Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 68 (2007) 79–94
Science of Computer Programming
www.elsevier.com/locate/scico

An action compiler targeting Standard ML

Jørgen Bøndergaard Iversen [1]

Systematic Software Engineering A/S, Søren Frichs Vej 39, DK-8000 Aarhus C, Denmark

Received 1 December 2004; received in revised form 20 April 2006; accepted 3 May 2006
Available online 27 April 2007

Abstract

We present an action compiler that can be used in connection with an action semantics based compiler generator. Our action compiler produces code with faster execution times than code produced by other action compilers, and for some nontrivial test examples it is only a factor of two slower than the code produced by the Gnu C Compiler. Targeting Standard ML makes the description of the code generation simple and easy to implement. The action compiler has been tested on a description of the Core of Standard ML and a subset of C.

Keywords: Compiler generation; Action semantics; Code generation; Standard ML

1. Introduction

Automatically generating a compiler from a formal description of a language does not always lead to efficient compilers. A formalism that supports easy construction of readable, complete, and reusable descriptions of most programming languages and at the same time has tool support for automatically generating efficient compilers seems to be nonexistent. One formalism that tries to satisfy these requirements on a language description formalism and allows automatic generation of efficient compilers is Action Semantics (AS) [1,2]. By efficient compilers we mean compilers that produce fast code, and not compilers that run fast or produce small code. An AS-based compiler generator produces a front end that maps each program in the described language to an action. The front end is then connected to an *action compiler*, and the result is a compiler for the described language. Previous results [3,4] have shown that it is possible to generate compilers that produce code that is less than ten-times slower than the code generated by handwritten compilers, and in some cases even as fast as only twice as slow. Some restrictions have been put on the actions handled by the compiler to achieve this result, and often the implementation of the code generator in the action compiler is very complicated.

We present an action compiler that produces more efficient code than previous action compilers, and on some examples only a factor two slower than the code produced by the Gnu C compiler. The code generator translates

E-mail address: jbiversen@gmail.com.

[1] The author was affiliated with BRICS (Basic Research in Computer Science, http://www.brics.dk, funded by the Danish National Research Foundation) at the University of Aarhus while the work reported in this paper was carried out.

0167-6423/$ - see front matter
doi:10.1016/j.scico.2006.05.013

actions to Standard ML (SML) [5] in a straightforward way. The SML code is then compiled to executable code using the MLton[2] compiler.

An action compiler annotates and transforms the action in several steps. Our action compiler performs type inference (Section 3) and code generation (Section 4), but no optimizations on the action as seen in previous work [6, 4,7–9]. Instead we generate code that can easily be optimized by MLton.

It is an advantage to be familiar with AS and SML, but not a prerequisite, when reading this paper. We will briefly introduce action semantics in the following section.

1.1. Action Semantics

Action Semantics (AS) is a hybrid of Denotational Semantics and Operational Semantics. As in a conventional denotational description, inductively defined semantic functions map programs (and declarations, expressions, statements, etc.) compositionally to their denotations, which model their behaviour. The difference is that here denotations are *actions* instead of higher-order functions.

An Action Semantic Description (ASD) of a programming language must describe the syntax of the language, semantic functions mapping the language constructs to actions, and semantic entities used in the semantic functions. ASDs of nontrivial languages, like Java [10] and SML [11], have already been constructed.

Actions are expressed in Action Notation (AN) [1,2], a notation resembling English but still strictly formal. AN consists of a *kernel* that is defined operationally; the rest of AN can be reduced to kernel notation. Actions are constructed from yielders, action constants, and action combinators, where yielders consist of data, data operations and predicates. Yielders are not part of the kernel.

The performance of an action might be seen as an evaluation of a function from data and bindings to data, with side effects like changing storage and sending messages. We shall often refer to the input data/bindings of an action as the *given data/bindings*. The action combinators correspond to different ways of composing functions to obtain different kinds of control and data flow in the evaluation. The evaluation can terminate in three different ways: *normally* (the performance of the enclosing action continues normally), *abruptly* (the enclosing action is skipped until an exception is handled), or *failing* (corresponding to abandoning the current alternative of a choice and trying alternative actions). AN has actions to represent evaluation of expressions, declarations, abstractions, manipulation of storage and communication between agents. The yielders can be used to inspect memory locations and compute data and bindings.

To limit this paper, we are not concerned with the actions used to represent communication between agents. Table 1 presents all kernel action combinators and constants, together with a short informal explanation. In Table 1, A ranges over actions.

Fig. 1 gives an example of an action. In line 1 the identifier "x" is provided. In line three a new memory location l_1, containing a random nonnegative integer, is allocated, and the action combinator in line two makes sure that line three is performed after line one and that the output from both evaluations is concatenated into the tuple (x, l_1). Line four passes the tuple to the action in line five which applies the data operator binding to it and produces the bindings map { x : l_1 }. The scope of these bindings is line seven where they are just returned as data.

```
(1)      (((result x)
(2)     and-then
(3)      (choose-nat then create))
(4)   then
(5)     (give binding))
(6) scope
(7)      copy-bindings
```

Fig. 1. Example of an action.

[2] http://www.mlton.org/.