UNSW
AUSTRALIA

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Extending a Purely Functional Language to Enable Low-Level Systems Programming

by

Luka Kerr

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Software Engineering

Submitted: December 11, 2020 Student ID: z5214138

Supervisor: Christine Rizkallah

Abstract

COGENT is a uniquely-typed functional programming language implemented in Haskell for developing trustworthy and efficient systems code. COGENT has a data-description language, called DARGENT, that allows programmers to specify how they want their data-types to be represented in memory.

Currently low-level systems programming in COGENT is feasible, although it leaves many features to be desired. To help combat this, two extensions to DARGENT and COGENT will be implemented – endianness annotations and dependent sized structures respectively. Endianness annotations allow the programmer to specify the byte order of integers in memory, and dependent sized structures allow for data structures, such as records and arrays whose size depends on an arbitrary value, to be defined.

The main aim of this thesis is to help better enable low-level systems programming in COGENT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Systems programming is found everywhere, from operating systems and game engines to industrial automation and medical devices. Most systems code is implemented in 'low-level' programming languages such as C and assembly which deal with memory access and manipulation directly. When compared to other more 'high level' programming languages such as Java, Python and Ruby, these low-level programming languages are considered to be much more unsafe and unsecure due to a focus on speed and control, rather than memory safety.

Consequently, the likelihood of bugs and vulnerabilities appearing in a program written in a low-level programming language is far greater than one written in a high level programming language. As such, the development of programming languages that have a focus on correctness and safety, but also ensure that programmers have control over how data is laid out, is welcomed when it comes to writing critical systems code.

COGENT [5, 6] is a programming language that has both a focus on correctness and the ability to control how data is laid out in memory. It achieves this

partially through the integration of a data description language called DARGENT [3]. DARGENT allows the programmer to specify how COGENT lays out its algebraic data types in memory through the use of layout expressions.

One feature of programming languages that allow for fine-grained control over data layouts is endianness annotations. Programmers have the ability to annotate integers with a flag denoting whether the integer should be laid out using big or little endianness.

Another useful feature to extend the expressivity of how data is laid out is dependent sized structures, which would allow data structures, such as records and arrays, whose size depends on a numeric value to be defined. Specifically, a key use case for this feature is to be able to represent certain kinds of systems data structures such as directory entries, as well as operations on these structures, directly in COGENT. In the case of directory entries, a character array representing the directory entry name would be stored in a record, where the array's size depends on some numeric field in the same record. This record would also be 'dependently sized' in that it would include a special field indicating its size which is dependent on its overall contents.

Due to the dynamic nature of these dependently sized structures, in that their size is not known at compile time, they must be stored in some sort of buffer with a fixed maximum capacity. Various safety properties must also be ensured when interfacing with this buffer and its contents, for example guaranteeing that a buffer element cannot be appended to the end of a buffer if the buffer is full.

Having these two aforementioned features implemented directly in COGENT would require less COGENT and C code to be written in total, and hopefully

more efficient C system code generated due to less interaction between COGENT and C. As COGENT also guarantees memory safety and eases verification [6], if this kind of systems code can be represented in COGENT, verification becomes easier and memory safety is guaranteed once the type system and verification framework has been adapted to account for these features.

In this thesis, two extensions to COGENT are presented – endianness annotations inside DARGENT layouts using a new layout syntax, and the introduction of three new types in COGENT to represent dependent sized structures and how they are stored. A complete implementation of endianness annotations has been achieved, as well as an initial design and partial implementation of dependent sized structures, including a pen and paper formalisation of the typing rules and dynamic semantics.

Chapter 2

Background

2.1 Cogent

COGENT [5] is a functional programming language equipped with a uniqueness type system that also supports interoperability with existing C code. Its goal is to enable systems code to be verified more easily [7]. It achieves this by generating C code, as well as both shallow embeddings of COGENT programs in Isabelle/HOL [8], an interactive theorem prover, and a proof that the generated C code refines this embedding [9].

Figure 2.1 displays the Haskell inspired [5] syntax of a small COGENT program that interacts with an abstract type `Buffer` and function `setChar` that are both implemented in C. The function `writeHi` operates on this buffer by writing the characters 'H' and 'i' at the specified indices.

```
1  -- Abstract type
2  type Buffer
3
4  -- Abstract function implemented in C
5  setChar : (Buffer, U32, U8) -> Buffer
6
7  writeHi : Buffer -> Buffer
8  writeHi buf =
9    let buf = setChar (buf, 0, 'H')
10   and buf = setChar (buf, 1, 'i')
11   in buf
```

Figure 2.1: A small COGENT program

Along with the ability to integrate with C, COGENT also operates on numerous algebraic data types including product types (records) and sum types (variants) as highlighted in Figure 2.2. These types seamlessly map to systems types, for example, C structs can be represented using product types, and tagged unions can be represented using sum types. COGENT also supports other functionality such as higher-order functions and polymorphism.

```
1  -- Product type
2  type Date = { year: U16, month: U8, day: U8 }
3
4  -- Variant type
5  type Maybe a = <Some a | None ()>
```

Figure 2.2: COGENT product and variant types

COGENT also has a foreign function interface (FFI) to C, in which data types and iterators over these types are implemented in C and called by COGENT code using the FFI. So far, these data types have not been fully verified as correct, and some of them even create an overhead on performance [3]. The development of a data description language (DDL), called DARGENT aims to solve this problem by providing efficient access to data types from COGENT.

2.1.1 Dargent

Dargent [3] is a DDL for Cogent that is currently under active development, allowing the programmer to specify how Cogent lays out its algebraic data types in memory. It implements syntax that allows data to be represented at specific bits and bytes when stored in a record or variant. This is fundamental when working with low-level systems, such as operating systems, especially when Cogent programs are integrated with pre-existing C programs that use much more exotic data representations.

A Dargent program consists of one or more layout declarations that allow for the layout of data types in memory to be expressed. Primitive types are laid out as a contiguous block of memory, whereas algebraic data types, such as records and variants, can be laid out by the programmer via a layout declaration.

