Python Slicer (Projector)

Oded Elbaz, Tomer Greenwald

Program Analysis 2016

Lecturer: Prof. Mooly Sagiv

תוכן

[מבוא 2](#_Toc441481796)

[איך להריץ? 3](#_Toc441481797)

# מבוא

בחרנו לממש Python Slicer עפ"י הצגת הפרויקט עבור Java בשיעור 12 ואישורך במייל לבצע את הפרויקט על קוד פיית'ון. קראנו לתוכנית Projector כיוון שהיא מקרינה את קוד הקלט על פני משתנה שבחר המשתמש.

הProjector מבצע אנליזה על הקוד ויוצר שני גרפים בעלי רשימת קודקודים משותפת.

כל קודקוד ברשימה מתאים לשורת קוד בתוכנית המקורית. ישנם שני סוגי קודקודים:

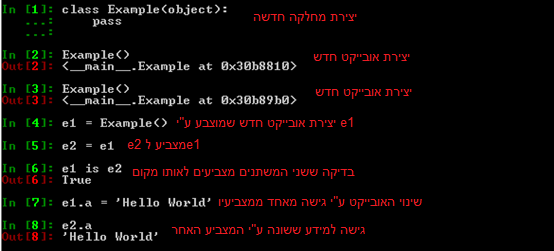
* Statement Node: מתאים לשורת השמה או ביטוי.
* Control Node: מתאים לשורת קוד שמשפיעה על הflow בתוכנית. לדוגמא if, else, while.

באשר לקשתות, סט אחד מתאר את הControl Flow בתוכנית, והשני מתאר את התלות הלוגית (Dependency) בין שתי שורות קוד.

הכוח הגדול של האנליזה טמון בעובדה שהיא מזהה שינויים באובייקט עליו אנחנו מצביעים, גם כאשר השינויים בוצעו ממצביע אחר.

תזכורת על שפת פיית'ון:

כל קריאה לConstructor יוצרת אובייקט חדש. שני משתנים יכולים להצביע אל אותו אובייקט. שינוי באובייקט משפיע על האובייקט ולא על המצביע, לכן נוכל לראות את השינוי ע"י גישה מכל אחד מהמשתנים שמצביעים אליו:

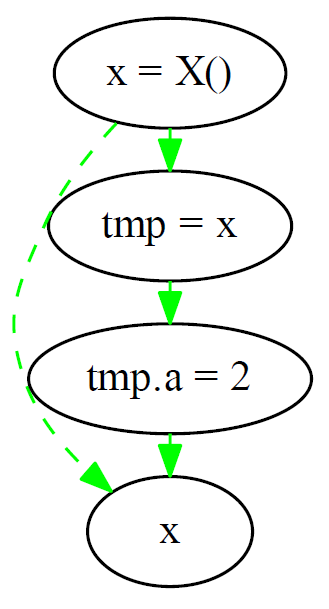


האנליזה יודעת לזהות ששני משתנים מצביעים לאותו אובייקט, וכן שורות קוד שגורמות לשינוי באובייקט זה. אנחנו עושים זאת באמצעות Points to Analysis כפי שיוסבר בהמשך. דוגמא (דוגמאות מורכבות יותר יינתנו בהמשך):



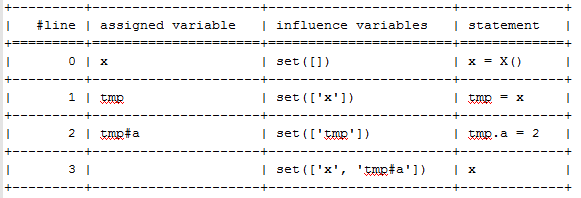
בדוגמא הזאת אנחנו מייצרים אובייקט חדש מסוג X שמצביעים אליו המשתנים x ו-tmp. אנחנו משנים את האובייקט דרך tmp, ורוצים לוודא שהמשתנה x מודע לשינויים הללו, כך שכאשר נקרין את הקוד על פני x, גם השורה tmp.a=2 תהיה בפלט (וכן השורה שמקשרת את tmp לאובייקט שעליו מצביע x).

