Static interpretation of higher-order modules in Futhark: functional GPU programming in the large

Martin Elsman, Troels Henriksen, Danil Annenkov, Cosmin Oancea

Static Interpretation of Higher-Order Modules in Futhark: Functional GPU Programming in the Large

MARTIN ELSMAN, University of Copenhagen, Denmark
TROELS HENRIKSEN, University of Copenhagen, Denmark
DANIL ANNENKOV*, Inria, France
COSMIN E. OANCEA, University of Copenhagen, Denmark

We present a higher-order module system for the purely functional data-parallel array language Futhark. The module language has the property that it is completely eliminated at compile time, yet it serves as a powerful tool for organizing libraries and complete programs. The presentation includes a static and a dynamic semantics for the language in terms of, respectively, a static type system and a provably terminating elaboration of terms into terms of an underlying target language. The development is formalised in Coq using a novel encoding of semantic objects based on products, sets, and finite maps. The module language features a unified treatment of module type abstraction and core language polymorphism and is rich enough for expressing practical forms of module composition.

CCS Concepts: • **Software and its engineering** → **Abstraction, modeling and modularity**; **Parallel programming languages**; **Functional languages**; **Modules / packages**; **Semantics**;

Additional Key Words and Phrases: modules, GPGPU, functional languages, compilers

ACM Reference Format:
Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. 2018. Static Interpretation of Higher-Order Modules in Futhark: Functional GPU Programming in the Large. *Proc. ACM Program. Lang.* 2, ICFP, Article 97 (September 2018), 31 pages. https://doi.org/10.1145/3236792

1 INTRODUCTION

Programming massively parallel hardware is increasingly becoming a central aspect of developing computational software kernels. Research in developing programming abstraction mechanisms for effectively utilizing massively parallel hardware has resulted in a number of available tools, ranging from low-level programming models, such as CUDA and OpenCL, over domain-specific languages and libraries, such as Accelerate [Chakravarty et al. 2011; McDonell et al. 2013], Obsidian [Claessen et al. 2012; Svensson 2011], Lift [Steuwer et al.

*Work done while at the University of Copenhagen.

Authors' addresses: Martin Elsman, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, mael@di.ku.dk; Troels Henriksen, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, athas@di.ku.dk; Danil Annenkov, GALLINETTE Research team, Inria, Nantes, France, danil.annenkov@inria.fr; Cosmin E. Oancea, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, coancea@di.ku.dk.

2475-1421/2018/9-ART97
https://doi.org/10.1145/3236792

2015, 2017], and CUBLAS, to full programming languages, such as SaC [Guo et al. 2011] and Futhark [Henriksen et al. 2014; Henriksen and Oancea 2014; Henriksen et al. 2017].

With software components that utilise massively parallel hardware getting larger and more complicated, an increasing need emerges in this area for good software structuring mechanisms and high reusability of developed software components. In the last decades, a number of abstraction mechanisms have been introduced that allow programmers to reason at a high level about the composition of developed software components. Such mechanisms include object-oriented class-based programming mechanisms, type class structuring mechanisms, as found in Haskell [Wadler and Blott 1989], module-languages as found in Standard ML [Milner et al. 1997] and OCaml [Leroy 1995], and extensions [Rossberg and Dreyer 2013] and combinations [Dreyer et al. 2007; White et al. 2015] thereof.

A common issue with the work on modules, and abstraction mechanisms in general, is that the mechanisms seldom give guarantees about their overhead. Often, the mechanisms introduce runtime overhead in terms of, for instance, dynamic method invocation, or packages (e.g., dictionaries or modules) being passed around at runtime. At best, an optimizing compiler can eliminate much of this overhead statically, but the interaction between the core language and the language for programming in the large is often subtle and without paying attention to the interaction mechanisms, it will be difficult for a compiler to provide guarantees that refactoring for increasing modularity will not affect performance.

One promising technique for solving some of these issues is the work on embedded domain-specific languages (EDSLs), which provide a structuring mechanism based on host language features for structuring and composing components. Examples of such approaches to orchestrate programming for massively parallel hardware include Accelerate, Obsidian, and Firepile [Nystrom et al. 2011], which are libraries, written in Haskell and Scala, for synthesizing code for running on GPGPUs. However, designing a well-working EDSL is not a trivial task as the developer needs to work hard, for instance to avoid unintended code duplication [Mainland and Morrisett 2010]. Another issue with EDSLs is that they can be difficult to incorporate in code generation pipelines as the language cannot necessarily be understood in isolation from its host-language, which may cause both practical problems and reasoning problems with respect to guaranteeing correctness of a compilation pipeline.

In this paper, we introduce the notion of higher-order static interpretation. The work is based on earlier work on static interpretation for Standard ML [Elsman 1999], which provides a technique for complete static elimination of modules, including first-order modules (called functors in Standard ML). The technique is similar to how C++ templates are eliminated at compile time with the difference that modules are type checked before they are compiled. The present paper extends the previous work to higher-order modules. We demonstrate the usefulness of the added functionality and argue for the need for higher-order modules and type abstraction. The module language is implemented on top of the monomorphic, first-order functional data-parallel language Futhark, which features a number of polymorphic second-order array combinators (SOACs) with parallel semantics, such as **map**, **reduce**, **scan**, and **filter**. The Futhark core language makes it possible to write data-parallel kernels that are often competitive with hand-written CUDA or OpenCL applications [Andreetta et al. 2016; Henriksen et al. 2016b, 2017; Larsen and Henriksen 2017]. We show that the practical implications of the module language give rise to highly reusable components, which, for instance, form the grounds of a Basis Library for Futhark.

Although the basic module language that we present is both elegant and simple, it features a number of derived forms, which makes the language practical while keeping the technical details minimal. Special derived forms allow programmers to declare certain

kinds of polymorphic higher-order functions and types. The declarations are turned into parameterised modules, which are then instantiated at the module level for each use.

An essential property of static interpretation of higher-order modules is that it is strongly normalizing, which is based on a logical-relation argument, similar to an argument for strong-normalisation of the simply-typed lambda-calculus [Tait 1967]. The important properties of the module language (e.g., normalisation of static interpretation) has been proven in Coq and both the Futhark compiler, featuring the full module language, and the Coq implementation are available online.

In summary, we claim the following novel contributions:

(1) We show how the concept of static interpretation can be extended to higher-order modules and present an implementation as part of a compiler for Futhark.
(2) We demonstrate the usefulness of a higher-order module system for GPU programming by presenting a number of modularised Futhark examples and by arguing (and demonstrating) that the modularisation has no influence on performance.
(3) Using a novel adaptation of a logical-relation argument, we show that static interpretation terminates and that target and source programs are typed consistently.
(4) We provide a formal development in Coq, which uses a novel encoding of semantic objects for modules based on products, sets, and finite maps. The encoding closely resembles the paper formalisation and the implementation for Futhark.

The paper is organised as follows. In the next section, we introduce the higher-order module language and present a motivating example demonstrating the use of higher-order modules in Futhark. In Section 3, we give an overview of how static interpretation turns higher-order modules into monomorphic core language Futhark code. In Section 4, we present the static semantics for the module language and show how the module language interacts with the core language. Section 5 presents the technicalities of static interpretation and Section 6 presents the main technical result, namely that static interpretation terminates and that the resulting program, from a type perspective, is consistent with the source program, which may possibly contain declarations and applications of higher-order modules. In Section 7, we briefly discuss the type inference algorithm. In Section 8, we present a number of derived forms for allowing programmers to use traditional syntax for polymorphic functions and function application for generating parameterised code at the module level and for instantiating this code also at the module level. In Section 9, we discuss the Coq development and in Section 10, we discuss the application of the module language in the concrete setting of the Futhark compiler. We present a few larger examples of Futhark modules and demonstrate that, in practice, the module language is a useful technology for improving code reuse and code abstraction and that it introduces no overhead with respect to performance of the generated code. In Section 11 and Section 12, we discuss related work and conclude.

2 THE MODULE LANGUAGE

The module language can be considered parameterised over a core language, which, for the purpose of the presentation, is a simple functional language. In the practical implementation, the core language is a full-fledged programming language for data-parallel programming featuring a number of second-order polymorphic array combinators (SOACs) with parallel semantics but with no support for user-defined polymorphic higher-order functions.

For the core language considered here, we assume denumerably infinite sets of *type identifiers* (*tid*), *value identifiers* (*vid*), and *module identifiers* (*mid*). For each of the above

$$
\begin{array}{lcl}
\mathit{mty} & ::= & \{\ \mathit{spec}\ \} \\
& | & \mathit{mtid} \\
& | & \mathit{mid} : \mathit{mty}_1 \to \mathit{mty}_2 \\
& | & \mathit{mty}\ \textbf{with}\ \mathit{longtid} = \mathit{ty} \\
\mathit{spec} & ::= & \textbf{val}\ \mathit{vid} : \mathit{ty} \\
& | & \textbf{type}\ \mathit{tid} \\
& | & \textbf{module}\ \mathit{mid} : \mathit{mty} \\
& | & \textbf{include}\ \mathit{mty} \\
& | & \mathit{spec}_1\ \mathit{spec}_2 \mid \epsilon
\end{array}
\qquad
\begin{array}{lcl}
\mathit{mexp} & ::= & \{\ \mathit{mdec}\ \} \\
& | & \mathit{mid} \mid \mathit{mexp}\,.\,\mathit{mid} \\
& | & \lambda \mathit{mid} : \mathit{mty} \to \mathit{mexp} \\
& | & \mathit{longmid}\ (\mathit{mexp}) \\
\mathit{mdec} & ::= & \mathit{dec} \\
& | & \textbf{type}\ \mathit{tid} = \mathit{ty} \\
& | & \textbf{module}\ \mathit{mid} = \mathit{mexp} \\
& | & \textbf{module type}\ \mathit{mtid} = \mathit{mty} \\
& | & \textbf{open}\ \mathit{mexp} \\
& | & \mathit{mdec}_1\ \mathit{mdec}_2 \mid \epsilon
\end{array}
$$

Fig. 1. Grammar for the module language excluding derived forms.

identifier sets X, we define the associated set of *long identifiers* Long$X$, inductively with X as the base set and *mid.longx* as the inductive case with *longx* $\in$ Long$X$ and *mid* being a module identifier. For the module language, we also assume a denumerably infinite set of *module type identifiers* (*mtid*). Long identifiers, such as $x.y.z$, allow users to use traditional dot-notation for accessing components deep within modules and the separation of identifier classes makes it clear in what syntactic category an identifier belongs.

The simple core language is defined by notions of *type expressions* (*ty*), *core language expressions* (*exp*), and *core language declarations* (*dec*):

$$
\begin{array}{lcl}
\mathit{ty} & ::= & \mathit{longtid} \mid \mathit{ty}_1 \to \mathit{ty}_2 \\
\mathit{exp} & ::= & \mathit{longvid} \mid \lambda \mathit{vid} \to \mathit{exp} \mid \mathit{exp}_1\ \mathit{exp}_2 \mid \mathit{exp} : \mathit{ty} \\
\mathit{dec} & ::= & \textbf{val}\ \mathit{vid} = \mathit{exp}
\end{array}
$$

The core language can be understood entirely in isolation from the module language except that long identifiers may be used to access values and types in modules. As will become apparent in the typing rules for the core language, the construct $\lambda \mathit{vid} \to \mathit{exp}$ binds *vid* in *exp* and the declaration of a value identifier in a **val**-declaration allows for other declarations to depend on it. For simplicity, composition of core language constructs are handled entirely by the module language and so is the possibility for declaring types. An important aspect here is that the dependency structure between declarations is not allowed to be cyclic.

The grammar for the module language is given in Figure 1. The module language is separated into a language for specifying module types (*mty*) and a language for declaring modules (*mdec*). The language for module types is a two-level language with sub-languages for specifying module components and for expressing module types. Similarly, the language for declaring (i.e., defining) modules is a two-level language for declaring module components and for expressing module manipulations. At the toplevel, a program is a module declaration, possibly consisting of a sequence of module declarations where later declarations may depend on earlier declarations. To resolve ambiguities, parentheses are allowed around module type expressions and around module expressions. As will become apparent from the typing rules, in declarations of the form $\mathit{mdec}_1\ \mathit{mdec}_2$, identifiers declared by mdec_1 are considered bound in mdec_2 (similar considerations hold for composing specifications and programs).

Without lack of generality, a number of additional constructs are supported by considering them derived forms, following [Milner et al. 1997] and [Rossberg et al. 2014]. Such derived forms include constructs for local module bindings, direct application of a module expression

to another module expression, constructs for opaque matching (in Standard ML terms), and support for type abbreviations in module type specifications:

$$\textbf{module}\ F\ (\ X : mty\ )\ =\ mexp \overset{\text{mdec}}{\Longrightarrow} \textbf{module}\ F = \lambda X : mty\ \rightarrow\ mexp \quad (1)$$

$$\textbf{local}\ mdec\ \textbf{in}\ mexp \overset{\text{mdec}}{\Longrightarrow} \textbf{open let}\ mdec\ \textbf{in}\ mexp \quad (2)$$

$$\textbf{let}\ mdec\ \textbf{in}\ mexp \overset{\text{mexp}}{\Longrightarrow} \{\ mdec\ \textbf{module}\ X = mexp\ \}\,.X \quad (3)$$

$$mexp_1\ (\ mexp_2\ ) \overset{\text{mexp}}{\Longrightarrow} \textbf{let module}\ F = mexp_1\ \textbf{in}\ F\ (mexp_2) \quad (4)$$

$$mexp : mty \overset{\text{mexp}}{\Longrightarrow} (\lambda X : mty\ \rightarrow\ X)\ (mexp) \quad (5)$$

$$\textbf{module}\ X : mty = mexp \overset{\text{mdec}}{\Longrightarrow} \textbf{module}\ X = mexp : mty \quad (6)$$

$$\textbf{module}\ F\ (\ X : mty_0\ ) : mty = mexp \overset{\text{mdec}}{\Longrightarrow} \textbf{module}\ F\ (\ X : mty_0\ ) = mexp : mty \quad (7)$$

$$\textbf{type}\ tid = ty \overset{\text{spec}}{\Longrightarrow} \textbf{include}\ (\{\ \textbf{type}\ tid\ \}\ \textbf{with}\ tid = ty) \quad (8)$$

Although the derived form of Equation 5, for matching a module expression to a module type expression, allows us to simplify the semantic treatment of the module language, we shall later present a specialised typing rule specifying how a module expression *mexp* is classified by a given module type expression *mty* under some environment. Type abbreviations in module type specifications are also considered a derived form (Equation 8) using an **include** construct and a **with**-constrained module type expression, which in itself is arguably more expressive than a type abbreviation, as it resembles a substitution and allows for reuse of module types. Another possible derived form introduces a construct for modeling the semantics of OCaml's **open** construct (not shown here), which opens a module in a local scope contrary to the semantics of **open** defined here, which resembles the semantics of **open** in Standard ML.

As described in more detail in Section 8, certain kinds of polymorphic higher-order functions and polymorphic types, as known from traditional higher-order polymorphic languages such as Standard ML, OCaml, and Haskell, can also be introduced as derived forms together with derived forms for instantiating such declared higher-order modules. By introducing derived forms only at the module level, we can be sure that all modules are indeed eliminated entirely at compile time.

We conclude this section by discussing a practical example, shown in Figure 2, in which the module language is used to extend, at library level, the set of SOACs supported in Futhark. Notice that the syntax for types is extended to also support *array types*, which take the form `[]`τ, for some τ, which may possibly also be an array type. Here, the aim is to implement segmented scan based on the natively supported **scan** bulk-parallel operator,[1] which is applied in the `main` function. The implementation starts by declaring a `Numeric` module type that exports a numeric type `t`, a zero element (of type `t`), and various numeric operations such as addition. A possible implementation is given by module `i32`, which instantiates `t` to 32-bit integers.

Next, the `Monoid` module type is declared to export a type `t`, together with a binary associative operator `op` with neutral element `ne:t`. A potential implementation for `Monoid`

[1] **scan** ⊙ e [a_1, ..., a_n] results in the array [e⊙a_1, ..., e⊙a_1 ⊙ ... ⊙a_n], where ⊙ is a binary associative operator with neutral element e. Segmented scan [Blelloch 1989, 1990] receives as extra argument a flag array, which records the start of a new segment with a one and zero otherwise, and results in a (flat) array obtained by applying **scan** on each segment of the input array.

```
module type Numeric = {
  type t
  val zero : t
  val + : t → t → t
}
module i32: Numeric with t = i32 = {
  type t   = i32
  let zero : i32 = 0i32
  let (x:i32) + (y:i32) = x intrinsics.+ y
}
module type Monoid = {
  type t              -- element type.
  val ne: t           -- Neutral element.
  val op: t → t → t -- Associative op.
}
module Plus(N : Numeric) : Monoid with t = N.t = {
  type t = N.t
  let ne : N.t = N.zero
  let op (a : N.t) (b : N.t) : N.t = a N.+ b
}
module SOACs = λE : {type t} → {
  ...
  module SgmScan (M : Monoid with t = E.t)
     : { val bulkop: []bool → []M.t → []M.t } = {
    let bulkop (flag: []bool) (arr: []M.t): []M.t = ...
  }
}
module SegSum = SOACs(i32).SgmScan(Plus(i32))
let main (flag : []bool) (arr : []i32) : []i32 =
  SegSum.bulkop flag arr
```

Fig. 2. Implementing segmented scan based on **scan** at library level in Futhark.

```
module type MT = {
  module F: (X:{ val b:i32 } → { val f:i32→i32 })
}
module H = λ(M:MT) → M.F { let b = 8 }
module Main =
  H ( { module F =
          λ(X:{ val b:i32 }) → { let f(x:i32) = X.b+x }
      })
let main (a:i32) : i32 = Main.f a
```

(a) Example source program.

```
val b = 8
```

```
val f = λ(x:i32) → b + x
```

```
val main = λ(a:i32) → f a
```

(b) Target language code.

Fig. 3. A (contrived) example demonstrating static interpretation in action. The program on the left is turned into the three snippets of declarations on the right.

is the parameterised module `Plus`, which receives a module parameter `N` implementing `Numeric` and instantiates `Monoid`'s `t`, `zero` and `op` with `N.t`, `N.zero` and `N.+`, respectively. Finally, new SOACs are derived from the ones natively supported by Futhark in the parameterised module `SOACs`, which receives as parameter a monoid `E` corresponding to the array element type, and declares an inner parameterised module for each new SOAC. For example, `SgmScan` takes as parameter a monoid-implementing module `M`, and implements the segmented-scan operator `bulkop` based on Futhark's **scan** (not shown). Function `main` computes a segmented scan with the plus operator on a (segmented) array of 32-bit integers.

3 EXECUTIVE SUMMARY

For the purpose of demonstrating static interpretation in action, consider the (contrived) example Futhark program in Figure 3a. Compared to the motivating example in Section 2, the program here makes no use of derived forms and it therefore serves well as a running example demonstrating the technique. The program declares a module type `MT` and a

higher-order module `H`, which is applied to a module containing a parameterised module `F`. The result of the module application is a module containing a function `f` of type `i32` → `i32`. The contained function is called in the `main` function with the input to the program.

The static semantics of the language, which we shall later define in terms of a set of elaboration rules, provides evidence that the code in Figure 3a is well-typed with the intended guarantee that the program can be executed without dynamic type errors occurring.

Static interpretation for the example in Figure 3a partially evaluates the program to achieve the three code snippets shown in Figure 3b. Each of the three code snippets contains monomorphic target code, which can be composed, analysed, and compiled without any module language considerations. This feature provides the target language implementor with the essential meta-level abstraction property that the module language features are orthogonal to the domain of the source language. In other words, the target language implementor need not worry that the added modularity constructs will have any influence on composing the generated code. This property is essential for targeting data-parallel architectures, such as GPUs, but may also be essential for a variety of other domain specific languages, including languages for financial contracts [Peyton Jones et al. 2000], probabilistic programming, hardware design, signal processing, and formal reasoning.

At the technical side, static interpretation takes care of renaming identifiers properly to resolve scope-issues. In Section 6, we demonstrate that if the source language is well-typed then (1) static interpretation will generate target language code and (2) the generated target language code will be consistent with the source language code with respect to the types of declared identifiers. An essential aspect of the approach is that the generation of target code is shown to be terminating using a logical-relations argument, which is properly formalised and backed up by a proof in Coq. For limiting the scope of the presentation, we shall not here provide the reader with a direct dynamic semantics for the module language. Instead, the dynamic semantics for the language can be understood as the composition of static interpretation and the dynamic semantics of the target language. Providing a direct dynamic semantics for the module language, along the lines of the dynamic semantics of Milner et al. [1997], would be straightforward and would resemble closely the rules for static interpretation.

In what follows, we develop the precise formalism that makes static interpretation possible, including a proper treatment of abstract types at the module language level.

4 STATIC SEMANTICS

The static semantics that we present here elaborates well-typed syntactic constructs into so-called semantic objects, which are based on well-established mathematical constructs such as products, finite maps, and sets. This style of elaboration approach resembles closely the style of elaboration set forward by Milner et al. [1997], although modified to use universal and existential quantification for treating certain objects equal up-to α-renaming, as set forward by Russo [1999] and Elsman [1999].

For the static semantics, we assume a denumerably infinite set TSet of *type variables* (t). A *semantic type* (or simply a type), ranged over by τ, takes the form:

$$\tau \quad ::= \quad t \mid \tau_1 \to \tau_2$$

Types relate straightforwardly to syntactic types with the difference that syntactic types contain type identifiers and semantic types contain type variables. This difference is essential in that it enables the support for type parameterisation and type abstraction.

$$
\begin{array}{rcl}
E = (TE, VE, ME, G) & \in & \mathrm{Env} = \mathrm{TEnv} \times \mathrm{VEnv} \times \mathrm{MEnv} \times \mathrm{MTEnv} \\
ME & \in & \mathrm{MEnv} = \mathrm{Mid} \xrightarrow{\text{fin}} \mathrm{Mod} \\
M & \in & \mathrm{Mod} = \mathrm{Env} \cup \mathrm{FunSig} \\
F = \forall T.(E, \Sigma) & \in & \mathrm{FunSig} = \mathrm{Fin}(\mathrm{TSet}) \times \mathrm{Env} \times \mathrm{MTy} \\
\Sigma = \exists T.M & \in & \mathrm{MTy} = \mathrm{Fin}(\mathrm{TSet}) \times \mathrm{Mod} \\
G & \in & \mathrm{MTEnv} = \mathrm{MTid} \xrightarrow{\text{fin}} \mathrm{MTy}
\end{array}
$$

Fig. 4. Module language semantic objects. Parameterised module types (F) and module types (Σ) are parameterised over finite sets of type variables (written $\mathrm{Fin}(\mathrm{TSet})$), ranged over by T.

At the core level, a *value environment* (VE) maps value identifiers (*vid*) to types and a *type environment* (TE) maps type identifiers (*tid*) to types.

The module language semantic objects are shown in Figure 4. The semantic objects constitute a number of mutually dependent inductive definitions. An *environment* (E) is a quadruple (TE, VE, ME, G) of a type environment TE, a variable environment VE, a *module environment* (ME), which maps module identifiers to modules, and a *module type environment* (G), which maps module type identifiers to module types. A *module* is either an environment E, representing a non-parameterised module, or a *parameterised module type* F, which is an object $\forall T.(E, \Sigma)$, for which the type variables in T are considered bound. A *module type* (Σ) is a pair, written $\exists T.M$, of a set of type variables T and a module M. In a module type $\exists T.M$, type variables in T are considered bound and we consider module types identical up-to renaming of bound variables and removal of type variables that do not appear in M. When T is empty, we often write M instead of $\exists \emptyset.M$. We consider module function types $\forall T.(E, \Sigma)$ identical up-to renaming of bound type variables and removal of type variables in T that do not occur free in (E, Σ).

When X is some tuple and when x is some identifier, we shall often write $X(x)$ for the result of looking up x in the appropriate projected finite map in X. Moreover, when long$x$ is some long identifier, we write $X(\text{long}x)$ to denote the lookup in X, possibly inductively through module environments.

When X and Y are finite maps, the *modification* of X by Y, written $X + Y$, is the map with $\mathrm{Dom}(X + Y) = \mathrm{Dom}\ X \cup \mathrm{Dom}\ Y$ and values

$$
(X + Y)(x) = \begin{cases} Y(x) & \text{if } x \in \mathrm{Dom}\ Y \\ X(x) & \text{otherwise} \end{cases}
$$

The notion of modification is extended point-wise to tuples, as are operations such as Dom, $\cap$, and $\cup$. A finite map X *extends* another finite map X', written $X \sqsupseteq X'$, if $\mathrm{Dom}\ X \supseteq \mathrm{Dom}\ X'$ and $X(x) = X'(x)$ for all $x \in \mathrm{Dom}\ X'$.

Given a particular kind of environment, such as a module environment ME, we shall often be implicit about its injection $(\{\}, \{\}, ME, \{\})$ into environments of type Env. Moreover, given an identifier, such as *tid*, its class specifies exactly that, given some type τ, $\{tid \mapsto \tau\}$ denotes a type environment of type TE, which again, by the above convention, can be injected implicitly into an environment of type Env.

As an example, if `t` is a type identifier, `a` and `b` are value identifiers, and `A` is a module identifier, we can write $\{\mathtt{t} \mapsto t\} + \{\mathtt{A} \mapsto \{\mathtt{a} \mapsto t\}\}$ for specifying the environment $E = (\{\mathtt{t} \mapsto t\}, \{\}, \{\mathtt{A} \mapsto E'\}, \{\})$, where $E' = (\{\}, \{\mathtt{a} \mapsto t\}, \{\}, \{\})$ and where $E \sqsupseteq \{\mathtt{t} \mapsto t\}$. Moreover, looking up the long identifier `A.a` in E, written $E(\mathtt{A.a})$, yields t.

$$
\begin{array}{rcl}
E_{Prg} & = & (\ \{\},\ \{\texttt{main} \mapsto (i32 \to i32)\},\ ME_{Prg},\ \{\texttt{MT} \mapsto \Sigma_{MT}\}\ ) \quad \textbf{where} \\
ME_{Prg} & = & \{\texttt{Main} \mapsto \texttt{M}_f,\ \texttt{H} \mapsto \forall\emptyset.(\ (\{\},\{\},\{\texttt{F} \mapsto \Sigma_F\},\{\}),\ \exists\emptyset.\texttt{M}_f\ )\} \\
\Sigma_{MT} & = & \exists\emptyset.(\{\},\ \{\},\ \{\texttt{F} \mapsto \Sigma_F\},\ \{\}) \\
\Sigma_F & = & \forall\emptyset.(\ (\{\},\{\texttt{b} \mapsto i32\},\{\},\{\}),\ \exists\emptyset.\texttt{M}_f) \\
\texttt{M}_f & = & (\ \{\},\ \{\texttt{f} \mapsto (i32 \to i32)\},\ \{\},\ \{\}\ )
\end{array}
$$

Fig. 5. The semantic objects at outermost (program) level for the code example in Figure 3.

Figure 5 shows the outer-level environment for the code example in Figure 3, which are obtained by applying the elaboration (typing) rules, which are presented in the remainder of this section.

4.1 Enrichment

Enrichment specifies that an environment, in an inductive sense, has the same or more elements than another environment. Enrichment is mutually and inductively defined on module environments, modules, and environments. The concept is central to the understanding of parameterised module application and matching. An environment $E' = (TE', VE', ME', G')$ *enriches* another environment $E = (TE, VE, ME, \{\})$, written $E' \succ E$, if $VE' \sqsupseteq VE$, $TE' \sqsupseteq TE$, and $ME' \succ ME$. A module environment ME' *enriches* another module environment ME, written $ME' \succ ME$, if Dom $ME' \supseteq$ Dom ME and $ME'(mid) \succ ME(mid)$ for each $mid \in$ Dom ME. A module $M' = E'$ *enriches* another module $M = E$, written $M' \succ M$, if $E' \succ E$. Moreover, a module $M' = \forall T'.(E', \Sigma')$ *enriches* another module $M = \forall T.(E, \Sigma)$, written $M' \succ M$, if $T' = T$ and $E \succ E'$ and $\Sigma' \succ \Sigma$.

Notice that enrichment for parameterised modules is contravariant in parameter environments. Notice also the special treatment of module type environments. Because a module type cannot specify bindings of module types (such a possibility creates a number of problems that are complementary to static interpretation), we can safely require that when $E' \succ E$, the module type environment in E is empty. A simpler approach was chosen than a more rigid semantic object structure that would capture precisely the non-supportedness of module type bindings inside modules.

4.2 Instantiation

Another central concept for understanding parameterised module application and module matching is the notion of instantiation, which is used for capturing that a particular module is a concrete instance of a generic module type. Instantiation is based on the notion of a *substitution* (S), which maps type variables to types. The result of applying a substitution S to an object X, written $S(X)$, is first to extend the substitution to be the identity outside its domain and then simultaneously substitute each type variable t in X with $S(t)$, after appropriately renaming bound type variables in X. The *support* of a substitution S, written Supp S, is the set of elements t for which $S(t) \neq t$.

Formally, then, an object A *instantiates* another object B, written $A \leq B$, if the judgement can be derived according to the following rules:

$$
\frac{\boxed{E \leq E'} \qquad \begin{array}{c}\text{Dom}(ME) = \text{Dom}(ME') \\ \forall mid \in \text{Dom}(ME), ME(mid) \leq ME'(mid)\end{array}}{(TE, VE, ME, G) \leq (TE, VE, ME', G)}
\qquad
\frac{\boxed{F \leq F'} \qquad \begin{array}{c}(E, \Sigma) \leq \forall T'.(E', \Sigma') \\ T \cap \text{tvs}(\forall T'.(E', \Sigma')) = \emptyset\end{array}}{\forall T.(E, \Sigma) \leq \forall T'.(E', \Sigma')}
$$

Type Expressions. $\boxed{E \vdash ty : \tau}$

$$\frac{E(longtid) = \tau}{E \vdash longtid : \tau} \quad (9) \qquad \frac{E \vdash ty_i : \tau_i \quad i = [1,2]}{E \vdash ty_1 \to ty_2 : \tau_1 \to \tau_2} \quad (10)$$

Core Language Expressions. $\boxed{E \vdash exp : \tau}$

$$\frac{E(longvid) = \tau}{E \vdash longvid : \tau} \quad (11) \qquad \frac{E \vdash exp : \tau \qquad E \vdash ty : \tau}{E \vdash exp : ty : \tau} \quad (12)$$

$$\frac{E + \{vid \mapsto \tau\} \vdash exp : \tau'}{E \vdash \lambda vid \to exp : \tau \to \tau'} \quad (13) \qquad \frac{E \vdash exp_1 : \tau \to \tau' \qquad E \vdash exp_2 : \tau}{E \vdash exp_1\ exp_2 : \tau'} \quad (14)$$

Fig. 6. Elaboration rules for the core language. The rules are straightforward but illustrate the interaction between the module language and the core language through the concept of long identifiers.

$$\boxed{E \leq \Sigma} \quad \frac{\mathrm{Supp}(S) \subseteq T \qquad E \leq S(E')}{E \leq \exists T.E'} \qquad \boxed{\Sigma \leq \Sigma'} \quad \frac{T \cap \mathrm{tvs}(\exists T'.E') = \emptyset \qquad E \leq \exists T'.E'}{\exists T.E \leq \exists T'.E'}$$

$$\boxed{(E,\Sigma) \leq \forall T'.(E',\Sigma')} \quad \frac{\mathrm{Supp}(S) \subseteq T \qquad S(E') \leq E \quad \Sigma \leq S(\Sigma')}{(E,\Sigma) \leq \forall T'.(E',\Sigma')}$$

Notice the contravariance in the rule for instantiating a parameterised module type and that instantiation is applied inductively in the rules, which differs from how instantiation is defined in the case for first-order modules [Milner et al. 1997]. For instance, with this revised instantiation relation, given that $\{\mathtt{t} \mapsto \tau\}$ represents an environment mapping a type constructor to a type τ, a concrete parameterised module type $\forall\emptyset.(\{\}, \{\mathtt{t} \mapsto i32\})$ instantiates the more abstract parameterised module type $\forall\emptyset.(\{\}, \exists\{t\}.\{\mathtt{t} \mapsto t\})$.

The notion of matching combines the notions of enrichment and instantiation. An environment E *matches* a module type Σ if there exists another environment E' such that $E \succ E'$ and $E' \leq \Sigma$. Matching is central in the elaboration rule for applications of parameterised modules and to the notions of encapsulation and abstraction, in general.

4.3 Elaboration

Elaboration of the module language is based on elaboration rules for the core language, which are straightforward and presented in Figure 6. The rules are split into rules for type expressions and rules for value expressions and allow for inferences of the forms $E \vdash ty : \tau$ and $E \vdash exp : \tau$, which are read "under assumptions E, the type expression ty (or value expression exp) is elaborated to have type τ". Notice how the core language interacts with the module language through the use of long identifiers in Rule 9 and Rule 11.

Elaboration of module types and specifications is defined as a mutual inductive relation allowing inferences among sentences of the forms $E \vdash mty : \Sigma$ and $E \vdash spec : \exists T.E'$. The rules are presented in Figure 7. There is a subtle difference between module type expressions (mty) and specifications ($spec$). Whereas module type expressions may elaborate to parameterised module types, specifications only elaborate to non-parameterised module types, which may, however, contain parameterised modules inside. Thus, in Rule 22, we require that the included module is a non-parameterised module type.

The most interesting of the rules are Rule 16, the rule for **with**-types, and Rule 18, the rule for expressing parameterised modules as dependent products [Harper and Lillibridge 1994].

Module Types. $\boxed{E \vdash mty : \Sigma}$

$$\frac{E(mtid) = \Sigma}{E \vdash mtid : \Sigma} \quad (15) \qquad \frac{E \vdash ty : \tau \quad E'(longtid) = t \quad t \in T \quad E \vdash mty : \exists T.E' \quad \Sigma = \exists(T \setminus \{t\}).(E'[\tau/t])}{E \vdash mty \textbf{ with } longtid = ty : \Sigma} \quad (16)$$

$$\frac{E \vdash spec : \Sigma}{E \vdash \{\ spec\ \} : \Sigma} \quad (17) \qquad \frac{E \vdash mty_1 : \exists T.E' \quad T \cap (\text{tvs}(E) \cup T') = \emptyset \quad E + \{mid \mapsto E'\} \vdash mty_2 : \exists T'.M}{E \vdash mid : mty_1 \to mty_2 : \forall T.(E', \exists T'.M)} \quad (18)$$

Module Specifications. $\boxed{E \vdash spec : \exists T.E'}$

$$\frac{}{E \vdash \textbf{type } tid : \exists\{t\}.\{tid \mapsto t\}} \quad (19) \qquad \frac{E \vdash ty : \tau}{E \vdash \textbf{val } vid : ty : \exists\emptyset.\{vid \mapsto \tau\}} \quad (20)$$

$$\frac{E \vdash mty : \exists T.M}{E \vdash \textbf{module } mid : mty : \exists T.\{mid \mapsto M\}} \quad (21) \qquad \frac{E \vdash mty : \exists T.E'}{E \vdash \textbf{include } mty : \exists T.E'} \quad (22)$$

$$\frac{E \vdash spec_1 : \exists T_1.E_1 \quad E + E_1 \vdash spec_2 : \exists T_2.E_2 \quad T_1 \cap (\text{tvs}(E) \cup T_2) = \emptyset \quad \text{Dom } E_1 \cap \text{Dom } E_2 = \emptyset}{E \vdash spec_1\ spec_2 : \exists(T_1 \cup T_2).(E_1 + E_2)} \quad (23) \qquad \frac{}{E \vdash \epsilon : \exists\emptyset.\{\}} \quad (24)$$

Fig. 7. Elaboration rules for module types and module specifications. This sub-language does not directly depend on the rules for module expressions and module declaration.

The rule for **with**-types (also called **where**-types in Standard ML) allows a programmer to refine a module type to a more concrete module type. The rule for expressing parameterised module types allow a programmer to express dependencies between the parameter module and the result module by referring to the module identifier associated with the parameter. As an example, consider the module type expression

$$\text{X:\{ } \textbf{type } \text{t } \textbf{val } \text{a} : \text{t } \} \to \{ \textbf{ val } \text{b} : \text{X.t } \}$$

This module type expression elaborates, using, among others, Rule (16), to the module type

$$\forall\{t\}.(\{\mathtt{t} \mapsto t, \mathtt{a} \mapsto t\}, \exists\emptyset.\{\mathtt{b} \mapsto t\})$$

An essential aspect of the semantic technique is that of requiring, for instance, that the sets T_1 and T_2 in Rule (23) are disjoint. This property can always be satisfied by α-renaming, which is applied often (and in the Coq implementation, explicitly) when proving properties.

4.4 Elaboration Rules for Module Expressions and Module Declarations

The elaboration rules for module language expressions and declarations are given in Figure 8 and allow inferences among sentences of the forms $E \vdash mdec : \Sigma$ and $E \vdash mexp : \exists T.E$. The rules make use of the previously introduced rules for module type expressions and core language declarations and types. Similarly to the elaboration difference between module type expressions and specifications, module expressions elaborate to general module types $\exists T.M$, whereas module declarations elaborate to non-parameterised module types $\exists T.E$.

The by far most complicated rule is Rule (29), the rule for application of a parameterised module. The rule looks up a parameterised module type $\forall T_0.(E_0, \Sigma_0)$ for the long module

Module Expressions. $\boxed{E \vdash mexp : \Sigma}$

$$\frac{E \vdash mdec : \Sigma}{E \vdash \{\ mdec\ \} : \Sigma}\ (25) \qquad \frac{E \vdash mexp : \exists T.E' \quad E'(mid) = E''}{E \vdash mexp\,.\,mid : \exists T.E''}\ (26) \qquad \frac{E(mid) = E'}{E \vdash mid : \exists \emptyset.E'}\ (27)$$

$$\frac{E \vdash mty : \exists T.E' \quad E + \{mid \mapsto E'\} \vdash mexp : \Sigma \quad T \cap \mathrm{tvs}(E) = \emptyset \quad F = \forall T.(E', \Sigma)}{E \vdash \lambda mid : mty\ \rightarrow\ mexp : \exists \emptyset.F}\ (28)$$

$$\frac{E \vdash mexp : \exists T.E' \quad T \cap T' = \emptyset \quad (E'', \exists T'.E''') \leq E(longmid) \quad E' \succ E'' \quad (T \cup T') \cap \mathrm{tvs}(E) = \emptyset}{E \vdash longmid\ (\ mexp\ ) : \exists (T \cup T').E'''}\ (29)$$

Module Declarations. $\boxed{E \vdash mdec : \exists T.E'}$

$$\frac{mdec = dec \quad E \vdash dec : E'}{E \vdash mdec : \exists \emptyset.E'}\ (30) \qquad \frac{E \vdash ty : \tau}{E \vdash \mathbf{type}\ tid = ty : \exists \emptyset.\{tid \mapsto \tau\}}\ (31)$$

$$\frac{E \vdash mexp : \exists T.M}{E \vdash \mathbf{module}\ mid = mexp : \exists T.\{mid \mapsto M\}}\ (32) \qquad \frac{E \vdash mexp : \Sigma}{E \vdash \mathbf{open}\ mexp : \Sigma}\ (33)$$

$$\frac{E \vdash mty : \Sigma}{E \vdash \mathbf{module\ type}\ mtid = mty : \exists \emptyset.\{mtid \mapsto \Sigma\}}\ (34) \qquad \frac{}{E \vdash \epsilon : \exists \emptyset.\{\}}\ (35)$$

$$\frac{T_1 \cap (\mathrm{tvs}(E) \cup T_2) = \emptyset \quad E \vdash mdec_1 : \exists T_1.E_1 \quad E + E_1 \vdash mdec_2 : \exists T_2.E_2}{E \vdash mdec_1\ mdec_2 : \exists (T_1 \cup T_2).(E_1 + E_2)}\ (36)$$

Fig. 8. Elaboration rules for module language expressions and declarations.

identifier in the environment and seeks to match the parameter module type $\exists T_0.E_0$ against a cut-down version (according to the enrichment relation) of the module type resulting from elaborating the argument module expression. The result of elaborating the application is the result module type, perhaps with additional abstract type variables stemming from elaborating the argument module expression. The need for also quantify over the type set T in the result module type comes from the desire to prove a property that if $E \vdash mexp : \exists T.E'$ then $\mathrm{tvs}(E') \subseteq \mathrm{tvs}(E) \cup T$. The remainder of the rules cover the other constructs of the language, which include Rule 26, for module projection, and Rule 28, for parameterised modules. In the rule for parameterised modules, the side condition $T \cap \mathrm{tvs}(E) = \emptyset$ expresses that T should be chosen sufficiently fresh. Notice in particular that, due to α-renaming, the judgement $E \vdash mty : \exists T.E'$ does not by itself entail this property. Another important aspect to notice is that both Rule 26 and Rule 36 are implicit about removal of bound type variables that do not occur under the binder, as objects are considered identical up-to removal of superfluous type variables. In Rule 36, for instance, due to shadowing of identifiers, a type variable in the set $T_1 \cup T_2$ may not necessarily occur in the environment $E_1 + E_2$.

To specify how a module expression *mexp* is classified by a module type expression *mty*, a specialised rule can be given:

$$\frac{E \vdash mexp : \exists T.E' \quad E \vdash mty : \Sigma \quad T \cap \mathrm{tvs}(E) = \emptyset \quad E' \succ E'' \quad E'' \leq \Sigma}{E \vdash mexp : mty : \Sigma}\ (37)$$

We see here that *mexp* is allowed to declare more identifiers than are specified by *mty* (expressed using $\succ$) and that declared identifiers in *mexp* may be less abstract than specified by *mty* (expressed using $\leq$). Formally, this rule is subsumed by the derived form in Equation 5, which turns matching into an application of a constrained identity functor. In the context of constructing libraries and modular applications for Futhark, we have not suffered from the lack of so-called *transparent module type matching*, for which the resulting module type Σ in Rule 37 needs to be changed to $\exists T.E''$. If needed, there is no technical reason for restricting static interpretation to support only opaque matching.

4.5 Demonstrating the Elaboration Rules on the Code Example of Figure 3

This section briefly discusses the manner in which the environments shown in Figure 5 were derived by the application of the elaboration rules. The module type `MT` in Figure 3 is declared to export a parameterised module `F`. The type of `F` is declared in the code to be `X:{val b:i32} → {val f:i32→i32}`, and is derived by Rule 18 to be $\Sigma_F = \forall\emptyset.(\ (\{\},\{\texttt{b} \mapsto \mathit{i32}\},\{\},\{\}),\ \exists\emptyset.\texttt{M}_f\ )$, where $\texttt{M}_f = (\{\},\{\texttt{f} \mapsto (\mathit{i32} \to \mathit{i32})\},\{\},\{\})$. Elaboration Rule 21 then extends the *ME* environment with the binding $\texttt{F} \mapsto \Sigma_F$ and Rule 34 extends the G environment with the binding $\texttt{MT} \mapsto \Sigma_{MT}$, where $\Sigma_{MT} = \exists\emptyset.(\{\},\{\},\{\texttt{F} \mapsto \Sigma_F\},\{\})$.

We discuss next the elaboration rules for `module H = λM:MT → M.F {let b=8}`. Rule 28 (i) deduces $\texttt{MT}:\Sigma_{MT}$ (by a lookup in G), (ii) adds the binding $\texttt{M} \mapsto \Sigma_{MT}$ to the *ME* environment, and (iii) computes the type of the application `M.F {let b=8}`. The latter step uses Rule 29 to derive the type $\exists\emptyset.(\{\},\{\texttt{f} \mapsto (\mathit{i32} \to \mathit{i32})\},\{\},\{\})$. It follows that Rule 28 computes the parameterised module's type to be $\Sigma_H = \forall\emptyset.((\{\},\{\},\{\texttt{F} \to \Sigma_F\},\{\}),\ \exists\emptyset.\texttt{M}_f)$, and Rule 32 adds the new binding $\texttt{H} \mapsto \Sigma_H$ to the *ME* environment.

We conclude by discussing the rules for `module Main`, which is defined as the application of the parameterised module `H` to the module expression `{module F = λX:{val b:i32} → {let f(x:i32)=X.b+x}}`. The type of `F` is derived by Rule 28 to be Σ_F in a manner similar to the derivation discussed before. Rule 29 then derives the type $\exists\emptyset.\texttt{M}_f$ for the application of the parameterised module `H` to the specification `{module F=...}`, and, finally, Rule 32 adds the new binding $\texttt{Main} \mapsto \texttt{M}_f$ to the *ME* environment.

5 STATIC INTERPRETATION

Static interpretation is the process of eliminating modules at compile time by translating modules into a sequence of target language definitions. The interpretation ensures that the target language declarations refer to previous declarations as specified by the source program. In the process, abstract types are eliminated, which results in a specialised program.

5.1 Target Language

We assume a denumerably infinite set LSet of *labels*, ranged over by l. Target expressions are basically identical to core level expressions with the modification that value identifiers are replaced with labels. For the simple core language that we are considering, *target expressions* (*ex*) and *target code* (c) take the form:

$$\begin{array}{lcl} ex & ::= & l \mid \lambda l \to ex \mid ex_1\ ex_2 \\ c & ::= & \textbf{val}\ l = ex \mid c_1\ ;\ c_2 \mid \epsilon \end{array}$$

The type system for the target language is simple (for the purpose of this paper) and allows inferences among sentences of the forms $\Gamma \vdash ex : \tau$ and $\Gamma \vdash c : \Gamma'$, which are read: "In the context Γ, the expression *ex* has type τ" and "in the context Γ, the target code c declares the context Γ'". Contexts Γ map labels to types. The type system for the target

$$\frac{\Gamma(l) = \tau}{\Gamma \vdash l : \tau} \ (38) \qquad \frac{\Gamma + \{l \mapsto \tau\} \vdash ex : \tau'}{\Gamma \vdash \lambda l \to ex : \tau \to \tau'} \ (39) \qquad \frac{\Gamma \vdash ex_1 : \tau' \to \tau \qquad \Gamma \vdash ex_2 : \tau'}{\Gamma \vdash ex_1 \ ex_2 : \tau} \ (40)$$

$$\frac{\Gamma \vdash c_1 : \Gamma_1 \qquad \Gamma + \Gamma_1 \vdash c_2 : \Gamma_2}{\Gamma \vdash c_1 \ ; \ c_2 : \Gamma_1 + \Gamma_2} \ (41) \qquad \frac{\Gamma \vdash ex : \tau}{\Gamma \vdash \mathbf{val}\ l = ex : \{l \mapsto \tau\}} \ (42) \qquad \frac{}{\Gamma \vdash \epsilon : \{\}} \ (43)$$

Fig. 9. Type rules for the target language. For the purpose of the presentation, the target language is simple (and standard) and mimics closely the source language with the difference that long identifiers are replaced with labels for referring to previously defined value declarations.

language is presented in Figure 9. Rule 41 is the rule for composing target code. The rules are straightforward (and standard) and we shall not describe them in details here.

5.2 Interpretation Objects

In the following, we shall use the term *name* to refer to either a type variable t or a label l. We write NSet to refer to the disjoint union of TSet and LSet. Moreover, we use N to range over finite subsets of NSet.

An *interpretation value environment* ($\mathcal{VE}$) maps value identifiers to a label and an associated type. An *interpretation environment* ($\mathcal{E}$) is a quadruple $(TE, \mathcal{VE}, \mathcal{ME}, G)$ of a type environment, an interpretation value environment, an interpretation module environment, and a module type environment. An *interpretation module environment* ($\mathcal{ME}$) maps module identifiers to module interpretations. A *module interpretation* ($\mathcal{M}$) is either an interpretation environment $\mathcal{E}$ or a functor closure Φ. A *functor closure* (Φ) is a triple $(\mathcal{E}, F, \lambda mid \to mexp)$ of an interpretation environment, a parameterised module type, and a representation of a parameterised module expression. Finally, an *interpretation target object* ($\exists N.(\mathcal{E}, c)$) is a triple of a name set, an interpretation environment, and a target code object.

5.3 Interpretation Erasure

For establishing a link between interpretation objects and elaboration objects, we introduce the concept of interpretation erasure. Given an interpretation object O, we define the *interpretation erasure* of O, written $\overline{O}$, as follows:

$$\begin{array}{rcl} \overline{(TE, \mathcal{VE}, \mathcal{ME}, G)} & = & (TE, \overline{\mathcal{VE}}, \overline{\mathcal{ME}}, G) \\ \overline{(\mathcal{E}, F, \lambda mid \to mexp)} & = & F \\ \overline{\{vid_i \mapsto l_i : \tau_i\}^n} & = & \{vid_i \mapsto \tau_i\}^n \end{array} \qquad \begin{array}{rcl} \overline{\{mid_i \mapsto \mathcal{M}_i\}^n} & = & \{mid_i \mapsto \overline{\mathcal{M}_i}\}^n \\ \overline{\exists N.(\mathcal{E}, c)} & = & \exists(\mathrm{TSet} \cap N).\overline{\mathcal{E}} \end{array}$$

As we shall see shortly, the concept of interpretation erasure makes it straightforward to specify various properties of the relation between static interpretation and elaboration.

5.4 Core Language Compilation

Core language expressions and declarations are compiled into target language expressions and declarations, respectively. The rules specifying the compilation are given in Figure 10 and allow inferences among sentences of the forms (1) $\mathcal{E} \vdash exp \Rightarrow ex, \tau$ and (2) $\mathcal{E} \vdash dec \Rightarrow \exists N.(\mathcal{E}', c)$. There are two interesting aspects to notice about the rules. First, accesses to long identifiers (i.e., in Rule 44) are compiled into label references. Second, core language value declarations are compiled into a label binding (i.e., Rule 48) for which the label is existentially bound. This existential binding allows for the static interpretation process to control the linking process. The remainder of the rules are straightforward.

Compiling Expressions. $\boxed{\mathcal{E} \vdash exp \Rightarrow ex, \tau}$

$$\frac{\mathcal{E}(longvid) = (l, \tau)}{\mathcal{E} \vdash longvid \Rightarrow l, \tau} \quad (44) \qquad \frac{\mathcal{E} + \{vid \mapsto (l, \tau)\} \vdash exp \Rightarrow ex, \tau'}{\mathcal{E} \vdash \lambda vid \to exp \Rightarrow \lambda l \to ex, \tau \to \tau'} \quad (45)$$

$$\frac{\mathcal{E} \vdash exp_1 \Rightarrow ex_1, \tau \to \tau' \qquad \mathcal{E} \vdash exp_2 \Rightarrow ex_2, \tau}{\mathcal{E} \vdash exp_1\, exp_2 \Rightarrow ex_1\ ex_2, \tau'} \quad (46) \qquad \frac{\mathcal{E} \vdash exp \Rightarrow ex, \tau \qquad \overline{\mathcal{E}} \vdash ty : \tau}{\mathcal{E} \vdash exp : ty \Rightarrow ex, \tau} \quad (47)$$

Compiling Declarations. $\boxed{\mathcal{E} \vdash dec \Rightarrow \exists N.(\mathcal{E}', c)}$

$$\frac{\mathcal{E} \vdash exp \Rightarrow ex, \tau \qquad l \notin \text{names}(\mathcal{E})}{\mathcal{E} \vdash \textbf{val}\ vid = exp \Rightarrow \exists\{l\}.\{vid \mapsto (l, \tau), \textbf{val}\ l = ex\}} \quad (48)$$

Fig. 10. Core language compilation.

The rules track type information and it is straightforward to establish the following property of the compilation:

Proposition 5.1. *If* $\mathcal{E} \vdash dec \Rightarrow \exists N.(\mathcal{E}', c)$ *then* $\overline{\mathcal{E}} \vdash dec \Rightarrow \overline{\exists N.(\mathcal{E}', c)}$.

5.5 Environment Filtering

Corresponding to the notion of enrichment for elaboration, we introduce a notion of filtering for the purpose of static interpretation, which filters interpretation environments to contain components as specified by an elaboration environment. Filtering is essential to the interpretation rule for applications of parameterised modules and is defined mutual inductively based on the structure of elaboration environments and elaboration module environments.

More formally, the *filtering* of an interpretation environment $\mathcal{E}$ to an elaboration environment E results in another interpretation environment $\mathcal{E}'$ with only elements from $\mathcal{E}$ that are also present in E. The filtering relation is defined by a number of inference rules that allow inferences among sentences of the forms (1) $\vdash \mathcal{E} :: E \Rightarrow \mathcal{E}'$, (2) $\vdash \mathcal{VE} :: VE \Rightarrow \mathcal{VE}'$, (3) $\vdash \mathcal{ME} :: ME \Rightarrow \mathcal{ME}'$, and (4) $\vdash \mathcal{M} :: M \Rightarrow \mathcal{M}'$. The inference rules for filtering are presented in Figure 11. It is a straightforward exercise to demonstrate that if $\vdash \mathcal{E} :: E \Rightarrow \mathcal{E}'$ then it holds that $\overline{\mathcal{E}} \succ E$.

5.6 Static Interpretation Rules

Static interpretation of the module language is defined by a number of mutually inductive inference rules allowing inferences among sentences of the forms (1) $\mathcal{E} \vdash mexp \Rightarrow \Psi$ and (2) $\mathcal{E} \vdash mdec \Rightarrow \exists N.(\mathcal{E}', c)$, which state that in an interpretation environment $\mathcal{E}$, static interpretation of a module expression *mexp* results in an interpretation target object Ψ, and static interpretation of a module declaration *mdec* results in an interpretation target object $\exists N.(\mathcal{E}', c)$. The rules for static interpretation are presented in Figure 12. In general, the structure of the rules resembles closely the structure of the elaboration rules for modules with the addition that static interpretation takes care of constructing and composing target code. In doing so, the composition of target code is controled using name binding exactly like type variable binding is used in the elaboration rules for controling the composition of elaboration environments. The only two rules that compose target code are Rule 58, the rule for application of a parameterised module, and Rule 65, the rule for sequential

Environments. $\boxed{\vdash \mathcal{E} :: E \Rightarrow \mathcal{E}'}$

$$\frac{\vdash \mathcal{V}E :: VE \Rightarrow \mathcal{V}E' \qquad \vdash \mathcal{M}E :: ME \Rightarrow \mathcal{M}E' \qquad TE \succ TE'}{\vdash (TE, \mathcal{V}E, \mathcal{M}E, \{\}) :: (TE', VE, ME, \{\}) \Rightarrow (TE', \mathcal{V}E', \mathcal{M}E', \{\})} \quad (49)$$

Value Environments. $\boxed{\vdash \mathcal{V}E :: VE \Rightarrow \mathcal{V}E'}$

$$\frac{m \geq n}{\vdash \{vid_i \mapsto l_i : \tau_i\}^m :: \{vid_i \mapsto \tau_i\}^n \Rightarrow \{vid_i \mapsto l_i : \tau_i\}^n} \quad (50)$$

Module Environments. $\boxed{\vdash \mathcal{M}E :: ME \Rightarrow \mathcal{M}E'}$

$$\frac{m \geq n \qquad \vdash \mathcal{M}_i :: M_i \Rightarrow \mathcal{M}'_i \quad i = 1..n}{\vdash \{mid_i \mapsto \mathcal{M}_i\}^m :: \{mid_i \mapsto M_i\}^n \Rightarrow \{mid_i \mapsto \mathcal{M}'_i\}^n} \quad (51)$$

Module Interpretations. $\boxed{\vdash \mathcal{M} :: M \Rightarrow \mathcal{M}'}$

$$\frac{\mathcal{M} = \mathcal{E} \qquad \vdash \mathcal{E} :: E \Rightarrow \mathcal{E}'}{\vdash \mathcal{M} :: E \Rightarrow \mathcal{E}'} \quad (52) \qquad \frac{\Phi = (\mathcal{E}, F', \lambda mid \rightarrow mexp) \qquad F' \succ F}{\vdash \Phi :: F \Rightarrow (\mathcal{E}, F, \lambda mid \rightarrow mexp)} \quad (53)$$

Fig. 11. Filtering relation specifying how an interpretation environment can be constrained by an elaboration environment to form a restricted interpretation environment.

composition of two module declarations. In Rule 58, environment filtering is used for filtering the argument module to hold only those components specified by the module type object. Notice also that the body of the parameterised module is extracted from the functor closure bound to the module identifier in the environment. Rule 57, on the other hand, takes care of constructing a functor closure object, but it also takes special care that the parameterised module elaborates under appropriate assumptions. The remaining rules are straightforward and follow closely the corresponding elaboration rules.

6 PROPERTIES

Before we can establish a static interpretation type soundness property, we first define a type consistency relation that relates interpretation environments, target language contexts, and substitutions providing concrete types for type variables that are considered abstract by elaboration. Due to the presence of higher-order parameterised modules, we shall make use of a logical relation argument [Tait 1967], which links the relation itself to the static interpretation. The relation is given by a number of inference rules, which are listed in Figure 13. The rules allow inferences among sentences of the forms (1) $E \models_S \mathcal{E} \triangleright \Gamma$, (2) $ME \models_S \mathcal{M}E \triangleright \Gamma$, (3) $M \models_S \mathcal{M} \triangleright \Gamma$, and (4) $VE \models_S \mathcal{V}E \triangleright \Gamma$. An essential property of the rules is that they are decreasing, structurally, in their left argument and therefore define a well-formed inductive relation. The two most interesting rules are Rule 68, which relates variables in elaboration and static interpretation environments with labels in target environments, and Rule 70, which relates parameterised modules in elaboration and static interpretation environments through static interpretation from appropriately related environments to resulting appropriately related results.

Based on the established consistency relation, we can now state a general property establishing (1) that static interpretation is possible and terminates for all elaborating programs and (2) that generated target programs are appropriately typed.

Module Expressions. $\boxed{\mathcal{E} \vdash mexp \Rightarrow \Psi}$

$$\frac{\mathcal{E} \vdash mdec \Rightarrow \exists N.(\mathcal{E}', c)}{\mathcal{E} \vdash \{\ mdec\ \} \Rightarrow \exists N.(\mathcal{E}', c)}\ (54) \qquad \frac{\mathcal{E}(mid) = \mathcal{E}'}{\mathcal{E} \vdash mid \Rightarrow \exists \emptyset.(\mathcal{E}', \epsilon)}\ (55)$$

$$\frac{\mathcal{E}'(mid) = \mathcal{E}'' \qquad \mathcal{E} \vdash mexp \Rightarrow \exists N.(\mathcal{E}', c)}{\mathcal{E} \vdash mexp\,.\,mid \Rightarrow \exists N.(\mathcal{E}'', c)}\ (56)$$

$$\frac{\begin{array}{c}\overline{\mathcal{E}} \vdash mty \Rightarrow \exists T.E \qquad T \cap \mathrm{names}(\mathcal{E}) = \emptyset \\ \overline{\mathcal{E}} + \{mid \mapsto E\} \vdash mexp : \Sigma \qquad F = \forall T.(E, \Sigma) \\ \Phi = (\mathcal{E}, F, \lambda mid \Rightarrow mexp)\end{array}}{\mathcal{E} \vdash \lambda mid : mty\ \rightarrow\ mexp \Rightarrow \exists \emptyset.\Phi}\ (57)$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash mexp \Rightarrow \exists N.(\mathcal{E}', c) \qquad (N \cup N') \cap \mathrm{names}(\mathcal{E}) = \emptyset \qquad N \cap N' = \emptyset \\ \mathcal{E}(longmid) = (\mathcal{E}_0, F, \lambda mid \Rightarrow mexp') \qquad (E', \exists T'.E'') \leq F \qquad T' \subseteq N' \\ \vdash \mathcal{E}' :: E' \Rightarrow \mathcal{E}'' \qquad \mathcal{E}_0 + \{mid \mapsto \mathcal{E}''\} \vdash mexp' \Rightarrow \exists N'.(\mathcal{E}''', c')\end{array}}{\mathcal{E} \vdash longmid(\ mexp\ ) \Rightarrow \exists (N \cup N').(\mathcal{E}''', c\ ;\ c')}\ (58)$$

Module Declarations. $\boxed{\mathcal{E} \vdash mdec \Rightarrow \exists N.(\mathcal{E}', c)}$

$$\frac{mdec = dec \qquad \mathcal{E} \vdash dec \Rightarrow \exists N.(\mathcal{E}', c)}{\mathcal{E} \vdash mdec \Rightarrow \exists N.(\mathcal{E}', c)}\ (59)$$

$$\frac{\overline{\mathcal{E}} \vdash ty : \tau}{\mathcal{E} \vdash \mathbf{type}\ tid = ty \Rightarrow \exists \emptyset.(\{tid \mapsto \tau\}, \epsilon)}\ (60)$$

$$\frac{\mathcal{E} \vdash mexp \Rightarrow \exists N.(\Phi, c)}{\mathcal{E} \vdash \mathbf{module}\ mid = mexp \Rightarrow \exists N.(\{mid \mapsto \Phi\}, c)}\ (61) \qquad \frac{}{\mathcal{E} \vdash \epsilon \Rightarrow \exists \emptyset.(\{\}, \epsilon)}\ (62)$$

$$\frac{\overline{\mathcal{E}} \vdash mty : \Sigma \qquad \mathcal{E}' = \{mtid \mapsto \Sigma\}}{\mathcal{E} \vdash \mathbf{module\ type}\ mtid = mty \Rightarrow \exists \emptyset.(\mathcal{E}', \epsilon)}\ (63) \qquad \frac{\mathcal{E} \vdash mexp \Rightarrow \Psi}{\mathcal{E} \vdash \mathbf{open}\ mexp \Rightarrow \Psi}\ (64)$$

$$\frac{\begin{array}{c}N_1 \cap (\mathrm{names}(\mathcal{E}) \cup N_2) = \emptyset \\ \mathcal{E} \vdash mdec_1 \Rightarrow \exists N_1.(\mathcal{E}_1, c_1) \qquad \mathcal{E} + \mathcal{E}_1 \vdash mdec_2 \Rightarrow \exists N_2.(\mathcal{E}_2, c_2)\end{array}}{\mathcal{E} \vdash mdec_1\ mdec_2 \Rightarrow \exists (N_1 \cup N_2).(\mathcal{E}_1 + \mathcal{E}_2, c_1\ ;\ c_2)}\ (65)$$

Fig. 12. Static interpretation rules for module expressions and module declarations.

Proposition 6.1. *(Static Interpretation Normalisation and Type Soundness) If* $E \vdash mdec : \exists T.E'$ *and* $E \models_S \mathcal{E} \triangleright \Gamma$ *then there exists* N, $\mathcal{E}'$, c, Γ', *and* S' *such that* $\mathcal{E} \vdash mdec \Rightarrow \exists N.(\mathcal{E}', c)$ *and* $N \supseteq T$ *and Dom* $S' = T$ *and* $E' \models_{S \circ S'} \mathcal{E}' \triangleright \Gamma'$ *and* $\Gamma \vdash c : \Gamma'$.

Proof. By mutual induction over *mdec* and *mexp*. □

The following corollary expresses a simplified version of the above proposition.

Corollary 6.2. *(Top-level Normalisation and Type Soundness) If* $\vdash mdec : \{x : i32\}$ *then there exists* N, l, c, *and* Γ *such that* $\vdash mdec \Rightarrow \exists (N \cup \{l\}).(\{x \mapsto (l, i32)\}, c)$ *and* $\vdash c : \Gamma + \{l \mapsto i32\}$.

$$\frac{\mathcal{E} = (TE, \mathcal{V}E, \mathcal{M}E, G) \quad VE \models_S \mathcal{V}E \triangleright \Gamma \quad ME \models_S \mathcal{M}E \triangleright \Gamma}{(TE, VE, ME, G) \models_S \mathcal{E} \triangleright \Gamma} \quad (66)$$

$$\frac{\mathrm{Dom}ME = \mathrm{Dom}\mathcal{M}E \quad \forall mid \in \mathrm{Dom}ME, ME(mid) \models_S \mathcal{M}E(mid) \triangleright \Gamma}{ME \models_S \mathcal{M}E \triangleright \Gamma} \quad (67)$$

$$\frac{\mathrm{Dom}\, VE = \mathrm{Dom}\mathcal{V}E \quad \forall x \in \mathrm{Dom}\, VE, \tau = VE(x) \wedge (l, \tau) = \mathcal{V}E(x) \wedge S(\tau) = \Gamma(l)}{VE \models_S \mathcal{V}E \triangleright \Gamma} \quad (68)$$

$$\frac{M = E \quad \mathcal{M} = \mathcal{E} \quad E \models_S \mathcal{E} \triangleright \Gamma}{M \models_S \mathcal{M} \triangleright \Gamma} \quad (69)$$

$$\frac{\begin{array}{l} M = \forall T.(E, \exists T'.E') \quad \mathcal{M} = (\mathcal{E}_0, \forall T.(E, \exists T'.E'), mid, mexp) \\ (\forall \mathcal{E}, S',\ \overline{\mathcal{E}} = E\ \wedge\ \mathrm{Dom}\ S' = T\ \wedge\ E \models_{S \circ S'} \mathcal{E} \triangleright \Gamma \implies \\ \quad \exists N', \mathcal{E}', c, S'',\ \mathcal{E}_0 + \{mid \mapsto \mathcal{E}\} \vdash mexp \Rightarrow \exists N'.(\mathcal{E}', c) \wedge \\ \quad \overline{\mathcal{E}'} = E'\ \wedge\ \mathrm{Dom}\ S'' = T'\ \wedge\ N' \supseteq T'\ \wedge \\ \quad E' \models_{S \circ S' \circ S''} \mathcal{E}' \triangleright \Gamma'\ \wedge\ \Gamma \vdash c : \Gamma') \end{array}}{M \models_S \mathcal{M} \triangleright \Gamma} \quad (70)$$

Fig. 13. Type consistency logical relation.

With local module declarations, *mdec* may contain both higher-order module declarations and complex applications of such modules. This corollary thus illustrates a non-trivial property, similar to showing for the simply-typed lambda calculus that if $\vdash e : i32$ then there exists an integer d such that $e \hookrightarrow^* d$. Notice also that, because the target language contains no local bindings, all declared labels in c escape to top-level and are described by the resulting context $\Gamma + \{l \mapsto i32\}$.

7 TYPE INFERENCE FOR THE MODULE LANGUAGE

We shall not here give a type inference algorithm for the module language but instead mention that type inference for the language becomes straightforward (i.e., syntax directed) if elaborated module types are what is called type explicit [Milner et al. 1997].

The notion of type explication is mutually inductively defined as follows. A module type $\Sigma = \exists T.E$ is *type explicit* if E is type explicit and for all $t \in T$, there exists a *longtid* such that $E(longtid) = t$. A module type $\Sigma = \exists T.F$ is *type explicit* if $T = \emptyset$ and F is type explicit. A parameterised module type $F = \forall T.(E, \Sigma)$ is *type explicit* if $\exists T.E$ is type explicit. An environment E is *type explicit* if all module types Σ in E are type explicit.

The importance of type explication becomes apparent when trying to match a concrete environment E' against a module type $\Sigma = \exists T.E$, which is what is required in the rule for application of parameterised modules. If Σ is type explicit, it becomes straightforward to decide whether there exists a substitution S such that $\mathrm{Dom}\ S = T$ and $S(E') \succ E$.

The following proposition states that our language for specifying module types generates type explicit module types:

Proposition 7.1. *(Type Explicit Module Types). If $E \vdash mty : \Sigma$ or $E \vdash spec : \Sigma$ then Σ is type explicit if E is type explicit.*

There are a number of good reasons for extending the language for expressing module types [Ramsey et al. 2005] to avoid unnecessary duplication of module type specification code. As long as such extensions come with guarantees that every expressible module type is type explicit, type inference for the module language will be straightforward.

8 POLYMORPHISM AND HIGHER-ORDER FUNCTIONS

In this section, we show how certain classes of polymorphic functions and polymorphic types at the source level can be treated as derived forms of parameterised modules and how certain instantiations can be treated as parameterised module applications.

8.1 Polymorphic Functions

Consider the following Futhark declaration of a function polymorphic in the types `t` and `s`:

```
let imap 't 's (f: (i32,t)→s) (a:[]t) : []s =
  map f (zip (iota (length a)) a)
```

This declaration can be treated as a special derived form of a declaration of a parameterised module:

```
module Imap (X: {type t type s val f: (i32,t)→s}) = {
  open X
  let imap (a:[]t) : []s = map f (zip (iota (length a)) a)
}
```

Now, for obtaining an instantiation of the module, it is possible to instantiate the "polymorphic function" `imap` to concrete arguments as follows:

```
let main () = imap (λ(i,x)→i+x) [1,2,3]
```

This code can then be translated into the following modularised code:

```
local module Imap_23 = Imap({type t=i32 type s=i32 let f (i,x) = i+x})
in { let main () = Imap_23.imap [1,2,3] }
```

This special short-hand for instantiating "polymorphic functions" is possible only in certain cases. For simplicity, it can be required that type parameter instantiations can be determined directly from the calling context and that the free variables of the passed arguments (e.g., the free variables in `λ(i,x)→i+x`) are all bound outside the containing module declaration. With a little more bookkeeping, it is possible to relax this requirement by implementing a version of lambda lifting [Johnsson 1985] that adds additional parameters to the declared function. Finally, with this approach, it is a requirement that the declared polymorphic functions do not themselves return functions; the scheme can support a number of higher-order functions but not first-class functions in general.

8.2 Polymorphic Types

Consider the following Futhark declaration of a "polymorphic type":

```
type pair 't 's = (t,[]s)
```

This type declaration can be translated into the following module declaration:

```
module Pair (X : {type t type s}) =
  open X
  type pair = (t,[]s)
}
```

For obtaining an instantiation of the module, it is possible to instantiate the "polymorphic type" `pair` to concrete arguments as follows:

```
type y = pair i32 bool
```

This code can be translated into the following module-level declaration:

```
local module Pair_33 = Pair({type t=i32 type s=bool})
in { type y = Pair_33.pair }
```

Although this technique has some limitations compared to supporting polymorphism at the core language level, the mechanism is simple and can, for certain application domains, relieve the language implementor from considering polymorphism at the core language level.

In practice, Futhark is relying on separate passes for eliminating core language polymorphism and higher-order functions [Hovgaard 2018]. These passes both occur after static interpretation of modules and do not interfere with static interpretation. The separation of the passes for core language monomorphication and core language elimination of higher-order functions from static interpretation allows for supporting a richer set of polymorphic higher-order functions.

9 FORMALISING THE DEVELOPMENT IN COQ

We have formalised, in the Coq proof assistant, essential parts of the definitions given in this paper along with the proof of static interpretation normalisation. In the course of the development, we have used additional axioms of functional extensionality and proof irrelevance for propositions. Also, for the development of nominal techniques, we assume a countably infinite set of variable names.

We have taken an extrinsic approach [Benton et al. 2012], as opposed to an intrinsic one, to the representation of the core language, the module language, and the target language, which keeps our implementation close to the approach presented in the paper. The extrinsic encoding has an advantage of being more suitable for code extraction to obtain a certified implementation. That is, we have implemented the abstract syntax as simple inductive data types and given separate inductive definitions for relations such as elaboration, typing, and so on. The semantic objects of Figure 4 have been implemented as mutually defined inductive types using Coq's `with` clause. The same approach is used for definitions of relations on environments. As described in Section 4, semantic objects are represented using finite maps and sets and indeed, the implementation makes use of Coq's standard library implementations of such objects. Specifically, we use the `FMapList` and `FSetList` implementations of the `FMap` and `FSet` interfaces, respectively. Both `FMapList` and `FSetList` make use of the `list` data type together with a property that the list is ordered according to a strict order on the underlying data structure. The strict order for the underlying list allows us to prove an extensionality property for environments and sets (assuming proof irrelevance). That is, for any two environments E_1 and E_2 we have $(\forall k, E_1(k) = E_2(k)) \rightarrow E_1 = E_2$. The equal sign $=$ refers to the Coq propositional equality, which means that we can use all the standard rewriting machinery instead of using setoid equality.

Unfortunately, we cannot use the definition of environments from the standard library to define semantic objects such as MEnv because of Coq's limitation that using type constructors for the environments from the standard library will violate Coq's strict positivity check for inductive definitions. To overcome this complication, we introduce an isomorphic pair-of-vectors representation of environments, where the first vector is an ordered vector of keys and the second vector is a vector of values. Separating keys and values in different vectors allows

us to define semantic objects in a way acceptable for Coq's strict-positivity checker. The idea of using an isomorphic representation is very similar to Wadler's notion of *views* [Wadler 1987]. We can see the two representations of environments as two views associated with the abstract "type" of environments, which corresponds to a module specifying operations on and properties of environments. Instead of defining all operations on the second representation of environments, we use Coq's coercion mechanism to insert coercion functions automatically, given by the isomorphism between the two representations. Our Coq development shows that it is sufficient to use a few properties of the isomorphism to transfer proofs of the properties from one representation to the other.

In order to prove theorems by induction over the structure of semantic objects, or relations containing mutual definitions, Coq's `Scheme` command is used to generate suitable induction principles. For some of the definitions, such as those for semantic objects and interpretation environments, the generated induction principles are not sufficiently strong, which is caused by the presence of nested inductive types; some constructors take environments as parameters, and the environments, being essentially lists, make the whole definition a nested inductive definition. For each of these cases, a suitable induction principle is defined manually, following essentially the same approach as in Section 3.8 of [Chlipala 2013].

For proof automation, we make use of the `crush` tactic from [Chlipala 2013] and some tactics from Pierce et al. [2016]. The structure of most of the proofs are kept explicit, though, using automation to resolve only the most tedious parts of the proofs.

The proof of static interpretation normalisation is carried out essentially using the same logical relation argument as presented in Figure 13. The logical relation is implemented as a fixpoint rather than as an inductive relation. The reason for this representation is essential. If the relation was defined as an inductive predicate, the definition would not pass the strict positivity constraint for inductive definitions in Coq. From the definition of our logical relation, it is straightforward to establish that the relation is well-formed because it is decreasing structurally in its left argument. For this reason, it can be expressed as a fixpoint definition, using also Coq's anonymous fix-construct, corresponding to the nested structure of the semantic objects. Unfortunately, we cannot keep our environment representation completely abstract, since we define the logical relation recursively on the structure of environments. Restrictions on fixpoint definitions in Coq require us to use a nested fixpoint on underlying structures in the definition of environments. Again, we use a pair-of-vectors view to define a corresponding nested fixpoint in the definition of the consistency relation of Figure 13.

We have kept the Coq development close to the representation in the paper. However, the filtering relation of Figure 11 does not define a filtering algorithm directly but serves as a specification for a filtering algorithm. In the proof of normalisation of static interpretation, we have to show the existence of a filtered environment. Due to the limitations of Coq's fixpoint constructs, we have defined a filtering algorithm as an inductively defined relation. We consider it future work to investigate the use of general recursion in Coq for filtering, which would be useful for applying code extraction to obtain a certified static interpretation implementation. We believe it to be a reasonable approach to separate the relational "declarative" definitions from definitions that compute for code extraction. One can then establish a correspondence between the relational and functional representations to show soundness of the implementation.

We are using a nominal approach [Gabbay and Pitts 2002; Pitts 2013] to define notions of freshness, fresh name generation, and α-equivalence relations. The approach gives us a uniform structuring principle to deal with variables in various data-structures involved in

the formalisation. Since there is no standard package in the Coq distribution implementing nominal techniques (the only available work is Brian Aydemir et al.'s Nominal Reasoning Techniques in Coq [Aydemir et al. 2007], but it is designed only for single variable bindings), we have developed our own implementation covering generalised binding structures such as bindings over sets of variables.

One example of the application of nominal techniques is the condition $l \notin names(\mathcal{E})$ (Rule 48, Figure 10). We define a nominal set of interpretation environments with labels as atoms. Then, the corresponding condition in Rule 48 can be expressed in terms of freshness with the requirement $l \# \mathcal{E}$ (l is fresh for $\mathcal{E}$). In the proofs, we use the fact that set of atoms is countably infinite to generate fresh (with respect to the appropriate context) labels.

The properties as they are described by propositions and theorems in the paper have not all been completely formalised with proper name abstractions. However, with the developed approach, we believe it is possible to complete the development. Our work-in-progress development contains further examples of the applications of nominal techniques. We use a freshness relation to define conditions involving disjointness of bound variables in such rules as Rule 23 (Figure 7). Notions of α-equivalence is defined in a general way as in Ranald Clouston's work on Generalised Name Abstraction for Nominal Sets [Clouston 2013]. We explicitly add α-conversion in the elaboration rules for functor application (Rule 29) and for the sequence of declarations (Rule 36). We provide examples illustrating when the addition of α-conversion is important for the elaboration of certain module expressions in the completely formal setting. The Coq development is available as an ACM DL artifact and from the Github repository https://github.com/diku-dk/futhark-icfp18.

10 FUTHARK LIBRARIES AND APPLICATIONS

We have already in Section 2 presented a Futhark library that makes use of higher-order modules. Steadily, a Basis Library for Futhark emerges. Currently, the library includes modules for operating on various kinds and sizes of numeric values, such as integral numbers, floating point numbers, complex numbers, and rational numbers. Other modules then exist that are parametric over the particular numeric instance. Such modules include a module for linear algebra routines. Other modules provide generic parallel routines for radix-based sorting, sparse matrices, and generation of Sobol sequences [Joe and Kuo 2003].

At application level, a number of programs have been written that make extensive use of the Futhark module language and for which it is essential that modules are eliminated completely at compile time. One example is an implementation of several variants of Conway's Game of Life, which differ in both rule sets and visualisation algorithms, but utilise the module system to obtain code reuse. The Game of Life is a rank-1 stencil on a two-dimensional grid, with a rule for how one cell changes its value from one (discrete) step to the next based on its current value and those of its neighbours. In Conway's formulation, each cell can be in one of two states (living or dead), but variations can have more states. We can describe the generic concept of a *rule set* as a module type:

```
module type rules = {
  type cell
  val step: cell → i32 → cell
  val value: cell → i32
  val weights: [][]i32
}
```

A rule set consists primarily of a cell type and a *step function*. To compute the value of cell c in step $i + 1$, the step function is called with the current value of c and the sum of all cells within the surrounding 3×3 *neighbourhood* (including c). This sum is computed by transforming each cell to an integer via `value`, then multiplying each cell with its corresponding weight in the `weights` array. An implementation of a full Game of Life can be described as follows:

```
module type game_of_life = {
  type cell
  val step: [][]cell → [][]cell
}
```

The `step` function runs one evolution of the Game of Life. We can define a parametric module that, given an implementation of the `rules` module type, gives us an implementation of the `game_of_life` signature:[2]

```
module gen_life(R: rules): game_of_life with cell = R.cell = {
  type cell = R.cell
  let all_neighbour_sums(world: [][]cell): [][]i32 = ...
  let step (world: [][]cell): [][]cell =
    let all_sums = all_neighbour_sums world
    in map2 (map2 R.step) world all_sums
}
```

This setup lets us run instances of the Game of Life, but we would also like to be able to visualise them on the screen. For visualisation, we define module types that describe both rule sets and a mechanism for getting a colour from a cell:

```
module type vis_rules = {
  include rules
  val colour: cell → argb.colour
  val init: bool → cell
  val uninit: cell → bool
}

module type vis_game_of_life = {
  include game_of_life
  val render: [][]cell → [][]argb.colour
}
```

The `render` function accepts an array of cells and produces an array of ARGB colour values encoded as integers (making use of the `argb` module from the Futhark Basis Library.) We also require that cells can be initialised from booleans, and transformed back into booleans, although these two functions need not be inverses of each other. This facility is useful for random initialisation; we shall see another use of it in Section 10.1.

We can define a parametric module that operates on modules of the above module types:

[2]The function `map2` maps over two arrays and invokes a function of two parameters, as `zipWith` of Haskell.

```
module gen_life_vis(R: vis_rules)
       : (vis_game_of_life with cell = R.cell) = {
  open (gen_life R)
  let render (world: [][]cell): [][]argb.colour =
    map (map R.colour) world
}
```

Using the above infrastructure, we can now define an implementation of the Game of Life:

```
module conway_rules: vis_rules with cell = bool = {
  type cell = bool
  let step (alive: cell) (neighbours: i32) =
    neighbours >= 2 && (neighbours == 3 || (alive && neighbours < 4))
  let value (b: cell) = if b then 1 else 0
  val weights = [[1,1,1], [1,0,1], [1,1,1]]
  let init (b: bool) = b
  let uninit (c: cell) = c
  let colour (b: cell) = if b then argb.black else argb.white
}
module conway = gen_life_vis conway_rules
```

We can equally well define a more complex four-state Game of Life variant:

```
module quad_rules: vis_rules = {
  type cell = i8
  let step (_c: cell) (neighbours: i32): i8 =
    let t = [0,0,0,2,2,3,3, 1,1,1,1,1,0,0,
             2,0,2,2,2,2,2, 0,2,1,2,3,3,3]
    in t[neighbours]
  let value (c: cell) = i32 c
  val weights = [[1,1,1], [1,1,1], [1,1,1]]
  let init (b: bool) = if b then 0 else 1
  let uninit (c: cell) = c != 1
  let colour (c: cell) =
    let colours = [argb.green, argb.black, argb.red, argb.blue]
    in colours[i32 c]
}
module quad = gen_life_vis quad_rules
```

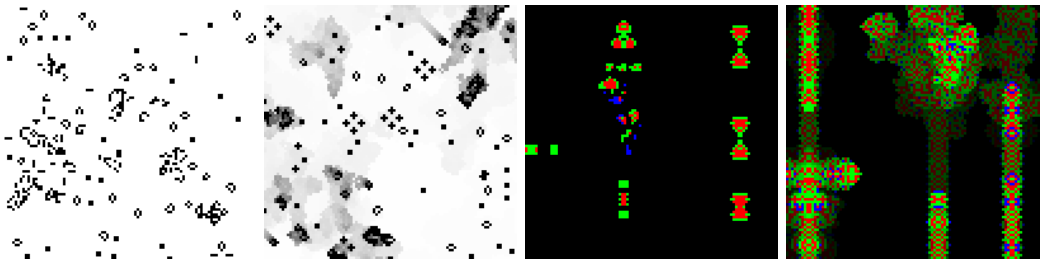
The resulting visualisations are shown on Figures 14a and 14c.

10.1 Multi-Parameter Parametric Modules

We can build more advanced visualisations on top of the above. For example, let us define a visualisation where cells that switch from being "alive" to "dead" gradually fade to the dead colour, instead of switching immediately. We can implement this feature elegantly as a parametric module, without interfering with any of the previous definitions. First, we define a module type for specifying the behaviour when a cell dies:

```
module type fading = { val dying_speed: f32 }
```

When a cell dies, it will gradually fade away from its last colour as a living cell, over a number of simulation steps controlled by `dying_speed`. We will treat a cell as dead when



(a) conway (b) conway_fading (c) quad (d) quad_fading

Fig. 14. Visualisations of variants of the Game of Life.

the result of calling `uninit` produces a false value. We can encapsulate this behavior as a parametric module:

```
module fading_life (F: fading) : vis_rules → vis_rules =
  λ(R: vis_rules): vis_rules → {
    type cell = (R.cell, argb.colour, f32)
    let value ((c,_,_): cell) = R.value c
    val weights = R.weights
    let init (b: bool) = (R.init b, argb.black, 10000.0)
    let uninit ((c,_,_): cell) = R.uninit c
    let dead (c: R.cell): bool = R.uninit c == false
    let step ((c,col,h): cell) (neighbours: i32) =
      let c' = R.step c neighbours
      let died = ! (dead c) && dead c'
      in (c', if died then R.colour c else col,
              if died then 0.0 else h + 1.0)
    let colour ((c,col,h): cell) =
      let normal = R.colour c in
      if dead c then argb.mix 1.0 col (h * F.dying_speed) normal
                else normal
}
```

Each cell is now represented as a triple of some underlying cell, the colour of the cell when it was last alive, and for how long it has been dead. Notice that the `fading_life` parametric module is *curried*. We can partially apply it to a module implementing the `fading` type to obtain another parametric module:

```
module slow_fader: (vis_rules → vis_rules) = fading_life {
  val dying_speed = 0.1
}
```

We can now apply `slow_fader` to previous rule set modules to obtain Game of Life variants with fadeout visualisation:

```
module conway_fading = slow_fader conway_rules
module quad_fading = slow_fader quad_rules
```

The resulting visualisations are shown on Figures 14b and 14d.

As static interpretation performs all module applications at compile-time, followed by vigorous function call inlining, the potential overheads of this high degree of decomposition are negated. Performance is thus equivalent to hand-written specialised implementations of our four Game of Life variants.

11 RELATED WORK

Related work fall into a number of different categories. As mentioned in the introduction, the concept of static interpretation of modules is not new and have been applied earlier in the context of the MLKit Standard ML compiler [Elsman 1999]. The concept of specialising modules (or Ada packages) is also not new and is applied for C++ templates [Veldhuizen 1999], for Ada packages, and also for the MLton compiler [Fluet and Weeks 2001]. Compared to this earlier work, the present work is extended to higher-order modules and the termination argument for static interpretation is more involved, due to the logical-relations argument.

Also related to this work is previous work on elaboration of ML modules. Aiming at typing an increasing class of higher-order modular programs, Leroy [1995] presents a distinction between applicative and generative higher-order modules, which provides the foundation for the implementation of higher-order modules in OCaml. Applicative higher-order modules allow for a refined theory of type equality, compared to the traditional generative module semantics provided by [Milner et al. 1997], by identifying applications of applicative higher-order modules when applied to identical arguments. Later developments, such as [Crary 2017], model applicative functors using a notion of totality (i.e., pure functions). The present development considers generative higher-order modules only, which are still rich enough to encode most modular patterns including modular type classes using fully transparent modules [Dreyer et al. 2007] and modular implicits [White et al. 2015]. Leroy also provides a modular modules implementation of a higher-order module system [Leroy 2000] and shows that this system can be applied to different core languages. He also briefly discusses compiling away modules in the context of first-order modules—related to the compilation of Ada packages, C++ templates, and the previous work on static interpretation of modules [Elsman 1999]. By supporting elimination of modules entirely at compile time, a modular module system will apply to a richer class of languages, including languages for which compilation does not provide a natural low-cost grouping strategy, which, among other domains, include domain-specific languages for data-parallel architectures, such as FPGAs and GPGPUs. This class of related work also includes the work by Russo [1999], which also, as in the present work, use universal and existential quantification as essential objects in the elaboration rules of the module language. In particular, modeling the generation of fresh type variables (type names in Standard ML) as existential quantification leads to an important scalable technique for proving properties about the language. Other related work in this area include the body of work on obtaining a type-theoretic account of ML modules and, in particular, of higher-order modules [Harper and Lillibridge 1994]. Because a module system can be understood as a limited dependently typed language, researchers has early on aimed at securing a phase-distinction between the static and dynamic aspects of modules [Harper et al. 1990]. Compared to our work, we go a step further and show that also the dynamic aspects of a higher-order module language can be executed (and the process expected to terminate) at compile time. The classical phase distinction is, however, still of absolute importance for a module language (to be distinguished from C++ templates) as it allows for programmers to design and reason about components in isolation. Related to this aspect and complementary to our work, Crary [2017] demonstrates a module system abstraction theorem, which says that an application context cannot distinguish two logically equivalent

modules. This body of work also includes the work on elaborating the Standard ML module language into an internal module language [Harper and Stone 2000]. Recently this work has led to a compilation of higher-order modules into F_ω, the higher-order polymorphic lambda calculus [Rossberg et al. 2014]. Compared to our work, which eliminates all module language constructs at compile time, Rossberg et al. [2014] make no distinction between core language and module language constructs in the target code. Moreover, the more complicated mechanisms supported by the language, for interaction between the module language and the core language, makes it difficult to argue that a compiler can effectively eliminate all module language constructs at compile time, by somehow partially evaluate the generated F_ω expressions. The same distinction holds between the present work and earlier work on compiling higher-order modules into an F_ω-like language with existential types [Shao 1999], which is the basis for compiling modules in the SML/NJ compiler.

Other work on modules include work on understanding how the module language may interact with mechanisms for separate compilation and smart recompilation [Shao and Appel 1993]. This body of work includes the work by Swasey et al. [2006] on extending the Standard ML language with a mechanism for truly separate compilation. Contrary to our work, the work by Swasey et al. [2006] allows for components to be compiled and understood in isolation and yet linked safely with the possibility that a component matching a particular module type can be replaced with another module matching the module type, without the need for recompiling the entire application. The earlier work on static interpretation [Elsman 1999] allows for modules, and even specialised version of parameterised modules, to be compiled separately but with the additional feature that implementation choices for exported identifiers would leak module boundaries and allow for special kinds of optimisations (including inlining of small functions) to be performed across module boundaries. The present work does not attempt at providing any mechanism for separate compilation, although a smart-recompilation framework could be useful, in particular for decreasing compile times for very large programs. At present, however, Futhark compile times are manageable and a better understanding of how to track whether a GPU kernel changes upon changes of source code is needed for constructing a reliable smart-recompilation framework for Futhark.

Also related to our work are alternative approaches at providing mechanised meta-theories for module languages, including the work by Rossberg et al. [2014], which comes with a Coq implementation of the work, but also the earlier work on using Twelf to provide a mechanised meta theory for Standard ML [Lee et al. 2007]. Compared to our work, however, none of these approaches attempts at eliminating modules at compile time. Another body of work related to mechanising the meta-theory of ML is the work on CakeML [Tan et al. 2016], which, however, supports only non-parameterised modules.

Another body of related work includes other mechanisms for removing abstractions at compile time including techniques for embedded domain specific languages (EDSLs), as discussed in the introduction, quoted domain specific languages [Najd et al. 2016], techniques for multi-stage programming, such as that by Taha and Sheard [2000], and just-in-time compilation techniques in general. Being able to construct programs, in a type-safe way, using meta-level computations (including the application of partial evaluation techniques) is a powerful concept. Static interpretation, however, allows for the user to use a module system to construct and apply powerful abstractions, yet knowing that the mechanism terminates and that the use of abstractions does not introduce any overhead. EDSL approaches leverage a host-language's abstraction mechanisms and its language features for constructing and composing programs at runtime. Such approaches provide the programmer with very powerful tooling. However, when, for instance, the source language is used as a target language for

another high-level language, both the host-language compiler and its runtime system is required to be part of the compiler pipeline. In such cases, a simpler approach, based on a self-contained language, may be preferable.

Also related to this work is Reynolds work on applying functor categories to compile the lambda-calculus part of an Algol-like language away at compile time, leaving only target code that is purely imperative [Reynolds 1995]. Unlike our approach, it is unclear how this approach applies to abstract types and a larger class of (also non-imperative) core languages.

As a final body of related work is the work on using logical relations for expressing normalisation and termination properties for the simply-typed lambda calculus and System F, which has been the inspiring work for establishing the property of termination for static interpretation. This body of work includes the seminal work by Tait [1967] and Girard [1971] on establishing the basic proof technique based on logical relations, including various adaptations of the technique [Donnelly and Xi 2007], and applications of the technique in proof assistants [Abel 2008].

12 CONCLUSION AND FUTURE WORK

We have described a technique for integrating a higher-order module language with a compiler for the data-parallel functional language Futhark. The technique, called static interpretation, eliminates entirely the module language at compile time and it can be established formally that the process terminates and yields well-typed target programs. We have shown, using a number of Futhark examples, that the structuring mechanism provided by the higher-order module language is a useful mechanism for modularity and reuse and that it, also in practice, leads to no overhead in the generated code. The formal development has been implemented and proved in Coq using a novel approach of working with semantic objects such as products, sets, and finite maps. The Coq implementation matches closely the structure of the development in both the paper and in the Haskell implementation for the Futhark language.

There are a number of possibilities for future work. First, it would be interesting to investigate whether it will be possible, perhaps in some limited fashion, to integrate modular type classes [Dreyer et al. 2007] or some of the modular implicits techniques [White et al. 2015] with the module language and still have static interpretation eliminate all module language constructs at compile time. Second, we expect that the work on establishing a standard basis library for Futhark will lead to the desire to extend the language for expressing module types using some of the suggestions given by Ramsey et al. [2005]. Third, Futhark is also used as a target language for compilation of high-level array languages, such as APL [Iverson 1962], into efficient GPU code [Annenkov and Elsman 2018; Elsman and Dybdal 2014; Henriksen et al. 2016a]. Using the new module language features, a more natural encapsulation of target language constructs is now possible, which will ease maintainability and simplify high-level code generation phases. Finally, an interesting project would be to extend the Coq implementation to make it possible to extract a certified "modular modules" implementation [Leroy 2000] for use with other domain specific languages than Futhark.

ACKNOWLEDGMENTS

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center "HIPERFIT: Functional High Performance Computing for Financial Information Technology" (http://hiperfit.dk) under contract number 10-092299. Any opinions, findings, and conclusions or

recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Danish Strategic Research Council.

REFERENCES

Andreas Abel. 2008. Normalization for the Simply-Typed Lambda-Calculus in Twelf. *Electron. Notes Theor. Comput. Sci.* 199 (Feb. 2008), 3–16.

Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. 2016. FinPar: A Parallel Financial Benchmark. *ACM Trans. Archit. Code Optim.* 13, 2, Article 18 (June 2016), 27 pages.

Danil Annenkov and Martin Elsman. 2018. Certified Compilation of Financial Contracts. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18)*. ACM.

Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. 2007. Nominal Reasoning Techniques in Coq. *Electron. Notes Theor. Comput. Sci.* 174, 5 (June 2007), 69–77. https://doi.org/10.1016/j.entcs.2007.01.028

Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. 2012. Strongly Typed Term Representations in Coq. *J. Autom. Reasoning* 49, 2 (2012), 141–159.

Guy E. Blelloch. 1989. Scans as Primitive Parallel Operations. *Computers, IEEE Transactions* 38, 11 (1989), 1526–1538.

Guy E Blelloch. 1990. *Vector models for data-parallel computing*. Vol. 75. MIT press Cambridge.

Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming (DAMP '11)*. ACM, 3–14.

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press.

Koen Claessen, Mary Sheeran, and Bo Joel Svensson. 2012. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Work. on Decl. Aspects of Multicore Prog DAMP*. 21–30.

Ranald Clouston. 2013. Generalised Name Abstraction for Nominal Sets. In *Foundations of Software Science and Computation Structures*, Frank Pfenning (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 434–449.

Karl Crary. 2017. Modules, Abstraction, and Parametric Polymorphism. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 100–113.

Kevin Donnelly and Hongwei Xi. 2007. A Formalization of Strong Normalization for Simply-Typed Lambda-Calculus and System F. *Electron. Notes Theor. Comput. Sci.* 174, 5 (June 2007), 109–125.

Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. 2007. Modular Type Classes. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 63–70.

Martin Elsman. 1999. Static Interpretation of Modules. In *Procedings of Fourth International Conference on Functional Programming (ICFP '99)*. ACM Press, 208–219.

Martin Elsman and Martin Dybdal. 2014. Compiling a Subset of APL Into a Typed Intermediate Language. In *Procs. Int. Workshop on Lib. Lang. and Compilers for Array Prog. (ARRAY)*. ACM.

Matthew Fluet and Stephen Weeks. 2001. Contification Using Dominators. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. ACM, New York, NY, USA, 2–13.

Murdoch J. Gabbay and Andrew M. Pitts. 2002. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing* 13, 3 (01 Jul 2002), 341–363. https://doi.org/10.1007/s001650200016

Jean Yves Girard. 1971. Interpretation Fonctionnelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur. In *Proceedings of the Second Scandinavian Logic Symposium*. North-Holland, 63–92.

Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2011. Breaking the GPU Programming Barrier with the Auto-parallelising SAC Compiler. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming (DAMP '11)*. ACM, New York, NY, USA, 15–24.

Robert Harper and Mark Lillibridge. 1994. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proceedings of the 21 ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '94)*. Portland, OR, 123–137.

Robert Harper, John C. Mitchell, and Eugenio Moggi. 1990. Higher-order Modules and the Phase Distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*. ACM, New York, NY, USA, 341–354.

Robert Harper and Christopher Stone. 2000. Proof, Language, and Interaction. MIT Press, Cambridge, MA, USA, Chapter A Type-theoretic Interpretation of Standard ML, 341–387.

Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. 2016a. APL on GPUs: A TAIL from the Past, Scribbled in Futhark. In *Procs. of the 5th Int. Workshop on Functional High-Performance Computing (FHPC '16)*. ACM, New York, NY, USA, 38–43.

Troels Henriksen, Martin Elsman, and Cosmin E Oancea. 2014. Size slicing: a hybrid approach to size inference in Futhark. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*. ACM, 31–42.

Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. 2016b. Design and GPGPU Performance of Futhark's Redomap Construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2016)*. ACM, New York, NY, USA, 17–24. https://doi.org/10.1145/2935323.2935326

Troels Henriksen and Cosmin E Oancea. 2014. Bounds checking: An instance of hybrid analysis. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM, 88.

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 556–571.

Anders Kiel Hovgaard. 2018. *Higher-Order Functions for a High-Performance Programming Language for GPUs*. Master's thesis. Department of Computer Science, University of Copenhagen, Universitetsparken 5, DK-2100, Denmark.

Kenneth E. Iverson. 1962. *A Programming Language*. John Wiley and Sons, Inc.

Stephen Joe and Frances Y. Kuo. 2003. Remark on Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator. *ACM Trans. Math. Softw.* 29, 1 (March 2003), 49–57.

Thomas Johnsson. 1985. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings of the international conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag New York, Inc., New York, NY, USA, 190–203. http://dl.acm.org/citation.cfm?id=5280.5292

Rasmus Wriedt Larsen and Troels Henriksen. 2017. Strategies for Regular Segmented Reductions on GPU. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing (FHPC 2017)*. ACM, New York, NY, USA, 42–52. https://doi.org/10.1145/3122948.3122952

Daniel K. Lee, Karl Crary, and Robert Harper. 2007. Towards a Mechanized Metatheory of Standard ML. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 173–184.

Xavier Leroy. 1995. Applicative Functors and Fully Transparent Higher-order Modules. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 142–153.

Xavier Leroy. 2000. A Modular Module System. *Journal of Functional Programming* 10, 3 (May 2000), 269–303.

Geoffrey Mainland and Greg Morrisett. 2010. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell (Haskell '10)*. ACM, New York, NY, USA, 67–78.

Trevor L. McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising Purely Functional GPU Programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised)*. MIT Press.

Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything Old is New Again: Quoted Domain-specific Languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '16)*. ACM, New York, NY, USA, 25–36.

Nathaniel Nystrom, Derek White, and Kishen Das. 2011. Firepile: Run-time Compilation for GPUs in Scala. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE '11)*. ACM, New York, NY, USA, 107–116.

Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. 2000. Composing Contracts: an Adventure in Financial Engineering (functional pearl). In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, 280–292.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2016. *Software Foundations*. Electronic textbook. Version 4.0. http://www.cis.upenn.edu/~bcpierce/sf.

Andrew M. Pitts. 2013. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, New York, NY, USA.

Norman Ramsey, Kathleen Fisher, and Paul Govereau. 2005. An Expressive Language of Signatures. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM, New York, NY, USA, 27–40.

John C. Reynolds. 1995. Using Functor Categories to Generate Intermediate Code. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 25–36.

Andreas Rossberg and Derek Dreyer. 2013. Mixin' Up the ML Module System. *ACM Transactions on Programming Languages and Systems* 35, 1, Article 2 (April 2013), 2:1–2:84 pages.

Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 2014. F-ing modules. *Journal of Functional Programming* 24, 5 (2014), 529–607.

Claudio V. Russo. 1999. Non-dependent Types for Standard ML Modules. In *Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming (PPDP '99)*. Springer-Verlag, London, UK, UK, 80–97.

Zhong Shao. 1999. Transparent Modules with Fully Syntatic Signatures. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*. ACM, New York, NY, USA, 220–232.

Zhong Shao and Andrew W. Appel. 1993. Smartest Recompilation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*. ACM, New York, NY, USA, 439–450.

Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. *SIGPLAN Not.* 50, 9 (Aug. 2015), 205–217.

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-parallel IR for High-performance GPU Code Generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO 2017)*. IEEE Press, Piscataway, NJ, USA, 74–85.

Joel Svensson. 2011. *Obsidian: GPU Kernel Programming in Haskell*. Ph.D. Dissertation. Chalmers University of Technology.

David Swasey, Tom Murphy VII, Karl Crary, and Robert Harper. 2006. A Separate Compilation Extension to Standard ML. In *Proceedings of the ACM SIGPLAN Workshop on ML (ML '06)*. ACM.

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248, 1 (2000), 211 – 242. PEPM'97.

William W. Tait. 1967. Intensional interpretations of functionals of finite type. *Journal of symbolic logic* 32 (1967), 198–212.

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2016. A New Verified Compiler Backend for CakeML. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 60–73.

Todd L. Veldhuizen. 1999. C++ Templates as Partial Evaluation. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. 13–18. Technical report BRICS-NS-99-1.

P. Wadler. 1987. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. ACM, New York, NY, USA, 307–313.

P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, USA, 60–76.

Leo White, Frédéric Bour, and Jeremy Yallop. 2015. Modular implicits. *EPTCS* 198, 2 (December 2015), 22–63. In Proceedings of ML/OCaml Workshop 2014. arXiv:1512.01438.