Using the Deep-Embedded ForSyDe
The ForSyDe PhD Course

Seyed Hosein Attarzadeh Niaki
Based on the ForSyDe tutorial by Alfonso Acosta

## Shallow vs. Deep Embedding

<table>
<thead>
<tr>
<th>Shallow Embedded ForSyDe</th>
<th>Deep Embedded ForSyDe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faster prototyping</td>
<td>Captures the structure of the system</td>
</tr>
<tr>
<td>Supports a wider subset of the host language</td>
<td>Allows synthesis and connection to other tools (VHDL, GraphML, ...)</td>
</tr>
<tr>
<td>Supports more data types</td>
<td>Is restricted to the SY MoC</td>
</tr>
<tr>
<td>Supports more Models of Computation (SY, UT, CT, DE, ...)</td>
<td>Fewer constructs and data types (depending on the backend)</td>
</tr>
<tr>
<td>Limited to simulation</td>
<td>Needs unstable language extensions</td>
</tr>
</tbody>
</table>
Shallow and Deep Signals

**Shallow signals are data streams, isomorphic to Haskell lists**

- `ForSyDe.Shallow.Signal`
  - `data Signal a = NullS | a :- Signal a`

**Deep Signals are abstract types that represent edges of the system graph**

- `ForSyDe.Signal`
  - `newtype Signal a = ...`

System Structure View

- **Shallow models**
  - Processes & process networks

- **Deep models**
  - Processes, process networks, systems and components
A Simple Incrementer – Function

{-# LANGUAGE TemplateHaskell #-}
module Plus1 where

import ForSyDe
import Data.Int (Int32)

-- A process function which adds one to its input
addOnef :: ProcFun (Int32 -> Int32)
addOnef = $(newProcFun [d|addOnef :: Int32 -> Int32
                        addOnef n = n + 1 |])
module Plus1 where

import ForSyDe
import Data.Int (Int32)

-- A process function which adds one to its input
addOnef :: ProcFun (Int32 -> Int32)
addOnef = $(newProcFun [d|addOnef :: Int32 -> Int32
    addOnef n = n + 1 |])
A Simple Incrementer – Function

{-# LANGUAGE TemplateHaskell #-}

module Plus1 where

import ForSyDe
import Data.Int (Int32)

-- A process function which adds one to its input
addOnef :: ProcFun (Int32 -> Int32)
addOnef = $ (newProcFun [d|addOnef :: Int32 -> Int32
  addOnef n = n + 1 |])

Captures the abstract syntax tree of the function
A Simple Incrementer – Process

-- A process which uses addOnef
plus1Proc :: Signal Int32 -> Signal Int32
plus1Proc = mapSY "plus1Proc" addOnef

Similar to shallow processes
A Simple Incrementer – Process

-- A process which uses addOnef
plus1Proc :: Signal Int32 -> Signal Int32
plus1Proc = mapSY "plus1Proc" addOnef
A Simple Incrementer – System

-- System definition associated to the process
plus1SysDef :: SysDef (Signal Int32 -> Signal Int32)
plus1SysDef = newSysDef plus1Proc "plus1"
                ["inSignal"] ["outSignal"]

• A system can be simulated using the simulate function

A Simple Incrementer – System

-- System definition associated to the process
plus1SysDef :: SysDef (Signal Int32 -> Signal Int32)
plus1SysDef = newSysDef plus1Proc "plus1"
                ["inSignal"] ["outSignal"]

• A system can be simulated using the simulate function
A Simple Incrementer – System

-- System definition associated to the process
plus1SysDef :: SysDef (Signal Int32 -> Signal Int32)
plus1SysDef = newSysDef plus1Proc "plus1"
["inSignal"] ["outSignal"]

• A system can be simulated using the simulate function

A Simple Incrementer – System

-- System definition associated to the process
plus1SysDef :: SysDef (Signal Int32 -> Signal Int32)
plus1SysDef = newSysDef plus1Proc "plus1"
["inSignal"] ["outSignal"]

• A system can be simulated using the simulate function

Demo!
Using Systems as Components – Doing +4 By +1+1+1+1

module AddFour where

import Plus1 (plus1SysDef)

import ForSyDe
import Data.Int (Int32)

addFourProc :: Signal Int32 -> Signal Int32
addFourProc = plus1Comp "plus1_1" .
  plus1Comp "plus1_2" .
  plus1Comp "plus1_3" .
  plus1Comp "plus1_4"

where plus1Comp id = instantiate id plus1SysDef

addFourSysDef :: SysDef (Signal Int32 -> Signal Int32)
addFourSysDef = newSysDef addFourProc "addFour" ["in1"] ["out1"]
Using Systems as Components – Doing +4 By +1+1+1+1

```haskell
module AddFour where

import Plus1 (plus1SysDef)
import ForSyDe
import Data.Int (Int32)

addFourProc :: Signal Int32 -> Signal Int32
addFourProc = plus1Comp "plus1_1" .
  plus1Comp "plus1_2" .
  plus1Comp "plus1_3" .
  plus1Comp "plus1_4"
  where plus1Comp id = instantiate id plus1SysDef

addFourSysDef :: SysDef (Signal Int32 -> Signal Int32)
addFourSysDef = newSysDef addFourProc "addFour" ["in1"] ["out1"]
```