Figure 2.3 demonstrates the syntax of a Dargent layout on a record type `XY` containing two fields, a `U8` and `U16`. The `layout` declaration that follows the type definition defines how `XY` is laid out in memory. The field `x` takes up 8 bits, starting at the 3rd byte, while the field `y` takes up 2 bytes starting at the 0th byte.

```
1 type XY = { x: U8, y: U16 }
2   layout
3     record { x: 8b at 3B, y: 2B at 0b }
```

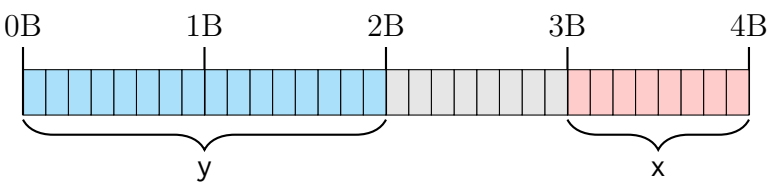

Figure 2.3: A Dargent layout and its memory representation

2.2 Endianness

Endianness is the sequential order of a range of bits or bytes representing a number, stored in memory [10]. Typically, endianness indicates the byte order of a multi-byte number, such as a `U32`. Table 2.1 and Table 2.2 display the two different types of endianness using the 32 bit number `0xFF00AA11` as an example. Specifically, these two types are big endian and little endian. Big endian refers to the ordering in which the most significant byte comes first and the least significant byte comes last. On the other hand, little endian is represented using the opposite ordering, in which the most significant byte comes last, and the least significant byte comes first.

Address	Data
0x0000	FF
0x0001	00
0x0002	AA
0x0003	11

Table 2.1: Big endian addressing

Address	Data
0x0000	11
0x0001	AA
0x0002	00
0x0003	FF

Table 2.2: Little endian addressing

The importance of endianness is illustrated in various significant types of systems and structures. Some common examples of where endianness comes into play are network protocols, file systems, computer processors and file formats. All of these systems and structures are related by the fact that they may use either big or little endian ordering (or both) to store, represent or transfer data.

When it comes to network protocols, the bit and byte-level order of data (known as the *network order*) transmitted over the wire is specified to be either big or little endian. In practice, most network protocols such as the Internet

Protocol (IP) [11] are big endian, however some, such as the Server Message Block (SMB) [12] protocol, are little endian. Due to this incompatibility, it is highly important to take endianness into account when developing programs that interact with one or many network protocols.

In a similar vein, a file system or a file format may have a different endianness depending on the machine that created it, or the file format itself. The type of endianness used is extremely important when it comes to file systems and file formats, as once data is written to a file in a specific endianness, it must also be read using that same endianness, otherwise the data read wouldn't match what was written.

The ability to specify endianness in a high level programming language is rare, with the compiler usually assuming an endianness based on a target processor. However, some programming languages built specifically for describing and manipulating binary data allow the programmer to annotate integers with a specific byte order. Two specific examples of these kinds of programming languages will be looked at – DataScript [13] and Preon [14]. These programming languages were chosen because they model exactly the kind of endianness annotations extension we want to implement in Dargent.

2.2.1 DataScript

DataScript [13] is a programming language that allows for multi-byte integers to be prefixed with an optional attribute denoting their byte order. Figure 2.4 highlights this functionality. If the `little` keyword is used, the integer is represented using little endianness, and if the `big` keyword is used, the integer is represented using big endianness. In the case that neither the `little` or `big`

keyword is provided, the DataScript compiler defaults to big endianness.

```
1 // Little endian
2 const little uint32 magic = 0xCAFEF00D;
3
4 // All big endian
5 const uint16 ACC_PUBLIC        = 0x0001;
6 const big uint16 ACC_PRIVATE   = 0x0002;
7 const big uint16 ACC_PROTECTED = 0x0004;
```

Figure 2.4: DataScript byte order attributes

2.2.2 Preon

Another programming language developed specifically for dealing with binary endcoded data is Preon [14]. It allows data structures to be represented in Java classes, accompanied by a variety of annotations that are applied when the data structure is mapped onto a bitstream encoded representation, using a decoder generated by Preon. One such annotation is the `@BoundNumber` annotation (displayed in Figure 2.5), which takes an optional `byteOrder` parameter to denote the byte order of an integer.

```
1 class Square {
2   @BoundNumber(byteOrder=LittleEndian) private int width;
3   @BoundNumber(byteOrder=LittleEndian) private int height;
4 }
```

Figure 2.5: Preon byte order annotations

2.3 Dependent Pairs

Given a pair, a *dependent pair* allows the type of the second element of that pair to depend on the value of the first element. More formally, a value of the

type $\Sigma(x : A)\ P(x)$ is a pair (a, b) where $a \in A$ and $b \in P(a)$ [15]. Dependent pairs represent existential quantification, they are made up of a witness for an existential claim as well as a proof that a property holds for it.

Dependent pairs are uncommon in most high level programming languages, including COGENT, and virtually non-existent in low-level ones. Having said that, there do exist programming languages where dependent pairs are integrated directly into the language's type system. Two such languages are Idris and Agda, which will be looked to understand how they incorporate dependent pairs. Afterwards, the use cases of dependent pairs will be discussed to further help understand the motivation behind why dependent pairs are useful.

2.3.1 Idris

Idris is a functional programming language with a focus on type-driven development. As such, it is equipped with many useful features including dependent types, optional laziness and of course dependent pairs. Figure 2.6 depicts the data type definition of a `DPair`, or a dependent pair. It can be seen that `DPair` takes two parameters, `a` which is a type, and `P` which is a function from `a` to another type, and returns a new type representing a dependent pair.

