# A Domain-Specific Language for Software-Defined Radio

Geoffrey Mainland(✉)

Department of Computer Science, Drexel University, Philadelphia, PA, USA
mainland@drexel.edu

**Abstract.** Software-defined radio (SDR) is a demanding domain; real-world wireless protocols require high data rates and low latency. Existing SDR platforms, typically based on FPGAs, provide the necessary substrate for meeting these requirements, but the high-level tools available to program them are not capable of fully exploiting the underlying hardware to meet rigorous performance requirements. Ziria [11] demonstrated that a high-level language can compete in this demanding space, but its design was ad-hoc and overly influenced by the needs of the compiler writer since its surface language does double duty as the compiler's intermediate language.

We present a re-formulation of Ziria's surface language that includes a new type system that allows this language, which is effectful, to elaborate into a pure, monadic language where effects such as input/output and reference manipulation can be distinguished purely by type. This re-formulation and its elaboration into a core language is embodied in a new compiler for Ziria, `kzc`. By choosing an appropriate type system, awkward syntactic distinctions currently made by Ziria can be eliminated, although our new implementation maintains source compatibility with the original compiler due to a large body of existing Ziria code (a full 802.11 physical layer implementation). Our contribution is a description of the surface language, its type system, and its elaboration into a core language. We also show that far from being limited to the SDR domain, the constructs built-in to Ziria are applicable to other resource-constrained domains that require high-speed data processing.

## 1 Introduction

Software-defined radio promises to bring the productivity benefits of software—fast development cycles and modular reuse of code—to the world of radio protocols. Radio platforms for SDR, such as USRP [3] and BladeRF [2], provide the necessary hardware for high-performance radio protocol implementations, but existing tools for programming these devices fall short on one or more dimensions. The fundamental issue is the tension between ease of programming and performance.

Although most SDR hardware incorporates an FPGA, which could be programmed directly, doing so requires not only the use of proprietary tools, but also fairly low-level knowledge of the underlying FPGA. Instead, platforms like GNU


Y. Lierler and W. Taha (Eds.): PADL 2017, LNCS 10137, pp. 173–188, 2017.
DOI: 10.1007/978-3-319-51676-9_12

Radio [4] offer a high-level toolkit of signal processing blocks written in Python and C++. These blocks are composed using a graph-based model where vertices, i.e., blocks, represent computation, and edges represent communication. While simple to program, this model does not result in high-speed, low-latency protocol implementations. As a programming model, the graph-based paradigm also has a number of shortcomings. First, it does not specify when a vertex's state is initialized. Although edges represent "communication," how control messages and data flow are differentiated is not well-defined, and it is unclear how one vertex could send a control message to another vertex, perhaps one to which it is not directly connected. There is also no well-defined method for control messages to reconfigure data flow in the graph. Finally, since each vertex is a black box, there is no opportunity to jointly optimize multiple vertices' operations.

SORA [14] was the first SDR programming platform to provide a purely software-based 802.11 a/b/g implementation that operated at speeds comparable to commodity 802.11 hardware. This was achieved with a carefully hand-tuned C++ implementation. The SORA implementation is so carefully tuned, that modifying it while maintaining performance is very difficult. For example, SORA relies crucially and frequently on lookup tables (LUTs) for performance, but these LUTs appear simply as array constants in the C source. Questions such as how these LUTs were generated (by hand?), how one should choose when to write a function as a LUT, and how one might go about changing an existing LUT are left unanswered.

Implementing radio protocols directly in FPGA hardware is typically accomplished using MATLAB/Simulink; both WARP [9] and SOFDM [5] take this approach. The resulting programs are graph-based system models that are synthesized to FPGA bitstreams. However, though they can fully exploit the underlying FPGA, these models are large, difficult to construct and reason about, and they are intimately tied to particular platform traits such as the FPGA clock rate and the bit width of the A/D converter in the radio front end. Furthermore, the MATLAB/Simulink environment does not offer constructs tailored to the SDR domain.

Ziria [11] is a high-level language for wireless physical layer (PHY) protocols, i.e., the portion of the radio stack that converts radio signals into bits, which are then passed on to another protocol handler, such as a MAC protocol. Ziria provides *both* programmability and SORA-level performance. This is achieved by a number of compiler optimizations—in effect, instead of an expert C programmer doing the work of translating a high-level specification of a wireless PHY protocol into efficient low-level C code, the Ziria compiler does the work. The optimizer's job is made easier by the nature of the restricted application domain: whole-program analysis is possible, arrays sizes are statically known, and communication between components is performed via built-in language constructs.

In this work, we show how to reformulate the Ziria surface language so that terms' types differentiate between the three effects that are meaningful in our setting: memory assignment/dereference, reading from a queue, and writing to a queue. The existing surface language uses syntactic constructs to distinguish

between code that performs memory assignment/dereferencing and code that performs IO via queues; our reformulation shows how this syntactic distinction can be eliminated in favor of a type-based distinction. This reformulation also enables elaboration of the effectful surface language into a pure, monadic core language. In other work [8], we show that novel source-to-source transformations on this core language can jointly optimize across multiple Ziria components, fusing them into a single loop; we include benchmarks demonstrating the performance effects of these optimizations in Sect. 6. Concretely, the contributions of this work are as follows:

- A type system, with a limited form of quantification, that expresses what are currently syntactic distinctions in Ziria as type distinctions.
- A method for elaborating the effectful Ziria surface language into a pure, monadic core language.
- A new continuation-based compilation model for Ziria.

