Strongly Typed Memory Areas

Programming Systems-Level Data Structures in a Functional Language

Iavor S. Diatchki[1] Mark P. Jones[2]

[1]OGI School of Science & Engineering (OHSU)

[2]Portland State University

Haskell Workshop 2006

Introduction

- Modern languages help programmers . . .
  - Module systems
  - Type systems
  - Automatic storage management
- . . . but are rarely used for system programming
  - Some non-technical reasons?
  - There is a genuine problem:

It is hard to encode and manipulate certain data structures in Haskell/ML.

Introduction

- Modern languages help programmers . . .
  - Module systems
  - Type systems
  - Automatic storage management
- . . . but are rarely used for system programming
  - Some non-technical reasons?
  - There is a genuine problem:

It is hard to encode and manipulate certain data structures in Haskell/ML.

Introduction

- Modern languages help programmers . . .
  - Module systems
  - Type systems
  - Automatic storage management
- . . . but are rarely used for system programming
  - Some non-technical reasons?
  - There is a genuine problem:

It is hard to encode and manipulate certain data structures in Haskell/ML.

Examples

- IA32
  - Page tables and directories
  - Interrupt and segment descriptor tables
  - Task state segments
  - Hardware exception contexts
- L4 kernel
  - Kernel information page
  - User thread control blocks
- Others. . .

Example: Programming the Display

- Program text mode display for a PC
- A region of memory at physical address `0xB8000`
- Conceptually 25 rows, with 80 columns each
- Each entry is a character and attribute

Using the Haskell FFI

```
scr :: Ptr Word8
scr = nullPtr ‘plusPtr‘ 0xB8000
```

```
writeChar :: Int → Int → Word8 → Char → IO ()
writeChar row col attr c
  | isInvalidPosition row col = fail "error"
  | otherwise
    = pokeElemOff scr ((row * 80) + col) w
  where
    w :: Word16
    w = (fromIntegral attr ‘shiftL‘ 8)
                  .|. fromIntegral (ord c)
```

Using the Haskell FFI


```
scr :: Ptr Word8
scr = nullPtr ‘plusPtr‘ 0xB8000
```

```
writeChar :: Int → Int → Word8 → Char → IO ()
writeChar row col attr c
  | isInvalidPosition row col = fail "error"
  | otherwise
    = pokeElemOff scr ((row * 80) + col) w
  where
  w :: Word16
  w = (fromIntegral attr ‘shiftL‘ 8)
                .|. fromIntegral (ord c)
```

Using the Haskell FFI


```
scr :: Ptr Word16
scr = nullPtr ‘plusPtr‘ 0xB8000
```

```
writeChar :: Int → Int → Word8 → Char → IO ()
writeChar row col attr c
  | isInvalidPosition row col = fail "error"
  | otherwise
    = pokeElemOff scr ((row * 80) + col) w
  where
  w :: Word16
  w = (fromIntegral attr ‘shiftL‘ 8)
                .|. fromIntegral (ord c)
```

Observations

- It is not pretty
- It can be unsafe
- It can be error prone
- . . . but it works!

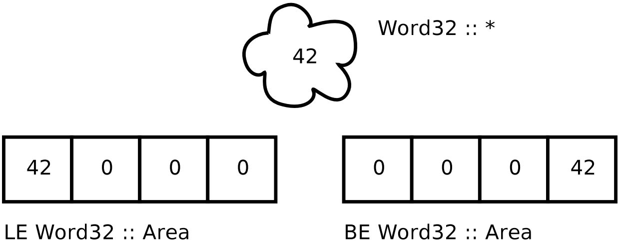
Our goal is to make it easier to program such things.

Values vs. Memory Areas

- The problem:
  - Values in a functional language are abstract.
  - Systems programs need to manipulate memory.
- The solution:
  - Introduce types to describe *memory areas*.
  - These types are of a new kind, `Area`.
  - Rigid representation, suitable for communication with external programs/devices.

Stored Values

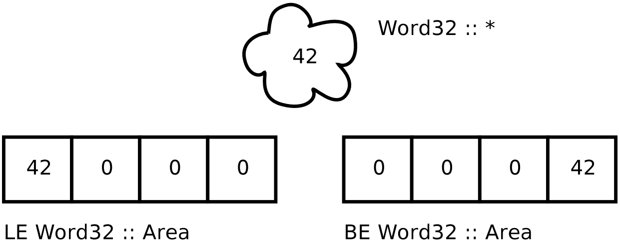
- Areas for *explicit representations* of *abstract* values:



- Two new type constructors to define basic areas:
  `BE,LE :: * → Area`
- For native order, use a (platform specific) synonym:
  type `Stored = ...`

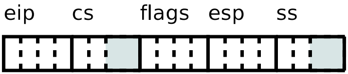
Stored Values

- Areas for *explicit representations* of *abstract* values:



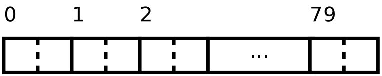
- Two new type constructors to define basic areas:
  `BE,LE :: * → Area`
- For native order, use a (platform specific) synonym:
  type `Stored = ...`

Structures




```
struct IRet where
  eip     :: Stored Word32
  cs      :: Stored SegDescr
           ; Stored Word16
  flags   :: Stored Word32
  esp     :: Stored Word32
  ss      :: Stored SegDescr
           ; Stored Word16
```

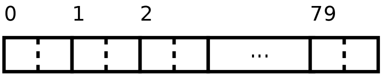
Arrays



`Array 80 (Stored Word16)`

- The type constructor:
  `Array :: Nat → Area → Area`
- The size of the array is a type of kind `Nat`
  `0,1,2,... :: Nat`

Arrays



`Array 80 (Stored Word16)`

- The type constructor:

  `Array :: Nat → Area → Area`

- The size of the array is a type of kind `Nat`

  `0,1,2,... :: Nat`

References

- Manipulate memory areas via references:

  `ARef :: Nat → Area → *`

  - Alignment (in bytes)
  - Description of target area

- Example:

  `ARef 4096 Page`

- When alignment is not important:

  `type Ref = ARef 1`

References

- Manipulate memory areas via references:

  ```
  ARef :: Nat → Area → *
  ```

  - Alignment (in bytes)
  - Description of target area

- Example:

  ```
  ARef 4096 Page
  ```

- When alignment is not important:

  ```
  type Ref = ARef 1
  ```

Marshaling

- Relate areas to the type of value they contain:

```
class ValIn r t | r ⇝ t where
  readRef   :: ARef a r → IO t
  writeRef  :: ARef a r → t → IO ()
```

- Instances for areas that contain stored values:

```
instance ValIn (LE Word32) Word32
instance ValIn (BE Word32) Word32
```

- No instances for values that do not have explicit representation (e.g., `Int → Int`)

Marshaling

- Relate areas to the type of value they contain:

```
class ValIn r t | r ⇝ t where
  readRef   :: ARef a r → IO t
  writeRef  :: ARef a r → t → IO ()
```

- Instances for areas that contain stored values:

```
instance ValIn (LE Word32) Word32
instance ValIn (BE Word32) Word32
```

- No instances for values that do not have explicit representation (e.g., `Int → Int`)

Marshaling

- Relate areas to the type of value they contain:

```
class ValIn r t | r ⇝ t where
  readRef  :: ARef a r → IO t
  writeRef :: ARef a r → t → IO ()
```

- Instances for areas that contain stored values:

```
instance ValIn (LE Word32) Word32
instance ValIn (BE Word32) Word32
```

- No instances for values that do not have explicit representation (e.g., `Int → Int`)

Size

- Operations that depend on the size of an area:

```
class SizeOf r (n::Nat) | r ⇝n where
  sizeOf  :: ARef a r → Int
  memCopy :: ARef a r → ARef b r → IO ()
  memZero :: ARef a r → IO ()
```

- All valid area types have an instance (automatically derived for structures):

```
instance SizeOf (LE Word32) 4
instance SizeOf (Array n r) (n * SizeOf r)
```

Size

- Operations that depend on the size of an area:

```
class SizeOf r (n::Nat) | r ⇝n where
  sizeOf  :: ARef a r → Int
  memCopy :: ARef a r → ARef b r → IO ()
  memZero :: ARef a r → IO ()
```

- All valid area types have an instance (automatically derived for structures):

```
instance SizeOf (LE Word32) 4
instance SizeOf (Array n r) (n * SizeOf r)
```

Size

- Operations that depend on the size of an area:

```
class SizeOf r (n::Nat) | r ⇝ n where
  sizeOf  :: ARef a r → Int
  memCopy :: ARef a r → ARef b r → IO ()
  memZero :: ARef a r → IO ()
```

- All valid area types have an instance (automatically derived for structures):

```
instance SizeOf (LE Word32) 4
instance SizeOf (Array n r) (n * SizeOf r)
```

Size

- Operations that depend on the size of an area:

```
class SizeOf r (n::Nat) | r ⇝ n where
  sizeOf  :: ARef a r → Int
  memCopy :: ARef a r → ARef b r → IO ()
  memZero :: ARef a r → IO ()
```

- All valid area types have an instance (automatically derived for structures):

```
instance SizeOf (LE Word32) 4
instance (n * SizeOf r = y)
           ⇒ SizeOf (Array n r) y
```

Size

- Operations that depend on the size of an area:

```
class SizeOf r (n::Nat) | r ⇝n where
  sizeOf  :: ARef a r → Int
  memCopy :: ARef a r → ARef b r → IO ()
  memZero :: ARef a r → IO ()
```

- All valid area types have an instance (automatically derived for structures):

```
instance SizeOf (LE Word32) 4
instance (n * SizeOf r = y)
           ⇒ SizeOf (Array n r) y
```

Size

- Operations that depend on the size of an area:

```
class SizeOf r (n::Nat) | r ⇝ n where
   sizeOf  :: ARef a r → Int
   memCopy :: ARef a r → ARef b r → IO ()
   memZero :: ARef a r → IO ()
```

- All valid area types have an instance (automatically derived for structures):

```
instance SizeOf (LE Word32) 4
instance (SizeOf r x, n * x = y)
           ⇒ SizeOf (Array n r) y
```

Size

- Operations that depend on the size of an area:

```
class SizeOf r (n::Nat) | r ⇝ n where
  sizeOf  :: ARef a r → Int
  memCopy :: ARef a r → ARef b r → IO ()
  memZero :: ARef a r → IO ()
```

- All valid area types have an instance (automatically derived for structures):

```
instance SizeOf (LE Word32) 4
instance (SizeOf r x, n * x = y)
         ⇒ SizeOf (Array n r) y
```

Accessing components (simp.)

- Pure operations, perform pointer arithmetic.
- Structures use generated projection functions:

  struct `Two` **where** `{ fst :: A; snd :: B }`

  ```
  (.fst) :: Ref Two → Ref A
  (.snd) :: Ref Two → Ref B
  ```
- Arrays use an indexing function:

  ```
  (@) :: Ref (Array n r) → Ix n → Ref r
  ```

Accessing components

- To access a component we need the sizes of the areas before it:
  - To compute the offset
  - To compute the alignment

```
(@) :: ARef a (Array n r)
    → Ix n
    → ARef (GCD a (SizeOf r)) r
```

- Example:

```
(@) :: ARef 4 (Array 32 (BE Word16))
    → Ix 32
    → ARef 2 (BE Word16)
```

Accessing components

- To access a component we need the sizes of the areas before it:
  - To compute the offset
  - To compute the alignment

```
(@) :: ARef a (Array n r)
    → Ix n
    → ARef (GCD a (SizeOf r)) r
```

- Example:

```
(@) :: ARef 4 (Array 32 (BE Word16))
    → Ix 32
    → ARef 2 (BE Word16)
```

Index Values

- The type `Ix n` denotes the sub-range $[0..n)$:

```
class Index n where
  toIx   :: Int → Ix n
  fromIx :: Ix n → Int
  minIx  :: Ix n
  maxIx  :: Ix n
  addIx  :: Int → Ix n → Maybe (Ix n)
```

- Higher-level control structures:

```
forEachIx :: Index n
          ⇒ (Ix n → IO a) → IO [a]
```

Example: Video RAM

```
type Rows     = 25
type Cols     = 80

type Row      = Array Cols (Stored Word16)
type Screen   = Array Rows Row


cls    :: Ref Screen → IO ()
cls scr = forEachIx_ (λ row →
            forEachIx_ (λ col →
              writeRef (scr @ row @ col) blank
            )
         )
```

Area Declarations

- Where do references come from?

  area *name* [**in** *region*] :: *type*

- Examples:

```
area pdir  :: ARef 4K (Array 1024 PDE)
area count :: Ref (Stored Int)
area videoRAM in videoRAM :: Ref Screen
```

- Compiler allocates and initializes suitably sized, aligned, and non-overlapping space for area declarations without an explicit region annotation
- Named regions configured using compile-time options

Related Work

- Lots!
- Programming Languages: C, Cyclone, Modula-3, Ada, Erlang, many others.
- FFIs
  - Data interoperability (Moby, SML/NJ)
  - Do we need C? Perhaps Haskell + memory areas + asm is enough. . .
- IDLs: DataScript, PADS
- Fancy types: sized types, singleton types + existentials, dependent types

Summary

- Kinds distinguish data with concrete and abstract representation
- Types enforce invariants on data:
  - References access areas
  - Index types ensure safe array access
  - Alignment restricts placement of memory areas
- A simple notation makes working with functional dependencies prettier