# A Dependently Typed Framework for Static Analysis of Program Execution Costs

Edwin Brady and Kevin Hammond

School of Computer Science,
University of St Andrews, St Andrews, Scotland
Tel: +44-1334-463253; Fax: +44-1334-463278
{eb,kh}@dcs.st-and.ac.uk

**Abstract.** This paper considers the use of dependent types to capture information about dynamic resource usage in a static type system. Dependent types allow us to give *(*explicit) proofs of properties with a program; we present a dependently typed core language TT, and define a framework within this language for representing size metrics and their properties. We give several examples of size bounded programs within this framework and show that we can construct proofs of their size bounds within TT. We further show how the framework handles recursive higher order functions and sum types, and contrast our system with previous work based on sized types.

## 1 Background and Motivation

Obtaining accurate information about the run-time time and space behaviour of computer software is important in a number of areas. One of the most significant of these is embedded systems. Embedded systems are becoming an increasingly important application area: today, more than 98% of *all* processors are used in embedded systems and the number of processors employed in such systems is increasing year on year. At the same time, the complexity of embedded software is growing apace. Assembly language, until recently the development language of choice, has consequently been supplanted by C/C++, and there is a growing trend towards the use of even higher-level languages. This trend towards increased expressivity is, however, in tension with the need to understand the dynamic run-time behaviour of embedded systems. Such understanding is critical for the construction of resource-bounded software.

Because there is a need for strong guarantees concerning the run-time behavior of embedded software to be available *at compile-time*, existing approaches have usually either focused on restricting the programming language so that only resource-bounded programs are expressible, or else relied on painstaking, and often manual and inaccurate, post-facto performance measurement and analysis. However, restricting the language deprives the programmer of many useful abstraction mechanisms (c.f. [22,27,28]). Conversely effective program analyses work at a low level of abstraction, and thus cannot deal effectively with high-level abstraction mechanisms, such as polymorphism, higher-order functions (e.g. `fold`), algebraic data types (e.g. `Either`), and general recursion.

A. Butterfield, C. Grelck, and F. Huch (Eds.): IFL 2005, LNCS 4015, pp. 74–90, 2006.

In this paper we develop a framework based on *dependent types* which is capable of expressing dynamic execution costs in the type system. We focus on a strict, purely functional expression language and exemplify our approach with reference to the size of a data structure. The approach is, however, general and should, in due course, be readily extensible to other metrics such as dynamic heap allocation, stack usage or time.

A key feature of a dependently typed setting is that it is possible to express more complex properties of programs than the usual simply typed frameworks in use in languages such as Standard ML or Haskell. In fact, computation is possible at the type level, and it is also possible to expose proof requirements that must be satisfied. These capabilities are exploited in the framework we present here to allow static calculation of cost bounds; we use type level computation to construct bounds on execution costs. In this way we can *statically* guarantee that costs lie within required limits.

### 1.1 Dependent Types

The characteristic feature of a dependent type system is that types may be predicated on values. Such systems have traditionally been applied to reasoning and program verification, as in the LEGO [17] and COQ [6] theorem provers. More recent research, however, has led to the use of dependent types in programming itself, for example Cayenne [2] and Epigram [20,19]. Our aim is to use dependent types to include explicit size information in programs, rather than as an external property. In this way, type checking subsumes checking of these properties.

### 1.2 Contributions

We have previously used sized type systems such as [15,24] to represent program execution cost; such systems seem attractive for this purpose because there is a clear link between, for example, data structure size and heap usage. However, there are limits to the expressivity of sized type systems. In particular, there is a limit to the form of expressions we can use to express size, leading to difficulty in giving accurate sizes to higher order functions. In this paper, we explore the benefits of using a dependently typed intermediate language to represent size constraints of a high level program:

- We can express more complex properties than those available in the sized type system; we are not restricted in the constraint language. Since we can write programs at the type level, we can extend the constraint language as we wish. In particular, this gives us more flexibility in expressing the cost of higher order functions. There need be no loss of size information — we can give each program as precise a size predicate as we need.
- With dependent types, we can verify the correctness of constraints given by an external inference system. A program with embedded size constraints is a complete, self-contained, checkable term; correctness of the constraints is verified by the typechecker, a small and well understood (and hence relatively straightforward to verify) program. We do not have to provide soundness or completeness proofs for our framework if we implement it entirely within a system already known to be sound and complete.