A CONTEXT-SENSITIVE FORMAL TRANSFORMATION FOR RESTRUCTURING PROGRAMS

A thesis presented to
the Graduate Faculty of
the University of Southwestern Louisiana
in partial fulfillment of the requirements for
the degree Master of Science

Jean-Christophe Deprez

Fall 1997

© Jean-Christophe Deprez

1997

All Rights Reserved

A CONTEXT-SENSITIVE FORMAL TRANSFORMATION FOR RESTRUCTURING PROGRAMS

Jean-Christophe Deprez

APPROVED:

Arun Lakhotia, Chair Associate Professor of Computer Science	William R. Edwards Jr. Associate Professor of Computer Science
Kemal Efe Associate Professor of Computer Science	Lewis Pyenson Dean, Graduate School

To the memory of

my father, Jean-Jacques Deprez

Acknowledgment

My deepest thanks to Dr. Arun Lakhotia for his supervision. Dr. Lakhotia was always available and gave some very constructive observations. I also want to give my greatest gratitude to Dr. William R. Edwards Jr. and Dr. Kemal Efe. Both gave valuable comments that significantly improved the quality of this thesis.

I want to express my thanks to my mother, Nadine Deprez Lemoine and my brother, Mathias Deprez who have shared every of my joys and disappointments. To both of you, and to my father whom I miss very much: Je vous aime.

The work was partially supported by a contract from the Department of Defense and a grant from the Department of Army, US Army Research Office. The contents of the paper do not necessarily reflect the position or the policy of the funding agencies, and no official endorsement should be inferred.

Abstract

Over time, due to repeated modifications, the structure of a software deteriorates, causing its logical threads to get intertwined, like noodles in a bowl of spaghetti. The threads become entangled to a point where the program can hardly be understood or modified. We have developed a transformation that restructures a software – disentangles its logical threads – to reduce its complexity. Our transformation is intuitive in that it mimics the activities a programmer may perform during manual restructuring. Our transformation is also general in that it is applicable to any re-engineering need. We have proven that the transformation does not alter the semantics of the restructured program. Our transformation is a composition of two primitive transformations: TUCK and SPILT. Given a set of statements specified by a programmer, TUCK identifies the computational thread related to these statements, and SPLIT separates this thread from the rest of the entangled computations.

A restructuring transformation such has that presented has not been studied before. Previous transformations for restructuring programs were either not intuitive, or not general, or did not preserve the semantics of programs. Thus, our transformation is better amenable for use in a re-engineering environment.

Biography

Mr. Jean-Christophe Deprez was born in Liège (Belgium) on January 1st, 1971. He graduated with a Bachelor degree in May 1994 from the University of Southwestern Louisiana. After a year spent in the industry, he came back to U.S.L. in 1995, under the Graduate School fellowship program. In Fall 1996, he started to work in software re-engineering, and participated in the successful completion of a project for the U.S. Department of Defense. Mr. Jean-Christophe Deprez is now planning to continue his Ph.D. studies in software re-engineering.

Table of Contents

Table of figures

1. Introduction

1.1 Motivation

Software is composed of many small logical threads. Over time, due to repeated modifications, the structure of a system deteriorates, causing its logical threads to get intertwined, like noodles in a bowl of spaghetti. The threads become entangled to a point where the program can hardly be understood or modified.

The deterioration of structure is not always an indication of poor programming practices. It is a law of software evolution [Lehman85, page 253]. A good design, by definition, optimizes on several constraints [Dasgupta94]. But a design that is sound for a given set of constraints may not be sound for another. Given the changes in the market, customer needs, and computing environment, the initial version of a software soon gets outdated. The software, optimally designed for an initial set of constraints, is modified to satisfy a different set of constrains. The modifications are performed under market and schedule pressures that do not leave room for optimizing the design to the new set of constraints. Every successive change compromises the optimality of the system design, and its structure deteriorates.

One wonders: Why couldn't companies discard the old system and develop a new one afresh, whose design is sound for the new constraints?
The answer comes from the following list of myths compiled by Yourdon [Yourdon92, pages 239-240]:

> *Myth 1:* We can always afford to scrap out the old system and replace them with new systems, as long as we can demonstrate to the users & management that the new system will be better.

Myth 2: We can be absolutely, positively, totally confident that a new replacement would be much, much, much better that the old one.

Myth 3: We can always figure out what the old system is doing and translate it into a new implementation.

According to Yourdon, world-class organizations have recognized these myths and have abandoned them. These myths further substantiate the necessity of software restructuring. Since we cannot always recover all of the specifications of an old system and cannot guarantee that it performs exactly what the users want, we must restructure the old system to enable new modifications.

1.2 Research objectives

To restructure a software is to change its internal structure without affecting its external behavior [Chikofsky90]. This thesis is directed towards developing a formal transformation for restructuring software. This transformation should separate the intertwined logical threads of an old program to reduce its complexity. The transformation should be intuitive, general, and semantics preserving. A transformation is intuitive if a programmer can anticipate its results. It is general if it can be applied to any re-engineering goal desired by a programmer. It is semantics preserving if it does not modify the external semantics of the transformed program.

1.3 Impact of the research

Our restructuring transformation when introduced in a software re-engineering environment will offer the following benefits:

- Reduction of maintenance costs: In the absence of automated support, programmers restructure software manually. The manually restructured programs must be tested to ensure their behavior is not changed. This increases the cost of maintenance. Programs restructured

using our transformation need not be re-tested since their external semantics is guaranteed to remain the same.

- Smooth migration of old code to new technology: Due to the rapid changes in technology, there is a constant need to migrate software developed using one language or design paradigm to another. Our transformation may be used to restructure the old code such that it effectively uses the advantages offered by a new paradigm.

1.4 Overview of the thesis

After this introduction, in Chapter 2, we discuss the strategy used to develop our transformation. In Chapter 3, we introduce the background information and terminology used for expressing our transformation. Chapter 4 presents the transformation in detail as well as the proof that it is semantics preserving. We explore the use of the transformation through two examples in Chapter 5. Before the conclusion, we review the related research in Chapter 6

2. Restructuring strategies

In this chapter we provide insight into the issues in restructuring a software and derive strategies that guide the development of our transformation.

It is our objective that the transformation we develop be intuitive, i.e., its results should be fairly close to what a programmer may do. To gain insight into how a programmer may restructure a program, in Section 2.1, we present a verbal simulation of a programmer restructuring some code. A step in this simulation consists of a question asked by a programmer and the answer to that question. In Section 2.2, the question/answer sequence is abstracted to model the restructuring process followed by the programmer. This process model consists of steps that a programmer takes during restructuring. Automated support for restructuring may be provided by automating one or more of these steps.

Our restructuring transformation automates the last step in the restructuring process. This step is further subdivided in two smaller steps, each of which can be mapped to a primitive transformation. These transformations are introduced in Section 2.3.

2.1 Simulated scenarios

```
1  Procedure Sale_Pay_Profit (days: integer;
                   cost: float; var sale: int_array;
                   var pay:  float; var profit: float;
                   process: boolean);
2   var i, j: integer;total_sale, total_pay: float;
3   begin
4      i:=0;
5      while i < days do begin
6           i := i + 1;
7           readln(sale[i])
8      end;
9      if process = True then begin
10       total_sale:=0;
11       total_pay:=0;
12       for j := 1 to days do begin
13          total_sale := total_sale + sale[i];
14          total_pay := total_pay + 0.1 * sale[i];
15          if sale[i] > 1000 then
16              total_pay := total_pay + 50;
17        end;
18        pay := total_pay / days + 100;
19        profit := 0.9 * total_sale - cost;
20     end;
21  end;
```

Figure 1: Program Sale_Pay_Profit. Example of a deteriorated function

Consider the program Sale_Pay_Profit of Figure 1. This program – as its name suggests – computes the sale, the pay, and the profit for an organization. A close inspection of the program reveals that it by passes the computation of sale, pay, and profit if its *'process'* parameter is not true. The computation of pay and sale are unrelated but for the fact that they are computed simultaneously for the same number of days.

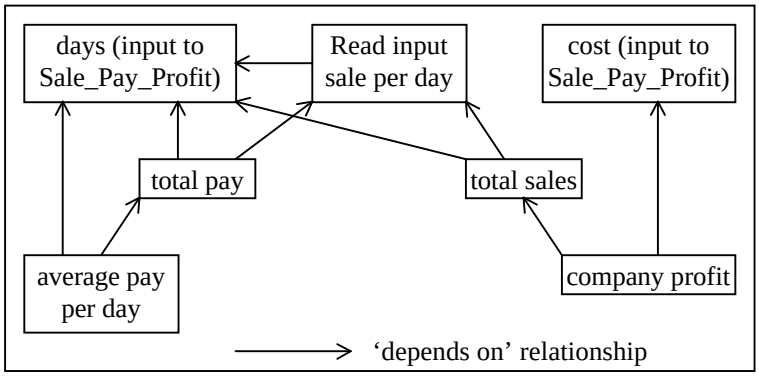
We now simulate the steps a programmer may follow to restructure this program. The steps are presented as a series of questions the programmer may ask, and the corresponding answers.

Question: What are the different tasks computed by the procedure?

Answer: The program performs the following tasks: read the sale per day from the input line, calculate the total sale, the overall pay, the average pay per day, and the company profit. These tasks depend on two inputs: The number of days worked and the cost of production. For uniformity, we treat these inputs as tasks too.

Question: How are the tasks dependent on each other?

Answer: The dependencies between tasks is given in the graph below.



Question: How are the results of the procedure being re-engineered used in other procedures? More specifically are all the results (reference parameters) of each task used outside this procedure? If so, are all its results used by the procedures calling it?

Answers to these questions, may influence the programmer's restructuring decisions but are outside the scope of this thesis. They are left for further research.

Question: What are the goals of the re-engineering tasks?

Answers: Let us assume these two different goals:

- Separate the parsing of input from the core of the program in preparation for modifying the user interface of the application. For efficiency reasons, the main part of the code must not be modified. Figure 2 contains the result of such restructuring
- Isolate each task in order to identify objects in the new program, as in Figure 3.

```
Procedure Sale_Pay_Profit (days: integer;
    cost: float; var sale: int_array;
    var pay:  float; var profit: float;
     process: boolean);
var j: integer; total_sale, total_pay: float;
begin
    sale := Read_Input(days, sale);
    if process = True then begin
      total_sale := 0;
      total_pay := 0;
      for j := 1 to days do begin
          total_sale :=total_sale + sale[j];
          total_pay := total_pay + 0.1 * sale[j];
          if sale[j] > 1000 then
              total_pay  := total_pay + 50;
      end;
      pay := total_pay / days + 100;
      profit := 0.9 * total_sale - cost;
    end;
end;
```

```
Function Read_Input(days: integer;
    var sale: int_array): int_array;
var i:integer;
begin
    i:=0;
    while i < days do begin
        i := i + 1;
        readln(sale[i])
    end;
    return (sale);
end;
```

Figure 2: Sale_Pay_Profit program restructured to separate the parsing of input from the core application

```
Procedure Sale_Pay_Profit (days: integer;
                cost: float; var sale: int_array;
                var pay:  float; var profit: float;
                  process: boolean);
var i, j, total_sales: integer;total_pay: float;
begin
    sale := Read_Input(days, sale);
    if process = True then begin
      total_pay := Compute_Pay(days, sale);
      total_sale := Compute_Sale(days, sale);
      pay := Compute_Avg_Pay(total_pay,
          days);
      profit := Compute_Profit(total_sale,
          cost);
    end;
end;

Function Read_Input(days:integer;
    var sale: int_array): int_array;
var i: integer;
begin
    i:=0;
    while i < days do begin
        i := i + 1;
        readln(sale[i]);
    end;
    return (sale);
end;
```

```
Function  Compute_Pay(days: integer;
    var sale: int_array): float;
var j: integer;
begin
    total_pay := 0;
    for j := 1 to days do
    begin
        total_pay := total_pay + 0.1 * sale[j];
        if sale[j] > 1000 then
            total_pay := total_pay + 50;
    end;
    return (total_pay);
end;

Function Compute_Sale(days: integer;
    var sale: int_array): floatt;
var j: integer;
begin
    total_sale := 0;
    for j := 1 to days do
    begin
        total_sale := total_sale + sale[j];
    end;
    return (total_sale);
end;
```

```
Function Compute_Avg_Pay
    (total_pay: float; days: integer): float;
var avg_pay:float;
begin
    avg_pay := total_pay / days + 100;
    return (avg_pay);
end;

Function Compute_Profit
    (total_pay: float; days: integer): float;
var profit: float;
begin
    profit := 0.9 * total_sale - cost;
    return (profit);
end;
```

Figure 3: Sale_Pay_Profit program restructured to have an object-based design.

2.2 Restructuring process model

The restructuring process in our simulated scenario may be abstracted as a sequence of the following steps:

1. Identify each task.
2. Identify the ‘depends on’ relations between each pair of tasks.
3. Determine the order in which the tasks are to be restructured.
4. Restructure each task:
    a) Identify the computations that influence the given task.
    b) Collect all these computations in a new module and create a function call to the new function in the appropriate position of the original procedure.

Each step in the above model may be considered to be independent of the other steps. Thus each step is a candidate for automated support.

2.3 Transformations for restructuring

In this thesis, we develop a transformation to restructure an individual task (Steps 4.a and 4.b). We assume that a subset of statements representing a task, the dependencies between the tasks, and the order in which the tasks should be restructured have already been identified.

