CGKB: Grafo Heterogêneo para Consultas Semanticamente Tipadas sobre Código-Fonte*

Jáder Louis de Souza Gonçalves[1], **Carolina Yukari Veludo Watanabe**[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Rondônia (UNIR)
Porto Velho – RO – Brasil

jaderlouis@proton.me, carolina@unir.br

Abstract. *Source-code analysis often requires queries that combine control-flow structure and data dependencies, such as finding variables defined inside loops and not reused later. Existing graph representations either flatten these distinctions or expose code through internal query models. This work proposes and evaluates CGKB (Code Graph Knowledge Base), a heterogeneous graph model for Python code with an explicit node/edge schema. CGKB integrates control-flow and data-dependency relations to support semantically typed queries over source code. We build CGKB for 114,998 APPS solutions and run five representative queries over 4,998 in-memory graphs. Results show 98.1% construction coverage and median query latency below 10 μs in the in-memory reference implementation.*

Resumo. *A análise de código-fonte frequentemente exige consultas que combinam estrutura de controle e dependências de dados, como encontrar variáveis definidas dentro de* loops *e não reutilizadas posteriormente. Representações atuais em grafo ocultam essas distinções ou expõem código por meio de modelos internos de consulta. Este trabalho propõe e avalia o CGKB (*Code Graph Knowledge Base*), grafo heterogêneo para código Python com esquema explícito de nós e arestas. O CGKB integra relações de fluxo de controle e dependência de dados para viabilizar consultas semanticamente tipadas sobre código-fonte. O modelo foi construído para 114.998 soluções APPS e avaliado por cinco consultas representativas sobre 4.998 grafos em memória. Os resultados indicam 98,1% de cobertura de construção e latência mediana inferior a 10 μs na implementação de referência em memória.*

1. Introdução

A escala dos repositórios de software contemporâneos torna o código-fonte um ativo de dados que demanda gestão, consulta e análise estruturada. Aplicações como busca semântica, detecção de vulnerabilidades, classificação de complexidade e integração com *pipelines* modernos de análise de código dependem de representações que tornem explícitas as relações de controle e dados presentes no código. Por exemplo, identificar variáveis definidas dentro de *loops* e não reutilizadas exige distinguir estruturas de controle, variáveis e arestas de definição/uso. Em uma representação homogênea, essa pergunta tende a

*Este trabalho utilizou ferramentas de inteligência artificial generativa para auxílio na revisão linguística e estruturação de texto. Os autores assumem total responsabilidade pelo conteúdo e pela verificação bibliográfica, conforme o Código de Conduta da Sociedade Brasileira de Computação (SBC).

virar busca textual ou travessia *ad hoc*; em um modelo tipado, ela se torna uma operação baseada em relações explícitas.

Três classes de abordagens dominam o estado da arte, cada uma com uma limitação concreta para uso como modelo de dados para consultas semanticamente tipadas. **(L1) Representações clássicas como grafo**: grafos de fluxo de controle (CFGs) [Allen 1970], grafos de dependência de programa (PDGs) [Ferrante et al. 1987] e o *Code Property Graph* (CPG) [Yamaguchi et al. 2014] adotam metaesquemas menos especializados para as consultas avaliadas neste trabalho e foram projetadas sobretudo para C/C++/Java, com suporte limitado a Python e sem formalização como rede de informação heterogênea. **(L2) Abordagens neurais** [Guo et al. 2021, Cummins et al. 2021] tratam o grafo como artefato intermediário para aprendizado, não como base de conhecimento persistível para exploração estruturada. **(L3) Ferramentas de consulta sobre código**, como o CodeQL [Avgustinov et al. 2016] (adotado pelo GitHub para detecção de vulnerabilidades em larga escala [Shen et al. 2025]) e o CodexGraph [Liu et al. 2025], oferecem linguagens de consulta sobre representações internas de código, mas não publicam esquema tipado nem avaliação por tipo de operação. Sem um esquema heterogêneo formal, torna-se difícil tratar código como dado tipado para consulta em bancos de grafos e para propagação por tipo em Redes Neurais em Grafos heterogêneas (GNNs, *Heterogeneous Graph Neural Networks* [Hu et al. 2020]).