Our contributions are embodied in kzc[1], a wholly new open source compiler for the Ziria language that is source-compatible with the existing compiler, wplc[2].

## 2 Background

We first give a brief overview of the Ziria surface language to provide necessary context. The surface language we describe is identical to the language described by Stewart et al. [11], and this section does not represent novel work. Ziria provides an imperative core wrapped with combinators for producer-consumer computations that operate over streams of data. We illustrate both components of the language in Listing 2.1, which is the Ziria implementation of the 802.11 scrambler [1, Sect. 16.2.4]. The purpose of the scrambler is to transform the transmitted bit stream so that it does not contain long runs of ones or zeros, either of which would make detection at the receiver more difficult.

Line 2 allocates mutable storage for the scrambler's state; this state is initialized with the (immutable) value of the parameter `init_scrmbl_st`. Both `init_scrmbl_st` and `scrmbl_st` are arrays of seven bits. After initializing the scrambler state, the scambler enters a **`repeat`** loop in line 4 that continually reads a bit from its input data stream using **`take`**, updates the scrambler state, transforms the consumed bit using the scrambler state, and finally outputs the transformed bit in line 14 using **`emit`**. Although not shown in this snippet, Ziria also allows immutable values to be bound with **`let`** (instead of **`var`**). The surface language does not include an explicit dereference construct, instead making dereferencing implicit, as shown on line 9.

The syntactic distinction between ref manipulation and input/output is made using **`do`** and **`seq`** blocks; a **`seq`** block sequences IO, whereas a **`do`** block sequences ref manipulation. The **`repeat`** language construct takes an IO action

[1] The Kyllini Ziria compiler. Ziria is another name for Mount Kyllini in Greece.
[2] The wireless programming language compiler.

```
1  fun comp scrambler(init_scrmbl_st: arr[7] bit) {
2    var scrmbl_st : arr[7] bit := init_scrmbl_st;
3
4    repeat seq {
5      x ← take;
6
7      var tmp : bit;
8      do {
9        tmp := (scrmbl_st[3] ^ scrmbl_st[0]);
10       scrmbl_st[0:5] := scrmbl_st[1:6];
11       scrmbl_st[6] := tmp;
12     };
13
14     emit (x^tmp)
15   }
16 }
```

**Listing 2.1. Ziria implementation of 802.11 scrambler.**

```
scrambler('1011101)
⋙
seq {
  var buf : arr[8] bit;
  for i in [0, 8] { x ← take; do { buf[i] := x;} };
  emits buf;
}
```

**Listing 2.2. Composition along the data path.**

and repeats it forever. The resulting computation is termed a *stream transformer* because it continually reads input, transforms it, and writes the transformed value to its output. Both **do** and **seq** blocks compose computations along the *control* path, and the syntax for this sort of composition is deliberately reminiscent of Haskell's `do` syntax, as seen in line 5.

## 2.1 Composition Along the Data Path

Given the scrambler, which ensures the bits we are transmitting will be sufficiently varied between 1 and 0, we need a way to compose it with other data producer/consumer components. Instead of composition along the control path, we want to compose the scrambler *along the data path*, which is accomplished using the *par* operator, ⋙.

Listing 2.2 shows an example of composition along both the control and data paths using the previously defined `scrambler` function. The first component in the data path is the `scrambler` computation. Note that producer-consumer computations are higher-order; the argument to `scrambler` here is a bit array

constant of length 7, which serves to initialize the scrambler's state. The second element in the data path reads 8 elements from its input, collecting them in a buffer, and then outputs them all at once using **emits**. The only difference between **emit** and **emits** is that the latter acts as though each element of its array argument were emitted one-by-one. Because the second element in the data path terminates, it is a *stream computer*. The distinction between a *stream transformer* and a *stream computer* is apparent from their types, the topic to which we now turn.

## 3 Typing Ziria Programs

The first contribution of this paper is a new type system for the Ziria surface language that makes a distinction between three effects: ref manipulation (assignment and dereferencing), reading (using **take**), and writing (using **emit**). The kzc compiler performs type inference using this type system, elaborating source language terms to a core language we describe in Sect. 4. We first informally sketch our types system.

Like Stewart et al. [11], we make use of an indexed type reminiscent of both monads and arrows [6], but we use a limited form of quantification to distinguish between effects. For example, we assign the scrambler in Listing 2.1 the moral type arr[7] bit → ST T bit bit. The type to the left of the arrow is the argument to `scrambler`. The type ST is indexed by three types: T, which indicates that this term is a *stream transformer*, and the two types, both bit, specifying the input and output types, respectively, of the computation. The second half of the *par* in listing 2.2 is instead assigned the type ST (C ()) bit (arr[8] bit). Because this computation terminates with the unit value, i.e., it is a *stream computer*, the first index to ST is now C (). The computation reads values of type bit and writes values of type arr[8] bit, so those types make up the final two indices.

The question remains: how do we differentiate between effects using types? For pure expressions, the answer is simple: pure expressions have a non-ST type. Expressions that manipulate references but *do not* perform IO could be assigned a type that quantifies over the input and output types of the data stream. For example, the expression `x := 1;` could be typed as $\forall\ \alpha\ \beta.$ ST (C ()) $\alpha\ \beta$. Similarly, computations that only read or write could be typed by quantifying over the appropriate index to ST.