This definition of a dependent pair also comes with a constructor `MkDPair` which takes two arguments, `x` of type `a`, and a function `P` of type `a -> Type` with `x` as an argument.

```
1 data DPair : (a : Type) -> (P : a -> Type) -> Type where
2   MkDPair : {P : a -> Type} -> (x : a) -> P x -> DPair a P
```

Figure 2.6: Idris `DPair` data type [1]

An example of an Idris type that may be used as the second element in a dependent pair is `Vect`, defined in Figure 2.7. This definition takes a `Nat`, representing the vector's length, and a `Type`, representing the type of elements in the vector. In fact, `Vect` is a *dependent type*, or a type whose definition depends on a value. This is primarily why `Vect`s are commonly found in dependent pairs, as their type depends on the value of the first element in the pair.

```
1 data Nat = Z | S Nat
2
3 data Vect : Nat -> Type -> Type where
4   Nil  : Vect Z a
5   (::) : a -> Vect k a -> Vect (S k) a
```

Figure 2.7: Idris `Vect` data type [1]

When it comes to using dependent pairs, Idris provides syntactic sugar to more easily construct a dependent pair. Specifically, a `DPair` combining a vector with its length may be defined as `DPair Nat (\n => Vect n Int)` using the standard `DPair` syntax, or as `(n : Nat ** Vect n Int)` using syntactic sugar. An example of how to construct a small vector is shown in Figure 2.8.

```
1 vec : (n : Nat ** Vect n Int)
2 vec = (3 ** [4, 6, 8])
```

Figure 2.8: Idris dependent pairs

2.3.2 Agda

Another programming language that supports dependent pairs is Agda. The definition of a dependent pair Σ is portrayed in Figure 2.9, and is fairly similar

to the Idris `DPair` type. In Agda, a dependent pair may represent existential quantification (as we have seen), but also a subset of a set.

```
1 data Σ (A : Set) (B : A → Set) : Set where
2   _,_ : (a : A) → (b : B a) → Σ A B
```

Figure 2.9: Agda Σ data type [2]

Assuming A is a type and P is a predicate over A, then the set of all elements such that P holds can be represented by the dependent pair $\Sigma\, A\, P$. For example, the finite set of natural numbers where all elements are less than some number $b \in \mathbb{N}$ can be represented as $\Sigma\, \mathbb{N}\, (\lambda a \rightarrow a < b)$.

2.3.3 Use Cases

In general, dependent pairs can be used to pass around a value x, together with some kind of proof that the value satisfies a proposition $P(x)$ [16]. This ensures in a type safe manner that whenever the pair is used (for example, as input to a function), that its value is guaranteed to satisfy the associated proof. This is very powerful, and an example of where it may be used is in the implementation of a `filter` function operating on a `Vect`.

Filter

Figure 2.10 displays the type of Idris's `filter` function. As we know from the definition above, a `Vect` has an associated value that represents its length. The reason `filter` uses dependent pairs is a matter of existential quantification, in that the length of the resulting `Vect` depends on both the predicate `f` and the `Vect` provided.

```
1 filter : (a -> Bool) -> Vect n a -> (p ** Vect p a)
```

Figure 2.10: Idris filter function type [1]

The type of `filter` could not possibly be `(a -> Bool) -> Vect n a -> Vect p a`, because there is no way to determine what the value of `p` is without first running the function. The type `Vect p a` itself doesn't prove that `p` is the length of the vector either, rather it establishes the *claim* that it is. A *proof* of this claim (i.e. an instance of `Vect`) takes the form of a dependent pair `(p ** Vect p a)`, which itself includes a *witness* `p` and a proof that the witness holds for the claim.

2.3.4 Relation To Dependent Sized Structures

As discussed in Chapter 1, dependent sized structures are data structures whose length, or size, depends on another integer value. They can be thought of as a modified form of a dependent pair, in the sense that rather than their type being dependent on a value, their size is. One such dependent sized structure is an array whose size depends on a value. This form of array is especially analogous to vectors, in that they both have a size, or length, that is dependent on another numeric value.

Chapter 3

Completed Work

3.1 Endiannness Annotations

The first extension to COGENT was primarily an engineering component which involved implementing new syntax for DARGENT layouts to allow the endianness of primitive types to be specified. A new keyword `using` was introduced to COGENT's surface language, as well as two types of endianness annotations – `BE` and `LE` representing big endianness and little endianness respectively. Users can append a suffix of `using BE` or `using LE` to a primitive type inside of a DARGENT layout, indicating to the compiler what kind of endianness they want the primitive type to be represented as when stored in memory.

3.1.1 Syntax

Figure 3.1 demonstrates a record type `XYZ` that contains three primitive types `x`, `y` and `z`. The layout of `XYZ` is specified, as well as the endianness for both `x` (little endian) and `y` (big endian). The endianness of field `z` is not specified, so it defaults to the machine's endianness.

```
1 type XYZ = { x: U32, y: U16, z: U32 }
2   layout
3     record { x: 4B at 2B using LE -- little endian
4            , y: 2B at 0B using BE -- big endian
5            , z: 32b at 6B         -- machine endian, by default
6            }
```

Figure 3.1: Endianness annotation syntax

3.1.2 Abstract Grammar

Figure 3.2 conveys DARGENT's abstract grammar, in which new additions have been highlighted in green. One new production has been added, called 'endianness', which specifies the two possible different types of endianness. We have also introduced two new cases for the layout expression ℓ, that is, s **using** e to allow for the endianness to be specified after a size, and s **at** s **using** e to allow for the endianness to be specified after an offset following a size.

sizes	s	::=	$n\mathsf{B} \mid n\mathsf{b} \mid n\mathsf{B} + m\mathsf{b}$	
layout expressions	ℓ	::=	s	(block of memory)
		|	s **using** e	(size endianness)
		|	s **at** s **using** e	(offset endianness)
		|	L	(another layout)
		|	ℓ **at** s	(offset operator)
		|	**record** $\{\overline{\mathsf{f}_i : \ell_i}\}$	(records)
		|	**variant** (ℓ) $\{\overline{\mathsf{K}_i\ (n_i) : \ell_i}\}$	(variants)
declarations	d	::=	**layout** $\mathsf{L} = \ell$	
layout names	L			
record field names	f			
variant constructors	K			
numbers	n, m	$\in$	$\mathbb{N}$	
endianness	e	::=	$\mathsf{LE} \mid \mathsf{BE}$	

Figure 3.2: DARGENT's updated abstract grammar [3]

The reason we can't simply add a single new case of ℓ **using** e is because we

only allow endianness annotations directly on primitive types, not record or variant layouts. If fields inside a record or variant had an endianness annotation of `LE` for example, and the record of variant had an endianness annotation of `BE`, it wouldn't be clear to both the compiler and the programmer which one would take precedence.

