A Methodology for Generating Verified Combinatorial Circuits*

Oleg Kiselyov
Fleet Numerical Meteorology and Oceanography
Center, Monterey, CA 93943
oleg@okmij.org

Kedar N. Swadi   Walid Taha
Department of Computer Science
Rice University
{kswadi,taha}@rice.edu

ABSTRACT

High-level programming languages offer significant expressivity but provide little or no guarantees about resource use. Resource-bounded languages — such as hardware-description languages — provide strong guarantees about the runtime behavior of computations but often lack mechanisms that allow programmers to write more structured, modular, and reusable programs. To overcome this basic tension in language design, recent work advocated the use of Resource-aware Programming (RAP) languages, which take into account the natural distinction between the development platform and the deployment platform for resource-constrained software.

This paper investigates the use of RAP languages for the generation of combinatorial circuits. The key challenge that we encounter is that the RAP approach does not safely admit a mechanism to express a posteriori (post-generation) optimizations. The paper proposes and studies the use of abstract interpretation to overcome this problem. The approach is illustrated using an in-depth analysis of the Fast Fourier Transform (FFT). The generated computations are comparable to those generated by FFTW.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Automatic synthesis*; D.1.1 [**Programming Techniques**]: Functional Programming

General Terms

Design, Languages, Performance

Keywords

Multi-stage programming, abstract interpretation

1. INTRODUCTION

Hardware description languages are primarily concerned with resource use. But except for very high-end applications, verifying the correctness of hardware systems can be prohibitively expensive. In contrast, software languages are primarily concerned with issues of expressivity, safety, clarity, and maintainability. Software languages provide abstraction mechanisms such as higher-order functions, polymorphism, and general recursion. Such abstraction mechanisms can make designs more maintainable and reusable. They can also keep programs close to the mathematical definitions of the algorithms they implement, which helps with ensuring correctness. Hardware description languages such as VHDL [16] and Verilog [32] provide only limited support for such abstract mechanisms. The growing interest in reconfigurable hardware invites us to consider the integration of the hardware and software worlds, and to consider how verification techniques from one world can be usefully applied in the other. Currently, programming reconfigurable hardware is hard [3]: First, software developers are typically not trained to design circuits. Second, specifying circuits by hand can be tedious, error prone, and difficult to maintain. The challenge in integrating both hardware and software worlds can be summarized by a key question:

> *How can we get the raw performance of hardware without giving up the expressivity and clarity of software?*

1.1 Generators and Manifest Interfaces

Recent work on Resource-aware Programming (RAP) [30] in the context of *software* generation suggests a promising approach to *hardware verification*: Rather than verifying circuits on a case-by-case basis, we propose that the circuit designer express generic specifications that can automatically *generate* a whole *family* of circuits. The technical novelty of this approach is that once the generic specification is verified, we are guaranteed that all generated circuits will be correct. In this approach, naively-generated circuits are correct by construction. More efficient circuits are correct because they are produced by systematic, verified improvements on a correct but naive generator and *not* by verifying a naive generator and verifying a posteriori (post-generation) optimizations that fix up the result of the generator. From the verification point of view, this means that we replace the problem of verifying transformations to one of verifying modifications to one program: the generator.

A classic example of such generators is one used by Selesnick and Burrus [11] to produce Fast Fourier Transform (FFT) circuits for prime-numbered sizes. A more extensive survey can be found in Frigo's account of the FFTW system [8]. Writing and using program generators, however, has its own challenges (cf. [27]). One such challenge is that manifest interfaces for generators are hard to

*Supported by NSF ITR-0113569 "Putting Multi-stage Annotations to Work."

EMSOFT'04, September 27–29, 2004, Pisa, Italy.
Copyright 2004 ACM 1-58113-860-1/04/0009 ...$5.00.

express in traditional type systems. For example, if strings, algebraic datatypes, parse trees, or even graphs are used to represent the generated program, they would only allow us to express a manifest interface with a type such as:

```
gen_fft : int -> circuit
```

where `circuit` is the type we choose to represent circuits with. As soon as we start *composing* generators — for example, if we want to build a circuit that computes the FFT, performs a multiplication, and then computes the inverse FFT — we run into a problem: The type `circuit` does not provide any static guarantees about the consistency or well-formedness of the composite circuit. To illustrate, assume we are given two trivial generators which take no inputs and produce an AND-gate and an inverter:

```
and : circuit
inv : circuit
```

A meaningless composition arises if we write:

```
let bad = inv --> and
```

where the connect operator `-->` is an infix operator that has the type `circuit × circuit -> circuit` and which wires the output of its first circuit to the input of the second circuit. The problem is that the second circuit does not have just one input but *two*, and the type system does not prevent this error: all circuits just have type `circuit`.

It is generally desirable that the `circuit` type be as expressive as possible, but at the same time only express values that are circuit-realizable. For example, the programmer might want to use abstractions such as lists (or any other dynamic data structure) in describing the circuit, but will need to know as early as possible in the development process that these uses can be realized using finite memory [10, 30].

1.2 Resource-aware Programming

Resource-aware Programming (RAP) languages [30] are designed to address the problems described above by

1. Providing a *highly expressive untyped substrate* supporting features such as dynamic data-structures, modules, objects, and higher-order functions.

2. Allowing the programmer to express the *stage distinction* between computation on the development platform and computation on the deployment platform. Convenient notation and static type safety can be ensured by using multi-stage programming (MSP) constructs [29, 28].

3. Using *advanced static type systems* to ensure that computations intended for execution on resource-bounded platforms are indeed resource-bounded [30, 31].

The combination of these three ingredients allows the programmer to use sophisticated abstraction mechanisms in programs that are statically guaranteed to generate only resource-bounded programs.

