Um Arcabouço para o Ensino de Análises Estáticas de Programas

Henrique Fürst Scheid[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

hfscheid@gmail.com

Abstract. *While the world becomes increasingly reliant on software and its underlying infrastructure, understanding and improving codebases becomes ever more crucial. Programs may be analyzed before or during runtime. When done prior, it is called Static Program Analysis. While useful, the set of tools and theories involved in this analysis require knowledge that may not be as readily available as in most mainstream domains of Computer Science. This lack of accessibility contradicts the field's modern relevance. Thus, the existing discipline* ***DCC888 - Static Program Analysis*** *offered by Universidade Federal de Minas Gerais stands out as one of the few available courses that extensively cover the subject. This project expands the discipline's resources with a set of programming activities that aim to build experience alongside theoretical knowledge.*

Resumo. *À medida que o mundo depende cada vez mais de software e sua infraestrutura subjacente o entendimento e a melhoria de bases de código se torna mais crucial. Programas podem ser analisados antes ou durante sua execução. Quando feita antes, trata-se da Análise Estática de Programas. Apesar de útil, o conjunto de ferramentas e teorias envolvidos nessa análise requer conhecimento nem sempre facilmente disponível como em outras áreas dominantes da ciência da computação. Tal falta de acessibilidade contradiz a relevância moderna do ramo. Dessa forma, a disciplina* ***DCC888 - Static Program Analysis*** *se destaca por ser um dos poucos cursos disponíveis que cobre o tema extensivamente. Este projeto expande os recursos da disciplina com um conjunto de atividades de programação que visam agregar experiência junto ao aprendizado teórico.*

1. Introdução

1.1. Caracterização do problema

O Departamento de Ciência da Computação da UFMG oferece a disciplina optativa **DCC888 - Static Program Analysis**. A ementa e demais informações da disciplina estão disponibilizadas no site da disciplina [Pereira 2024], incluindo as apresentações de todas as aulas. De acordo com a aula introdutória [Pereira 2021b], o objetivo do curso é apresentar ao discente como alterar um programa automaticamente de forma a manter sua semântica e melhorar sua performance de acordo com alguma métrica. A disciplina se organiza em 24 aulas individualmente acompanhadas de listas de exercícios teóricos, apresentação de slides e bibliografia. Historicamente, além das listas teóricas, a disciplina utiliza provas e trabalhos práticos como recursos avaliativos.

A Análise Estática de Programas envolve um conjunto diverso de conceitos oriundos da ciência da computação, da matemática e até da linguística (como gramáticas lives de contexto). Sua história e conteúdo são cobertos em sala de aula e reforçados pelas atividades que as acompanham. Apenas os trabalhos práticos, no entanto, representam a oportunidade de se implementarem as técnicas estudadas. A disparidade entre os reforços teórico e prático pode ocasionar em grandes barreiras de dificuldade durante a execução dos trabalhos. Trata-se de um cenário pouco ideal, tendo em vista sua natureza avaliativa. Além disso, espera-se que estudantes concluam a disciplina com novas habilidades no tocante à análise de programas.

1.2. Motivação

É de interesse dos discentes e da instituição de ensino maximizar o aproveitamento do curso. As teorias e técnicas estudadas são utilizadas pela iniciativa privada e pela Academia, e chegam a ser mandatórias em alguns setores [Thomson 2021]. *Linters* e *Language Servers* são ubíquos em empresas de software e garantem corretude e convencionalidade de código entre programadores, e sua eficácia beneficia de informações oriundas da análise estática como *liveness analysis*. A diversificação de linguagens de programação modernas [GitHub, Inc 2022] resulta de esforços em implementação e melhoria de compiladores para novas linguagens. Se reconhece, pois, a demanda pelo profissional com conhecimento prático aliado ao conhecimento teórico em Análise Estática de Programas. Adicionalmente, trata-se de um domínio no qual a Universidade Federal de Minas Gerais pode exercer crescente influência.

A relevância da disciplina supera o uso de Análise Estática de Programas no desenvolvimento de software. Ainda de acordo com o material introdutório da disciplina [Pereira 2021b], a pesquisa em compiladores é relevante à medida que software é empregado de forma ubíqua pela sociedade. Apesar de seu ritmo de progresso não se assemelhar à evolução do hardware, a crescente escala do investimento e da infraestrutura que sustenta o uso de software aumenta a significância de qualquer otimização. Observa-se que grandes empresas lidam com tecnologia de compilação, como a Google [Google 2024b], NVIDIA [NVIDIA Corporation 2024] e Intel [Intel Corporation].

1.3. Objetivos

O objetivo primário do projeto é introduzir exercícios práticos que reforcem a capacidade dos alunos de implementar e utilizar os conhecimentos adquiridos em aula no contexto da programação. Além disso, a disposição das atividades junto às suas respectivas aulas visa auxiliar o acompanhamento da disciplina pelos alunos. Finalmente, as atividades apresentam mais um recurso para avaliação e distribuição de pontos durante o semestre.

2. Trabalhos relacionados

Foi possível encontrar cursos de Análise Estática de Programas oferecidos por algumas universidades além da UFMG:

1. Universidade de Aarhus, Dinamarca
2. Universidade de Stuttgart, Alemanha
3. Universidade de Tecnologia de Delft, Holanda

A disciplina **Static Program Analysis** ofertada pela Universidade de Aarhus [Møller and Schwartzbach 2024] apresenta uma coleção de 12 exercícios de programação em apoio às aulas [Møller 2021], semelhante ao trabalho realizado neste projeto. Os exercícios consistem em completar a implementação de um compilador de uma linguagem de programação chamada TIP. Especificamente, a maioria dos exercícios envolve completar funções do compilador da linguagem, que por sua vez são instrumentais para alguma análise estática. O projeto está implementado em Scala e está disponível no GitHub.

O Laboratório de Software da Universidade de Suttgart consiste em um grupo de pesquisa com esforços em análise estática de dinâmica de programas [Universidade de Stuttgart 2024]. A ementa da disciplina **Program Analysis** ministrada em 2023/2024 é disponível na internet [Universidade de Stuttgart 2023]. A disciplina conta com listas de exercícios teóricos e um projeto semestral. O projeto semestral aborda a implementação de uma *Taint Analysis* intraprocedural em programas JavaScript [Universidade de Stuttgart 2020]. A implementação da análise é feita sobre o **Google Closure Compiler** [Google 2024a].

A Universidade de Tecnologia de Delft oferta a disciplina **Compiler Construction** [Universidade de Técnica de Delft 2024], que inclui conceitos de Análise Estática de Programas. A disciplina conta com uma série de exercícios e um projeto envolvendo a implementação de um compilador utilizando a ferramenta **Spoofax** [Spoofax Team 2024].

A busca por material auxiliar fora do departamento oferece poucos resultados. A pesquisa por recursos para o aprendizado de Análise Estática de Programas na internet retorna artigos e entradas de blogs majoritariamente introdutórios. Exemplos excepcionais são a coletânea de recursos hospedados no GitHub pelo usuário **MattPD** [MattPD 2022], o blog introdutório à ferramenta **CodeQL** por Sylwia Budzynska [Budzynska 2024] e o blog sobre Análise Estática de Programas em Python pelo usuário Rahul hospedados no site Deepsource [Rahul 2020]. Todavia, nenhum dos recursos citados aborda o conjunto de teorias com a mesma granularidade que a disciplina, e não são constituídos de forma a explorar os conhecimentos previstos para um aluno do curso.

3. Estrutura do Trabalho

3.1. Base de código

Todo o código do projeto está hospedado em um repositório no GitHub [Scheid and Pereira 2024a] contendo um diretório para cada atividade. O fork contendo as atividades resolvidas encontra-se em [Scheid and Pereira 2024b]. Todas as atividades são implementadas em Python. Cada tarefa se encontra em um diretório dedicado contendo seu código-base e um conjunto de programas para teste.

O código-base consiste na implementação da linguagem-modelo utilizada ao longo das atividades, incluindo seu parser e seu interpretador, e no algoritmo de análise estática implementado pelo discente. Finalmente, cada atividade contém um *driver.py*

responsável por executar a tarefa, utilizado durante a avaliação automática (figura 1). As atividades possuem ordem definida. Assim como na evolução real do campo de estudo, técnicas desenvolvidas nos trabalhos mais tardios são construídas utilizando implementações anteriores. Dessa forma, algumas atividades reutilizam a solução de atividades anteriores. O benefício da abordagem é duplo: por um lado, observa-se na prática a utilidade de técnicas iniciais. Por outro, é possível elaborar atividades mais complexas sem revelar as soluções para os problemas iniciais. Enquanto este beneficia o docente e sua metodologia de avaliação, aquele impacta a perspectiva discente sobre a construção do conhecimento da área.

