Lazy Algebraic Types in Isabelle/HOL

Andreas Lochbihler[1,⋆] and Pascal Stoop[2]

[1] Digital Asset (Switzerland) GmbH, Zurich, Switzerland,
`mail@andreas-lochbihler.de`
[2] Department of Computer Science, ETH Zurich, Zurich, Switzerland
`pstoop@student.ethz.ch`

Abstract. This paper presents the tool CodeLazy for Isabelle/HOL. It hooks into Isabelle's code generator such that the generated code evaluates a user-specified set of type constructors lazily, even in target languages with eager evaluation. The lazy type must be algebraic, i.e., values must be built from constructors and a corresponding case operator decomposes them. Every datatype and codatatype is algebraic and thus eligible for lazification.
Lazification is transparent to the user: definitions, theorems, and the reasoning in HOL need not be changed. Instead, CodeLazy transforms the code equations for functions on lazy types when code is generated. It thus makes code-generation-based Isabelle tools like evaluation and quickcheck available for codatatypes, where eager evaluation frequently causes non-termination. The transformations preserve the code generator's correctness guarantees as they are checked by Isabelle's kernel.

1 Introduction

Over the last six years, Isabelle/HOL has become the HOL-based prover with the best support for codatatypes [2–5], which are frequently used for modelling infinite data. Alas, Isabelle features based on code generation have been lagging behind in this respect: Interactive evaluation with the **value** command does not terminate for infinite data as it compiles to the eager target language Standard ML[3] and so does quickcheck when the claim involves infinite data. This constitutes a serious obstacle as evaluation provides important early feedback on and intuition about the formalised model.

For example, lazy lists are modelled as the following codatatype with the constructors *LNil* for the empty list, written [], and *LCons* for consing, written infix as $\cdot$. The selectors *lhd* and *ltl* return the head and tail of a non-empty lazy list.

codatatype $'a\ llist = LNil \mid LCons\ (lhd\colon\ 'a)\ (ltl\colon\ \langle 'a\ llist\rangle)$

[⋆] Most of this work was done while the first author was at the Institute of Information Security at ETH Zurich.

[3] The first author recently extended the **value** command with compilation to the lazy language Haskell [9], but this does not help, as pretty printing of results remains eager and therefore does not terminate for infinite data structures.

Suppose that we have defined two functions on lazy lists: *up n* produces the list of ascending numbers starting with n and *xs* **!** n returns the n-th element of the list *xs*.

primcorec *up* :: *nat* ⇒ *nat llist* **where** ‹*up n* = *n* · *up* (*n* + *1*)›

fun *lnth* :: ‹*'a llist* ⇒ *nat* ⇒ *'a*› (**infixl !** *100*) **where**
  ‹(*x* · *xs*) **!** *0* = *x*›
| ‹(*x* · *xs*) **!** (*Suc n*) = *xs* **!** *n*›

We now want to prove the following lemma. However, a typo slipped in and we have actually written a wrong statement:

lemma *lnth-up*: ‹*up n* **!** *m* = *m* + *m*›

Quickcheck, which is run automatically on all lemmas, is unable to spot the error because it tries to construct the infinite list *up n* for some choice of n until it runs out of memory or hits the timeout. And if we try to manually evaluate the two sides, we are out of luck again. The **value** command does not terminate for the same reason.

value ‹*up 5* **!** *10*›

Our tool CodeLazy addresses these issues: Quickcheck finds a counterexample and **value** terminates. Moreover, laziness is not restricted to interactive development tools like Quickcheck and **value**. It is equally useful when proving theorems by evaluation and in generated application code. Not only codatatypes benefit; laziness also simplifies data-driven programming with finite data.

CodeLazy hooks into Isabelle's code generator such that the generated code *lazily* evaluates a user-specified set of type constructors such as *llist*, even in target languages with eager evaluation. It changes the representation of a lazy type similar to what Wadler et al. have called the "even with difficulty" style [19] and adapts the code equations of functions on lazy types accordingly. In particular, constructors are replaced by lazy constructors and pattern matches are replaced with case operators that explicitly force the evaluation. To that end, CodeLazy implements a new algorithm that eliminates pattern-matching, which can be used independently and is more powerful than existing implementations. Other Isabelle/HOL tools like **case-of-simps** could broaden their scope by switching to our algorithm.[4] CodeLazy runs with Isabelle2017 and is available at `http://www.andreas-lochbihler.de/pub/code_lazy.zip`. It is scheduled for inclusion in the next Isabelle release as part of `HOL-Library`.

There is only one requirement on the lazy type constructor: it must be algebraic, as captured by Isabelle's **free-constructors** abstraction. That is, values must be built from constructors and a corresponding case operator decomposes them. Every datatype and codatatype is algebraic and thus eligible for lazification. This includes in particular mutually recursive (co)datatypes and recursion

[4] In fact, we initially tried to use **case-of-simps** to eliminate the patterns, but quickly ran into it failing on overlapping and missing patterns. We therefore developed and implemented our own pattern-matching algorithm.

through arbitrary bounded natural functors. No induction or coinduction principle is required. Exotic types like streams with uncountably many elements are therefore algebraic, too.

Two design goals have guided our development: First, lazification should be transparent to the user's definitions, theorems, and proofs. So the reasoning is not affected by laziness. This is crucial for seamless integration into the Isabelle/HOL ecosystem as all existing packages and tools can be used without change. We achieve this goal by automatically deriving a lazy view and transforming the code equations right before code generation.

Second, the trusted computing base should be kept as small as possible. Lazification should not be able to introduce inconsistencies, even when theorems are proven by execution. To that end, CodeLazy has all transformations of the code equations checked by Isabelle's inference kernel. We nevertheless need a small extension that makes it possible to use the result of a single lazy evaluation several times in one execution.