For example, rather than using one concrete type to represent circuits, RAP languages provide an abstract datatype parameterized by information about the generated code. The type of the two trivial generators above would be:

```
and : (bool × bool -> bool) code
inv : (bool -> bool) code
```

The type of the connect operator `-->` would be refined from being

```
circuit × circuit -> circuit
```

to being

```
(α -> β)code × (β -> γ)code -> (α -> γ)code
```

where α, β, and γ are generic type variables that must always be instantiated consistently. With this extra information, the type system can reject the above `bad` declaration, because the type variable β cannot be instantiated to both the output of `inv` (which is `bool`) and the input of `and` (which is `bool×bool`). Note that the type of this function is similar to the type of the standard mathematical function composition operation:

```
(α -> β) × (β -> γ) -> (α -> γ)
```

In addition to ensuring that the generated programs will be well-typed, RAP languages can also ensure that the generated programs satisfy various notions of resource constraints (cf. [30]).

1.3 Problem

To ensure that generated programs are well-typed and resource-bounded *before* they are generated, the code type in a RAP language must remain abstract. Providing constructs for traversing values of this type jeopardizes the soundness and decidability of static typing [27], and complicates reasoning about the correctness of programs written in these languages [28]. At the same time, not being able to look inside code means that a posteriori optimizations cannot be expressed within the language. While such optimizations can still be implemented as stand-alone source-to-source transformations *outside* the language, doing so invalidates the safety and resource-boundedness guarantees.

We distinguish two forms of a posteriori optimizations: generic ones that are independent of the application, and ones that are specific to the application. Generic optimizations are generally well-tested and are less likely to invalidate the guarantees provided by the RAP setting. But domain-specific optimizations written by the programmer for the particular application are less likely to have been tested as extensively, and are therefore more problematic. At the same time, systems such as FFTW make a strong case for the practical importance of such domain-specific optimizations [8]. We are therefore faced with a technical problem:

> *How can we implement domain-specific optimizations without losing the benefits of the RAP framework?*

1.4 Contributions

The paper proposes the use of abstract interpretation [5] on program generators to avoid the need for a posteriori optimization. This allows us to generate the desired circuits without losing the guarantees provided by RAP languages. The benefits of the proposed technique extend to the untyped setting, as it avoids the generation of large circuits in the first place, thus reducing the overall runtime needed to generate acceptable code. From the verification point of view, this approach replaces the problem of verifying a source-to-source transformation to that of verifying the correctness of a finite set of optimizations on *one* specific program: the generator.

In the proposed method, abstract interpretation is carried out after four initial, standard steps for building program generators [27]:

1. Implement the input-output behavior in an expressive, type-safe language such as OCaml [15]. As a running example, we will use FFT. For FFT, this step is just implementing the Cooley-Tukey recurrence for computing the FFT.

2. Verify the correctness of the input-output behavior of this program. Because we used an expressive language to implement FFT, this step reduces to making sure, first, that the textbook definition of the Cooley-Tukey recurrence is transcribed correctly, and, second, that the program is correctly transformed into a monadic style. The monadic transformation is well-defined and mechanizable [21], and this paper will explain why monadic transformation (or style) is convenient when using the proposed approach. Often, it is not necessary to convert the whole program into monadic style. So, when the transformation is done by hand, it is only done as needed.

3. Identify the relevant computational stages for the program (cf. [27]). This involves determining the parts of the computation that can be done on the development platform, and those that must be left to the deployment platform.

4. Add staging annotations. In this step, staging constructs (hygienic quasi-quotations) ensure that this is done in a semantically transparent manner. A two-level type system understands that we are using quasi-quotations to generate programs (cf. [31]) and can ensure that there are no inconsistent uses of first- and second-stage inputs. A RAP type system [30] goes further and can ensure that second-stage computations only use features and resources that are available on the target platform.

The source code of the resulting generator is often a concise, minor variation on the result of the first step. If the quality of the generated code is not satisfactory, the paper proposes the following additional step:

5. Use abstract interpretation techniques to shift more computations to the development platform rather than the deployment platform. This step generally leads to smaller and more efficient circuits.

In the short term, this technology can reduce the time and effort needed for programming reconfigurable hardware. Reconfigurable hardware [6, 17] has the potential for delivering significant performance improvement in computationally intensive application domains. For example, Najjar et al. [22] report speedups of 10-800 times over highly-tuned software implementations. Rather than performing the numerically intensive kernels in the "native" software platform where the rest of the application is implemented, their system "offshores" these kernels to field-programmable gate arrays (FPGAs). Field-programmable hardware is well-suited for massively-parallel implementations of computations that can be expressed as combinatorial circuits. In the longer term, we hope that the proposed approach can have a positive impact on VLSI logic design and verification.

1.5 Related Work

Our work builds on a long tradition of using functional programming languages to describe hardware circuits, such as Ruby [12], Lava [2], Hawk [14], HML [18], Hydra [23], reFLect [9]. However, none of these languages provide the kind of manifest interfaces (static types) discussed above.

Hardware-description languages recognize the need for macros to distinguish circuits descriptions from circuits. An example is SA-C [3], a single-assignment, array-based language. While SA-C's facility is an improvement on the C macro system, it provides neither the expressivity of higher-order languages nor the manifest interfaces and the static guarantees delivered by RAP type systems.

McKay and Singh [19] use partial evaluation (an automated approach to staging) for dynamic specialization of FPGAs. But their specialization and optimizations of circuits use intensional analysis of programs. For any partial evaluation system, the user can only chose between two levels of abstraction: Either to treat the tool as a black box, in which case all control over the generation process is delegated to the tool, or to treat the tool as a white box, in which case all well-formed and correctness guarantees about the generated code are void. Two-level static type systems allow the programmer to *safely* gain full control over the generation process.

SAFL+ [26] allows the programmer to breakdown a computation into components implemented by hardware or software. SAFL+ is first-order, monomorphically typed language, and the *whole* computation must be resource bounded. Thus, from the RAP point of view, the whole SAFL+ computation is performed on the deployment platform. HardwareC [13] is similar to SAFL+, but uses only C-like imperative features.