Unfortunately, the simple quantification strategy does not allow us to properly differentiate between terms that perform IO using take and emit and those that do not. Consider the following example:

```
seq { x ← take; return 1; }
```

What type should we assign this term? Its type must certainly have the form ST (C int) $\alpha\ \beta$ for some $\alpha$ and $\beta$. It is also clear that we need to quantify over $\beta$ because the expression does not write to the data stream. However, although it does read from its input stream, the term is *agnostic to the type* of the data it reads, so it seems reasonable to quantify over *both* $\alpha$ and $\beta$. We conclude that

this expression should have type $\forall\ \alpha\ \beta.\ \mathsf{ST}\ (\mathsf{C}\ \mathsf{int})\ \alpha\ \beta$. Similar reasoning leads us to assign the same type to this term, which *does not* perform any input or output:

```
seq { return 1; }
```

The root of the problem is that our quantification scheme does not allow us to differentiate between terms that are polymorphic in the value read from the data stream and terms that do not read from the data stream at all.

Our solution will be to add a fourth index to the ST type—but what should this index be? Since we want to know whether or not a term reads a value from its input stream, we could make the index a type-level Boolean. We could also add an additional type-level construct analogous to the $\mathsf{C}\ \alpha/\mathsf{T}$ construct we use to differentiate between transformers and computers, but this makes the type system more complicated. Instead of adding something new, we will reuse existing type system mechanisms—in particular, unification. Our new type index will be left free until a read occurs, at which point it will be unified with the type index that specifies the type of the input stream. Therefore, when these two indices are equal, we know a read has occurred, and if they are not equal, we know that a read *has not* occurred.

### 3.1 A Type System for Differentiating Effects

Figure 1 shows the language of types for Ziria terms. We do not include array types here as they clutter the presentation, and adding them is not difficult. Base types, $\tau$, are as one would expect. Types in ST allow quantification over base types in the indices of ST. The first index, $\omega$, specifies whether this computation is a stream transformer (T) or a stream computer ($\mathsf{C}\ \tau$). We will shortly see the details of how the other three indices are used to indicate read/write behavior. For completeness, we include the details of reference handling. Note that types are stratified so that although references can always be passed to a function, they can never be returned from a function, i.e., only "downward reference funargs" are allowed. This ensures that a reference can never escape its defining scope, eliminating the need for garbage collection. This reduction in expressivity is perfectly acceptable in our domain.

$$
\begin{array}{rcl}
\nu, \tau & ::= & \alpha, \beta, \gamma \\
& \mid & () \\
& \mid & \mathsf{bool} \\
& \mid & \mathsf{int} \\
\omega & ::= & \mathsf{C}\ \tau \\
& \mid & \mathsf{T} \\
\mu & ::= & \forall\overline{\alpha}.\mathsf{ST}\ \omega\ \tau\ \tau\ \tau \\
\rho & ::= & \tau \\
& \mid & \mathsf{ref}\ \tau \\
\sigma & ::= & \tau \\
& \mid & \mu \\
\phi & ::= & \rho \\
& \mid & \rho_1 \ldots \rho_n \to \sigma \\
\Gamma & ::= & \cdot \\
& \mid & x : \theta, \Gamma \\
\theta & ::= & \tau \\
& \mid & \mathsf{ref}\ \tau \\
& \mid & \mu \\
& \mid & \rho_1 \ldots \rho_n \to \sigma
\end{array}
$$

**Fig. 1.** Ziria type language.

The declarative formulation of the Ziria typing relation is shown in Fig. 2. We include the T-DEREF rule even though, as stated earlier, dereferences are *implicit* in the surface language. We return to this point in Sect. 4. Rule T-TAKE forces the second and third index of the ST type to both be $\alpha$, although it still quantifies over $\alpha$. This type reflects the fact

$\boxed{\Gamma \vdash e : \theta}$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad x : \mathsf{ref}\ \tau_1, \Gamma \vdash e_2 : \mathsf{ST}\ \omega\ \alpha\ \beta\ \gamma}{\Gamma \vdash \mathsf{letref}\ x = e_1\ \mathsf{in}\ e_2 : \mathsf{ST}\ \omega\ \alpha\ \beta\ \gamma}\ (\text{T-LetRef})$$

$$\frac{x : \mathsf{ref}\ \tau \in \Gamma}{\Gamma \vdash !x : \forall \alpha\ \beta\ \gamma.\ \mathsf{ST}\ (\mathsf{C}\ \tau)\ \alpha\ \beta\ \gamma}\ (\text{T-Deref})$$

$$\frac{x : \mathsf{ref}\ \tau \in \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \forall \alpha\ \beta\ \gamma.\ \mathsf{ST}\ (\mathsf{C}\ ())\ \alpha\ \beta\ \gamma}\ (\text{T-Assign})$$

$$\frac{\Gamma \vdash c : \tau}{\Gamma \vdash \mathsf{return}\ c : \forall \alpha\ \beta\ \gamma.\ \mathsf{ST}\ (\mathsf{C}\ \tau)\ \alpha\ \beta\ \gamma}\ (\text{T-Return})$$

$$\frac{\Gamma \vdash c_1 : \mathsf{ST}\ (\mathsf{C}\ \nu)\ \alpha\ \beta\ \gamma \quad x : \nu, \Gamma \vdash c_2 : \mathsf{ST}\ \omega\ \alpha\ \beta\ \gamma}{\Gamma \vdash x \leftarrow c_1;\ c_2 : \mathsf{ST}\ \omega\ \alpha\ \beta\ \gamma}\ (\text{T-Bind})$$