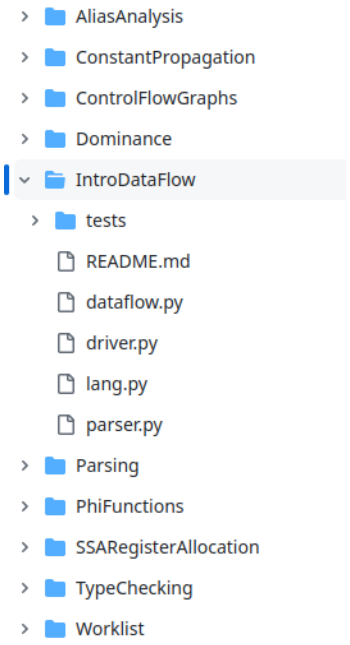


Figura 1. Estrutura de arquivos de uma atividade

4. Implementação

A linguagem é implementada nos arquivos **lang.py** utilizando uma hierarquia de classes que implementam o ambiente de variáveis e as instruções do programa. O comportamento das classes é documentado via *doctest*. A linguagem originalmente utiliza como memória uma pilha virtualmente ilimitada, referida pelo programa como seu ambiente (*environment*). A classe *Env* empilha as variáveis criadas durante a execução do programa, e pode ser inicializada já com variáveis criadas. Uma consulta ao ambiente resgata a última definição de uma variável, sem removê-la da pilha. Observa-se que, portanto, a definição de toda variável na linguagem-modelo é global. (Figura 2).

```
class Env:
    """
    A table that associates variables with values. The environment is
    implemented as a stack, so that previous bindings of a variable V remain
    available in the environment if V is overassigned.

    Example:
        >>> e = Env()
        >>> e.set("a", 2)
        >>> e.set("a", 3)
        >>> e.get("a")
        3

        >>> e = Env({"b": 5})
        >>> e.set("a", 2)
        >>> e.get("a") + e.get("b")
        7
    """

    def __init__(s, initial_args={}):
        s.env = deque()
        for var, value in initial_args.items():
            s.env.appendleft((var, value))

    def get(self, var): ...
            raise LookupError(f"Absent key {var}")

    def set(s, var, value): ...
        s.env.appendleft((var, value))

    def dump(s): ...
            print(f"{var}: {value}")
```

Figura 2. Implementação do ambiente de variáveis

A hierarquia de classes *Inst* (figuras 3 e 4) implementa as operações de adição, multiplicação, comparações (“menor que” e “maior ou igual a”) e salto condicional (“branch if true”). Cada objeto representando uma instrução armazena em si o índice numérico da instrução (sua posição no código), seus operandos e a referência para os objetos de suas instruções antecessoras e sucessoras. Programas nessa linguagem são representados como um encadeamento de tais objetos. O método *eval* de cada instrução recebe um ambiente (*Env*), consulta os valores de seus operandos e empilha uma nova variável com o resultado (figura 5).

```
class Inst(ABC):
    """
    The representation of instructions. All that an instruction has, that is
    common among all the instructions, is the next_inst attribute. This
    attribute determines the next instruction that will be fetched after this
    instruction runs. Also, every instruction has an index, which is always
    different. The index is incremented whenever a new instruction is created.
    """

    next_index = 0

    def __init__(self): ...
        Inst.next_index += 1

    def add_next(self, next_inst):
        self.nexts.append(next_inst)
        next_inst.preds.append(self)

    @classmethod
    @abstractmethod
    def definition(self):
        raise NotImplementedError

    @classmethod
    @abstractmethod
    def uses(self):
        raise NotImplementedError

    def get_next(self): ...
            return None
```

Figura 3. Definição genérica de uma instrução da linguagem-modelo

```
class Mul(BinOp):
    """
    Example:
        >>> a = Mul("a", "b0", "b1")
        >>> e = Env({"b0":2, "b1":3})
        >>> a.eval(e)
        >>> e.get("a")
        6
    """

    def eval(s, env):
        env.set(s.dst, env.get(s.src0) * env.get(s.src1))

    def get_opcode(self):
        return "*"
```

Figura 5. Método de computação para a instrução de multiplicação

4.1. Linguagem-modelo

O código-fonte da linguagem-modelo consiste em uma linha inicial definindo o seu ambiente, seguido pelo programa em si (figura 6). Cada linha deve corresponder a uma instrução. A linguagem se assemelha a um pequeno código de montagem turing-completo cujas possíveis operações são:

Figura 4. Hierarquia de classes das instruções da linguagem-modelo

- Adição: d = add x y $\rightarrow d = x + y$
- Multiplicação: d = mul x y $\rightarrow d = x * y$
- Maior ou igual: d = geq x y $\rightarrow d = x \geq y$
- Menor que: d = lth x y $\rightarrow d = x < y$
- Salto condicional: bt cond 10
- Funções Phi: x = phi x1 x2 x3 ... $\rightarrow x = \phi(x_1, x_2, x_3, ...)$ (apenas em atividades mais avançadas)

```
{"zero": 0, "one": 1, "three": 3, "iter": 9}
count0 = add zero three
pred0 = add zero one
fib0 = add zero one
count1 = phi count0 count2
pred1 = phi pred0 pred2
fib1 = phi fib0 fib2
aux = add zero fib1
fib2 = add pred1 fib1
pred2 = add zero aux
count2 = add count1 one
repeat = geq iter count2
bt repeat 3
end = add zero zero
```

Figura 6. Programa para calcular nono elemento da sequência de Fibonacci na linguagem-modelo

Instruções de salto se referem ao índice da instrução, que começa do 0 após a definição do ambiente. O *parsing* e a interpretação do programa são implementados no arquivo **parser.py**. O *parser* lê a lista de strings correspondente às linhas de código-fonte, extrai o **Env** da primeira linha e um subtipo de *Inst* para cada linha restante. As

instruções são encadeadas às demais, levando-se em consideração os saltos condicionais. A interpretação dos programas resultantes itera sobre o encadeamento das instruções, utilizando o método *eval* para atualizar o ambiente original a cada passo. O estado final do ambiente corresponde ao resultado do programa (figura 7). A implementação de programas utilizando a linguagem-modelo e de um *parser* correspondem aos dois primeiros laboratórios, cuja finalidade é familiarizar os discentes com a linguagem-modelo.

```
>>> import lang
>>> import parser
>>> with open('tests/safe_fib.txt', 'r') as f:
...     lines = f.readlines()
...
>>> env, prog = parser.file2cfg_and_env(lines)
>>> lang.interp(prog[0], env)
<lang.Env object at 0x71f8112f0950>
>>> env.env
deque([('end', 0), ('repeat', False), ('count2', 10), ('pred2', 21), ('fib2', 34), ('aux', 21), ('fib1', 21), ('pred1
', 13), ('count1', 9), ('repeat', True), ('count2', 9), ('pred2', 13), ('fib2', 21), ('aux', 13), ('fib1', 13), ('pre
d1', 8), ('count1', 8), ('repeat', True), ('count2', 8), ('pred2', 8), ('fib2', 13), ('aux', 8), ('fib1', 8), ('pred1
', 5), ('count1', 7), ('repeat', True), ('count2', 7), ('pred2', 5), ('fib2', 8), ('aux', 5), ('fib1', 5), ('pred1',
3), ('count1', 6), ('repeat', True), ('count2', 6), ('pred2', 3), ('fib2', 5), ('aux', 3), ('fib1', 3), ('pred1', 2),
 ('count1', 5), ('repeat', True), ('count2', 5), ('pred2', 2), ('fib2', 3), ('aux', 2), ('fib1', 2), ('pred1', 1), ('
count1', 4), ('repeat', True), ('count2', 4), ('pred2', 1), ('fib2', 2), ('aux', 1), ('fib1', 1), ('pred1', 1), ('cou
nt1', 3), ('fib0', 1), ('pred0', 1), ('count0', 3), ('iter', 9), ('three', 3), ('one', 1), ('zero', 0)])
>>> env.get('fib2')
34
```

