On the Soundness of Auto-completion Services for Dynamically Typed Languages

Damian Frölich
dfrolich@acm.org
Informatics Institute, University of Amsterdam
Amsterdam, the Netherlands

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org
Informatics Institute, University of Amsterdam
Amsterdam, the Netherlands

Abstract

Giving auto-completion candidates for dynamically typed languages requires complex analysis of the source code, especially when the goal is to ensure that the completion candidates do not introduce bugs. In this paper, we introduce an approach that builds upon abstract interpretation and the scope graph framework to obtain an over-approximation of the name binding seen at run-time. The over-approximation contains enough information to implement auto-completion services such that the given suggestions do not introduce name binding errors. To demonstrate our approach, we compare the suggestions given by our approach with the state of the art completion services on a subset of the Python programming language.

CCS Concepts: • **Software and its engineering** → *General programming languages*; *Development frameworks and environments*; *Software notations and tools*.

Keywords: scope graphs, name resolution, editor services, abstract interpretation, auto-complete

ACM Reference Format:
Damian Frölich and L. Thomas van Binsbergen. 2024. On the Soundness of Auto-completion Services for Dynamically Typed Languages. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '24), October 21–22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3689484.3690734

1 Introduction

Today's developers have access to an abundance of programming tools that increase their productivity by assisting the programming activity itself or the management of (large) code bases through, for example, version control or delivering project insights. Integrated Development Environments (IDEs) integrate such tools and serve as the main interface for programmers during programming-related activities. Today, most IDEs are decoupled from language implementations or programming analysis tools such that a single IDE can be used for multiple languages and multiple IDEs can be used to work on the same code base (e.g., chosen based on programmer-preference). The Language Server Protocol[1] (LSP) plays a crucial role in this decoupling, interfacing between the environment and so-called *language services* such as 'go-to definition' and auto-complete. As such, the client-server architecture of the LSP addresses the $n \times m$ problem of supporting n languages in m IDEs, reducing the (overall) engineering efforts for both IDEs and languages. Moreover, by modularizing the server back-end into distinct services, tools can be developed that could specialize in particular program analyses or programmer feedback.

Language services typically analyze source code. The type of service determines the density of information to be extracted. For example, an abstract syntax tree provides enough information to implement semantic highlighting (an extension of syntax highlighting). Other services may require additional information, such as the types of variables or the declaration to which a reference resolves. For statically typed languages, the structure enforced by type systems aid the implementation of such services. Inherent to statically typed languages is a static type-checker. In contrast, a significantly more complex analysis is needed to perform (static) type-checking for dynamically typed languages as the type of a variable may be determined by program input [17]. More generally, editor services for dynamically typed languages may require complex analyses which may not always capture all cases, or make a trade-off between soundness and completeness.

The state of the art in auto-complete services for Python exhibit this phenomenon, with tools generally choosing completeness over soundness. For example, an auto-complete service providing auto-completion candidates for the program in Figure 1 at the position indicated by the question mark, needs to take into account that the `obj` variable can point to an object of class `A` or to an object of class `B`, depending on user-provided input. As a result, the completion

[1] https://microsoft.github.io/language-server-protocol/

```
class A:                obj = A()
  x = 5                 obj.x = input()
  z = 10
                        if obj.x:
class B:                    obj = B()
  x = 0
  y = 5                 print(obj.?)
```

Figure 1. Python example on which completions from auto-completion services can introduce erroneous execution paths. The ? indicates the position of the cursor, i.e. the source location from which an auto-complete request is performed.

candidates y and z introduce an erroneous execution path since they are not present in both A and B. The x field is present in both classes and is therefore a sound completion candidate, since in both run-time paths that field is present on the object assigned to the obj variable. However, Pylance[2] and Jedi[3], two prominent editor service implementations for Python, give all three fields as completion candidates. This result is complete as it contains all candidates that create a valid program flow but is not sound as some candidates introduce erroneous program flows (with respect to name resolution).

With small programs, the programmer may be able to detect unsound candidates and handle them appropriately. However, with more complex programs, maintaining the oversight required to do so becomes challenging. Furthermore, the execution of some erroneous execution paths may be rare, complicating bug discovery. Obtaining such erroneous completion candidates is mentioned as one of the most concerning issues by practitioners when using auto-completion tools [33].

In this paper, we introduce an approach that sets a first step towards the creation of sound editor services for dynamically typed languages. The approach leverages abstract interpretation and scope graphs to build a model for resolving names in programs. Using this model, we can construct auto-completion services that are sound with respect to the name binding of a program across the different execution paths the program embeds. Concretely, in this paper we make the following contributions:

- an approach based on abstract interpretation and scope graphs for the implementation of sound completion services;
- an implementation of said approach in Haskell;
- a test set of Python programs with key name binding challenges that result in unsound completion candidates with the state of the art editor services.

[2]https://github.com/microsoft/pylance-release
[3]https://jedi.readthedocs.io/en/latest/

The paper is structured as follows. In Section 2 we give the necessary background. In Section 3 we discuss the difficulties of applying scope graphs to dynamically typed languages, and present our extension to the scope graph framework. We follow this up by obtaining scope graphs from the run-time heap in Section 4, and via abstract interpretation in Section 5. In Section 6 we introduce an implementation of our approach, and demonstrate it in Section 7 via a comparison with the state of the art editor services for Python. We discuss the results from our experiments and our approach in Section 8. We finalize with related work in Section 9, and our conclusion in Section 10.

2 Background

Our approach utilizes abstract interpretation to obtain a sound over-approximation of the name binding seen at run-time. The name binding is captured using scope graphs. These two components combined form the basis for an auto-completion service implementation.

2.1 Abstract Interpretation

Abstract interpretation [2] provides a unified framework for sound static analysis by over-approximation of the dynamic semantics of a programming language. With abstract interpretation, the concrete domain is approximated by an abstract domain. The abstract domain has less computational needs. However, since it is an over-approximation, some information is lost, affecting completeness. The concrete and abstract domain are related via a Galois connection, with which values from two different partially ordered sets can be related. A Galois connection on two partially ordered sets $(C, \leq_C)$ and $(A, \leq_A)$ is given by two monotone functions $\alpha : C \longrightarrow A$ and $\gamma : A \longrightarrow C$, such that $\alpha(c) \leq_a a \iff c \leq_c \gamma(a)$ holds for all $c \in C$ and $a \in A$.

A common example to illustrate abstract domains is the sign abstraction of integers. In this example, the concrete domain is the set of integers ($\mathbb{Z}$) and the abstract domain is the set of signs: $P = \{\bot, <0, Z, >0, \top\}$. The abstraction function maps sets of integers to a (shared) sign.

$$\alpha(\emptyset) = \bot \qquad \alpha(\{0\}) = Z$$
$$\alpha(\{i \mid i < 0\}) = <0 \qquad \alpha(\{i \mid i > 0\}) = >0$$
$$\alpha(\mathbb{Z}) = \top$$

The concretization function can be mechanically derived by following the requirements of a Galois connection. The abstract semantics mirrors the concrete semantics. Operations on integers are thus mirrored by operations on signs, for example $Z + {>0} = {>0}$ and ${<0} + {>0} = \top$. In the second case, we lose information. The result can be positive, negative, or zero, thus the operation yields $\top$ (representing the set of all integers). Furthermore, these operations are strict on $\bot$ elements, e.g., $\bot + Z = \bot$ (with $\bot$ representing divergent computations). Using our abstraction, we can determine the sign of integer expressions. The obtained information can be used, for example, by a compiler to optimize conditionals that are always true or false.