Este trabalho propõe e avalia o CGKB (*Code Graph Knowledge Base*), um esquema em *property graph* formalizado como *Heterogeneous Information Network* (HIN) [Sun and Han 2012, Sun et al. 2022] para código Python, que integra fluxo de controle (CFG) e dependências de dados (DDG) sob esquema explícito de 13 tipos de nós e 10 tipos de arestas. O escopo é avaliar se o esquema é formal, rico e explorável por consultas; não se reivindica ganho em tarefas finais como detecção de vulnerabilidades ou classificação. Neste trabalho, o foco não está em melhorar tarefas *downstream* de engenharia de software, mas em investigar código-fonte como dado consultável por tipos semânticos. As contribuições são: **(C1)** um esquema heterogêneo formal (HIN) com 13 tipos de nós e 10 de arestas no mesmo meta-esquema, distinguindo-se do CPG (cuja taxonomia varia por implementação) e mapeável para SGBDs de grafos e GNNs; **(C2)** avaliação estrutural e de escalabilidade em larga escala sobre 114.998 programas do APPS [Hendrycks et al. 2021]; **(C3)** ablação evidenciando a complementaridade de CFG e DDG; e **(C4)** avaliação experimental com cinco consultas semanticamente tipadas sobre 4.998 grafos em memória, mostrando que o esquema expressa consultas difíceis de formular em representações homogêneas, com latência mediana inferior a 10 μs.

2. Trabalhos Relacionados

Representações clássicas e neurais. CFG [Allen 1970] e PDG [Ferrante et al. 1987] adotam esquemas homogêneos (1–3 tipos de nós) e Weiser [Weiser 1984] formalizou o *program slicing* sobre dependências de dados. O CPG [Yamaguchi et al. 2014] unifica árvore de sintaxe abstrata (AST, *Abstract Syntax Tree*), CFG e PDG com exportação a Neo4j, mas sua taxonomia varia por implementação e não organiza os elementos no mesmo meta-esquema heterogêneo avaliado neste trabalho. Song et al. [Song et al. 2024] propuseram o HACG, grafo heterogêneo abstrato com tipos de aresta para fluxo e dados, sem formalização de esquema heterogêneo nem avaliação de consultas. Zhang et al. [Zhang et al. 2024] aplicaram *transformers* heterogêneos sobre CPGs para

detecção de vulnerabilidades, demonstrando o ganho da heterogeneidade, sem definir esquema formal independente. Abordagens neurais [Guo et al. 2021, Cummins et al. 2021] usam o grafo como entrada de modelos, não como modelo de dados persistível.

Consultas e ontologias para código-fonte. Yamaguchi et al. [Yamaguchi et al. 2012] demonstraram extrapolação de vulnerabilidades via travessias sobre ASTs. O CodeQL [Avgustinov et al. 2016] é a ferramenta de referência para consultas declarativas, com larga avaliação em projetos embarcados [Shen et al. 2025], porém opera sobre representação relacional interna sem esquema heterogêneo formal publicado e sem avaliação por tipo de operação. Liu et al. [Liu et al. 2025] propuseram o CodexGraph, que persiste código em bancos de grafos com esquema tipado, contudo limitado a três tipos de nós e sem formalização como rede heterogênea. No eixo de *ontologias* para código, Atzeni e Atzori [Atzeni and Atzori 2017] propuseram a CodeOntology, ontologia OWL para representação RDF de código Java, com foco em interoperabilidade semântica, mas sem avaliação de desempenho de consulta sobre o esquema. Na comunidade de bancos de dados, trabalhos recentes do SBBD [Vidal et al. 2024, Angonese and Galante 2024] exploraram grafos de conhecimento e *embeddings* heterogêneos, mas nenhum aplicou tais técnicas a código-fonte com avaliação de operações de consulta.