Figura 7. Ambiente final após cálculo do nono elemento da sequência de Fibonacci

O arquivo **driver.py** consiste no ponto de entrada do programa. Seu papel é executar o código da atividade (incluindo a implementação do discente) contra um conjunto de casos de teste. Comumente os casos de teste são programas escritos em arquivos de texto. O resultado da execução do **driver.py** de cada atividade é determinístico a fim de ser comparado com um padrão pré-existente que corresponde ao resultado esperado. Dessa forma tem-se um mecanismo de avaliação automática. Os resultados comumente são impressos na saída padrão (*stdout*) (figura 8).

```python
import sys
from todo import *

if __name__ == "__main__":
    lines = sys.stdin.readlines()
    option = lines[0].strip()
    if option == "test_min":
        print(test_min(int(lines[1]), int(lines[2])))
    elif option == "test_min3":
        print(test_min3(int(lines[1]), int(lines[2]), int(lines[3])))
    elif option == "test_div":
        print(test_div(int(lines[1]), int(lines[2])))
    elif option == "fact":
        print(test_fact(int(lines[1])))
    else:
        print("Invalid option: {option}")
```

Figura 8. Ponto de entrada para a execução automática de uma atividade

O projeto conta com 10 atividades, cuja execução acompanha a ordem da disciplina:

1. *Control-Flow Graphs*
2. *Parsing*
3. *Data-Flow Analysis*
4. *Worklist Algorithms*
5. *Dominators*
6. *Semantics of Phi-Functions*
7. *Constant Propagation*
8. *Alias Analysis*
9. *Type Checking*
10. *Register Allocation*

Todos os exercícios foram criados em duas etapas. Primeiro se determinou e implementou por completo alguma técnica de Análise Estática de Programas de acordo com a ementa da disciplina. As implementações incluindo as soluções encontram-se em [Scheid and Pereira 2024b]. Um novo exercício utiliza o código-base do exercício anterior (comumente a definição da linguagem e seu parser) e introduz uma nova aplicação. Novas aplicações consistem em novos algoritmos ou estruturas de dados para a resolução de um novo problema de análise estática de programas. Os resultados obtidos pela nova aplicação são validados manualmente contra um conjunto de casos de teste adequado (e comumente criado para a atividade).

Finda a primeira etapa é escolhido algum trecho do programa que isole uma técnica ou algoritmo cujo reforço é julgado de especial interesse para o discente. O trecho é omitido e, se necessário, seu isolamento é garantido pela reestruturação do programa que o envolve. Dessa forma objetiva-se isolar a atuação do aluno a completar funções ou conjuntos de funções em uma classe, maximizando a clareza do exercício. A contribuição esperada do aluno é destacada pela documentação *doctest* presente no arquivo de atividade, que explica o que deve ser feito, qual o comportamento esperado e quais os recursos auxiliares disponíveis.

4.2. Control-Flow Graphs

A primeira atividade introduz a linguagem-modelo, e seu objetivo é familiarizar os participantes da disciplina com o código-base que será utilizado ao longo do curso. A tarefa consiste em escrever programas na linguagem-modelo diretamente em Python, utilizando objetos que representem as instruções e encadeando-os manualmente. O discente atua no arquivo **todo.py**, onde deve completar algumas funções incompletas. O resultado esperado é documentado dentro das próprias funções, e pode ser testado via *doctest* (figura 9).

```python
def test_div(m, n):
    """
    Stores in the variable 'answer' the integer division of 'm' and 'n'.

    Examples:
        >>> test_div(30, 4)
        7
        >>> test_div(4, 3)
        1
        >>> test_div(1, 3)
        0
    """
    # TODO: Implement this method
    return env.get("answer")
```

Figura 9. Lacuna para implementação do discente

A verificação automática da primeira atividade faz uma chamada às funções implementadas com valor de entrada fornecido pela entrada-padrão. A saída das funções é impressa na saída padrão e comparada com o valor esperado (figura 8).

4.3. Parsing

A segunda atividade demanda que os alunos implementem um *parser* de programas da linguagem-modelo que leia arquivos de texto e gere os seus respectivos programas encadeando os devidos objetos de instruções em Python conforme descrito em 4.1. A implementação do *parser* será reutilizada em outras atividades. A atividade conta com um conjunto de programas de teste. A verificação automática utiliza o *parser* implementado pelo discente para gerar os programas em python interpretá-los. O resultado dos programas é impresso na saída-padrão e comparado com o valor esperado (figura 10).

```python
import sys
import todo

from lang import interp

if __name__ == "__main__":
    lines = sys.stdin.readlines()
    env, program = todo.file2cfg_and_env(lines)
    final_env = interp(program[0], env)
    final_env.dump()
```

Figura 10. Ponto de entrada da segunda atividade

4.4. Introduction to Data-Flow Analysis

Durante a terceira atividade o discente exercita a Análise de Fluxo de Dados, ou Dataflow Analysis. Trata-se do conteúdo visto durante as primeiras 3 aulas da disciplina. Uma Análise de Fluxo de Dados consiste em extrair qual informação está disponível em cada ponto de um programa. É possível, por exemplo, fazer uma análise de quais variáveis estão “vivas” (ou seja, prestes a serem utilizadas) imediatamente antes e após cada instrução do programa. Uma análise de fluxo de dados representa a informação que

entra e a informação que sai de cada ponto em um programa. Fixada uma instrução em um programa, o fluxo de informações é descrito como uma operação sobre a informação de outros pontos do programa, de acordo com sua direção de propagação (figura 11). A análise de fluxo de dados pode ser mais ou menos conservadora. Por exemplo, para cada instrução é possível afirmar que não há nenhuma informação. Portanto, estabelecer equações que descrevem o fluxo de dados em um programa pode ser interpretado como uma restrição que limita inferiormente ou superiormente como a informação chega e sai de cada instrução.

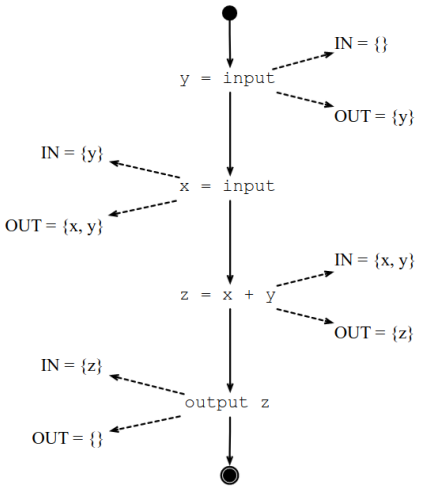


Figura 11. Exemplo do fluxo de dados entre instruções em um programa (retirado de [Pereira 2021a])

O código a ser implementado encontra-se em **dataflow.py**. Equações de fluxos de dados são representadas por uma hierarquia de classes *DataFlowEq*. As equações de fluxo de dados são resolvidas pelos métodos *eval*, comum a toda a hierarquia, e *eval_aux*, que inclui a lógica específica de cada tipo de equação própria a uma análise. A atividade consiste em completar os métodos *eval_aux* das classes de equações para *liveness analysis* (figuras 12 e 13). Além disso, o conjunto de equações para análise do programa deve ser gerado também por uma função implementada pelo aluno. A análise de *reaching definitions* está implementada como exemplo (figura 13).

```
class LivenessAnalysisIN_Eq(IN_Eq):
    def eval_aux(self, data_flow_env: dict[str, set]) -> None:
        """
        Produces the IN set of liveness analysis. The IN set is given by the
        equation below, considering the instruction `p) v = E`:

        IN[p] = uses(E) + (OUT[p] - {v})

        Example:
            >>> Inst.next_index = 0
            >>> i0 = Add('x', 'a', 'b')
            >>> df = LivenessAnalysisIN_Eq(i0)
            >>> sorted(df.eval_aux({'OUT_0': {'x'}}))
            ['a', 'b']
        """
        # TODO: implement this method
```

Figura 12. Espaço para implementação da análise de variáveis vivas

```
class ReachingDefs_IN_Eq(IN_Eq):
    """
    This concrete class implements the meet operation for reaching-definition
    analysis. The meet operation produces the IN set of a program point. This
    IN set is the union of the OUT set of the predecessors of this point.
    """

    def eval_aux(self, data_flow_env):
        """
        The evaluation of the meet operation over reaching definitions is the
        union of the OUT sets of the predecessors of the instruction.

        Example:
            >>> Inst.next_index = 0
            >>> i0 = Add('x', 'a', 'b')
            >>> i1 = Add('x', 'c', 'd')
            >>> i2 = Add('y', 'x', 'x')
            >>> i0.add_next(i2)
            >>> i1.add_next(i2)
            >>> df = ReachingDefs_IN_Eq(i2)
            >>> sorted(df.eval_aux({'OUT_0': {('x', 0)}, 'OUT_1': {('x', 1)}}))
            [('x', 0), ('x', 1)]
        """
        solution = set()
        for inst in self.inst.preds:
            solution = solution.union(data_flow_env[name_out(inst.ID)])
        return solution
```