2.1.1 Abstract Interpretation and Natural Semantics.

Schmidt [25, 27] applied the framework of abstract interpretation to languages with an operational semantics in the small-step style (of Plotkin [22]) or big-step style (of Kahn [9]). In the approach, concrete and abstract computations are related by safety relations on values and trees. Every computation represented as a concrete tree needs to be mirrored by an abstract tree and the safety relation on trees needs to be preserved. When a safety relation ($safe$) is both U-closed ($c \ safe \ a \wedge a \sqsubseteq a' \implies c \ safe \ a'$) and G-closed ($c \ safe \ \sqcap \{a' \mid c \ safe \ a'\}$), a Galois connection is obtained for free.

2.2 Scope Graphs

Scope graphs have been introduced by Neron et al. [18] to capture name binding in a language-independent manner. In this paper we adopt the formal definition of scope graphs given in Figure 2. The vertices of a scope graph represent scopes, references, or declarations, and are assumed to be unique such that each declaration and reference belongs to one scope. References and declarations are indexed to distinguish occurrences with the same name at different source locations. When the index is clear from the context, we omit it. Edges denote the relation between scoping elements. A reference has an edge to the scope in which it occurs ($x_i^R \longrightarrow s$). Scopes have edges towards the declarations made within that scope ($s \longrightarrow x_i^D$). Scopes can also have labeled edges towards other scopes ($s \longrightarrow_l s'$). This can be used to model, for example, lexically nested scopes. Finally, a declaration can have an edge towards a scope when it introduces this scope ($x_i^D \longrightarrow s$). For example, the scope of an object, containing the fields and methods of that object. Object access (x.y) is then translated into resolving the variable x and determining whether the resulting declaration has an outgoing scope s. The variable y is then resolved starting from scope s. In this paper, as well as in [18], labels $\mathbf{P}$ (parent) and $\mathbf{I}$ (import) are used. However, the usage of imports in this paper is more rudimentary than that of [18]. We assume every scope has at most one parent (i.e., one outgoing edge labeled $\mathbf{P}$).

Cycles are permitted in scope graphs. To resolve a reference to a declaration, a resolution calculus is defined that describes correct resolution paths in a scope graph, given by the $\longmapsto$ relation in Figure 2. The resolution calculus keeps track of a set of seen scopes and only visits a scope once. A valid path in a scope graph is a sequence of edges moving from the current scope to scope s via the edge with label l, denoted by $\mathbf{E}(l, s)$; and declarations reachable from the current scope, denoted by $\mathbf{D}(x_i^D)$. A resolution from a reference in scope s_1 to a declaration y in parent scope s_2 is described by the following path viewed from scope s_1: $\mathbf{E}(\mathbf{P}, s_2) \cdot \mathbf{D}(y_i^D)$. Using the resolution calculus and a language-specific ordering on paths, we can define the visible declarations in a scope by finding all shortest resolution paths reachable from the scope, which is defined in Figure 2 by the $\rightharpoonup$ arrow, i.e. $s \rightharpoonup (s', x_j^D)$ states that from scope s the declaration x_j^D in scope s' is visible (not shadowed).

A reference can resolve to multiple declarations by finding multiple 'resolution paths'. Language-specific path orderings and well-formedness predicates are used to determine the desired resolution. In the formal model this is reflected in the *(ResE)* rule and the *(Vis)* rule. The ordering selects the 'nearest' declaration site. Different name binding policies can be modeled with the well-formedness predicate [29]. The chosen policies may still not select a unique resolution, for example, when the program is invalid. The scope graph framework leaves to the user to determine whether this is (un)desirable based on the context in which the framework is used.

Prior work has identified a correspondence between static name binding in scope graphs and heap-allocated frames [23]. In essence, a scope graph functions as a blueprint for the heap at run-time. The correspondence brings several benefits, such as uniform type soundness proofs and sound garbage collection. In this work, we utilize this correspondence in the other direction: we use the heap and frames approach as a blueprint for building scope graphs.

3 Scope Graphs for Dynamic Languages

Dynamic languages may be dynamic in several regards, e.g., name resolution may be dynamic, the types of variables may be established dynamically, or programs may be extended dynamically. Similarly, at least three types of correctness can be identified: name binding correctness, type correctness, and syntactic correctness. In this paper, we are primarily interested in the first type of correctness and consider an auto-completion candidate to be sound when its inclusion yields no name resolution errors at run-time. In this section, we describe how we annotate and build scope graphs to model dynamic name resolution, a first step in our approach towards sound (auto-completion) services for dynamically typed languages.

Scope graphs have been used to implement sound auto-complete services for statically typed languages [20]. We observed two main problems when applying scope graphs to dynamically typed languages, which we shall demonstrate by comparing the scope graph of a small statically typed (functional) program and the scope graph of a dynamically typed program (following the name binding semantics of Python). Figure 3 contains the example programs and the corresponding scope graphs. In both programs, some dynamic input is assigned to the variable x and is used for branching into one of two conditional branches, which both introduce a binding. (The subsequent code in the branches

Scope graph

$$s \in \mathit{ScopeId}$$

$$v \in \mathit{Vertex} ::= s \mid x_i^D \mid x_i^R$$

$$e \in \mathit{Edge} ::= s \rightarrow_l s \mid s \rightarrow x_i^D \mid x_i^R \rightarrow s \mid x_i^D \rightarrow s$$

$$l \in \mathit{Label} ::= \mathbf{P} \mid \mathbf{I}$$

$$G \in \mathit{ScopeGraph} ::= \mathcal{P}(\mathit{Vertex}) \times \mathcal{P}(\mathit{Edge})$$

Projection functions

$$K(s) = \{l \mapsto \{s' \mid s \rightarrow_l s'\}\}$$

$$D(s) = \{x_i^D \mid s \rightarrow x_i^D\}$$

$$R(s) = \{x_i^R \mid x_i^R \rightarrow s\}$$

Resolution paths

$$p \in \mathit{Path} ::= \mathbf{D}(x_i^D) \mid \mathbf{E}(l, s) \cdot p$$

Paths

$$\frac{\vdash_G s \rightarrow x_i^D}{\vdash_G \mathbf{D}(x_i^D) : s \longmapsto (s, x_i^D)} \qquad (\textsc{ResD})$$

$$\frac{\begin{array}{c} s' \notin S \qquad S \vdash_G s \rightarrow_l s' \qquad \{s'\} \cup S \vdash_G p : s' \longmapsto (s'', x_i^D) \\ WF(\mathbf{E}(l, s) \cdot p) \end{array}}{S \vdash_G \mathbf{E}(l, s) \cdot p : s \longmapsto (s'', x_i^D)} \qquad (\textsc{ResE})$$

$$\frac{\vdash_G x_i^R \rightarrow s \qquad \{s\} \vdash_G p : s \rightharpoonup (s', x_i^D)}{\vdash_G p : x_i^R \longmapsto (s', x_i^D)} \qquad (\textsc{ResR})$$

