Lazifying case constants

Lars Hupel

March 17, 2025

Abstract

Isabelle's code generator performs various adaptations for target languages. Among others, case statements are printed as match expressions. Internally, this is a sophisticated procedure, because in HOL, case statements are represented as nested calls to the case combinators as generated by the datatype package. Furthermore, the procedure relies on laziness of match expressions in the target language, i.e., that branches guarded by patterns that fail to match are not evaluated. Similarly, `if-then-else` is printed to the corresponding construct in the target language. This entry provides tooling to replace these special cases in the code generator by ignoring these target language features, instead printing case expressions and `if-then-else` as functions.

1 Introduction

theory *Lazy-Case*
  imports *Main*
  keywords *lazify* :: *thy-decl*
begin

Importing this theory adds a preprocessing step to the code generator: All case constants (and *If*) are replaced by "lazy" versions; i.e., new constants that evaluate the cases lazily. For example, the type of *case-list* is $'a \Rightarrow ('b \Rightarrow 'b\ list \Rightarrow 'a) \Rightarrow 'b\ list \Rightarrow 'a$. A new constant is created with the type $(unit \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'b\ list \Rightarrow 'a) \Rightarrow 'b\ list \Rightarrow 'a$. All fully-applied occurrences of the standard case constants are rewritten (using the [*code-unfold*] attribute).

The motivation for doing this is twofold:

1. Reconstructing match expressions is complicated. For existing target languages, this theory reduces the amount of code that has to be trusted in the code generator, because the transformation goes through the kernel.
2. It lays the groundwork to support targets that do not have syntactic constructs for case expressions or that cannot be used for some reason, or targets where lazy evaluation of branching constructs is not given.

The obvious downside is that this construction will usually degrade performance of generated code. To some extent, an optimising compiler that performs inlining can alleviate that.

2 Setup

If is just an alias for *case-bool*.

lemma [*code-unfold*]: *HOL.If P t f* = *case-bool t f P* ⟨*proof*⟩

⟨*ML*⟩

end

3 Usage

theory *Test-Lazy-Case*
imports *Lazy-Case*
begin

This entry provides a **datatype** plugin and a separate command. The plugin runs by default on all defined datatypes, but it can be disabled individually:

datatype (*plugins del*: *lazy-case*) *'a tree* = *Node* | *Fork 'a 'a tree list*

context begin

The **lazify** command can be used to add lazy constants if the plugin has been disabled during datatype definition.

lazify *tree*

end

Nested and mutual recursion are supported.

datatype
  'a mlist1 = *MNil1* | *MCons1 'a 'a mlist2* **and**
  'a mlist2 = *MNil2* | *MCons2 'a 'a mlist1*

Records are supported.

record *meep* =
  x1 :: *nat*
  x2 :: *int*

4 Examples

definition *test* **where**
test x ⟷ (*if x then True else False*)

definition *test'* **where**
$test' = case\text{-}bool\ True\ False$

definition *test''* **where**
$test''\ xs = (case\ xs\ of\ [] \Rightarrow False \mid \text{-} \Rightarrow True)$

fun *fac* :: *nat* ⇒ *nat* **where**
$fac\ n = (if\ n \leq 1\ then\ 1\ else\ n * fac\ (n - 1))$

lemma *map-tree*[*code*]:
  $map\text{-}tree\ f\ t = (case\ t\ of\ Node \Rightarrow Node \mid Fork\ x\ ts \Rightarrow Fork\ (f\ x)\ (map\ (map\text{-}tree\ f)\ ts))$
⟨*proof*⟩

The generated code uses neither target-language `if-then-else` nor match expressions.

export-code *test test' test'' fac map-tree* **in** *SML*
⟨*ML*⟩
end