The paper is structured as follows: We describe the lazification approach and its implementation in §2. In §3, we present the user interface, several examples and discuss the limitations. Related work is discussed in §4. We conclude in §5 with future uses cases for different parts of our implementation.

2 Lazy Algebraic Types in the Generated Code

In this section, we describe how we model laziness in HOL (§2.1, §2.2), how CodeLazy constructs the lazy view for an algebraic type (2.3), and how it hooks into the code generator (§2.4, §2.5). We assume that the type that shall become lazy must have been defined before in the logic.

Our approach works with arbitrary algebraic types. All datatypes and codatatypes definable with Isabelle's (co)datatype package [4] fall into this class. In particular, this includes mutually recursive (co)datatypes and recursion through arbitrary bounded natural functors. Other definition mechanisms like records are covered, too. Non-free types such as (multi)sets and unordered pairs are not algebraic. In principle, our approach can handle non-uniform (co)datatypes [5], but Isabelle's code generator cannot handle them. We therefore restrict the presentation to uniform algebraic types.

Formally, an *algebraic type* is a type constructor $\overline{\alpha}\ \kappa$ equipped with constructors $C_i :: \overline{\sigma_i} \Rightarrow \overline{\alpha}\ \kappa$ $(1 \le i \le n)$ and a case combinator *case-*κ:: $\overline{(\overline{\sigma_i} \Rightarrow \beta)} \Rightarrow \overline{\alpha}\ \kappa \Rightarrow \beta$ where all type variables in σ_i must occur in $\overline{\alpha}$ and every occurrence of κ in any $\overline{\sigma_i}$ takes the arguments $\overline{\alpha}$. The constructors must be free, i.e., injective, mutually disjoint, and exhaustive; and the case combinator must be an eliminator: *case-*$\kappa\ \overline{f}\ (C_i\ \overline{x}) = f_i\ \overline{x}$.

No induction or coinduction principle is needed for algebraic types. Exotic types like streams with uncountably many elements are therefore algebraic, too. Moreover, in mutually recursive (co)datatypes, we can choose to treat only some of them as lazy and others as eager.

2.1 Suspensions

As pointed out by Wadler et al. [19], laziness can be captured in an eager language by a suspension type *'a lazy* with two primitive operations:

- *delay* :: (*unit* ⇒ *'a*) ⇒ *'a lazy* wraps the computation *f* in a suspension, and
- *force* :: *'a lazy* ⇒ *'a* unwraps and runs the stored computation.

Since HOL has no notion of evaluation, we model the suspension type *'a lazy* simply as a copy of *'a*. So a suspension *'a lazy* is isomorphic to its result *'a*. The *delay* operation, not *force*, "runs" the computation in the logic by applying the unit closure to ().

typedef *'a lazy* = ‹*UNIV* :: *'a set*›

lift-definition *delay* :: ‹(*unit* ⇒ *'a*) ⇒ *'a lazy*› **is** ‹λ*f*. *f* ()›
lift-definition *force* :: ‹*'a lazy* ⇒ *'a*› **is** ‹λ*x*. *x*›

Only when generating code can we talk about evaluation order and this is when we specify *delay* and *force*'s evaluation behaviour: *delay* becomes *'a lazy*'s code constructor and it is *force*'s code equation that supplies () to run the computation.

code-datatype *delay*
lemma *force-code* [*code*]: ‹*force* (*delay f*) = *f* ()›

In a call-by-value language, *delay* now delays the evaluation and *force* forces it. For evaluation with Isabelle's simplifier (*code-simp*, **value** [*simp*]), we instruct it to not evaluate the suspended computation using a congruence rule:

lemma *delay-lazy-cong*: ‹*delay f* = *delay f*›
setup ‹*Code-Simp.map-ss* (*Simplifier.add-cong* @{*thm delay-lazy-cong*})›

These declarations yield a basic kind of lazy evaluation. For example, the following command evaluates to the string *''x''*. Without laziness, the evaluation would fail with a pattern-match error caused by taking the head of the empty list.

value [*code*] ‹*let x* = (*''x''*, *delay* (λ-. *hd* [])) *in fst x*›

2.2 Caching Results

The suspensions from the previous section suffice to implement lazy evaluation. Some computations, however, may be evaluated multiple times. For example,

value [*code*] ‹*let x* = *delay* (λ-. *length ''Isabelle''*) *in force x* + *force x*›

computes twice the length of the string *''Isabelle''*. The reason is that the suspension *x* does not cache the computation. Each *force x* therefore computes the length afresh. But we would prefer that only the first *force* on a suspension runs the computation and all subsequent *force*s return the result of the first *force*.

To reuse a result of a shared suspension, the suspension must cache the result. As HOL does not know about evaluation and sharing, caching cannot be expressed in the logic. Even worse, we cannot express it in the higher-order rewrite system model of the code generator [6]. Instead, we provide manual implementations for the suspension type *lazy* for all target languages using **code-printing** declarations. Users must trust that these implementations correctly implement the caching. We emphasize that this is the only extension to the trusted computing base in our implementation.

In Standard ML, the implementation uses a reference cell; *delay* initialises the cell with the unit closure and *force* updates it with the evaluation result, a value or an exception. Subsequent calls to *force* merely return the cached value or raise the exception again.

```
datatype 'a content = Delay of unit -> 'a | Value of 'a | Exn of exn;
datatype 'a lazy    = Lazy of 'a content ref;

fun delay f = Lazy (ref (Delay f));

fun force (Lazy x) = case !x of
   Delay f => (
     let val res = f (); val _ = x := Value res; in res end
     handle exn => (x := Exn exn; raise exn))
  | Value x => x
  | Exn exn => raise exn;
```

For Haskell, we use the definition in the logic where *delay* already supplies the argument. This works because Haskell itself is lazy.

```
newtype Lazy a = Lazy a;
delay f = Lazy (f ());
force (Lazy x) = x;
```

In OCaml, we map *lazy* to the built-in `Lazy.t` type from the standard library. In Scala, a `lazy val` takes care of the caching.