$$\frac{\Gamma \vdash c_1 : \mathsf{ST}\ (\mathsf{C}\ \nu)\ \alpha\ \beta\ \gamma \quad \Gamma \vdash c_2 : \mathsf{ST}\ \omega\ \alpha\ \beta\ \gamma}{\Gamma \vdash c_1;\ c_2 : \mathsf{ST}\ \omega\ \alpha\ \beta\ \gamma}\ (\text{T-Seq})$$

$$\Gamma \vdash \mathsf{take} : \forall \alpha\ \beta.\ \mathsf{ST}\ (\mathsf{C}\ \alpha)\ \alpha\ \alpha\ \beta\ (\text{T-Take})$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{emit}\ e : \forall \alpha\ \beta.\ \mathsf{ST}\ (\mathsf{C}\ ())\ \alpha\ \beta\ \tau}\ (\text{T-Emit})$$

$$\frac{\Gamma \vdash c : \mathsf{ST}\ (\mathsf{C}\ ())\ \alpha\ \beta\ \gamma}{\Gamma \vdash \mathsf{repeat}\ c : \mathsf{ST}\ \mathsf{T}\ \alpha\ \beta\ \gamma}\ (\text{T-Repeat})$$

$$\frac{\Gamma_1 \oplus \Gamma_2 = \Gamma \quad \Gamma_1 \vdash c_1 : \mathsf{ST}\ \omega_1\ \alpha\ \alpha\ \beta \quad \Gamma_2 \vdash c_2 : \mathsf{ST}\ \omega_2\ \beta\ \beta\ \gamma}{\Gamma \vdash c_1 \ggg c_2 : \mathsf{ST}\ (\omega_1 \sqcup \omega_2)\ \alpha\ \alpha\ \gamma}\ (\text{T-Par})$$

$$\mathsf{C}\ \alpha \sqcup \mathsf{T} = \mathsf{C}\ \alpha \qquad \mathsf{T} \sqcup \mathsf{C}\ \alpha = \mathsf{C}\ \alpha \qquad \mathsf{T} \sqcup \mathsf{T} = \mathsf{T}$$

$$\frac{\Gamma \vdash c : \forall \overline{\alpha}.\ \mathsf{ST}\ \omega\ \tau_1\ \tau_2\ \tau_3}{\Gamma \vdash c : [\overline{\alpha \mapsto \tau}]\ \mathsf{ST}\ \omega\ \tau_1\ \tau_2\ \tau_3}\ (\text{T-Inst})$$

$$\frac{\overline{\alpha} \notin fvs(\Gamma) \quad \Gamma \vdash c : \mathsf{ST}\ \omega\ \tau_1\ \tau_2\ \tau_3}{\Gamma \vdash c : \forall \overline{\alpha}.\ \mathsf{ST}\ \omega\ \tau_1\ \tau_2\ \tau_3}\ (\text{T-Gen})$$

**Fig. 2.** Declarative typing relation for Ziria.

that we are reading from the data stream, although we are polymorphic in the value being read. During type inference, use of take is what causes unification of the two type indices as mentioned above. Rule T-Emit says that **emit** is polymorphic in the input type of the data stream, but it constrains the fourth index of the ST type (the *data stream output type* index) to be $\tau$, the type of the value being emitted. Table 1 maps types to their conceptual meanings, showing how types differentiate between effects. The essential idea is that a term with an ST type in which the second and third indices (the *data stream input type* indices) are identical reads from its input data stream, even if it is polymorphic in the type that is read. If the second and third indices *differ*, then the term does not read from its input stream. As a final example, the following identify transformer has the type $\forall \alpha.\ \mathsf{ST}\ \alpha\ \alpha\ \alpha$:

**Table 1.** Conceptual meaning of quantification in ST types.

| Type | Conceptual meaning |
|---|---|
| $\forall\alpha\ \beta\ \gamma.\ \mathsf{ST}\ \omega\ \alpha\ \beta\ \gamma$ | A computation that may assign or dereference memory but does not perform IO |
| $\forall\alpha\ \gamma.\ \mathsf{ST}\ \omega\ \alpha\ \alpha\ \gamma$ | A computation that reads one or more values from the data stream but does nothing with the read value(s) |
| $\forall\gamma.\ \mathsf{ST}\ \omega\ \tau\ \tau\ \gamma$ | A computation that reads one or more values of type $\tau$ from the data stream |
| $\mathsf{ST}\ \omega\ \tau_1\ \tau_1\ \tau_2$ | A computation that reads one or more values of type $\tau_1$ from the input data stream and writes one or more values of type $\tau_2$ to the output data stream |

```
repeat seq { x ← take; emit x; }
```

In implementing the kzc compiler, we certainly wanted to differentiate between pure and impure code for purposes of optimization; that is easily done via the ST type. However, we also want to differentiate between impure code that uses memory references and code that may perform IO. The new type system in Fig. 2 allows for this. In the original incarnation of Ziria, this distinction was made syntactically via seq and do, and programmers had to manually "lift" code that used references into the ST monad. With the new type system, it is now possible to eliminate the do/seq distinction from the surface language; we plan to add a new alternative syntax that does this, but for compatibility reasons we have not yet done so.