The VHDL Back-end

- The VHDL Back-end is invoked on system definitions using `writeVHDL` functions

```haskell
writeVHDL :: SysDef a -> IO ()
writeVHDLOps :: VHDLOps -> SysDef a -> IO ()
```

- Generates
  - Main VHDL entity and architecture
  - VHDL files for components used
  - Translation of polymorphic types/functions
- A global `forsyde_lib.vhd` file is installed together with the ForSyDe package
Generating Hardware for \texttt{addFourSysDef}

\texttt{writeVHDL addFourSysDef}

The GraphML Back-end

- This back-end generates XML files for
  - Intermediate representation of systems
  - Generating graphs
- The GraphML back-end is invoked on system definitions using \texttt{writeGraphML} functions

\texttt{writeGraphML :: SysDef a \rightarrow IO ()}
\texttt{writeGraphML\_Ops :: GraphML\_Ops \rightarrow SysDef a \rightarrow IO ()}

- The generated file can be visualized using the yEd program from yWorks
Deep-Embedded Limitations

- Supported Data-Types
  - For the VHDL Back-end
    - Data.Int\{8, 16, 32\}, Bool, ForSyDe.Bit
    - Tuples and Data.Param.FSVec
    - Enumerated types
  - In Functions:
    - Points-free notation is not allowed
    - Only a single clause is permitted

Enumerated Types

- Algebraic Data Types with zero arity
  - E.g. \texttt{data} Direction = \texttt{Left | Down | Right | Up}
- Should be made instances of the following Typeclasses:
  - ProcType (done automatically by Data & Lift)
  - Data (requires instantiation of Typeable)
  - Typeable (is done by DeriveDataTypeable \texttt{ext.})
  - Lift (deriveLift1 from Language.Haskell.TH.Lift)
Keypad encoder I

{-# LANGUAGE TemplateHaskell, DeriveDataTypeable #-}
module Encoder where

import ForSyDe
import Language.Haskell.TH.Lift (deriveLift1)

import Prelude hiding (Left, Right)
import Data.Generics (Data, Typeable)
import Data.Param.FSVec
import Data.TypeLevel.Num hiding ((==))

data Direction = Left | Down | Right | Up
   deriving (Typeable, Data, Show)

$(deriveLift1 ''Direction)
Keypad encoder I

{-# LANGUAGE TemplateHaskell, DeriveDataTypeable #-}
module Encoder where

import ForSyDe
import Language.Haskell.TH.Lift (deriveLift1)

import Prelude hiding (Left, Right)
import Data.Generics (Data, Typeable)
import Data.Param.FSVec
import Data.TypeLevel.Num hiding (==)

data Direction = Left | Down | Right | Up
  deriving (Typeable, Data, Show)

$(deriveLift1 ''Direction)
Keypad encoder I

{-# LANGUAGE TemplateHaskell, DeriveDataTypeable #-}

module Encoder where

import ForSyDe
import Language.Haskell.TH.Lift (deriveLift1)

import Prelude hiding (Left, Right)
import Data.Generics (Data, Typeable)
import Data.Param.FSVec
import Data.TypeLevel.Num hiding ((==))

data Direction = Left | Down | Right | Up

deriving (Typeable, Data, Show)

$(deriveLift1 ''Direction)

Keypad encoder II

define

encoderFun :: ProcFun (FSVec D4 Bit -> AbstExt Direction)
encoderFun = $(newProcFun
  [d] encode :: FSVec D4 Bit -> AbstExt Direction
    encode v = if v ! d0 == H then Prst Left else
                    if v ! d1 == H then Prst Down else
                    if v ! d2 == H then Prst Right else
                    if v ! d3 == H then Prst Up else Abst [ ])

encoderProc :: Signal (FSVec D4 Bit) -> Signal (AbstExt Direction)
encoderProc = mapSY "encoder" encoderFun

encoderSysDef :: SysDef (Signal (FSVec D4 Bit) -> Signal (AbstExt Direction))
encoderSysDef = newSysDef encoderProc "KeypadEncoder"
  ["arrowBits"] ["direction"]
Keypad encoder II

encoderFun :: ProcFun (FSVec D4 Bit -> AbstExt Direction)
encoderFun = $(newProcFun
[d] encode :: FSVec D4 Bit -> AbstExt Direction
   encode v = if v ! d0 == H then Prst Left else
               if v ! d1 == H then Prst Down else
               if v ! d2 == H then Prst Right else
               if v ! d3 == H then Prst Up else Abst [])

encoderProc :: Signal (FSVec D4 Bit) -> Signal (AbstExt Direction)
encoderProc = mapSY "encoder" encoderFun

coderSysDef :: SysDef (Signal (FSVec D4 Bit) -> Signal (AbstExt Direction))
encoderSysDef = newSysDef encoderProc "KeypadEncoder"
   ["arrowBits"] ["direction"]

Demo!

Summary – Modeling Workflow

Create process functions
(newProcFun)

Create system function
(possibly using process constructors i.e.
mapSY etc.)

Create System Definition
(newSysDef)

Create an instance
(instanceX)

Simulate etc.
Related Literature I


Related Literature II