Caching also makes it possible to pretty-print evaluated suspensions to the user. Pretty printing must not evaluate a suspended computation because there might be exceptions or non-termination hiding in it. We therefore implemented a term reconstruction function for *'a lazy* values in Standard ML, OCaml, and Scala that checks whether a *lazy* value has been forced and, if so, pretty-prints the result. For example, the first value command below prints _ for the unevaluated suspension x. The second command prints *3 · 4 · 5 · 6 · 7 · 8 · -* because computing y has evaluated the lazy list x to depth 5. Without sharing, both value commands would print _ .

value [*code*] ‹*let* x = *up 3 in* x›
value [*code*] ‹*let* x = *up 3*; $y = x$ **!** *5 in* x›

For Haskell, no term reconstruction is available for suspensions because we did not find an implementation-independent and reliable way to figure out whether a `Lazy` value has been evaluated. Consequently, the following shows only an uninformative _ as the result.

value [*GHC*] ⟨*let x = up 3*; *y = x* **!** *5 in x*⟩

2.3 Bringing Suspensions into an Algebraic Type

To turn an algebraic type into a lazy algebraic type, we wrap every value in a suspension for the code generator. For example, we want to pretend to the code generator that the codatatype *llist* was defined as follows.

codatatype *'a llist = Lazy-llist* (*unlazy-llist*: ⟨*'a llist-lazy lazy*⟩)
  and *'a llist-lazy = LNil-Lazy* | *LCons-Lazy* ⟨*'a*⟩ ⟨*'a llist*⟩

So a lazy list *xs* is a suspension whose result is either the empty list *LNil-Lazy* or a *LCons-Lazy x xs'* whose tail *xs'* is again a suspension. Wadler et al. [19] call this the even style with difficulty.

Clearly, we do not want to actually define *llist* like this in the logic as this would severely complicate working with lazy types. Indeed, the algebraic type has already been constructed in the logic. So there is no point in redefining it in a complicated way. Instead, CodeLazy definitionally constructs the mutually recursive view from the existing algebraic type as follows.

The key observation is that *'a llist-lazy* is isomorphic to *'a llist* as *'a lazy* is isomorphic to *'a*. So, given an algebraic type $\overline{\alpha}\ \kappa$ with constructors C_i, we define

– $\overline{\alpha}\ \kappa$*-lazy* as a copy of $\overline{\alpha}\ \kappa$ with the bijection *Rep-κ-lazy* and *Abs-κ-lazy*,
– *Lazy-κ x = Rep-κ-lazy (force x)* and *unlazy-κ x = delay (λ-. Abs-κ-lazy x)*,
– C_i*-Lazy* $\overline{x}$ = *Abs-κ-lazy* (C_i $\overline{x}$), and
– *case-κ-lazy* $\overline{f}$ *x = case-κ* $\overline{f}$ *(Rep-κ-lazy x)*.

From these definitions and κ's algebraicness, it is easy to derive the following equations by term rewriting:

$$C_i\ \overline{x} = \mathit{Lazy}\text{-}\kappa\ (\mathit{delay}\ (\lambda\text{-}.\ C_i\text{-}\mathit{Lazy}\ \overline{x})) \qquad (1)$$

$$\mathit{case}\text{-}\kappa\ \overline{f}\ x = \mathit{case}\text{-}\kappa\text{-}\mathit{lazy}\ \overline{f}\ (\mathit{force}\ (\mathit{unlazy}\text{-}\kappa\ x)) \qquad (2)$$

$$\mathit{case}\text{-}\kappa\text{-}\mathit{lazy}\ \overline{f}\ (C_i\text{-}\mathit{lazy}\ \overline{x}) = f_i\ \overline{x} \qquad (3)$$

The first two (1) and (2) show how κ's constructors and case combinator can be implemented in terms of the new constructors and case combinator, respectively. As one would expect, the constructor *delay*s the evaluation and the case combinator *force*s it. The last equation (3) implements the new case operator by pattern matching.[5] We can now switch to the new constructors in the code generator.

code-datatype $\overline{\mathit{Lazy}\text{-}\kappa}$
code-datatype $\overline{C_i\text{-}\mathit{Lazy}}$

[5] From these equations, our implementation derives a case certificate such that the code generator uses the target language case syntax for *case-κ-lazy*.

All these definitions, derivations and declarations are automated by the command **code-lazy-type**. Thus, after the command

code-lazy-type *llist*

the code generator views *'a llist* as if it had been defined in the mutually recursive way shown at the beginning of this section.

2.4 Transforming the Code Equations

With the lazy view on an algebraic type in place, we must also replace the old constructors and case combinators with the lazy ones in all code equations. Otherwise, we might not get the benefits of laziness or code generation may fail altogether. For example, *up*'s original specification $up\ n = n \cdot up\ (n + 1)$ will cause non-termination in an eager language like Standard ML because the list constructor $\cdot$ in the logic is now an ordinary function in the generated code. Call-by-value thus evaluates the argument $up\ (n + 1)$ *before* applying $\cdot$. So $\cdot$ cannot delay the evaluation and *up* therefore never terminates. Instead, *up* should be implemented as follows:

$$up\ n = \mathit{Lazy\text{-}llist}\ (\mathit{delay}\ (\lambda\text{-}.\ \mathit{LCons\text{-}Lazy}\ n\ (up\ (n + 1)))).$$

Clearly, users do not want to manually state and prove such an equation for all their functions. Fortunately, this equation can be obtained by unfolding (1) in *up*'s specification. The command **code-lazy-type** therefore adds the equations (1) and (2) as unfold equations to the code preprocessor. This eliminates all occurrences of the logical constructors and case combinator on the right-hand sides of the code equations.

