An Introduction to Indirect Code Completion

Nhat[1], Vadim Zaytsev[1]

[1]*Formal Methods & Tools (FMT), University of Twente, Enschede, The Netherlands*

Abstract

Code completion plays a vital role in enhancing software development productivity and quality. It has evolved from simple spell checkers to advanced AI-powered tools, yet the core principle remains the same: to provide code suggestions directly where the completion was initiated. In this paper, we proposed to generalise it to *indirect code completion* (ICC), which can be initiated in one place while having suggestions proposed elsewhere. By analysing the structure and properties of ICC in the context of object-oriented languages, we identified several properties that can be used to characterise ICC. We also introduced a set of questions and a taxonomy to categorise ICC into over 20 different application patterns.

Keywords

code completion, coding context

1. Introduction and Background

The "type less, write more" idea has become familiar to most developers, with applications ranging from query completion in modern search engines based on popular queries, to character suggestions when typing in languages with thousands of characters. For code, completion, when triggered, commonly appears in the form of a list of suggestions based on the user's prompt and the surrounding context. The earliest code completion (CC) systems are merely spell checkers [1] by today's standards. Modern CC systems can also collect and prioritise suggestions of words that already appeared in the file/codebase [2]. This led to inspection systems that perform syntactic and semantic analysis on the codebase to gain an understanding of the code to provide richer and more contextually relevant completions.

Code completion has become a norm, an expectation, and a reliance for modern IDEs and code editors to have a CC system that can offer correct types, variables, fields, methods, classes, and other language-specific constructs. Today, this kind of modern code completion is commonly known as intelligent code completion, code suggestion, code prediction, auto-completion, etc., depending on the provider. It significantly enhances the coding experience and productivity, while improving code quality by increasing efficiency and accuracy, promoting discoverability, reducing cognitive load and enhancing focus, and encouraging consistency and best practices.

Machine learning (ML) and artificial intelligence (AI) have been recently and rapidly integrated into CC solutions, such as GitHub Copilot or JetBrains AI Assistant. Even for these intelligent systems, the core principle of most, if not all, CC remains the same: to provide suggestions for relevant code elements directly where the CC was initiated. For example, the following code could have three consecutive CC events, with the user-written completion prompts in blue and IDE-proposed completion edit in red:

```
1  pizzaBuilder.withCrust().withSalami().withGarlic() // RawPizza
2              .bake()|                               // BakedPizza
3
4  pizzaBuilder.withCrust().withSalami().withGarlic() // RawPizza
5              .bake().slice()|                       // ReadyPizza
6
7  pizzaBuilder.withCrust().withSalami().withGarlic()
8              .bake().slice().serve()|
```

BENEVOL'25: Belgium-Netherlands Software Evolution Workshop 2025, November 17–18, 2025, Enschede, The Netherlands
research@nhat.run (Nhat); vadim@grammarware.net (V. Zaytsev)
https://grammarware.net/ (V. Zaytsev)
0009-0004-3110-9946 (Nhat); 0000-0001-7764-4224 (V. Zaytsev)

Instead, we could think of the following one event that can produce the same result:

```
pizzaBuilder.withCrust().withSalami().withGarlic()
            .bake().slice().serve()
```

Here, in green, we denoted the target of the completion: that is, the place where the edit is aiming at. While not necessarily winning in keystrokes, this *bridge* ICC event supports the developers in letting them focus on the essentials while it can add the necessary. Another example could be an *insert* ICC event that allows the user to add another topping without moving the caret:

```
pizzaBuilder.withCrust().withSalami().withGarlic()
            .bake().slice().serve().withHam

pizzaBuilder.withCrust().withSalami().withGarlic().withHam()
            .bake().slice().serve()
```

Collectively, we call these kinds of code completion ***indirect code completion*** (ICC), as opposed to the traditional *direct code completion* (DCC) where the edit is applied directly at the location of the caret. In the following sections, we will explore in more detail how ICC can minimise developer's distractions by performing completions elsewhere without navigating to the target place first. This example merely serves as a motivation, but style-wise besides the (already popular) *builder* pattern, it is also applicable to the *fluent interface* design pattern for its extensive use of *method chaining* and *method cascading*.