The input to the restructuring process is a task, therefore, we must determine how to represent a task. A programmer identifies a task, by analyzing the statements of a function, so it is natural to define a task as a set of statements. Also, we want our transformation to be useful in interactive tools. To reduce the overhead of the user of such interactive tools, (s)he should only have to select a very few crucial statements relevant to a given task and our transformation should determine the rest of the statements that belong to that task. We refer to the crucial statements as a set of *'seed'* statements and the word *'task'* defines all the statements related to a seed.

We now have an outline for our transformation: It first collects all the statements related to an input seed. This collection of statements identifies a task. Next, it extracts and creates a new function for that task. The transformation has two distinct steps, each of which can be defined as a primitive transformation:

- **TUCK: Identify the entire task from the input seed**

  Given an initial seed − a set of statements − the transformation TUCK creates a task containing all the statements related to that seed.

- **SPLIT: Extract the task**

  The transformation SPLIT extracts a task and generates a new function for it. In doing so, this transformation should not change the external behavior of the function.

To generate good solutions, TUCK should take into account the context[1] in which the statements of the seed are placed in the function. If TUCK determines several contexts for a seed, it should create a solution for each context. TUCK must also deal with statement interleaving. If statements not related to the seed are interleaved with statements related with the seed then TUCK should not collect the non-relevant statements. For example, in Figure 1, the computations of total_pay and total_sale are interleaved. They should be separated on restructuring.

[1] The meaning of 'context' will be clarified in Chapter 4 where a precise definition of TUCK is introduced. For now, a context can simply be viewed as a region of the function within which TUCK may find the statements related to a seed.

3. Background

TUCK and SPLIT both make use of control flow graphs (CFG), program dependence graph (PDG), and slicing to computes their results. For this thesis to be self-contained, we only present the definitions necessary for our discussion and our algorithms. For more details the reader is referred to other literature [Aho86, Ferrante87, Muchnick97, Ottenstein84, Tips95, Weiser84].

We define the notion of a conditional path for both the CFG and the PDG. This notion is not discussed in previous literature and is necessary to exploit some important properties of both the graphs.

In this chapter, we merely present the definitions and some explanations to clarify the definitions. The intuition behind the definitions are introduced in the next chapter when we introduce our transformation. We discuss CFG, PDG, and slicing in Sections 3.1, 3.2, and 3.3, respectively.

3.1 Control flow graph (CFG)

Definition: A ***control flow graph*** of a module M, denoted ***CFG(M)***, is a graph G = <V, E> where V is a set of nodes in CFG(M) and E is a set of edges in CFG(M). A edge between two CFG nodes v_1 and v_2, denoted $v_1 \rightarrow v_2$, represents a control flow from v_1 to v_2. A CFG edge may be of one of three types: Always, True or False. In addition, a CFG has a unique start node, Start(M), and a unique end node, End(M), such that there exists a path from the Start(M) to every other node and there is a path from every other node to End(M). The CFG of Sale_Pay_Profit is shown in Figure 4.

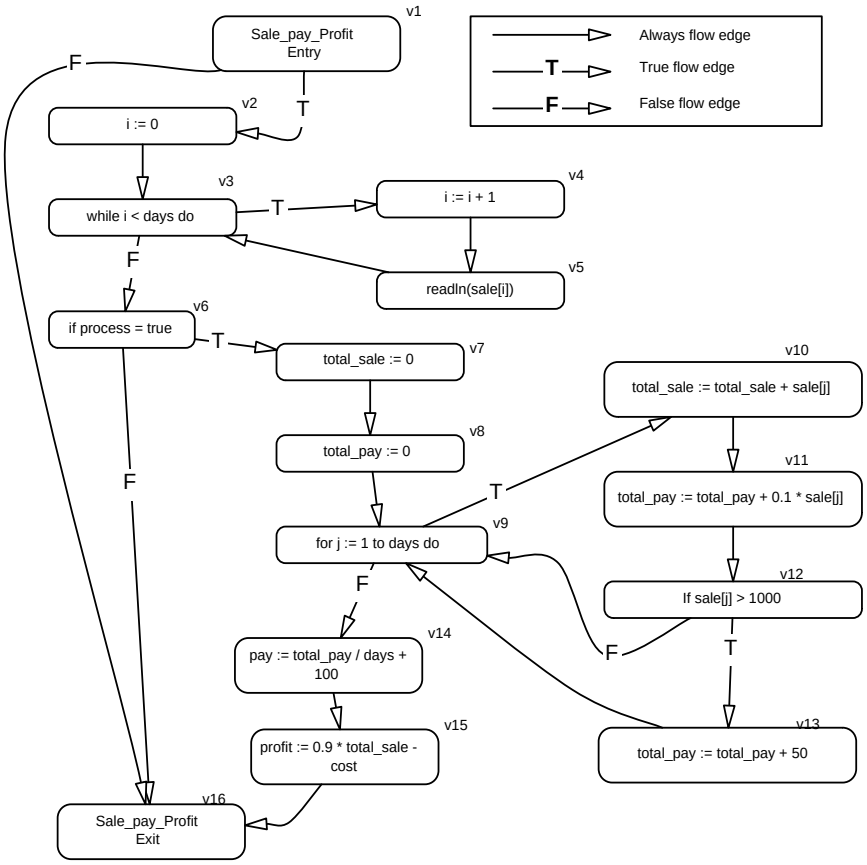


Figure 4: CFG for Sale_Pay_Profit procedure of Figure 1.

Definition: A ***path*** in CFG(M) from CFG node v_1 to CFG node v_n, denoted $\boldsymbol{v_1 \rightarrow^* v_n}$, is a sequence of zero or more CFG nodes that belong to CFG(M) such that $\forall i$, $1 <= i < n$, $v_i \rightarrow v_{i+1}$ is a CFG edge in CFG(M).

The first node and the last node in the sequence as well as all the intermediate nodes are said to be in the path.

Definition: A *conditional path* in CFG(M) from CFG node v_1 to CFG node v_n, denoted $\boldsymbol{v_1 \rightarrow^*_c v_n}$, is a CFG path $v_1 \rightarrow^* v_n$ such that $v_1 \rightarrow v_2$ is a CFG edge in CFG(M) and this edge is type c where $c \in$ {True, False}.

The tag ***c*** on the edge represents the type (True or False) of the first edge in the path. The types of the subsequent edges in the path do not matter in this definition. There exist two possible instantiations for a conditional path in CFG(M) from v_1 to v_n: (1) $v_1 \rightarrow^{*}_{True} v_n$ or (2) $v_1 \rightarrow^{*}_{False} v_n$.

3.1.1 Post-dominator relationships in CFG

Definition: ***Post-dominator*** & ***immediate post-dominator*** in CFG(M): A CFG node *w* is the post-dominator of a CFG node *v* if path from *v* to End(M) contains *w*. Furthermore, *w* is the immediate post-dominator of *v*, denoted ***ipdom(v)***, if every post-dominator of *v*, other than *v* and *w*, also post-dominates *w*. Figure 5 gives the immediate post dominator relation of CFG in Figure 4.

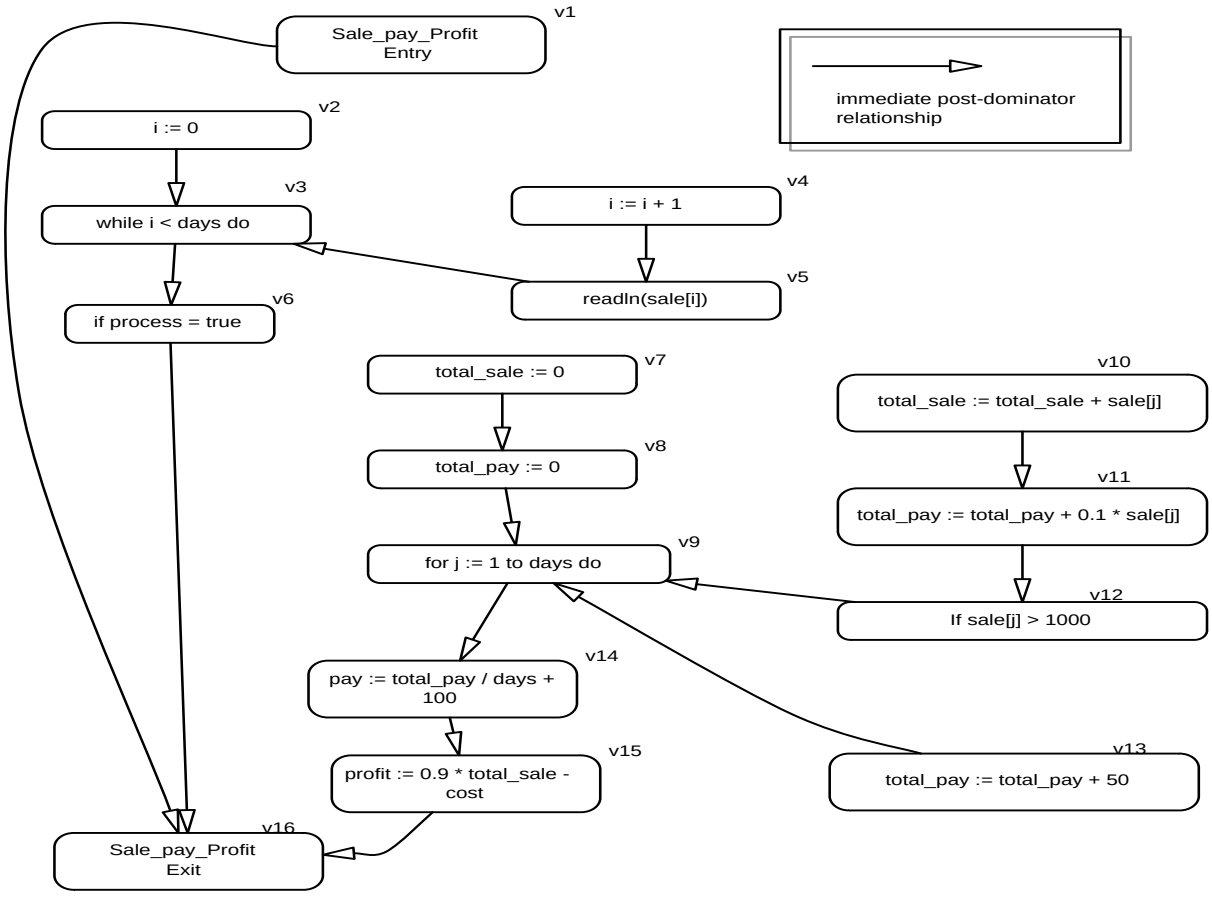


Figure 5: Immediate post-dominator relation: From the immediate post-dominator tree it is possible to identify the ipdom of any CFG node.

3.2 Program dependence graph (PDG):

Definition: A ***program dependence graph*** of module M, denoted ***PDG(M)***, is a graph P = <N, E> where N is a set of PDG nodes in PDG(M) and E is a set of PDG edges in PDG(M). A PDG edge between n_1 and n_2, denoted $\boldsymbol{n_1 \Rightarrow n_2}$, represents the data or control dependence between the statements of M. For further details on the exact meaning of PDG dependencies, we refer the user to [Ottenstein84, Weiser84]. A PDG edge may be one of three types: Data, True or False. Figure 6 illustrates the PDG of Sale_Pay_Profit.

Because the rest of our definitions depends only on the control dependence of a PDG, we have added a distinct definition to easily refer to such edges.

Definition: A ***control Edge*** from n_1 to n_2, denoted $\boldsymbol{n_1 \Rightarrow_c n_2}$, is a PDG edge where $c \in$ {True, False}

Definition: A ***control path of PDG(M)*** from PDG node n_1 to PDG node n_n, denoted $n_1 \Rightarrow^* n_n$, is a sequence of PDG nodes that belong to PDG(M) such that $\forall i, 1 <= i < n$, $n_i \Rightarrow_{c_i} n_{i+1}$ is a control edge in PDG(M) where $c_i \in$ {True, False}. A *control path* is also reflexive, therefore a PDG node n belongs to $n \Rightarrow^* n$.

Definition: A ***conditional control path in PDG(M)*** from PDG node n_1 to PDG node n_n, denoted $\boldsymbol{n_1 \Rightarrow_c^* n_n}$, is a control path , $n_1 \Rightarrow^* n_n$ such that $n_1 \Rightarrow_c n_2$ is a PDG edge in PDG(M).

The tag ***c*** on the edge represent the type (True, or False) of the first edge in the control path. The type of the other edges is not relevant in this definition. Therefore, there exists two possible instantiations for a conditional PDG control path from n_1 to n_n:

(1) $n_1 \Rightarrow^*_{True} n_n$ or (2) $n_1 \Rightarrow^*_{False} n_n$.