Figura 13. Exemplo de implementação de equação de fluxo de dados

A verificação automática recebe um programa-teste e o analisa com a implementação do aluno. A informação da análise é utilizada para verificar se o programa utiliza alguma variável antes de sua definição, o que tornaria o programa incorreto (figura 14).

```
if __name__ == "__main__":
    """
    If you want to see the program, you can use the function call
    `print_instructions(program)`
        >>> l0 = '{"a": 1, "b": 3, "x": 42, "z": 0}'
        >>> l1 = 'bt a 2'
        >>> l2 = 'x = add a b'
        >>> l3 = 'x = add x z'
        >>> _, program = file2cfg_and_env([l0, l1, l2, l3])
        >>> print_instructions(program)
    """
    lang.Inst.next_index = 0
    lines = sys.stdin.readlines()
    env, program = parser.file2cfg_and_env(lines)
    equations = dataflow.liveness_constraint_gen(program)
    df_env = dataflow.abstract_interp(equations)
    init_in = df_env[dataflow.name_in(program[0].ID)]
    check_environment(env, init_in)
```

Figura 14. Verificação automática para a Introduction to Data-Flow Analysis

4.5. Worklist Algorithms

Ainda relacionado às Análises de Fluxo de Dados, a quarta atividade foca em como resolver o conjunto de equações gerado na atividade anterior. Equações de fluxo de dados interferem umas as outras, à medida que a informação de entrada de uma instrução é uma função da informação de saída de outra. Dessa forma, o resultado já computado de uma equação pode ser se tornar obsoleto após a computação do resultado de outra equação distante por uma cadeia de interferências. As equações devem ser repetidamente calculadas até que todos os resultados se estabilizem. A abordagem ingênua é caoticamente iterar por todas as equações e resolvê-las até que não se observem mais alterações na solução global. Durante a terceira atividade utiliza-se esse método, já implementado, para a solução dos sistemas de equações implementados pelos alunos (figura 15).

O estudante deverá implementar a função que resolve as equações de forma mais sofisticada que explore a relação de dependência entre restrições (figura 16). Em específico, espera-se que o discente implemente um grafo de dependências entre as instruções na função *build_dependence_graph*. O grafo será utilizado por um algoritmo de *worklist*, também implementado pelo discente, que deve manter de forma ordenada quais restrições podem ser avaliadas de forma a minimizar o número de demais restrições impactadas pelo seu resultado (figuras 17).

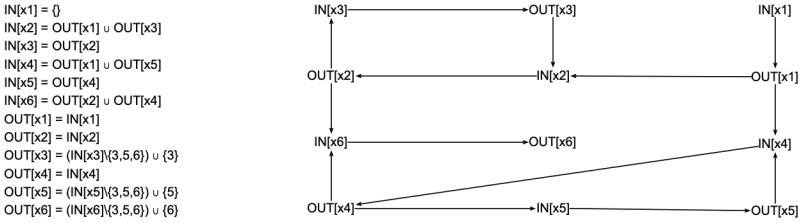


Figura 16. Relações de dependência entre equações de uma análise de fluxo de dados (retirado de [Scheid and Pereira 2024a])

```python
def abstract_interp(equations):
    """
    This function iterates on the equations, solving them in the order in which
    they appear. It returns an environment with the solution to the data-flow
    analysis.

    Example for reaching-definition analysis:
        >>> Inst.next_index = 0
        >>> i0 = Add('c', 'a', 'b')
        >>> i1 = Mul('d', 'c', 'a')
        >>> i0.add_next(i1)
        >>> eqs = reaching_defs_constraint_gen([i0, i1])
        >>> (sol, num_evals) = abstract_interp(eqs)
        >>> f"OUT_0: {sorted(sol['OUT_0'])}, Num Evals: {num_evals}"
        "OUT_0: [('c', 0)], Num Evals: 12"
    """
    from functools import reduce

    DataFlowEq.num_evals = 0
    env = {eq.name(): set() for eq in equations}
    changed = True
    while changed:
        changed = reduce(lambda acc, eq: eq.eval(env) or acc, equations, False)
    return (env, DataFlowEq.num_evals)
```

Figura 15. Solução de sistemas de equações de fluxo de dados por iterações caóticas

A classe ancestral de todas as equações de fluxo de dados mantém o rastro de todas as vezes que uma equação é computada (figura 18). Dessa forma é possível comparar o número de computações necessárias para a resolução do sistema de equações entre o algoritmo ingênuo e a solução implementada pelo aluno. A avaliação automática verifica se o resultado encontrado pelas soluções é semelhante, e se houve melhoria no número de iterações (figura 19).

```python
def eval(self, data_flow_env) -> bool:
    """
    This method implements the abstract evaluation of a data-flow equation.
    Notice that the actual semantics of this evaluation will be implemented
    by the `èval_aux` method, which is abstract.
    """
    DataFlowEq.num_evals += 1
    old_env = data_flow_env[self.name()]
    data_flow_env[self.name()] = self.eval_aux(data_flow_env)
    return True if data_flow_env[self.name()] != old_env else False
```

Figura 18. O número de iterações é armazenado pela superclasse das equações de fluxo de dados

```python
def abstract_interp_worklist(equations) -> tuple[Env, int]:
    """
    This function solves the system of equations using a worklist. Once an
    equation E is evaluated, and the evaluation changes the environment, only
    the dependencies of E are pushed onto the worklist.

    Example for reaching-definition analysis:
        >>> Inst.next_index = 0
        >>> i0 = Add('c', 'a', 'b')
        >>> i1 = Mul('d', 'c', 'a')
        >>> i0.add_next(i1)
        >>> eqs = reaching_defs_constraint_gen([i0, i1])
        >>> (sol, num_evals) = abstract_interp_worklist(eqs)
        >>> f"OUT_0: {sorted(sol['OUT_0'])}"
        "OUT_0: [('c', 0)]"
    """
    # TODO: implement this method
    from collections import defaultdict

    DataFlowEq.num_evals = 0
    env = defaultdict(list)
    return (env, DataFlowEq.num_evals)
```

```python
def build_dependence_graph(equations) -> dict[str, list[DataFlowEq]]:
    """
    This function builds the dependence graph of equations.

    Example:
        >>> Inst.next_index = 0
        >>> i0 = Add('c', 'a', 'b')
        >>> i1 = Mul('d', 'c', 'a')
        >>> i0.add_next(i1)
        >>> eqs = reaching_defs_constraint_gen([i0, i1])
        >>> deps = build_dependence_graph(eqs)
        >>> [eq.name() for eq in deps['IN_0']]
        ['OUT_0']
    """
    # TODO: implement this method
    dep_graph = {eq.name(): [] for eq in equations}
    return dep_graph
```

Figura 17. Funções incompletas para a solução de sistemas de equações inter-dependentes por worklists

```python
def chaotic_solver(program):
    equations = dataflow.reaching_defs_constraint_gen(program)
    return dataflow.abstract_interp(equations)


def worklist_solver(program):
    equations = dataflow.reaching_defs_constraint_gen(program)
    return dataflow.abstract_interp_worklist(equations)


if __name__ == "__main__":
    """
    This function reads a program, and solves reaching definition analysis
    for it, using either chaotic iterations or the worklist-based algorithm.
    """
    lang.Inst.next_index = 0
    lines = sys.stdin.readlines()
    env, program = parser.file2cfg_and_env(lines)
    (env_chaotic, n_chaotic) = chaotic_solver(program)
    (env_worklist, n_worklist) = worklist_solver(program)
    print(f"Are the environments the same? {env_chaotic == env_worklist}")
    print(f"Used less than {n_chaotic} iterations? {n_worklist <= n_chaotic}")
```