While it is simple to jump to the beginning or the end of a line (e.g., using the Home or End key), moving the caret to some specific location requires intention and finesse, potentially disrupting the flow [3]. Caret movement is a frequent and basic action yet often taken for granted. It is usually done with a mouse pointer for IDEs with a graphical interface or by a combination of modes, commands, shortcuts, and controls for purely textual ones. For the latter, it is worth noting that these cursor controls in *Vim* or *Emacs* mostly involve jumping back or forth a number of characters, words, sentences, lines, or paragraphs, — all constructs of natural languages, not programming ones.

Indirect code competition provides a quick and simple way to make indirect edits in conceptually related places and reduce caret jumps while writing code. With ICC, a sequence of direct code completion and navigation events can be replaced with a single ICC event, enhancing productivity while reducing time, errors, and cognitive load. By allowing suggestions that are not bound to the initiated location, ICC can have more opportunities to complete and support the coding process.

In this paper, we present our approach and analysis of the structure and properties of indirect code completion in § 2, specifically focusing on the context of object-oriented languages. Using these properties as a base, a comprehensive taxonomy for different categories of ICC is defined. § 4 concludes the paper with some remarks and future research possibilities. Although some early evaluation of ICC has been conducted with promising results (and can be shared during the workshop), a more comprehensive evaluation is left for future work and is not included in this paper.

2. Categories of Indirect Code Completion

This section explains our approach to investigating the concept of indirect code completion for object-oriented languages. Let us define a member f of the type (class, module, package) T as $T.f(P) \to R$, if its arguments are P and its return type is R. In some cases when this is important, we will write out all arguments individually: $T.f(P_0, P_1, \dots, P_k) \to R$ for some $k \in \mathbb{N}$.

If these elements are to be chained, then we can have a series of $T_i.f_i$ for which $R_i \equiv T_{i+1}$, and write the chain down as $T_0.f_0(P_0) \to T_1.f_1(P_1) \to T_2 \dots T_n.f_n(P_n) \to R_n$ or even as $T_0.f_0 \to T_1.f_1 \to T_2 \dots T_n.f_n \to R_n$ if parameters are not important. To identify and categorise the possible kinds of ICC, let us examine the following element chain with a hypothetical ICC:

$$f_0.f_1 \dots f_i.g.f_{i+1} \dots f_n.x \dots f_{n+1} \dots f_{n'} \tag{1}$$

In the above chain (1), a completion prompt x is provided by the user in the middle of the chain after the element f_n. For simplicity's sake, the remaining elements from f_{n+1} onward are temporarily ignored. Here, x is prompted with the intention to make a completion edit g after the target element f_i.

Consider a completion edit g. In direct code completion, the completion edit is almost always closely related to the completion prompt x, typically based on the element's name. For instance, given a completion prompt `toStr`, a logical completion edit that the user aimed for could be `toString`, which starts with the same characters. However, g does not always need to be closely related to x and can be *fresh* as we have seen from the motivational example before.

The idea of ICC is based on the ability to complete a piece of code that is not directly at the position where the CC was initiated. In that sense, to be indirect, the completion edit g needs to be at a certain *distance* to the completion prompt x.

In the specific case in (1), the completion edit g is placed between the elements f_i and f_{i+1}. I.e., the *completion action* was to *insert* g after the target element f_i, leaving all original elements intact. An alternative completion action could have been to *replace* f_i with g instead of inserting it. Finally, the target element could also be used as an argument for g: this *wrapping* action results in $g(f_i)$.