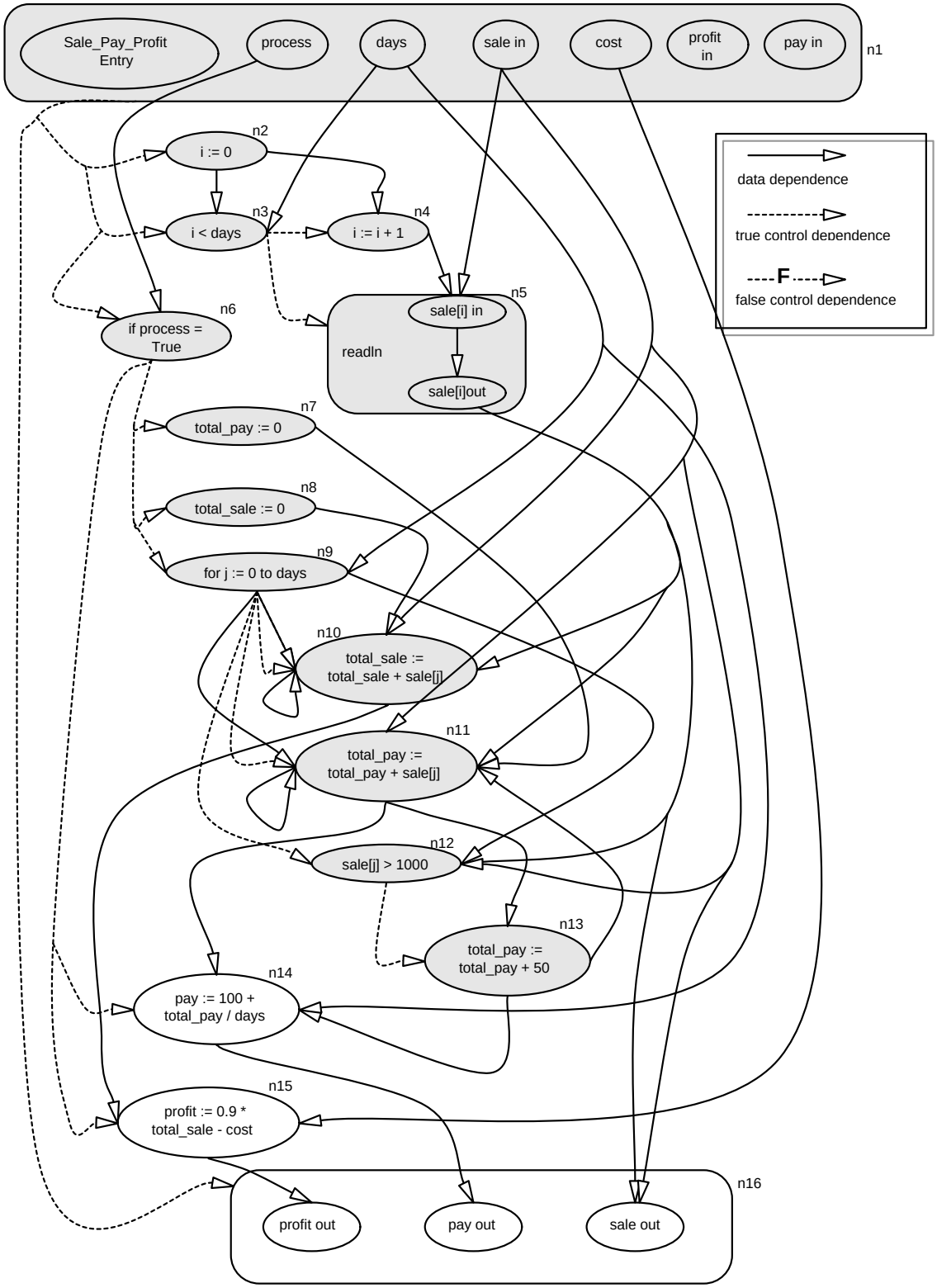


Figure 6: PDG for Sale_Pay_Profit program. The shaded nodes represent the slice on node n13.

3.2.1 Control relationships in PDG

This section introduces four definitions that use the notions of control path and conditional control path. The first definition creates a set of the PDG nodes that have a path to a given PDG node through only one of their control dependencies. Therefore, if a node *p* has a control path to another node *q* through its True branch then *p* should not have a control path to *q* through its False branch. It is only possible for a node to reach another node through both of its control path in the presence of `goto` statements. The PDG in Figure 7 illustrates such condition.

Definition: A ***definite control node p*** of another PDG node n, denoted ***dc-node(n)***, is an ordered pair *(c, p)* such that *(c, p)* ∈ *dc-node(n)* iff *c* ∈ {True, False} & if $p \Rightarrow_c^* n$ then $\sim(p \Rightarrow_{\sim c}^* n)$.

- ~*c* means the complement of *c*, thus in our case if *c* = True then ~*c* = False and reciprocally if *c* = False then ~*c* = True.
- $p \Rightarrow_c^* n$ is used as a predicate. It states that there exist a path from *p* to *n* through the *c* control dependence edge of *p*
- $\sim(p \Rightarrow_{\sim c}^* n)$ is also a predicate. It states that there should not exist a conditional control path from *p* to *n* through the ~*c* control dependence edge of *p*.

An element in dc-node(n) definitely controls n in the following sence: If *(c, p)* ∈ *dc-node(n)* then in CFG(M) there does not exist a path $p \Rightarrow_{\sim c}^* n$ and there exist a path $p \Rightarrow_c^* n$

Definition: A ***definite control node p of a set*** *of PDG nodes S*, denoted ***dc-set(S)***: is an ordered pair *(c, p)* such that *(c, p)* ∈ *dc-set(S)* iff $\bigcap_{\forall n \in S}$ dc-node(n).

All elements of dc-set(S) definitely control every element of S. It is quite possible for two elements belonging to dc-set(S) to also definitely control each other. That is $(c_1, n_1) \in$ *dc-set(S) and* $(c_2, n_2) \in$ *dc-set(S)* and $(c_1, n_1) \in$ *dc-node(*n_2*) and* $(c_2, n_2) \in$ *dc-node(*n_1*)*. The following

definition restrict this case by removing such definite control nodes. In other word, from the dc-set, it selects the nodes which have a control path between each other, furthermore, the control path must exist only in one direction. Therefore, if two nodes *n* and *m* belong to a given dc-set either *n* may have control path to *m* or *m* may have a control path to *n* but not both may be true. This is equivalent to an exclusive OR (XOR) between the control dependencies between *n* and *m*. The PDG in Figure 7 shows an illustration of the definition.

Definition: A ***single definite control*** *node p of a set of PDG nodes S*, denoted ***sdc(S)***, is an ordered pair *(c, p)* such that *(c, p)* ∈ *sdc(S)* iff *(c, p)* ∈ dc-set(S) &

∀ *(q, n)* ∈ dc-set(S), $p = n$ or $(((p \Rightarrow_c^* n) \& \sim(n \Rightarrow_q^* p)) \text{ or } (\sim(p \Rightarrow_c^* n) \& (n \Rightarrow_q^* p)))$.

Definition: The ***nearest single definite control*** *node p of a set of PDG nodes S*, denoted ***nsdc(S)***, is an ordered pair *(c, p)* ∈ sdc(S) such that PDG node *p* does not have a control path to any other sdc(S).

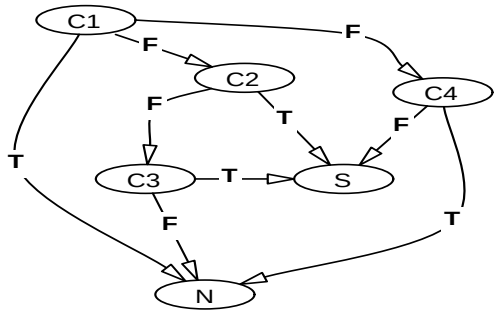


Figure 7: This sample PDG illustrates the different properties defined above.

Definite control node: dc-set({S}) = { (False, C1), (True, C3), (False, C4)}. C2 is not a dc-set({S}) because it can reach S through both of its control path True and False.

Single definite control: sdc({S}) = {(False,C1)}. In this case, C3 and C4 are not sdc({S}) because they do not control each other even though both reach S from one of their control path.

3.3 Slice

Definition: A ***slice*** , denoted ***slice(P, n),*** is the backwards reflexive transitive closure of the PDG edges with respect to a PDG node *n* in PDG *P*. In Figure 6, the shaded nodes represent the slice on n13.

4. Restructuring transformation

This chapter presents the main contribution of this thesis, a context-sensitive transformation for software restructuring. As stated in Chapter 2, our transformation is composed of two primitive transformations, TUCK and SPILT. The two transformations are presented in Section 4.1 and 4.2, respectively. Section 4.3 proves that SPLIT transformation does not change the semantics of the original programs. Since the TUCK transformation does not modify programs, it is tautologically true that TUCK does not alter the semantics of those programs. Section 4.4 presents our restructuring transformation by composing TUCK and SPLIT. That our transformation preserves the program's semantics follows from the proof of its components.

Our transformation uses control flow graph (CFG) and program dependence graph (PDG) introduced in Chapter 3. We assume the existence of the CFG and the PDG for the program being restructured.

4.1 TUCK

The input to TUCK is, (a) a CFG and (b) a set of CFG nodes representing the seed. Its output is a solution for each restructuring context. The restructuring context is a set of CFG nodes within which the computations related to a seed are identified. Therefore, the output of TUCK is a set of ordered pairs where the first element is a set of CFG nodes corresponding to a context and the second element is a subset of the first element identifying the set of CFG nodes representing an entire task within that context.

Signature[2]:

TUCK: CFG → ***P***(CFG node) → ***P***(***P***(CFG node), ***P***(CFG node))

Input: (1) a CFG to be restructured

(2) a set of CFG Nodes representing the initial seed.

Output: a set of ordered pairs where the first element represents a context and the second element a task within that context.

4.1.1 Intuitive reasoning behind TUCK

The initial work of TUCK is to determine all the different contexts associated to an input seed. For each context, it identifies only the computations relevant to the seed within that context. Slicing is a convenient method to extract the computation relevant to specific statements (or a seed), thus we need to find a way to define a context.

A parallel with natural languages can help. As a word might have different meaning depending on its context, (i.e., as we read a phrase, a paragraph or an entire passage containing a word, we could attach different meanings to that word depending on each of its contexts), a computational task may also have different meanings as we abstract it out to its different contexts (i.e., within its block, its module, its file, the entire program). Just as natural languages have the concepts of punctuation, paragraph, section, chapter, volume to delimit a region from which to associate a context to a word, we need to define or translate such concepts to the world of programming.

A *'context'* should be a closed region of statements which may influence or be influenced by the input seed. We also need to keep in mind the goal of the overall restructuring transformation, which is to identify potential new modules. In procedural languages, modules have

[2] We use ***P*** to mean power set.

a single entry. So, a programming *'context'* should have a similar property. A single definite control (sdc) of the input seed S is a node from which it is possible to reach every node in the input seed along one conditional control path but not the other. Thus we can utilize a sdc to create a context.

We now need to identify a single entry region for each element of the sdc(S) such that it contains the seed. To obtain accurate results, the region should be as small as possible. We formally define the concept of minimal region of a single definite control node, called restructuring context, in the next section. A restructuring context creates a boundary around the input seed S for an element of a sdc(S). Each element of sdc(S) provides a different context for the restructuring and the nsdc(S) determines the smallest restructuring context of an input seed.

4.1.2 Formal definition and properties of TUCK

In the following theorem and definitions, we assume: ***(c, p)*** ∈ ***sdc(S)*** where *S* is the input seed.

Theorem: Nodes that are directly control dependent on the *c* condition of *p* in the PDG form a simple path in the post dominator tree of the CFG [Ferrante87].

Since the nodes directly controlled by *p* with a control dependence of type *c* form a simple path in the post-dominator tree, we can create a sequence with these nodes.

Definition: From the above theorem, we can define a ***control sequence*** of nodes d_1 through d_n, denoted $[d_1, \ldots, d_n]$, where d_i, .., d_n form a simple path in the post-dominator tree and $\boldsymbol{p \Rightarrow_c d_i}$. We know that such *c* exists since *p* belong to sdc(S).

Figure 8 identifies a control sequence in the PDG of Sale_Pay_Profit.

Definition: A ***minimal control sequence*** of nodes d_i through d_j that controls some node of *S*, denoted seq(S), is a sub-sequence of a control sequence for an element in sdc(S) where d_i has the

smallest *i* in the control sequence [d_1, …, d_n] for which there exists a path $d_i \Rightarrow^* s$ for some $s \in S$, and d_j has the largest *j* in the control sequence [d_1, …, d_n] for which there exists a path $d_j \Rightarrow^* s$ for some $s \in S$.

Figure 8 illustrates the minimal control sequence definition in the PDG of Sale_Pay_Profit.

Definition: A ***restructuring context***, ***rc(c,p)***, where ***(c,p)*** is a single definite control of seed *S*, is the set of all the nodes *m* that belong to the PDG nodes of PDG(M) such that

$m \in rc(c, p)$ iff $(c,p) \in$ sdc(S) and $d \in$ seq(c, p) and $d \Rightarrow^* s$, where $s \in S$.

Figure 8 identifies such node in the PDG of Sale_Pay_Profit.

The important properties of rc(c, p):

- **Single Entry**:

  rc(c, p) defines a flow graph with single entry since there exists only one direct entry point of rc(c, p), which is the first node in the minimal control sequence (by definition of rc(c, p) & seq(S)).

- **Single Exit**:

  There might be several nodes from which the flow exits rc(c, p) but they all flow to the immediate post dominator of the last node in the minimal control sequence (by definition of rc(c, p) & seq(S)).

- **Closed**:

  The single entry and single exit properties ensure that rc(c, p) is closed in the following sense: $\forall q \in$ rc(c, p) & $\forall$m $\notin$ rc(c, p)

  if $m \rightarrow q$ is an edge in CFG(M) then q is the entry of rc(c, p) or

if $q \rightarrow m$ is an edge in CFG (M) then m is the ipdom of the last node in the minimal control sequence.