Figura 19. Avaliação automática para atividade de Worklists

4.6. Dominators

Diz-se que uma instrução A *domina* outra instrução B se todos os possíveis caminhos de execução de um programa necessariamente avaliam A antes de B. Determinar as relações de dominância entre instruções é útil para a análise de programas, pro exemplo, para se encontrarem laços ou realizar outras operações mais complexas. Nesta atividade os alunos devem implementar uma análise de fluxo de dados que encontre as relações de dominância entre as instruções de um programa. Deve-se implementar as equações *DominanceEq* de forma semelhante à terceira atividade 4.4 (figura 20). Além disso, deve-se implementar a geração e a solução do sistema de equações de dominância para um dado programa (figura 21).

```
class Dominance_Eq(DataFlowEq):
    """
    This concrete class implements the meet operation for the dominance
    analysis. The dominators of data-flow vertex v is the intersection of the
    dominators of the predecessors of v.
    """

    def eval_aux(self, env: dict[str, set[int]]) -> set[int]:
        """
        The evaluation of the meet operation for the dominance relation.
        Basically: D[n] = {n} U Intersection(D[p], for p in n.preds)

        Example:
            >>> Inst.next_index = 0
            >>> i0 = Add('x', 'a', 'b')
            >>> df = Dominance_Eq(i0)
            >>> sorted(df.eval_aux({}))
            [0]

            >>> Inst.next_index = 0
            >>> i0 = Add('x', 'a', 'b')
            >>> i1 = Add('y', 'x', 'a')
            >>> i2 = Add('y', 'x', 'b')
            >>> i3 = Add('z', 'x', 'y')
            >>> i0.add_next(i1)
            >>> i0.add_next(i2)
            >>> i1.add_next(i3)
            >>> i2.add_next(i3)
            >>> df = Dominance_Eq(i3)
            >>> sorted(df.eval_aux({'1': {0, 1}, '2': {0, 2}}))
            [0, 3]
        """
        # TODO: Implement this method.
        return set()
```

Figura 20. Lacuna para implementação da análise de dominância entre instruções

```
def abstract_interp(equations: list[Dominance_Eq]) -> dict[str, set[int]]: ...
    return env

def dominance_constraint_gen(insts: list[Inst]) -> list[Dominance_Eq]: ...
    return []
```

Figura 21. Funções para a geração e solução das equações de análise de dominância

A solução implementada é utilizada durante o cálculo de dominância para programas de teste durante a verificação automática. O resultado é impresso na saída padrão e comparado ao valor esperado (figura 22).

```
if __name__ == "__main__":
    """
    If you want to see the program, you can use the function call
    `print_instructions(program)`
        >>> l0 = '{"a": 1, "b": 3, "x": 42, "z": 0}'
        >>> l1 = 'bt a 2'
        >>> l2 = 'x = add a b'
        >>> l3 = 'x = add x z'
        >>> _, program = file2cfg_and_env([l0, l1, l2, l3])
        >>> print_instructions(program)
    """
    lang.Inst.next_index = 0
    lines = sys.stdin.readlines()
    env, program = parser.file2cfg_and_env(lines)
    equations = dataflow.dominance_constraint_gen(program)
    dom_tree = dataflow.abstract_interp(equations)
    for ID in sorted(dom_tree):
        print(f"D({ID}): {sorted(dom_tree[str(ID)])}")
```

Figura 22. Avaliação automática para atividade Dominators

4.7. Semantics of Phi-Functions

A sexta atividade está relacionada à representação de programas em formato SSA (*Static Single-Assignment*). Programas nesse formato contam com uma definição única para cada variável, de forma que seu valor é inalterado durante toda a sua vida. A atribuição de novos valores gera novas variáveis. O formato SSA possui uma importante implicação: todo uso de uma variável é dominado pela sua definição. Diferentemente de um programa convencional, onde uma variável pode ser definida em distintos ramos de execução e a seguir utilizada após sua convergência, programas em formato SSA devem primeiro definir uma nova variável que lide com a colisão de informação para depois utilizá-la. A resolução de tais colisões é feita por funções ϕ, que definem a nova variável de acordo com o último valor utilizado em tempo de execução (figuras 23, 24).

$$
\begin{array}{ll}
L_0: a = x + y & L_0: a_1 = x_0 + y_0 \\
\ _1: b = a - 1 & \ _1: b_1 = a_1 - 1 \\
\ _2: a = y + b & \ _2: a_2 = y_0 + b_1 \\
\ _3: b = 4 * x & \ _3: b_2 = 4 * x_0 \\
\ _4: a = a + b & \ _4: a_3 = a_2 + b_2
\end{array}
$$

Figura 23. Comparação entre um programa fora (esq.) e dentro (dir.) do formato de Static Single-Assignment (SSA) (retirado de [Pereira 2021d])

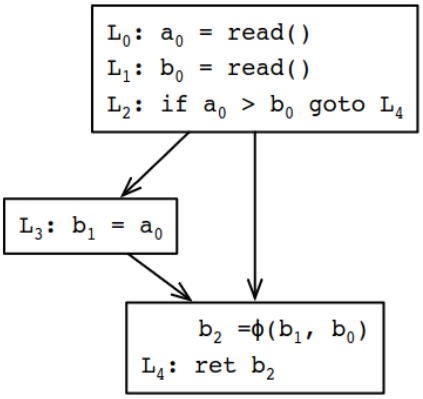


Figura 24. Exemplo de utilização de uma função ϕ (retirado de [Pereira 2021d])

Nessa atividade são introduzidas duas classes para a representação de funções ϕ na linguagem-modelo: *Phi* e *PhiBlock*, onde *PhiBlocks* implementam conjuntos de instruções ϕ executadas simultaneamente. A primeira classe já está implementada, cabendo ao discente implementar a criação e avaliação de *PhiBlocks* (figura 25). A implementação é feita diretamente sobre o arquivo contendo o código da linguagem. A avaliação automática conta com um conjunto de programas-teste já escritos em formato SSA, utilizando funções ϕ, que são interpretados e cujo resultado é impresso na saída padrão. Seu resultado é comparado a um valor previamente estabelecido (figura 26).

```
class PhiBlock(Inst):

    def __init__(self, phis, selector_IDs): ...
        super().__init__()

    def definition(self): ...
        return [phi.definition() for phi in self.phis]

    def uses(self): ...
        return sum([phi.uses() for phi in self.phis], [])

    def eval(self, env: Env, PC: int):
        # TODO: Read all the definitions
        # TODO: Assign all the uses:
        pass
```

Figura 25. Classe incompleta para avaliação de funções ϕ

```python
import sys
from programs import *

if __name__ == "__main__":
    lines = sys.stdin.readlines()
    option = lines[0].strip()
    if option == "test_min":
        print(test_min(int(lines[1]), int(lines[2])))
    elif option == "test_min3":
        print(test_min3(int(lines[1]), int(lines[2]), int(lines[3])))
    elif option == "test_div":
        print(test_div(int(lines[1]), int(lines[2])))
    elif option == "fact":
        print(test_fact(int(lines[1])))
    elif option == "fib":
        print(test_fib(int(lines[1])))
    elif option == "fib_swap_problem":
        print(test_fib_swap_problem(int(lines[1])))
    elif option == "test_fib_swap_problem_fixed_with_phi_blocks":
        print(test_fib_swap_problem_fixed_with_phi_blocks(int(lines[1])))
    else:
        print("Invalid option: {option}")
```

Figura 26. Avaliação automática para atividade de Phi Functions

4.8. Constant Propagation

Ainda trabalhando com programas em formato SSA, a atividade visa envolver o aluno na aplicação de técnicas da Análise Estática de Programas para a otimização de código. Recorrentemente encontram-se em programas operações redundantes sobre valores constantes. Uma variável C resultante da operação de duas variáveis constantes A e B pode ser previamente computada em tempo de compilação, reduzindo assim o custo computacional da execução do programa. Otimizações desse cunho utilizam a análise de *constant propagation* (propagação de constantes), onde o estado de uma variável pode evoluir de "indeterminado" para "constante" ou "inconstante" (figura 27).

Dois conceitos importantes são explorados na atividade além da implementação da análise em si. Uma análise tradicional (como as implementadas na terceira atividade) resulta em uma série de resultados representando a informação que entra e sai de cada instrução em um programa, sendo uma análise *densa*. O uso de programas em formato SSA, todavia, permite uma análise de propagação de constantes *esparsa*, onde informação é extraída diretamente da definição de variáveis ao invés das interações entre variáveis e instruções específicas. Em segundo lugar, o estado das variáveis durante a análise evolui em apenas um sentido, seguindo a estrutura de um reticulado (*lattice*) (figura 27). Tal estrutura é recorrentemente utilizada em análises sobre programas.