Until now, an ICC only targets one element f_i. The *range* of the target, however, is not limited to one. Instead of f_i, a sequence of elements $f_i \dots f_{i+j}$ can be targeted as once, e.g., with the wrapping action:

$$f_0.f_1 \dots f_{i-1}.g(f_i \dots f_{i+j}).f_{i+j+1} \dots f_n.x| \; f_{n+1} \dots f_{n'} \tag{2}$$

Similarly, this generalisation also holds for the completion edits. A completion edit can also consist of a sequence of elements $g_0 \dots g_m$ of a *length* greater than one. An example of this with the replace action could be:

$$f_0.f_1 \dots \cancel{f_i \dots f_{i+j}} g_0 \dots g_m.f_{i+j+1} \dots f_n.x| \; f_{n+1} \dots f_{n'} \tag{3}$$

As can be observed from the above chains, the ICCs targeted elements *backward* to the completion prompt x, given the writing direction is from left to right. Let us turn focus to the previously ignored tail elements of the chain, i.e., in the *forward* direction. Similarly, all the previously defined properties also hold for the forward direction, with one exception being that the completion prompt is now effectively prepended to the tail chain. For example, the following chain is a forward version of (3):

$$f_0 \dots f_n \; x|.f_{n+1}.f_{n+2} \dots \cancel{f_i \dots f_{i+j}} g_0 \dots g_m.f_{i+j+1} \dots f_{n'} \tag{4}$$

More compactly, both backward (3) and forward (4) ICC can be represented together using the under-arrows (e.g., $\underrightarrow{x}$ for forward and $\underleftarrow{x}$ for backward) to denote the completion direction relative to where the completion was initiated, using both if both are possible starting points:

$$\dots \underrightarrow{x}|f_0.f_1 \dots \cancel{f_i \dots f_{i+j}} g_0 \dots g_m.f_{i+j+1} \dots f_n \underleftarrow{x}| \dots \tag{5}$$

Collectively, these properties (target distance, target range, completion action, completion edit freshness, completion edit length, completion direction, and depth) form the basis of the proposed ICC categorisation. To further streamline the analysis, we gathered and organised completion properties into a set of questions:

- What is the *distance* from x to the first target f_i?
- What is the *range* of the target elements?
- What is the *completion action*?
- What is the *freshness* of the completion edit?
- What is the *length* of the completion edit?
- What is the completion *direction* relative to x?

Answering these questions results in a combination of properties, which are used to categorise the ICC. These properties can also help distinguish between direct and indirect code completion. To illustrate this, let us consider a simple direct code completion $f_0.f_1 \dots f_n.x|$. Direct code completion is a special case of CC in the forward direction with a target range of only one element f_n directly before the completion prompt x. The insert action is used by direct code completion to append a completion edit x of length 1 that is closely related to the completion prompt x.

Table 1
Summary of indirect code completion categories for the pattern $\ldots \underset{\rightarrow}{x|}.f_0.f_1 \ldots f_i \ldots f_{i+j} \ldots f_n.\underset{\leftarrow}{x|} \ldots$