3.1.3 Implementation

Figure 3.3 indicates the main stages of the COGENT compiler. The stages highlighted in green are the ones that were modified to implement endianness annotations, each of which will be discussed next.

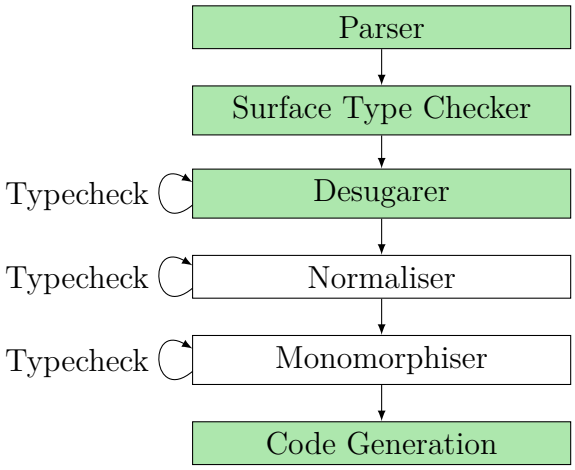

Figure 3.3: COGENT's main compiler stages

Parser

As new surface syntax was introduced, COGENT's parser needed to be modified. Three new 'reserved names' were added – `using`, `BE` and `LE`. Now when parsing a DARGENT layout representation expression we check for a `using` keyword and parse it, along with the endianness specified, into a `DataLayoutExpr`

type using a newly introduced `Endian` constructor. This `Endian` constructor takes a `DataLayoutExpr` expression as its first parameter and an `Endianness` type as its second parameter. The `Endianness` type is defined as `data Endianness = LE | BE | ME`, where `ME` represents machine endianness.

An example of a COGENT program that includes a type definition and associated DARGENT layout is shown in Figure 3.4. Given this program, a relevant part of the abstract syntax tree (AST) parsed is displayed in Figure 3.5. As we can see, the type `X` has been laid out as a `Record`, and inside this `Record` we have an `Endian` type containing an expression on how to layout the field `x`, as well as the `LE` endianness specified.

```
1 type X = { x: U32 }
2   layout
3     record { x: 4B using LE }
```

Figure 3.4: A simple COGENT type definition with endianness annotations

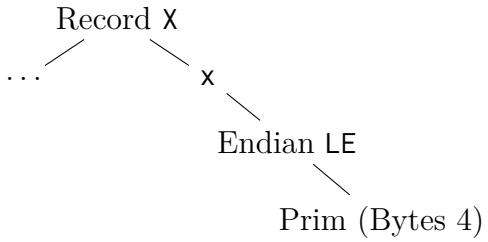

Figure 3.5: Part of the AST produced from Figure 3.4's layout expression

Type Checker

During the type checking phase, we perform type checking on the expression inside the `Endian` type. As we require at least a size to be specified inside a DARGENT layout on a primitive type, if this expression (with a size) does not

type check, we know that the same expression with an endianness annotation also will not type check. If the expression does type check, then it will also type check with an endianness annotation.

Desugarer

COGENT's desugarer was also modified, specifically when desugaring layouts. A case was added to handle `Endian` types (or `TLEndian` in the desugarer) in which we desugar the `Endian` expression and return a `PrimLayout` with the specified endiannness attached. A `PrimLayout` is used to represent the bit-level layout of primitive fields. Elsewhere, wherever `PrimLayout`'s are constructed during desugaring, we attach an endiannenss of `ME`, as we either do not know the endianness specified, or an endianness wasn't specific at all.

Code Generation

In the code generation stage, we generate different getters and setters for fields depending on each field's endianness. If the endianness is `ME`, we don't convert between endianness at all. If the endianness is `BE` or `LE` we perform the following steps inside the COGENT compiler:

1. Include in the generated C output, a C header file called `cogent-endianness.h` containing C functions to convert between endianness for COGENT's various primitive types (`U8`'s, `U16`'s, `U32`'s, and `U64`'s)
2. Generate one of these endianness conversion function names
3. Wrap this endianness conversion function around the expression located in the body of each relevant field's getter and setter

Whenever the values of fields are used inside a COGENT program, these getters and setters are called to convert the field's value to the required endianness.

An example of getting and setting a value `x` on a little endian machine is displayed in Figure 3.6. On the left we have a COGENT program that contains a record with a field `x` as big endian, as well as a function to set `x` to the value 1, and one to double the value of `x`. On the right we have some of the relevant generated C code for the COGENT program.

```
type Rec = { x: U32 }
  layout
    record {
      x: 4B using BE
    }

init : Rec -> Rec
init r = r { x = 1 }

doubleX : Rec -> Rec
doubleX r {x} =
  r { x = x * 2 }
```

```
#include <cogent-endianness.h>
...

static inline
u32 d2_get_x(t1 *b)
{
  return be_u32_swap(
    (u32) d3_get_x_part0(b) << 0U);
}

...

static inline void
d4_set_x(t1 *b, u32 v)
{
  d5_set_x_part0(
    b,
    be_u32_swap(v) >> 0U & 4294967295U
  );
}

...
```

Figure 3.6: COGENT program and relevant generated C code

When `x` is used in the COGENT program, for example in the `doubleX` function, `x`'s getter `d2_get_x` is called which itself fetches the (big endian) value of `x` from memory and converts it into little endian for the COGENT program to use. Whenever `x` is set in the COGENT program, `x`'s setter `d4_set_x` is called which first converts `x` from little endian back into big endian, and then stores

it in memory again. As we can see, it is always the case that the in-memory representation of a value is stored in the specified endianness.

3.2 Dependent Sized Structures

The second extension to COGENT had both a reasearch and engineering component. As outlined in Chapter 1, we want to implement various kinds of dependent sized structures. As no form of dependent types or structures is implemented in COGENT currently, a large part of this extension was formulating a design for both the various structures themselves, as well as the operations on these structures, based off of real-world systems code.