1.6 Organization of this Paper

Section 2 gives a quick introduction to the basics of staging and explains the basic approach in the context of a minimal example. Though the static type system of MetaOCaml checks only for type correctness and not circuit-realizability of `code`, it is sufficient to present and validate the idea of abstract interpretation on code generators. Section 3 describes how the Fast Fourier Transform (FFT) can be expressed in a functional language and then staged. In Section 4 we describe generation-time optimizations. These optimizations are enabled by abstract interpretation of the generated code. We show the effectiveness of this technique by comparing the number of floating point operations to those in the results of both naively staged versions and the FFTW systems. Section 5 concludes.

2. ABSTRACT INTERPRETATION OF `POWER`

Staging constructs are a mechanism for distinguishing computational stages in a program. The following minimal example illustrates the use of these constructs in MetaOCaml [4, 20]:

```
let rec power n x =
   if n=0 then .<1>.
          else .< .~x * .~(power (n-1) x)>.
let power3 = .<fun x -> .~(power 3 .<x>.)>.
```

Ignoring the staging constructs (brackets `.<`e`>.` and escapes `.~`e) the above code is a standard definition of a function that computes x^n, which is then used to define the specialized function x^3. Without staging, the last step simply returns a function that would invoke the `power` function every time it gets invoked with a value for x. In contrast, the staged version builds a function that computes the third power directly (that is, using only multiplication). To see how the staging constructs work, we can start from the last statement in the code above. Whereas a term `fun x ->` e `x` is a value, an annotated term `.<fun x -> .~(`e `.<x>.)>.` is not, because the outer brackets contain an escaped expression that still needs to be evaluated. Brackets mean that we want to construct a future stage computation, and escapes mean that we want to perform an immediate computation *while* building the bracketed computation. In a multi-stage language, these constructs are not hints, they are imperatives. Thus, the application e `.<x>.` must be performed even though `x` is still an uninstantiated symbol. In the `power` example, `power 3 .<x>.` is performed immediately, once and for all, and not repeated every time we have a new value for `x`. In the body of the definition of the function `power`, the recursive application of `power` is escaped to ensure its immediate execution in the first stage. Evaluating the definition of `power3` first results in

```
.<fun x -> x * x * x * 1>.
```

Whereas implementing the unstaged definition of `power3` in hardware is non-trivial, the staged one evaluates to a program that is clearly circuit-realizable.

2.1 Why Abstract Interpretation?

To give a minimal example of why abstract interpretation on generators is useful, consider the presence of the multiplication by `1` in the body of power3. Resorting to a posteriori techniques for eliminating such unnecessary computations *after* the code has been generated requires making the `code` data type less abstract. This has the disadvantage of voiding equational reasoning principles on computation inside brackets [28], as this essentially reduces them to syntactic quotations (cf. [1, Footnote 32]). Equally importantly, ensuring static type safety would then necessitate the use of higher-order types (cf. [25]). Abstract interpretation allows us to avoid these problems and still achieve essentially the same result.

The first step in applying abstract interpretation is to identify the *concrete domain*, which is generally the code type used in our program, and which we would like to look inside.

```
type concrete_code = (int code)
```

This type is generally implicit in the original program. The second step is to design an *abstract domain* that provides us with more information about the code value. For example, we can use:

```
type abstract_code =
      One
    | Any of (int code)
```

This type splits the single case in `concrete_code` into two: The first indicates that we have more information about the code value, namely, that it is the literal `1`. The second says that we have no additional information about the code value. Note that while the terms concrete and abstract may seem backwards, they are not: the abstract type approximates the *second stage value* of the concrete type.

To see how this abstract type contains more information than the original type, all we need to do is to present the *concretization* function that converts any `abstract_code` into `concrete_code`:

```
let conc (c:abstract_code):concrete_code =
  match c with
      One -> .<1>.
    | Any c -> c
```

The next step is to lift all the operators from the concrete type to the abstract type, so that some useful work can be done in the first stage, and less work is left for the second stage. The only operator that the `power` function uses is the multiplication operator `*`. To achieve the desired effect, we define the corresponding abstract operator `**` as follows:

```
let ( ** ) x y =
  match (x,y) with
      (One, One) -> One
    | (One, y)   -> y
    | (x,   One) -> x
    | (x,   y)   ->
   Any .< .~(conc x) * .~(conc y)>.
```

As can be seen from this definition, the `abstract_code` type makes it possible to directly express optimizations that would have required inspecting the `concrete_code` type.

Whereas the concretization function can be expressed within the language, the *abstraction function* that would go the other way cannot. In particular, the latter requires inspecting values of the abstract type `code`.

The staged function can now be expressed as:

```
let rec power n x =
   if n=0 then One
          else x ** (power (n-1) x)
```

and evaluating the declaration:

```
let power3 =
 .<fun x -> .~(conc (power 3 (Any .<x>.)))>.
```

would yield precisely the desired result:

```
.<fun x -> x * x * x>.
```

and the last multiplication is eliminated. For this simple example, abstract interpretation on the generated code provides a systematic and safe approach to achieving essentially the same results as a posteriori optimization. The next two sections show how this scales to the more substantial example of FFT, yielding results comparable to those produced by the FFTW system.

3. STAGING FFT

FFT finds applications in many time-critical embedded applications, and it is therefore important to be able to generate efficient circuits for it. Using FFT as a running example for this paper, this section begins by showing how FFT can be implemented in a purely functional manner in OCaml. Then, MetaOCaml's staging constructs are used to express a variant of this function that can be specialized with respect to the size of the input vector.

3.1 An Unstaged FFT Implementation

The basic Cooley-Tukey recurrence can be implemented as:

```
let rec fft dir l =
  if (List.length l = 1) then
    l
  else
    let (e,o) = split l in
    let y0 = fft dir e  in
    let y1 = fft dir o  in
    merge dir y0 y1
```

with the two parameters being: the direction flag `dir`, and the input vector `l` represented as a list. If the input vector has length one, we simply return it. Otherwise, we split the vector into even and odd components `(e,o)`, recursively apply `fft` to these subvectors, and then `merge` the results.