Compl. action	Special case	Compl. target		Compl. edit			Applied pattern
		Element	Range	Element	Param.	Length	
insert	—	f_i	1	x	—	1	$\ldots \underset{\rightarrow}{\lvert} f_0 \ldots f_i.x.f_{i+1} \ldots f_n \underset{\leftarrow}{\rvert} \ldots$
	append	f_n	1	x	—	1	$\ldots f_0 \ldots f_n.\underset{\leftrightarrow}{x\rvert} \ldots$
fill	—	f_i	1	g	—	1	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_i.g.f_{i+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
		f_i	1	$g_0 \ldots g_m$	—	$m+1$	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_i.g_0 \ldots g_m.f_{i+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
	bridge	$\underset{\rightarrow}{f_0}$ or $\underset{\leftarrow}{f_n}$	1	g	—	1	$\ldots \underset{\rightarrow}{x|.g.f_0}.f_1 \ldots f_{n-1}.\underset{\leftarrow}{f_n.g.x|} \ldots$
		$\underset{\rightarrow}{f_0}$ or $\underset{\leftarrow}{f_n}$	1	$g_0 \ldots g_m$	—	$m+1$	$\ldots \underset{\rightarrow}{x|.g_0 \ldots g_m.f_0}.f_1 \ldots f_{n-1}.\underset{\leftarrow}{f_n.g_0 \ldots g_m.x|} \ldots$
displace	—	f_i	1	x	—	1	$\ldots \underset{\rightarrow}{\lvert} f_0 \ldots f_{i-1}.\cancel{f_i} x.f_{i+1} \ldots f_n \underset{\leftarrow}{\rvert} \ldots$
		$f_i(P_i)$	1	x	P_i	1	$\ldots \underset{\rightarrow}{\lvert} f_0 \ldots f_{i-1}.\cancel{f_i} x(P_i).f_{i+1} \ldots f_n \underset{\leftarrow}{\rvert} \ldots$
		$f_i \ldots f_{i+j}$	$j+1$	x	—	1	$\ldots \underset{\rightarrow}{\lvert} f_0 \ldots f_{i-1}.\cancel{f_i \ldots f_{i+j}} x.f_{i+j+1} \ldots f_n \underset{\leftarrow}{\rvert} \ldots$
	discard	$f_0 \ldots f_n$	$n+1$	x	—	1	$\ldots \underset{\rightarrow}{\lvert} x\cancel{f_0 \ldots f_n}$ or $\cancel{f_0 \ldots f_n} x \underset{\leftarrow}{\rvert} \ldots$
replace	—	f_i	1	g	—	1	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_{i-1}.\cancel{f_i} g.f_{i+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
		$f_i(P_i)$	1	g	P_i	1	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_{i-1}.\cancel{f_i} g(P_i).f_{i+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
		f_i	1	$g_0 \ldots g_m$	—	$m+1$	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_{i-1}.\cancel{f_i} g_0 \ldots g_m.f_{i+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
		$f_i \ldots f_{i+j}$	$j+1$	g	—	1	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_{i-1}.\cancel{f_i \ldots f_{i+j}} g.f_{i+j+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
		$f_i \ldots f_{i+j}$	$j+1$	$g_0 \ldots g_m$	—	$m+1$	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_{i-1}.\cancel{f_i \ldots f_{i+j}} g_0 \ldots g_m.f_{i+j+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
	rewrite	$f_0 \ldots f_n$	$n+1$	g	—	1	$\ldots \underset{\rightarrow}{x|}.g\cancel{f_0 \ldots f_n}$ or $\cancel{f_0 \ldots f_n} g.\underset{\leftarrow}{x|} \ldots$
		$f_0 \ldots f_n$	$n+1$	$g_0 \ldots g_m$	—	$m+1$	$\ldots \underset{\rightarrow}{x|}.g_0 \ldots g_m\cancel{f_0 \ldots f_n}$ or $\cancel{f_0 \ldots f_n} g_0 \ldots g_m.\underset{\leftarrow}{x|} \ldots$
	expand	x	1	g	—	1	$\ldots f_0 \ldots f_n.\cancel{x} g \underset{\leftrightarrow}{\rvert} \ldots$
		x	1	$g_0 \ldots g_m$	—	$m+1$	$\ldots f_0 \ldots f_n.\cancel{x} g_0 \ldots g_m \underset{\leftrightarrow}{\rvert} \ldots$
wrap	—	f_i	1	x	f_i	1	$\ldots \underset{\rightarrow}{\lvert} f_0 \ldots f_{i-1}.x(f_i).f_{i+1} \ldots f_n \underset{\leftarrow}{\rvert} \ldots$
		$f_i \ldots f_{i+j}$	$j+1$	x	$f_i \ldots f_{i+j}$	1	$\ldots \underset{\rightarrow}{\lvert} f_0 \ldots f_{i-1}.x(f_i \ldots f_{i+j}).f_{i+j+1} \ldots f_n \underset{\leftarrow}{\rvert} \ldots$
	surround	$f_0 \ldots f_n$	$n+1$	x	$f_0 \ldots f_n$	1	$\ldots \underset{\rightarrow}{\lvert} x(f_0 \ldots f_n)$ or $x(f_0 \ldots f_n) \underset{\leftarrow}{\rvert} \ldots$
reshape	—	f_i	1	g	f_i	1	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_{i-1}.g(f_i).f_{i+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
		$f_i \ldots f_{i+j}$	$j+1$	g	$f_i \ldots f_{i+j}$	1	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_{i-1}.g(f_i \ldots f_{i+j}).f_{i+j+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
	adapt	$f_0 \ldots f_n$	$n+1$	g	$f_0 \ldots f_n$	1	$\ldots \underset{\rightarrow}{x|}.g(f_0 \ldots f_n)$ or $g(f_0 \ldots f_n).\underset{\leftarrow}{x|} \ldots$
	fit	x	1	g	x	1	$\ldots f_0 \ldots f_n.\underset{\leftrightarrow}{g(x)\rvert} \ldots$