3.2.1 Overview

The first part of this extension was identifying exactly what types would be needed in COGENT to represent the data structures we want to implement. As the main motivation for this extension is to further enable systems programming, we focused on three data structures commonly used in systems code – buffers, structs and arrays. The two use cases that drove this decision are directory entries and network packets, both of which require a buffer-like structure to store 'dependently sized' structs containing both their own size, and a dependently sized array that holds data. Figure 3.7 portrays an example of such a structure, specifically a directory entry in the ext2 file system. In this example, the structure's size is dependent on the `rec_len` field, and the character array `name[]`'s size is dependent on the `name_len` field.

```
1 struct ext2_dir_entry_2 {
2   uint32_t inode;      /* Inode number */
3   uint16_t rec_len;    /* Directory entry length */
4   uint8_t  name_len;   /* Name length */
5   uint8_t  file_type;
6   char     name[];     /* File name, up to EXT2_NAME_LEN */
7 }
```

Figure 3.7: Linux kernel ext2 file system directory entry structure [4]

To store such dependently sized structs in C, systems programmers often pre-allocate a fixed-size buffer on the heap that contains pointers to these structs. Each struct has a field denoting its size, which is used to compute where the next struct begins. Two operations that are often used to interact with this buffer are iteration over the packed structs in the buffer, and appending a new struct after the last inserted struct in the buffer. Figure 3.8 portrays this type of buffer, as well as the information used to perform iteration and appending.

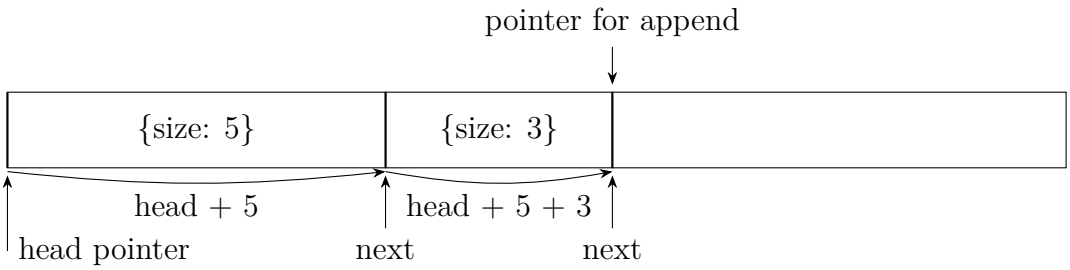

Figure 3.8: Buffer data structure diagram

It is important in systems programs that this buffer is allocated on the heap and not on the stack because its content is typically shared amongst functions. Although, this sharing goes against the essence of linear types. In COGENT pointers must have a linear type [6], allowing COGENT programs to have both an imperative and a functional view. The imperative view allows

for representing pointers, and the functional view allows these pointers to be replaced by their value and directly forgotten about without changing the meaning of the program (i.e. referential transparency). This switch from an imperative to a functional view is essential for creating the certifying compiler which eases the verification of COGENT code.

Another important pre-requisite to the design is understanding that the elements of the dependently sized arrays in the use cases we want to consider usually do not have a linear type, they are often just characters or other unboxed types of some sort. Moreover, we never want to perform any complex operations on these arrays, all we want to do is either access them or store them inside a sequence of structures in a linked-list fashion and iterate over the linked-list in an efficient manner. These restrictions significantly ease our design.

3.2.2 Design

The main part of the design of these dependent sized structures was devising how they would be stored given that their size is unknown at compile time. Furthermore, both dependent sized records and dependent sized arrays operate very similarly to existing records and arrays in COGENT, the key difference is that extra information is recorded about the value that their sizes depend on. Thus, the focus of the design was on the buffer type and its associated operations, as well as retaining memory safety. The proof that memory safety is retained is beyond the scope of this thesis.

The final design that was developed involves a buffer data structure that contains three important components:

- a pointer to the block of memory representing the buffer's contents;
- an integer $\mathtt{n}$ representing the buffer's maximum fixed size;
- an integer $\mathtt{c}$ representing the buffer's current initialised capacity.

Both the maximum fixed size $\mathtt{n}$ and the current initialised capacity $\mathtt{c}$ are important when performing operations on the buffer such as iterating and appending, as we want to ensure that these operations do not overrun the bounds of the buffer, or overwrite existing contents of the buffer. The condition $0 \leq \mathtt{c} \leq \mathtt{n}$ should always hold before and after each operation on the buffer. At the type level, only $\mathtt{n}$ will be known due to its static nature, whereas $\mathtt{c}$ will form part of the dynamic data structure.

When it comes to the elements stored in the buffer, as mentioned earlier, these are pointers to dependently sized structures in the form of records that must contain a field indicating their size in bytes, along with any other regular fields too. This is required as it allows for 'pointer hopping' from one structure to the next. Inside these structures, there can exist pointers to dependently sized arrays, which hold similar information, namely the field that their size depends on, as well as the type of elements they hold. This field must be stored in the same structure the array is stored in. Figure 3.9 demonstrates an extended version of Figure 3.8, including these extra constraints and information. The syntax `a: asize U8[]` represents an array containing `U8`s, whose size is dependent on the field `asize`.

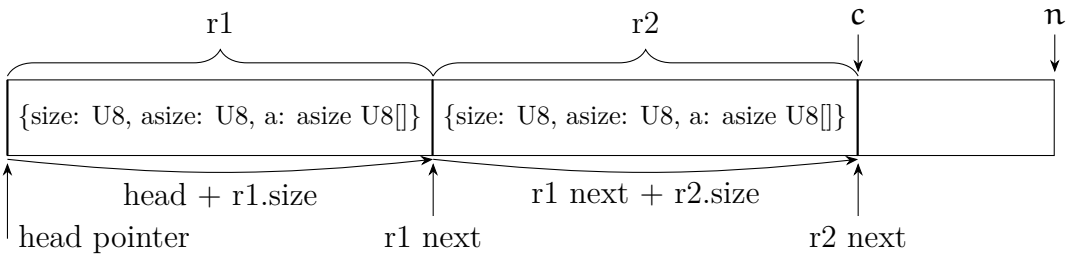

Figure 3.9: Extended buffer data structure diagram

In theory, the size of the dependent sized records could be deduced from the sizes of the types of each field in the record, as well as the value of the field that the dependent sized array depends on. For now, to simplify the initial design and compiler implementation, we have chosen to require specifying the size of the dependent sized record explicitly. In the future, the size of a dependent sized record could be computed by the compiler automatically, and syntactic sugar could be introduced allowing the programmer to specify the size if they wish to do so.

