A Programming Model for Reconfigurable Computing Based in Functional Concurrency

Bill Harrison, Ian Graves, Adam Procter, Michela Becchi, & Gerard Allwein
ReCoSoC 2016
Mission/Safety-critical, *Reconfigurable* Systems

- Highly (Re)configurable Architectures/FPGAs
  - Many Specially Tailored, “One Off” Components
  - Reuse of Off-the-shelf components
  - “Mix and Match” comes to Hardware

- **Challenge:** High Assurance in this environment
  - Want the flexibility and speed of development
  - ...but also need formal guarantees of security & safety for critical systems
Mission/Safety-critical, *Reconfigurable* Systems

- Highly (Re)configurable Architectures/FPGAs
  - Many Specially Tailored, “One Off” Components
  - Reuse of Off-the-shelf components
  - “Mix and Match” comes to Hardware

- **Challenge:** High Assurance in this environment
  - Want the flexibility and speed of development
  - ...but also **need** formal guarantees of security & safety for critical systems

- **Unpleasant Reality:** Traditional HW Verification cannot cope with “Mix & Match”
  - Too slow & expensive for “one off” components
  - Why? Time spent “formalizing” hardware design
Language-based Approach to High Assurance Hardware

▶ “The Three P’s”
  ▶ DSLs & Language Virtualization
  ▶ Delite [Olukoton, Ienne]

▶ ReWire
  ▶ Fourth P: Provability
  ▶ Rigorous Semantics supports High Assurance
    ▶ Security & Safety Properties
    ▶ Formal Methods Productivity
Focus on Productivity
A Programming Model for Reconfigurable Computing Based in Functional Concurrency

- Recent Work:
  - Provability [FPT15]
  - Performance [ARC15]
  - Portability [LCTES15]

- Software Engineering “Virtues”
  - Abstraction, Modularity, Program Comprehension, etc.
  - ReWire
    - Functional Language supporting Concurrency
    - ...thereby common concurrency templates
ReWire Functional Hardware Description Language

- Inherits Haskell’s good qualities
  - Pure functions & types, monads, equational reasoning, etc.
  - Formal denotational semantics [HarrisonKieburtz05,Harrison05]
- Language design identifies HW representable programs
  - Mainly restrictions on recursion in functions and data
  - Built-in abstractions for clocked/parallel computations
  - “Connect Logic”: Types & operators for HW abstractions.
Reasoning about ReWire Programs

Ordinary Equational Reasoning on Functional Programs:

\[ e_1 = e_2 = \ldots = e_n \]

replaces “equals for equals”, uses induction/coinduction, etc.
Reasoning about ReWire Programs

Ordinary Equational Reasoning on Functional Programs:

\[ e_1 = e_2 = \ldots = e_n \]

replaces “equals for equals”, uses induction/coinduction, etc.

Ex: Hardware Verification from [FPT15]

Theorem (Correctness of Iterative Salsa20)

For all nonces \( n, n_0, \ldots, n_9 \) :: W128 and input streams is of the form \([(\text{High}, n), (\text{Low}, n_0), \ldots, (\text{Low}, n_9), \ldots]\), then:

\[ \text{salsa20} n = \text{nth 10 (feed is s1s20dev)} \]
Abstract Types for Devices

- **Built-in Type** `Dev i o`
  - Parameterized by input and output types, `i` and `o`
- Construct devices by building `Dev i o` values with **constructors**
- ReWire compiler translates `Dev i o` into synthesizable VHDL
- `Dev i o` is a “reactive resumption monad”
  - Algebraic structure for clocked, synchronous parallelism
  - Useful for specifying secure systems
    - [LCTES15,JCS09]
Iteration Constructor

\[ d = \text{iter } f \circ \]

\[
\text{iter} :: (i \to o) \to o \to \text{Dev } i \circ
\]

\[
i_t \to f(i_t) \to i_{t+1}
\]
Parallelism Constructor

\[
<\&> :: \text{Dev } i_1 \ o_1 \rightarrow \\
\text{Dev } i_2 \ o_2 \rightarrow \\
\text{Dev } (i_1,i_2) \ (o_1,o_2)
\]
Feedback Constructor

refold :: (o₁ → o₂) → (o₁ → i₂ → i₁) → Dev i₁ o₁ → Dev i₂ o₂

refold out conn d

\[ o' = \text{out } o \]
Representing **Dev i o** as a circuit

![Diagram showing a circuit](image)
Mealy Machines
Ex: Mealy Machine for Mutex

```plaintext
Unlocked
  /\  (ReqLock, _)/ (LockGrant, NullRsp)
  |   (Release, _)/ (Ack, NullRsp)
  \  (Release)/ (NullRsp, Ack)
  ^ (_ ,_ ) / (NullRsp, NullRsp)
  | (_ ,_ ) / (NullRsp, NullRsp)
  | (NullRsp, LockGrant)
  | (NullRsp, LockGrant)
  | (_ ,_ ) / (LockGrant, NullRsp)
Left Locked
  Left
  Locked

Bill Harrison
ReCoSoC16
```
Implementing Mealy Machines in Connect Logic

