




High-level Views on Low-level Representations

Iavor S. Diatchki, Mark P. Jones, Rebekah Leslie

OGI School of Science and Engineering, OHSU
Portland State University

*This research is supported in part by the National Science Foundation
Advanced Programming Languages for Embedded Systems.*



Introduction



- Algebraic datatypes promote a high-level view of data that hides many low-level implementation details.
- Many applications require the use of *bitdata*: data that is stored in bit fields and accessed as part of a single machine word.
- We explain how a modern functional language like ML or Haskell can be extended with mechanisms for specifying and using bitdata.



Example: PCI Device Addresses



- PCI devices are identified with a 16-bit address:



- Example: function 3, of device 6, on bus 1:
 - hex: 0x0133
 - binary: 00000001 00110 011





Example: Timeouts in L4

now	<table border="1"><tr><td>0</td><td>1₍₅₎</td><td>0₍₁₀₎</td></tr></table>	0	1 ₍₅₎	0 ₍₁₀₎	= 0
0	1 ₍₅₎	0 ₍₁₀₎			
period	<table border="1"><tr><td>0</td><td>e₍₅₎</td><td>m₍₁₀₎</td></tr></table>	0	e ₍₅₎	m ₍₁₀₎	= $2^e m \mu s$
0	e ₍₅₎	m ₍₁₀₎			
never	<table border="1"><tr><td colspan="3">0₍₁₆₎</td></tr></table>	0 ₍₁₆₎			= ∞
0 ₍₁₆₎					



Example: Instructions for the Z80

			n	r	s	
00	$s_{(3)}$	$r_{(3)}$	SHIFT s, r	000	B	RLC
01	$r_{(3)}$	$n_{(3)}$	BIT r, n	001	C	RRC
10	$r_{(3)}$	$n_{(3)}$	RES r, n	010	D	RL
11	$r_{(3)}$	$n_{(3)}$	SET r, n	011	E	RR
				100	H	SLA
				101	L	SRA
				110	(HL)	—
				111	A	SRL

Observations about Bitdata

- Often the bit patterns that are used in bitdata encodings are determined by *external* specifications:
 - Software: the ABI for an OS kernel.
 - Hardware: the register layouts of a device.
- A common pattern in many encodings:
 - use some bits to store *values*,
 - use some bits as *tags*, to distinguish values.
- Similar to sum-of-products user defined types.

The Perils of Bit Twiddling

- Common bit-twiddling techniques obfuscate the operations, e.g. $(w \gg 24) \& 0xff$
 - Programs are hard to read, debug, and modify.
 - More difficult for a compiler to generate good code.
 - May result in loss of type information: all data is treated as a word
- We want programmers to avoid artificial encoding.

Algebraic Datatypes

```
data PCI          = PCI { bus :: Bit8
                        , dev  :: Bit5
                        , fun  :: Bit3 }
data Timeout     = Now | Period Int | Never
data Op          = Shift ShiftOp Reg
                  | Bit  Reg Bit3
                  | Res  Reg Bit3
                  | Set  Reg Bit3
```

- Marshaling overheads
- Writing parsing/unparsing functions is tedious and error prone.

External IDLs



- Define bitdata with an external IDL
- Use tools to generate bit twiddling functions automatically
- Weak integration with host language:
 - weaker typing,
 - no pattern matching,
 - harder to generate efficient code.



Our Approach



- Built-in language support:
 - stronger typing,
 - more static checks,
 - potentially more optimization opportunities.
- Start with a small, standard functional language.
- Two language extensions:
 - Support for basic bit-level manipulation.
 - Defining new bitdata types that are distinguished from their underlying representation.



Example: PCI Device Address

```
bitdata PCI
  = PCI { bus :: Bit 8
         , dev :: Bit 5
         , fun :: Bit 3
         }
```

- New keyword `bitdata`
- A type for bit vectors `Bit`
- Left-most field in most significant position

Example: Timeouts in L4



```
data Timeout
```

```
= Period { e :: Bit 5, m :: Bit 10 }
```

```
| Now
```

```
| Never
```



Example: Timeouts in L4



```
bitdata Timeout
```

```
= Period { e :: Bit 5, m :: Bit 10 }  
  as B0 # e # m
```

```
| Now  
  as B0 # (1 :: Bit 5) # (0 :: Bit 10)
```

```
| Never  
  as 0
```



Example: Instruction for the Z80

```
bitdata Ops
  = Shift { op :: ShOp, reg :: Reg }
    as B00 # op # reg
    ...
```

```
bitdata ShOp
  = RLC as B000 | RRC as B001
  | RL  as B010 | RR  as B011
  | SLA as B100 | SRA as B101
  | SRL as B111
```

```
bitdata Reg = ...
```

Example: Using Bitdata

```
toMicro Now      = Just 0
toMicro Never    = Nothing
toMicro (Period { m = m, e = e })
                = Just (2e * m)
```

```
-- 10000 micro seconds
timeout          = Period { m = 625, e = 4 }
```

Type System



- Hindley-Milner with qualified types:

$$\kappa = \mathbb{N} \mid * \mid \kappa \rightarrow \kappa$$

$$\sigma = \forall \bar{\alpha}. \bar{\pi} \Rightarrow \tau$$

$$\tau = \tau \tau \mid \alpha \mid c$$

- \mathbb{N} is a kind for natural numbers: 0, 1, ...

