program with the placeholder inserted (*Initial Analysis*), the time for the code completion search algorithm (*Code Completion*), and the total time that includes both and some minor housekeeping. The total time reflects and affects the user experience, as it is the time from code completion being invoked until it presents a list of proposals.

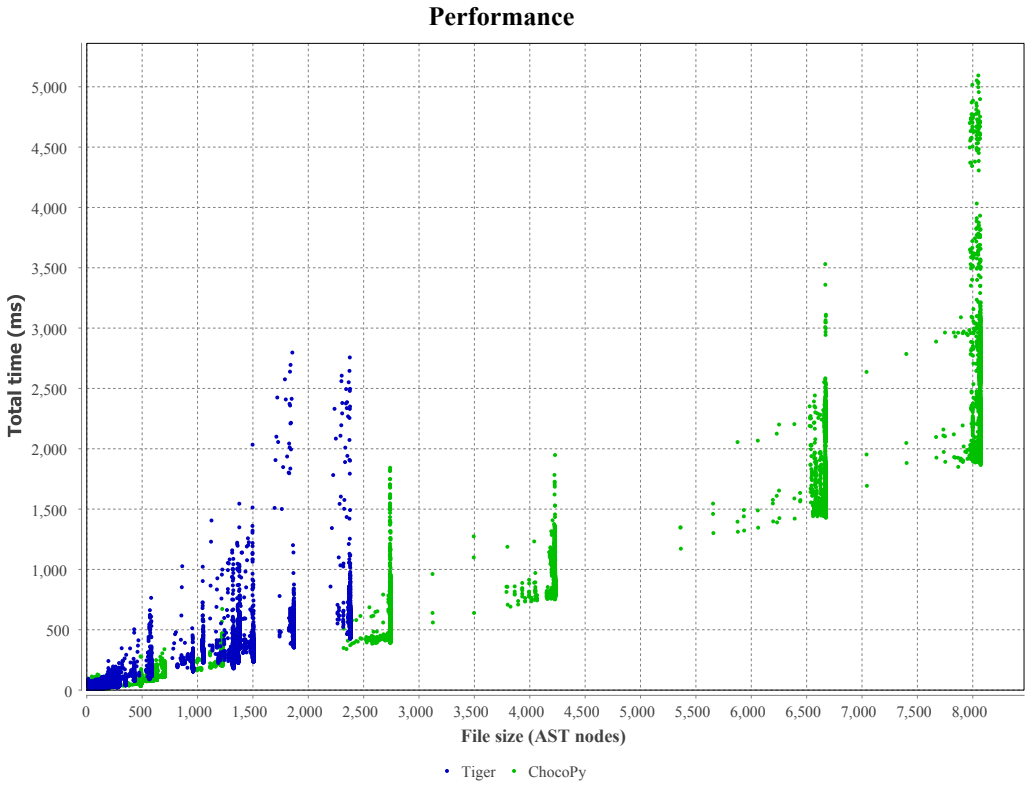


Fig. 10. Code completion algorithm performance as a scatter plot, showing the time spent on each of the 60 624 Tiger and ChocoPy code completion invocations against the number of AST nodes in the test file.

6.4 Analysis

The test results indicate that 70.60% of code completion invocations in our test set can be completed in 1 000 ms or less, from being invoked to presenting the code completion proposals. In 99.17% of cases the code completion algorithm itself completes within 1 000 ms, and the rest is accounted for by the initial analysis. In 114 test cases (0.19%) the tests ran significantly slower, taking more than 4 seconds to conclude for certain large test files with many declarations in scope.