Yet, the old constructors may also appear on the left-hand sides in pattern matches. For example, **!**'s equations

$$(x \cdot xs)\ \mathbf{!}\ 0 = x \qquad\qquad (x \cdot xs)\ \mathbf{!}\ \mathit{Suc}\ n = xs\ \mathbf{!}\ n \tag{4}$$

pattern-match on $\cdot$. Here, we cannot simply unfold (2) on the left, because that would bring in a *delay* and a λ abstraction, neither of which can be pattern-matched on in the target languages. Rather, we must eliminate the pattern matching on the left. This is done in two steps. First, we replace all patterns with eager constructors of lazy types on the left-hand side with a fresh variable and perform a case analysis on this variable on the right-hand side:

$xs\ \mathbf{!}\ 0 = (\mathsf{case}\ xs\ \mathsf{of}\ [] \Rightarrow ?\ |\ x \cdot xs' \Rightarrow x)$
$xs\ \mathbf{!}\ \mathit{Suc}\ n = (\mathsf{case}\ xs\ \mathsf{of}\ [] \Rightarrow ?\ |\ x \cdot xs' \Rightarrow xs'\ \mathbf{!}\ n)$

Second, the case operators on the right-hand side are replaced with the lazy case combinator by rewriting with (2):

$xs\ \mathbf{!}\ 0 = (\mathsf{case}\ \mathit{force}\ (\mathit{unlazy\text{-}llist}\ xs)\ \mathsf{of}\ \mathit{LNil\text{-}lazy} \Rightarrow ?\ |\ \mathit{LCons\text{-}lazy}\ x\ xs' \Rightarrow x)$
$xs\ \mathbf{!}\ \mathit{Suc}\ n = (\mathsf{case}\ \mathit{force}\ (\mathit{unlazy\text{-}llist}\ xs)\ \mathsf{of}\ \mathit{LNil\text{-}lazy} \Rightarrow ?\ |\ \mathit{LCons\text{-}lazy}\ x\ xs' \Rightarrow xs'\ \mathbf{!}\ n)$

The command **code-lazy-type** installs a function transformer in the code preprocessor which eliminates the patterns from the code equations. The second step is already covered by the unfold equations.

The big question is what we should put for the question marks. Indeed, the pattern matches in (4) do not cover the case where the list is empty. At run time, these cases should fail with an exception. But in HOL, where every function is total, there is no notion of an exception. So we fill the question mark with something that makes the code equations provable in HOL, namely *missing-pattern-match (STR ''Missing pattern in lnth'') (λ-. lhs)* where *lhs* denotes the left-hand side of the equation. Here, the HOL function *missing-pattern-match msg f* is defined as *f* () in the logic while it raises an exception with the message *msg* in the generated code.

In this paper, we only describe the key features of the pattern matching elimination algorithm. The full details with a correctness proof can be found in the second author's thesis [16]. The algorithm takes a list of equational theorems with the same head function symbol and the same number of arguments and outputs a list of equational theorems such that

1. all pattern matches on constructors of lazy types have been transformed into case combinators on the right-hand side,
2. all pattern matches on non-lazy types remain unless they appear inside a constructor of a lazy type, and
3. the eager evaluation behaviour is semantically equivalent to the input; in particular, earlier equations take precedence over later ones when their patterns overlap and *missing-pattern-match* errors are added for missing pattern.

The first property ensures that the second step can replace the eager case combinator with the lazy one. The second property is crucial to support non-free code constructors in patterns. For example, sets are implemented by default using lists via the non-free code constructor *set* :: *'a list* ⇒ *'a set*. The code equations therefore pattern-match on *set*, but these matches cannot be converted into a case operator in HOL because the constructor is not free. By leaving such matches on the left-hand side, our algorithm does not run into the problem of the non-existing case combinator. For example, it transforms the two equations

$\textit{insert-all}\ A\ []=A$
$\textit{insert-all}\ (\textit{set}\ \textit{ys})\ (x \cdot \textit{xs}) = \textit{insert-all}\ (\textit{set}\ (x \cdot \textit{ys}))\ \textit{xs}$

into

$\textit{insert-all}\ (\textit{set}\ \textit{ys})\ \textit{xs} = (\mathsf{case}\ \textit{xs}\ \mathsf{of}\ [] \Rightarrow \textit{set}\ \textit{ys} \mid x \cdot \textit{xs}' \Rightarrow \textit{insert-all}\ (\textit{set}\ (x \cdot \textit{ys}))\ \textit{xs}')$
$\textit{insert-all}\ A\ \textit{xs} = (\mathsf{case}\ \textit{xs}\ \mathsf{of}\ [] \Rightarrow A \mid x \cdot \textit{xs}' \Rightarrow ?)$

where the question mark abbreviates again a pattern-match error.