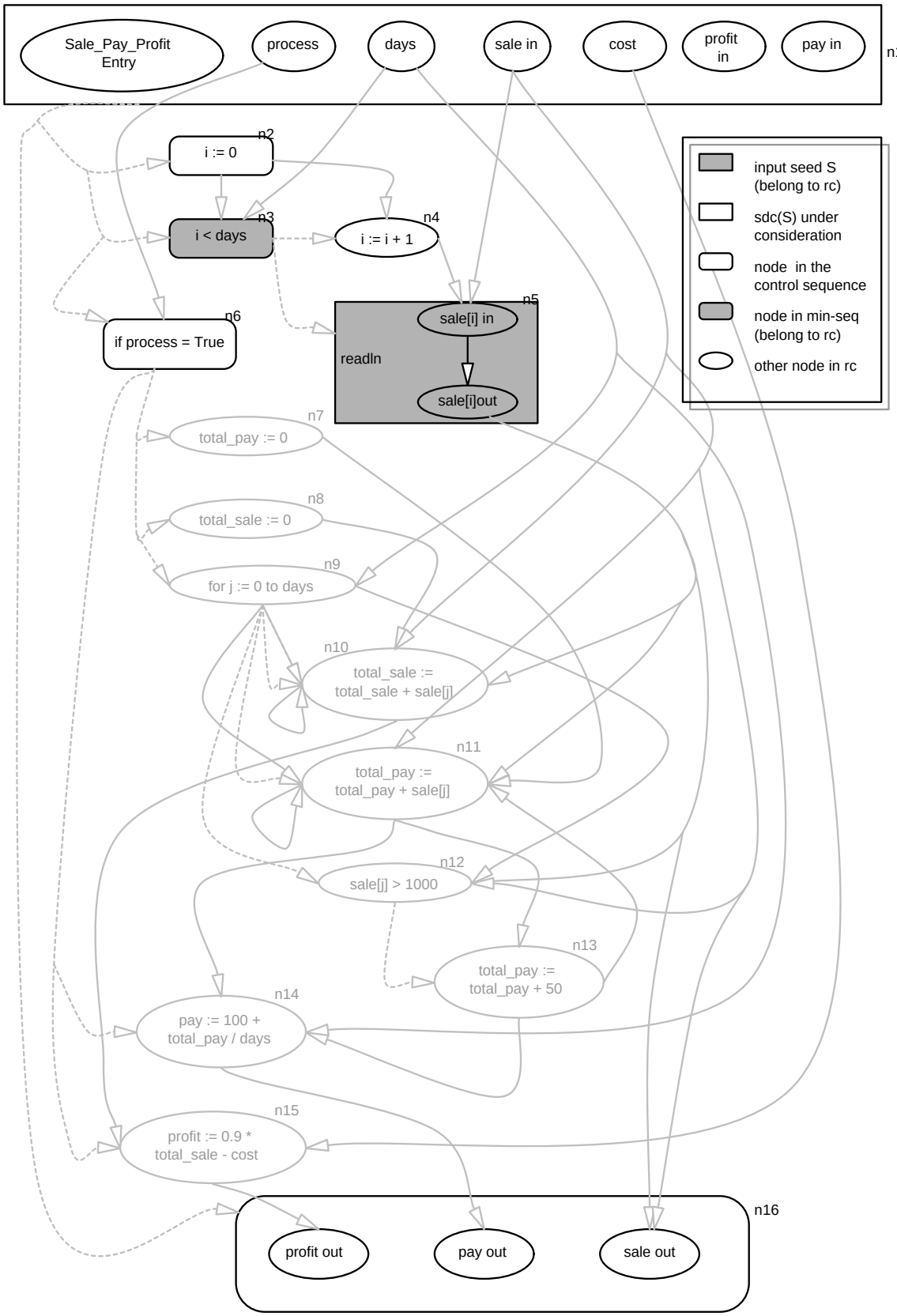


Figure 8: The restructuring context (rc) of (True, n1) with the seed n5 = {n3, n4, n5}.

Derived from its properties, rc(c, p) could be extracted as a separate module, a function-call CFG node to the new function could be created for rc(c, p) as well as the necessary CFG edges to maintain the identical control flow. Extracting rc(c, p) into a new module would not modify the semantics of the original program since neither the control flow nor the data flow[3] have changed. Therefore, the semantics of the inter-procedural PDG would be identical. From this observation, we can treat rc(c, p) as if it was a separate module without actually creating the new module for rc(c, p). We use this fact to avoid dealing with inter-procedural CFG and PDG which add unnecessary complexity to the proof (for SPLIT).

Lemma: Using the above observation, a slice contained within rc(c, p) preserves all the properties of a regular slice since rc(c, p) can be extracted as a separate module. The equivalent to the slice on a PDG node *n* within rc(c, p) is:

slice(PDG(M), n) ∩ rc(c, p)

4.1.3 TUCK: Algorithm

```
Input:  P: CFG(M)   /* CFG for module M */
        S: P(CFG-Nodes)= set of CFG-Nodes  /* Input seed */
```

Processing of TUCK:

```
sol = ∅
all-sol = ∅
for every (c, m) ∈ sdc(S) do
     for every n ∈ S do
          sol = sol U (slice(P, n) ∩ B((c,m)))
     end
     all-sols = all-sols U (B((c,m)),sol)
end
```

```
Output: all-sols: P(P(CFG-Nodes),P(CFG-Nodes))  /*  set of ordered
        pairs where the first element represent a context and the
        second identifies a subset of nodes consisting of an
        entire task for S within the context */
```

[3] Eventually, we also must introduce parameters to the new function to maintain the data flow coming in and out of the new function equivalent to the data flow of the original module.

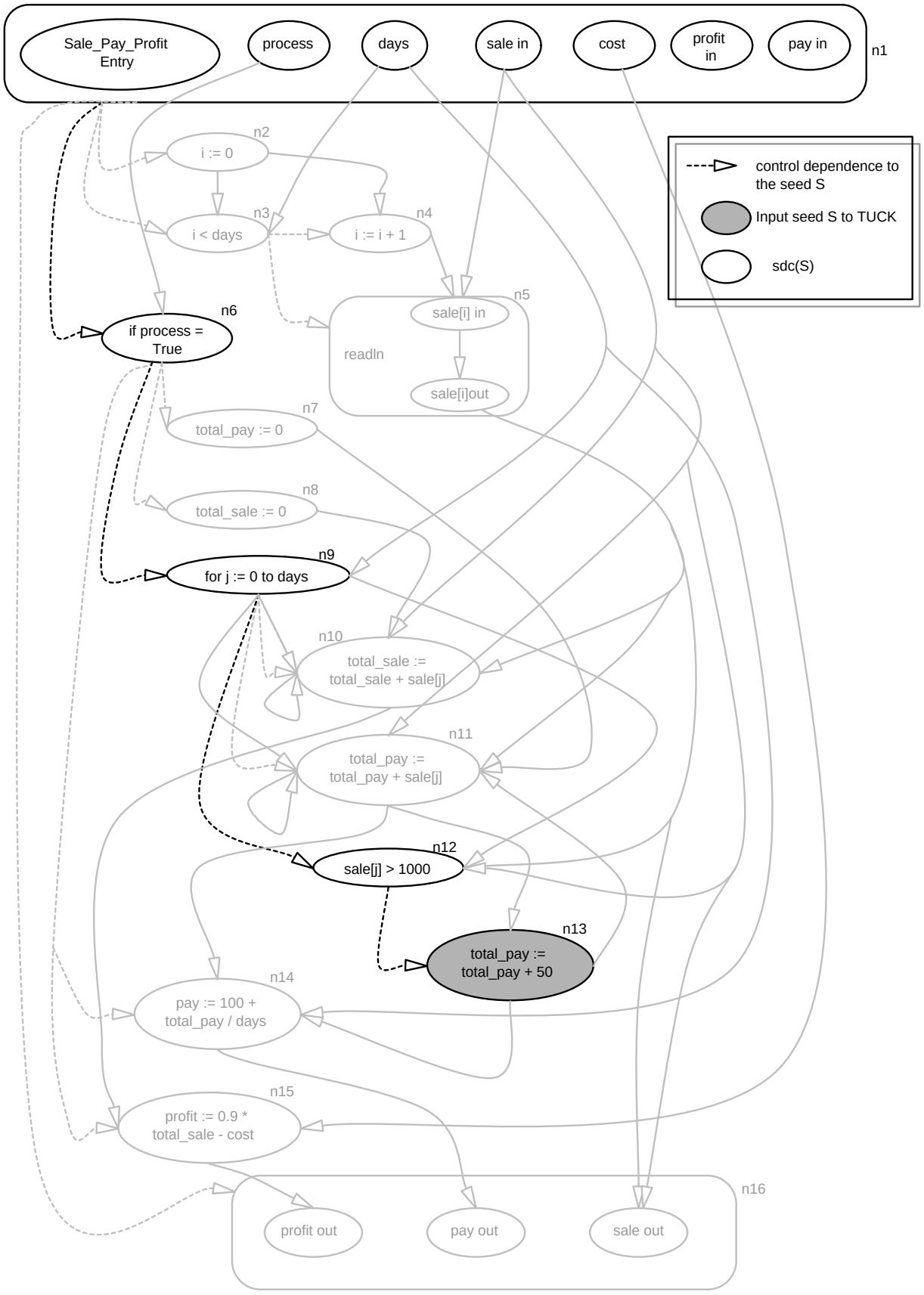


Figure 9: TUCK first computes the single definite control nodes (sdc) for the initial seed, n13. In this case, the seed is the singleton {n13} but generally it can be a set. The resulting set, sdc({n13}) = { n12, n9, n6, n1}.

The result of sdc({*n13*}), Figure 9, is the set of single definite control nodes {*(True, n12), (True, n9), (True, n6), (True, n1)*}. Each of the nodes *n12*, n9, n6 and *n1* along with their condition defines a restructuring context. The result of *rc*(*True, n12*) is a set only composed of one PDG node: *n13*, the initial seed. Thus, the result of TUCK within the context defined by *n12* would only identify the statement `total_pay:=total_pay+50`. This result seems redundant, nevertheless, it is not incorrect and may even be desirable under certain circumstances. On the other hand, such an example does not highlight most of the concepts of TUCK as well as the next transformation, SPLIT. Therefore we only show the result of TUCK on context defined by *n9* in Figure 10, followed by TUCK on context defined by n6 in Figure 11 and on context defined by n1 in Figure 12.

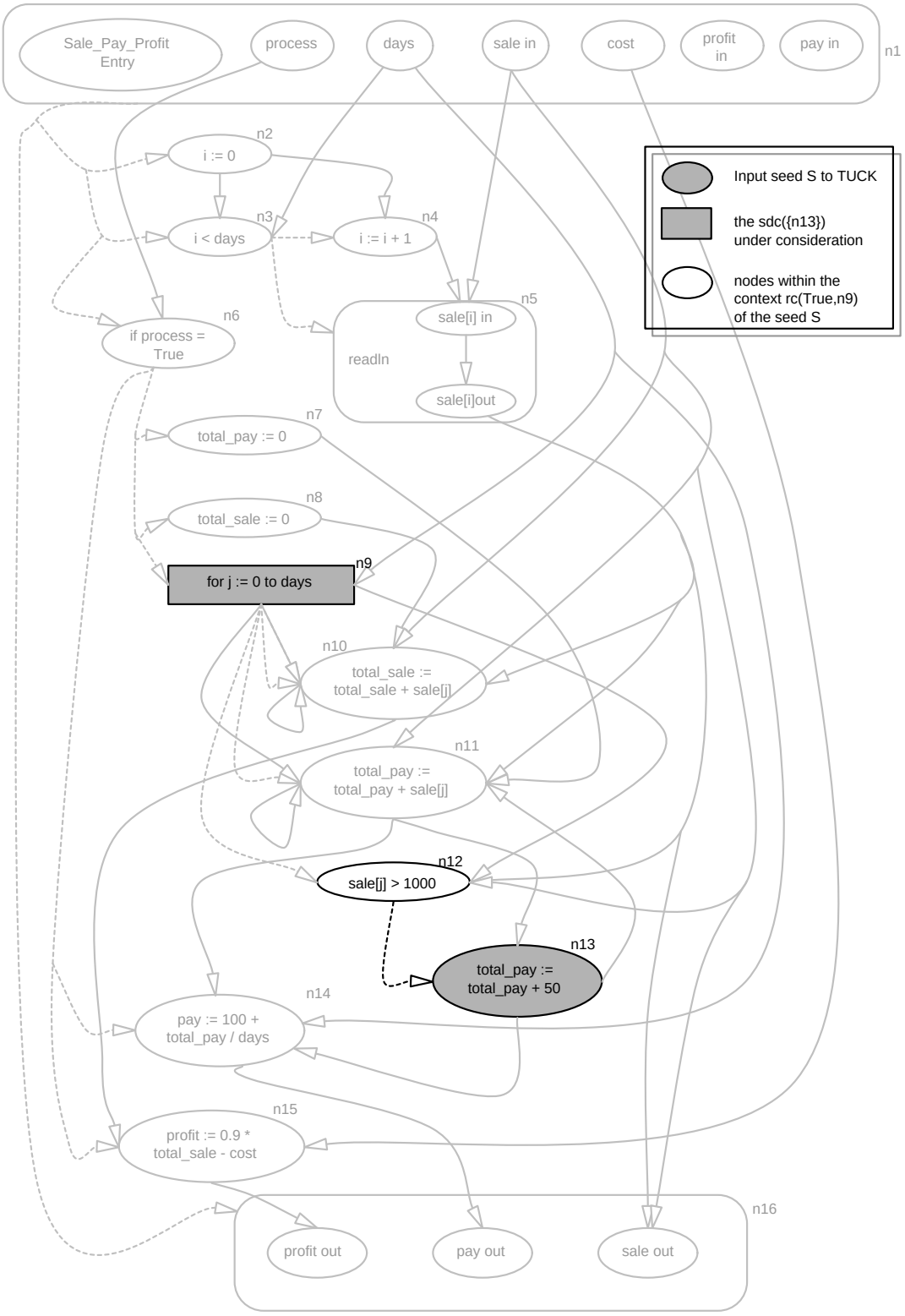


Figure 10: TUCK({n13}) within the restructuring context defined by n9. The result of the slice within this context is{ n12, n13}.