3. Modelo Proposto: CGKB

O CGKB é um grafo heterogêneo tipado [Sun et al. 2022], construído por análise estática determinística sobre a AST de programas Python, integrando fluxo de controle (CFG) [Allen 1970] e dependências de dados (DDG) [Ferrante et al. 1987] sob um esquema tipado único. A construção ocorre em duas fases. Na *Fase 1* (CFG), cada estrutura de controle (sequência, decisão, repetição) gera um nó com tipo correspondente a um dos 10 tipos CFG e arestas `null`, `true`, `false`, `body`, `exit` ou `loop_back` conectando-o aos sucessores. Na *Fase 2* (DDG), cada definição de variável adiciona um nó `var` ligado por aresta `def`, e cada uso alcançável da variável recebe aresta `use`. O resultado é um grafo tipado $G = (V, E, \tau_v, \tau_e)$ com $\tau_v : V \to T_V$, $|T_V| = 13$, e $\tau_e : E \to T_E$, $|T_E| = 10$. Uma HIN é um grafo cujos nós e arestas pertencem a múltiplos tipos com semântica distinta [Sun and Han 2012]; no CGKB, essa heterogeneidade aparece no próprio esquema de nós e arestas, sem depender de rótulos textuais externos. A Figura 1 ilustra como um trecho curto gera nós de controle, variáveis e arestas `def`/`use` no mesmo esquema.

O metaesquema $\mathcal{M} = (T_V, T_E, \phi)$ define quais pares de tipos cada aresta admite. Em um SGBD de grafos, T_V e T_E são mapeáveis para rótulos de nós e tipos de relacionamentos, enquanto atributos viram propriedades; ϕ atua como restrição de integridade e habilita otimização de consultas por tipo. As principais restrições são: arestas `null` conectam apenas pares de nós CFG; ramos `true` e `false` partem de nós condicionais (`ci`); arestas `body`, `exit` e `loop_back` envolvem nós de *loop* (`cl`); `def`/`use` operam sobre nós `var`; e `call`/`return` ligam chamadas a corpos de função. A Tabela 1 apresenta o esquema utilizado. O nó `merge` explicita convergências de fluxo como nó tipado, preservando informação estrutural necessária para que GNNs heterogêneas propaguem mensagens por tipo de relação [Hu et al. 2020]. O modelo é *determinístico* (travessia de AST e *def-use* por escopo léxico), apresentou alta cobertura empírica em APPS (98,1%, Sec. 5) e suas duas camadas são *complementares*: nem CFG nem DDG sozinha preserva a informação capturada pela combinação, conforme verificado por ablação (Sec. 5).

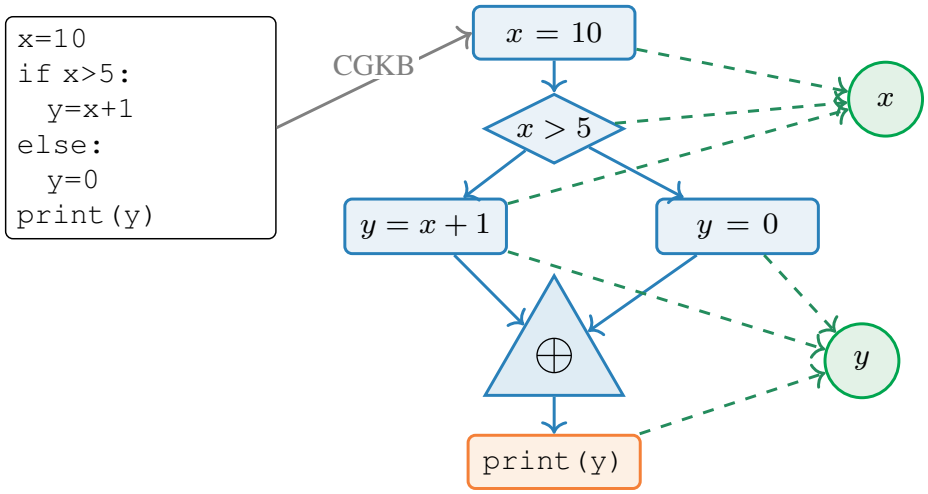

Figura 1. Mapeamento compacto: retângulos/losango azuis são nós CFG; círculos verdes são variáveis DDG; arestas azuis representam fluxo de controle; arestas verdes tracejadas representam relações `def`/`use`; $\oplus$ denota o nó `merge`, onde os ramos do `if`/`else` convergem.