### 3.2 Typing Composition Along the Control Path

The rules T-Bind and T-Seq support composition along the control path. The only notable aspect of these rules is the way the first index of the ST type assigned to the overall term relates to the first index of the ST type of each subterm. The first subterm being sequenced must be a computer, i.e., it must compute a value and terminate, so the its $\omega$ index must be $\mathsf{C}\ \nu$. The second subterm *may or may not* terminate. That is, it may be a transformer, so its $\omega$ index is unconstrained. The overall term then has an $\omega$ index matching that of the second subterm being sequenced. Note that we could remove the T-Seq rule and treat sequencing as syntactic sugar for bind.

### 3.3 Typing Composition Along the Data Path

Typing composition along the data path is done by the rule T-Par. Unlike the rules for composition along the control path, the subterms $c_1$ and $c_2$ of T-Par have types whose $\tau$ indices (the third through fourth indices in the ST type) that may differ between the two subterms' types. Since $\ggg$ represents composition

along the data path, the terms' types are instead constrained so that the data stream output index of the type of $c_1$ matches the data stream input type indices of the type of $c_2$.

The T-Par rule uses of the join operator, $\cdot \sqcup \cdot$, to determine the $\omega$ type index of the result of the par. This operator guarantees that two stream transformers may be composed on the data path, as may a stream computer and a stream transformer, but it prevents two stream computers from being composed along the data path. We could imagine adding a fourth case to the join operator, $\mathsf{C}\ \alpha \sqcup \mathsf{C}\ \alpha = \mathsf{C}\ \alpha$, but this complicates the semantics as it requires additional synchronization on the final computed result. This change would also complicate the implementation; with the current semantics, we are guaranteed that at most one side of the par will ever terminate and call the par's continuation.

The final subtlety in T-Par is the context splitting operation, $\oplus$. The context splitting operation $\oplus$ splits the portion of the context that contains variables that have type $\mathsf{ref}\ \tau$, leaving the rest of the context as-is. This ensures that the type environments for the two subterms, $\Gamma_1$ and $\Gamma_2$, contain completely distinct sets of references, thus preventing race conditions. An additional check on function calls ensures aliasing does not occur, $\oplus$ can perform a purely syntactic check on $\Gamma$; see Mainland [8] for details.

### 3.4 Type Inference in Practice

The described typing relation is declarative. When, then, do we apply rules T-Gen and T-Inst? Similar to standard syntax-directed systems based on Hindley-Milner, we instantiate types immediately and generalize at "let"; for example, when inferring the type of a function body, we immediately instantiate any occurrence of **`take`** or **`emit`**, and we then generalize once we have inferred the type of the entire function body. Inference makes use of the standard unification algorithm. We plan to formalize the inference algorithm, but on its own it is standard—the novel aspect of inference is the use of the indexed type $\mathsf{ST}$ to differentiate between various effects and the process of elaborating to the core language, which we describe in the next section.

### 3.5 Types for Streaming Combinators

The type system we have presented supports a general form of stream combinator and is not specific to the SDR domain or the Ziria language. The technique we use to reflect read operations in the $\mathsf{ST}$ type by forcing unification of two type indices is even more general. In effect, we are differentiating between two kinds of polymorphism: polymorphism that arises because read values are used polymorphically, as in the identity function, and polymorphism that arises because values aren't read at all. Because we are simply forcing type equality—in our case, via the typing rule for $\mathsf{take}$—we minimize the number of extra features that need to be added to the type system. We expect these techniques to be transferable to any domain where typed streaming combinators are useful.

## 4 Elaborating to Core

The kzc compiler performs type inference on the Ziria surface language and elaborates it to the core language given in Fig. 3. Unlike the surface language, the core language contains only a single syntactic category: expressions. There is no need for a syntactic distinction between pure terms, terms that use memory references, and terms that perform IO, because the type system described in Sect. 3 provides the needed distinctions. Also unlike the surface language, the core language makes memory dereferencing explicit. Explicit memory reference operations in the core language make some analyses in the compiler easier to perform; for example, it allows the compiler to determine that an expression is pure merely based on its type. However, forcing the programmer to use explicit dereferencing in the surface language seems overly burdensome; despite our use of monadic bind, we want the surface language to be as close to typical "curly brace and semicolon" imperative code as possible while still being fundamentally functional.

Elaboration makes use of a new form of judgment:

$$\mathscr{F}; \Gamma \vdash^{val} e : \tau \rightsquigarrow \mathscr{F}'; e'$$

Like the typing judgment, the elaboration judgment assigns a type $\tau$ to a term $e$. However, it also elaborates a surface language term $e$ to a core term $e'$. Recall from Sect. 3.1 that references are not first-class in Ziria—they can never be returned from a function or otherwise escape the scope of their originating binder. This judgment form is a *value elaboration*; it elaborates a Ziria term, which may contain implicit dereferences, into a core term in which all dereferences are made explicit. The extra component $\mathscr{F}$ that is threaded through the elaboration judgment is the elaborated term's *value context*; it is a function from core terms to core terms that transforms an elaborated term so that all implicit dereferences are bound.

$$
\begin{array}{lll}
e, c ::= & k & \text{(constant)} \\
\mid & x & \text{(variable)} \\
\mid & \mathit{unop}\ e & \\
\mid & e_1\ \mathit{binop}\ e_2 & \\
\mid & \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & \\
\mid & \mathsf{let}\ x : \tau = e_1\ \mathsf{in}\ e_2 & \\
\mid & \mathsf{letfun}\ f(\overline{x_i : \rho_i}) : \sigma = e_1\ \mathsf{in}\ e_2 & \\
\mid & \mathsf{letref}\ x : \mathsf{ref}\ \tau = e_1\ \mathsf{in}\ e_2 & \\
\mid & f\ e_1\ \ldots\ e_n & \\
\mid & !x & \\
\mid & x := e & \\
\mid & \mathsf{return}\ e & \\
\mid & (x : \tau) \leftarrow c\ ;\ c & \text{(bind)} \\
\mid & c\ ;\ c & \text{(sequence)} \\
\mid & \mathsf{take} & \\
\mid & \mathsf{emit}\ e & \\
\mid & \mathsf{repeat}\ c & \\
\mid & c_1 \ggg c_2 & \text{(par)}
\end{array}
$$