Visible declarations

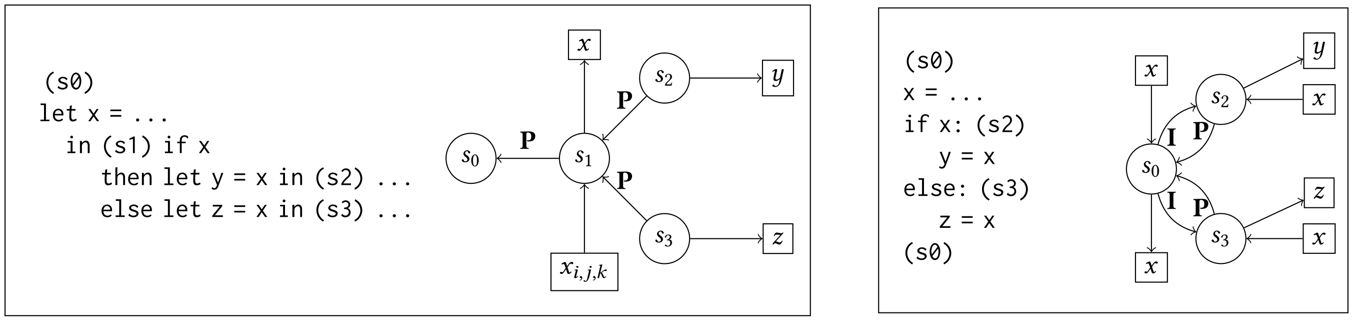
$$\frac{\begin{array}{c} \{s\} \vdash_G p : s \longmapsto (s', x_i^D) \\ \forall j, p', s''(\{s\} \vdash_G p' : s \longmapsto (s'', x_j^D) \implies p' \nless p) \end{array}}{\vdash_G p : s \rightharpoonup (s', x_i^D)} \qquad (\textsc{Vis})$$

Figure 2. Formal definition of scope graphs with resolution calculus, and parameterized by a well-formedness predicate over paths and an ordering on paths. Based on earlier definitions [23, 30]
.

is irrelevant to our example.) Both programs have a 'global' scope labeled s_0 (in the graph and program text) and evaluate the dynamic input in this scope before it is assigned to x. The assignment to x creates a new scope (s_1) in case of the statically typed program. For the dynamically typed program, it adds a declaration to the global scope. Based on the value of the x variable, one of the two bodies is executed. In both bodies, an assignment is performed. In case of the statically typed program, the right sides of the assignments are evaluated in scope s_1, and for both declarations a new scope is created s_2 and s_3, respectively. For the dynamically typed program, a new scope is created for the `if` body (s_2) and for the `else` body (s_3), in which the respective bodies are executed. Both scopes have a parent edge to the global scope. The global scope also has an import edge to both scopes. The import edge is required because after the `if-else` we are back in the global scope (s_0), but the declarations made in the body are still reachable, which is modeled using the import edges. In our statically typed program, this is not the case. From a resolution perspective, we could have given a variety of different scope graphs for the dynamically typed program that gives the same resolution results. For example, one scope containing all declarations. Why we have opted to display the scope graph for the dynamically typed program as having multiple scopes, becomes apparent shortly.

The first problem is that a dynamically typed language puts less restrictions a-priori on a program, which makes the resulting scope graph an over-approximation of the name binding seen at run-time. As a result, reasoning with the resulting scope graph requires care. In our example, this is illustrated by the fact that in both programs, the declarations y and z are never in scope at the same time. This is captured in the scope graph obtained from the statically typed program. However, the scope graph for the dynamically typed language does model that both declarations are in scope: we can take an import edge from s_0 to both s_2 and s_3 and obtain the respective declarations. This is incorrect. The dynamically typed program actually has two scope graphs. One in which there is an import edge towards scope s_2, and thus y is declared; and one in which there is an import edge towards scope s_3, and thus z is declared.

The second problem is that the scope graph for the dynamically typed program only describes the name binding at a specific program point. As a result, the obtained scope graph cannot be used to reason about every program point. In our example this is illustrated if we use the obtained scope graphs and query the available declarations in the conditional of the `if-else`. In the statically typed scope graph, the scope in which the conditional is executed is s_1, so we start our resolution from that scope, and obtain only x as a declaration. For the dynamically typed language, the condition expression is executed in scope s_0, and from that scope we obtain the declarations x, y, z, where y and z are obtained via the import edges. However, at that point, only x is in scope. Our scope graph for the dynamically typed language is thus only valid at the end of our program, but not at intermediate stages.



```
(s0)
let x = ...
  in (s1) if x
     then let y = x in (s2) ...
     else let z = x in (s3) ...
```

```
(s0)
x = ...
if x: (s2)
   y = x
else: (s3)
   z = x
(s0)
```

Figure 3. A similar program written in a statically typed (functional) language and a dynamically typed language with Python like scoping rules, with their corresponding scope graphs. Scopes changes in the programs are indicated with parentheses. i, j, k denote source locations for the different references to x in scope s_1.

3.1 Annotated Scope Graphs

To solve the first problem, we extend the scope graph framework with annotated vertices to denote uncertainty.

As observed, a scope graph for a dynamically typed language describes an over-approximation of the scope graphs seen at run-time. In our example, the obtained over-approximation only describes which variables *might* be in scope. However, it is unknown which variables are *definitely* in scope. Reasoning with our example scope graph results in unsound completion candidates. To overcome this, we extend the scope graph definition with annotations on declarations and references. We define the set of annotations as $An = \{N, D, M\}$, representing not present, definitely present, and maybe present, respectively. The annotation set forms a join-semilattice by the following partial order $N \leq_{An} M$ and $D \leq_{An} M$. The vertices of the scope graph model are updated with an annotation component: $v \in Vertex ::= s \mid (x_i^D, a) \mid (x_i^R, a)$ where $a \in An$. We also add an annotation to labels on edges, which models the uncertainty of edges between scopes. We do not annotate scopes, because we operate from a scope perspective: the correct annotation for a scope differs depending on the scope from which we observe the annotated scope.

Using the annotated scope graph model, we define a partial order on scope graphs as follows, where D_v projects the annotated vertices with annotation D, and D_e projects the labeled edges with annotation D.

$$
\begin{aligned}
&(V_1, E_1) \leq_s (V_2, E_2) \textit{ iff} \\
&\qquad \forall v \in V_1.\ \exists v' \in V_2.\ v \leq_v v' \\
&\qquad D_v(V_2) \subseteq D_v(V_1) \\
&\qquad \forall e \in E_1.\ \exists e' \in E_2.\ e \leq_e e' \\
&\qquad D_e(E_2) \subseteq D_e(E_1)
\end{aligned}
$$

The order on vertices ($\leq_v$) and edges ($\leq_e$) is defined by equality on vertices and labels, and by comparison of the annotations using the $\leq_{An}$ relation. A small excerpt of the full definition is given by the following two cases: $s \leq_v s'$ *iff* $s = s'$ and $(v, a) \leq_v (v', b)$ *iff* $v = v \wedge a \leq_{An} b$. We lift this partial order to a lattice ($\sqcup_s$) by adding a bottom and top element. The partial order on scope graphs captures the idea of over-approximation as introduced in the previous section. The second and last constraints ensure that the definitely present vertices and edges in the second scope graph are also present in the first graph. This ensures that conclusions made over the definitely present vertices and edges in the 'abstract' scope graph also hold in the concrete scope graph.

Figure 4 displays the two scope graphs observed in our dynamically typed example program and the annotated scope graph that over-approximates both. In our annotated scope graph, all the declaration are annotated as definitely present. But, the two import labels are annotated as maybe present. Performing a query from s_0 will not follow the maybe present edges, and thus will not obtain the declarations y and z, only the declaration x. When performing a query from another scope, for example s_2, we will collect both y and x as definitely present declarations. From the perspective of scope s_2, this indeed holds.

3.2 Context-Dependent Name Resolution

To solve the second problem, we introduce context-dependent name resolution, in which resolutions are performed in the context of an index. We use source locations as the index. The location captures the program location from where the resolution is performed, and affects the resolution results by only accepting declarations that have been declared before the program location from where the resolution occurs. In our example, this means that when we resolve from the program location `if ?`, we only find `x` to be in scope, since both `y` and `z` are defined after the `if ?` location. To achieve this, we extend scopes with a location component, and update the path rules from the resolution calculus to propagate a location that affects the available paths. The updated rules are displayed in Figure 5. Both the *ResD* and *ResE* rules now contain a comparison on indices, which results in the filtering of several paths that are present without this condition.