Tab. 1 summarises the different categories of ICC: we primarily categorise ICCs by the completion action, then the targeted element(s), and by the completion edit. In some categories, special cases can be identified by a certain combination of properties. For each category, there can be some variants that differ in the distance, range, length, and direction of the completion.

Let us take a closer look at *Insert*, one of the simplest kinds of ICC. In this category, a completion edit x closely related to the completion prompt x is inserted into the chain after the target element f_i, leaving all the original elements in the chain intact. An example of *insert* ICC was shown in the introduction. More formally, an *insert* ICC can be written as $f_0.f_1 \ldots f_i.x.f_{i+1} \ldots f_n.|f_{n+1} \ldots f_{n'}$.

Here, x can only be chained between f_i and f_{i+1}, if the insertion of x can maintain the type flow of the chain, i.e., x must also be a member of R_i and f_{i+1} is a member of T_x. When $i = n$, the completion edit x is inserted at the exact place where the completion was initiated, effectively a direct code completion.

One can analyse and define other categories of ICC similarly, which, in the interest of space, we will leave up to the reader. As for a brief overview, *fill* is similar to *insert*, but the completion edits are *fresh* elements $g/g_0 \dots g_m$. As a special case of *fill*, *bridge* fills the gap between the completion prompt x and the adjacent target element, as was also seen in the motivating example. Instead of just inserting an element after, *displace* and *replace* also remove the target element(s), effectively displacing/replacing them with the completion edit x or $g/g_0 \dots g_m$ respectively, optionally inherit the parameter P_i of the target. *Discard* and *rewrite* are special cases of *displace* and *replace* that remove the entire head or tail, depending on the completion direction. *Expand* is another special case of *replace* that targets the prompt x itself. A different pair of completion actions are *wrap* and *reshape*; Instead of removing, they consider the target element(s) as arguments for their completion edit x or g respectively, with the special cases *surround* and *adapt* that targeting the entire head or tail, depending on the completion direction. An even more special case *fit* that targets the completion prompt x itself.

3. Related Work

Numerous approaches have been proposed to expand code completion in various dimensions [4, 5, 6]. However, none of these studies investigate the idea of completing the code indirectly. The closest approaches to ICC are template-based completion techniques.

One of the most common template-based completion techniques is code snippet generation/completion. In *Visual Studio (Code)*, snippets [7] are part of *IntelliSense* [8]. In *JetBrains' IDEs*, it is called *Live templates* [9]. Regardless of the branding, the idea of snippet generation stays the same: replace an abbreviated predefined "template identifier" with a predefined template. E.g., with the prompt `fori`, the completion system will replace this with a predefined template `for (int i = 0; i < ?; i++) {...}`. Snippet templates are typically repeating code patterns that can be defined to reduce coding effort.

In a similar manner, *JetBrains' postfix completion* [10] allows predefined templates to be used on an expression via an abbreviated postfix. For example, given an expression `isActive()` that can be evaluated to an expression `boolean`, the postfix completion `isActive().if` matches the predefined template `if($EXPR$) {...}` with template identifier `if` postfixed to an `boolean` expression, resulting in the completion `if(isActive()) {...}`.