3.2 Complex Numbers and Auxiliary Functions

Complex numbers are represented as pairs of OCaml floats, and operations such as complex `add`ition are implemented as follows:

```
let add ((r1,i1), (r2, i2)) =
       ((r1 +. r2), (i1 +. i2))
```

The functions `split`, and `merge` that are used in defining `fft` are implemented as follows:

```
let rec  split l =
  match l with
    [] -> ([], [])
  | x::y::xs ->
      let (a,b) = split xs in (x::a, y::b)

let rec merge dir l1 l2 =
  let n = 2 * List.length l1 in
  let rec mg l1 l2 j =
    match (l1, l2) with
      (x::xs, y::ys) ->
        let z1 = mult (w dir n j, y) in
        let zx = add (x, z1) in
        let zy = sub (x, z1) in
        let (a,b) = (mg xs ys (j+1)) in
        (zx::a, zy::b)
    | _ -> ([], []) in
  let (a,b) = mg l1 l2 0 in (a @ b)
```

The function `w` computes powers of the nth complex root of unity.

3.3 Monadic Sharing

To avoid explosion in the size of the generated code, we use a monadic library for sharing [7]. This requires that we rewrite parts of the FFT program into a monadic style with explicit open recursion:

```
let fft dir f l =
  if (List.length l = 1) then
    ret l
  else
    let  (e,o) = split l in
    bind (f e) (fun y0 ->
    bind (f o) (fun y1 ->
    ret (merge dir y0 y1)))
```

The new parameter `f` is now used in place of all recursive calls to `fft dir` in the body of the original function. Return (`ret`) and `bind` are the two standard monadic operators [21, 33]. To get exactly the same functionality as in the original program, we would use a monadic library where `ret` is the identity function (no-op), and `bind` passes the result of the first argument to the the second one. Details of the monadic library for monadic sharing are beyond the scope of this paper, and are described elsewhere [7]. For the purposes of this paper, the reader should view it as a library that avoids code duplication during generation.

For this example, only the `fft` and `merge` functions need to be converted into monadic style.

3.4 Staging FFT

To stage the FFT with respect to *the size* of the input vector, we add staging annotations to get a staged FFT function `fft_ms`. The staged function takes a vector of *code values* that denote the delayed elements. Since the operation of `split` is parametric in the elements, it requires no explicit staging. The `merge` function is now written in a monadic style, and the only other change is the use of staged versions of the complex arithmetic functions, namely, `w_s`, `add_s`, `sub_s`, and `mult_s` (The `retS` operator is exactly the same as the `ret` operator used before). These staged functions are achieved by adding staging annotations to the original ones. For example, the `add_s` function is now defined as:

```
let add_s ((r1,i1), (r2, i2)) =
    ((.<.~r1 +. .~r2>.), (.<.~i1 +. .~i2>.))

let merge_ms dir l1 l2 =
  let n = 2 * List.length l1 in
  let rec mg l1 l2 j =
    match (l1, l2) with
     (x::xs, y::ys) ->
       bind (retS (mult_s (w_s dir n j, y)))
                                   (fun z1 ->
       bind (retS (add_s (x, z1)))(fun zx ->
       bind (retS (sub_s (x, z1)))(fun zy ->
       bind (mg xs ys (j+1)) (fun (a,b) ->
       retS (zx::a, zy::b)))))
    | _ -> retS ([], []) in
    bind (mg l1 l2 0) (fun (a,b) ->
    retS (a @ b))
```

The FFT function now uses the staged, monadic version of `merge`:

```
let fft_ms dir f l =
  if (List.length l = 1) then retS l
  else
    let (e,o)  = split l in
    bind (f e) (fun y0 ->
    bind (f o) (fun y1 ->
    merge_ms dir y0 y1))
```

To use `fft_ms`, it is passed to a monadic fixed point operator `y_sm`, and the resulting monadic value is passed to an appropriate monadic run combinator `runM`:

```
.<fun x ->
   .~(run (y_sm (fft_ms 1.0)) (2 * n) .<x>.)>.
```

In what follows we will focus on the quality of the code generated from this computation, and how it can be improved through the use of abstract interpretation.

3.5 Generated Code

Auxiliary functions and a non-standard run construct (see [7]) allow MetaOCaml to output C function code corresponding to the staged definition specialized for the size of the input vector. The preliminary syntax for the non-standard run construct is currently `.!{Trx.run_gcc}`, where `gcc` can be replaced by names of different back-end compilers such as `icc` or `f90`.