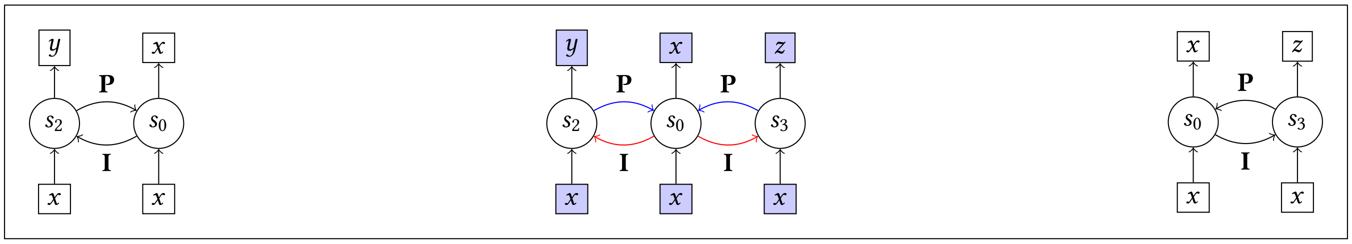


Figure 4. The two scope graphs seen at run-time for the Python example (left and right), and the combined scope graph with annotations (middle). For both edges and vertices, a blueish color represents the D annotation and a red color the M annotation.

The *Vis* rules also needs to be adapted, but only by including the location in the context.

3.3 Multi-Stage Resolution

More complex naming structures require multiple resolution steps. As shown earlier, an object access `x.y` first requires the resolution of the variable `x` to determine its type and associated scope, which can then be used to resolve `y`. Determining the type in a dynamically typed language means to determine its value(s). Consequently, name resolution for dynamically typed languages depends on the values of variables. To be able to support more complex queries, we combine our previous extensions to the scope graph model with abstract interpretation.

4 Obtaining Scope Graphs via Heaps

To obtain an over-approximating scope graph while also having access to the types of values assigned to variables, we utilize abstract interpretation. In our abstract domain, we utilize the 1-to-1 correspondence between scopes and frames to obtain scope graphs from heaps. Our abstract interpretation results in an over-approximation of the abstract heaps seen at all program points. This provides us with a mapping from variables to values per program point. The abstract heaps are then translated into scope graphs and joined into one over-approximating scope graph.

4.1 Updated Heaps and Frames

The original definition of heap and frames uses the statically obtained scope graph for resolution. However, we do not have such a scope graph and need to modify the original definition. Our modification is given in Figure 6. A frame is thus a 4-tuple containing a scope id; a set of references made within the scope of the frame; a mapping from labels to frame ids, modeling connections between frames; and a mapping from declarations to values. A heap is a mapping from frame ids to frames. Two types of relations are defined: frame operations ($\Rightarrow$) and the resolution relation ($\longrightarrow$). Frame operations operate in the context of a modifiable heap, which is represented by the $/h$ syntax. The resolution rules operate under a immutable heap (indicated by $\vdash_h$) and in the context of a set of seen frame ids (F).

The main differences with the original formulation are that the frame component is extended with a set of references (Σ) and that resolution of variables to declarations happens at run-time. This is reflected by the change in the *DLookup* rule, and the introduction of the *DPath* and *EPath* rules. The *DLookup* rule results in an *Addr* which points to a unique location in the heap via the frame id and the declaration, and contains the resolution path. The path can be used in the dynamic semantics of a language to decide whether an existing declaration needs to be overwritten or shadowed. The remaining two elements provide the required data for both the *get* and *set* operations to get and respectively set values at the specific location. The extended definition also includes the *Link* rule, which makes it possible to extend the dynamic links component of frames. Moreover, in contrast to the original definition, the available declarations are not fixed, as indicated by the removal of the requirement that a declaration is in the domain of $D_h(f)$ in the *set* rule.

Although we have removed the static scope graph dependency, we still have a *ScopeId* component in the definition of frames. This component is required to enable the translation from frames to scope graphs, since multiple frames can refer to the same scope, which is something we want reflected in the scope graph obtained from this translation: the information in all those frames needs to be captured by the shared scope in the resulting scope graph. A language semantics thus needs to label frames with this information. In practice, the source location of frame producing constructs, such as functions, can be used as the scope id. With this in mind, we give a translation function (ϕ) that translate a heap into a scope graph, which uses a helper function (ψ) that translates a frame into a scope graph. $Dom_s(h)$ projects all frames with scope id s.

$$\begin{aligned}\psi(\langle s, \Sigma, ks, \sigma\rangle) = \{s \longrightarrow (d, D) \mid \forall d \in Dom(\sigma)\}\\ \cup \{(r, D) \longrightarrow s \mid \forall r \in \Sigma\} \cup \{s \longrightarrow_{(l,D)} s'\\ \mid \forall l \in Label \wedge S(ks(l)) = s'\}\end{aligned}$$

$$\phi(h) = \bigcup\{\bigsqcup\{\psi(h(f)) \mid f \in Dom_s(h)\} \mid s \in \mathcal{S}(h)\}$$

$$\frac{k \vdash_G s \longrightarrow x_i^D \qquad i \le k}{k \vdash_G \mathbf{D}(x_i^D) : s \mapsto (s, x_i^D)} \quad (\textsc{ResD})$$

$$\frac{s'_{k_2} \notin S \qquad k, S \vdash_G s_{k_1} \longrightarrow_l s'_{k_2} \qquad k_2 \le k \qquad k_1, \{s'_{k_2}\} \cup S \vdash_G p : s' \mapsto (s'', x_i^D) \qquad WF(\mathbf{E}(l,s) \cdot p)}{k, S \vdash_G \mathbf{E}(l,s) \cdot p : s \mapsto (s'', x_i^D)} \quad (\textsc{ResE})$$

$$\frac{\vdash_G x_i^R \longrightarrow s \qquad i, \{s\} \vdash_G p : s \rightharpoonup (s', x_i^D)}{\vdash_G p : x_i^R \mapsto (s', x_i^D)} \quad (\textsc{ResR})$$

Figure 5. Resolution calculus extended with context-dependent name resolution via a visibility index.

The translation is sound with respect to definitely present edges, references, and declarations, which follows from the definition of the $\le_s$ relation. The translation is not sound on maybe present entities, i.e. the scope graph can contain resolution paths with M labels that are not present in the heap. Reasoning thus happens purely with the D annotated values.

4.2 Natural Semantics with Heap and Frames

The modifications to the heap and frames definition requires additional bookkeeping in the dynamic semantics of a language. Frames need to be annotated with their scope id, and modified heaps need to be propagated. Also, the dynamic semantics needs to decide when a declaration is constructed versus when an existing declaration is used. To illustrate these adjuncts, we introduce a small dynamically typed language à la Cousot [1] with functions, which we utilize as a running example throughout the rest of this paper. The concrete syntax specification is given in Figure 7, and the dynamic semantics is given in Figure 8. As in prior work [23], we define the dynamic semantics in the context of a current frame id and the current heap: $f \vdash e/h \Longrightarrow v/h'$ means that program e evaluates in the context of heap h with the current frame identified by f to v and updated heap h'.