Conceptually, these template-based completion techniques resemble *expand* ICC, replacing the completion prompt with a completion edit. If the template is closely related to the completion prompt, it can also be considered as a case of *append*. Nevertheless, these techniques only use predefined templates, individually established, and do not actively offer code completion suggestions as ICC does.

4. Conclusion and Future Work

In this paper, we have introduced *indirect code completion (ICC)* — a novel approach that allows code completion to make edits to other existing elements that are not restricted to the location where the completion was initiated. Analysis on the structure and properties of ICC in the context of object-oriented languages led to the categorisation of different application patterns. Each category can further be formalised to define the requirements and constraints that could serve as the foundation and aid the design and implementation of ICC.

ICC opens the door to more powerful and complex suggestions that could greatly increase productivity and code quality. New research opportunities could explore what could be ICC suggestion candidates, better-fitting searching algorithms, or ICC in other paradigms. Advancements in the field of (direct) code completion and AI can also be applied to ICC to provide richer and more relevant suggestions. It also remains to be seen which theoretically possible ICC kinds are practically useful. Early evaluation using the *MSR 2018* [11] dataset returned promising results [6], where a considerable number of equivalent sequences of direct code completion and other interaction events can be replaced by a single ICC. Our current focus is on developing an implementation of ICC so that more accurate evaluation can be done in future studies.

Declaration on Generative AI

The authors have not employed any Generative AI tools to create, change or rephrase the content of this document.

References

[1] J. L. Peterson, Computer Programs for Detecting and Correcting Spelling Errors, Communications of the ACM 23 (1980) 676–687. doi:10.1145/359038.359041.

[2] M. Asaduzzaman, C. K. Roy, K. A. Schneider, D. Hou, CSCC: Simple, Efficient, Context Sensitive Code Completion, in: Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2014, pp. 71–80. doi:10.1109/ICSME.2014.29.

[3] S. Janssens, V. Zaytsev, Go with the Flow: Software Engineers and Distractions, in: T. Kühn, V. Sousa, S. Abrahão, T. C. Lethbridge, E. Renaux, B. Selić (Eds.), MoDELS'22 Companion Proceedings: Sixth International Workshop on Human Factors in Modeling / Modeling of Human Factors (HuFaMo), 2022, pp. 934–938. doi:10.1145/3550356.3559101.

[4] L. L. Nunes da Silva Jr., T. Nazareth de Oliveira, A. Plastino, L. G. P. Murta, Vertical Code Completion: Going Beyond the Current Ctrl+Space, in: Proceedings of the Sixth Brazilian Symposium on Software Components (SBCARS): Architectures and Reuse, IEEE CS, 2012, pp. 81–90. doi:10.1109/SBCARS.2012.22.

[5] Y. Y. Lee, S. Harwell, S. Khurshid, D. Marinov, Temporal Code Completion and Navigation, in: Proceedings of the 35th International Conference on Software Engineering (ICSE), IEEE CS, 2013, pp. 1181–1184. doi:10.1109/ICSE.2013.6606673.

[6] Nhat, V. Zaytsev, CoCoCoLa: Code Completion Control Language, in: Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE), ACM, 2025, p. 1–12. doi:10.1145/3742876.3742883.

[7] Microsoft, Snippets in Visual Studio Code, 2025. URL: https://code.visualstudio.com/docs/editing/userdefinedsnippets.

[8] Microsoft, IntelliSense in Visual Studio, 2025. URL: https://learn.microsoft.com/en-us/visualstudio/ide/using-intellisense.

[9] IntelliJ, Live templates | IntelliJ IDEA Documentaion, 2025. URL: https://www.jetbrains.com/help/idea/using-live-templates.html.

[10] IntelliJ, Postfix code completion | IntelliJ IDEA Documentaion, 2025. URL: https://www.jetbrains.com/help/idea/postfix-code-completion.html.

[11] International Conference on Mining Software Repositories (MSR), MSR 2018 — Mining Challenge, 2018. URL: https://2018.msrconf.org/track/msr-2018-Mining-Challenge.