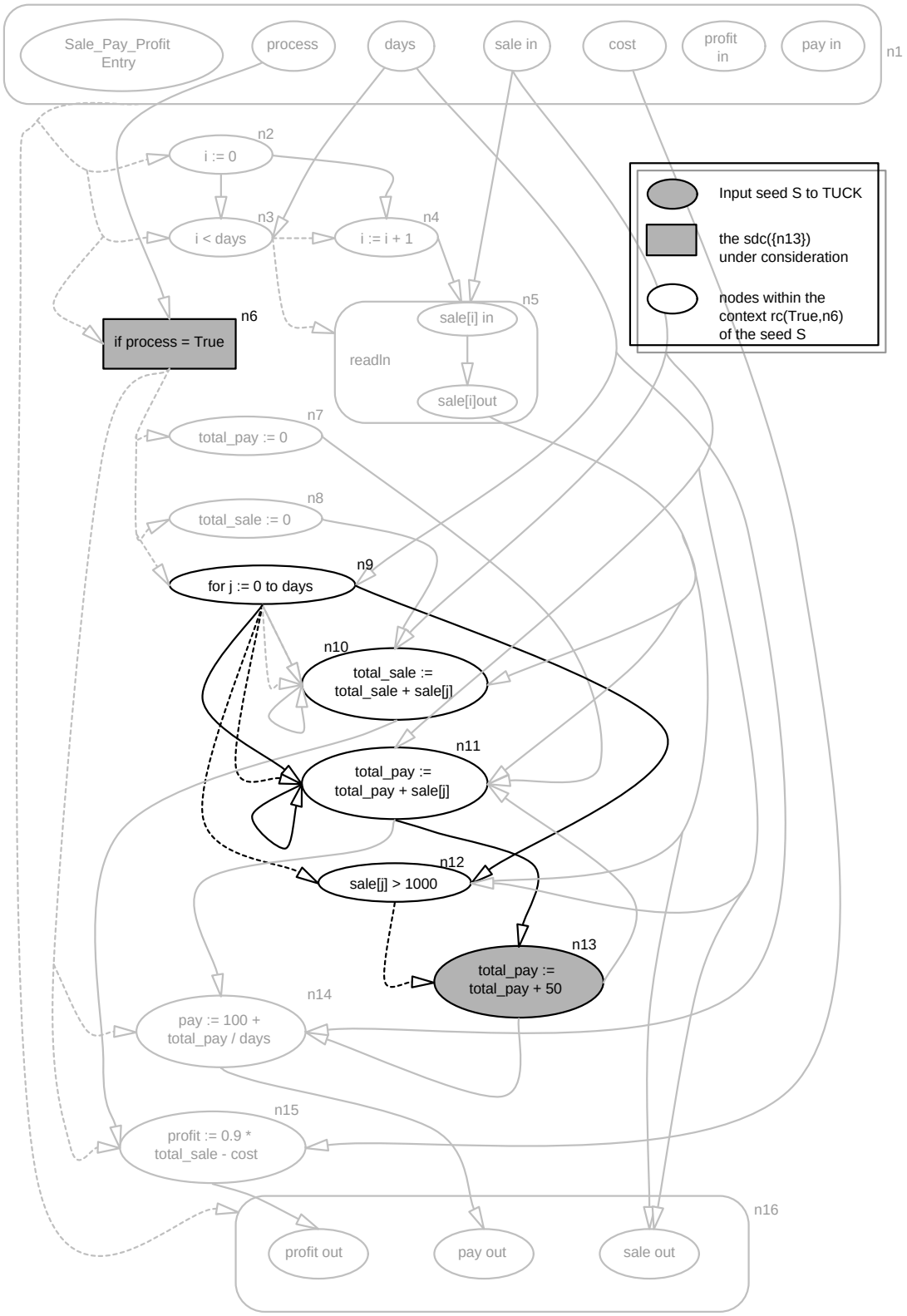


Figure 11: TUCK({n13}) within the restructuring context defined by n6. In this case the slice on n13 within rc(True, n6) is {n9, n11, n12, n13}

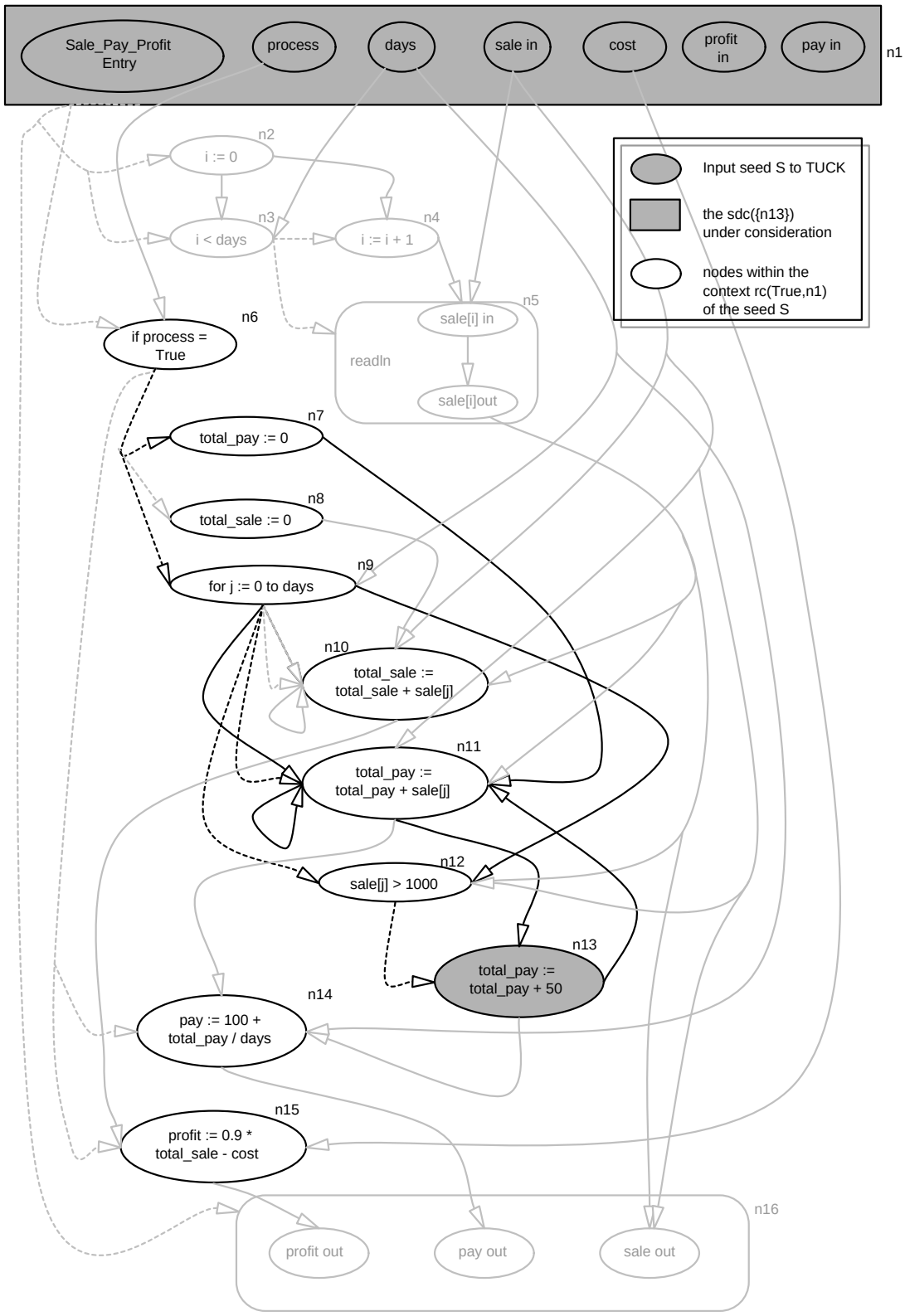


Figure 12: TUCK({n13}) within the restructuring context defined by n1. The slice on n13 within rc(True, n1) is {n6, n7, n9, n11, n12, n13}

4.2 SPLIT

The SPLIT transformation splits a single-entry, single-exit region into two regions. For our restructuring transformation to be useful, it is necessary that SPLIT preserves the semantics of the original module. SPLIT does the following, given a CFG(M) and a set of CFG nodes, that represent a task, it analyzes whether the task may be extracted from its context without affecting the semantics of the original module. If it can be, SPLIT separates the CFG in two, the first CFG is a task equivalent to the input task (which becomes a new module) and the second CFG is equivalent to the remaining tasks computed by of the original module. Also, to preserve the semantics of the program, SPLIT introduces a function call to the new CFG in the original function. We prove that SPLIT does not affect the semantics of the original program in Section 4.3. To avoid dealing with inter-procedural CFG & PDG in our proof, we have broken SPLIT in two steps. The first step modifies the original CFG but still keeps its results into one CFG. The second step uses the new CFG to extract the task into a new module. In this two-step approach, we prove that the first step, which changes the CFG does not modify the semantics. Most of the issues are handled by the first step. From the CFG resulting from step 1, it is easy to create a new module without affecting the semantics.

Signature:

SPLIT: CFG × ***P***(CFG node) → CFG × CFG

Input: (1) CFG of module M.

(2) a set of CFG Nodes X representing an entire task to be extracted

Output[4] (1) the original CFG from which X has been extracted

(2) a new CFG which computes the task X

[4] When the restructuring is not possible then the original CFG is unchanged and the second CFG is undefined

In the rest of this chapter we assume the following definition for B:

B = rc(c, p) where (c, p) = nsdc(X)

Definition: *IN(X, v).* Given a set of PDG nodes X and a variable *v*, ***IN(X, v)*** is true if there exists at least one definition of *v* outside X and that reaches a use of the variable *v* that belongs to X.

$\exists$ d $\notin$ X & d is a definition of *v* & $\exists$ u $\in$ X & d $\Rightarrow$ u

Definition: *OUT(X, v)*. Given a set of PDG nodes X, ***OUT(X, v)*** is true if there exists at least one definition of *v* that belongs to X and that reaches a use of the variable *v* that does not belong X.

$\exists$ d $\in$ X & d is a definition of v & $\exists$ u $\notin$ X & d $\Rightarrow$ u

Definition: Local. Variable *v* is '**Local**' to a set of PDG nodes X if (a) there is no dependence reaching any uses of *v* in X from a definition of *v* outside X, and (b) there is no dependence reaching any use of *v* outside X from a definition of *v* in X.

v $\in$ **Local**(X) iff ~OUT(X, v) and ~IN(X, v)

Definition: Value. Variable *v* is '**Value**' to a set of PDG nodes X if (a) it has only dependencies reaching uses of *v* in X from a definition of *v* outside X and (b) it does not have any definitions of *v* within X reaching uses of *v* outside X.

v $\in$ **Value**(X) iff ~OUT(X, v) and IN(X, v)

Definition: Outvar. Variable *v* is '**Outvar**' to a set of PDG nodes X if it has at least one definition of *v* within X reaching a use of *v* outside X.

v $\in$ **Outvar**(X) iff OUT(X, *v*)

4.2.1 Intuitive explanation of SPLIT

Let us assume that SPLIT tries to extract a set of PDG nodes X which constitutes an entire task. Let Y denote the tasks remaining in the original CFG. The SPLIT transformation will replace the subgraph B by the graph equivalent to X; Y or Y; X. SPLIT must ensure that this modification does not change the semantics of the original CFG. In the event that X and Y are totally

independent ($X \cap Y = \varnothing$), SPLIT may choose any order (X; Y or Y; X) without changing the semantics. The difficulty arises when X and Y are not independent ($X \cap Y \neq \varnothing$). This means that some computations are needed in task X as well as in the remaining tasks Y. In the general case, $X \cap Y \neq \varnothing$ would imply neither X; Y nor Y; X preserve the behavior of B. However we intend to convert X to a module with variables in Local(X) as its local variables, variables in Value(X) as its value parameters, and variables in Outvar(X) as its reference parameters. SPLIT can make use that fact, and it may duplicate computations modifying the local variables (Local(X)) and the value parameters (Value(X)) of X since these computations will be visible only within the syntactic scope of X and will not influence Y. The Table below exhaustively enumerates all the conditions determining the decisions of SPLIT. The conditions are not mutually exclusive. It states that if $Outvar(X) \cap Outvar(Y)$ is not empty then the tasks X and Y cannot be separated.
If $Outvar(X) \cap Value(Y) \neq \varnothing$ then only the ordering Y; X is permissible. The case for $Outvar(X) \cap Value(Y) \neq \varnothing$ is symmetric. However that $Value(X) \cap Value(Y) \neq \varnothing$ has no significance on the ordering. Since Local(X) identifies variables totally internal to X, such variables do not affect the decision taken by SPLIT. Thus the Local sets do not appear in the table

$A \cap B \neq \varnothing$	Outvar(X)	Value(X)
Outvar(Y)	Cannot be split	X; Y
Value(Y)	Y; X	X; Y or Y; X

The Table reads as follow: i.e., $Outvar(Y) \cap Outvar(X) \neq \varnothing$ ➔ Cannot be split.