Moreover, the second property helps to mitigate the exponential blow-up that a full conversion from patterns to decision trees can cause, namely if the blow-up is due to patterns that need not be eliminated completely. For example, consider an algebraic type with constructors $C_1,\dots,C_n$ $(n{>}1)$ and a list of $m{+}1$ equations for the $m{+}1$-ary function *f*, and another type with a single constructor D with one argument.

$$\begin{array}{llllllll}
f & C_1 & \_ & \_ & \dots & \_ & (\mathrm{D\ x}) & = \dots \\
f & \_ & C_1 & \_ & \dots & \_ & (\mathrm{D\ x}) & = \dots \\
\vdots & \vdots & & \ddots & & \vdots & \vdots & \vdots \\
f & \_ & \_ & \dots & \_ & C_1 & (\mathrm{D\ x}) & = \dots \\
f & \_ & \_ & \_ & \dots & \_ & (\mathrm{D\ x}) & = \dots
\end{array}$$

If only the matches on D must be eliminated, our algorithm just moves the matches to the right hand side, i.e., the transformation yields again $m{+}1$ equations. Eliminating all patterns would be much worse, though; there would be only one equation, but the decision tree of case operators on the right has a branching factor of n in the first m level and a branching factor of *1* on the last level. This makes n^m cases overall.

The blow-up cannot be avoided in general, though. For example, if the equations are as follows

$$\begin{array}{llllllll}
f & C_1 & \_ & \_ & \dots & \_ & 1 & = \dots \\
f & \_ & C_1 & \_ & \dots & \_ & 2 & = \dots \\
\vdots & \vdots & & \ddots & & \vdots & \vdots & \vdots \\
f & \_ & \_ & \dots & \_ & C_1 & \mathrm{m} & = \dots \\
f & \_ & \_ & \_ & \dots & \_ & \_ & = \dots
\end{array}$$

and we want to eliminate the pattern matching on the numbers in the last argument. Now, we cannot simply replace the patterns in the last argument by a variable, because this would change the pattern-matching behaviour for the other arguments. In this case, our algorithm eliminates all patterns and it is easy to see that there is no better solution. So we end up with a decision tree with $(m+1)\cdot 2^m$ cases.

Finally, in the third property, the restriction to *eager* evaluation is justified by the call-by-value target languages. In fact, the evaluation behaviour in a call-by-need setting may change. For example, the zip function on lazy lists given by the equations

$\mathit{lzip}\ \mathit{xs}\ [] = []$
$\mathit{lzip}\ []\ \mathit{ys} = []$
$\mathit{lzip}\ (x \cdot \mathit{xs})\ (y \cdot \mathit{ys}) = (x,\ y) \cdot \mathit{lzip}\ \mathit{xs}\ \mathit{ys}$

evaluates in a lazy programming language like Haskell the second argument first. The pattern-matching elimination algorithm, however, eliminates the patterns from left to right, so the generated case expression on the right-hand side forces the first argument first. We do not consider this a problem because there is little point in using our laziness converter for Haskell. Rather, we recommend to deactivate the conversion when generating Haskell code.

2.5 Pretty Evaluation Results

By automatically generating the lazy view and converting the code equations, we manage to hide the laziness complications from the user's definitions and proofs. We now describe how CodeLazy hides the laziness view from the user when pretty-printing evaluation results, e.g., from the **value** command. Without further measures, the command

value [*code*] ⟨*let x* = *up 3*; *y* = *x* **!** *5 in x*⟩

prints the following monstrosity:

Lazy-llist (*delay* (λ-. *LCons-Lazy 3* (*Lazy-llist* (*delay* (λ-. *LCons-Lazy 4* (*Lazy-llist* (*delay* (λ-. *LCons-Lazy 5* (*Lazy-llist* (*delay* (λ-. *LCons-Lazy 6* (*Lazy-llist* (*delay* (λ-. *LCons-Lazy 7* (*Lazy-llist* (*delay* (λ-. *LCons-Lazy 8* (*Lazy-llist* (*delay* -))))))))))))))))))))

Fortunately, it is easy to instrument the code generator's post processor to instead output *3* · *4* · *5* · *6* · *7* · *8* · -. Two steps are necessary:

- Print the term *Lazy-κ (delay Pure.dummy)* as -. Term reconstruction uses this term to represent unevaluated parts.
- Fold (1). This replaces the lazy constructors with the original ones, which are then pretty-printed as usual.

The command **code-lazy-type** configures the post processor accordingly.

3 User interface

This section describes our tool's user interface and how it integrates with other Isabelle features. This may serve as a reference for users. We also present several examples and discuss the practical limitations.

3.1 Documentation

The CodeLazy tool is implemented in the theory `Code_Lazy` and the corresponding ML files. The theory formalises the type *'a lazy* from §2.1 and provides the serialisation instructions for caching (§2.2). It also defines six user-level commands:

- **code-lazy-type** *tycon*
- **activate-lazy-type** *tycon*
- **deactivate-lazy-type** *tycon*
- **print-lazy-types**
- **activate-lazy-types**
- **deactivate-lazy-types**

The commands on the left all take one type constructor name as an argument; those on the right do not take any argument.

The command **code-lazy-type** is the main workhorse. It defines the lazy constructors for the given type constructor and derives the relevant theorems as described in §2.3, and activates the lazy view as described in §2.4 and §2.5. The argument *tycon* must satisfy the following conditions:

– *tycon* must be a type constructor in HOL, i.e., no compound type nor a type abbreviation,
– *tycon* must have been registered as a free constructor type with a case operator, e.g., via the command **free-constructors**; the (**co**)**datatype** command automatically registers the defined types, and
– *tycon*'s type arguments must all have sort *type*.