For an input vector of size 4 we get the following output, the result of translating to C is:

```
int __fun_def(double * x_234 ) {
  double x1_235 ; double y1_236 ; double x1_237 ;
  double y1_238 ; double x1_239 ; double y1_240 ;
  double x1_241 ; double y1_242 ; double y1_243 ;
  double y2_244 ; double y1_245 ; double y2_246 ;
  double y1_247 ; double y2_248 ; double y1_249 ;
  double y2_250 ; double y1_251 ; double y2_252 ;
  double y1_253 ; double y2_254 ; double y1_255 ;
  double y2_256 ; double y1_257 ; double y2_258 ;

  x1_235 = x_234[0]; y1_236 = x_234[1];
  x1_237 = x_234[2]; y1_238 = x_234[3];
  x1_239 = x_234[4]; y1_240 = x_234[5];
  x1_241 = x_234[6]; y1_242 = x_234[7];
  y1_243 = x1_235; y2_244 = y1_236;
  y1_245 = x1_239; y2_246 = y1_240;
  y1_247 = y1_243 + (1. * y1_245 - 0. * y2_246);
  y2_248 = y2_244 + (1. * y2_246 + y1_245 * 0.);
  y1_249 = y1_243 - (1. * y1_245 - 0. * y2_246);
  y2_250 = y2_244 - (1. * y2_246 + y1_245 * 0.);
  y1_251 = x1_237; y2_252 = y1_238;
  y1_253 = x1_241; y2_254 = y1_242;
  y1_255 = y1_251 + (1. * y1_253 - 0. * y2_254);
  y2_256 = y2_252 + (1. * y2_254 + y1_253 * 0.);
  y1_257 = y1_251 - (1. * y1_253 - 0. * y2_254);
  y2_258 = y2_252 - (1. * y2_254 + y1_253 * 0.);
  x_234[0] = y1_247 + (1. * y1_255 - 0. * y2_256);
  x_234[1] = y2_248 + (1. * y2_256 + y1_255 * 0.);
  x_234[2] = y1_249
    + (6.12303176911e-17 * y1_257 - -1. * y2_258);
  x_234[3] = y2_250
    + (6.12303176911e-17 * y2_258 + y1_257 * -1.);
  x_234[4] = y1_247 - (1. * y1_255 - 0. * y2_256);
  x_234[5] = y2_248 - (1. * y2_256 + y1_255 * 0.);
  x_234[6] = y1_249
    - (6.12303176911e-17 * y1_257 - -1. * y2_258);
  x_234[7] = y2_250
    - (6.12303176911e-17 * y2_258 + y1_257 * -1.);
  x_234; return 0;
}
```

The function `__fun_def` takes an array of four complex numbers, realized as an array of four pairs of `doubles`; two floating-point numbers in a pair represent the real and the imaginary parts of the complex number, respectively. The function computes the FFT in-place: on exit from the function, the input array will contain the computed transform. This code represents the fully unfolded complex FFT computation for the sample size of 4. It is a single-assignment and straight-line code, that can easily be translated to combinatorial circuits.

3.6 What's Wrong with the Generated Code?

The generated code points to a need for domain-specific optimizations. It suffers from obvious problems that include having

- Repeated computations such as `(1. * y1_245 - 0. * y2_246);` that appear as subexpressions in larger expressions.
- Statements such as `x1_235=x_234[0];` assign array elements to temporaries, and these temporaries are used only once. We also have statements such as `y1_243=x1_235;`. Both kinds of statements perform unnecessary moves.
- Trivial but expensive floating point multiplication by factors such as 1.0 and 0.0.
- Round-off errors that result in unnecessary computation, such as multiplication by 6.12303176911e-17. The exact 4-point FFT does not contain such factors, as they should be exactly zero. Furthermore, replacing such factors by exact zeros would lead to the cascade of further simplifications.

While aggressive compiler optimizations might eliminate some of these problems, for many of them, *ensuring* that they are eliminated requires knowledge about the FFT algorithm. Note that this does *not* mean that a posteriori optimizations are needed. In what follows we illustrate this point and show that there are benefits to focusing on writing better generators rather than on fixing the results of simple generators.

4. ABSTRACT INTERPRETATION OF `FFT`

Through the use of abstract interpretation and a series of refinements to the generator, we will show how the problems identified at the end of the last section can be addressed. A key feature of all these modifications is that they extract information about the second-stage (or generated) computations and make it available in the first stage.

4.1 Abstraction Domain

We want to avoid code duplication, and we want to avoid trivial multiplications and additions. Therefore, we need two abstraction domains, which we stack one on top of the other. Both domains successively refine the `float` type that was used for the elements of the vector produced by FFT.

The first domain

```
type maybeValue = Val of float code
  | Exp of float code
```

keeps track of whether a code value is cheap to duplicate. A complicated expression is tagged `Exp`, while a simple expression (a float literal, for example) is tagged `Val`.

The second domain

```
type abstract_code =
        Lit of float
      | Any of float * maybeValue
```

allows us to construct complex arithmetic operators that discriminate between literals that are known at generation time and computations that are not. Computations that have values unknown at the generation stage are tagged with `Any`. If this value represents multiplication by a known factor, we keep that factor as a floating-point number.

Both datatypes are analogous to the one discussed in Section 2.1, but they carry more information about the code value. We must stress that this information — if the fragment is literal, is a simple variable reference, etc. — is *not* obtained by looking inside the code fragment. Rather, we make note of this information when we *generate* the code fragment, from data available to us at generation time. We never look inside code after it is generated. Elements of these types can be viewed as (sometimes exact) approximations of the future-stage value. Note also that abstract interpretation is with respect to the generated *code fragment*.

4.2 The Abstract Interpretation

Just as we did in Section 2.1, we define concretization functions for the two abstract domains defined above.

```
let mVconc = function
    (Val x) -> x
  | (Exp x) -> x
let conc = function (Lit x) -> Val .<x>.
  | Any(1.0,x)->  x
  | Any(-1.0,x)-> Exp .<-. .~(mVconc x)>.
  | Any(factor,x)-> Exp .<factor *. .~(mVconc x)>.
```

These functions successively forget information. The result of `mVconc` is the opaque code value; all information we had about the value at the generation stage is forgotten, while the function `conc` internalizes the multiplication factor of the computation, and yields a `maybeValue`.

4.3 Avoiding Code Duplication and Trivial Bindings

The butterfly operation offers an opportunity for avoiding repeated computation. In this operation, represented by the following code snippet which appears in `merge_ms` above:

```
   ...
   bind (retS (mult_s (w_s dir n j, y)))
                               (fun z1 ->
   bind (retS (add_s (x, z1))) (fun zx ->
   bind (retS (sub_s (x, z1))) (fun zy ->
   ...
```

