IMPLEMENTING TRUE SEPARATE COMPILATION
THE BLECH MODULE SYSTEM

FRIEDRICH GRETZ
FRANZ-JOSEF GROSCH

SYNCHRON, NOVEMBER 2020
- ONLINE-

BOSCH

The Blech synchronous language Blech
Update since Synchron ’19

blog articles,
documentation

- Visit us online: www.blech-lang.org

Blech → C
compiler

- Check out the implementation at https://github.com/boschresearch/blech
  - Compiles on all platforms! (Linux, Mac, Windows)
  - A VS Code plug-in can be found at https://github.com/boschresearch/blech-tools/releases

IDE, installer, ...

- Follow us at https://twitter.com/BlechLanguage
- Get in touch with us by email or https://blech-lang.slack.com

Talking about separate compilation in Blech

Blech, imperative synchronous programming!

Friedrich Gretz
Robert Bosch GmbH
Corporate Research
Renningen, Germany
Friedrich.Gretz@de.bosch.com

Franz-Josef Grosch
Robert Bosch GmbH
Corporate Research
Renningen, Germany
Franz-Josef.Grosch@de.bosch.com

Forum on specification & Design Languages 2018

- introduction of black-box activities,
- causality only based on input/output interface

FORUM ON SPECIFICATION AND DESIGN LANGUAGES – FDL 2020 1

Synchronized Shared Memory and Procedural Abstraction: Towards a Formal Semantics of Blech

F. Gretz and F-J. Grosch (Bosch Corporate Research) and M. Mendler and S. Scheele (Bamberg University)

Forum on specification & Design Languages 2020

- provide a formal semantics for this kind of procedural abstraction

BOSCH

Talking about separate compilation in Blech

- **Today:**
  - a (synchronous) program is not just one file collecting all activities
  - software architecture, separation of concerns, reuse of “packages” or “libraries” of software

➔ modules

- Organising code in files and collections thereof is nothing new: e.g. Java classes + JAR
- **Engineering** task:
  - what granularity of name spaces and access rights do we need?
  - how does this integrate with C?
  - how does this integrate with a synchronous language and causality checking?

Gretz, Grosch | 2020-11-26

Talking about separate compilation in Blech

Running example: RingBuffer

- no external dependency management,
- no local imports,
- no shadowing

```
import rb "data_structures/ringbuffer"

module exposes SlidingAverage

param Threshold: nat32 = 10000

activity SlidingAverage (value: nat32) (average: nat32)
   var buf: rb.RingBuffer = rb.initialise()
   repeat
      if value <= Threshold then
         rb.push(value)(buf)
      end
      average = rb.average(buf)
      await true
   end
end
```

upstream module

export to downstream client-code

- simple visibility properties
- gathered in one declaration at the beginning

access through given name rb

Gretz, Grosch | 2020-11-26

Module system design challenges
1. Mapping names

Blech – hierarchical name spaces

data_structures

ringbuffer.blc

push (value: nat32) (buf: RingBuffer)

…

slidingaverage.blc

rb

SlidingAverage (value: nat32) (average: nat32)

C – flat global name space

```
void blc_data_structures_ringbuffer_push (
  blc_nat32 value,
  struct blc_data_structures_ringbuffer_RingBuffer * average
  )

blc_slidingaverage_SlidingAverage (
  blc_nat32 value,
  blc_nat32 * average
  )
```


Gretz, Grosch | 2020-11-26

BOSCH

Module system design challenges
2. Compiling dependencies automatically

Blech – hierarchical name spaces

data_structures

ringbuffer.blc

push (value: nat32) (buf: RingBuffer)

…

slidingaverage.blc

rb

SlidingAverage (value: nat32) (average: nat32)

blechc slidingaverage.blc

blechc data_structures/ringbuffer.blc

blech/data_structures/ringbuffer.**c**

blech/data_structures/ringbuffer.**h**

blech/data_structures/ringbuffer.**blh**

blech/slidingaverage.**c**	C code
blech/slidingaverage.**h**	C header
blech/slidingaverage.**blh**	Blech interface

Gretz, Grosch | 2020-11-26

BOSCH

Module system design challenges
3. Separate compilation, relying on interfaces

Blech sources available

blechc slidingaverage.blc

↓

blechc data_structures/ringbuffer.**blc**

↓

blech/data_structures/ringbuffer.**c**

blech/data_structures/ringbuffer.**h**

blech/data_structures/ringbuffer.**blh**

↓

blech/slidingaverage.**c**

blech/slidingaverage.**h**

blech/slidingaverage.**blh**

Blech header + C source

blechc slidingaverage.blc

↓

blechc blech/data_structures/ringbuffer.**blh**

↓

blech/slidingaverage.**c**

blech/slidingaverage.**h**

blech/slidingaverage.**blh**

blech/data_structures/ringbuffer.**c**

blech/data_structures/ringbuffer.**h**

Blech header + C object

blechc slidingaverage.blc

↓

blechc blech/data_structures/ringbuffer.**blh**

↓

blech/slidingaverage.**c**

blech/slidingaverage.**h**

blech/slidingaverage.**blh**

blech/data_structures/ringbuffer.**o**

blech/data_structures/ringbuffer.**h**

Gretz, Grosch | 2020-11-26

Module system design challenges
3. Separate compilation, relying on interfaces

Blech sources available

- all sources available to the programmer

Blech header + C source

- API of imported Blech module is available
- Blech implementation is secret
- generated C code is available

Blech header + C object

- API of imported Blech module is available
- Blech implementation is secret
- C code is secret

Gretz, Grosch | 2020-11-26

Module system design challenges
3. Separate compilation, relying on interfaces

Implementation: ringbuffer.**blc**

```
module exposes initialise, push, average

const Size: nat8 = 10

struct RingBuffer
   var buffer: [Size]nat32
   var nextIndex: nat8
   var count: nat8
end

function initialise () returns RingBuffer
   return { }
end
```

```
function push (value: nat32) (rb: RingBuffer)
   rb.buffer[rb.nextIndex] = value
   rb.nextIndex = rb.nextIndex + 1
   if rb.count = Size then // ringbuffer ist completely filled
      rb.nextIndex = rb.nextIndex % Size
   else
      rb.count = rb.count + 1
   end
end

function average (rb: RingBuffer) returns nat32
   var idx: nat8 = 0
   var avg: nat32 = 0
   while idx < rb.count do
      avg = avg + rb.buffer[idx]
   end
   return avg / rb.count
end
```


Gretz, Grosch | 2020-11-26

Module system design challenges
3. Separate compilation, relying on interfaces

Interface: ringbuffer.**blh** (generated by blechc)

```
signature

type RingBuffer

function initialise () returns RingBuffer

function push (value: nat32) (rb: RingBuffer)

function average (rb: RingBuffer) returns nat32
```

the type is used by an exposed function and therefore is ***implicitly*** exposed

functions were ***explicitly*** exposed

the module constant “Size” is not exposed at all and unknown outside the ringbuffer module

Gretz, Grosch | 2020-11-26

Module system design challenges
3. Separate compilation, relying on interfaces

What to do with singletons?

```
module exposes Monitor

@[CFunction (binding = "wifi_is_online()", header = "wifi.h")]
extern singleton function wifiIsOnline () returns bool

activity Monitor () (leds: LEDs)
   repeat
      leds.wifiLed = wifiIsOnline()
      await true
   end
end
```

```
signature

type LEDs

singleton wifiIsOnline

singleton wifiIsOnline activity Monitor (leds: LEDs)
```

Monitor must not be called concurrently with anything that uses wifiIsOnline (including itself).

Gretz, Grosch | 2020-11-26

Module system design challenges
White-box unit testing

```
internal import rb "ringbuffer"

@[EntryPoint]
activity TestPush ()
   var buf: rb.RingBuffer = rb.initialise()
   var i: nat8 = 0
   repeat
      assert buf.nextIndex < rb.Size
      assert buf.nextIndex == i % rb.Size
      assert buf.count >= 0
      assert buf.count <= rb.Size

      rb.push(42)(buf) // the value is irrelevant
      i = i + 1

      if i < rb.Size then assert buf.count == i
      else assert buf.count == rb.Size end
      await true
   until i == 255 end
end
```

Internal import

- only possible if Blech code is available
- allows to separate testing code from product code

Gretz, Grosch | 2020-11-26

Design pragmatics
Layered Architecture

data_structures

ringbuffer.blc

slidingaverage.blc

main.blc

```
import sa "slidingaverage"

@[EntryPoint]
activity Main (sensor: nat32) (sensorAverage: nat32)
   run sa.SlidingAverage(sensor)(sensorAverage)
end
```

- absence of import cycles checked automatically
- differentiate modules and programs
  - programs contain an entry point and cannot be imported, no blh is generated
- separate testability of each module (or layer)

Summary

- Module = file
- Everything visible within file (or within internal import)
- A declaration is either exposed or not (opaque types / singletons automatically exposed if necessary)
- Generation of a “Blech API” (*.blh) which
  - does not reveal implementation details
  - suffices for downstream code generation
- Layered architecture
- Modules could be wrapped to packages (cf. Blog)
- Modules organise name spaces but they do not address generic data structures → future work
- Implementation work in progress, planned release end of 2020.

Gretz, Grosch | 2020-11-26

Design pragmatics
Syntax

- dependencies clearly visible in code instead of external project configuration files
- simple visibility properties
- gathered in one declaration at the beginning

Design pragmatics
White-box testing

- recap:
  - FDL’18 explained the principle of black-box activities, causality only based on input/output interface
  - FDL’20 provide a formal semantics for this kind of procedural abstraction
- Today:
  - a (synchronous) program is not just one file collecting all activities
  - software architecture, separation of concerns, reuse of “packages” or “libraries” of software
  - ➔ collect types, activities, functions into a module (= file)
  - collect modules into library
  - this is common place for standard languages (Java JARs, Rust crates, ...)
  - our main challenges:
    - mapping to C where all names are global without producing clashes
    - compiling everything that a module/program needs automatically, unlike C/C++ where you need to specify all dependencies in a makefile manually
    - lift black-box approach to modules, meaning modules may be precompiled and the compiler relies only on the module’s interface which we call “Signature”

- Design choices in detail
  - name mangling: encode folder structure, module name into the name of every static element
  - exposing (types) explicitly vs implicitly vs not-at-all
    - within module scope (file scope) everything is visible
    - functions, activities, constants are either exposed to the client or not
    - types which are not explicitly exposed but are required by the parameter list of exposed functions/activites are exported as abstract types (i.e. just names)
    - for unit testing, special white-box import to keep implementation file and test file separate (source code needed, signature file insufficient, but that is given for testing)
  - singletons
    - activities which access global (external) memory become singleton in Blech, activities calling singletons become singletons themselves
    - in order to causality check such singletons (prevent concurrent use to itself) the signature must contain the “reason” i.e. names for why they are singleton

- Lessons learnt? or why are we telling you this??
  - engineering challenge
  - causality analysis with module signatures, no global analysis required
  - separate compilation with precompiled sources for synchronous lang
  - make sure non-exported elements do not leak to the outside
  - KISS
- Status of implementation?
- Future work: generics (orthogonal to modules)

BOSCH