Table 2. Benchmark Results

	Average	Median	95th pctl	99th pctl	Maximum
Tiger (Code Completion)	61 ms	35 ms	180 ms	414 ms	2 719 ms
Tiger (Initial Analysis)	206 ms	206 ms	458 ms	486 ms	836 ms
Tiger (Total)	267 ms	259 ms	595 ms	837 ms	2 808 ms
ChocoPy (Code Completion)	144 ms	18 ms	710 ms	991 ms	3 158 ms
ChocoPy (Initial Analysis)	967 ms	815 ms	2 029 ms	2 077 ms	2 871 ms
ChocoPy (Total)	1 111 ms	919 ms	2 603 ms	2 889 ms	5 104 ms
All (Code Completion)	113 ms	32 ms	675 ms	894 ms	3 158 ms
All (Initial Analysis)	683 ms	371 ms	1 991 ms	2 058 ms	2 871 ms
All (Total)	797 ms	422 ms	2 280 ms	2 785 ms	5 104 ms

The performance of this initial analysis depends on both the complexity of the static semantics specification and the size of the program, and accounts for the empty space at the bottom of the graph in fig. 10. As the test programs for ChocoPy get larger, the initial analysis also takes more time. However, developers are working on making the analysis in Statix faster (and eventually incremental), and this will also bring the total time down.

The performance of the code completion search strategy itself is governed by the number of reachable declarations and the performance of the `expandQuery` strategy, which searches the scope graph for all possible (sets of) declarations a query could resolve to. In the worst case, for n declarations this will try to find all 2^n power sets of declarations, but in practice, most declarations only resolve to a singleton set.

Distinct Checks. A pattern commonly written by language developers in their Statix specification is to check whether a declaration is *distinct* by adding the declaration to the scope graph, followed by an assertion that it is the only declaration in the scope graph:

```
declareType(s, x, T) :-
  s → Type{x} with typeOfDecl T,                  // add declaration to scope graph
  typeOfDecl of Type{x} in s ⟼ [_].               // assert declaration is distinct
```

Performing code completion on the identifier placeholder of a declaration will expand on the query constraint to find existing declarations. However, applying any of these declarations will always fail, as that would add a second declaration to the scope graph, making the query no longer satisfiable. Therefore, code completion rejects all existing declarations, which is sound, but this fruitless search of the scope graph does have a measurable negative performance impact. In practice, this will not be a problem, as users rarely invoke code completion on the name of a declaration.

Permissive Lookups. Similarly, another pattern sometimes used is to allow *permissive lookups* when performing a query. This ensures that a possible error about duplicate declarations is shown on the declaration site rather than the reference site.

```
lookupType(s, x) = T :- {x'}
  typeOfDecl of Type{x} in s ⟼ [(_, (Type{x'}, T)), _].
```

However, this query will cause our code completion algorithm to try all possible sets of declarations that the query could resolve to, even though only the singleton sets are relevant. Again, this has no impact on the soundness of the algorithm but does have a performance impact.

6.5 Threats to Validity

The performance tests were performed on stand-alone Tiger and ChocoPy files of up to 31 766 characters (1 091 lines, 8 060 AST nodes), where 60% are smaller than a 10 000 characters. While

performance seems linear in the size of the program, our algorithm might show different performance characteristics for much larger files or multiple dependent files. The performance of the algorithm is mostly governed by the performance of the Statix solver, which for larger files will have to track more variables in its unifier, and more nodes in the scope graph.

We were unable to test the algorithm on a language with a much larger static semantics specification, but we expect the size of the program to have a bigger impact on performance. A larger specification mostly affects the number of rules that a predicate can be expanded into, whereas a larger program affects the size of the context that the Statix solver has to deal with.

The code completion algorithm is parameterized by the language's specification of static semantics in Statix. While van Antwerpen et al. [2018] and Rouvoet et al. [2020] show that many interesting name binding and type system concepts can be expressed in Statix, it is not clear what language concepts Statix cannot express and therefore to which languages this code completion approach does not apply.

7 RELATED WORK

In this section, we compare our work to related work on code completion.

Code Completion in IDEs. Editors for programming languages in IDEs such Eclipse[3] and IntelliJ[4] provide mature and sophisticated code completion services. However, implementing these editor services requires a considerable implementation effort in addition to the other aspects of the language implementation, an effort that has to be repeated for each IDE. Proposals such as the Language Server Protocol[5] (LSP) aim to reduce the extra effort required to support each editor, but do not offer more than a common interface for the code completion editor service, one that still requires a custom implementation for each language.