As stated earlier, the conditions enumerated in the table are not mutually exclusive. It is likely that X and Y may satisfy more than one condition. SPLIT picks the most stringent recommendation. The recommendations conflict when $Outvar(X) \cap Value(Y) \neq \varnothing$ and $Outvar(X) \cap Value(Y) \neq \varnothing$. In that case, the CFG cannot be split. It is also possible for all the

intersections to be empty. In such case, SPLIT may choose any order between X and Y since none of the computations needed by X are used by Y, and vice versa.

4.2.2 SPLIT: Algorithm

Input: CFG(M): the CFG of module M
a set of CFG Nodes X representing a task to extract

Processing of SPLIT:

```
C = nsdc(X)
B = rc(C)
~X = B - X
Y = {slice(PDG(M), n) | n in ~X} ∩ B

/* Define ordering of X and Y upon their local, value and outvar
   variables */
Case outvar(X) ∩ outvar(Y) ≠ ∅
     OR (outvar(X) ∩ (value(Y) ∪ local(Y)) ≠ ∅ AND
         outvar(Y) ∩ (value(X) ∪ local(X))) ≠ ∅
     /* Restructuring fails, return same CFG In this situation
     the only possible restructuring is to extract the whole B
     into a new module. */
Case outvar(X) ∩ (value(Y) ∪ local(Y)) ≠ ∅
     /* ordering: Y;X */
     For all v ∈ outvar(X) ∩ (value(Y) ∪ local(Y))
          V' = append [v' := v] to V'
          X' = Create a copy of the CFG equivalent to X
          Y' = Create a copy of the CFG equivalent to Y where
               each v is substitute by v'
          Append Y' to V' -> G'
          Append X' to G' -> G''
          Replace B by G'' in CFG(M)
     End For
Otherwise
     /* outvar(Y) ∩ (value(X) ∪ local(X)) OR
        (value(X) ∪ local(X)) ∩ (value(Y) ∪ local(Y)) OR
        all intersections are empty */
     /* ordering: X;Y */
     For all v ∈ (outvar(Y) ∩ (value(X) ∪ local(X))) ∪
                 ((value(X) ∪ local(X)) ∩ (value(Y) ∪ local(Y)))
          V' = append [v' := v] to V'
          X' = Create a copy of a CFG equivalent to X where
               each v is substitute by v'
          Y' = Create a copy of the CFG equivalent to Y
          Append X' to V' -> G'
          Append Y' to G' -> G''
          Replace B by G'' in CFG(M)
     End For
End Case
Extract X' from G'' -> (O, T) where O and T are CFG's
```

Output: CFG O: the original CFG from which X has been extracted
CFG T: a new CFG for the task X

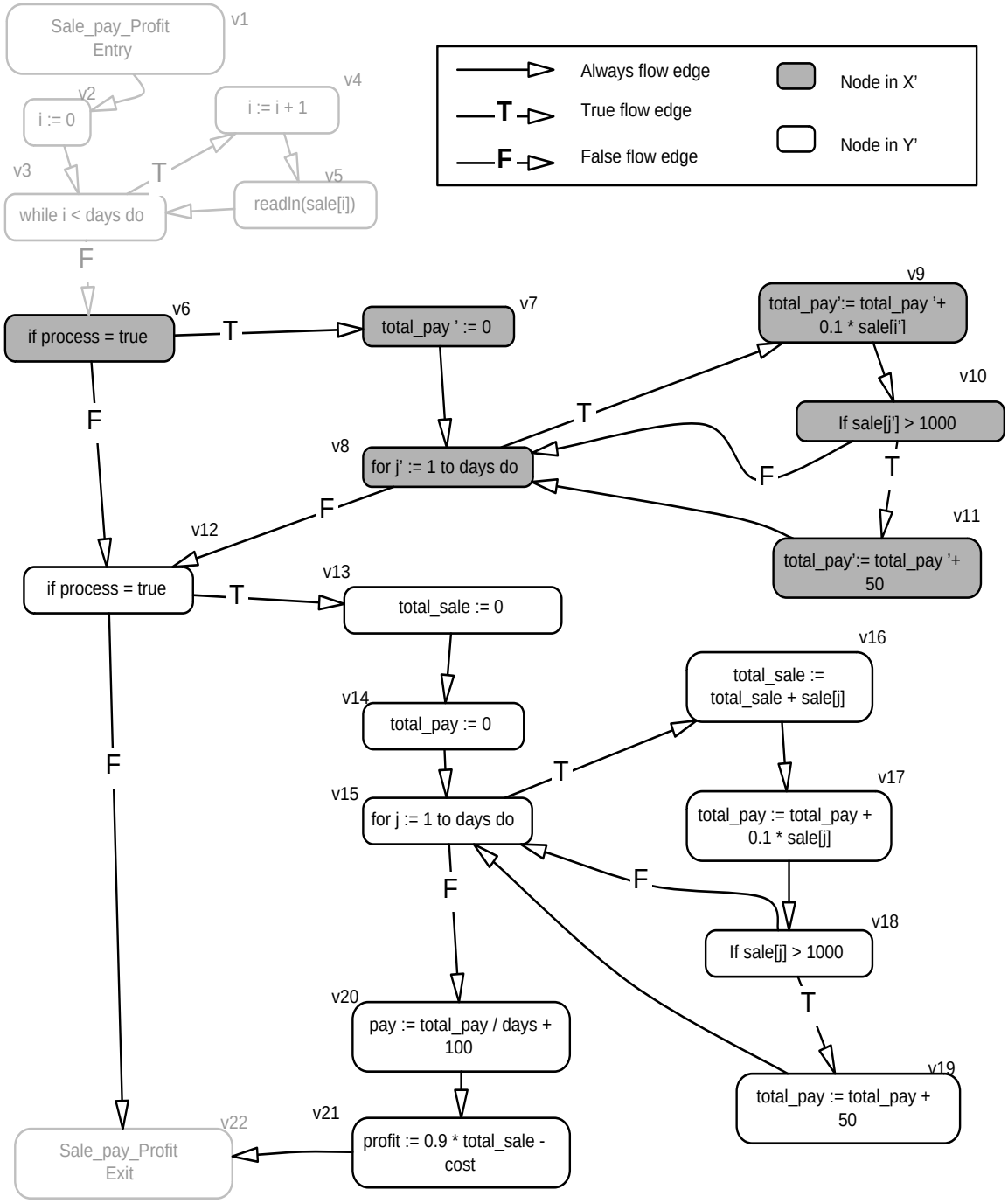


Figure 13: New CFG created by the first step of SPLIT. This is the solution for SPLIT using the result of TUCK from Figure 12.

4.2.3 Creating a new module using the new CFG

It is now straightforward to create the new module for the input task X. Since X' is identical to X, we can replace it by a call to the new module and put all the statement of X' in that new module. Every Value variable becomes a value parameter, each Local variable becomes a local variable of the new module and the Outvar variables become reference parameters.

4.3 Proof that SPLIT preserves the semantics of the original module

The step in our algorithm for which we need to prove that the semantics has not change is where B is replaced by G'' which is either V';X';Y' or by V';Y';X'. If we can prove that in the case where Outvar(X) $\cap$ (Value(Y) $\cup$ Local(Y)) $\neq \varnothing$ then V';Y';X' has the same semantics as B, then by symmetry, the result is also valid for the case of

Outvar(Y) $\cap$ (Value(X) $\cup$ Local(X)) $\neq \varnothing$ with the reverse ordering V';X';Y'.

Also, by their two definitions, v $\in$ Outvar(X) iff OUT(X,v) and implicitly (IN(X,v) or ~ IN(X,v)) and v $\in$ Value(X) iff ~OUT(X,v) and IN(X,v). This means that semantics of Value does not add any constraint not taken care of by the semantics of Outvar, since Value only deals with definitions that reach in and Outvar deals with both, definitions that reach in and out. Therefore, if the semantics of the original module remains unchanged with Outvar(Y) $\cap$ (Value(X) $\cup$ Local(X)) and the reordering V';X';Y' then such reordering will also keep the original semantics unchanged in the case of (Value(X) $\cup$ Local(X)) $\cap$ (Value(Y) $\cup$ Local(Y)).

We are then left to prove the following case: If Outvar(X) $\cap$ (Value(Y) $\cup$ Local(Y)) $\neq \varnothing$ then V';Y';X' has the same semantics as B.

In our proof, we make use of works on the semantics of PDG's and slices. Cartwright and Felleisen show that a PDG has the same the semantics as the program it represents [Cartwright89].

Venkatesh adds to the work of Cartwright and Feleisen and proved that slicing preserves the semantics of a program for the variable under consideration [Venkatesh91]. Because we use the semantics of slicing in the proof, we must impose a property on the input nodes X to SPLIT. The input X must have the following property:

Input property: If $X \subseteq B \subseteq PDG(M)$ then $(\forall p \in (B - X) \ \& \ p' \in X, \exists \sim(p \Rightarrow p'))$

There are two very important observations to make on this property:

1. $X = \{slice(n) \mid n \in X\} \cap B$. X is the equivalent to a slice on its nodes contained within B. (by definition of the input property)
2. TUCK creates an output with such property.

Lemma: If $Outvar(X) \cap (Value(Y) \cup Local(Y)) \neq \varnothing$ then V';Y';X' has the same semantics as B.

- ***Semantics of the new statements, v' := v***

V' is the introduction of statement `v' := v` (where `v'` is a new variable in each statement) for each variable in $Outvar(X) \cap (Value(Y) \cup Local(Y))$. These statement are inserted in the new CFG before X' and Y' (see algorithm). They do not affect the semantics of B since v' are new variables.

- ***Semantics of X and Y taken separately***

X is equivalent to a slice on its nodes contained within B (imposed by the input property). Since a slice preserves the semantics of the variables under consideration then the variables used within X will have the same values after the execution of X whether X is executed by itself or X is executed in B.

From the algorithm, Y too, is a slice. Therefore, whether Y is executed by itself or Y is executed in B, for the same input, its variables will have the same resulting values.

Thus, since $X \cup \sim X = X \cup Y = B$ then separately, X and Y execute all the computations of B. So, if we can order X and Y and keep the semantics of B then the execution of X and Y or the execution of B will give the same results.

- ***Combining X and Y while preserving the semantics: Ordering Y; X***

$\forall v \in Outvar(X) \cap (Value(Y) \cup Local(Y))$, we know that, If $n \in Y$ & n is a definition of v & $n' \notin B$ then $\sim(n \Rightarrow n')$. In other words, since a variable v is only Value or Local to Y then by definition, no PDG node defines the variable v within Y and creates a data dependence to a use of the same variable v outside B. On the other hand, to make sure that the node *n* does not reach any node $p \in X$, we can replace every occurrence of the variable v (which is defined by n) by its corresponding variable v' in Y. Since v' equals v before the execution of Y then v' has the same semantics as v. Now, since Y only modifies v' instead of v, we know that v will reach X with the same value as in the original function.

Therefore, the ordering V';Y';X' has the same semantics as B where V' is a set of statement v' := v, Y' is a copy of Y where the variable v has been substituted by v' and X' is the identical copy of X.

4.4 Composing TUCK and SPLIT

In this section, we show how TUCK and SPLIT may be composed to create a context-sensitive formal transformation. The TUCK transformation identifies a set of tasks and the SPLIT transformation separates a task into a new function. Thus, our transformation should first call TUCK followed by SPLIT. TUCK creates a set of tasks but SPLIT can only extract one of them. Therefore, after TUCK, a user or an automated tool must make a decision upon which result of TUCK creates the best solution for the restructuring. After the decision, SPLIT may be called with the selected result, and extract it into a new function.

Signature:

CSFT: CFG × ***P***(CFG node) → CFG × CFG

Input: (1) a CFG to be restructured

(2) a set of CFG nodes representing the initial seed.

Output[5]: (1) the original CFG from which the selected task has been extracted

(2) a new CFG which computes the selected task.

4.4.1 Context-sensitive formal transformation: Algorithm

Input :

```
G: CFG(M)
S: the Initial Seed = P(CFG nodes in CFG(M))
```

Processing of the restructuring algorithm:

```
TUCK-Sol = TUCK(G, S)
(C, T) ∈ TUCK-Sol      /* this step represents the decision
                           among the contexts*/
SPLIT-Sol = SPLIT(G,T)
```

Output:

```
SPLIT-Sol: CFG(M) from which T has been extracted
           a new CFG computing T
```

[5] When the restructuring is not possible then the original CFG is unchanged and the second CFG is undefined

5. Examples: Restructuring of Sale_Pay_Profit

This chapter illustrates the use of the restructuring transformation on the Sale_Pay_Profit function introduced in Figure 1. In Chapter 2, we have proposed two ways of restructuring this function. We will show how both the solutions can be derived using our transformation. First, we restructure Sale_Pay_Profit to separate the input parsing from the main processing of the program, then we show how to achieve the restructuring of Sale_Pay_Profit to recover objects. In the first example, we present the application of the transformation through three illustrations. In the second example, where the transformation is applied several times, we only show the input to the transformation directly followed by its output. Also, the output of one step automatically becomes the input of the next step.