זוהי תוצאת האנליזה:



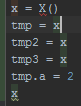
אנחנו רואים ש-x בשורה האחרונה מושפע על ידי ההשמה הראשונה למשתנה וע"י ההשמה tmp.a. tmp.a מושפע מההשמה למשתנה tmp.

"מאחורי הקלעים" האנליזה מייצרת את הטבלה הבאה:



ניתן לראות שהאנליזה יודעת שx בשורה 3 מושפע מx ו-tmp.a.

למרות שהשורה tmp=x משפיעה על x בשורה 3, tmp לא מופיע ברשימת המשתנים המשפיעים. בנינו את האנליזה כך כיוון שאנחנו לא רוצים להוסיף את כל המשתנים שמצביעים אל האובייקט, אלא רק אלו שמשפיעים על התוצאה. השורה tmp=x תתווסף לפלט מכיוון שtmp.a=2 תלוי בtmp. כך לדוגמא, אם נסתכל על הקוד הבא:



ההשמות של האוביקט אל tmp2 ו-tmp3 לא ישפיעו על x.

הנחת העבודה היא שהקוד בצורת SSA, כך שההשמה לכל משתנה היא יחידה. לכן, אם, לדוגמא, tmp.a נמצא ברשימת המשתנים המשפיעים, עלינו לחפש היכן בוצעה ההשמה היחידה לtmp.a.

כפי שניתן לראות כבר מהדוגמא הפשוטה הנ"ל, הקוד יודע לטפל בגישה לAttributes של משתנה. נשים לב שזוהי הדרך היחידה לשנות אובייקט. במילים אחרות, שינוי אובייקט הוא שינוי הAttributes שלו.

לאחר שבנינו את הגרפים, הקרנת הקוד היא כבר עניין טכני – אנחנו הולכים מהסוף להתחלה עד אשר אנחנו מזהים את הפעם הראשונה שבה המשתנה המוקרן מושפע. משם אנחנו מטיילים אחורה על פני קשתות הDependency. כל פעם שאנחנו מגיעים לקודקוד חדש, אנחנו בודקים האם יש Control Flow בין שני הקודקודים. במידה ויש, אנחנו מוסיפים את הקודקוד החדש לקוד המוקרן וכן את כל הControl Nodes בדרך.

לאחר שהאנליזה הסתיימה אנחנו יוצרים שלושה סוגי פלטים:

1. רשימה מלאה של תוצאת האנליזה (analysis\_result.txt)
   1. טבלה המתארת איזה משתנים משפיעים ומושפעים עבור כל שורת קוד.
   2. רשימת Control Edges
   3. רשימת Dependency Edges
2. המחשה גרפית של הגרפים שהאנליזה יצרה:
   1. Out.gv\_control.pdf: גרף הControl Flow בתוכנית
   2. Out.gv\_dep.pdf: גרף הDependency בתוכנית
   3. Out.gv\_all.pdf: איחוד של שני הגרפים הקודמים
3. הקוד המוקרן על פני אחד המשתנים (projected\_code.py)

# איך להריץ?

התוכנית נבדקה ועובדת על פני פלטפורמת Windows, ופיית'ון 2.7.

החבילות הבאות צריכות להיות מותקנות על מנת שיהיה אפשר להריץ את התוכנית:

* Astor – חבילה שבאמצעותה אנחנו הופכים אובייקטי AST לשורת קוד:

<https://pypi.python.org/pypi/astor/0.5>

* Graphviz – חבילה שבאמצעותה אנחנו מדפיסים בצורה גרפית את הגרפים שאנחנו יוצרים:

<https://pypi.python.org/pypi/graphviz/0.4.8>

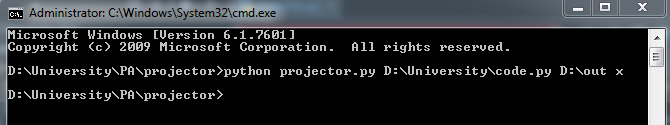
אם ברצונך להריץ את הטסטים, עליך להתקין pytest:

<https://pypi.python.org/pypi/pytest/2.8.5>

כאשר כל התלויות מוכנות, ניתן להריץ את האנליזה בצורה הבאה:

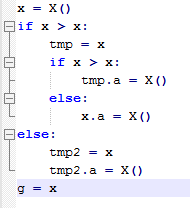
python projector.py <original\_code> <output\_dir> <projected variable>

דוגמאת הרצה:



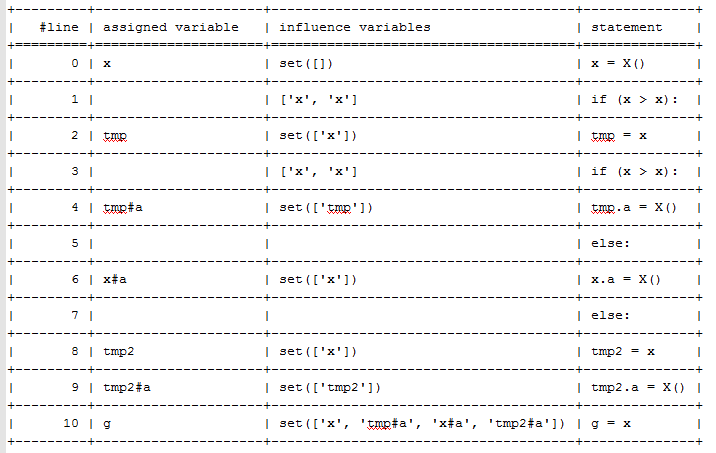
# כיצד לקרוא את הפלט

על מנת להדגים את קריאת הפלט נריץ את האנליזה על התוכנית הבאה:



ההטלה שבחרתי להדגים היא על פני משתנה g.

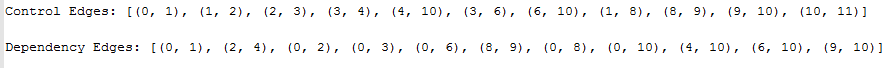
ראשית נתרכז בטבלה שבפלט analysis\_result.txt:



ניתן לראות שישנה שורה עבור כל שורת קוד.

כל שורה מציגה איזה משתנה מושפע כתוצאה מהפעלת השורה (assigned variable), ואיזה משתנים משפיעים על הרצת השורה (influence variable). כך לדוגמא אם נסתכל על שורה 2, המשתנה המושפע הוא tmp כי זוהי שורת השמה למשתנה זה, והמשתנה המשפיע עליו הוא x. עבור שורה 1, אף משתנה לא מושפע מהרצת שורה זאת (זוהי שורת if), אך המשתנים המשפיע על שורה זאת הוא x.

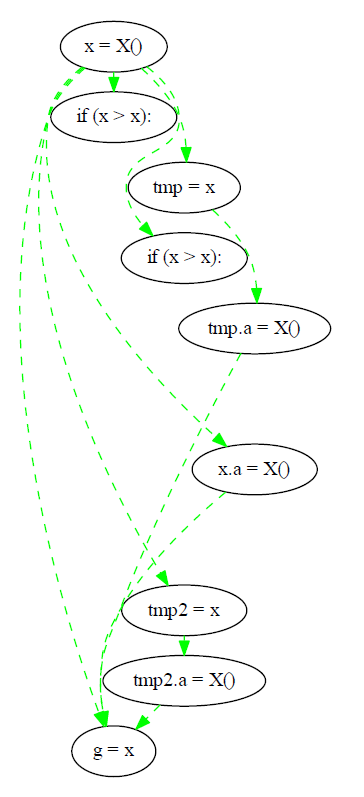
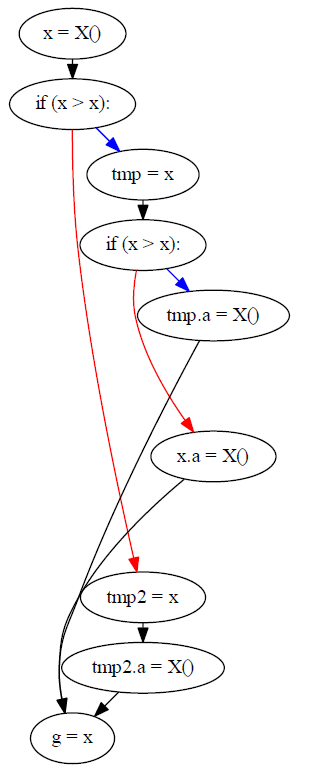
כעת נתרכז ברשימות הקשתות המופיעות באותו קובץ:



Control Edges מתארת את הControl Flow של הקוד. אם הקוד טורי אזי יש קשת בין שורת קוד לשורה הבאה אחריה, לדוגמא 0 ל-1. אם יש קפיצות (לדוגמא בif או while), אזי יש קשת עבור כל קפיצה מתאימה. לדוגמא 1 ל-2 ו1 ל-8.

Dependency Edges היא רשימה המחזיקה את התלויות הלוגיות. לדוגמא זיהינו בשורה 10 את התלות בx, tmp.a, x.a, tmp2.a, ולכן יש קשת אל שורה 10 מההשמות אל משתנים אלו (נזכיר שאנחנו מניחים שהקוד בצורת SSA).

קבצי הpdf שנוצרו בתיקיית הפלט מציגים את שני הגרפים בצורה גרפית:



ולבסוף ניתן לראות את הקוד המוקרן בקובץ projected\_code.py.

הערה:

כפי שניתן לראות, הטבלה שבקובץ analysis\_result הינה ליבת האנליזה, כל שאר הפעולות הן טכניות מעליה. לכן מעתה נציג במסמך רק את הטבלה הזאת.

כמו כן, הדוגמאות שנציג יהיו קצרות מאוד, כי אנחנו רוצים להתרכז במהות ולא לבלבל עם מספר רב של שורות קוד. ניתן להרחיב את הדוגמאות והקונספטים ליצירת קוד מורכב יותר.

# הנחות עבודה

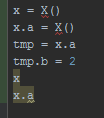
* האנליזה היא intraprocedural.
* קוד הקלט הוא בצורת SSA (כיוון שאנחנו לא מאפשרים פונקציות, אזי אנחנו כן תומכים בהשמה למשתנה בשני branchים שונים, ואנחנו מתחייבים להיות Sound אך לא מדויקים ביותר). אם הקוד אינו בצורת SSA יכול להיות שהאנליזה תעבוד כמו שצריך, אך אין הבטחה לכך.
* כל משתנה מחזיק אובייקט או Integer (ניתן להרחיב כך שהמשתנה יחזיק גם טיפוסים נוספים, אך זהו עניין טכני).
* הקוד אינו מכיל הגדרות מכל סוג שהוא, ההנחה היא שהכל מוגדר כפי שצריך (לדוגמא אם קוראים לConstructor של המחלקה X אזי המחלקה מוגדרת כראוי).
* האנליזה מטפלת בהשמה של ערך או פעולה בינארית (ניתן להרחיב, זהו הליך טכני).
* הקוד יודע לטפל במבנה של If-else, while וקינון ביניהם.
* אובייקטים
  + האובייקטים הם מהצורה:



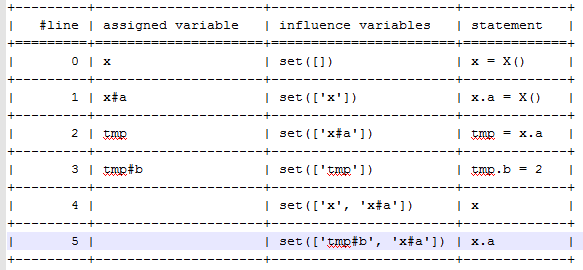
כלומר בברירת מחדל ללא Attributes, ניתן להוסיף אותם דינמית.

* + לא ניתן לגשת לAttribute של Attribute, לצורך כך יש להשתמש במשתנה עזר.
  + הפרויקט עובד על Shallow Pointers, לכן לדוגמא x.a.a משפיע על x.a ולא על x (בצורה דומה x.a משפיע על x).
  + הקריאה לConstructor חוקית.
  + דוגמא לסעיף אובייקטים:

נניח נתון הקוד הבא:



אזי האנליזה תייצר את הטבלה הבאה עבורו:



# אלגוריתם האנליזה

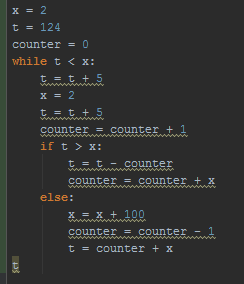
בחלק זה נתאר את האלגוריתם כאשר אין Aliasing. את שדרוג האלגוריתם על מנת שיתמוך בAliasing נפרט בחלק הבא.