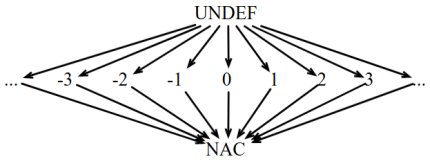


Figura 27. Reticulado representando os possíveis estados de uma variável durante análise de propagação de constantes

Nessa atividade o discente deve implementar a análise de propagação de constantes via métodos *eval_aux* de modo semelhante a atividades anteriores (figura 28). Observa-se que uma nova instrução, *Read*, foi adicionada à linguagem para introduzir variáveis de valor desconhecido antes da execução (portanto, valores "inconstantes"). A avaliação automática lê os programas de teste e o analisa utilizando a implementação do aluno. O resultado da análise é impresso na saída padrão, onde é comparado com um valor previamente estabelecido (figura 29).

```
class SparseConstantPropagationEq(DataFlowEq): ...
        return ''


class BranchEq(SparseConstantPropagationEq):
    def eval_aux(self, data_flow_env: Env):
        return

class BinOpEq(SparseConstantPropagationEq): ...
        pass

class ReadEq(SparseConstantPropagationEq): ...
        pass

class PhiEq(SparseConstantPropagationEq): ...
        pass
```

Figura 28. Classes de equações de fluxo de dados para análise de propagação de constantes

```
if __name__ == "__main__":
    lang.Inst.next_index = 0
    lines = sys.stdin.readlines()
    env, program = parser.file2cfg_and_env(lines)
    equations = dataflow.constant_prop_constraint_gen(program)
    result_env, _ = dataflow.abstract_interp(equations, env)
    print(result_env.to_dict())
```

Figura 29. Avaliação automática para a atividade de Constant Propagation

4.9. Alias Analysis

É possível implementar uma análise estática de quais possíveis endereços de memória são referenciados por um ponteiro. Trata-se de uma análise conservadora, visto que deve-se considerar todos os possíveis rumos de execução de um programa. A oitava atividade implementa uma análise estática para o estudo de ponteiros, conhecida como *pointer analysis* ou *alias analysis*. Especificamente, implementa-se uma *Andersen-style pointer analysis* baseada em [Andersen 1994], utilizando um conjunto específico de regras de propagação de informação (figura 30). No início da análise, todo ponteiro do programa possui um conjunto vazio de alvos. Em seguida as regras são iterativamente

aplicadas, de forma que cada ponteiro ganha novos alvos via alocação de memória ou associação com outros ponteiros.

Statement	Constraint name	Constraint
`a = &b`	base	P(a) ⊇ {b}
`a = b`	simple	P(a) ⊇ P(b)
`a = *b`	load	t ∈ P(b) ⇒ P(a) ⊇ P(t)
`*a = b`	store	t ∈ P(a) ⇒ P(t) ⊇ P(b)

Figura 30. Regras para propagação de informação de alvos entre ponteiros durante análise de Andersen (retirado de [Pereira 2021c])

A atividade primeiro expande a linguagem-modelo para incluir operações de ponteiros. Criou-se a classe *Storage* para simular memória dinamicamente alocável. Foram criadas em seguida as novas instruções:

- *Store*: $*ref = src$ armazena o valor de src no espaço de memória referido por ref.
- *Load*: $dst = *ref$ lê o valor armazenado no endereço referido por ref.
- *Alloca*: $ref = alloca$ reserva espaço de memória e atribui seu endereço a ref.
- *Move*: $v = movew$ copia o conteúdo de w para dentro de v.

```python
class Edge:
    """
    This class implements the edge of the points-to graph that is used to
    solve Andersen-style alias analysis.
    """

    def __init__(self, dst: str, src: str): ...
        self.dst = dst

    def eval(self, env: dict[str, set[str]]) -> bool:
        """
        Evaluating an edge such as dst -> src means copying every pointer in
        Alias(dst) into Alias(src). This function retuyrns True if the
        points-to set of dst changes after the evaluation.

        Example:
            >>> e = Edge('a', 'b')
            >>> env = {'a': {'ref_1'}, 'b': {'ref_0'}}
            >>> result = e.eval(env)
            >>> f"{result}: {sorted(env['a'])}"
            "True: ['ref_0', 'ref_1']"
        """
        # TODO: Implement this method.
        return False
```

Figura 31. Classe representando referenciação de um ponteiro a um alvo durante análise de Andersen

Crucial para a análise é o fato de que, à medida que um ponteiro recebe os valores de novos ponteiros, a cadeia de referências entre ponteiros de propaga e novas equações de fluxo de informação entre ponteiros devem ser contempladas. A análise se baseia pois na geração dinâmica de equações, além de sua computação. A análise termina

quando o conjunto de valores para cada ponteiro se estabiliza. A atividade consiste em implementar a análise através do arquivo **alias.py**. Utiliza-se uma nova estrutura de dados *Edge* representando a relação de referenciação de um *alias* a um alvo. Seu método *eval* desempenha um papel semelhante aos métodos homônimos de outras análises, e deve ser implementado (figura 31). Em seguida devem-se completar as funções *propagate_alias_info*, *evaluate_st_constraints* e *evaluate_ld_constraints* (figura 32). Finalmente, o estado inicial da análise deve ser criado através da função *init_env*, e a computação da análise deve ser coordenada pela função *abstract_interp*. A avaliação automática utiliza programas de teste e compara o resultado da análise implementada pelo discente com valores pré-estabelecidos (figura 33).

```python
def init_env(insts) -> dict[str, set[str]]: ...
    return None


def propagate_alias_info(edges, env: dict[str, set[str]]) -> bool: ...
    return None


def evaluate_st_constraints(insts, env: dict[str, set[str]]) -> list[Edge]: ...
    return None


def evaluate_ld_constraints(insts, env: dict[str, set[str]]) -> list[Edge]: ...
    return None


def abstract_interp(insts) -> dict[str, set[str]]: ...
    return None
```

Figura 32. Funções para a implementação da análise de ponteiros

```python
def check_pointers(program):
    """
    Runs Andersen-Style points-to analysis on the input program, and check the
    abstract state of six variable names: p0, p1, p2, ref_0, ref_1 and ref_2.
    This method is ony used to automatically grade the assignment.
    """
    abstract_env = alias.abstract_interp(program)
    if abstract_env:
        print(f"Alias(p0): {sorted(abstract_env.setdefault('p0', set()))}")
        print(f"Alias(p1): {sorted(abstract_env.setdefault('p1', set()))}")
        print(f"Alias(p2): {sorted(abstract_env.setdefault('p2', set()))}")
        print(f"Alias(ref0): {sorted(abstract_env.setdefault('ref_0', set()))}")
        print(f"Alias(ref1): {sorted(abstract_env.setdefault('ref_1', set()))}")
        print(f"Alias(ref2): {sorted(abstract_env.setdefault('ref_2', set()))}")
    else:
        print("The program has no memory allocation.")


if __name__ == "__main__":
    lines = sys.stdin.readlines()
    env, program = parser.file2cfg_and_env(lines)
    check_pointers(program)
```

Figura 33. Avaliação automática para a tarefa de Alias Analysis

4.10. Type Checking

Idealmente, um algoritmo apresenta três propriedades que garantem sua corretude:

1. **Progresso**: Se o estado do algoritmo é válido, o algoritmo executa o próximo passo
2. **Preservação**: Se o estado do algoritmo é válido e o algoritmo executa um passo, o estado resultante é válido
3. **Terminação**: O número de passos é finito.

Determinar se um programa termina ou não é uma tarefa difícil, como apresentado ilustremente pelo Problema da Parada. Há situações, no entanto, em que é possível determinar se um programa atende aos primeiros 2 requisitos. Um programa que utiliza regras de tipos em suas operações pode ser verificado quanto ao seu progresso e preservação. Se ambos forem atendidos, tem-se que o programa é seguro (*safe*). A nona atividade visa trabalhar com a verificação da segurança de programas através da análise estática.