Most of the resolution is hidden within the frames definition. In the dynamic semantics we only differentiate between declarations and modifications. In our example language, this is done with the *S-assign* and *S-assign-new* rules. The *S-assign-new* rule creates a declaration in the current frame, while the *S-assign* rule updates an existing declaration. In addition, most rules of the dynamic semantics now modify the heap, as indicated by the change in subscripts. This is especially apparent in the *E-min* rule. Expressions generally do not alter the heap. In our case, heap modifications might arise due to variable references, which extend the Σ set of the current frame. The *Def-fn* rule demonstrates the choice of scope ids when constructing frames. In the rule, the source label, indicated by the l subscript, is stored in the closure (identified by *Clos*). When calling a function, described by the *E-call* rule, the source label is used as the scope id for the newly created frame in which the body of the function is evaluated. Multiple calls to the same function thus result in different frames that share the same scope id.

5 Abstract Interpretation with Heap and Frames

Having defined heap and frames without an underlying scope graph, we focus on abstract interpretation using this new definition. We provide a language-independent abstraction, which is parameterized by an abstraction for values.

5.1 Safe Heap and Frames

We provide a language-independent abstract definition of heap and frames, and define a safety relation between concrete and abstract heap and frames. Using this safety relation, we obtain a Galois connection between the concrete and abstract definitions. In our abstract definition, we restrict the number of frame ids to a finite amount. This requires removal of the $f' \notin Dom(h)$ premise in the abstract definition of the *InitFrame* rule. Moreover, the slots function becomes a mapping from declarations to abstract values, with the requirement that the abstract values domain forms a complete lattice. We also annotate declarations, references, and labeled edges in our abstract frames. Also, we define an ordering on abstract frames that is similar to the ordering defined on annotated scope graphs, but altered to align with the heap and frames definition, and extended with a constraint on slots: $\forall d \in Dom(\sigma_1).\sigma_1(d) \le \sigma_2(d)$, where σ_1 and σ_2 are the slots functions for the two frames under comparison. We also require that the associated scopes of the two compared frames are equal. This ordering is also extended to a lattice. With our abstract definition, we define a safety relation between concrete heaps and abstract heaps.

h_c safeHeap h_a iff
    $\forall f \in Dom(h_c).\ \exists f_a \in Dom(h_a).\ h_c(f)$ safeFrame $h_a(f_a)$,

Frames and Heaps

$$f \in \mathit{FrameId} = \{f_1, f_2, \dots\}$$
$$ks \in \mathit{DynLinks} = \mathit{Label} \longrightarrow \mathit{FrameId}$$
$$\sigma \in \mathit{Slots} = \mathit{Decl} \longrightarrow \mathit{Val}$$
$$\Sigma \in \mathit{Refs} = \mathcal{P}(\mathit{Vars})$$
$$\langle s, \Sigma, ks, \sigma\rangle \in \mathit{Frame} = \mathit{ScopeId} \times \mathit{Refs} \times \mathit{DynLinks} \times \mathit{Slots}$$
$$h \in \mathit{Heap} = \mathit{FrameId} \longrightarrow \mathit{Frame}$$

Projection functions

$$\mathcal{S}_h(f) = s \text{ where } h(f) = \langle s, \Sigma, ks, \sigma, \rangle$$
$$\mathcal{R}_h(f) = \Sigma \text{ where } h(f) = \langle s, \Sigma, ks, \sigma, \rangle$$
$$\mathcal{K}_h(f) = ks \text{ where } h(f) = \langle s, \Sigma, ks, \sigma, \rangle$$
$$\mathcal{D}_h(f) = \sigma \text{ where } h(f) = \langle s, \Sigma, ks, \sigma, \rangle$$

Frame modifications

$$\frac{f' \notin \mathit{Dom}(h)}{\mathit{initFrame}(s, \Sigma, ks, \sigma)/h \Rightarrow f'/h[f' \mapsto \langle s, \Sigma, ks, \sigma\rangle]} \ (\text{INITFRAME})$$

$$\frac{h' = K_h(f)[l \mapsto f']}{\mathit{link}(f, f', l)/h \Rightarrow h'} \ (\text{LINK})$$

Paths

$$\frac{x_j^D \in \mathit{Dom}(\mathcal{D}_h(f))}{\vdash_h \mathbf{D}(x_j^D) : f \longrightarrow (f, x_j^D)} \ (\text{DPATH})$$

$$\frac{x_j^D \notin \mathit{Dom}(\mathcal{D}_h(f)) \quad l \in \mathit{Dom}(\mathcal{K}_h(f)) \quad \mathcal{K}_h(f)(l) = f' \quad f' \notin F \qquad F \cup \{f'\} \vdash_h p : f' \longrightarrow (f'', x_j^D) \quad \mathit{WP}(p)}{F \vdash_h \mathbf{E}(l, f) \cdot p \longrightarrow (f'', x_j^D)} \ (\text{EPATH})$$

Dynamic lookup

$$\frac{\{f\} \vdash_h p : f \longrightarrow (f', x_j^D) \qquad \nexists k, p', f''.(p' : f \longrightarrow (f'', x_k^D) \wedge p' < p)}{\mathit{lookup}(f, x_i^R)/h \Rightarrow \mathit{Addr}(f', x_j^D, p)/h[f \mapsto (\mathcal{R}_h(f) \mapsto \mathcal{R}_h(f) \cup \{x_i^R\})} \ (\text{DLOOKUP})$$

Slot value operations

$$\frac{x_i^D \in \mathit{Dom}(\mathcal{D}_h(f)) \qquad \mathcal{D}_h(f)(x_i^D) = v}{\mathit{get}(f, x_i^D)/h \Rightarrow v} \ (\text{GET})$$

$$\frac{}{\mathit{set}(f, x_i^D, v)/h \Rightarrow ()/h[f \mapsto (D_h(f)[x_i^D \mapsto v])]} \ (\text{SET})$$

Figure 6. Modified formal definition of frames and heaps [23].

$x, fn \in \mathit{Var}$		(variables, function variables)
$p \in P$	$::= l$	(programs)
$l \in Sl$	$::= s\ l \mid \epsilon$	(lists of statements)
$s \in S$	$::= x = e \mid \text{if}(e)\ \{l_1\}\ \text{else}\ \{l_2\} \mid \text{def}\ fn(x)\ \{\ l\ \}$	(statements)
$e \in E$	$::= e_1 - e_2 \mid 1 \mid x \mid fn(e)$	(expressions)

Figure 7. Grammar of a simple procedural language.

$$\begin{aligned}
\langle s, \Sigma, ks, \sigma\rangle_{h_c} \text{ safeFrame } \langle s_a, \Sigma_a, ks_a, \sigma_a\rangle_{h_a} \text{ iff} \\
& s = s_a \wedge \Sigma \subseteq \Sigma_a \\
& \forall l \in \mathit{Dom}(ks).\ h_c(ks(l))\ \mathit{safeFrame}\ h_a(ks_a(l)) \\
& \forall d \in \mathit{Dom}(\sigma).\ \sigma(d)\ \mathit{safeVal}\ \sigma_a(d) \\
& D_\sigma(\sigma_a) \subseteq \mathit{Dom}(\sigma) \wedge D_{ks}(ks_a) \subseteq \mathit{Dom}(ks)
\end{aligned}$$

The *safeFrame* relation is parametric in the language-specific *safeVal* relation, and recursive, so the largest set satisfying the relation is required. D_σ projects the definitely present declarations, and D_{ks} projects the definitely present links.

We define the collection semantics as a function $coll_t\ p : \mathit{ProgramPoint} \longrightarrow \mathcal{P}(\mathit{Heap} \times \mathit{Heap})$ from program points to heaps, with t the tree representing an execution.

$$\begin{aligned}
coll_t(p) = &\{(h_1, h_2) \mid f \vdash p/h_1 \Rightarrow v/h_2 \text{ is a state in } t\} \\
&\cup \{(h_1, h_2) \mid f \vdash p/h_1 \Rightarrow h_2 \text{ is a state in } t\}
\end{aligned}$$

The collection semantics collects all heap pairs seen at a program point, with which we can obtain the pair of over-approximating scope graphs for a program point: $\mathcal{G}_t(p) = (\bigsqcup\{\phi(h_1) \mid (h_1, h_2) \in coll_t\ p\}, \bigsqcup\{\phi(h_2) \mid (h_1, h_2) \in coll_t\ p\})$.