אנחנו מתארים את הקוד רק מבחינה לוגית על מנת לשמור על הסדר וההבנה. מבחינה טכנית הקוד מסובך בהרבה ויש בו עוד פעולות רבות.

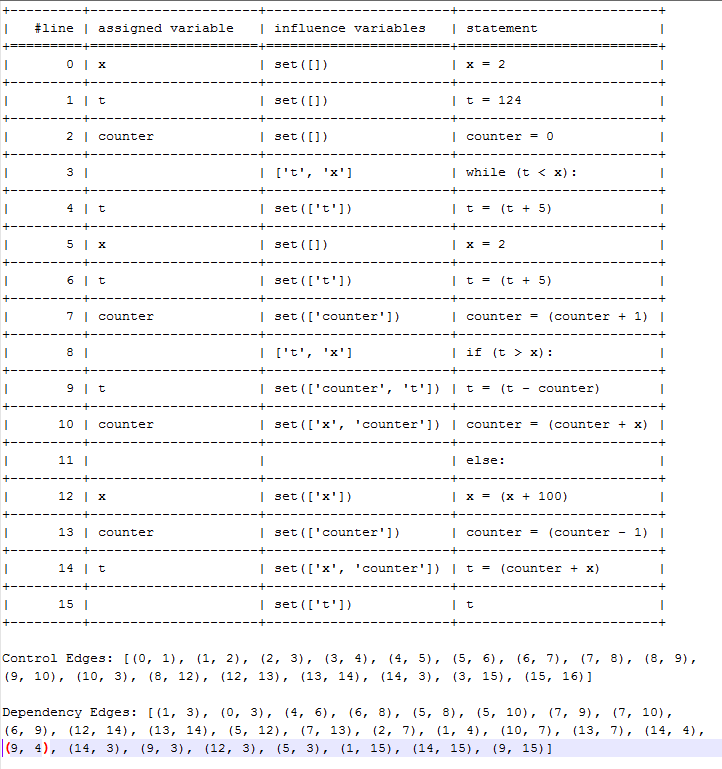
עבור כל שורת קוד שאנחנו מגיעים אליה:

* עבור שורת השמה:
  + מצא את המשתנה שההשמה מתבצעת אליו
  + מצא אילו משתנים משפיעים על ההשמה:
    - בפעולה בינארית – כל אחד מהמשתנים המשתתפים בפעולה הבינארית
    - בהשמת משתנה – המשתנה עצמו וכל השמה לAttributeים שלו
    - בהשמת Attribute – הAttribute עצמו וכל השמה לAttributeים שלו
    - אם משימים לAttribute – המשתנה עצמו
  + צור קודקוד חדש המייצג את הצומת
  + נסה ליצור קשתות dependency בין השורות בהם מבצעים השמה למשתנים המשפיעים לבין השורה הנוכחית:
    - אנחנו עושים זאת באמצעות מילון המחזיק עבור כל משתנה מתי הוא הושם לאחרונה. המילון יכול למפות למספר שורות קוד שונות, כפי שהוסבר בהנחות העבודה.
    - ייתכן שהמשתנה המשפיע אינו נמצא במילון. זה קורה בגלל שאנחנו בקטע קוד פנימי, והמשתנה הוגדר בבלוק החיצוני. אנחנו מכניסים את המשתנה הנעלם למילון נוסף, ומסמנים על איזה שורות הוא משפיע.
  + צור קשתות Control לצומת הבאה.
  + עדכן את המילון המחזיק לכל משתנה מתי הוא הושם בפעם האחרונה.
* עבור שורת ביטוי:
  + הטיפול מתבצע בצורה דומה פרט לכך שלא מוצאים את המשתנה שההשמה מתבצעת אליו ולא מעדכנים את המילון
* עבור שורת IF:
  + צור קשתות תלות בין התנאי שבIF וההשמות של המשתנים הרלוונטים
  + צור קודקוד Control
  + צור קשת עבור הThen ועבור הelse
  + עבור החלק של הThen וה Else:
    - הרץ את האנליזה בצורה רקורסיבית על החלקים הפנימיים
    - אחד בין תוצאת האנליזה הפנימית לריצה הנוכחית:
      * אחד את רשימת הקודקודים
      * אחד את רשימת הקשתות
      * נסה לטפל במשתנים לא ידועים של הקוד הפנימי
    - תקן קשתות שמצביעות לא למקום הנכון (לדוגמא אם יש בלוק if-else בתוך if אז הסוף של הif הפנימי עלול לעשות קשת ממנו לקוד שאחרי הelse הפנימי, למרות שהוא צריך להצביע לקוד שאחרי הelse החיצוני).
    - עשה merge למילון ששומר מתי משתנה נראה לאחרונה
* עבור שורת While:
  + הטיפול דומה לטיפול בIF
  + השוני הוא שכיוון והריצה הראשונה של הלולאה יכולה להשפיע על הריצה השניה אזי מריצים את האנליזה פעמיים על קטע הקוד ומעדכנים את המילונים בהתאם.

