Lazy UTP

Walter Guttmann

Institut für Programmiermethodik und Compilerbau
Universität Ulm, 89069 Ulm, Germany
walter.guttmann@uni-ulm.de

Abstract. We integrate non-strict computations into the Unifying Theories of Programming. After showing that this is not possible with designs, we develop a new relational model representing undefinedness independently of non-termination. The relations satisfy additional healthiness conditions that model dependence in computations in an elegant algebraic form using partial orders. Programs can be executed according to the principle of lazy evaluation, otherwise known from functional programming languages. We extend the theory to support infinite data structures and give examples to show their use in programs.

1 Introduction

Our goal is to extend the Unifying Theories of Programming (UTP) by non-strict computations. Consider the statement \( P = \text{def} (x_1, x_2 := \frac{1}{0}, 2) \) that simultaneously assigns an undefined value to \( x_1 \) and 2 to \( x_2 \). In UTP and most conventional languages its execution fails, but we want undefined expressions to remain harmless if their value is not needed. This is standard in functional programming languages with lazy evaluation like Haskell [25], Clean [26] and Miranda [37]. Yet also in an imperative language the equation \( P; (x_1 := x_2) = (x_1, x_2 := 2, 2) \) can be reasonable since the value of \( x_1 \) after the execution of \( P \) is never used. This is confirmed by the following Haskell program that implements \( P; (x_1 := x_2) \) in monadic style:

```haskell
import Data.IORef;
main = do r <- newIORef (div 1 0, 2)
          modifyIORef r (\(x1,x2) -> (x2,x2))
          x <- readIORef r
          print x
```

It prints \((2, 2)\) terminating successfully, but would abort if \((x2, x2)\) was changed to \((x1, x1)\). With non-strict computations available, programs can be expressed more freely since less attention has to be paid to avoid non-termination. For example, in functional programming languages they enable the use of infinite data structures. They too are not supported by UTP so far.

Regarding the statement \( P \) again, we have to address that UTP models undefinedness as non-termination [15, page 78]. In particular, \( P = (\text{false } \implies \text{true}) \) holds, hence \( P \) is the never terminating program (the solution of the recursive
In consequence there is no distinction between undefinedness of individual variables; actually $P = (x_1, x_2 := 2, \frac{1}{0})$ holds. Moreover, computations are strict in the sense that $P; (x_1 := x_2)$ is again the endless loop.

In some contexts such a uniform treatment of non-termination and undefinedness is not appropriate. UTP’s point of view is that of the specifier who does not care whether a program loops indefinitely or aborts due to an error, since in both cases it does not fulfil its objective. We can, however, argue for a differentiation between finite and infinite failure. From the users’ point of view, errors can actually be observed about executions of programs whereas non-termination cannot. From the programmers’ and language designers’ point of view, errors might be recovered from, for example, by exception handling. From the theorists’ point of view, error detection is semidecidable in contrast to non-termination which is not semidecidable. We therefore strive for a theory that separates undefinedness and non-termination. It is then manifest to regard variables individually to obtain an even finer distinction.

As explained in Section 2, UTP’s designs are not adequate to support non-strict computations. Let us therefore describe our new approach. As usual, we represent undefinedness of individual variables by adding a special value $\perp$ to their ranges. We add another special element $\infty$ to distinguish non-termination from undefinedness. The difficulty is to choose the relations and operations (that model computations) such that, on the one hand, they handle these special values correctly and, on the other hand, they are continuous. The latter is required to iteratively approximate the solutions to recursive equations, which corresponds to the evaluation of recursion in practice. Furthermore, key constructs such as composition and choice should retain their familiar relational meaning to obtain nice algebraic properties. We solve this problem by introducing a partial order on the ranges of variables and states, and forming the closure of relations with respect to this order.

Section 3 gives the relational basics. A compendium of relations modelling the programming constructs known from UTP is presented in Section 4. We identify several healthiness conditions they satisfy, starting with isotony and the left and right unit laws. In Section 5, we derive further properties, namely finite branching, continuity and totality. We thus obtain a theory similar to that of designs, but describing non-strict computations, able to yield defined results in spite of undefined inputs. Moreover, it is sufficient to execute only those parts of a program necessary to calculate the final results, which can improve efficiency.

Our framework can also be applied to programs with infinite data structures. Several examples constructing and modifying infinite lists are discussed in Section 6. We also show how to express in our framework the class of fold- and unfold-computations on (finite and infinite) lists. They are well-known in functional programming languages and include such operations as map and filter, the building blocks of list comprehensions.

With lazy execution comes the need to consider dependences between individual computations. Such dependences also play a role in optimising program transformations like those performed in compilers. Their structure is investigated in Section 7. Starting from the observation that non-strict computations with