Compiling Exceptions Correctly

Graham Hutton and Joel Wright

Abstract. Exceptions are an important feature of modern programming languages, but their compilation has traditionally been viewed as an advanced topic. In this article we show that the basic method of compiling exceptions using stack unwinding can be explained and verified both simply and precisely, using elementary functional programming techniques. In particular, we develop a compiler for a small language with exceptions, together with a proof of its correctness.

1 Introduction

Most modern programming languages support some form of programming with *exceptions*, typically based upon a primitive that abandons the current computation and *throws* an exception, together with a primitive that *catches* an exception and continues with another computation. In this article we consider the problem of compiling such exception primitives.

Exceptions have traditionally been viewed as an advanced topic in compilation, usually being discussed only briefly in courses, textbooks, and research articles, and in many cases not at all. In this article, we show that the basic method of compiling exceptions using *stack unwinding* can in fact be explained and verified both simply and precisely, using elementary functional programming techniques. In particular, we develop a compiler for a small language with exceptions, together with a proof of its correctness with respect to a formal semantics for this language. Surprisingly, this appears to be the first time that a compiler for exceptions has been proved to be correct.

In order to focus on the essence of the problem and avoid getting bogged down in other details, we adopt a particularly simple language comprising just four components, namely integer values, an addition operator, a single exceptional value called throw, and a catch operator for this value. This language does not provide features that are necessary for actual programming, but it *does* provide just what we need for our expository purposes in this article. In particular, integers and addition constitute a minimal language in which to consider computation using a stack, and throw and catch constitute a minimal extension in which such computations can involve exceptions.

Our development proceeds in three steps, starting with the language of integer values and addition, then adding throw and catch to this language, and finally adding explicit jumps to the virtual machine. Starting with a simpler language allows us to introduce our approach to compilation and its correctness without the extra complexity of exceptions. In turn, deferring the introduction of jumps allows us to introduce our approach to the compilation of exceptions without the extra complexity of dealing with jump addresses.