The current design preserves memory safety inside the buffer via various dynamic checks due to the fact that the sizes of the dependently sized structures are not known at compile time. These dynamic checks will be explained in more depth in Section 3.2.5. Part of the future work of this thesis is formulating a way to perform these memory safety guarantees statically.

3.2.3 Implementation

Similar to endianness annotations, various stages of the COGENT compiler needed modification to implement the buffer and the two dependent sized structures. These stages were primarily the parser and the type checker. The code generation stage was left unmodified, and instead extra effort was put

into finalising the C implementation of the buffer, as well as developing the typing rules for operations on the buffer.

Parser

COGENT's parser was modified to handle the three new types introduced into it's surface syntax. These type are `DArray` for dependent sized arrays, `DRecord` for dependent sized records, and `Buffer` for buffers. When parsing, we now check for a `Buffer` type that contains `DRecord`s, which themselves can contain `DArray`s as members. Figure 3.10 conveys a `Buffer` type that can now be parsed, containing both dependent sized records and arrays. A dependent sized record looks just like a regular record, except with a field `size` prefixed with the character `>` to indicate to the compiler which field to use to get the record's size. A `name` field also exists, having the type of a dependent sized array to represent the directory entry name, of which its size depends on the `name_size` field.

```
1 type DirEntry = Buffer[2000](#{ >size: U8
2                               ,  name_size: U8
3                               ,  name: DArray name_size U8 })
```

Figure 3.10: Directory entry buffer type

Type Checker

During COGENT's type checking phase, there are various well-formedness checks that are performed on the three new types mentioned above to ensure they are used correctly. Notable checks include:

- ensuring `DArray`s are only used inside `DRecord`s;

- ensuring a `DArray`s size depends on a numeric field that exists inside the same `DRecord`;
- ensuring a `DRecord` has a field indicating its size, and this field is numeric.

C Implementation

Before the code generation phase is completed, a C implementation of the buffer and the dependent sized structures, along with operations on these data structures, needed to be implemented and refined. This C implementation models the design detailed in Section 3.2.2, and follows precisely how systems programmers would implement such structures. Figure 3.11 conveys the data structure definition of the buffer, which contains the same components outlined in Section 3.2.2.

```
1 struct Buffer {
2   int current_capacity;
3   int max_capacity;
4   struct DRecord *buf;
5 };
```

Figure 3.11: C Buffer data structure

Initialising the buffer needs to be done in C and called by COGENT code using the FFI. This is because the buffer contains memory that is dynamically allocated. All other operations on the buffer will be able to be invoked directly inside a COGENT program.

3.2.4 Typing Rules

Along with these new types comes several operations that can be performed, especially surrounding the `Buffer` type, as this is the main structure that is

used to store and read data from. As we don't know the size of dependent sized arrays at compile time, all the type system knows about them is the field that their size depends on, which is not very useful to the programmer. As such, they have been excluded from the following typing rules. A type inference rule is given as a relation of the form $\Gamma \vdash e : \tau$ where Γ is a typing context, e is a term and τ is a type. The syntax used in the typing rules extend the syntax of the basic fragment of COGENT [5].

Dependent Sized Records

A dependent sized record type, written $\{\ell : \chi, \overline{f_i : \tau_i}\}$, where an $\overline{\text{overline}}$ indicates a set, consists of a special field ℓ of numeric type χ that represents the records size in bytes, and one or more fields f_i of type τ_i, where f_i could be a dependent sized array. Dependent sized records are boxed, and extend all the typing rules [5] of a regular boxed record (with sigil ⓡ and ℓ as a normal field), with one additional rule for getting the record's size as displayed in Figure 3.12.

$$\frac{\Gamma \vdash r : \{\ell : \chi, \overline{f_i : \tau_i}\}}{\Gamma \vdash \texttt{size}\ r : \chi}\text{RSIZE}$$

Figure 3.12: Dependent sized record additional typing rule

Buffers

A `Buffer` type, written $\left[n : \chi, \overrightarrow{e_i : \tau}\right]$, where a $\overrightarrow{\text{harpoon}}$ indicates a list of zero or more, consists of a maxium fixed capacity n in bytes of numeric type χ, and zero or more elements e_i of type τ that are stored in the buffer. Figure 3.13 shows the typing rules for all `Buffer` operations we currently want to include.

$$\frac{\Gamma \vdash b : \left[n : \chi, \overrightarrow{e_i : \tau}\right]}{\Gamma \vdash \texttt{size}\ b : \chi}\textsc{Size} \qquad \frac{\Gamma \vdash b : \left[n : \chi, \overrightarrow{e_i : \tau}\right]}{\Gamma \vdash \texttt{capacity}\ b : \chi}\textsc{Capacity}$$

$$\frac{\Gamma \vdash b : \left[n : \chi, \overrightarrow{e_i : \tau}\right]}{\Gamma \vdash \texttt{head}\ b : \langle \texttt{None}\ () \mid \texttt{Some}\ \tau \rangle}\textsc{Head} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash b : \left[n : \chi, \overrightarrow{e_i : \tau}\right]}{\Gamma \vdash b +\!\!+ e : \left[n : \chi, \overrightarrow{e_i : \tau}\right]}\textsc{Append}$$

$$\frac{\Gamma \vdash h : \tau \qquad \Gamma \vdash e : \tau \qquad \Gamma \vdash c \in \mathbb{Z}}{\Gamma \vdash \texttt{next}\ e\ h\ c : \langle \texttt{None}\ () \mid \texttt{Some}\ \tau \rangle}\textsc{Next}$$

$$\frac{\begin{array}{cc}\Gamma \vdash t : K \to \texttt{Bool} & \Gamma \vdash acc : K \\ \Gamma \vdash b : \left[n : \chi, \overrightarrow{e_i : \tau}\right] & \Gamma \vdash f : \tau \to K \to K\end{array}}{\Gamma \vdash \texttt{iter}\ b\ t\ f\ acc : K}\textsc{Iterate}$$

Figure 3.13: Buffer typing rules

Of the above typing rules, the most interesting ones are Head, Next and Iterate. Both Head and Next have the same return type, either the type of a buffer element, or nothing in the case the buffer is empty or there is no next element. Iterate takes in three key arguments along with the buffer: a termination function t, a mapping function f and an initial value acc to use for the accumulator. Its return type is K representing the final accumulated value that may have been modified by f.