4. Consultas sobre o Grafo CGKB

Consultas semanticamente tipadas sobre código-fonte exigem distinções explícitas entre controle, dados e tipos semânticos. Para bancos de dados, a utilidade do modelo depende de o grafo produzido admitir esse tipo de consulta com desempenho previsível. Definem-se quatro classes de consultas sobre G: (i) *filtro por tipo* $\sigma_t(G) = \{v \mid \tau_v(v) = t\}$, $O(|V|)$, análoga a σ em álgebra relacional; (ii) *travessia tipada*, restrita a tipos de aresta e nó, $O(|V| + |E|)$, análoga a junção tipada; (iii) *busca de caminhos* entre nós tipados, $O(|V| + |E|)$ com busca em profundidade limitada; (iv) *agregação por tipo*, análoga ao operador γ. A Tabela 2 define cinco consultas representativas (Q1–Q5) cobrindo as quatro classes, ordenadas por complexidade crescente, correspondendo a necessidades práticas de análise (filtragem, identificação de variáveis em *loops*, *backward slicing* [Weiser 1984], propagação parâmetro→retorno e detecção de variáveis não utilizadas). Essas consultas representam cenários difíceis de expressar em modelos homogêneos sem distinção explícita entre controle e dados.

A Tabela 3 compara essas consultas no escopo dos esquemas avaliados. A heterogeneidade é determinante: Q2–Q4 requerem distinção entre tipos de nó e aresta. CFG puro pode filtrar nós de controle, mas não representa variáveis/DDG; assim, Q2, Q3 e Q5 dependem de heurísticas externas. Texto/regex aproxima Q1 e Q5, mas sem garantia semântica. O CPG suporta Q1, porém atende Q2–Q5 apenas parcialmente, pois não organiza elementos como `cl`, `var` e `rv` no mesmo meta-esquema HIN/*property graph* usado pelo CGKB.

5. Avaliação Experimental

Aparato. Utilizou-se o APPS [Hendrycks et al. 2021], com 117.232 soluções Python de 9–30 LOC em três dificuldades. A avaliação mede se o esquema captura tipos distintos, se CFG/DDG são complementares e se consultas práticas têm baixa latência na implementação de referência em memória.

As métricas são: cobertura de construção; tamanho do grafo ($|V|$, $|E|$); hete-

Tabela 1. Esquema do modelo CGKB: 13 tipos de nós e 10 tipos de arestas.

Nós (13 tipos)			Arestas (10 tipos)		
Cat.	**Tipo**	**Descrição**	**Cat.**	**Tipo**	**Semântica**
DDG	`var`	Variável	CFG	`null`	Fluxo sequencial
	`cv`	Atribuição		`true`	Ramo verdadeiro
	`dp`	Parâmetro		`false`	Ramo falso
CFG	`start`	Início		`body`	Entrada de *loop*
	`end`	Final		`exit`	Saída de *loop*
	`ci`	Condicional		`loop_back`	Retorno ao *loop*
	`cl`	*Loop*		`call`	Invocação
	`merge`	Convergência		`return`	Retorno
	`func_body`	Corpo de função	DDG	`def`	Define variável
	`rv`	Retorno		`use`	Consome variável
	`call`	Chamada			
I/O	`print`	Saída			
	`imp`	Importação			

Tabela 2. Cinco consultas representativas sobre o modelo CGKB.