5.2 Reasoning with Over-Approximated Scope Graphs

Using the over-approximated scope graphs defined previously, we can define auto-completion services. When the evaluation is productive – $(h_1, p) \Rightarrow h_2 \implies G(h_1) \leq G(h_2)$ – we can take the resulting heaps of the root of the computation tree and reason with the obtained scope graph pair. Otherwise, reasoning does not happen on one scope graph, instead it happens on the scope graph of the program point from where we are reasoning. The advantage of reasoning

$$\frac{f \vdash e/h_1 \Rightarrow v_1/h_2 \qquad lookup(h_2, f, x) \not\Rightarrow \qquad set(h_2, f, x) \Rightarrow h_3}{f \vdash x = e/h_1 \Rightarrow h_3}\ (\textsc{S-assign-new})$$

$$\frac{f \vdash E_1/h_1 \Rightarrow V_1/h_2 \qquad f \vdash E_2/h_2 \Rightarrow V_2/h_3 \qquad V_1, V_2 \in \mathbb{Z}}{f \vdash E_1 - E_2/h_1 \Rightarrow V_1 - V_2/h_3}\ (\textsc{E-min})$$

$$\frac{f \vdash e/h_1 \Rightarrow v_1/h_2 \qquad lookup(h_2, f, x) \Rightarrow Addr(f', x_i^D, p)/h_3 \qquad set(h_3, f', x) \Rightarrow ()/h_4}{f \vdash x = e/h_1 \Rightarrow h_4}\ (\textsc{S-assign})$$

$$\frac{set(h_1, f, fn, Clos(x, sl, l, fn)) \Rightarrow ()/h_2}{f \vdash (def\, fn(x)\{sl\})_l/h_1 \Rightarrow h_2}\ (\textsc{def-fn})$$

$$\frac{\begin{array}{c} f \vdash e_1/h_1 \Rightarrow v_1/h_2 \qquad lookup(h_2, f, fn) \Rightarrow Addr(f', fn^D, p)/h_3 \\ get(h_3, f', fn^D) \Rightarrow Clos(x, b, l, fn_c) \\ initFrame(l, \{\}, \{(P, fn_c)\}, \{(x, v_1)\})/h_3 \Rightarrow f''/h_4 \\ f'' \vdash b/h_4 \Rightarrow v_2/h_5 \end{array}}{f \vdash fn(e_1)/h_1 \Rightarrow v_2/h_5}\ (\textsc{E-call})$$

Figure 8. Big-step operational semantics of part of our running example with explicit frame and heaps.

with one scope graph, is that edit operations can be interpreted as a modification on the single scope graph. As a result, the editor service does not need to re-evaluate the program on every edit. Auto-completion candidates at program point p for computation tree t are given by the visible declarations that are definitely present, where $S(p)$ gives the scope associated with program point p.

$$AC_t(p) = \{x_i^D \mid \vdash_G S(p) \rightharpoonup (s', (x_i^D, D))\}.$$

6 Implementation

We have implemented our approach in Haskell, providing a library for the construction of concrete and abstract interpreters with explicit heap and frames operations.

6.1 Heaps and Frames

The main component of our implementation is the encoding of our formalized heap and frames model. A substantial part of this encoding is a simple 1-to-1 translation from the formal model to Haskell code. Heap and frames are implemented as parameterized data types, with the parameters representing the type for frame ids, scope ids, type of declarations, and the type of values.

```
data Heap f s d v = Heap (Map f (Frame f s d v))
data Frame f s d v = Frame
{sid :: s, refs :: Set d, ks :: Map Label f, slots :: Map d v}
```

The operations on heap and frames follow the formal model and are defined in terms of the parameterized data types. One difference is the handling of frame ids. In our implementation, fresh frame ids need to be provided by the language, since the implementation is parametric in the type of frame ids.

6.2 Abstract and Concrete Interpreters

The abstract and concrete interpreters are language-specific. To ease the construction of these interpreters, we provide several helper functions around a systematic approach based on Sturdy [11], which splits a language implementation into three parts: a generic interpreter, a concrete interpreter, and an abstract interpreter. The generic interpreter describes functionality that is common among all interpretations, and is defined in terms of indeterminate operations. These operations are determined by the concrete and abstract interpreters, resulting in a working interpreter in the respective domain. We define an abstract signature for the operations of the language using type classes. Instances of the type class then give an interpretation to the signature.

6.3 Collecting Semantics and Termination

We utilize lenses [7] to get access to the parameterized heap from the opaque interpreter context. Interpreters are written in an open recursive style, where recursive evaluations are handled by continuation functions. We provide a generic continuation function that annotates program points with the current heap and ensures termination via the productive caching algorithm [4]. Program points are identified by a label, which we obtain via a type class. To sequence computations we utilize monads [16, 32]. Our continuation function is thus parametric in the monad it evaluates in. The final set of annotations corresponds to the collecting semantics.

7 Experiment Design and Results

To demonstrate our approach, we implement a small subset of Python and compare the auto-completion candidates given by our approach with the state of the art auto-completion service providers for Python. To ensure correctness of our implementation, we utilize the existing Python (3.12.3) implementation as an oracle. We have chosen Python because it is a popular programming language[4] and has several editor service implementations, making it a suitable vehicle for us to provide a compelling example of our approach. Our subset supports functions with parameters, classes and objects, primitive types and operations on those types, if-else statements, and while loops.

We have constructed a test set of programs around our subset with key name binding challenges for completion services [8]. In our experiment, we filter the possible completion candidates by prefixing all variables with an x, and only completing on variables starting with an x. This is to prevent the completion list from being filled with standard Python constructs, such as `__name__`.

To implement an abstract interpreter for our Python subset, we use the approach from prior work [17], which performs type checking using abstract interpretation. Our abstract domain thus maps values to their type. The modeling of the heap is slightly different, since we use our heap and frames approach, which is a less compact representation.

To determine in-scope variables, we execute the Python interpreter and execute the `dir` and `globals` functions, which provide access to the available names in the current scope and the global variables currently in scope, respectively. The `dir` function can also be used to obtain the attributes of an object. We execute our programs with these functions placed at the location where we will perform an auto-completion request to obtain the variables in-scope at run-time at the specific location. We run the programs under a variety of inputs to ensure 100% path coverage and then take the intersection of the variables obtained from these executions, which corresponds to the set of variables that are present among all possible paths. In case of the `fn-not-called` program, we obtained in-scope variables by utilizing the respective Python file as a library and testing the function externally.

We have chosen three auto-completion service providers for Python: Pylance, PyCharm[5] (professional edition), and Jedi. Pylance is an implementation of the LSP for Python and uses Pyright[6], a static type checker for Python; PyCharm is a Python IDE developed by JetBrains, which provides Python support via a plugin[7]; and Jedi is a static analysis tool for Python with a focus on auto-completion and goto-definition.

As not all implementations have a public API, we have performed our experiments manually, via VSCode[8] for Jedi and Pylance, and using the PyCharm GUI for PyCharm. For our approach, we have implemented the auto-complete part of the LSP protocol and also used interactions via VSCode to evaluate our approach.

7.1 Results

The results of our experiment are displayed in Table 1, as precision and recall pairs. A precision of 100% equates to soundness. A recall of 100% equates to completeness. In the table, P describes the number of sound completion candidates, and $P + N$ describes the number of sound and unsound completion candidates. This number is constrained by the variables in a program. Modifying the values of P and N can result in a different percentages for precision and recall, but the relation with respect to P and N stays consistent.

