Certified Optimisation of Stream Operations Using Heterogeneous Staging

James Lowenthal
University of Cambridge, UK
jl946@cam.ac.uk

Jeremy Yallop
University of Cambridge, UK
jdy22@cam.ac.uk

State-of-the-art multi-stage languages currently provide no guarantees beyond the well-typedness of the resulting code. For safety- and security-critical systems, formal proofs of correctness are necessary to establish confidence in the system. In this work, we embed C [5] within Agda [2] to enable mechanised proof over the staged C code. We then use this embedding to implement the Strymonas streams library [7] and prove the correctness of its transformations.

1 Introduction

Staging [9] is a generative metaprogramming technique that allows quoted code to be manipulated as data. Generative metaprogramming is often essential for performance [10], since it allows programs to incorporate optimisations based on domain-specific knowledge not available to compilers. State-of-the-art multi-stage languages (e.g., BER MetaOCaml [6]) are expressive and safe: code generators are written in a full-scale language, and well-typed generators are guaranteed to generate well-typed code.

However, existing languages have two key limitations. First, although they make guarantees about *static* semantics of generated programs, there are no guarantees of the *dynamic* semantics. Consequently, it is sometimes necessary to debug generated code; this is a difficult task, since the issue is caused indirectly by the generating program, which may no longer be running when the error is discovered. Recent work [1] generates specifications alongside code and then runs solvers to verify the specifications. However, where a specification is not met, debugging again involves examining the generating program.

Second, most multi-stage languages are *homogeneous*, where the generator and generated code use the same language. Most languages are not well-suited to both tasks; high-level languages are useful for code generators, while low-level languages are better targets for generated code.

We address both of these limitations, decoupling generating and generated languages in a system of *heterogeneous* staging (Figure 1). With our approach, the generating language may be arbitrarily expressive, while the low-level object language may have desirable performance properties. Where the host language is dependently-typed, generators can make arbitrary guarantees about generated code.

To illustrate our design, we first embed of a fragment of C [5] within the dependently-typed language Agda [2], then build a certified and heterogeneous reimplementation of the Strymonas streams library [7].

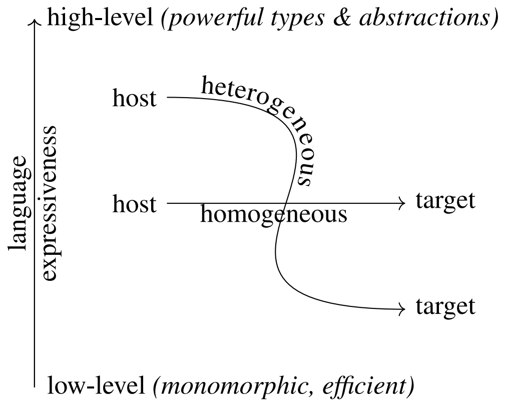
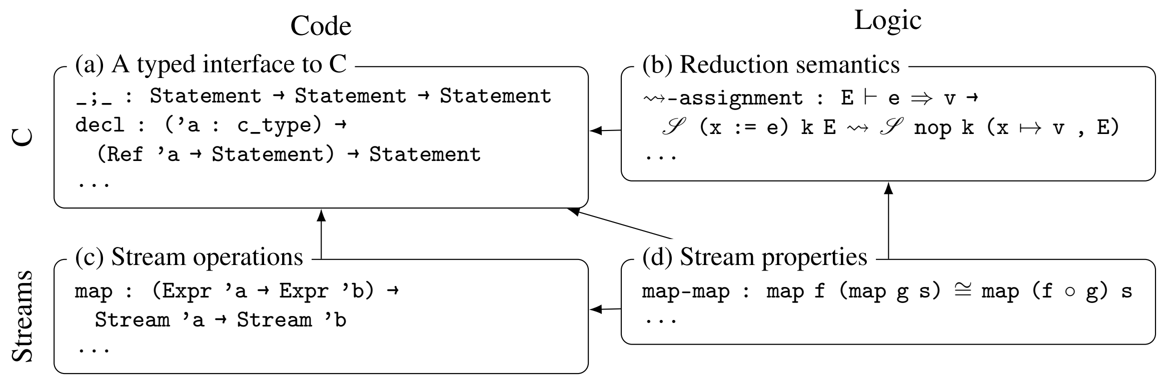

high-level (powerful types & abstractions)
host
heterogeneous
host
homogeneous
target
target
language expressiveness
low-level (monomorphic, efficient)

Figure 1: Generative metaprogramming: homogeneous vs. heterogeneous

Submitted to:
VPT 2020



```
_;_ : Statement → Statement → Statement
decl : ('a : c_type) →
  (Ref 'a → Statement) → Statement
...
```

```
⇝-assignment : E ⊢ e ⇒ v →
  𝒮 (x := e) k E ⇝ 𝒮 nop k (x ↦ v , E)
...
```

```
map : (Expr 'a → Expr 'b) →
  Stream 'a → Stream 'b
...
```

```
map-map : map f (map g s) ≅ map (f ∘ g) s
...
```

Figure 2: High-level architecture; arrows indicate dependencies between components.

2 Status

Figure 2 illustrates our high-level framework. Rows divide language-specific and streams-specific components; columns separate executable components from proofs.

2.1 Embedded Language and Semantics

Our fragment of C (Figure 2a) includes pure expressions, potentially side-effectful statements, and references. The tagless-final style [3] enables typed embeddings of domain-specific languages, and we use this to embed our fragment of C in Agda. Dynamic memory allocation, higher-order functions and recursion are excluded from our fragment, because they would dramatically increase the complexity of the fragment and the code generated by the Strymonas library does not require these constructs. These omissions make the embedded language Turing-incomplete and make termination decidable.

The embedding of a language and the encoding of its semantics should be separate, because there are multiple paradigms for expressing dynamic program behaviour, and proofs over dynamic semantics are not always required. From an abstract semantic model, properties can be proven across all conforming implementations of the language. We follow this principle and encode the CompCert C small-step reduction semantics [8] (Figure 2b) separately from our language specification.

We use two notions of semantic equivalence: expression and program. Expressions are equivalent if they resolve to the same value under all valid environments. Pure programs (statements) are equivalent if their execution converges or neither program terminates. From a suitable abstract semantic model, β- and η-equivalences of programs can be concluded.

2.2 Streams Library

The Strymonas library [7] uses staging and stream fusion [4] to implement the generality of stream processing with full elimination of overheads. The correctness of each transformation is shown with pen-and-paper proofs and examples. Using our embedding of C, we have reimplemented the Strymonas in Agda (Figure 2c). We currently support `fold`, `unfold`, `map`, `flatmap`, `filter`, and `take` operators, but not `zip`.

Streams are equivalent if the programs generated by folding over the streams are equivalent. That is,

$$s \cong t \equiv (\forall f\ z.\ \texttt{fold}\ f\ z\ s \cong_p \texttt{fold}\ f\ z\ t)$$

where $\cong_p$ is the program equivalence relation. With this definition, we prove natural stream equivalences (Figure 2d), such as $(\texttt{filter}\ (\lambda x.\ \texttt{true})\ s) \cong s$ and $(\texttt{map}\ f\ (\texttt{map}\ g\ s)) \cong \texttt{map}\ (f \circ g)\ s$.

3 Future Work

Our immediate focus for further work is enabling zip functionality using only host language pairs which are unpacked in the generated code. We will then show correctness by deriving natural properties of zip.

Heterogeneous staging typically uses a pretty-printer to delegate compilation of the generated code to a dedicated compiler. For complete end-to-end verification, the high-level dynamic semantics of the pretty-printer must be tied to the semantics of the delegate compiler. We propose that this can be partially achieved by proving that direct evaluation of the AST conforms to the necessary semantics, and that the pretty-printer only outputs this AST. A compiler that guarantees the same semantics should then be used, to ensure that the resulting binary also conforms.

Generative metaprogramming is often used to exploit domain-specific knowledge for performance gains. In our work, we have used knowledge of streams to enable generation of optimised code. To validate these optimisations, we will benchmark our streams library against prior work.

References

[1] Nada Amin & Tiark Rompf (2017): *LMS-Verify: Abstraction without Regret for Verified Systems Programming*. SIGPLAN Not. 52(1), p. 859–873, doi:10.1145/3093333.3009867.

[2] Ana Bove, Peter Dybjer & Ulf Norell (2009): *A Brief Overview of Agda - A Functional Language with Dependent Types*. In: Theorem Proving in Higher Order Logics, doi:10.1007/978-3-642-03359-9_6.

[3] Jacques Carette, Oleg Kiselyov & Chung-chieh Shan (2009): *Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages*. J. Funct. Program. 19(5), doi:10.1017/S0956796809007205.

[4] Duncan Coutts, Roman Leshchinskiy & Don Stewart (2007): *Stream fusion: from lists to streams to nothing at all*. In: ACM SIGPLAN International Conference on Functional Programming, doi:10.1145/1291151.1291199.

[5] Brian W. Kernighan & Dennis Ritchie (1978): *The C Programming Language*. Prentice-Hall.

[6] Oleg Kiselyov (2014): *The Design and Implementation of BER MetaOCaml - System Description*. In: Functional and Logic Programming, doi:10.1007/978-3-319-07151-0_6.

[7] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos & Yannis Smaragdakis (2017): *Stream Fusion, to Completeness*. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, doi:10.1145/3009837.3009880.

[8] Robbert Krebbers, Xavier Leroy & Freek Wiedijk (2014): *Formal C Semantics: CompCert and the C Standard*. In: Interactive Theorem Proving, doi:10.1007/978-3-319-08970-6_36.

[9] Tiark Rompf & Martin Odersky (2012): *Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs*. Commun. ACM 55(6), doi:10.1145/2184319.2184345.

[10] Tim Sheard (2001): *Accomplishments and Research Challenges in Meta-programming*. In Walid Taha, editor: Semantics, Applications, and Implementation of Program Generation, doi:10.1007/3-540-44806-3_2.