דוגמאת קוד:



תוצאת האנליזה:



הסבר:

* על שורות 0-3 אין השפעות חיצוניות. כאשר הקוד מטפל בכל אחת מהן הוא מעדכן מתי הוא ראה את כל אחד מהמשתנים הללו לאחרונה.
* התנאי של הwhile הוא t<x, לכן t ו-x משפיעים ומתווספות קשתות (1, 3) ו-(0, 3).
* עבור הריצה הראשונה של הלולאה מפעילים את האנליזה בצורה רקורסיבית, כעת הקוד הנבדק הוא גוף הלולאה.
  + השורה t=t+5 מושפעת מהt החיצוני. אכן נוצרת הקשת (1,4), אך כפי שניתן לראות היא נוצרת בשלב מאוחר יחסית, כי כשאנחנו עובדים על הקוד הפנימי אנחנו לא יודעים מיהו t ואנחנו מכניסים אותו בהתחלה ל"רשימת המשתנים הלא ידועים". מעדכנים מתי שמנו ערך פעם אחרונה לt.
  + השורה x=2 לא מושפעת מכלום, מעדכנים מתי שמנו ערך לx.
  + השורה t=t+5 (6) כעת מושפעת מהt בשורה 4, לכן נוצרת הקשת (4, 6) ואנחנו מעדכנים את המילון.
  + Counter = counter + 1 טיפול דומה לשורה הראשונה t=t+5
  + מטפלים בif ובelse בנפרד אך בצורה דומה לדרך שבה אנחנו מטפלים בWhile. לאחר מכן ממזגים את תוצאת האנליזה שלהם לגוף הwhile:
    - לאחר פעולת המיזוג t הושם לאחרונה בשני מקומות שונים – 14 ו-9. גם Counter הושם לאחרונה בשני מקומות – 10 ו13.
  + מעלים את תוצאת האנליזה של block הwhile וממזגים אותה עם הבלוק הראשי. נשים לב שעכשיו לדוגמא t נראה ב3 מקומות שונים – 1, 9 ו14. (אם היינו יודעים בוודאות שלולאת הwhile מתבצעת אזי היינו יכולים לשכוח מההשמה בשורה 1, אך אנחנו לא יודעים זאת).
    - כמו כן, אנחנו יוצרים קשתות עבור כל המשתנים שלא ידענו היכן הוגדרו. יכול להיות שגם בבלוק החיצוני המשתנים הללו לא ידועים, ובמקרה כזה נסמן את המשתנים ככאלו, ונחכה לבלוק חיצוני יותר שיטפל בהם).
* עבור כל ריצת לולאה נוספת אנחנו מבצעים את האנליזה על גוף הלולאה פעם נוספת. כך לדוגמא נוצרות הקשתות (9, 4) ו-(14,4).

# Points to analysis

# דוגמאות