The results highlight a strategy present in the current state of the art: completeness is preferred over soundness. Consequently, even on simple programs, the state of the art introduces completion candidates that can introduce name binding errors when selected by the user. Our approach prefers soundness over completeness, and presents no unsound completion candidates with respect to name binding for programs in our experiment. Additionally, among the state of the art we observe similar results. The small differences can be explained by the underlying approach of the different services, or by the development hours put into a service. Another observation is that recall is generally around a 100%. In most cases, obtaining a high percentage for recall can be easy, because an editor service can collect all variables in the program and present those as completion candidates.

Some of the programs in our demonstration could also be handled by the state of the art tools by strengthening their analyses without adding significant complexity. There are also programs, such as *add-field*, which require more complex analysis and the gain of soundness on such candidates might not be worth the increase in complexity. Nevertheless, the set of evaluated programs gives an overview of key name binding constructs on which analysis can be improved in the state of the art auto-complete services for Python.

8 Discussion

In this section we compare our approach to the state of the art, and give several suggestions for future work, including steps towards (full) soundness.

8.1 Missing Candidates and Uncalled Functions

Our approach misses several valid completion candidates. For example, the *if-else1* program contains a condition that is always false, which is detected by Jedi. Since we used an abstract domain that maps values to types, we lose such information, which can lead to lower recall. This is also shown by the program *if-else-both*, where both branches of an `if-else` are taken by repeating the statement but taking a different branch. Due to our chosen abstract domain, our approach fails to determine that the variables introduced in

[4] https://survey.stackoverflow.co/2024/technology#2-programming-scripting-and-markup-languages
[5] https://www.jetbrains.com/pycharm/
[6] https://github.com/microsoft/pyright
[7] https://plugins.jetbrains.com/plugin/631-python
[8] https://code.visualstudio.com/

Table 1. Results of our experiment on the state of the art on the Python test set. P denotes the sound completion candidates. $P+N$ denotes the total available completion candidates, so sound and unsound. Precision describes the percentage of completion candidates that were sound compared to all given completion candidates by a tool. Recall describes the percentage of sound completion candidates that were reported by the tool compared to all possible sound completion candidates at the program point. N/A indicates that a tool gave no completion candidates.

		PyCharm		**Jedi**		**Pylance**		**Our work**	
Program	$P : (P + N)$	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*
self-ref	1 : 2	50%	100%	50%	100%	50%	100%	100%	100%
if-else1	2 : 3	66.7%	100%	100%	100%	66.7%	100%	100%	50%
if-else2	1 : 3	33.3%	100%	33.3%	100%	33.3%	100%	100%	100%
if-else3	2 : 3	100%	100%	66.7%	100%	66.7%	100%	100%	100%
if-else-both	3 : 3	100%	100%	100%	100%	100%	100%	100%	33%
duck-type	1 : 3	33.3%	100%	33.3%	100%	33.3%	100%	100%	100%
use-before-define	1 : 2	100%	100%	100%	100%	50%	100%	100%	100%
use-before-define-fn	1 : 2	50%	100%	50%	100%	50%	100%	100%	100%
add-field	2 : 2	100%	50%	100%	50%	100%	50%	100%	100%
fn-not-called	2 : 2	100%	100%	100%	100%	100%	100%	N/A	N/A
obj-param	1 : 3	33.3%	100%	33.3%	100%	N/A	N/A	100%	100%
global-add	2 : 2	100%	50%	100%	100%	100%	100%	100%	100%
not-global-add	1 : 2	100%	100%	50%	100%	50%	100%	100%	100%

both branches are definitely in scope. This can be solved by modifying the abstract domain. However, as is inherent to abstract interpretation, there is a trade-off between performance and the precision of the abstract domain. In future work, we would like to investigate the effects of different abstract domains on the recall and performance.

A major drawback of our approach is that uncalled functions have an associated empty scope. Ergo, completion candidates within such a function are limited to what is available via the parent scope. This is tested by the *fn-not-called* program. The state of the art has no problem with this specific case. In future work we aim to investigate suitable conditions under which we can safely simulate the calling of these functions to still obtain an over-approximated scope graph.

8.2 Being Fully Sound

In this work we have made a first step towards our goal, which is to obtain sound editor services. To attain it, we need to fully formalize our approach and prove that our heap and frames operations respect the safety relations. We are currently in the process of proving our approach using Lean [5]. Furthermore, languages need to prove that their abstract evaluation relation is sound by showing that it respects that safety relation.

During informal discussions with Python developers, they indicated that unsound completion candidates can helpful during exploration or refactoring. Hence, just providing sound completion candidates might hamper developers during such activities. We overcome this by retaining the possibly unsound completion candidates. These can still be provided as candidates by utilizing the annotations to provide developers with more information. For example, by giving unsound suggestions an indicator to communicate that they might introduce erroneous execution paths. This would also provide a fallback option for uncalled functions. In future work, we aim to investigate this further and compare the user experience via user studies.

8.3 Incremental Analysis and Imports

The original scope graph framework supports module imports. We have omitted this to simplify the presentation. However, we do not see any theoretical difficulties in supporting module imports with an extra rule to the resolution calculus. However, difficulties with modules might arise with respect to scalability and usefulness. Building detailed scoping information for modules which are not modified by the programmer is not beneficial. Nevertheless, imported functions can modify variables in such a way that they affect the scoping in the module in which the programmer is working. Think of a function that adds fields to an object. To handle this, we want to investigate capturing the transformations made by imported functions using predicate transformer semantics [6].

8.4 Handling of Incorrect Programs

In this paper we have worked under the assumption that programs are correct, while editor services often operate on broken programs or programs with holes. In case of broken programs, we can continue our abstract interpretation but mark all results as unsound, and show them with an annotation to indicate the possible introduction of erroneous execution paths. For programs with holes, a language can define the semantics and do the interpretation over such programs. However, it is unclear how scalable this is. Alternatively, language designers can define a projection from programs with holes to programs without . The interpretation is then performed on the program without holes and the results can be applied on the program with holes. Furthermore, for simple editing operations, we can modify the scope graph without re-interpretation of the program, and can therefore work with holed programs. We thus do not see any substantial difficulties in adapting our approach to handle incorrect program and programs with holes.

9 Related Work

Scope graphs [18] have been used as a blueprint for dynamic memory [23]. Van Antwerpen et al. [29, 30] have used scope graphs to perform static analysis of types, initialization, and name binding in a language-independent manner. Zwaan et al. [36] give a detailed overview of these developments.

Scope graphs have also been used to obtain language-parametric editor services for statically typed languages [21]; to preserve well-typedness during automated refactoring [15]; and in the construction of completion services for statically typed languages [20] that support both syntactic and semantics completions. Compared to our work, we do not see an immediate way to obtain preservation of well-typedness for automated refactorings, due to incompleteness. With regards to completion services, prior work focused on statically typed languages and used grammars and type specifications. Our work requires an abstract interpreter, and is purely focused on semantic completions and dynamically typed languages.

Stack graphs [3] is a modification of scope graphs that support file-incremental analysis with type-dependent look ups, and powers code navigation at GitHub. The approach is mostly focused on code navigation, which boils down to resolving references to declarations. To be able to support new languages with ease, the authors have constructed a graph construction language on top off the tree-sitter parser framework. Languages which have a tree-sitter parser can define patterns that map language constructs to operations on stack graphs, independent of the type of language. To handle more complex situations, the approach uses data-flow analysis. However, it is unclear whether the data-flow semantics is extracted out of patterns or requires additional effort. With our approach, the primary focus is on completion services, and no additional effort is required to support more complex name resolution scenarios. However, the initial effort required by our approach is much more substantial due to the need of a working abstract interpreter.