5.1 Example 1: Separate input parsing

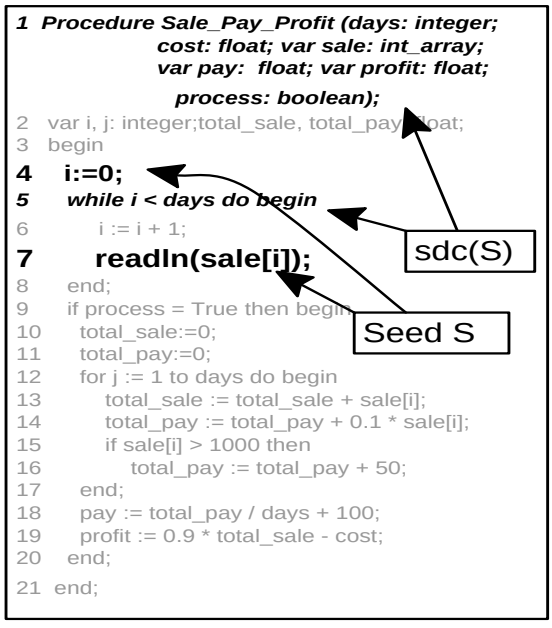


Figure 14 : TUCK first identifies all the single definite control of the seed S.

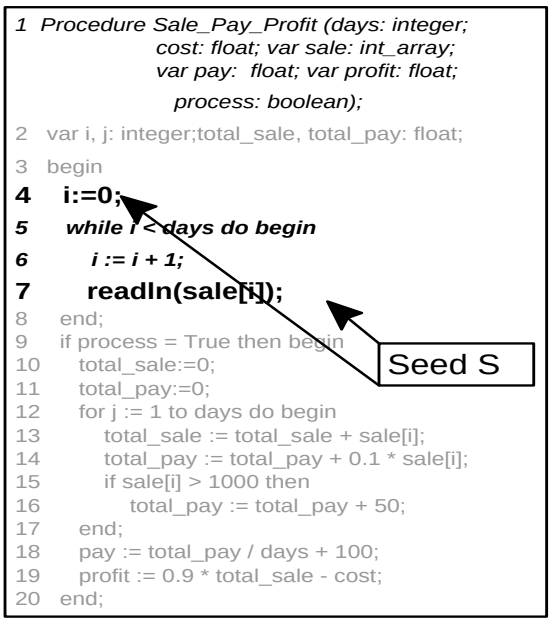


Figure 15: The sdc on line 1 is the context chosen to restructure the function. So the slice stays within the restructuring context defined by line 1. The highlighted statements are in slice(S).

```
Procedure Sale_Pay_Profit (days: integer;
    cost: float; var sale: int_array;
    var pay:  float; var profit: float;
    process: boolean);
var i, j: integer; total_sale, total_pay: float;
begin
    sale := Read_Input(days, sale);
    if process = true then begin
      total_sale := 0;
      total_pay := 0;
      for j := 1 to days do
      begin
        total_sale :=total_sale + sale[j];
        total_pay := total_pay + 0.1 * sale[j];
        if sale[j] > 1000 then
            total_pay  := total_pay + 50;
      end;
      pay := total_pay / days + 100;
      profit := 0.9 * total_sale - cost;
    end;
end;
```

```
Function Read_Input(days: integer;
    var sale: int_array): int_array;
var i: integer;
begin
    i := 0;
    while i < days do
    begin
        i := i + 1;
        readln(sale[i])
    end;
    return (sale);
end;
```

New function

Figure 16: The result of SPLIT to separate the input parsing from the processing of Sale_Pay_Profit.

5.2 Example 2: Restructure to have object design

In this example, we will create as many functions as possible to recover objects to move the program to an object-oriented design. In each of the figures, we use the same conventions. On the right side is the new function generated by the restructuring transformation, on the left side is the original function which has been modified with a function call to the new function replacing the task extracted. Also, the original function automatically becomes the input to the next restructuring transformations, thus, we identify the new seed and the element of the sdc(S) used to create the next result. Since the first step of this second example is identical to Example 1, we start from the output of Example 1.

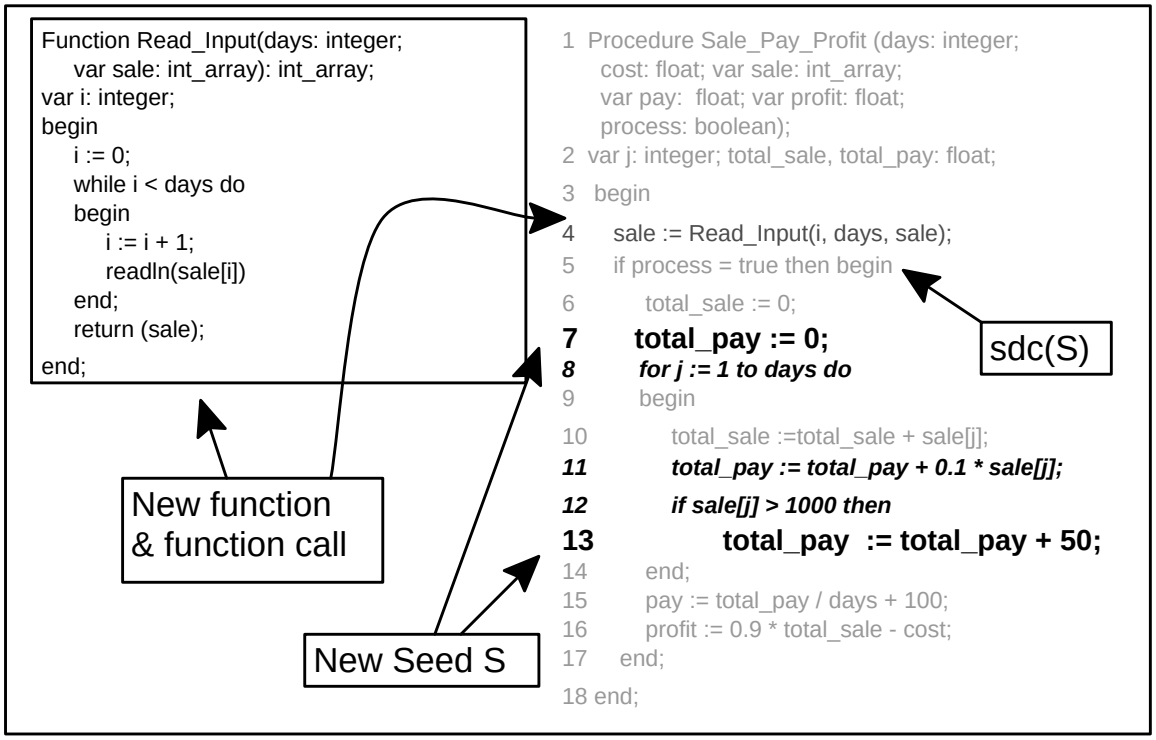


Figure 17: Read_Input is extracted from Sale_Pay_Profit. Now, lines 7 and 13 become the seed for the next restructuring. The sdc under consideration is on line 4 and the rc is defined by lines 7,8, 9, 10, 11, 12, and 13.

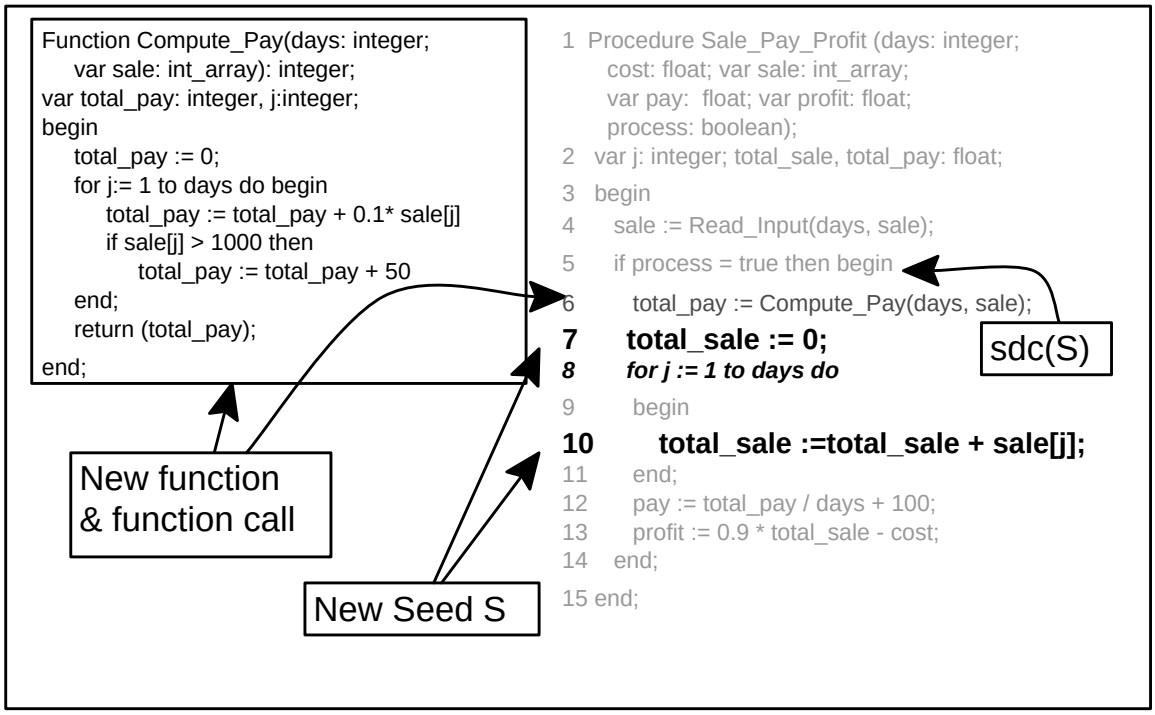


Figure 18: Total_Pay is extracted from Sale_Pay_Profit. Now, lines 7 and 10 become the seed for the next restructuring. The sdc under consideration is on line 5 and the rc is defined by lines 7,8, 9, and 10.

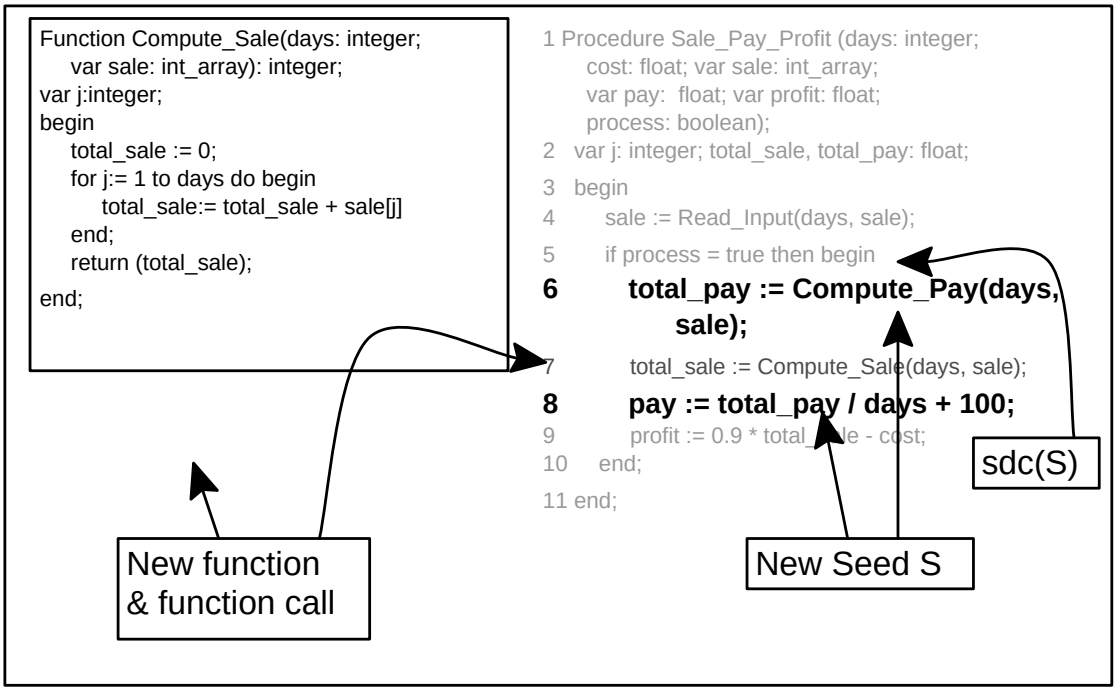


Figure 19: Total_Sale is extracted from Sale_Pay_Profit. Now, lines 6 and 8 become the seed for the next restructuring. The sdc under consideration is on Line 5 and the rc is defined by lines 6, 7, and 8.