The value of our work is to provide a reliable code completion service without much additional work for the language engineer, making the effort of implementing the service reusable between languages. That is particularly interesting for language engineers working on new languages with a small team.

Textual Language Workbenches. Language workbenches aim to support the implementation of rich editors for (domain-specific) programming languages. This usually includes support for implementing code completion, but typically requires language-specific implementation effort by the language engineer.

Xtext supports the generation of IDEs in Eclipse and other platforms [Xtext Team 2021]. Xtext's generates basic support for code completion, including reference proposals, from the grammar and scope providers. However, more sophisticated code completion needs to be implemented manually. Code completion does not take validation (type checking) rules into account. Such checks need to be built into code completion manually.

Rascal [Klint et al. 2009, 2010, 2019] provides TypePal, a framework for name analysis, type checking, and type inference [Rascal 2021]. TypePal provides useful utilities for writing type checkers and a name analysis representation. However, code completion cannot be derived automatically from such type checkers, but needs to be defined as a separate computation that takes the analysis result from the type checker as a parameter.

Spoofax [Kats and Visser 2010] supports sound and complete *syntactic* code completion by representing incompleteness through explicit placeholders and by synthesizing completion proposals

[3] https://www.eclipse.org/
[4] https://www.jetbrains.com/idea/
[5] https://microsoft.github.io/language-server-protocol/

from the grammar [de Souza Amorim et al. 2016]. They use placeholder inference to propose extensions of complete programs or to repair syntactic errors around the caret. We build on their work, restricting the syntactically correct proposals further by applying type checking and through name resolution in order to provide context-sensitive completion proposals. As future work, placeholder inference could be extended to propose (quick) fixes for static semantic errors in programs.

Sasano and Choi [2021] use the *postfix sentential form* of a partial LR parse to determine candidates for code completion in a language-parametric way. However, they explicitly ignore any syntax following the caret that might restrict the possible candidates. And, as the title of the paper suggests, they focus only on syntactic code completion; the types and scoping rules of the program are ignored and identifiers are not proposed. Therefore, their approach might propose completions that introduce semantic errors into the program.

Structure Editors. Structure editors or projectional editors operate on a structured representation via a projection of that representation. Representation of incomplete programs and completing such placeholders by means of code completion is an essential aspect of such editors.

The MPS language workbench is based on projectional editing [Jetbrains 2021]. A language is defined by means of a structure definition, characterizing the abstract structure of programs. An editor is created by defining projections and editing actions on structures. While MPS supports a rich set of projections (including graphical and diagram languages), it does not support automatic derivation of sound and complete synthesis of code completions from a language definition, including its static semantics. Edit actions in MPS are not guaranteed to be sound with respect to the type system rules of the language. That is, type constraints and other well-formedness constraints are not considered and need to be duplicated when synthesizing completion proposals.

Steimann et al. [2017] address this issue in a prototype *robust projectional editor* extending MPS that supports *robust editing*, i.e., *editing that leaves a well-formed program well-formed*. This is realized by complementing the possibly unsound basic edits of MPS by proposing additional edits that repair the model to restore well-formedness. This can not only filter out invalid edits, but may also involve making non-local changes in the program. The RPE is realized by encoding well-formedness as a constraint satisfaction problem and computing edit operations using constraint solving techniques. Because it would be too time-consuming to compute all possible edit scenarios, and too unwieldy for the user to use, the RPE has an interactive solution space explorer that starts with few solutions and expands only upon specific user request. In this work, we focus on the more basic operation of code completions, but provide a predictable editor service. Furthermore, robust editing in the RPE can only be applied to entirely well-formed programs. In contrast, the Statix solver we use can deal with programs with errors; it solves as many constraints as it can and report the failing constraints as errors. By extension, our code completion algorithm can synthesize sound completion proposals for the placeholder, by ignoring errors in the rest of the program.

