Constructor Functions

Lars Hupel

March 17, 2025

Abstract

Isabelle's code generator performs various adaptations for target languages. Among others, constructor applications have to be fully saturated. That means that for constructor calls occuring as arguments to higher-order functions, synthetic lambdas have to be inserted. This entry provides tooling to avoid this construction altogether by introducing constructor functions.

1 Introduction

theory *Constructor-Funs*
  imports *Main*
  keywords *constructor-funs* :: *thy-decl*
begin

Importing this theory adds a preprocessing step to the code generator: All datatype constructors are replaced by functions, and all constructor calls are replaced by function calls. For example, for the *Suc* constructor of *nat*, a new constant with the same type and the definition $suc' \ n = Suc \ n$ is created. Then, all instances of *Suc* (except for in the constructor functions themselves) are replaced. Note that the constructor functions are defined in eta-long form.

Note that this does not affect constructors declared by **code-datatype**, only **datatype** (and **free-constructors**).

The motivation for doing this is to avoid target-specific lambda-insertion by the code generator. In some target languages, constructors cannot be used as functions. As a consequence, terms like *map Suc xs* have to be printed as *map* (*fn x => Suc x*) *xs*). This entails generation of fresh names outside of the proof kernel. The transformation provided by this theory ensures that all constructor calls are fully saturated. This makes supporting target languages that forbid partially-applied constructor calls much easier.

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

ML-file ‹*constructor-funs.ML*›
setup ‹*Constructor-Funs.setup*›

end

3 Usage

theory *Test-Constructor-Funs*
imports *Constructor-Funs*
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*: *constructor-funs*) *'a tree* = *Node* | *Fork 'a 'a tree list*

context begin

The **constructor-funs** command can be used to add constructor functions if the plugin has been disabled during datatype definition.

constructor-funs *tree*

end

Records are supported.

record *'a meep* =
  field1 :: *'a*
  field2 :: *nat*

Nested and mutual recursion are supported.

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

4 Examples

datatype *'a seq* = *Nil* | *Cons 'a 'a seq*

fun *app* :: *'a seq* ⇒ *'a seq* ⇒ *'a seq* **where**
app Nil ys = *ys* |
app (*Cons x xs*) *ys* = *Cons x* (*app xs ys*)

fun *map* **where**
map - Nil = *Nil* |
map f (*Cons x xs*) = *Cons* (*f x*) (*map f xs*)

definition *bla* = *map* (*Cons True*) *Nil*

The generated code never calls constructors directly, but only through regular functions. These functions are defined in eta-long form.

declare [[*constructor-funs*]]

export-code *app bla plus-nat-inst.plus-nat* **in** *SML*

export-code *app bla plus-nat-inst.plus-nat* **checking** *SML Scala*

end