The command may be invoked on the same type constructor only once; otherwise an error occurs. After the command has been invoked on *tycon*, its constructors and the case combinator must not change any more. Moreover, no custom serialisation with **code-printing** may be used for the type and its primitive operations.[6]

The commands **activate-lazy-type** and **deactivate-lazy-type** activate or deactivate the laziness of the given type constructor in the generated code; **code-lazy-type** must previously have been called on the type constructor. After deactivation, the code generator uses the eager constructors again and the code equations are no longer transformed. The commands **activate-lazy-types** and **deactivate-lazy-types** do the same, but for *all* type constructors on which **code-lazy-type** has been called. All lazy types should be deactivated, e.g., when generating code to a lazy language like Haskell or when the conversion fails for debugging.

The command **print-lazy-types** prints a list of all lazy types with their lazy constructors and the lazy case combinator and whether they are currently active. For example, if lazy lists and streams have been registered as lazy, but only lazy lists are active, the output looks as follows:

```
llist: 'a llist = Lazy-llist (unlazy-llist: 'a llist-lazy lazy)
  and 'a llist-lazy = LNil-Lazy | LCons-Lazy 'a 'a llist
    for case: case-llist-lazy
stream (inactive): 'a stream = Lazy-stream (unlazy-stream: 'a stream-lazy lazy)
  and 'a stream-lazy = SCons-Lazy 'a 'a stream
    for case: case-stream-lazy
```

3.2 Examples

We have already presented some running examples with lazy lists. We now switch to binary trees as they demonstrate the effects of sharing more clearly. To keep things simple, we use unlabelled binary trees of possibly infinite depth:

```
codatatype tree = L | Node tree tree (infix △ 900)
code-lazy-type tree
```

and define three functions on them:

– *subtree p t* descends to *t*'s subtree at position *p*,
– *mk-tree n* constructs a balanced tree of the given depth, and
– *inftree* denotes the complete infinite binary tree.

[6] Isabelle's *list* type cannot thus be made lazy as it is mapped to target-language lists.

function *subtree* :: ⟨*bool list* ⇒ *tree* ⇒ *tree*⟩ **where**
  ⟨*subtree* [] *t* = *t*⟩
| ⟨*subtree* (*True* # *p*) (*l* △ *r*) = *subtree p l*⟩
| ⟨*subtree* (*False* # *p*) (*l* △ *r*) = *subtree p r*⟩
| ⟨*subtree* - *L* = *L*⟩

fun *mk-tree* :: ⟨*nat* ⇒ *tree*⟩ **where**
  ⟨*mk-tree 0* = *L*⟩
| ⟨*mk-tree* (*Suc n*) = (*let t* = *mk-tree n in t* △ *t*)⟩

primcorec *inftree* :: *tree* **where** ⟨*inftree* = *inftree* △ *inftree*⟩

Importantly, the function definitions use the constructors from the **codatatype** definition, even though we have registered *tree* as a lazy type. This way, all the existing reasoning support can be used for proofs about these functions. Nevertheless, the tree will be lazy when we execute the functions—thanks to the on-the-fly transformation of the code equations.

For example, **value** [*code*] ⟨*mk-tree 10*⟩ just outputs -. Next, we bind the tree to variable *t* and descend into *t* using *subtree*:

value [*code*] ⟨*let t* = *mk-tree 10*; - = *subtree* [*True*, *False*, *True*] *t in t*⟩

The evaluated expression is provably equal to *mk-tree 10* in HOL, but lazy evaluation in Standard ML now yields

$$((\text{-} \mathbin{\triangle} \text{-}) \mathbin{\triangle} (\text{-} \mathbin{\triangle} \text{-})) \mathbin{\triangle} ((\text{-} \mathbin{\triangle} \text{-}) \mathbin{\triangle} (\text{-} \mathbin{\triangle} \text{-})) \tag{5}$$

which is quite different from -. Although the result of the *subtree* function is discarded, the *subtree* function is evaluated nevertheless and—thanks to the sharing in the *lazy* type—this evaluation is preserved when the HOL term for *t* is reconstructed.

But wait! The evaluation result seems to indicate that *subtree* has evaluated the tree *t* completely to depth 3 instead of just along the one path. Shouldn't the result rather be the following?

$$(\text{-} \mathbin{\triangle} (\text{-} \mathbin{\triangle} \text{-})) \mathbin{\triangle} \text{-} \tag{6}$$

The answer to this puzzle is again sharing. In *mk-tree*'s definition, the recursive call is *let*-bound to *t*. The two children of every node in a tree created by *mk-tree* are therefore shared, and evaluation of one child thus implicitly propagates to the other child. If we eliminate the sharing, e.g., with the following code equation

$$\textit{mk-tree}\ (\textit{Suc}\ n) = \textit{mk-tree}\ n \mathbin{\triangle} \textit{mk-tree}\ n \tag{7}$$

then we get the expected output (6). These examples demonstrate that users should consider the computational consequences of their definitions—or of their code equations.

Apart from these sharing complications, CODELAZY makes it possible to use common idioms from lazy languages, in particular data-driven programming. For example, *lzip* (*up 0*) *xs* pairs all elements in the lazy list *xs* with their index. Similarly, a popular implementation of Eratosthenes's sieve is also executable:

$$sieve\ [] = [] \qquad sieve\ (x \cdot xs) = x \cdot sieve\ (lfilter\ (\lambda y.\ \neg\ x\ dvd\ y)\ xs)$$
$$primes = sieve\ (up\ 2)$$

value [*code*] ‹*ltake 10 primes*›

where *lfilter* is the filter function on lazy lists [3,10] with the code equations

$$lfilter\ P\ [] = []$$
$$lfilter\ P\ (x \cdot xs) = (\textsf{if}\ P\ x\ \textsf{then}\ x \cdot lfilter\ P\ xs\ \textsf{else}\ lfilter\ P\ xs)$$

and *ltake n xs* puts the first *n* elements of *xs* into a non-lazy list.