the multiplication `mult_s (w_s dir n j, y)` is used in the subsequent `add_s` and `sub_s` operations. One might expect that binding the result of this value to `z1` avoids repeating this computation. This is indeed the case in the unstaged program. But in the staged program, the multiplication in the first line is really symbolic, and because `z1` is used in two places, this computation is duplicated. We already mentioned one technique for avoiding some forms of code explosion (monadic sharing [7]). The butterfly problem points to a need for a concise and controlled way for naming intermediate results of generation, so as to avoid the duplication of expressions that contain computations. We achieve this by defining a variant of the monadic return operator that names its argument so that only the name gets duplicated. We call this variant `retN` and define it in Appendix A. While `retN` operates on single values, we must deal with tuples in the `maybeValue` domain that represent the complex number. Therefore, we “raise” `retN` to work with values of this type using the function `liftcM retN_v` defined in Appendix A, which allows us to generate name bindings only for the `Exp` variant and avoid generation of trivial bindings (for the `Val` variant). The resultant code for the merge operation shown below

```
let merge_mv dir l1 l2 =
  let n = 2 * List.length l1 in
  let rec mg l1 l2 j =
    match (l1, l2) with
      (x::xs, y::ys) ->
        bind ((liftcM retN_v) x) (fun x ->
          bind ((liftcM retN_v) y) (fun y ->
          bind ((liftcM retN_v)
                 (mult_sv (w_sv dir n j, y)))
                                   (fun z1 ->
          bind (retS (add_sv (x, z1))) (fun zx ->
          bind (retS (sub_sv (x, z1))) (fun zy ->
          bind (mg xs ys (j+1)) (fun (a,b) ->
          retS (zx::a, zy::b)))))))
      | _ -> retS ([], []) in
      bind (mg l1 l2 0) (fun (a,b) ->
      retS (a @ b))
```

is different from the `merge_ms` function in two ways: first, it uses `liftcM retN_v` instead of `retS` wherever we have an opportunity for a controlled naming of expressions, and second, it uses complex arithmetic operators, viz., `add_sv`, `sub_sv`, `mult_sv`, and `w_sv` that work on the `maybeValue` domain rather than on the `float` domain. The `add_sv` operator is a simple change from the staged add operator `add_s`:

```
let mV_add x y =
  let xc = mVconc x in
  let yc = mVconc y in
  Exp .<.~xc +. .~yc>.
let add_sv ((r1,i1), (r2, i2)) =
     (mV_add r1 r2, mV_add i1 i2)
```

The generated code that results form replacing `merge_ms` with `merge_mv` in the `fft_ms` function avoids the duplication of expressions, and also avoids the generation of names for trivial expressions.

4.4 Avoiding Trivial Operations

As in the case of the multiplication function in Section 2.1, once we replace the code type by the abstract domain type, we must lift the staged complex arithmetic functions into the abstract domain. In essence, this means that we have to study how each of these operations should be defined for each of the different cases that can arise as a result of using abstract interpretation. Just as the abstract multiplication operator (`**`) in Section 2.1 made use of the identity $x * 1 = x$ to avoid redundant computations in the generated code, we can similarly use identities to avoid unnecessary additions and multiplications in the FFT code.

For example, in the addition function `add_a` defined over the abstract domain `abstract_code`, we can use the identity $x + 0.0 = x$ to avoid the generation of an unnecessary addition operator. In this function, the `Lit` variant allows us to discriminate zero values from others, and thus, we can perform a case analysis as given by the code fragment below:

```
let rec add_a (n1, n2) =
  ...
  match (n1, n2) with
    (Lit 0.0,x) -> x
  | (x, Lit 0.0) -> x
  ...
```

Similarly, if we know that the value pairs have the same factors, we use this information to avoid the generation of unnecessary multiplications using the identity $f * x + f * y = f * (x + y)$, as given by the code fragment

```
  | (Any (fx,x), Any (fy,y)) ->
      if fx = fy then Any (fx,mV_add x y) else ...
```

In this case, we generate the code for addition, `mV_add x y` but we do not generate the code for multiplication by the common factor. Rather, we carry the factor along. Multiplication by the factor will be generated only when needed. The above excerpt illustrates how information known from the annotations of the input fragments is used to set the appropriate annotations on the output fragments. Again, we never examine the generated code itself.

The addition operation `add_a` on individual values is used to define the addition operator for complex values:

```
let add_ta ((r1, i1), (r2, i2)) =
  (add_a (r1, r2), add_a (i1, i2))
```

Subtraction (`sub_ta`) and multiplication (`mul_ta`) operators that use abstract interpretation to avoid the generation of trivial operations in the code are similarly defined.

4.5 Avoiding Round-off Errors

The factors in the FFT algorithm are the roots of unity of the order N, where N is the sample size. Because N is known at generation time, we can compute the factors then. The following function `w_a` does the computation and returns the result as a pair of `Lit` floating-point numbers representing one complex number.

```
let w_a dir n j =
  (* exp( dir* -2PI I j/n), where dir is +/-1 *)
  if j = 0 then (Lit 1.0, Lit 0.0) else
  if 2*j = n then (Lit (-1.0), Lit 0.0) else
  (* exp(dir* -PI I) *)
  if 4*j = n then (Lit 0.0, Lit (-. dir)) else
  (* exp( dir* -PI/2 I) *)
  if 4*j = 3*n then (Lit 0.0, Lit dir) else
  (* exp( dir* -3*PI/2 I) *)
  if 8*j mod n = 0 then (* 8j/n must be odd *)
    let quadrant = ((8*j / n) - 1)/2 and
        cos_signs= [| 1.0; -1.0; -1.0; 1.0 |] and
        sin_signs= [| 1.0;  1.0; -1.0;-1.0 |] and
        csh = cos (pi /. 4.0) in
    let quadrant = if dir = -1.0 then quadrant
                   else 3 - quadrant in
    (Lit (csh *. cos_signs .(quadrant)),
     Lit (csh *. sin_signs .(quadrant)))
  else
    let theta = dir *.
                ((float_of_int (-2 * j)) *. pi) /.
                (float_of_int n) in
    (Lit (cos theta), Lit (sin theta))
```

We generate exact values where possible (e.g., $cos\frac{\pi}{2}$ is exactly zero). When computing $e^{\frac{\pi i}{4}}$ we ensure the real and the imaginary parts are identical. Due to the specifics of the library trigonometric functions, the computed value of $cos\frac{\pi}{4}$ is not identical to the computed value of $sin\frac{\pi}{4}$.