3.2.5 Dynamic Semantics

Whilst the typing rules for buffers and dependent sized records are useful, they don't convey much about the behaviour or the dynamic checks performed for each operation. The dynamic semantic rules below aim to bridge the gap between the purely functional typing rules introduced, and the prospective generated C code. These rules are specified as a big-step evaluation relation $V \vdash e \Downarrow v$ where an expression e is evaluated in an environment V to a resulting

value ν. Figure 3.14 demonstrates the syntax used for these rules, extended from COGENT's syntax for dynamic semantics interpretations [5].

$$
\begin{array}{llcll}
\text{value semantics values} & \nu & ::= & \{\ell \mapsto \nu, \overline{\mathsf{f}_\mathsf{i} \mapsto \nu_\mathsf{i}}\} & (\textit{buffer elements}) \\
& & | & \left[\mathfrak{n}, \mathsf{c}, \overline{[\mathsf{e}_\mathsf{i} \mapsto \nu_\mathsf{i}]}\right] & (\textit{buffers}) \\
& & | & \ldots & \\
\text{numbers} & \mathfrak{n}, \mathsf{c} & \in & \mathbb{Z} &
\end{array}
$$

Figure 3.14: Syntax for dynamic semantics interpretations

Dependent Sized Records

The main operation we care about for dependent sized records is `size`, as highlighted in the typing rules earlier. The size of a dependent sized record evaluates to the field inside the record used to indicate its size. The rule for this operation is displayed in Figure 3.15.

$$\frac{V \vdash \mathsf{r} \Downarrow \{\ell \mapsto \nu, \overline{\mathsf{f}_\mathsf{i} \mapsto \nu_\mathsf{i}}\}}{V \vdash \mathtt{size}\ \mathsf{r} \Downarrow \nu}\text{VRSIZE}$$

Figure 3.15: Dependent sized record dynamic semantics evaluation rule

Buffers

Buffers have many more operations we want to model the dynamic semantics for. We also introduce an expression $\%\mathsf{e}$ that represents the address of a buffer element e in memory, along with a rule to convert from an address to an integer (VADDRESS), and a rule to convert from an integer to an address (VCAST). The rules for all buffer operations are displayed in Figure 3.16.

$$\frac{V \vdash b \Downarrow \left[n, c, \overrightarrow{[e_i \mapsto v_i]}\right]}{V \vdash \texttt{size}\ b \Downarrow n}\textsc{VSize} \qquad \frac{V \vdash b \Downarrow \left[n, c, \overrightarrow{[e_i \mapsto v_i]}\right]}{V \vdash \texttt{capacity}\ b \Downarrow c}\textsc{VCapacity}$$

$$\frac{\begin{array}{c} n \in \mathbb{Z} \\ V \vdash e \Downarrow \{\ell \mapsto v, \overrightarrow{f_i \mapsto v_i}\} \end{array}}{V \vdash \%e \Downarrow n}\textsc{VAddress} \qquad \frac{\begin{array}{c} n \in \mathbb{Z} \\ V \vdash e \Downarrow \{\ell \mapsto v, \overrightarrow{f_i \mapsto v_i}\} \\ V \vdash \%e \Downarrow e' \end{array}}{V \vdash \mathit{cast}(n) \Downarrow e'}\textsc{VCast}$$

$$\frac{V \vdash \texttt{capacity}\ b \Downarrow 0}{V \vdash \texttt{head}\ b \Downarrow \texttt{None}\ ()}\textsc{VHead-None} \qquad \frac{\begin{array}{c} V \vdash \%b \Downarrow b' \\ V \vdash \texttt{capacity}\ b \Downarrow c > 0 \\ V \vdash \mathit{cast}(b') \Downarrow e \end{array}}{V \vdash \texttt{head}\ b \Downarrow \texttt{Some}\ e}\textsc{VHead-Some}$$

$$\frac{\begin{array}{ll} V \vdash e \Downarrow \{\ell_e \mapsto v_e, \overrightarrow{f_{e_i} \mapsto v_{e_i}}\} & V \vdash \%e \Downarrow e' \\ V \vdash h \Downarrow \{\ell_h \mapsto v_h, \overrightarrow{f_{h_i} \mapsto v_{h_i}}\} & V \vdash \%h \Downarrow h' \\ & V \vdash e' + v_e - h' \geq c \end{array}}{V \vdash \texttt{next}\ e\ h\ c \Downarrow \texttt{None}\ ()}\textsc{VNext-None}$$

$$\frac{\begin{array}{lll} V \vdash e \Downarrow \{\ell_e \mapsto v_e, \overrightarrow{f_{e_i} \mapsto v_{e_i}}\} & V \vdash \%e \Downarrow e' & v = e' + v_e \\ V \vdash h \Downarrow \{\ell_h \mapsto v_h, \overrightarrow{f_{h_i} \mapsto v_{h_i}}\} & V \vdash \%h \Downarrow h' & V \vdash v - h' < c \\ & & V \vdash \mathit{cast}(v) \Downarrow n \end{array}}{V \vdash \texttt{next}\ e\ h\ c \Downarrow \texttt{Some}\ n}\textsc{VNext-Some}$$

$$\frac{\begin{array}{ll} V \vdash e \Downarrow \{\ell \mapsto v, \overrightarrow{f_i \mapsto v_i}\} & c' = c + v \\ V \vdash b \Downarrow \left[n, c, \overrightarrow{[e_i \mapsto v_i]}\right] & V \vdash c' \leq n \end{array}}{V \vdash b +\!\!+ e \Downarrow \left[n, c', \overrightarrow{[e_i \mapsto v_i}, e_{i+1} \mapsto \{\ell \mapsto v, \overrightarrow{f_i \mapsto v_i}\}\right]}\textsc{VAppend}$$

$$\frac{\begin{array}{c} V \vdash \texttt{size}\ e \Downarrow e' \\ V \vdash \texttt{size}\ b \Downarrow b_s \\ V \vdash \texttt{capacity}\ b \Downarrow b_c \quad V \vdash b_c + e' > b_s \end{array}}{V \vdash b +\!\!+ e \Downarrow b}\textsc{VAppend-Full}$$