3.3 Limitations

We now turn to CodeLazy's limitations, both conceptually and implementation-wise. In the worst case, unaware users may run into errors at code generation time or find out that the generated code evaluates more than they had expected. But these limitations can never result in wrong output results because the transformations are proven correct within Isabelle.

On the conceptual level, it is crucial to understand that CodeLazy only introduces lazy *types*, not lazy *functions*. For the target languages Standard ML, OCaml and Scala, the evaluation behaviour for function calls remains call-by-value. So expressions in argument positions are evaluated before the function call. What CodeLazy changes is that the argument evaluation stops at the first lazy constructor. Consequently, the transformed code equations may evaluate more in the eager language than the original code equations would have evaluated in a lazy language. For example, the function *mk-tree* from the previous section has a recursive call that is not *syntactically* guarded by a constructor. Hence, when we call *mk-tree 10*, all ten recursive calls execute immediately and allocate ten lazy values in memory. This can be observed by tracing the evaluation with the following code equation:

$$mk\text{-}tree\ (Suc\ n) = (\textsf{let}\ \text{-} = Debug.flush\ n;\ t = mk\text{-}tree\ n\ \textsf{in}\ t \bigtriangleup t)$$

where *Debug.flush* from theory `HOL-Library.Debug` outputs the given value on the tracing channel. Then **value** [*code*] ‹*mk-tree 10*› shows ten tracing values, counting down from nine to zero. Conversely, the recursive call in (7) is syntactically guarded by △. If we add tracing to this equation, we only get one tracing value for nine as expected.

In general, lazy types work well if all (co)recursive calls are syntactically guarded by a constructor. The **primcorec** command [4] requires such a guarding constructor—except that it automatically unfolds *let*s, which makes its syntactic check slightly too weak for lazy types. We nevertheless conjecture that most corecursive definitions in practice syntactically guard the corecursive calls. Non-primitively corecursive definitions with **corec** [3] may contain unguarded calls (e.g., the *lfilter* function). These functions may therefore evaluate more than intended. For example, evalutating $lhd\ (lfilter\ even\ [1,\ 2,\ 3,\ hd\ [],\ 4])$ raises an exception because the *force* in *lhd* triggers the evaluation of the tail of the *lfilter*-ed list and *lfilter*'s code equation has an unguarded call that forces the evaluation of hd [].

Another limitation of our approach is that for pattern matching, only constructors registered with **free-constructors** may appear inside an eager constructor of a lazy type. For example, the patterns $[L]$, $x \bigtriangleup (L \bigtriangleup y)$, and $Suc\ n \cdot xs$ are all fine. Yet, code equations may contain patterns that violate this rule, e.g., the pattern $[set\ []]$ for matching on a singleton lazy list that contains the empty set (expressed with the non-free code constructor *set*). If such a pattern occurs on the left of a code equation, the transformation of the code equations will fail with an error. In practice, we have yet not encountered such a pattern, so this is a theoretical limitation for the moment.

On the implementation level, sharing may get lost when the pattern matches are eliminated and when the code is generated. For example, the algorithm eliminates *foo*'s pattern matches as shown below.

fun $foo :: \langle 'a\ llist \Rightarrow tree \Rightarrow 'a\ llist \rangle$ **where**
$\langle foo\ xs \quad L \quad = xs \rangle$
$|\ \langle foo\ [] \quad (l \bigtriangleup r) = [] \rangle$
$|\ \langle foo\ (x \cdot xs) \quad (l \bigtriangleup r) = xs \rangle$

$foo\ xs\ t =$
$\mathsf{case}\ xs\ \mathsf{of}\ [] \Rightarrow \mathsf{case}\ t\ \mathsf{of}\ L \Rightarrow []\ |\ \text{-} \Rightarrow []\ |\ x \cdot xs' \Rightarrow \mathsf{case}\ t\ \mathsf{of}\ L \Rightarrow x \cdot xs'\ |\ l \bigtriangleup r \Rightarrow xs'$

For $xs = x \cdot xs'$ and $t = L$, not the list xs as in the original code equation is returned, but it is reconstructed from the head x and the tail xs'. So $ys = foo\ xs\ L$ and xs are not shared any more, only their tails are. Such a difference might be observable in run-time and memory consumption. We believe that our algorithm could in principle be adapted to prevent such sharing loss. This is left as future work.

Sharing is also lost when the code generator introduces unit closures for top-level bindings. For *inftree*, e.g., it generates the Standard ML code

```
fun inftree () =
  Lazy_tree (Lazy.lazy (fn _ => Node_Lazy (inftree (), inftree ())));
val inftree = inftree ();
```

Consequently, we cannot observe the sharing when descending into *inftree* as we did with *mk-tree* in (5).

Apart from code generation, two more evaluators use the code equations: normalisation by evaluation and term rewriting. For term rewriting, lazy types work similarly to what we have described here, although the evaluation order may be slightly different. Normalisation by evaluation does not offer any control over the evaluation order and is therefore unsupported.

4 Related Work

Letouzey [8] has implemented a similar transformation for coinductive data types in Coq's OCaml code generator. His transformation is simpler than ours because

Coq ensures syntactic guardedness and patterns have already been converted into case combinators. Despite guardedness in Coq, he too observed the problem of unnecessary function argument evaluations as discussed in §3.3 and suggest to manually add *delay*s and *force*s to the generated code. In comparison, CODELAZY is superior in two respects: First, Letouzey's transformation, which is run upon every code generation, is not verified and therefore part of the trusted code base. In contrast, we only trust the code adaptation for the *lazy* types, which can be inspected once and for all. Second, Letouzey's transformation has only been implemented for codatatypes whereas CODELAZY can be used for any algebraic type.

Lochbihler and Maximova [11] have developed a stream fusion library in Isabelle for inductive and coinductive lists. Stream fusion may introduce laziness as a side effect. Stream fusion could handle the example from the introduction where the producer *up* and the consumer **!** are next to each other in a single code equation, but it does not work in general. Moreover, users must write their functions in a fusable form or manually prove an appropriate characterisation. In contrast, CODELAZY requires only a single **code-lazy-type** command and the rest is automated.

Converting pattern-matches into case combinators has a long history in HOL-based provers. Slind's TFL package [15] converts pattern matches into a decision tree of case combinators. Isabelle's **recdef** package implements a similar algorithm. Noschinski's converter **case-of-simps** combines a set of pattern-matching equations into one with case combinators on the right. All these implementations assume that the patterns do not overlap and they eliminate all patterns. Our pattern elimination algorithm is more flexible: (i) It is possible to eliminate only some patterns instead of all. This makes it possible to handle non-algebraic pattern matches in unaffected positions. (ii) It can handle overlapping and missing patterns, using the order of equations for disambiguation.

Overlapping and missing patterns are also handled by a number of algorithms that compile pattern matches to decision trees [1, 12–14]. All these algorithms eliminate all pattern matches. So none of them were directly applicable to our problem.

Tuerk et al. [18] suggested a new encoding for pattern matches in the HOL4 prover that avoids the exponential blow-up of decision trees, from which our algorithm also suffers. It would be interesting to port this representation to Isabelle/HOL and integrate it with Isabelle's code generator. Then, our algorithm for eliminating pattern matches could be much simpler.

5 Conclusion

CODELAZY brings lazy evaluation to algebraic types in Isabelle/HOL by adding suspensions to their code representation and forcing to the functions' code equations. These transformations preserve the partial correctness guarantee of the code generator as Isabelle's kernel checks them. This is possible as HOL does not specify the evaluation order. Making a type lazy does not affect its logical representation and is thus transparent to user definitions, theorems, and proofs.

Adding the forcing often requires to eliminate pattern matching from a set of equations. Our algorithm handles overlapping and missing patterns and can eliminate some patterns while leaving others. Isabelle's other pattern-matching elimination algorithms lack these features and this lack has caused problems [7,17]. It should not be too hard to replace the algorithm in `Code_Target_Nat` that eliminates pattern matches on *nat*s and the algorithm behind **case-of-simps** from `Simps_Case_Conv`.

References

1. M. Baudinet and D. MacQueen. Tree pattern matching for ML. Technical report, AT&T Bell laboratories, 1985. `https://www.classes.cs.uchicago.edu/archive/2011/spring/22620-1/papers/macqueen-baudinet85.pdf`.
2. J. Biendarra, J. C. Blanchette, A. Bouzy, M. Desharnais, M. Fleury, J. Hölzl, O. Kunčar, A. Lochbihler, F. Meier, L. Panny, A. Popescu, C. Sternagel, R. Thiemann, and D. Traytel. Foundational (co)datatypes and (co)recursion for higher-order logic. In C. Dixon and M. Finger, editors, *FroCoS 2017*, volume 10483 of *LNCS*, pages 3–21. Springer, 2017.
3. J. C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel. Friends with benefits: Implementing corecursion in foundational proof assistants. In H. Yang, editor, *ESOP 2017*, LNCS, pages 111–140. Springer, 2017.
4. J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In G. Klein and R. Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 93–110. Springer, 2014.
5. J. C. Blanchette, F. Meier, A. Popescu, and D. Traytel. Foundational nonuniform (co)datatypes for higher-order logic. In *LICS 2017*, pages 1–12. IEEE, 2017.
6. F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.
7. F. Hellauer. Code_Target_Nat and threefold pattern matching. Isabelle mailing list, `https://lists.cam.ac.uk/mailman/htdig/cl-isabelle-users/2017-May/msg00032.html`, May 2017.
8. P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, 2004.
9. A. Lochbihler. Fast machine words in Isabelle/HOL. In *ITP 2018*, LNCS. Springer, 2018. To appear.
10. A. Lochbihler and J. Hölzl. Recursive functions on lazy lists via domains and topologies. In G. Klein and R. Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS (LNAI)*, pages 341–357. Springer, 2014.
11. A. Lochbihler and A. Maximova. Stream fusion for Isabelle's code generator. In C. Urban and X. Zhang, editors, *ITP 2015*, volume 9236 of *LNCS*, pages 270–277. Springer, 2015.
12. L. Maranget. Compiling pattern matching to good decision trees. In *ML 2008*, pages 35–46. ACM, 2008.
13. M. Pettersson. A term pattern-match compiler inspired by finite automata theory. In *CC 1992*, pages 258–270. Springer, 1992.
14. P. Sestoft. ML pattern match compilation and partial evaluation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, pages 446–464. Springer, 1996.

15. K. Slind. Function definition in higher-order logic. In *TPHOLs 1996*, LNCS, pages 381–397. Springer, 1996.
16. P. Stoop. A compiler for lazy datatypes in Isabelle/HOL. Bachelor's thesis, Department of Computer Science, ETH Zurich, 2017.
17. R. Thiemann. Problems with code-generator. Isabelle mailing list, `https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2016-September/msg00078.html`, September 2016.
18. T. Tuerk, M. O. Myreen, and R. Kumar. Pattern matches in HOL: A new representation and improved code generation. In C. Urban and X. Zhang, editors, *ITP 2015*, volume 9236 of *LNCS*, pages 453–468. Springer, 2015.
19. P. Wadler, W. Taha, and D. MacQueen. How to add laziness to a strict language without even being odd. In *Workshop on Standard ML*, 1998.