A linguagem-modelo foi expandida para incluir regras de tipos em suas operações. Da mesma forma que os métodos *eval* computam o valor de uma dada expressão durante a interpretação do programa, os novos métodos *type_eval* computam as suas respectivas regras de tipo durante a verificação de tipos. Os tipos de dados da linguagem-modelo foram implementados como um *Enum* em Python contendo dois possíveis valores: *NUM* e *BOOL*. Da mesma forma como a associação entre variáveis e valores é mantida por uma classe *Env*, uma nova classe *TypeEnv* foi criada para mapear cada variável a seu tipo. É possível que os tipos encontrados em uma dada instrução sejam incompatíveis, caso o qual a verificação de tipos levanta uma exceção indicando que o programa não é seguro. Para esse fim foi criada a exceção *InstTypeErr*, cuja mensagem de erro é fundamental durante a verificação automática (figuras 34, 35, 36).

```
class LangType(Enum):
    NUM = 1
    BOOL = 2
```

Figura 34. Definição dos tipos para a linguagem-modelo

```python
class TypeEnv(Env):
    @classmethod
    def from_env(cls, env: Env):
        d = env.to_dict()
        type_d = dict()
        for k, v in d.items():
            t = type(v)
            if t is int:
                type_d[k] = LangType.NUM
            elif t is bool:
                type_d[k] = LangType.BOOL
            else:
                raise TypeEnvErr
        return TypeEnv(type_d)

    def set(s, var, value: LangType):
        """
        This method wraps the inherited set(var, value) method, making
        sure only LangType values are accepted.
        """
        if type(value) is not LangType:
            raise TypeEnvErr
        else:
            s.env.appendleft((var, value))
```

Figura 35. Implementação do ambiente de tipos para Type Analysis

```python
class InstTypeErr(Exception):
    def __init__(s, inst: Inst, expected: LangType, found: LangType):
        s.inst = inst
        s.expected = expected
        s.found = found
        message = (
            f"Type error in instruction {inst.ID}\n"
            f"Expected: {expected}, found: {found}"
        )
        super().__init__(message)
```

Figura 36. Definição da exceção específica para erros de tipo durante Type Analysis

Cabe aos discentes implementar as funções *type_eval* de acordo com cada instrução, inclusive o caso em que uma exceção deve ser lançada (figura 37). A implementação é feita no próprio arquivo **lang.py**. A avaliação automática conta com programas de teste cuja segurança é analisada com auxílio da implementação do aluno. O resultado da verificação de tipo dos programas é impresso na saída padrão, onde é comparado com um resultado pré-estabelecido (figura 38).

```python
class ReadBool(Inst):
    """
    The ReadNum instruction introduces non-constant values to the program.
    This blocking instruction requests a numerical input from the user.
    """

    def type_eval(s, type_env):
        type_env.set(s.dst, LangType.BOOL)
```

Figura 37. Exemplo de método para computação da verificação de tipos

```
if __name__ == "__main__":
    lang.Inst.next_index = 0
    lines = sys.stdin.readlines()
    env, program = parser.file2cfg_and_env(lines)
    try:
        lang.type_check(program[0], lang.TypeEnv.from_env(env))
        print("The program type checks.")
    except lang.InstTypeErr as e:
        print(e)
```

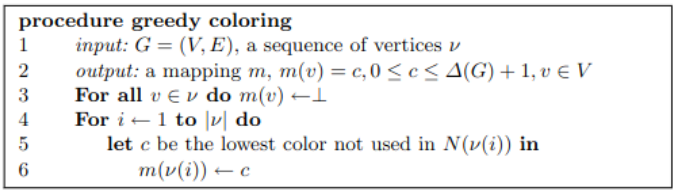
Figura 38. Avaliação automática para a tarefa de Type Checking

4.11. Register Allocation

Quando se discute a otimização de software, uma das restrições mais relevantes são as limitações do hardware subjacente. Uma forte discrepância entre as duas tecnologias é a perspectiva sobre memória: enquanto o hardware possui um número limitado de registradores, programas podem trabalhar com um número arbitrário de variáveis temporárias. Se todos os registradores estiverem ocupados e uma nova variável temporária for introduzida, ocorre um *spill*: o valor de um dos registradores é armazenado em memória, e o registrador passa a ser ocupado pela nova variável. O problema de alocação registradores consiste em determinar a melhor forma possível de se distribuírem os registradores da máquina entre as variáveis de um programa, minimizando a necessidade de se realizarem *spills*. Trata-se de um problema NP-completo [Chaitin 1982]: a alocação de registradores é análoga à coloração mínima de um grafo de interferência entre as variáveis de um programa. Um grafo de interferência é construído tomando-se as variáveis de um programa como vértices e adicionando arestas entre variáveis cujo tempo de vida se sobrepõe. O tempo de vida é calculado via *liveness analysis*.

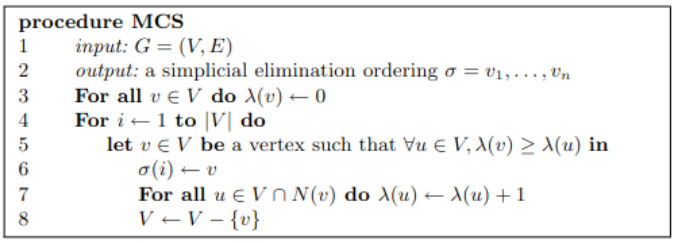
Há subconjunto do problema, no entanto, mais tratável. Conforme demonstrado por [Hack et al. 2006], programas em formato SSA possuem grafos de interferência entre variáveis de um tipo específico, nomeadamente cordais. Grafos cordais são excepcionais por permitirem soluções de complexidade polinomial para alguns problemas outrora NP-completos como a coloração mínima. Segue que o problema de alocação de registradores em programas SSA pode ser resolvido de forma eficiente [Pereira and Palsberg 2005].

A última atividade consiste na implementação da primeira etapa do algoritmo proposto por [Pereira and Palsberg 2005]: a coloração mínima do grafo de interferências de um programa em formato SSA. O algoritmo emprega a coloração gulosa (figura 39) sobre o grafo de interferência de variáveis sua *ordenação de eliminação simplical*, resultado da aplicação do algoritmo de *maximum cardinality search* (figura 40). O discente deverá implementar ambos algoritmos, cujo resultado conjunto será a alocação mínima de registradores de um programa anterior à introdução de *spillings* (figura 41). Durante a verificação automática utiliza-se a implementação do aluno para calcular a alocação de registradores de um conjunto de programas de teste. O resultado das alocações é impresso na saída padrão e comparado com valores pré-determinados (fgura 42).



procedure greedy coloring
1 *input:* $G = (V, E)$, a sequence of vertices ν
2 *output:* a mapping m, $m(v) = c, 0 \leq c \leq \Delta(G) + 1, v \in V$
3 **For all** $v \in \nu$ **do** $m(v) \leftarrow \perp$
4 **For** $i \leftarrow 1$ **to** $|\nu|$ **do**
5 **let** c be the lowest color not used in $N(\nu(i))$ **in**
6 $m(\nu(i)) \leftarrow c$

Figura 39. Algoritmo de coloração gulosa de grafos (retirado de [Pereira and Palsberg 2005])



procedure MCS
1 *input:* $G = (V, E)$
2 *output:* a simplicial elimination ordering $\sigma = v_1, \ldots, v_n$
3 **For all** $v \in V$ **do** $\lambda(v) \leftarrow 0$
4 **For** $i \leftarrow 1$ **to** $|V|$ **do**
5 **let** $v \in V$ **be** a vertex such that $\forall u \in V, \lambda(v) \geq \lambda(u)$ **in**
6 $\sigma(i) \leftarrow v$
7 **For all** $u \in V \cap N(v)$ **do** $\lambda(u) \leftarrow \lambda(u) + 1$
8 $V \leftarrow V - \{v\}$

Figura 40. Algoritmo de Maximum Cardinality Search (retirado de [Pereira and Palsberg 2005])

```python
class InterferenceGraph:
    def __init__(s, program: list[Inst]):
        liveness = abstract_interp(liveness_constraint_gen(program))
        s.graph = defaultdict(lambda: set())
        s.mcsgraph = defaultdict(lambda: set())
        s.weights = dict()
        s.maxweight = 0
        # build intersection graph
        for values in liveness.values():
            for var in values:
                s.graph[var] |= set(values) - set([var])
                s.mcsgraph[var] |= set(values) - set([var])
        for var in s.graph.keys():
            s.weights[var] = 0
        # Sort graph vertices (variables) according to weight
        s.weightedVs = sorted(s.weights.keys(), key=lambda x: s.weights[x])

    def N(s, v: str) -> set[str]:
        return s.graph[v]

    def size(s):
        return len(s.mcsgraph)

    def greedy_coloring(s, sequence: list[str]) -> tuple[dict[str, int], int]:
        # TODO: implement this method

    def maximum_cardinality_search(s):
        # TODO: implement this method
```

Figura 41. Implementação parcial da coloração de grafos para alocação de registradores