**Fig. 3.** The expression core language.

The intuition behind the function $\mathscr{F}$ is that it will insert the necessary bindings around an elaborated term to ensure that dereferenced values are properly bound. For example, if we have a surface language term $x + y$ where $x$ and $y$ are references of type ref int, it will be elaborated to a term $x' + y'$, where $x'$ and $y'$ are fresh variables, along with a value context:

$$\lambda e.(x' : \mathsf{int}) \leftarrow !x\ ; (y' : \mathsf{int}) \leftarrow !y\ ; e$$

The value context will continue to accumulate bindings until it is applied. Figure 4 shows a fragment of the elaboration rules; we do not include the full

set of rules due to space constraints. Note that in rule V-If, the subterms are all elaborated with empty value contexts, i.e., the identity function, and the resulting value contexts are applied immediately to the subterms. This ensures, for example, that dereferences required for the then branch are performed only within the then branch. The $\mathbf{binop}_{\tau_1,\tau_2}$ meta-function maps a surface language binary operator *binop* whose arguments have types $\tau_1$ and $\tau_2$ to the type of the operator's result; this allows us to, for example, overload + at multiple numeric types.

$$\boxed{\mathscr{F}; \Gamma \vdash^{val} e : \tau \rightsquigarrow \mathscr{F}'; e'}$$

$$\frac{\Gamma \vdash v : \mathsf{ref}\ \tau \qquad v' \text{ fresh}}{\mathscr{F}; \Gamma \vdash^{val} v : \tau \rightsquigarrow \mathscr{F} \circ \lambda e.\, \boxed{(v' : \tau) \leftarrow !v\ ;\ e}\,; v'} \quad \text{(V-Var)}$$

$$\frac{\mathscr{F}; \Gamma \vdash^{val} e_1 : \tau_1 \rightsquigarrow \mathscr{F}'; e_1' \qquad \mathscr{F}'; \Gamma \vdash^{val} e_2 : \tau_2 \rightsquigarrow \mathscr{F}''; e_2' \qquad \mathbf{binop}_{\tau_1,\tau_2}(binop) \text{ defined}}{\mathscr{F}; \Gamma \vdash^{val} e_1\ binop\ e_2 : \mathbf{binop}_{\tau_1,\tau_2}(binop) \rightsquigarrow \mathscr{F}''; e_1'\ [\![binop]\!]\ e_2'} \quad \text{(V-Binop)}$$

$$\frac{\begin{array}{c}\mathbf{id}; \Gamma \vdash^{val} e_1 : \mathsf{bool} \rightsquigarrow \mathscr{F}_1; e_1' \\ \mathbf{id}; \Gamma \vdash^{val} e_2 : \tau \rightsquigarrow \mathscr{F}_2; e_2' \qquad \mathbf{id}; \Gamma \vdash^{val} e_3 : \tau \rightsquigarrow \mathscr{F}_3; e_3'\end{array}}{\mathscr{F}; \Gamma \vdash^{val} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \rightsquigarrow \mathscr{F}; \mathscr{F}_1\ (\mathsf{if}\ e_1'\ \mathsf{then}\ (\mathscr{F}_2 e_2')\ \mathsf{else}\ (\mathscr{F}_3 e_3'))} \quad \text{(V-If)}$$

**Fig. 4.** Value elaboration relation for Ziria.

The process of maintaining a value context and elaborating to a pure, monadic language allows us to provide an impure surface language to the user, who does not have to worry about manually sequencing dereferencing, while reaping the benefits of a pure, monadic core language within the compiler.

## 5 Compilation Model

Stewart et al. [11] describe a `tick-proc` compilation model for compiling Ziria terms to C. In this model, each Ziria computation compiles to two blocks of code: a `tick` block that determines whether the computation needs to consume from the data stream to proceed, in which case it jumps to the upstream computation, or if it has data to emit, in which case it jumps to the downstream component. If the computation can proceed without IO, the `proc` block of code is executed. This compilation model results in overhead for every sequenced computation, since each sequenced computation requires both a `tick` and a `proc` block even if the computation itself does not perform IO.

Our compilation model is based on the observation that the only time one computation needs to "jump" to another computation is inside a *par* construct, $c_1 \ggg c_2$, when $c_2$ is executing and needs to read from upstream, or when $c_1$ is executing and needs to write downstream. Conceptually, we track the current continuation of both $c_1$ and $c_2$. When we are executing $c_2$ and encounter a **`take`**, we save the current continuation and jump to $c_1$'s saved continuation. When we then encounter an **`emit`** in $c_1$, we save its current continuation, save a pointer to the emitted value, and jump to $c_2$'s current continuation with the pointer as an argument. This gives rise to a coroutine-style execution model.