Hazelnut [Omar et al. 2017] is a structure editor based on a typed lambda calculus with holes, that is designed to study formal models and general principles of (structure) editing. This includes the construction of programs with locally type incorrect terms through non-empty holes, which form a type barrier between an expected type and a provided type. The Hazelnut calculus formalizes edit actions to manipulate programs with (non-empty) holes such that types are preserved. Subsequent work by Omar et al. [2019] and Lubin et al. [2020] explore the execution and program synthesis of programs with holes. These are interesting avenues to explore. While we have not extended our language definitions with explicit non-empty holes, proposals such as `subterms.$LExp` in our running example are in fact similar. While `subterms` is not type correct in the context of the placeholder, a possible 'continuation' could be. It would be useful to generalize such edit contexts.

Reusing Compiler Tools. Our code completion algorithm is reusable across languages in the Spoofax language workbench. There are other approaches to reusable compiler tools or interfaces for them.

Luo et al. [2019] propose *MagpieBridge*, an approach to integrate static analysis in IDEs and editors similar to Language Server Protocol (LSP). They do this by integrating Soot, Doop, and WALA-based analysis through MagpieBridge with LSP, supporting any editor that supports LSP.

Söderberg and Hedin [2011] use *reference attribute grammars* (RAGs) in JastAdd to define semantic editor services, where large parts of existing RAG-based compilers can be reused. Their code completion service returns only 'accesses' (declarations) and no syntax.

Pacak and Erdweg [2019] describe their vision for a semantic editor service by translating the typing rules to Datalog, and use existing Datalog solvers to not only verify the correctness of a program but also to answer 'code completion' queries: given a type, find expressions and variables of that type in scope. However, their translation into Datalog has to deal with infinite domains of types and typing contexts, and is not language-parametric.

Name Capture and Qualified Names. Name capture concerns the introduction of a reference to a different declaration than intended. The *name-fix* algorithm of Erdweg et al. [2014] repairs name capture in code generators by computing a renaming of generated names. This kind of repair is not needed for our code completion, since our algorithm only computes names that are in scope, and thus are capture-free by construction. However, a limitation of our approach is that it will not directly propose names that are shadowed, although the user can still reach them by inserting the qualifier through code completion, and then invoking code completion on the qualified placeholder.

In their work on identifier renaming for Java, Schäfer et al. [2008] introduce fixes for renamings that would lead to capture by qualifying names. For example, introducing the prefix `this.` can deconflict a reference to a class variable and a method parameter. Translating that approach would lead to proposing shadowed names if a deconflicting qualifier is available. It is an interesting future extension of our work to provide language-specific extensions to our language-parametric algorithm that provide such repairs.

Type-Directed Programming. In his work on Oleg, McBride [2000] developed a type theory with holes, placeholders for components of programs, to formalize programs under development. This work was later refined in the dependently typed language Epigram [McBride 2004; McBride and McKinna 2004]. This has inspired the development of type-directed programming in subsequent dependently typed programming languages. Norell [2007] describes interaction points/holes in Agda, which piggyback on how Agda represents meta-variables internally, using a constraint-based approach based on McBride's treatment of meta-variables in Epigram. The language provides an Emacs integration with support for elaborating pattern match cases, summarizing and visualizing local bindings that could be used to fill the hole, and automatic code completion. The `auto` command of Agda performs a type-directed proof search trying to find a completion proposal considering constructors, local variables, and recursive calls, and picks the first candidate that fits [Lindblad and Benke 2004]. However, the `auto` command is neither sound nor complete. Similar support is provided by Idris [Brady 2013] and Haskell.

It is an interesting challenge to apply our code completion algorithm to dependently typed languages. However, a complication of dependently typed programming languages is that the type checker requires normalization (computation) at compile-time. In principle, Statix can be used to specify a normalizer for type expressions. However, we have not defined a dependently typed language in Statix, let alone experimented with code completion for such a language.

Program Synthesis. Semantic code completion is a very limited form of program synthesis, where the user intent is provided by the constraints imposed by the surrounding partial program, and the search is performed over possible solver states. The search strategy performs an enumerative search and checks whether each possible solution satisfies the program's semantic constraint [Gulwani et al. 2017].

While program synthesis systems such as Sketch [Solar-Lezama 2009] can take the dynamic semantics of the code into account, our code completion algorithm only uses the static semantics as specified by the language developer.

