Minimum Information Code in a Pure Functional Language with Data Types

S. Berardi\textsuperscript{1} and L. Boerio\textsuperscript{2}

\textsuperscript{1} Dipartimento di Informatica, Universita' di Torino
Corso Svizzera 185, 10149 Torino, Italy
stefano@di.unito.it

\textsuperscript{2} Dipartimento di Informatica, Universita' di Torino
Corso Svizzera 185, 10149 Torino, Italy
lucab@di.unito.it

Abstract. In this paper we study programs written in a purely functional language with Data Types. We introduce a class of redundant code, the minimum information code, consisting either of "dead" code (code we may avoid evaluating), or of code whose "useful part" is constant. Both kinds of code may be removed, speeding-up the evaluation in this way: we introduce an algorithm for doing it. We prove the correctness of the method and we characterize the code we remove.

1 Introduction

We often write programs by combining and instantiating pieces of code we already have, that is, by a massive re-use of older code. In this way, it may happen that some pieces of code we use compute information which is now useless, while some others may be turned into constant functions. The first kind of code is called dead code, the second kind of code, constant code. The problem of discovering which pieces of code are constant, and thus may be evaluated at compile time, is called Binding Time Analysis (see for example [11]). Both the removal of dead code and constant code are essential to achieve efficiency. None of the two problems may be solved completely, because there is no algorithm able to decide in general if a piece of code returns a useless or constant answer under all possible inputs. Yet interesting subproblems and related problems of both issues have been solved (e.g. in [11] and in [2]).

The topic of this paper will be a curious problem, related to, but not perfectly coinciding with Binding Time Analysis for typed $\lambda$-calculus with data types (an ML-like language). Rather than trying to detect, say, the pieces of code returning a constant integer value 0, 1, 2, 3 ..., we will try to detect the pieces of codes returning (in symbol form) either no useful information, or a constant and a minimum amount of useful information. We call such code minimum information code for short. Since integers in data types have the notation 0 or $S(t)$, a piece of code returns a constant and minimum amount of useful symbols if it always returns 0 or $S(t)$, where 0, $S$ are useful in the rest of the program, while $t$ is useless (it is dead code). To put it in other words, a piece of code is minimum
information if we use only the first symbol of its output, and this first symbol is always the same.

A few remarks in order to make the concept precise. Pieces of codes returning a constant and useful output 1, 2, 3 ... are not minimum informative, since 1, 2, 3 ... have notation $S(0)$, $S(S(0))$, $S(S(S(0)))$, ... consisting of more than one symbol. Thus not all constant code is minimum information code. A piece of code always returning 0, true, false, nil, ... is both constant and minimum information code. A trivial example of minimum information code which is not constant is $S(x)$ in $t = it(S(x), a, \lambda_b)$. The term $t$ evaluates to $(\lambda_b)it(x, ...)$, hence to $b$; thus the value of $x$ is of no use ($x$ is dead code), and $S(x)$ is minimum informative. If we know it in advance, we may replace $t$ with $b$, without having to evaluate $x$ in its environment.

Through examples (see section 6), we may convince ourselves that minimum information code arises naturally if we program in a functional language using data types. In this paper we address the question of detecting and removing it. Again, the general problem is undecidable, but we have found an interesting decidable subproblem: to find the pieces of code which we may retype by a singleton type (even if the actual type is not a singleton). The singleton types we use are $\Omega$ (used to type useless code) and singleton data types. The only inhabitant of any singleton data type $D$ has a one-symbol notation, hence any piece of code $u$ we may retype with $D$ is a minimum information code. The method we have used in order to check whether we may assign the type $D$ to $u$ is built over a method introduced in [13], [4], [7] and [5] in order to discover dead code. We first remove subterms which are dead code (performing in this way a first optimization of the term), and all types corresponding to types assigned to removed subterms. In this way we may remove some subtypes of the original type $T$ of $u$. Suppose $T$ is turned into $T'$. The idea is now of checking whether $T'$ is a singleton data type: if it is, we may replace $u$ with the only inhabitant of $T'$. We are forced to delay an example of this technique until after we have introduced the syntax for data types.

The contributions of this paper will be

- a formal definition, through chains of simplifications, of a technique for removing dead and minimum informative code, and a proof that it is correct;
- a proof of the fact that all possible chains of the simplifications converge to the same term; this allow us to chose any strategy we like for simplifying a term, without affecting the final result.

By building over an algorithm for removing dead code we introduced in [8], it is easy to compute a simplification chain requiring at most quadratic time in the term we want to simplify. We conjecture the existence of an algorithm solving the same problem in linear time.

This is the plan of the paper. In section 2 we introduce the typed $\lambda$-calculus with data types we will deal with in this paper. In section 3 we informally describe the method for removing dead code introduced in [8], and a method for removing minimum information code built over it. In section 4 we give the