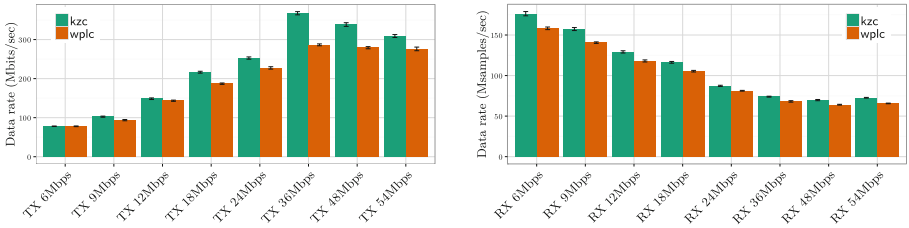
Since our compiler is a whole-program compiler, we can map this execution model to C code by using either GCC-style first-class labels, which are available in clang, gcc, and Intel's icc, or we can use a single `switch` statement to trampoline between continuations. Like the original Ziria compiler, for single-threaded Ziria code we completely avoid queues by storing a pointer to emitted values instead of queueing the values. Unlike the original Ziria compiler, we can also avoid copying emitted values in most cases using a data flow analysis that makes use of the fact that dereferences are explicit in our core language [8]. Our new compilation model imposes zero overhead for sequencing computation that do not perform IO.

## 6 Evaluation

The type system described in Sect. 3, elaboration to the core language described in Sect. 4, and compilation model described in Sect. 5 are all implemented in the kzc compiler. The existing Ziria WiFi implementation can be compiled with kzc, which also passes the extensive Ziria test suite. In this section we provide a performance evaluation to demonstrate that kzc works and that the new implementation strategies it uses do not impose additional overhead—in fact, kzc produces better code than the existing Ziria compiler, wplc. The performance results we provide are fully described by Mainland [8]; we do not claim the demonstrated performance improvements as contributions in this paper. All data was collected on an i7–4770 CPU running at 3.40 GHz under Ubuntu 16.04, generated C code was compiled with GCC 5.4[3], all runs were repeated 100 times, and we assume a normal distribution. All Ziria programs evaluated in this section are taken from the publicly available Ziria release [12].

The transmitter and receiver performance of kzc and wplc are shown in Figs. 5a and b. The ratios of the data rates of the two implementations are given in Fig. 6a. Code compiled by kzc is always as fast as code compiled by wplc, and in most cases it is at least 10% faster. The relative performance of individual pipeline blocks is broken out in Fig. 6b. We use the same runtime primitives as wplc, so the performance differences between the two implementations can be attributed directly to the differences in their compilation models. Our original expectation was that there was limited room for improvement in the transmitter and receiver pipelines because they use primitive blocks like FFT, IFFT, and

[3] `-march=native -mtune=native -Ofast`.




**(a)** Transmitter performance **(b)** Receiver performance

**Fig. 5. Transmitter and receiver data rates.** The receiver consumes a quadrature signal consisting of pairs of 16-bit numbers representing IQ samples. The transmitter consumes bits. Error bars show one standard deviation above and below the mean.

Viterbi, which tend to be the bottlenecks. However, we are pleased that we were nonetheless able to gain a 10% performance increase over an already highly-optimizing compiler.

## 7 Related Work

### 7.1 SDR

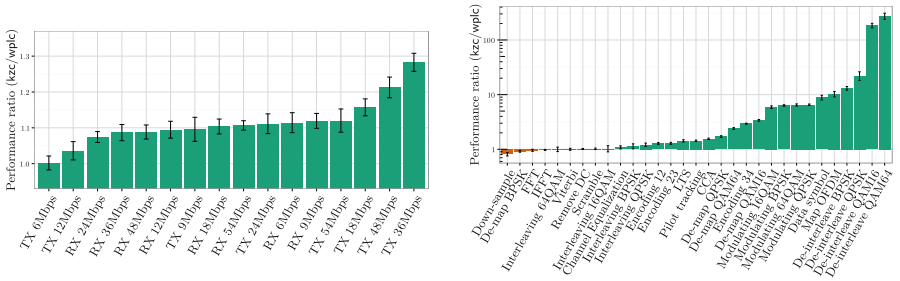
Our work is directly based on the original Ziria compiler [11]. Although we do not reuse any code from the Ziria compiler, we evaluate our implementation using Ziria's WiFi implementation, including its standard library routines, written in C, such as FFT, IFFT, and Viterbi.

Most SDR platforms are based on FPGAs [9,10]. Platforms supporting development of SDR applications on commodity CPUs have become more common [3,14], in particular due to the availability of the GNU Radio [4] environment. There are numerous approaches to programming SDR applications; however, these platforms do not provide the combination of performance and powerful abstractions needed for SDR, instead relying on graph-based models of signal processing.

Mainland [8] describes a number of source-to-source transformations on the core language from Sect. 4 and additional optimizations that are responsible for much of the performance increase over wplc shown in Sect. 6.

### 7.2 Capturing Effects in Types

If we were to re-cast Ziria as an embedded DSL, especially if we were to embed it in Haskell, extensible effects [7] would be an obvious path to differentiating between pure terms, terms that manipulate memory references, and terms that perform IO. However, utilizing extensible effects in our setting would require a substantially more general—and more complicated—type system. The type system we present in Sect. 3 has just enough features to support our




**(a)** Performance improvement of transmitter/receiver pipelines at different 802.11 data rates

**(b)** Performance improvement of individual blocks

**Fig. 6. Performance improvement ratios.** These figures show the relative improvement of kzc over wplc both for entire transmitter/receiver pipelines and for individual blocks. The vertical axis gives the ratio of the throughput of the kzc-compiled version to the throughput of the wplc-compiled version. Error bars show the bound of the ratio when the two metrics being compared range from one standard deviation below the mean to one standard deviation above the mean. Note that Fig. 6a uses a linear scale, whereas Fig. 6b uses a log scale.

requirements, and we have not previously seen the technique of constraining two type indices to be equal in order to distinguish between a term that consumes a value, but is polymorphic in its input, and a term that is polymorphic in its input because it doesn't consume anything at all.

