Proposal for a Bachelor’s thesis

A compiler for lazy datatypes in Isabelle/HOL

Supervisors: Dr. Andreas Lochbihler
Professor: Prof. David Basin
Issue date:

Prerequisites

• Very good skills in functional programming (e.g., from the FMFP course)
• Knowledge about logic and the \(\lambda\)-calculus (e.g., from the FMFP course)
• Interest in getting to know state-of-the-art theorem prover technology

Introduction

Computing with infinite data structures requires lazy evaluation. For example, the infinite list \([0..]\) of all natural numbers can be used to pair all elements in a list \(xs\) with their index: \(\text{zip } [0..] \, xs\). In a call-by-value language without laziness, the program \(\text{zip } [0..] \, xs\) would first try to fully evaluate the infinite list \([0..]\) and therefore fail to terminate. Unlike Haskell, many functional programming languages (e.g., Standard ML, OCaml) are not lazy and therefore do not admit such data-driven programs concisely.

Wadler et al. [5] have described two techniques to implement lazy evaluation for datatypes in Standard ML. They rely on a primitive type constructor \(\text{Lazy } a\) with two operations:

1. \(\text{delay } :: (() \rightarrow a) \rightarrow \text{Lazy } a\) suspends an evaluation by wrapping the computation in the type \(\text{Lazy}\), and
2. \(\text{force } :: \text{Lazy } a \rightarrow a\) unwraps a \(\text{Lazy}\) value, runs the suspended computation and caches the result.

Thus, data-driven programs can be expressed in strict languages, too.

The proof assistant Isabelle/HOL can be used to write functional programs and prove properties of them. Its code generator compiles such programs to four target languages: Standard ML, Scala, OCaml, and Haskell. The code generator is used for interactively evaluating programs, generating verified implementations, and searching for counterexample to conjectured properties (similar to Haskell’s Quickcheck).
data Stream a = Cons a (Stream a)

up n = Cons n (up (n + 1))

skip (Cons x (Cons _ xs)) = Cons x (skip xs)

nth 0 (Cons x xs) = x
nth n (Cons x xs) = nth (n - 1) xs

Figure 1: Input program using Haskell syntax

newtype Stream a = Lazy_Stream (Lazy (Stream_ a))
data Stream_ a = Cons_ a (Stream a)

unstream (Lazy_Stream s) = s

up n = Lazy_Stream (delay (\_ -> Cons_ n (up (n + 1))))

skip s = case force (unstream s) of
    Cons_ x s' => case force (unstream s') of
        Cons_ _ xs => Lazy_Stream (delay (\_ -> Cons_ x (skip xs)))

nth 0 s = case force (unstream s) of Cons_ x xs => x
nth n s = case force (unstream s) of Cons_ x xs => nth (n - 1) xs

Figure 2: Transformed program of Fig. 1 with lazy streams

Unfortunately, only Haskell supports lazy evaluation natively. Therefore, infinite datatype values like \([0..]\) cannot be used in programs compiled to Standard ML, Scala, or OCaml.

Objectives

The goal of this thesis is to write a plug-in to Isabelle/HOL’s code generator such that it implements datatypes lazily in the strict target languages Standard ML, Scala, and OCaml. The plug-in should preprocess the Isabelle/HOL programs and encapsulate the laziness in a type Lazy similar to the approach by Wadler et al.\(^1\)\(^2\)

Figures 1 and 2 show an example of such a transformation (using Haskell syntax). The datatype Stream a of infinite lists gets replaced by two mutually recursive datatypes Stream a and Stream_ a, where the new Stream a merely suspends the computation of a stream. Accordingly, pattern-matching on Cons in the original program is transformed to forcing.

\(^1\)In Scala, lazy values can be marked with the keyword lazy val, but Isabelle/HOL’s code generator does not support this keyword.

\(^2\)Letouzey [2, §4.2.2] has implemented a similar encoding of laziness for the code generator of the proof assistant Coq, which compiles to OCaml code.
the lazy stream.
The transformation can be implemented in two parts.

1. An Isabelle/HOL command (issued by the user) to tell the system to implement a
datatype lazily. This command generates a the new datatype as a copy (Stream_ with
constructor Cons_) and functions between them (Lazy_Stream and unstream) and
proves relevant theorems about them. For example,

\[
\text{Cons} \ x \ \text{xs} = \text{Lazy_Stream} \ (\text{delay} \ (\_ \ \rightarrow \ \text{Cons}_\ (-) \ \text{xs})).
\]

2. A preprocessor for Isabelle/HOL programs as a plug-in to the code generator. The
plug-in eliminates the former constructors (Cons) and related operations (like case
expressions) from the Isabelle/HOL program and produces an Isabelle/HOL program
which uses the new constructors (Cons_). The transformation also proves the two
programs equivalent.

**Tasks**

This project can be subdivided into the following tasks:

1. Make yourself familiar with Isabelle/HOL and the code generator (e.g., working through
the relevant tutorials [3, 1]. Read the relevant literature on implementing lazy evalu-
ation in strict languages [5], [4, §5.12], [2, §4.2.2]

2. Define a type Lazy a in Isabelle/HOL and configure the code generator to compile it
to the laziness type in the target languages Standard ML.

3. Design an algorithm to transform Isabelle/HOL programs into lazy programs for a
user-specified set of datatypes. Implement the algorithm in Standard ML, which is
the implementation language of Isabelle/HOL.

4. Implement the command for the user to specify that a datatype be implemented lazily.

5. Test your algorithm and the command thoroughly.

6. (optional) Extend item 2 to the target languages Scala and OCaml.

7. (optional) Extend everything to support constructors which are lazy only in some
arguments as specified by the user.

8. Write the final report and prepare the presentation.

---

3Implementations of such a type exist in Standard ML (http://mlton.org/Lazy) and Isabelle/ML (http://isabelle.in.tum.de/repos/isabelle/file/d3996d5873dd/src/Pure/Concurrent/lazy.ML). Scala supports lazy values directly and OCaml’s library has a module Lazy.

4An example of a manual transformation for possibly infinite lists can be found at https://www.isa-afp.org/browser_info/current/ AFP/Coinductive/Lazy_LList.html
Deliverables

The following deliverables are due at the end of the project:

**Final report** The final report should consist of an introduction; a theoretical background section; one or more sections describing the challenges in Isabelle/HOL, the implementation, and the test cases; and a conclusion. The report may be written in English or German.

**Isabelle/HOL theories** Complete Isabelle/HOL development that runs with the latest release or a recent developer’s version.

**Presentation** At the end of the project, a presentation of 20 minutes must be given during an InfSec group seminar. It should give an overview and discuss the most important highlights of the work.

References