9.1 Data-Flow Analysis and Abstract Interpretation

Many editor services use some kind of data-flow analysis [13, 19] to support more complex scenarios. For example, use-definition chains [12] can be used to determine to which declaration(s) a reference belongs. The monotone framework [10] provides a reusable pattern for defining data-flow analysis in a systematic manner.

Data-flow analysis and abstract interpretation are tightly connected [26, 28]. Our approach could be more data-flow oriented, using the monotone framework. Nevertheless, the usage of abstract interpretation in combination with the code structuring technique promoted by Sturdy [11], provides much opportunity for reuse between the interpreters.

Prior work combined abstract interpretation with statistical models to provide completion candidates [24]. The approach was evaluated on Java, and only a small percentage of the completion proposals gave type errors. It is unclear whether that included name binding errors, and how it performs on a dynamically typed language.

9.2 Code Completion Using Machine Learning

In recent years, code tasks, such as auto-completion, have received significant attention from the machine learning community [34]. Transformers [31] have been used by Kim et al. [14] to improve candidates over pre-existing machine-learning based techniques, and shows promising results, but incorrect predictions are possible. Besides just giving completion suggestions, modern systems are capable of more by synthesizing snippets from comments or from context, such as GitHub Copilot[9] and Tabnine[10]. We have not included such systems in our comparison since the tasks performed by these systems is rather different and requires a different form of evaluation [35].

10 Conclusion

In this paper we have investigated the construction of auto-completion services for dynamically typed languages, such that the given completion candidates do not cause name binding errors. To achieve this, we have extended the scope graph framework with annotations and context-dependent name resolution. Furthermore, we have used abstract interpretation to obtain a sound over-approximation of the name binding seen at run-time. Combined with the 1-to-1 correspondence between scopes and frames, we obtained an over-approximated scope graph. To demonstrate our approach, we applied it to a small subset of Python and compared it to the state of the art editor services. On this test set, our approach outperformed the state of the art with respect to the sound completion candidates, and sometimes also with respect to completeness. However, uncalled functions presents a difficulty for our approach, missing some valid completion candidates. Finally, we have discussed the steps needed to obtain auto-completion services that are fully sound with respect to name binding.

[9]https://github.com/features/copilot/
[10]https://www.tabnine.com/

References

[1] Patrick Cousot. 2021. *Principles of abstract interpretation.* MIT Press.

[2] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[3] Douglas A. Creager and Hendrik van Antwerpen. 2023. Stack Graphs: Name Resolution at Scale. In *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands (OASIcs, Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:12. https://doi.org/10.4230/OASICS.EVCS.2023.8

[4] David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proc. ACM Program. Lang.* 1, ICFP (2017), 12:1–12:25. https://doi.org/10.1145/3110256

[5] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

[6] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975

[7] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (2007), 17. https://doi.org/10.1145/1232420.1232424

[8] Damian Frolich and Thomas van Binsbergen. 2024. A selection of Python programs with key name binding challenges for auto-completion services. https://doi.org/10.5281/zenodo.13628718

[9] Gilles Kahn. 1987. Natural Semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings (Lecture Notes in Computer Science, Vol. 247)*, Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.). Springer, 22–39. https://doi.org/10.1007/BFB0039592

[10] John B. Kam and Jeffrey D. Ullman. 1977. Monotone Data Flow Analysis Frameworks. *Acta Informatica* 7 (1977), 305–317. https://doi.org/10.1007/BF00290339

[11] Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional soundness proofs of abstract interpreters. *Proc. ACM Program. Lang.* 2, ICFP (2018), 72:1–72:26. https://doi.org/10.1145/3236767

[12] Ken Kennedy. 1978. Use-Definition Chains with Applications. *Comput. Lang.* 3, 3 (1978), 163–179. https://doi.org/10.1016/0096-0551(78)90009-7

[13] Uday P. Khedker, Amitabha Sanyal, and Bageshri Sathe. 2009. *Data Flow Analysis - Theory and Practice.* CRC Press.

[14] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021.* IEEE, 150–162. https://doi.org/10.1109/ICSE43902.2021.00026

[15] Luka Miljak, Casper Bach Poulsen, and Flip van Spaendonck. 2023. Verifying Well-Typedness Preservation of Refactorings using Scope Graphs. In *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2023, Seattle, WA, USA, 18 July 2023*, Aaron Tomb (Ed.). ACM, 44–50. https://doi.org/10.1145/3605156.3606455

[16] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[17] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2020. Static Type Analysis by Abstract Interpretation of Python Programs. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:29. https://doi.org/10.4230/LIPICS.ECOOP.2020.17

[18] Pierre Neron, 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

[19] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis.* Springer. https://doi.org/10.1007/978-3-662-03811-6

[20] Daniël 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 (2022), 1–30. https://doi.org/10.1145/3527329

[21] 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 für Informatik, 26:1–26:18. https://doi.org/10.4230/LIPICS.ECOOP.2019.26

[22] Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.* 60-61 (2004), 17–139.

[23] Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. 2016. Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:26. https://doi.org/10.4230/LIPICS.ECOOP.2016.20

[24] 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, 419–428. https://doi.org/10.1145/2594291.2594321

[25] David A. Schmidt. 1995. Natural-Semantics-Based Abstract Interpretation (Preliminary Version). In *Static Analysis, Second International Symposium, SAS'95, Glasgow, UK, September 25-27, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 983)*, Alan Mycroft (Ed.). Springer, 1–18. https://doi.org/10.1007/3-540-60360-3_28

[26] David A. Schmidt. 1998. Data Flow Analysis is Model Checking of Abstract Interpretations. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, David B. MacQueen and Luca Cardelli (Eds.). ACM, 38–48. https://doi.org/10.1145/268946.268950

[27] David A. Schmidt. 1998. Trace-Based Abstract Interpretation of Operational Semantics. *LISP Symb. Comput.* 10, 3 (1998), 237–271.

[28] David A. Schmidt and Bernhard Steffen. 1998. Program Analysis *as* Model Checking of Abstract Interpretations. In *Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1503)*, Giorgio Levi (Ed.). Springer, 351–380. https://doi.org/10.1007/3-540-49727-7_22

[29] Hendrik van Antwerpen, Pierre Neron, 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

[30] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as types. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 114:1–114:30. https://doi.org/10.1145/3276484

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008.

[32] Philip Wadler. 1992. Monads for functional programming. In *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992 (NATO ASI Series, Vol. 118)*, Manfred Broy (Ed.). Springer, 233–264. https://doi.org/10.1007/978-3-662-02880-3_8

[33] Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023. How Practitioners Expect Code Completion?. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1294–1306. https://doi.org/10.1145/3611643.3616280

[34] Man-Fai Wong, Shangxin Guo, Ching Nam Hang, Siu-Wai Ho, and Chee-Wei Tan. 2023. Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review. *Entropy* 25, 6 (2023), 888. https://doi.org/10.3390/E25060888

[35] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the quality of GitHub copilot's code generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2022, Singapore, Singapore, 17 November 2022*, Shane McIntosh, Weiyi Shang, and Gema Rodríguez-Pérez (Eds.). ACM, 62–71. https://doi.org/10.1145/3558489.3559072

[36] Aron Zwaan and Hendrik van Antwerpen. 2023. Scope Graphs: The Story so Far. In *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands (OASIcs, Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:13. https://doi.org/10.4230/OASICS.EVCS.2023.32

Received 2024-06-18; accepted 2024-08-15