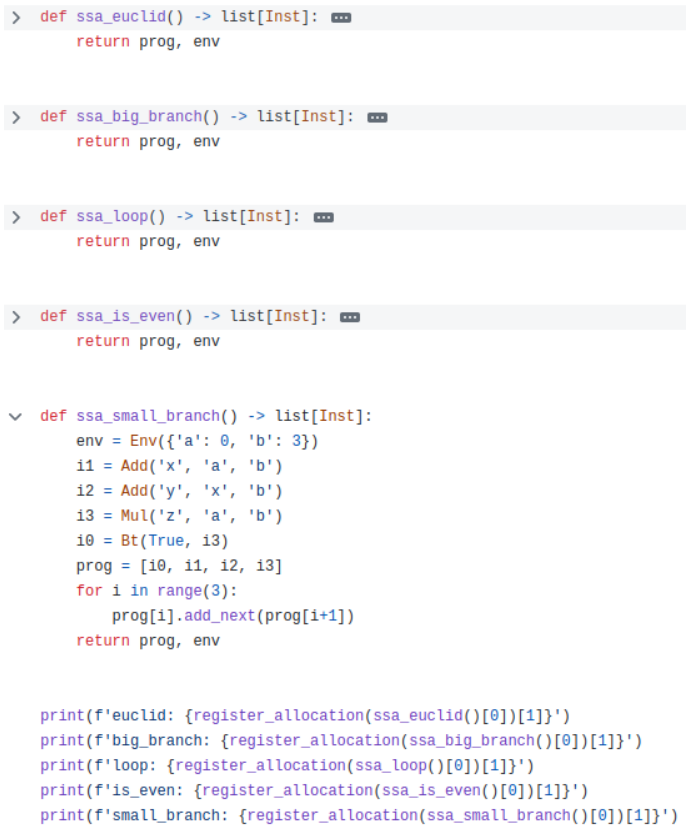


Figura 42. Avaliação automática para a atividade de SSA Register Allocation

5. Fechamento

A Análise Estática de Programas apresenta um arcabouço de teorias e ferramentas que promovem melhorias em diversos aspectos do código antes de sua execução. Sua aplicação resulta em programas com maior eficiência computacional, corretude e mesmo segurança. Tais melhorias representam vantagens econômicas, ambientais e sociais paraa o processo produtivo de diversas aplicações. Todavia, a significância do ramo de pesquisa não é refletida pela disponibilidade de material para seu estudo. Dessa forma há um grande ganho em aprimorar o ensino dessa teoria oferecido pelo Departamento de Ciência da Computação da UFMG.

A contribuição deste projeto está na expansão do material didático da disciplina **DCC88 - Static Program Analysis** na forma de dez atividades práticas versionadas no github que podem ser disponibilizadas como VPLs via Moodle. Cada atividade corresponde a um subconjunto das aulas ministradas na disciplina, e consiste em induzir os discentes a completarem um programa correspondente a procedimento instrumental para a realização de uma análise estática. Acompanham as atividades programas-exemplo e documentação *doctest* que atuam como casos de teste para se avaliar a implementação dos discentes.

A implementação das atividades consistiu em um aprendizado bivalente. Por um lado, exercitou-se por parte do autor a compreensão da teoria subjacente de Análise Estática de Programas envolvida em cada atividade. Para cada atividade foi necessário revisar o tópico das aulas correspondentes e implementar um programa que realizasse al-

guma operação vista em aula. Lançando mão do código-base, foi necessário elaborar e implementar em Python distintas análises de fluxo de dados e algoritmos para a resolução de tais sistemas de equações. Foi também necessário trabalhar com conceitos de teoria dos grafos para a construção de grafos de dominância e coloração de grafos cordais.

Por outra perspectiva, a escrita de tais códigos como atividades práticas voltadas para futuros discentes orientou sua estruturação de modo a favorecer seu valor didático. Além de atender a boas práticas de engenharia de software, observou-se como isolar e omitir os algoritmos que correspondem às técnicas vistas em aula. Dessa forma é possível exigir que o aluno realize as tarefas completando seções de código com comportamento e extensão bem-definidos.

Tem-se como trabalhos futuros a melhoria das atividades criadas mediante observação da experiência de sua aplicação em aula e a criação de mais casos de teste para as atividades. A maior variedade de programas de teste visa capturar resultados oriundos de implementações parcialmente corretas, aumentando a nuance da avaliação automática. Espera-se compor um conjunto de atividades que maximize o aproveitamento da disciplina pelos alunos, e que forneça um método a mais de organização e avaliação durante o ministrar da disciplina.

Referências

Andersen, L. O. (1994). Program analysis and specialization for the c programming language.

Budzynska, S. (2024). Codeql zero to hero part 1: the fundamentals of static analysis for vulnerability research. `https://github.blog/developer-skills/github/codeql-zero-to-hero-part-1-the-fundamentals-of-static-analysis-for-vulnerability-research/`. Acessado em 27-07-2024.

Chaitin, G. J. (1982). Register allocation & spilling via graph coloring. *ACM Sigplan Notices*, 17(6):98–101.

GitHub, Inc (2022). `https://octoverse.github.com/2022/top-programming-languages`. Acessado em 23-07-2024.

Google (2024a). Google closure compiler. `https://github.com/google/closure-compiler`. Acessado em 27-07-2024.

Google (2024b). What is v8? `https://v8.dev/`. Acessado em 27-07-2024.

Hack, S., Grund, D., and Goos, G. (2006). Register allocation for programs in ssa-form. In *Compiler Construction: 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006. Proceedings 15*, pages 247–262. Springer.

Intel Corporation. Intel® oneapi dpc++/c++ compiler. `https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html`. Acessado em 27-07-2024.

MattPD (2022). Program analysis resources. `https://gist.github.com/MattPD/00573ee14bf85ccac6bed3c0678ddbef`. Acessado em 27-07-2024.

Møller, A. (2021). Tip exercises. https://github.com/cs-au-dk/TIP/wiki/TIP-exercises. Acessado em 27-07-2024.

Møller, A. and Schwartzbach, M. I. (2024). Static progam analysis. https://cs.au.dk/~amoeller/spa/. Acessado em 27-07-2024.

NVIDIA Corporation (2024). Nvidia cuda compiler driver nvcc. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html. Acessado em 27-07-2024.

Pereira, F. M. Q. (2021a). Data flow analysis. https://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/IntroDataFlow.pdf. Acessado em 27-07-2024.

Pereira, F. M. Q. (2021b). Introduction to code analysis and optimization. https://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/Introduction.pdf. Acessado em 27-07-2024.

Pereira, F. M. Q. (2021c). Pointer analysis. https://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/PointerAnalysis.pdf. Acessado em 27-07-2024.

Pereira, F. M. Q. (2021d). Static single assignment form. https://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/StaticSingleAssignment.pdf. Acessado em 27-07-2024.

Pereira, F. M. Q. (2024). Static program analysis - dcc888. https://homepages.dcc.ufmg.br/~fernando/classes/dcc888/. Acessado em 27-07-2024.

Pereira, F. M. Q. and Palsberg, J. (2005). Register allocation via coloring of chordal graphs. In *Asian Symposium on Programming Languages and Systems*, pages 315–329. Springer.

Rahul (2020). A hands-on introduction to static code analysis. https://deepsource.com/blog/introduction-static-code-analysis. Acessado em 27-07-2024.

Scheid, H. F. and Pereira, F. M. Q. (2024a). Dcc888. https://github.com/pronesto/DCC888. Acessado em 23-07-2024.

Scheid, H. F. and Pereira, F. M. Q. (2024b). Dcc888. https://github.com/hfscheid/DCC888/tree/solved-exercises. Acessado em 23-07-2024.

Spoofax Team (2024). Spoofax: The language designer's workbech. https://spoofax.dev/. Acessado em 27-07-2024.

Thomson, P. (2021). Static analysis: An introduction: The fundamental challenge of software engineering is one of complexity. *Queue*, 19(4):29–41.

Universidade de Stuttgart (2020). Intra-procedural static taint analysis of javascript programs. https://software-lab.org/teaching/winter2020/pa/course_project.pdf. Acessado em 27-07-2024.

Universidade de Stuttgart (2023). Progam analysis. https://software-lab.org/teaching/winter2023/pa/. Acessado em 27-07-2024.

Universidade de Stuttgart (2024). Sola - about us. https://www.software-lab.org/. Acessado em 27-07-2024.

Universidade de Técnica de Delft (2024). Homework assignments. https://studiegids.tudelft.nl/a101_displayCourse.do?course_id=48294. Acessado em 27-07-2024.