ID	Nome	Descrição	Classe
Q1	Filtro por tipo	Todos os nós `var` do grafo	Filtro
Q2	Variáveis em *loops*	`var` com `def`/`use` em `cl`	Travessia
Q3	*Backward slice*	*Slice* transitivo via DDG	Travessia
Q4	Parâmetro→retorno	Caminhos de `dp` a `rv`	Caminhos
Q5	Variáveis não utilizadas	Nós `var` com `def` mas sem `use`	Agregação

rogeneidade por entropia normalizada H_{norm}, que mede diversidade estrutural de tipos; riqueza semântica por SRI, que mede a presença relativa de dependências de dados; e latência mediana de construção e consulta. A implementação de referência usa Python 3.12 com `dict`/`set` como *backend* em memória, preservando o esquema em *property graph* para isolar o custo do esquema e das consultas sem *overhead* de SGBD. Os tempos não medem desempenho em Neo4j; persistência Cypher é direção futura. As consultas Q1–Q5 foram executadas sobre 4.998 grafos estratificados.

Cobertura e estrutura. O CGKB produziu grafos para 114.998 das 117.232 soluções (98,1%; falhas devidas a erros de sintaxe do código original). A Tabela 4 resume o tamanho dos grafos e duas evidências de que o modelo não corresponde apenas a uma expansão estrutural de CFG: $H_{norm} = 0{,}800$ indica distribuição heterogênea de tipos, e SRI$= 0{,}572$ mostra presença substantiva de dependências de dados.

Ablação e escalabilidade. A ablação confirma a complementaridade das camadas **(C3)**: remover DDG mantém todos os nós mas elimina 46,9% das arestas; remover CFG elimina 36,7% dos nós e 53,1% das arestas. A escalabilidade é quase-linear, com expoente 1,09 ($R^2 = 0{,}845$) e latência mediana de construção de 0,72 ms, indicando que o custo adicional da tipagem permanece baixo **(C2)**.

Tabela 3. Expressividade comparativa: quais consultas cada representação suporta. ● = nativa, ○ = parcial/aproximada, – = não suportada no escopo do esquema avaliado.

Representação	Q1	Q2	Q3	Q4	Q5
CFG-only	○	–	–	–	–
Texto/regex	○	–	–	–	○
CPG	●	○	○	○	○
CGKB	●	●	●	●	●

Tabela 4. Caracterização estrutural dos grafos CGKB por dificuldade.

Métrica	Intro.	Inter.	Comp.	Geral
$\|V\|$ (nós)	15,9	34,8	46,4	26,3
$\|E\|$ (arestas)	21,8	57,3	77,4	41,2
H_{norm} (nós)	0,816	0,784	0,747	0,800
SRI	0,541	0,582	0,606	0,572

Consultas. A Tabela 5 reporta Q1–Q5 sobre 4.998 grafos. Na implementação de referência em memória, todas têm mediana inferior a 10 μs, incluindo consultas que combinam controle e dados, como variáveis em *loops* e *backward slicing*. Isso materializa **(C4)**: o esquema permite formular consultas úteis sem recorrer a busca textual ou travessias dependentes de rótulos *ad hoc*.

Tabela 5. Desempenho das consultas representativas sobre 4.998 grafos CGKB. Tempos medianos em microssegundos (μs).

ID	Consulta	Med.	|Res.|	Sel.
Q1	Filtro por tipo	1,2	9,2	34,9%
Q2	Variáveis em *loops*	8,1	4,3	36,4%
Q3	*Backward slice*	2,1	2,9	14,8%
Q4	Param.→retorno	9,2	8,9	35,4%
Q5	Var. não utilizadas	2,7	0,7	6,9%

Comparação com o estado da arte. O diferencial do CGKB é expressar consultas semanticamente tipadas: CFG/PDG exigem informação adicional para Q2–Q5, enquanto o CGKB combina esquema tipado (13 nós, 10 arestas), CFG+DDG e Python nativo num único meta-esquema.