As a result of these successive refinements to the code, the function for merge is now defined as:

```
let merge_a dir (l1, l2) =
  let n = 2 * List.length l1 in
  let rec mg l1 l2 j =
    match (l1, l2) with
      (x::xs, y::ys) ->
        bind ((liftcM retN_va) x) (fun x ->
        bind ((liftcM retN_va) y) (fun y ->
        bind ((liftcM retN_va)
              (mul_ta (w_a dir n j, y)))
                              (fun z1 ->
        bind (retS (add_ta (x, z1))) (fun zx ->
        bind (retS (sub_ta (x, z1))) (fun zy ->
        bind ((mg xs ys (j+1))) (fun (a,b) ->
        retS (zx::a, zy::b)))))))
    | _ -> retS ([], []) in
    bind (mg l1 l2 0) (fun (a,b) -> retS (a @ b))
```

The stepwise refinement from the earlier `merge_mv` function only involves changing some of the operators to new operators viz., `retN_va` that abstracts away the sign of the factor while performing a name binding similar to that done by `retN_v` earlier, `mul_ta`, `add_ta`, and `sub_ta`, which perform optimized complex arithmetic operations, and `w_a` which generates exact float values where possible. The only change in the FFT function is to use `merge_a` instead of `merge_mv` to effect abstract interpretation during code generation.

4.6 Generated Code

Using the staged FFT function described above to generate code for the 4-point FFT yields the following code:

```
int __fun_def(double * x_382 ) {
  double y1_383 ; double y2_384 ; double y1_385 ;
  double y2_386 ; double y1_387 ; double y2_388 ;
  double y1_389 ; double y2_390 ; double y1_391 ;
  double y2_392 ; double y1_393 ; double y2_394 ;
  double y1_395 ; double y2_396 ; double y1_397 ;
  double y2_398 ;
  y1_383 = x_382[0]; y2_384 = x_382[1];
  y1_385 = x_382[4]; y2_386 = x_382[5];
  y1_387 = y1_383 + y1_385;
  y2_388 = y2_384 + y2_386;
  y1_389 = y1_383 - y1_385;
  y2_390 = y2_384 - y2_386;
  y1_391 = x_382[2]; y2_392 = x_382[3];
  y1_393 = x_382[6]; y2_394 = x_382[7];
  y1_395 = y1_391 + y1_393;
  y2_396 = y2_392 + y2_394;
  y1_397 = y1_391 - y1_393;
  y2_398 = y2_392 - y2_394;
  x_382[0] = y1_387 + y1_395;
  x_382[1] = y2_388 + y2_396;
  x_382[2] = y1_389 + y2_398;
  x_382[3] = y2_390 - y1_397;
  x_382[4] = y1_387 - y1_395;
  x_382[5] = y2_388 - y2_396;
  x_382[6] = y1_389 - y2_398;
  x_382[7] = y2_390 + y1_397;
  x_382; return 0;
}
```

The code contains fewer operations than were present before abstract interpretation, and all the problems that we pointed out are addressed. In fact, the code contains no floating-point multiplications at all. Inspecting the generated code for 8-point complex FFT shows that it uses only 4 floating-point multiplications and 52 floating-point additions and subtractions, which is exactly the same number of operations in the code generated by FFTW.

The table in Figure 1 summarizes our measurements of the effect of abstract interpretation for FFT. The first column gives the size of the FFT input vector. The second column gives the number of floating-point multiplications/additions (or subtractions) in the code resulting from direct staging. The third column shows the number of multiplications/additions in the code resulting from using our abstract interpretation techniques with staging. The last column shows the number of multiplications/additions in code generated by FFTW for the various problem sizes.

The table indicates that the generated FFT circuits are improved by abstract interpretation, and that abstract interpretation produces circuits with almost as few floating point operations as does FFTW.[1]

[1] The numbers for FFTW are obtained from its codelets. FFTW does not have codelets for sample sizes 128 and 256. These values are estimates based on the values for smaller sizes and on the general FFT algorithm.

Size	Direct staging	Abst. Interp.	FFTW[4] [8]
4	32/32	0/16	0/16
8	96/96	4/52	4/52
16	256/256	28/150	24/144
32	640/640	108/398	84/372
64	1536/1536	332/998	248/912
128	3584/3584	908/2406	752/2208
256	8192/8192	2316/5638	2016/5184

Figure 1: The number of floating-point multiplications/additions for FFT transforms for different sample sizes.

5. CONCLUSIONS

We have proposed a methodology for writing generators that can produce a family of efficient combinatorial circuits. By building on top of RAP languages, the programmer is guaranteed that any generated program would be well-typed and circuit-realizable. Because the generator is written in an expressive language, it is easier to ascertain the correctness of the generator and, in turn, the correctness of the full family of generated circuits. We have illustrated how using staged memoization and abstract interpretation makes it possible to refine the generators by a series of small modifications until they generate efficient circuits. A key feature of our methodology is that it avoids adding constructs for intensional analysis of the generated code. This ensures that the static well-typedness and resource boundedness guarantees of the RAP language are preserved. While the inability to traverse the generated code severely limits a posteriori optimizations, the running example of the FFT circuits shows that abstract interpretation provides promising alternative approach that allows us to keep the next-stage datatype abstract.

An immediate goal for future work is to see if all optimizations performed by FFTW can be achieved using abstract interpretation. In particular, the implementation presented here performs only obvious optimizations. More broadly, we are interested in developing monadic libraries to support the use of abstract interpretation in resource-aware programming.

6. ACKNOWLEDGMENTS

We would like to thank Jason Eckhardt, Stephan Ellner, and Roumen Kaiabachev for helpful comments.

7. REFERENCES

[1] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press (Cambridge, MA) and McGraw-Hill Book Company (New York, NY), 1996. Available online from `http://mitpress.mit.edu/sicp/full-text/book/book.html`.