It is not clear how to type Ziria's *par* combinator ($\ggg$) in an EDSL setting. We see this as an argument for a non-embedded DSL. Choosing a stand-alone DSL also allows us to provide syntax that is more familiar to Ziria's likely customers, imperative programmers, and provide an impure surface language.

### 7.3 Elaboration to a Pure Language

Our technique for elaborating the impure surface language into a pure, monadic core language is reminiscent of the technique described by Swamy et al. [13] for adding monadic programming to ML. Our elaboration is constrained to a single monad (ST) and, again, provides just enough type system support for the feature we desire. Implementing a more general, extensible system of elaboration would require a significantly more complicated type system and compiler.

## 8 Conclusions and Future Work

We have presented a type system and elaboration procedure for mapping the high-level, impure language Ziria to a pure, monadic core language where terms' effects are distinguished by their type rather than syntactically. We have also described an improved compilation model for Ziria that avoids unnecessary

control flow and imposes zero additional overhead for sequencing computations that do not perform IO. All work we describe is implemented in the kzc compiler, and benchmarks show our implementation improves upon the existing Ziria system.

Although a more complicated type system could perhaps capture our elaboration procedure and technique for tracking effects in types, we believe we have hit a domain-specific sweet-spot—a more general type system would require a more complex implementation. Far from being limited to the software-defined radio domain, our techniques apply to general producer-consumer computations where combinators like take and emit are provided as language built-ins. Providing these built-in communication primitives allows kzc to use our efficient compilation method—the compiler is able to know when communication between components is occurring and can optimize this communication across components.

### 8.1 Future Work

Our immediate goal is to eliminate the **seq**/**do** distinction in the surface language via a new Ziria dialect, thereby providing a more natural surface language for SDR programmers. In order to provide backwards-compatibility, this will likely require adding a simple module system to allow for code written in both Ziria dialects to coexist in the same program. We will not abandon whole-program compilation, as this is vital for cross-component optimizations such as fusion [8].

Longer term, we plan to target the FPGA hardware in common SDR platforms directly by generating HDL, such as VHDL or Verilog, directly from Ziria and gradually moving portions of the 802.11 pipeline into hardware. We are also actively working on implementing blocks like FFT, IFFT, and Viterbi directly in Ziria, with promising results. Eventually, we hope to re-implement SOFDM [5] in Ziria and use that experience to make Ziria a viable language for hardware development. We also plan to broaden the applicability Ziria, including applications such as wireless MAC protocols and video codecs. Finally, we plan to fully formalize our algorithmic inference algorithm.

## References

1. IEEE Std 802.11™–2012, pp. 1–2793, March 2012
2. bladeRF Software Defined Radio, September 2016. https://www.nuand.com/
3. USRP Software Defined Radio (SDR) online catalog, September 2016. https://www.ettus.com/product
4. Blossom, E.: GNU radio: tools for exploring the radio frequency spectrum. Linux J. **2004**(122), 4 (2004)
5. Chacko, J., Sahin, C., Nguyen, D., Pfeil, D., Kandasamy, N., Dandekar, K.: FPGA-based latency-insensitive OFDM pipeline for wireless research. In: Proceedings of the 2014 IEEE Conference on High Performance Extreme Computing Conference (HPEC 2014), Waltham, MA, pp. 1–6, September 2014

6. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. **37**(1–3), 67–111 (2000)
7. Kiselyov, O., Sabry, A., Swords, C.: Extensible effects: an alternative to monad transformers. In: Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell 2013), Boston, MA, pp. 59–70, September 2013
8. Mainland, G.: Better living through operational semantics: an optimizing compiler for radio protocols (2016, in submission)
9. Murphy, P., Sabharwal, A., Aazhang, B.: Design of WARP: a flexible wireless open-access research platform. In: Proceedings of the 14th European Signal Processing Conference (EUSIPCO 2006), Florence, Italy, pp. 1–5, September 2006
10. Ng, M.C., Fleming, K.E., Vutukuru, M., Gross, S., Arvind, Balakrishnan, H.: Airblue: a system for cross-layer wireless protocol development. In: Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2010), La Jolla, CA, pp. 4:1–4:11 (2010)
11. Stewart, G., Gowda, M., Mainland, G., Radunovic, B., Vytiniotis, D., Agulló, C.L.: Ziria: an optimizing compiler for wireless PHY programming. In: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015), Istanbul, Turkey, March 2015
12. Stewart, G., Vytiniotis, D., Mainland, G., Radunovic, B., de Vries, E.: Ziria, version 85cc34db, April 2016. https://github.com/dimitriv/Ziria
13. Swamy, N., Guts, N., Leijen, D., Hicks, M.: Lightweight monadic programming in ML. In: Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011), Tokyo, Japan, pp. 15–27, September 2011
14. Tan, K., Zhang, J., Fang, J., Liu, H., Ye, Y., Wang, S., Zhang, Y., Wu, H., Wang, W., Voelker, G.M.: Sora: high performance software radio using general purpose multi-core processors. In: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2009), Boston, MA, pp. 75–90, April 2009