Stream Fusion for Isabelle's Code Generator

Rough Diamond

Andreas Lochbihler and Alexandra Maximova

Institute of Information Security, Department of Computer Science, ETH Zurich, Switzerland

Abstract. Stream fusion eliminates intermediate lists in functional code. We formalise stream fusion for finite and coinductive lists in Isabelle/HOL and implement the transformation in the code preprocessor. Our initial results show that optimisations during code extraction can boost the performance of the generated code, but the transformation requires further engineering to be usable in practice.

1 Introduction

Over the last decade, code extraction spurred interest in writing executable specifications rather than abstract models in theorem provers [7,12]. For example, it is now possible to verify a conference management system in a prover and compile the model into a usable implementation [7]. Yet, satisfactory performance is hard to achieve. Existing work on efficiency [8,9] focuses on making efficient data structures available in the prover. The potential of optimisation during code extraction has been neglected so far.

Code extraction can boost efficiency in two ways. On the one hand, the generated code can use optimised libraries of the target language. To that end, one specifies manually how types and functions in the logic are mapped to the library. Such a mapping is unverified, i.e., the mapping and the libraries become part of the trusted code base. This is against the spirit of verification and should thus be avoided. On the other hand, extraction itself can transform and optimise the code. This seems sensible for three reasons. First, the transformations and the verification are carried out in the same formal framework. This ensures that they fit together. Second, the extractor can exploit the proven theorems (e.g., invariants, congruences) for optimisation. This knowledge gets lost during translation, as only the definitions are extracted. Thus, the target language compiler cannot exploit it. Third, the evaluation order and strictness requirements of the logic may be weaker than in the target language. This gives the extractor more freedom.

There is little benefit in re-implementing in the extractor all optimisations of the target language. Instead, one should focus on transformations that enable more optimisations in the target language. Fusion techniques [1,4,13] are a good candidate, as they transform a function designed for high-level proofs into one designed for optimisation.

In this paper, we focus on stream fusion (see §2 for an introduction), as it is more powerful than other fusion techniques [1,3]. We have formalised stream fusion in Isabelle/HOL with its restrictive type system for finite and coinductive lists and implemented the transformation in the code generator (§3). It is designed such that fusion affects neither definitions nor proofs in applications. Our evaluation on micro-benchmarks shows that the transformation improves the run time of the generated code by 24 % to 93 % (§4). Unfortunately, the transformation hardly triggers for applications at present. We have identified the obstacles and discuss how they could be overcome (§6).

The formalisation and implementation are available online [11].

2 Background on Stream Fusion

Stream fusion [1,2,3] transforms programs to enable optimisations. Consider, for example, the program $\mathit{sum\text{-}odd\text{-}sq}\ n = \mathit{sum}\ (\mathit{map}\ \mathit{sq}\ (\mathit{filter}\ \mathit{odd}\ [1\,..\,n]))$. The function *sum-odd-sq* computes the sum of the squares (computed by *sq*) of all odd numbers up to n. The definition is designed for proving, as it composes functions like building blocks, so proofs can use the existing lemmas about them. Yet, if *sum-odd-sq* is implemented as given, three lists are allocated at run-time: all numbers up to n, all odd numbers up to n, and their squares. In lazy languages, the lists are not allocated as a whole, but the cells still are. Ideally, no list would be needed at all. But as *sum*, *map*, and *filter* are recursive, compilers are unlikely to inline and optimise them aggressively.

Stream fusion transforms *sum-odd-sq* such that there is only one recursive function left. Then, subsequent optimisations like inlining and call specialisation can get rid of the intermediate allocations. To that end, it replaces a list of type $\alpha\ \mathit{list}$ by a stream, which consists of a generator $g :: \sigma \Rightarrow (\alpha,\ \sigma)\ \mathit{step}$ and a state $s :: \sigma$. Here, *step* has three constructors: *Done* indicates the end of the stream, $\mathit{Yield}\ x\ s$ produces the next element x and a new state s, and $\mathit{Skip}\ s$ represents a stuttering step. Lists and streams are linked via two functions *stream* and *unstream*, which satisfy $\mathit{unstream}\ (\mathit{stream}, xs) = xs$.

$$\mathit{stream}\ [\,]\ =\ \mathit{Done} \qquad\qquad \mathit{stream}\ (x \cdot xs)\ =\ \mathit{Yield}\ x\ xs$$

$$\begin{aligned}\mathit{unstream}\ (g,s) = (\mathit{case}\ g\ s\ \mathit{of}\ \mathit{Done} &\Rightarrow [\,] \quad \mid \mathit{Skip}\ s' \Rightarrow \mathit{unstream}\ (g,s')\\ \mid \mathit{Yield}\ x\ s' &\Rightarrow x \cdot \mathit{unstream}\ (g,s'))\end{aligned}$$

Functions on lists fall into three groups: producers such as $[\_\,..\,\_]$ return a list, consumers like *sum* take a list, and transformers (e.g., *filter* and *map*) do both. Every such function f has a stream counterpart $f\mathit{S}$. For example, *filterS* transforms a generator g into another generator $\mathit{filterS}\ P\ g$ as follows.

$$\begin{aligned}\mathit{filterS}\ P\ g\ s = (\mathit{case}\ g\ s\ \mathit{of}\ \mathit{Done} &\Rightarrow \mathit{Done} \quad \mid \mathit{Skip}\ s' \Rightarrow \mathit{Skip}\ s'\\ \mid \mathit{Yield}\ x\ s' &\Rightarrow \mathit{if}\ P\ x\ \mathit{then}\ \mathit{Yield}\ x\ s'\ \mathit{else}\ \mathit{Skip}\ s')\end{aligned}$$

A fusion equation of the form $f\ xs = \mathit{unstream}\ (f\mathit{S}\ \mathit{stream}, xs)$ links f and $f\mathit{S}$, e.g., $\mathit{filter}\,P\,xs = \mathit{unstream}\ (\mathit{filterS}\ P\ \mathit{stream}, xs)$. Equations for producers and consumers omit *stream* and *unstream* as in $[n\,..\,m] = \mathit{unstream}\ ([\_\,..\,m]_S, n)$ and $\mathit{sum}\,xs = \mathit{sumS}\ \mathit{stream}\,xs$.

The stream fusion transformation operates in two steps. First, it replaces all list functions with stream functions using the fusion equations. This introduces conversions from streams to lists and back. Second, it eliminates adjacent conversions as in $\mathit{unstream}\ (f_1\ \mathit{stream}, \mathit{unstream}\ (f_2\ g, s))$, i.e., we get $\mathit{unstream}\ (f_1\ (f_2\ g), s)$. In the end, only functions on streams should be left. Coutts [1] identifies sufficient conditions for this to happen. Among others, only consumers of a stream may be recursive and transformers must pass *Skip*s along unchanged. Then, later compiler stages can inline the functions and eliminate the step constructors and thereby the unnecessary allocations.

Note that the second step can change the type σ of the state for f_1. That is why streams actually have the existential type $\exists\sigma.\ (\sigma \Rightarrow (\alpha,\ \sigma)\ \mathit{step}) \times \sigma$. Thus, a consumer or transformer cannot examine the state type of the stream it takes. Hence, it can use the state only via the supplied generator, i.e., it behaves the same for all state types.

Transforming a function on lists to one on streams must currently be done manually. So, stream fusion requires that the programmer uses only library functions for which the setup has been provided. This restriction applies to other fusion techniques, too [4,13].

3 Formalising and Performing Stream Fusion in Isabelle/HOL

Due to the restriction to pre-defined library functions, it is sensible to express fusion in HOL and prove it correct. Otherwise, the generated code must use those library functions of the target language, provided that they are available (we only know of list implementations in GHC [2,3]). Such adaptations exist partly, e.g., for the Haskell function *concatMap*, but they are unverified and error-prone, so it is better to avoid them.

Unfortunately, stream fusion cannot be formalised as is in HOL for two reasons. First, HOL does not have existential type quantifiers. Thus, the type variable σ cannot be hidden in the stream type. Consequently, the elimination of adjacent occurrences of *stream* and *unstream* cannot be expressed as an equality in HOL, because the state type changes. Neither would Coutts' induction proof over types with a logical relation [1] work, as HOL types are not syntactic. Second, stream fusion is designed for coinductive sequence types, as generators need not terminate. However, the formalisation should support finite lists, too, as they are the workhorse in Isabelle/HOL. Even coinductive lists [10], which may be infinite, pose a definitional challenge, as a generator might always return *Skip*, i.e., it refuses to decide whether the list ends. In a domain-theoretic setting, *unstream* could return undefined, but in HOL it must make a choice.

We avoid the first problem by changing the format of the fusion equations. In the former $f\,xs = \mathit{unstream}\,(\mathit{fS}\,\mathit{stream}, xs)$, we instantiate xs with $\mathit{unstream}\,(g,s)$ and eliminate the *stream*-*unstream* pair immediately in the equation. Thus, our format

$$f\,(\mathit{unstream}\,(g,s)) = \mathit{unstream}\,(\mathit{fS}\,g,s) \tag{1}$$

avoids *stream* entirely. Both formats are equivalent, as we get the standard equation back by setting $g = \mathit{stream}$ and $s = xs$ and rewriting with $\mathit{unstream}\,(\mathit{stream}, xs) = xs$. In our example, we have $[x..y] = \mathit{unstream}\,([\_..y]_S, x)$ and $\mathit{filter}\,P\,(\mathit{unstream}\,(g,s)) = \mathit{unstream}\,(\mathit{filterS}\,P\,g,s)$ and $\mathit{sum}\,(\mathit{unstream}\,(g,s)) = \mathit{sumS}\,g\,s$.

We address the second problem by defining two subtypes of generators. First, *terminating* generators that always reach *Done* after finitely many iterations. Second, *productive* generators whose iteration contains only finitely many consecutive *Skip*s. The subtypes are defined using **typedef** and implemented via data refinement in the code generator [5]. For finite lists, we define *unstream* on terminating generators by well-founded recursion. Thus, well-founded induction is our proof principle for the fusion equations. For coinductive lists, we define *unstream* on arbitrary generators as a least fixpoint in the domain of functions on prefix-ordered lists [10], and proofs are by fixpoint induction. Thus, *unstream* interprets infinitely many consecutive *Skip*s as the end of the list. Additionally, we lift *unstream* to the type of productive generators, as some functions have fusible implementations only for productive generators. For example, concatenation $++$ of two coinductive lists is fusible only if the first list has a productive generator. Otherwise, the fusion equation $\mathit{unstream}\,(g_1,s_1) ++ \mathit{unstream}\,(g_2,s_2) = \mathit{unstream}\,(\mathit{appendS}\;g_1\;g_2\;s_2, s_1)$ does not hold: for $g_1 = \mathit{Skip}$, the left hand side is $\mathit{unstream}\,(g_2,s_2)$, but the right hand side equals $[]$, as *appendS* must pass *Skip*s along.

We have defined stream versions for the fusible list functions in Isabelle/HOL's list library, i.e., 4 producers, 17 transformers, and 13 consumers, and proved fusion equations for them. For *concatMap*, we also formalised the *flatten* operator from [3], which is easier to optimise. The consumers were the easiest, as they can be defined in terms of their list counterpart and *unstream*. The fusion equation and the recursive

code equations were proved automatically from the definition. Producers and transformers are not recursive either, but the proofs of termination and the fusion equation require inductions. For coinductive lists, we have 3 producers, 10 transformers, and 7 consumers. When possible, they come in two versions for productive and arbitrary generators. Proofs are by induction on productivity and fixpoint induction, respectively.

The stream fusion transformation itself is implemented as a rewrite procedure in the code generator. Its preprocessor invokes the procedure on all subexpressions of the right-hand side of each code equation. The procedure tries to rewrite the given expression with the fusion equations. It succeeds only if there are no *unstream* functions left at the end; otherwise, the transformation is discarded for this invocation. Our format (1) for the fusion equations ensures that the rewriting terminates, as the *unstream*s are pushed from producers outwards through transformers to consumers. The check for left-over *unstream*s ensures that fusion transformation is complete, as only consumers can eliminate the *unstream*s that producers have introduced. It does not seem sensible to leave *unstream*s in the code, although other fusion systems do so [3,4]. In our setting, the target language compiler does not know that *stream* and *unstream* cancel out. Thus, the conversions would end up in the compiled code and might slow down the execution.

Our implementation is extensible. Users can register new *unstream* functions for other sequence types and new fusion equations for their constants. Overlapping fusion equations are tried in the order of registration. This allows us to use a specialised fusion equation for *flatten* when the inner generator does not depend on the outer's state.

4 Evaluation

To evaluate the potential of stream fusion in Isabelle/HOL, we applied it to three micro-benchmarks (enum, nested, merge) from [3]. They all consist of folding addition on integers over lists generated by *concatMap* and $[\_..\_]$, i.e., they are designed to demonstrate the potential benefit of fusion. We generated Haskell and SML code with fusion enabled and disabled, and compiled it under GHC, PolyML, and mlton. Table 1 lists the run times averaged over ten runs (the parameter n has been set to 10 000 for enum and merge and to 1000 for nested). The measurements were performed on a 64-bit 2.4 GHz Intel i7-3630QM with 16 GB of RAM running Ubuntu Linux 12.04 LTS.

Surprisingly, the Haskell code with stream fusion enabled is slower than without. By looking at GHC's intermediate representation of the programs, we discovered that GHC does not eliminate all *step* constructors, because it does not specialise the consumer *foldlS* to the given combination of transformers and producers. Apparently, *foldlS* itself being recursive prevents the transformation. We manually applied the static argument transformation (SAT) to the generated consumer code such that the recursion occurs only in a nested function as shown in Figure 1. Then, the specialisation happens and stream fusion enables run time improvements between 31 % and 42 % (row fusion+SAT in Table 1). Unfortunately, Isabelle's code generator cannot generate recursive subfunctions, although this can be expressed with local contexts in Isabelle/HOL itself.

The SML tests show that heavily optimising compilers like mlton (approx. 93 % faster) profit from stream fusion more than less optimising ones like PolyML (24 % to 32 % faster). The manual SAT hardly affects PolyML and mlton as the differences are not statistically significant. Note that the folding in the test cases ensures that everything

compiler	GHC 7.8.4 with `-O3`			PolyML 5.5.2			mlton 20100608		
micro-benchmark	enum	nested	merge	enum	nested	merge	enum	nested	merge
no fusion	1.33	5.24	1.38	16.2	60.1	16.6	5.19	30.4	5.48
fusion	1.53	5.61	1.48	12.3	41.1	12.3	.395	1.89	.392
fusion+SAT	.918	3.05	.934	12.3	41.1	12.3	.388	1.90	.389

Table 1. Run times in seconds averaged over ten runs; the relative standard deviation is $< 2.2\,\%$.

```
foldlS g f z s =
  (case generator g s of {
     Done -> z;
     Skip a -> foldlS g f z a;
     Yield a sa -> let { za = f z a; }
       in Prelude.seq za (foldlS g f za sa); });
```

```
foldlS g f = go
  where { go z s = (case generator g s of {
                      Done -> z;
                      Skip sa -> go z sa;
                      Yield a sa -> let { za = f z a; }
                        in Prelude.seq za (go za sa); })};
```

Fig. 1. Haskell code for *foldlS* as generated by Isabelle (left) and after manually applying SAT (right). The cast operator *generator* applies a terminating generator to a state.

gets evaluated eventually. Thus, the performance gains are due to saving allocations and enabling subsequent optimisations. In particular, the effects of laziness introduced by stream fusion can be neglected. Fusion replaces strict lists with streams a.k.a. lazy lists, i.e., it introduces laziness. This can result in huge savings, as we noted previously [9].

5 Related Work

Coutts [1] introduced stream fusion for Haskell and identified sufficient conditions for stream fusion being an optimisation. He calls our fusion equations "data abstraction properties" and proves some of them on paper, but his implementation uses the traditional format. He justifies eliminating *stream-unstream* pairs by induction over type syntax and invariants preserved by a fixed set of library functions.

Recently, Farmer et al. [3] showed that stream fusion outperforms other fusion techniques [4,13] when *concatMap* receives special treatment. Unlike in GHC's RULES system, the fusion equations necessary for that can be directly expressed in Isabelle.

Huffman [6] formalised stream fusion in Isabelle/HOLCF and proved fusion rules for seven functions on domain-theoretic lists. He focuses on proving Coutts' fusion equations correct, but does not implement the transformation itself. As HOLCF is incompatible with the code generator, we cannot use his work for our purposes.

Lammich's framework [8] transforms Isabelle/HOL programs such that they use use efficient data structures. It assumes that the user has carefully written the program for efficiency. So, it does not attempt to eliminate any intermediate data structures.

6 Conclusion and Future Work

We have formalised stream fusion in Isabelle/HOL and implemented it in its code generator. Our initial results show that transformations performed during code extraction from theorem provers can make the compiled code much faster.

In fact, our implementation is just a first step. The transformation works well on code written with fusion in mind as in [9]. Yet, it hardly triggers in ordinary user-space programs. For example, the termination checker CeTA [12] generates 38 K lines of Haskell code, but stream fusion applies only once. Two main issues prevent performing stream fusion more widely. First, we perform fusion only when a single code equation contains the complete chain from producers via transformers to consumers. That is, if the calls to the producer and consumer occur in different functions, the preprocessor cannot see this and fusion is not applied. Control operators and *let* bindings break the chain, too. The first issue can be addressed by improving the implementation. Transformations like *let* floating and inlining of non-recursive functions can help to bring fusible functions together. Care is needed to ensure that sharing is preserved. Currently, Isabelle's code preprocessor does not support such global transformations. Moreover, for Haskell, support for local recursive functions is desirable. We leave this as future work.

Second, list functions are often defined recursively, even if they can be expressed with list combinators. Stream fusion ignores them, as there is no automatic conversion to streams. Yet, this restriction applies to all fusion implementations we know. At the moment, users must either prove the alternative definition in terms of combinators or define the counterparts on streams themselves. Fortunately, the existing definitions (and proofs) remain unchanged, as such additions are local.

Acknowledgements We thank Joachim Breitner for helping with analysing the GHC compilation. He, Ralf Sasse, and David Basin helped to improve the presentation.

References

1. Coutts, D.: Stream Fusion: Practical shortcut fusion for coinductive sequence types. Ph.D. thesis, University of Oxford (2010)
2. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion: From lists to streams to nothing at all. In: ICFP'07. pp. 315–326. ACM (2007)
3. Farmer, A., Höner zu Siederdissen, C., Gill, A.: The HERMIT in the stream. In: PEPM'14. pp. 97–108. ACM (2014)
4. Gill, A., Launchbury, J., Peyton Jones, S.L.: A short cut to deforestation. In: FPCA'93. pp. 223–232. ACM (1993)
5. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: ITP'13. LNCS 7998, pp. 100–115. Springer (2013)
6. Huffman, B.: Stream fusion. Archive of Formal Proofs, formal proof development (2009), http://afp.sf.net/entries/Stream-Fusion.shtml
7. Kanav, S., Lammich, P., Popescu, A.: A conference management system with verified document confidentiality. In: CAV'14. LNCS 8559, pp. 167–183. Springer (2014)
8. Lammich, P.: Automatic data refinement. In: ITP'13. LNCS 7998, pp. 84–99. Springer (2013)
9. Lochbihler, A.: Light-weight containers for Isabelle: efficient, extensible, nestable. In: ITP'13. LNCS 7998, pp. 116–132. Springer (2013)
10. Lochbihler, A., Hölzl, J.: Recursive functions on lazy lists via domains and topologies. In: ITP'14. LNCS (LNAI) 8558, pp. 341–357. Springer (2014)
11. Lochbihler, A., Maximova, A.: Stream fusion in HOL with code generation. Archive of Formal Proofs, formal proof development (2014), http://afp.sf.net/entries/Stream_Fusion_Code.shtml
12. Sternagel, C., Thiemann, R.: Ceta 2.18. http://cl-informatik.uibk.ac.at/software/ceta/ (2014)
13. Svenningsson, J.: Shortcut fusion for accumulating parameters & zip-like functions. In: ICFP'02. pp. 124–132. ACM (2002)