- Example:

`addBits ::`

`Width a ⇒ Bit a → Bit a → Bit a`

- Using *improvement* is essential to infer nice types



Bit Vectors

- Type constructor $Bit :: \mathbb{N} \rightarrow *$
- Predicate $Width$ restricts widths
- Usual operations: `addBits`, `orBits`, ..., etc.
- Joining and splitting bit vectors
 - (#) $:: (a + b = c)$
 $\Rightarrow Bit\ a \rightarrow Bit\ b \rightarrow Bit\ c$
- (#) also works in patterns

Example

```
aimPCI :: PCI → Bit 8 → H ()
aimPCI p (w # _)
    = outL 0xCF8
          (B10000000 # toBits p # w # B00)
```

```
outL    :: IOPort → Bit 32 → H ()
toBits  :: BitRep a n ⇒ a → Bit n
```



Bitdata is not Free

now	<table border="1"><tr><td>0</td><td>1₍₅₎</td><td>0₍₁₀₎</td></tr></table>	0	1 ₍₅₎	0 ₍₁₀₎	= 0
0	1 ₍₅₎	0 ₍₁₀₎			
period	<table border="1"><tr><td>0</td><td>e₍₅₎</td><td>m₍₁₀₎</td></tr></table>	0	e ₍₅₎	m ₍₁₀₎	= $2^e m \mu s$
0	e ₍₅₎	m ₍₁₀₎			
never	<table border="1"><tr><td colspan="3">0₍₁₆₎</td></tr></table>	0 ₍₁₆₎			= ∞
0 ₍₁₆₎					



Bitdata is not Free



now	<table border="1"><tr><td>0</td><td>1₍₅₎</td><td>0₍₁₀₎</td></tr></table>	0	1 ₍₅₎	0 ₍₁₀₎	= 0
0	1 ₍₅₎	0 ₍₁₀₎			
period	<table border="1"><tr><td>0</td><td>$e_{(5)}$</td><td>$m_{(10)}$</td></tr></table>	0	$e_{(5)}$	$m_{(10)}$	= $2^e m \mu s$
0	$e_{(5)}$	$m_{(10)}$			
never	<table border="1"><tr><td colspan="3">0₍₁₆₎</td></tr></table>	0 ₍₁₆₎			= ∞
0 ₍₁₆₎					

- Redundancy: $(m = 2, e = 0) = 2\mu s = (m = 1, e = 1)$.



Bitdata is not Free



now	<table border="1"><tr><td>0</td><td>1₍₅₎</td><td>0₍₁₀₎</td></tr></table>	0	1 ₍₅₎	0 ₍₁₀₎	= 0
0	1 ₍₅₎	0 ₍₁₀₎			
period	<table border="1"><tr><td>0</td><td>e₍₅₎</td><td>m₍₁₀₎</td></tr></table>	0	e ₍₅₎	m ₍₁₀₎	= $2^e m \mu s$
0	e ₍₅₎	m ₍₁₀₎			
never	<table border="1"><tr><td colspan="3">0₍₁₆₎</td></tr></table>	0 ₍₁₆₎			= ∞
0 ₍₁₆₎					

- Redundancy: $(m = 2, e = 0) = 2\mu s = (m = 1, e = 1)$.
- Confusion: Now and Never overlap with Period.



Bitdata is not Free



now	<table border="1"><tr><td>0</td><td>1₍₅₎</td><td>0₍₁₀₎</td></tr></table>	0	1 ₍₅₎	0 ₍₁₀₎	= 0
0	1 ₍₅₎	0 ₍₁₀₎			
period	<table border="1"><tr><td>0</td><td>e₍₅₎</td><td>m₍₁₀₎</td></tr></table>	0	e ₍₅₎	m ₍₁₀₎	= $2^e m \mu s$
0	e ₍₅₎	m ₍₁₀₎			
never	<table border="1"><tr><td colspan="3">0₍₁₆₎</td></tr></table>	0 ₍₁₆₎			= ∞
0 ₍₁₆₎					

- Redundancy: $(m = 2, e = 0) = 2\mu s = (m = 1, e = 1)$.
- Confusion: Now and Never overlap with Period.
- Junk: no value has 1 in its most significant bit.



Confusion and Views



```
bitdata DWord
= Int32    { val      :: Bit 32  }

| VirtAddr { dir      :: Bit 10
              , tab    :: Bit 10
              , offset :: Bit 12  }

| Bytes    { b3      :: Bit 8
              , b2    :: Bit 8
              , b1    :: Bit 8
              , b0    :: Bit 8  }
```



Static Analysis

- Analyze bitdata declarations to help programmers
- We use BDDs to represent sets of bit-patterns

`confusion C D = cover C ∩ cover D`

`junk T = ~ (∪ {cover C | C ← ctrs T})`

- Should also analyze function definitions
 - Polymorphism makes analysis more conservative

`f :: Width a => Bit a -> Bit 1`

`f 0 = B1`

`f 1 = B0`

Example



Warning: The type Timeout contains junk:

```
*** 1~~~~~
```

Warning: Constructors Now and Period
of type Timeout overlap

```
*** 0000010000000000
```

Warning: Constructors Period and Never
of type Timeout overlap

```
*** 0000000000000000
```



Related work

- Language support:
 - C/C++, Ada
 - Cryptol
 - BlueSpec
- IDLs:
 - SLED [Ramsey, Fernández '97]
 - PADS [Fisher, Gruber '05]
 - DataScript [Back '02]
 - Devil [Réveillère '01]

Summary



- Bitdata is common in systems programming.
- Two language extensions for working with bitdata:
 - Working with bit vectors
 - User defined bitdata
- Bitdata manipulation is similar to working with ADTs.
- Bitdata manipulation is type-safe and efficient.

The End