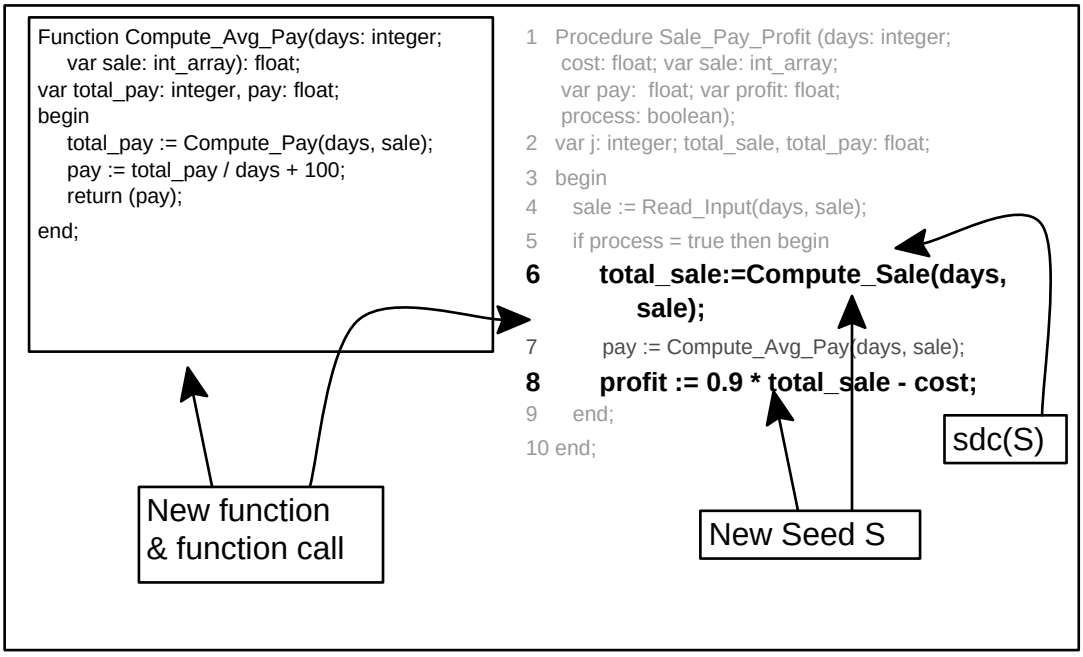


Figure 20: The average Pay is extracted from Sale_Pay_Profit. Now, lines 6 and 8 become the seed for the next restructuring. The sdc under consideration is on line 5 and the rc is defined by lines 6, 7, and 8.

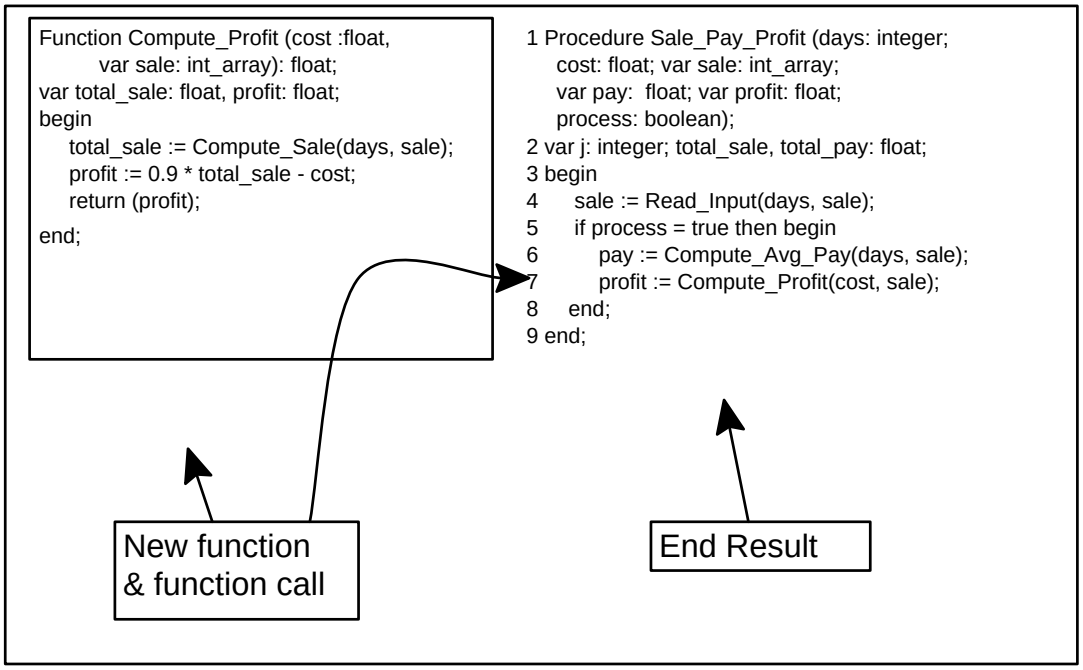


Figure 21: The end result is the function Sale_Pay_Profit semantically unchanged but now calling many smaller functions. Each of the function can become a method of an object in a object-oriented design.

Our illustrations show how our restructuring transformation can be used to accomplish two different re-engineering goals. The result in Figure 16 is identical to the one of Figure 2. The solution proposed in Figure 21 is semantically equivalent to the program of Figure 3. The differences are, in Figure 21, Compute_Pay and Compute_Sale are respectively called by Compute_Avg_Pay and Profit whereas in Figure 3, the main procedure Sale_Pay_Profit calls Compute_Pay and Compute_Sale. These differences appear in the last two application of our transformations. In Figure 19 and 20, Compute_Pay and Compute_Sale are parts of the seed. Therefore, they are extracted and put in the new functions. Results identical to Figure 3, will be achieved if line 8 is used as the only seed in the last two transformations. Since our transformation preserves the semantics, all the different results for the Sale_Pay_Profit restructuring are semantically equivalent, even though they differ in their internal structure.

6. Related works

In this chapter, we present a comparison of our research with other efforts in software restructuring. Most of the efforts, like ours, use slicing to decompose functions. However, they do not have a mechanism for bounding the slice to a region! Hence they have a limited application. Also related to our work is the identification of interleaved computation [Rugaber96, Rugaber95a, Rugaber95] and the inverse problem to ours, function composition or integration [Horwitz89]. We do not survey these research because they are complementary to our work.

Sneed and Jandrasics [Sneed87]

Sneed and Jandrasics have presented a technique that uses the control flow of a COBOL program to identify code segments that can be converted into modules. For instance, they create a module for a loop or a section containing more than 200 statements. In the absence of any cue from control statements, they propose that the continuous blocks of 800 statements be broken into separate modules. Since a statement is placed in at most one module, this approach does not reorder nor duplicate code. Therefore, the semantics of the original program is not changed. On the other hand, since the modules created contain the same sequence of statements as in the original program, this technique does not separate intertwined threads of logic. Thus it neither helps separating reusable code components nor in making the program easier to modify.

Kim et al. [Kim94] and Kang & Bieman [Kang96]

The restructuring techniques of Kim et al. and Kang and Bieman use the cohesion (though Kim et al. call it coupling) between output variables (i.e., reference parameters, global variables) of a function. Using cohesion, they create groups of variables around which a function can be restructured. They then use program slicing to extract the useful statements. Their techniques differ in (a) how cohesion is measured, (b) how the related variables are grouped, (c) the class of

program for which the technique is safe (i.e., does not produce incorrect result), and (d) the class for which it produces correct results. For further details, we refer the user to the original works. These research have focused on the techniques to identify and extract reusable code. They have not addressed the problem of replacing the extracted code in the original program by call to the newly created function. Their algorithm for identifying related computation too is limited because they do not have a mechanism to bound the slice. Once they have identified groups of related variables, they use a standard slicing algorithm to extract the computations, this implies that they collect the computations throughout the entire function.

Ward and Bull [Bull94, Ward93]

Ward's work involved semantics preserving transformation for functional languages and Bull extended that work to wide spectrum languages. They provided an environment with direct transformation on the AST of a program. Ward proved that the transformations preserved the semantics of the original AST for functional languages. These primitive transformations are usually too simple to be useful as such, nevertheless they can be composed to create more powerful transformations. However, using the transformation in their catalogue, one cannot create our TUCK and SPLIT transformations. Also, there transformation cannot bound the scope of a slice and are therefore incapable of creating restructuring alternatives similar to ours.

7. Conclusion

Programmers have been re-engineering code manually since the first program was written. However, as a research discipline software re-engineering is fairly new. Most of the research efforts on automated support for software re-engineering has focused on extracting information that can be used for re-engineering. For instance, there has been a significant amount of work on identifying reusable code. There has not been significant, if any, work in automatically, modifying the code itself.

To the best of our knowledge, this thesis presents the first formal transformation for restructuring amenable for use in an automated re-engineering environment. Previous transformations proposed for similar problems are either not proven to be correct, or do not correspond to steps intuitively used by programmers, or are not general enough to be applicable in all but a few situations.

In developing our transformation, we have extended work in program slicing by inventing a mechanism to limit a slice to a single-entry, single-exit region. This bounded slice, called TUCK, provides a way to identify a meaningful computational thread within a context. We have then provided a transformation, SPLIT, for extracting such a bounded slice into a separate function by splitting a single-entry, single-exit region defined by TUCK into two regions.

Our vision is to automate the software re-engineering decisions as much as possible and to provide tools that require a minimal interaction. The enterprise of such tools will be to automate most of the restructuring decisions. To automate our transformation, it would be necessary to automatically generate the input seed to TUCK. The PDG and the CFG do not provide the appropriate abstractions for identifying seeds. The smallest unit of computation in these abstractions is a statement. These representations are too fine-grained, and contain too much

detail. To help identify the initial seed, we need abstractions that summarize computation performed by several statements. Lakhotia and Nandigam's variable dependence graph (VDG) and pair-wise cohesion table are two such abstractions [Lakhotia92, Nandigam95]. Work is in progress to use these abstractions to generate seeds automatically.

Currently, TUCK proposes several options for restructuring a function. Not all options are equally good. The problem of choosing the best option is deferred to the user. For the transformation to be practical in an interactive tool, the number of solutions proposed by TUCK should be limited to a select few such that the user is not burdened with unnecessary choices. In a batch tool, the set of options must be reduced to one. This may be important when dealing with large programs for which time and space usage may grow rapidly with the number of options. Further research is needed to generate only the useful solutions, and to rank these solutions.

8. References

[Aho86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, techniques, and tools*. Addison-Wesley, Menlo Park, CA, 1986.

[Bull94] Tim Bull. *Software maintenance by program transformation in a wide spectrum language*. PhD Dissertation, Durham University, Durham, England, 1994.

[Cartwright89] R. Cartwright and M. Felleisen. *The semantics of program dependence.* SIGPLAN Notices, 24(7):13–27, 1989.

[Chikofsky90] E. Chikofsky and J. H. Cross. *Re-engineering existing systems*. IEEE Software 7(1):13–17, 1990.

[Dasgupta94] Subrata Dasgupta. *Creativity in invention and design.* Cambridge University Press, New York, 1994.

[Ferrante87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. *The program dependence graph and its use in optimization*. ACM Transactions on Programming Languages and Systems, 9:319–349, 1987.

[Horwitz89] Susan Horwitz, Jan Prins, and Thomas Reps. *Integrating non-interfering versions of programs*. ACM Transactions on Programming Languages and Systems, 11:345–387, 1989.

[Kang96] B-K Kang and James Bieman. *Using design cohesion to visualize, quantify, and restructure software*. In C. V. Ramammoorthy, ed., Proceedings of the Eighth International Conference on Software Engineering and Knowledge Engineering (SEKE'96), Lake Tahoe, NV, pp 222–229, 10-12 June 1996, IEEE Society Press, Los Alamitos, CA.

[Kim94] Hyeon Soo Kim, In Sang Chung, and Yong Rae Kwon. *Restructuring programs through program slicing*. International Journal of Software Engineering and Knowledge Engineering, 4:349–368, 1994.

[Lakhotia92] Arun Lakhotia. *Rule-based approach to computing module cohesion*. In Vic Basili, ed., Proceedings of the 15th International Conference on Software Engineering, Baltimore, MD, pp 35–44, 17-21May 1993, IEEE Society Press, Los Alamitos, CA.

[Lehman85] Lehman, M. M. and L. A. Belady. *Program evolution*. Academic Press, London, 1985.

[Muchnick97] Steven S. Muchnick. *Advanced compiler design & implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[Nandigam95] Jagadeesh Nandigam. *A measure for module cohesion*. PhD dissertation, University of Southwestern Louisiana, Lafayette, 1995.

[Ottenstein84] Karl J. Ottenstein and Linda M. Ottenstein. *The program dependence graph in a software development environment*. ACM SIGPLAN Notices, 19(5):177-184, 1984.

[Rugaber96] Spencer Rugaber, Kurt Stirewalt, and Linda Wills. *Understanding interleaved code*. Automated Software Engineering, 3(1-2):47–76, 1996.

[Rugaber95a] Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills. *Detecting interleaving*. In Mari Georges, ed., Proceedings of the International Conference on Software Maintenance, Nice, France, pp 265–274, 16-20 October 1995, IEEE Computer Society Press, Los Alamitos, CA.

[Rugaber95] Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills. *The interleaving problem in program understanding*. In Elliot Chikofsky, ed., Proceedings of 2nd Working Conference on Reverse Engineering, Toronto, Ontario, pp 166–175, 14-16 July 1995, IEEE Computer Society Press, Los Alamitos, CA.

[Sneed87] Harry M. Sneed and Gabor Jandrasics. *Software recycling*. In Wilma Osborne, ed., Proceedings of the Conference on Software Maintenance, Miami, FL, pp 82–90, 16-19 October 1987, IEEE Society Press, Los Alamitos, CA.

[Tip95] Frank Tip. *A survey of program slicing techniques*. Journal of Programming Languages, 3:121–181, 1995.

[Venkatesh91] G. A. Venkatesh. *The semantics approach to program slicing*. In Brent Hailpern, ed., Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, Toronto, Ontario, pp 107–119, 26-28 June 1991, IEEE Society Press, Los Alamitos, CA.

[Ward93] Martin Ward. *Abstracting a specification from code*. Journal of Software Maintenance: Research and Practice, 5:101–122, 1993.

[Weiser84] M. Weiser. *Program slicing*. IEEE Transactions on Software Engineering, 10:352–357, 1984.

[Yourdon92] Edward Yourdon. *Decline and fall of the American programmer*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

ERROR: undefined
OFFENDING COMMAND:

STACK: