Compensations in an Imperative Programming Language

Adrian Francalanza
CS Dept, ICT
University of Malta
adrian.francalanza@um.edu.mt

Gordon Pace
CS Dept, ICT
University of Malta
gordon.pace@um.edu.mt

Lydia Vella
CS Dept, ICT
University of Malta
lvel0019@um.edu.mt

ABSTRACT

Numerous mechanisms are used to deal with failure of systems or processes, one of which is that of compensating actions. A compensation can be seen as a program which somehow cancels out the effects of another — and by organising code in such a way so as to associate each program with its compensation, enables implicit recovery to a sane state if part of a computation somehow fails. In this paper we present an imperative programming language natively supporting the notion of compensations, thus enabling the programmer to program using a notion of compensations at the source level of the system under development.

1. INTRODUCTION

The abnormal termination of a process may compromise the data integrity as well as the overall consistency of the system. For this reason, failure handling is deemed to be an essential task in software solutions. While several different methods may be employed for handling failure, perhaps the two most predominant are reparation and compensation. Using a reparation mechanism, one would attempt a block of code, whilst having a second block of code that one can execute to repair the state of the system should anything go wrong while the first block is executing. Such a mechanism is already present in numerous programming languages through the use of exception handling.

However, this approach cannot be used for all types of failure. In some cases, attempting to correct the state of the system through the use of exceptions may not be sufficient, as in the case of hardware failure, where the execution of the exception handler alone cannot repair the state of the system [6]. A possible option for resuming execution in such cases would be to have a backup system available that one can resort to. For this to work effectively, checkpoints need to be recorded, such that the backup system may resume its execution from a recorded checkpoint.

A further enhancement to this approach is presented through the notion of transactions. A transaction is essentially a number of actions, grouped together in an single atomic block, such that either all of the actions within the transaction succeed, or if they fail, no trace of their execution remains. This allows for the interleaving of parallel processes, as well as for failures to be hidden from the programmer [8]. This is similar to the use of checkpoints, where a checkpoint would be created at the very beginning of each transaction, the only difference being that all actions following the checkpoint are undone upon returning to that particular checkpoint, using what is referred to as a rollback mechanism [6]. This approach has yielded quite decent results especially when dealing with database operations.

However, there remain a number of situations where atomic transactions cannot be applied. The main problem with transactions is that these usually place locks on the shared resources used within the transactions e.g., database tables . Modern enterprise systems often involve numerous long-running transactions, where the duration of the transaction itself, together with the fact that long-running transactions usually access a greater number of shared resources, would undoubtedly give rise to lengthy locking delays, reducing the throughput of the system [8]. Furthermore, transactions may interact with other systems. Here, once again, the use of locking is inadequate since a system often is prohibited from creating a lock on another system's resources. Interaction might also take place with other agents, such as humans in the real world, where checkpoints cannot be applied, and should there be failures, rollback would not be viable.

A solution to these problems may be found through the use of compensations. A compensation is an activity defined by the programmer, associated to some other activity or process. When executed a compensation cancels out the effects of the activity to which it is associated. Unlike the notion of rollback, a compensation does not simply remove all evidence of a transaction from the database. A trace of the failed activity remains, however, this is followed by a trace of its compensating activity [6].

In this paper we develop programming constructs to deal with compensations in the context of an imperative programming language. Since this involves the design of a language, it is important to keep in mind the simplicity and reliability of the language whilst keeping syntax additions to a minimum. The designed language should allow the programmer to represent compensations in a natural and comprehen-

sible manner. The language is evaluated by means of a case study, through which the effectiveness of the language may be assessed. As a case study, a simple File Transfer Protocol (FTP) client command line interface was implemented, having compensating activities associated to each of the FTP commands, and developed in both the designed language as well as an imperative language, to enable comparison of the two. The case study demonstrates the improved readability and maintainability of programs employing compensations, developed using the designed language, over those developed using a conventional imperative programming language.

2. COMPENSATIONS DESIGN OPTIONS

A compensation can be described as a *history-dependent reparation* whereby the recovery required to be computed by the reparation is determined at runtime, based on previous execution paths taken i.e., the history of the program execution so far. The differences between compensations and reparations are best understood w.r.t., the following example using a commonly used reparation mechanism, the try-catch command.

$$\mathsf{try}\ \{\ \mathsf{while}\ e\ \mathsf{do}\ c_1\}\ \mathsf{catch}\ \{c_2\}$$

The reparation c_2 in the code above is usually *static* in nature, in the sense that its execution is the same for every instance where $\mathsf{while}\ e\ \mathsf{do}\ c_1$ throws an exception. Situations however arise whereby it would be advantageous to make the reparation c_2 *dependent* on the execution path of $\mathsf{while}\ e\ \mathsf{do}\ c_1$ before it throws the exception. For instance, it would be natural for the program to want to execute c_2 for every iteration of c_1 completed. Although this functionality can, of course, be coded within a Turing complete language, it introduced the possibility for coding errors and affects the intelligibility of the code. This abstraction can however be neatly expressed through compensations, and in this study we seek to explore the integration of such a mechanism within a standard algol-like language.

There are various design decisions that need to be analysed before such a mechanism is incorporated. While doing so, it is important to keep in mind the simplicity and reliability of the language, whilst keeping syntax additions to a minimal. The design choices explored include:

Installation of Compensations: At which instant should compensating actions be installed and to which activity or aspect should compensation installation be associated?

Scoping of compensations: A compensation should have a scope, identifying the boundaries within which the program may triggered it. Furthermore, one has a choice as to what action to take once the end of a scope is reached — should compensations pertaining to that particular scope be kept, or should they be discarded?

Activation of compensations: The whole point behind the specification of compensations is so that they can be triggered later on, so as to undo some forward action. This raises the question of whether compensations should be activated implicitly or explicitly. Implicit activation of compensations implies that compensations are activated through the occurrence of an exception in the system or the failure of an action. This means that the application programmer would have no means by which to force the activation of the currently installed compensation handlers. Explicit activation, on the other hand, entails that the programming language contains a special construct, through which all the installed compensations since the beginning of a scope may be triggered.

Recovery checkpoints: Once compensations are triggered, the program can be seen as though it is running 'backwards', undoing the previous actions. However, in most cases, it is desirable to allow the program to recover and continue to progress forwards in its execution. Through such mechanisms upon encountering failure, the program may set off into backward recovery mode, compensating its executed activities up to the point where an alternative route of execution is encountered.

Variable handling: The use of variables in compensating activities raises an important implementation issue. Since the installation and the activation of compensations occur at different stages within a program, the use of global variables within compensating activities may have unforeseen consequences. For example, consider a variable x, being used to hold the cost of an item which is used to debt a client's account. The compensation of the debt action would most naturally be that of crediting the account by x (possible subtracting some processing fee). However, the value of x might change by the time the failure occurs — clearly the compensation should be triggered with the old value of x (at the time of installation of the compensation), and not the value of x at the time of recovery.

3. A COMPENSATION LANGUAGE

Our compensation-aware language assumes the normal constructs found in a standard imperative language with side-effect free expressions e and commands c that include variable assignment, sequential composition, branching, iteration and exception handling. In addition we add four new compensation-related constructs for compensation scoping, installation, activation and purging. The syntax of our language is summarised in Figure 1.

The runtime execution of our compensation-aware programs assumes that the underlying system offers functionality for registering compensating blocks of code in a LIFO structure; these compensations can also be grouped and identified through a scope name. Intuitively, the execution can be in two modes: in the *forward* mode the program executes the normal commands in order whereas in the *backward* mode the program executes the compensations installed. We next discuss each of the compensation constructs.

Compensation Installation This construct enables the programmer to associate a compensation block of code c_2 with another block of code c_1.

$$\mathsf{undo}\ c_1\ \mathsf{with}\ c_2$$

The block of code c_2 is installed upon the successful termination of sub-program c_1.

$$
\begin{array}{lcl}
n & \in & \textbf{ScopeName} \\
v & \in & \textbf{VarName} \\
e & \in & \textbf{Expression} \\
 & ::= & n \mid v \mid e \text{ and } e \mid \text{not } e \mid e + e \mid e \text{ - } e \mid \ldots \\
c & \in & \textbf{Command} \\
 & ::= & \text{skip} \mid c; c \mid v := e \mid \text{if } e \text{ then } c \text{ else } c \\
 & \mid & \text{while } e \text{ do } c \mid \text{try } c \text{ catch } c \mid \text{scope } n \ \ c \\
 & \mid & \text{undo } c \text{ with } c \mid \text{compensate } n \mid \text{purge } n
\end{array}
$$

Figure 1: The compensation language syntax

The Scoping Construct Using this construct, a programmer may create a new compensation scope with name n such that, all compensations installed during the execution of the sub-program c pertain to this scope.

$$\text{scope } n\, c$$

Scoping, or compensation grouping, acts both as an identifier for a group of compensations and well as a delimiter when compensations are activated - the backward execution mode. Indeed we require a mechanism to specify *up to which point* a program needs to compensate until it continues computing in forward mode again.

Compensation Activation Through this construct, the programmer may explicitly activate all the installed compensations, in reverse order of installation, until the beginning of the scope named n. Use of the compensate n construct outside the scope of n should not be permitted and should generate a compile time error.

$$\text{compensate } n$$

Compensation Purging This constructs serves the purpose of discarding currently installed compensations. Calling this construct from within a scope named n discards all the installed compensations since the beginning of that scope. Again, use of this construct outside a scope with matching name should not be permitted.

$$\text{purge } n$$

Example 1. Consider the following sub-program:

scope *one*{
  undo c_1 with c_1';
  scope *two*{
    try {
      try {
        undo c_2 with c_2'
        } catch { compensate *one*};
        undo c_3 with c_3';
        c_4
    } catch { compensate *two*; purge *one*}
  }
}

The four compensation constructs give us a finer-grained control to be able to dynamically program compensations as part of the language. For instance, if the execution of c_2 fails and throws an exception, then we compensate up to scope *one*. This means that c_1' is compensated before we continue executing from c_3. Interestingly however, the compensations executed by compensate *two* are determined *dynamically* by the execution. Assuming that c_2 executes successfully, then if c_3 fails, only compensation c_2' is executed. However if c_4 fails, then compesations c_3' and c_2' are executed in that order. Purging then allows us to dispose of the remaining compensations up to scope *one*, i.e., c_1'.

The four constructs seem to satisfy the minimality constraints. The expressivity of our compensation constructs are asses in two ways. First, we show how higher level constructs usually associated with compensations can be expressed as syntactic sugaring using our four constructs. Second, in Section 4 we consider the utility of these construct in constructing a medium sized application where compensations are a useful abstraction.

Attempt Otherwise This construct is used to give the program an alternative. That is, should program `P` fail, all the completed actions within `P` are compensated for, and the program resumes by executing program `Q`.

```
attempt scname { P } otherwise { Q }
```

This construct may be defined in terms of the basic language constructs as follows:

```
scope scname {
  undo {
    try {
      P
    } catch {
      Q
    }
  }
}
```

Local Scope The purpose of this construct is to discard all accumulated compensations upon the successful termination program `P`.

```
local_scope scname { P }
```

This may be defined in terms of the basic language constructs as follows:

```
scope scname {
  ...
  purge scname;
}
```

Local Attempt Otherwise Similar to the previously discussed construct, this command invokes the process `Q` should `P` fail, that is, after compensating for all the previously completed actions within `P`. However, an additional feature of this construct is that all accumulated compensations are discarded upon the successful termination of either of `P` or `Q`.

```
local_attempt scname { P } otherwise { Q }
```

This may be defined in terms of the basic language constructs in the following manner:

```
scope scname {
   try {
     P;
     purge scname;
   } catch {
     Q;
     purge scname;
   }
}
```

Override Compensation The principal function behind this construct is to replace the number of compensations accumulated within program `P` with a single global compensation `Q`.

```
override_comp  scname { P }  with { Q }|
```

This may be defined in terms of the basic language constructs as follows:

```
undo {
  scope scname {
    P;
    purge scname;
  }
} with {
  Q;
}
```

4. CASE STUDY

To evaluate the designed language, a simple File Transfer Protocol (FTP) client command line interface was implemented using both the designed compensation-enhanced language as well as the C programming language. The command line interface accepts from the user a selected subset of the basic FTP commands and defines compensating actions for each of the supported commands, enabling the end user to undo any previously executed command.

However, the availability of compensations as a built-in construct in the language, allows for new FTP commands to be defined, which broaden the range of functions available to the user from those offered in a standard FTP command line client and which may be easily programmed using the available compensation constructs, the most notable of which probably being checkpoints. The checkpoint command enables the user to establish a point in the program's execution, defined by a name given by the end-user himself, such that, at any further stage in the program, the user may decide to return to a particular checkpoint, identified by its name, undoing all the executed commands subsequent to that checkpoint.

Using the designed language, checkpoints may be easily programmed as follows:

```
int createCheckpoint(string name) {
  scope name {
    string input {
    while (input != close) {
      gets(input);
      if (input == get)
      then {
        undo { getFile{}; }
        with { printf("Deleting local file"); }
      }
      ...
      ...
      ...
      if(input == checkpoint)
      then {
        printf("Enter name for checkpoint");
        string input2 {
            gets(input2);
            createCheckpoint(input2);
        }
      }
      if (input == return)
      then {
        string toReturn {
          printf("Which checkpoint to return to?");
          gets(toReturnTo);
          compensate toReturnTo;
        }
      }
    }
  };
}
```

Similarly, a command that may be easily programmed using the compensation oriented language is the undo last command. As its name states, this command enables the user to undo the last executed command.

5. COMPILING COMPENSATIONS

To encode compensations, a naive approach would be to implement the checkpoint mechanism using lists. Considering the following sequence of events within a program:

```
Checkpoint A ; a ; b ; Checkpoint B ; c ; d
```

One is faced with two options. The first would be to keep a list of all compensations of actions performed after the last declared checkpoint in the program. In such a case, the program would offer no means by which to return to checkpoint `A` once checkpoint `B` has been declared. Alternatively, one can also keep a separate list of committed actions for each of the checkpoints declared. However, in this case, and without additional information, if the user decides to return to checkpoint `A` after returning to checkpoint `B`, the compensations for actions `c` and `d` would be executed twice, since they would appear on both the list of checkpoint `A` and that of checkpoint `B`.

To effectively store installed compensations, and allow the programmer to explicitly activate them in reverse of the

chronological order in which their forward transactions were completed, the appropriate structures need to be employed. The proposed solution uses a global stack of instructions which may be invoked by the program. Each stack location may contain either a compensating action, that is, a block of code representing a compensation for a previously executed action in the program, or the name of a scope, a string representing the name of a currently open scope within the program.

Different instructions supported by the language affect the global compensation stack in different ways. The execution of imperative language instructions leaves the stack unaffected. Upon encountering an `undo...with` block, the `undo` block is executed and the action within the `with` block is pushed onto the stack. Similarly, whenever a scope is encountered its name is pushed onto the stack. At the end of the scope the name is removed, however all compensations installed during the scope are left on the stack. Then, to compensate a particular scope `n`, compensations are popped off the stack and executed until the name `n` is found. However, any scope names found on the stack, including `n` itself are not removed from the stack.

Two alternative approaches were recognised for handling variables within compensations. The first solution is to create local copies of the global variables within the compensation functions themselves. Nonetheless, this approach still does not eliminate the issue completely, as portrayed by the following scenario:

```
...
scope T {
  ...
  while(i<10){
    x = 100;
    undo {
      Debit Client Account(x);
    }
    with {
      Credit Client Account(x);
    }
    x = x + 10;
    i--;
  }
  ...
  ...
  ...
  compensate T;
  ...
```

In this example, the local value of x changes with every iteration of the while loop. Since the value of x is being stored as a local variable in the credit account function, it is being modified with every iteration. Therefore, then upon the completion of the tenth iteration, the local value of x would be equivalent to the value of x during the tenth iteration of the while loop. Consequently, while running the sequence of credit account compensations, the value of x would not reflect the actual value during each of their respective forward actions, but merely reflect the value of x during the execution of the last action completed in the loop, that is, the tenth action.

To solve the problem of handling compensations within loops, a slight modification over the latter approach is proposed. Associated to each compensation, a structure containing all the variables being used within that compensation should be created. Then, with every compensation that is installed on the global compensation stack, its respective variable-structure should be instantiated and pushed onto a global variable stack, for use with the installed compensation. Similarly, whenever a compensation is popped off the global compensation stack, its respective variable-structure is also popped off the global variable stack. Using this approach, the values of variables used within compensations when these are executed is the same as the value at the instant when the compensation was installed.

6. EVALUATION

In comparing the two implementations, it is evident that the designed language offers a more structured and clear approach for defining compensating activities in a program, than its equivalent C implementation. The declaration of compensation pairs is concise and rather comprehensible, concealing from the programmer the underlying mechanism used for creating and installing compensations. Similarly, the scopes pertaining to each compensating activity are clearly defined, and the activation of all installed compensations lying within a particular scope may be achieved using one simple construct. The use of meaningful constructs for handling compensations ensures that programs developed using the designed language may be easily read by others, thus increasing their maintainability.

Although the same functionalities may be achieved using the imperative C programming language, the code produced to install and activate compensations exposes all the underlying mechanisms used for compensation handling, making the code much more complex to read and understand. The process of declaring and installing compensating activities is a rather complex one and the code produced is somewhat cluttered when compared to the implementation using the designed language. The language's compensation mechanism does however impose a cost, that is incurred by every program developed in the designed language, regardless to whether it uses compensations or not. This overhead is the creation of the global compensation stack, and the global variable stack, together with their respective push and pop methods, as well as other global variables used by the stack. These are created every time a program is compiled, however, they are never used in programs which do not make use of compensating activities.

The strength of the designed language lies purely in the way it handles compensating activities, providing the appropriate constructs and syntax to program compensations in a natural, effortless manner. Furthermore the availability of syntactic sugar constructs further enhances the simplicity, readability and maintainability offered by the language.

7. RELATED WORK

A compensable transaction is composed of a pair of programs: a forward action followed by its associated compensating action. The forward action is reminiscent of conven-

tional transactions, that is, it either completes successfully or causes the system to be rolled-back to the state it was in prior to the execution of the transaction. As the compiler starts executing a transaction, the compensation for each successfully executed instruction is remembered. The stream of accumulated compensations continues to build up during execution, up to the point where the system encounters a failure, at which point all the accumulated compensations are executed.

The use of this mechanism for handling failure makes the idea of *atomic* long-running transactions feasible, since they may be based on a weaker notion of atomicity which relies on compensations [7]. This means that long-running transactions are split into a number of shorter, atomic ones. Yet a weaker notion of atomicity remains, because should one of these shorter-duration transactions fail, compensations for all the sub-transactions forming part of the long-running transaction, that were committed prior to the failure of the transaction are invoked, cancelling out the effect of these transactions. An additional advantage of using compensations is that interaction with other systems or external entities in the real world becomes possible, seeing as how it can easily be undone at a later stage if need for it arises [9].

Existing formalisms dealing with compensations include: Sagas Calculi [7] in which long-lived transactions are split up into a number of independent atomic activities, executed collectively as a non-atomic Saga, Compensating CSP [4] which extends CSP to deal with long-running transactions through the use of compensations and StAC [3], a business process modelling language supporting the use of the compensation construct.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have focused on the introduction of compensation-handling constructs in a sequential programming language, and the issues rising in such a setting. Typically, compensations are handled using libraries, rather than through specific constructs at the programming language level. In the case study, we have shown how compensations can also be used to program scenarios not strictly related to error-recovery.

Moving to a parallel programming language poses interesting additional design questions, most of which should, however, be orthogonal to the ones addressed in this paper. For instance, compensating parallel processes would introduce a choice regarding the triggering of compensations of concurrent processes — should their compensations be triggered concurrently? It is interesting to see how these issues can be addressed effectively in a programming language setting.

9. REFERENCES

[1] Roberto Bruni. Theoretical foundations for compensations in flow composition languages. In *In Principles of Programming Languages*, pages 209–220. ACM Press, 2005.

[2] Roberto Bruni, Michael Butler, Carla Ferreira, Tony Hoare, Hernán Melgratti, and Ugo Montanari. Comparing two approaches to compensable flow composition. *CONCUR 2005 - Concurrency Theory*, pages 383–397, 2005.

[3] Michael Butler and Carla Ferreira. An operational semantics for stac, a language for modelling long-running business transactions. In *In Coordination 2004, volume 2949 of LNCS*, pages 87–104. Springer-Verlag, 2004.

[4] Michael Butler, Carla Ferreira, Peter Henderson, Y Chessell, Catherine Griffin, David Vines, Y Chessell, Catherine Griffin, David Vines, Michael Butler, Carla Ferreira, and Peter Henderson. Extending the concept of transaction compensation. *IBM Systems Journal*, 41:743–758, 2002.

[5] Michael Butler and Shamim Ripon. Executable semantics for compensating csp. In *In: Proc. of the 2nd International Workshop on Web Services and Formal Methods. LNCS 3670*, pages 243–256. Springer, 2005.

[6] Christian Colombo and Gordon J. Pace. Compensations. *Technical Report 2010-01, Department of Computer Science, University of Malta*, 2010.

[7] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259, New York, NY, USA, 1987. ACM.

[8] Paul Greenfield, Alan Fekete, Julian Jang, and Dean Kuo. Compensation is not enough. In *Proceedings of the 7th International Conference on Enterprise Distributed Object Computing*, page 232, Washington, DC, USA, 2003. IEEE Computer Society.

[9] Tony Hoare. Compensable transactions. *Microsoft Research, Cambridge, England*, 2007.