[2] Per Bjesse, Koen Claessen, and Mary Sheeran. Lava: Hardware design in Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 174–184, 1998.

[3] W. Boehm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a single assignment programming language to reconfigurable systems. In *Supercomputing*, number 21, pages 117–130, 2002.

[4] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[6] A.H. DeHon. The density advantage of configurable computing. In *IEEE Computer*, pages 41–49, Apr 2000.

[7] Jason Eckhardt, Roumen Kaiabachev, Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. Monadic multi-stage programming. In preparation, July 2004.

[8] Matteo Frigo. A Fast Fourier Transform compiler. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 169–180, 1999.

[9] Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. In *Fifth International Workshop on Designing Correct Circuits*, March 2004.

[10] Martin Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

[11] C. S. Burrus I. W. Selesnick. Automatic generation of prime length FFT programs. In *IEEE Transactions on Signal Processing*, pages 14–24, Jan 1996.

[12] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in ruby. In C. C. Morgan R. S. Bird and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science. Springer Verlag, 1993.

[13] D. Ku and G.D. Micheli. HardwareC - a language for hardware design (version 2.0). Technical Report CSL-TR-90-419, Stanford University, April 1990.

[14] John Launchbury, Jeffrey R. Lewis, and Byron Cook. On embedding a microarchitectural design language within Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*, Paris, France, September 1999.

[15] Xavier Leroy. Objective Caml, 2000. Available from http://caml.inria.fr/ocaml/.

[16] R. Lipsett, E. Marschner, and M. Shaded. VHDL - The Language. In *IEEE Design and Test of Computers*, pages 28–41, April 1986.

[17] W.H. Mangione-Smith. Seeking solutions in configurable computing. In *IEEE Computer*, pages 38–43, Dec 1997.

[18] Andrew K. Martin. HML: A language for high-level design of high-frequency circuits. In *Fifth International Workshop on Designing Correct Circuits*, March 2004.

[19] Nicholas McKay and Satnam Singh. Dynamic specialisation of XC6200 FPGAs by partial evaluation. In *Field-Programmable Logic and Applications. From FPGAs to Computing Paradigm: 8th International Workshop, FPL '98*, Tallinn, Estonia, August 1998.

[20] MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from http://www.metaocaml.org/, 2003.

[21] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[22] Walid A. Najjar, Wim Boehm, Bruce A. Draper, Jeff Hammes, Robert Rinker, J. Ross Beveridge, Monica Chawathe, and Charles Ross. High-level language abstraction for reconfigurable computing. In *IEEE Computer*, pages 63–69, August 2003.

[23] John O'Donnell. Integrating formal methods with digital circuit design in Hydra. In *Fifth International Workshop on Designing Correct Circuits*, March 2004.

[24] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from ftp://cse.ogi.edu/pub/tech-reports/README.html.

[25] Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *the International Conference on Functional Programming (ICFP '02)*, Pittsburgh, USA, October 2002. ACM.

[26] R. Sharp and A. Mycroft. A higher-level language for hardware synthesis. In *Proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 2001.

[27] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [24].

[28] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Maniplation (PEPM)*, Boston, 2000. ACM Press.

[29] Walid Taha. A gentle introduction to multi-stage programming. In Don Batory, Charles Consel, Christian Lengauer, and Martin Odersky, editors, *Domain-specific Program Generation*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[30] Walid Taha, Stephan Ellner, and Hongwei Xi. Generating Imperative, Heap-Bounded Programs in a Functional Setting. In *Proceedings of the Third International Conference on Embedded Software*, Philadelphia, PA, October 2003.

[31] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 2003.

[32] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 3rd edition, 1996.

[33] Philip Wadler. Comprehending monads. In *Proceedings of Symposium on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990. ACM.

APPENDIX

A. THE STATE-CONTINUATION MONAD AND THE RETN OPERATOR

We write the fft function in a monadic style so as to allow us to generate let-bindings for code fragments corresponding to subcomputations. By referencing let-bound variables in the code corresponding to later computations, we avoid duplicating code fragments. This technique, called monadic sharing [7] requires using a monad that takes a state (or, a memoization table) s, and a continuation k, that represents the remaining computation. Having the monad use the state s allows us to keep track of which subcomputations have already been let-bound, while the continuation k in the monad allows the code that is not yet generated to refer to the let-bound variable rather than copy code for the entire subcomputation. The monadic ret and bind operations are:

```
type (α, σ, κ) m = σ  -> (σ -> α -> κ) -> κ
let (ret,bind) =
   let ret a = fun s k -> k s a in
   let bind a f = fun s k
            -> a s (fun s′ b -> f b s′ k)
   in (ret,bind)
```

The return of this monad takes a store s and a continuation k, and passes both the state and a to the continuation. The bind of this monad passes to the monadic value a a store s and a new continuation. The new continuation first evaluates the function f using the new store s′ and continues with k.

Avoiding the duplication of code for common subexpressions is a similar problem, and we solve it in the monadic setting by using a nonstandard monadic operator retN that generates let-bindings for the common subexpressions, and suitably modifies the remaining computation to use these let-bindings. This operator is written as follows:

```
  let retN a = fun s k ->
     .<let z = .~a in .~(k s .<z>.)>.
```

and binds its argument a to a new name z. This new name is then passed to the continuation k, so that z appears in place of a in all the remaining code.

For the application-specific domain maybeValue used in this paper, we use retN to define another operator that generates bindings only for nontrivial expressions:

```
let retN_v = function
   Val _ as v -> retS v
 | Exp _ as x -> bind (retN (mVconc x))
                      (fun x -> retS (Val x))
```

To allow this operator to work on complex values (tuples with real and imaginary parts), we define liftcM to lift retN_v to work on tuples:

```
let liftcM op (x,y) =
  bind (op x) (fun nx ->
  bind (op y) (fun ny ->
  retS (nx,ny)))
```