Program synthesis also requires the user to specify a specification, query, or constraints to find an appropriate solution, even when using top-down type-directed completion of partial expressions [Perelman et al. 2012]. Static semantic code completion only needs the partial input program and the static semantics specification of the language, without any additional user input.

Then again, contrary to most program synthesis, semantic code completion does not aim to find the best solution or even all possible solutions exhaustively. Instead, it performs a breadth-first search to find the immediately reachable (partial) solutions for the user to select from. This search is directed by the semantic rules of the language, and rule specializations in the language specification can be added to direct the search into certain directions for a more user-friendly experience without affecting the completeness or soundness of the proposals.

Machine Learning. Asaduzzaman et al. [2014] and Raychev et al. [2014] use a large library of examples and histories to determine completion proposals and their ranking, using heuristics and statistical language models respectively. While these approaches can be largely language-independent, they are not sound, complete, or suitable for languages such as DSLs, that do not have a large code sample base to draw from. While we use a simple heuristic to rank proposals (most specific first), it would be useful to involve some sort of statistical model in this ranking, even in the face of small code bases or new languages.

Evaluation. Hellendoorn et al. [2019] conclude that many synthetic benchmarks do not match real-world use of code completion. For example, real-world completion proposals were not clearly focused on any particular group of tokens. Also, they indicate that 'time saved', the time difference between the user finding the code completion proposal and typing the syntax or identifier manually, is often not a good measure, as the time between invoking code completion and accepting a proposal varies wildly. Their analysis is mostly geared toward code completion engines that use heuristics and statistics for their proposals. In this work, we have focused on a *correct-by-construction* derivation of code completion from language definitions. This is an important first step, but improving the presentation of code completion, in particular, the ranking as noted above, based on user evaluation will be an important future work.

8 FUTURE WORK

In this section, we discuss some opportunities for improvements to the code completion algorithms usability or performance that do not impact its soundness or completeness.

Deterministic Expansion. Without impacting the soundness or completeness, the code completion algorithm should perform *deterministic expansion* when there is only one possible expansion of a placeholder in a proposal. For example, for inserting a method call `foo(Int, Int)` that has two parameters, code completion currently suggests `foo($Exps)`, a method call with an argument list of indeterminate length. Deterministic expansion on `$Exps` would find that the only possible expansion is `$Exp, $Exp`, a list of two argument placeholders, and therefore code completion should immediately suggest `foo($Exp, $Exp)` instead.

We have an implementation of deterministic expansion, but for each placeholder in each proposal, it wasted a lot of time searching for a single expansion when there was none, and this had a significant negative impact on the overall performance. Therefore, we dropped it from this paper, but one proposal is to amend the static semantics specification with annotations that indicate when applying deterministic expansion makes sense.

Heuristics for Expansion. We currently expand a predicate only once, unless it is a plain injection of the form `Stmt = Exp`. To manually force the algorithm to expand further, *rule specialization* (section 4.5) can be used. However, we would like to identify a reliable heuristic that can automatically determine whether to continue expanding a proposal and when to stop. Similarly, when the proposal includes a placeholder that, when expanded several times more, could expand to a name, it would be better to present the user with this name than a placeholder.

Ranking Proposals. Ranking proposals was out of the scope for this paper, and therefore we ranked proposals by their specificity. However, we could employ statistical and/or example-based code completion techniques [Asaduzzaman et al. 2014, 2016; Bruch et al. 2009; Raychev et al. 2014], although their use might be hampered by the limited codebases available for certain DSLs.

Parallelization. While our implementation of the code completion algorithm is single-threaded, one possible optimization is to run the search strategies in parallel. This would be made possible by the search states being immutable, and the search strategy not sharing mutable state. A related improvement could be to show the code completion proposals to the user as they are being discovered by the algorithm, while the algorithm continues exploring alternatives in the background.

Search Strategies. The search strategies are not only useful for code completion, but potentially also in other language-parametric editor services that depend on the program's static analysis. In future work, we would like to investigate code generation, code navigation, structural search and replace, linting, and various refactorings. The search strategies could also be used to provide intelligent suggestions, such as relating parameter names in completion proposals to similarly named (and type compatible) variables in scope. Debuggers that display relevant information in the scope around a breakpoint could also benefit from search strategies to find, for example, the values of all variables in scope, or the members of a particular object.

9 CONCLUSION

We have presented a language-parametric code completion editor service that is sound and complete. It uses search strategies to find all type-safe proposals, including syntax, identifiers, and function calls. It does this language-parametrically for languages defined in the Spoofax language workbench, relying only on the Statix constraint solver and the existing syntax-directed static semantics specification of the object language. Minimal effort is needed to produce a complete list of sound proposals, but small changes to the static semantics specification can be used to direct code completion to produce more useful proposals. Through three steps of expanding predicates, injections, and queries, it attempts to find a set of useful proposals, one of which the user can select and insert at the caret location in the source code document. The algorithm rejects unsound proposals, and we demonstrated its completeness with respect to the static semantics specification by having the algorithm reconstruct test files from scratch using repeated invocations of code completion. We have applied our algorithm to Tiger and ChocoPy for the tests in the evaluation, and Micro-C$^\sharp$ for the examples in this paper.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback on previous revisions of this paper.

REFERENCES

Andrew W. Appel. 2002. *Modern Compiler Implementation in Java, 2nd edition.* Cambridge University Press.

Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Daqing Hou. 2014. Context-Sensitive Code Completion Tool for Better API Usability. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE, 621–624. https://doi.org/10.1109/ICSME.2014.110

Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Daqing Hou. 2016. A Simple, Efficient, Context-sensitive Approach for Code Completion. *Journal of Software Maintenance* 28, 7 (2016), 512–541. https://doi.org/10.1002/smr.1791

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, Hans van Vliet and ValÃ©rie Issarny (Eds.). ACM, 213–222. https://doi.org/10.1145/1595696.1595728

Luis Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. 2016. Principled syntactic code completion using placeholders. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, Tijs van der Storm, Emilie Balland, and Dániel Varró (Eds.). ACM, 163–175. https://doi.org/10.1145/2997364.2997374

Luis Eduardo de Souza Amorim and Eelco Visser. 2020. Multi-purpose Syntax Definition with SDF3. In *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12310)*, Frank S. de Boer and Antonio Cerone (Eds.). Springer, 1–23. https://doi.org/10.1007/978-3-030-58768-0_1

Sebastian Erdweg, Tijs van der Storm, and Yi Dai. 2014. Capture-Avoiding and Hygienic Program Transformations. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*, Richard Jones (Ed.). Springer, 489–514. https://doi.org/10.1007/978-3-662-44202-9_20

Martin Fowler. 2005. Language Workbenches: The Killer-App for Domain Specific Languages? http://www.martinfowler.com/articles/languageWorkbench.html

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. 2019. When code completion fails: a case study on real-world completions. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Gunter Mussbacher, Joanne M. Atlee, and Tevfik Bultan (Eds.). IEEE / ACM, 960–970. https://dl.acm.org/citation.cfm?id=3339625

Jetbrains. 2021. *Jetbrains MPS.* http://jetbrains.com/mps/

Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, Reno/-Tahoe, Nevada, 444–463. https://doi.org/10.1145/1869459.1869497

Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*. IEEE Computer Society, 168–177. https://doi.org/10.1109/SCAM.2009.28

Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2010. EASY Meta-Programming with Rascal. Leveraging the Extract-Analyze-SYnthesize Paradigm for Meta-Programming. In *Proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'09) (LNCS)*. Springer. to appear.

Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2019. Rascal, 10 Years Later. In *19th International Working Conference on Source Code Analysis and Manipulation, SCAM 2019, Cleveland, OH, USA, September 30 - October 1, 2019*. IEEE, 139. https://doi.org/10.1109/SCAM.2019.00023

Fredrik Lindblad and Marcin Benke. 2004. A Tool for Automated Theorem Proving in Agda. In *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3839)*, Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner (Eds.). Springer, 154–169. https://doi.org/10.1007/11617990_10

Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program sketching with live bidirectional evaluation. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020). https://doi.org/10.1145/3408991

Linghui Luo, Julian Dolby, and Eric Bodden. 2019. MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper). In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July*

15-19, 2019, London, United Kingdom (LIPIcs, Vol. 134), Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2019.21

Conor McBride. 2000. *Dependently typed functional programs and their proofs*. Ph. D. Dissertation. University of Edinburgh, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.561753 British Library, EThOS.

Conor McBride. 2004. Epigram: Practical Programming with Dependent Types. In *Advanced Functional Programming, 5th International School, AFP 2004, Tartu, Estonia, August 14-21, 2004, Revised Lectures (Lecture Notes in Computer Science, Vol. 3622)*, Varmo Vene and Tarmo Uustalu (Eds.). Springer, 130–170. https://doi.org/10.1007/11546382_3

Conor McBride and James McKinna. 2004. The view from the left. *Journal of Functional Programming* 14, 1 (2004), 69–111. https://doi.org/10.1017/S0956796803004829

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph. D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages* 3 (2019). https://dl.acm.org/citation.cfm?id=3290327

Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 86–99. http://dl.acm.org/citation.cfm?id=3009900

André Pacak and Sebastian Erdweg. 2019. Generating incremental type services. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, Oscar Nierstrasz, Jeff Gray, and Bruno C. D. S. Oliveira (Eds.). ACM, 197–201. https://doi.org/10.1145/3357766.3359534

Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. 2019. ChocoPy: A Programming Language for Compilers Courses. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E (SPLASH-E 2019)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3358711.3361627

Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. 2022. *Artifact for "Language-Parametric Static Semantic Code Completion"*. https://doi.org/10.5281/zenodo.6367565

Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, and Eelco Visser. 2019. Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper). In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs, Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2019.26

Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. 2012. Type-directed completion of partial expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 275–286. https://doi.org/10.1145/2254064.2254098

Rascal. 2021. *TypePal: Name and Type Analysis Made Easy*. http://docs.rascal-mpl.org/unstable/TypePal

Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 44. https://doi.org/10.1145/2594291.2594321

Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). https://doi.org/10.1145/3428248

Isao Sasano and Kwanghoon Choi. 2021. A text-based syntax completion method using LR parsing. In *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2021, Virtual Event, Denmark, January 18-19, 2021*, Sam Lindley and Torben Æ. Mogensen (Eds.). ACM, 32–43. https://doi.org/10.1145/3441296.3441395

Max Schäfer, Torbjörn Ekman, and Oege de Moor. 2008. Sound and extensible renaming for Java. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, Gail E. Harris (Ed.). ACM, 277–294. https://doi.org/10.1145/1449764.1449787

Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5904)*, Zhenjiang Hu (Ed.). Springer, 4–13. https://doi.org/10.1007/978-3-642-10672-9_3

Friedrich Steimann, Marcus Frenkel, and Markus Voelter. 2017. Robust projectional editing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, Benoît Combemale, Marjan Mernik, and Bernhard Rumpe (Eds.). ACM, 79–90. https://doi.org/10.1145/3136014.3136034

Michael Steindorfer. 2017. *Efficient Immutable Collections*. Ph. D. Dissertation. Universiteit van Amsterdam. Advisor(s) Paul Klint and Jurgen J. Vinju.

Emma Söderberg and Görel Hedin. 2011. Building semantic editors using JastAdd: tool demonstration. In *Language Descriptions, Tools and Applications, LDTA 2011, Saarbrücken, Germany, March 26-27, 2011. Proceeding*, Claus Brabrand and Eric Van Wyk (Eds.). ACM, 11. https://doi.org/10.1145/1988783.1988794

Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2016. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 49–60. https://doi.org/10.1145/2847538.2847543

Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as types. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018). https://doi.org/10.1145/3276484

Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, Baltimore, Maryland, United States, 13–26. https://doi.org/10.1145/289423.289425

Xtext Team. 2021. *Xtext*. https://www.eclipse.org/Xtext/