$$\frac{\begin{array}{ll} V \vdash f\ v_0\ acc \Downarrow acc' & V \vdash t\ acc' \vdash s \\ V \vdash b \Downarrow \left[n, c, \overrightarrow{[e_i \mapsto v_i]}\right] & \text{for each } i > 0 \wedge \neg s, V \vdash f\ v_i\ acc' \Downarrow acc' \end{array}}{V \vdash \texttt{iter}\ b\ t\ f\ acc \Downarrow acc'}\textsc{VIterate}$$

$$\frac{b \vdash \left[n, c, \overrightarrow{[e_i \mapsto v_i]}\right] \qquad V \vdash acc \Downarrow k \qquad c \leq 0}{V \vdash \texttt{iter}\ b\ t\ f\ acc \Downarrow k}\textsc{VIterate-Empty}$$

Figure 3.16: Buffer dynamic semantics evaluation rules

Out of the operations presented by the dynamic semantic rules, two are important for preserving properties related to memory safety. These are appending and iteration, which modify the buffer and read from elements in the buffer respectively.

The two rules VAppend and VAppend-Full represent the possible cases when appending. The rule VAppend-Full handles the case when the buffer is full, or in other words when a value $\mathtt{c}'$ representing the new capacity of the buffer including the size of the element to be appended, is greater than the buffer's maximum fixed size $\mathtt{n}$. In this case, the original buffer is returned. The other rule VAppend handles the case when there is enough free space to append an element after the last inserted element in the buffer, in which the updated buffer including this new element is returned.

When it comes to iteration, the two rules VIterate and VIterate-Empty handle similar cases as above, except care about whether the buffer is empty, rather than full. In the case that the buffer is empty, the rule VIterate-Empty evaluates the initial provided value of the accumulator `acc` and returns that value. Otherwise, for each element in the buffer the mapping function `f` is applied to produce a new accumulated value $\mathtt{acc}'$. If at any point after each application of `f` the termination function `t` evaluates to `true`, iteration is stopped. After iteration has stopped, either by termination or reaching the end of the elements in the buffer, the final accumulated value $\mathtt{acc}'$ is returned.

Chapter 4

Conclusion

Overall, two main contributions to COGENT were made as a result of this thesis – endianness annotations and the design of a buffer type that stores dependently sized structures.

As a result of the new endianness annotation extension, less C code has to be written when marshalling data into a format suitable for storage or transmission, for example across a network protocol. Moreover, one of DARGENT's plans for future work was the extension to support wire formats, allowing the layout of data to be described as it is transmitted over a network link [3]. Endianness annotations help to achieve this goal, by allowing specification of the byte level endianess used for in-memory data structures.

In a similar vein, one of COGENT's plans for future work was better supporting more exotic data representations, including "dynamically sized objects" [5]. Whilst our buffer type and its dependently sized structures were not completely implemented in the COGENT compiler, the work done surrounding their design and formalisation will provide value to future work regarding the implementation of the buffer, its dependently sized elements and its various operations.

4.1 Future Work

As the implementation of dependent sized structures was not finalised, a large part of the future work for this thesis is completing the compiler implementation of these structures. This includes the addition of the numerous buffer operations, as well as code generation for the buffer and dependent sized structures. Moreover, more operations could be defined on the buffer type to enable greater control over manipulation of the buffer's elements.

In terms of memory safety, currently most safety properties are ensured via dynamic checks due to the fact that the size of each dependent sized structure is not known at compile time. A key portion of the future work on these structures is to devise a way to certify memory safety properties statically, possibly using a strategy such as symbolic arithmetic to compute the sizes at compile time.

Furthermore, as only a pen and paper formalisation of the typing rules and dynamic semantics was presented in this thesis, eventually a more structured proof using Isabelle/HOL will be required once the implementation of these structures has been finalised.

Bibliography

[1] Edwin Brady. *Programming in Idris: A Tutorial*. 2012.

[2] Peter Dybjer. An introduction to programming and proving in agda. Unpublished, 01 2018.

[3] Liam O'Connor, Zilin Chen, Partha Susarla, Christine Rizkallah, Gerwin Klein, and Gabriele Keller. Bringing effortless refinement of data layouts to Cogent. In *ISoLA*, 2018.

[4] Dave Poirier. *The Second Extended File System*. 2001.

[5] Liam O'Connor. *Type Systems for Systems Types*. PhD thesis, University of New South Wales. Computer Science & Engineering, 2019.

[6] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 89102, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342193. doi: 10.1145/2951913.2951940. URL `https://doi.org/10.1145/2951913.2951940`.

[7] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 16, page 175188, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340915. doi: 10.1145/2872362.2872404. URL `https://doi.org/10.1145/2872362.2872404`.

[8] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3540433767.

[9] Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. A framework for the automatic formal verification of refinement from Cogent to C. pages 323–340, 08 2016. ISBN 978-3-319-43143-7. doi: 10.1007/978-3-319-43144-4_20.

[10] Mohit Arora. *Handling Endianness*, pages 155–168. Springer New York, New York, NY, 2012. ISBN 978-1-4614-0397-5. doi: 10.1007/978-1-4614-0397-5_7. URL https://doi.org/10.1007/978-1-4614-0397-5_7.

[11] Information Sciences Institute University of Southern California. Internet protocol. RFC 791, IETF, 09 1981. URL https://tools.ietf.org/rfc/rfc791.txt.

[12] Microsoft Corporation. *Server Message Block (SMB) Protocol*. 2018.

[13] Godmar Back. Datascript - a specification and scripting language for binary data. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, GPCE 02, page 6677, Berlin, Heidelberg, 2002. Springer-Verlag. ISBN 3540442847.

[14] Wilfred Springer. Preon introduction. http://www.scribd.com/doc/8128172/Preon-Introduction, 09 2008.

[15] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. πσ: Dependent types without the sugar. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming*, pages 40–55, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-12251-4.

[16] Finn Teegen. Idris: A functional programming language with dependent types. In *Programmiersprachen und Übersetzerkonstruktion Seminar*, 2015.