6. Considerações Finais

O CGKB modela código Python como dado consultável em um grafo tipado, avaliado em 114.998 programas, com integração CFG/DDG, heterogeneidade estrutural e cinco consultas com mediana abaixo de 10 μs na implementação em memória. As limitações incluem o *backend* em memória, a ausência de avaliação em SGBDs de grafos e *downstream*, e o uso de código competitivo. Como trabalho futuro, Neo4j/Cypher, código de produção e GNNs heterogêneas podem ampliar as consultas semanticamente tipadas sobre código-fonte.

Referências

Allen, F. E. (1970). Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19.

Angonese, S. F. and Galante, R. (2024). Composition of heterogeneous node embeddings – unlocking the power of heterogeneous graph representation. In *Simpósio Brasileiro de Banco de Dados (SBBD)*, pages 626–638. SBC.

Atzeni, M. and Atzori, M. (2017). CodeOntology: RDF-ization of source code. In *International Semantic Web Conference (ISWC)*, volume 10588 of *LNCS*, pages 20–28. Springer.

Avgustinov, P., de Moor, O., Jones, M. P., and Schäfer, M. (2016). QL: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP)*, volume 56 of *LIPIcs*, pages 2:1–2:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T., O'Boyle, M. F. P., and Leather, H. (2021). ProGraML: A graph-based program representation for data flow analysis and compiler optimizations. In *International Conference on Machine Learning (ICML)*, volume 139 of *Proceedings of Machine Learning Research*, pages 2244–2253. PMLR.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., and Zhou, M. (2021). GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations (ICLR)*. OpenReview.net.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. (2021). Measuring coding challenge competence with APPS. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*. Curran Associates.

Hu, Z., Dong, Y., Wang, K., and Sun, Y. (2020). Heterogeneous graph transformer. In *The Web Conference (WWW)*, pages 2704–2710. ACM.

Liu, X., Lan, B., Hu, Z., Liu, Y., Zhang, Z., Wang, F., Shieh, M., and Zhou, W. (2025). CodexGraph: Bridging large language models and code repositories via code graph databases. In *Proceedings of the 2025 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 142–160. Association for Computational Linguistics.

Shen, M., Pillai, A. A., Yuan, B. A., Davis, J. C., and Machiry, A. (2025). Finding 709 defects in 258 projects: An experience report on applying CodeQL to open-source embedded software (experience paper). *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1077–1100.

Song, S., Zhang, M., Li, S., and Liu, H. (2024). Learning heterogeneous abstract code graph representations for program comprehension. In *31st Asia-Pacific Software Engineering Conference (APSEC)*, pages 241–250. IEEE.

Sun, Y. and Han, J. (2012). Mining heterogeneous information networks: A structural analysis approach. *ACM SIGKDD Explorations Newsletter*, 14(2):20–28.

Sun, Y., Han, J., Yan, X., Yu, P. S., and Wu, T. (2022). Heterogeneous information networks: the past, the present, and the future. *Proceedings of the VLDB Endowment (PVLDB)*, 15(12):3807–3811.

Vidal, V. M. P., Freitas, R., Arruda, N., Casanova, M. A., and Renso, C. (2024). A data design pattern for building and exploring semantic views of enterprise knowledge graphs. In *Simpósio Brasileiro de Banco de Dados (SBBD)*, pages 1–13. SBC.

Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357.

Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. In *IEEE Symposium on Security and Privacy (S&P)*, pages 590–604. IEEE.

Yamaguchi, F., Lottmann, M., and Rieck, K. (2012). Generalized vulnerability extrapolation using abstract syntax trees. In *Annual Computer Security Applications Conference (ACSAC)*, pages 359–368. ACM.

Zhang, T., Xu, R., Zhang, J., Liu, Y., Chen, X., Yin, J., and Zheng, X. (2024). DSHGT: Dual-supervisors heterogeneous graph transformer – a pioneer study of using heterogeneous graph learning for detecting software vulnerabilities. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 33(8):1–31.