### States

- Unlocked
- LeftLocked
- RightLocked

### Data types

- Req: ReqLock | Release | NullReq
- Rsp: LockGrant | Ack | NullRsp

### Transition Function

\[
\delta :: \text{State} \to (\text{Req}, \text{Req}) \to (\text{State}, (\text{Rsp}, \text{Rsp}))
\]

- \(\delta \text{ Unlocked} (\text{ReqLock}, _) = (\text{LeftLocked}, (\text{LockGrant}, \text{NullRsp}))\)
- \(\delta \text{ Unlocked} (_, \text{ReqLock}) = (\text{RightLocked}, (\text{NullRsp}, \text{LockGrant}))\)
- \(\delta \text{ Unlocked} (_ _, _) = (\text{Unlocked}, (\text{NullRsp}, \text{NullRsp}))\)

- \(\delta \text{ LeftLocked} (\text{Release}, _) = (\text{Unlocked}, (\text{Ack}, \text{NullRsp}))\)
- \(\delta \text{ LeftLocked} (_ _, _) = (\text{LeftLocked}, (\text{LockGrant}, \text{NullRsp}))\)

- \(\delta \text{ RightLocked} (_, \text{Release}) = (\text{Unlocked}, (\text{NullRsp}, \text{Ack}))\)
- \(\delta \text{ RightLocked} (_ _, _) = (\text{RightLocked}, (\text{NullRsp}, \text{LockGrant}))\)

### ReWire Device

\[
\text{mutex} :: \text{Dev} (\text{Req}, \text{Req}) (\text{Rsp}, \text{Rsp})
\]

\[
\text{mutex} = \text{iterS} \delta (\text{Unlocked}, (\text{NullRsp}, \text{NullRsp}))
\]
Implementing Mealy Machines in Connect Logic

**States**

data State = Unlocked | LeftLocked | RightLocked

data Req = ReqLock | Release | NullReq

data Rsp = LockGrant | Ack | NullRsp
Implementing Mealy Machines in Connect Logic

States

\[
data\ State = \text{Unlocked} \mid \text{LeftLocked} \mid \text{RightLocked}
\]

\[
data\ Req = \text{ReqLock} \mid \text{Release} \mid \text{NullReq}
\]

\[
data\ Rsp = \text{LockGrant} \mid \text{Ack} \mid \text{NullRsp}
\]

Transition Function

\[
delta::\ State \rightarrow (\text{Req,Req}) \rightarrow (\text{State},(\text{Rsp,Rsp}))
\]

\[
delta\ \text{Unlocked}\ (\text{ReqLock},_)
\]

\[
= (\text{LeftLocked}, (\text{LockGrant},\text{NullRsp}))
\]

\[
delta\ \text{Unlocked}\ (_,\text{ReqLock})
\]

\[
= (\text{RightLocked}, (\text{NullRsp},\text{LockGrant}))
\]

\[
delta\ \text{Unlocked}\ (_,_)
\]

\[
= (\text{Unlocked}, (\text{NullRsp},\text{NullRsp}))
\]

\[
delta\ \text{LeftLocked}\ (\text{Release},_)
\]

\[
= (\text{Unlocked}, (\text{Ack},\text{NullRsp}))
\]

\[
delta\ \text{LeftLocked}\ (_,_)
\]

\[
= (\text{LeftLocked}, (\text{LockGrant},\text{NullRsp}))
\]

\[
delta\ \text{RightLocked}\ (_,\text{Release})
\]

\[
= (\text{Unlocked}, (\text{NullRsp},\text{Ack}))
\]

\[
delta\ \text{RightLocked}\ (_,_)
\]

\[
= (\text{RightLocked}, (\text{NullRsp},\text{LockGrant}))
\]
Implementing Mealy Machines in Connect Logic

**States**

data State = Unlocked | LeftLocked | RightLocked
data Req = ReqLock | Release | NullReq
data Rsp = LockGrant | Ack | NullRsp

**Transition Function**

delta :: State -> (Req,Req) -> (State,(Rsp,Rsp))
delta Unlocked (ReqLock,_)
  = (LeftLocked, (LockGrant,NullRsp))
delta Unlocked (_,ReqLock)
  = (RightLocked, (NullRsp,LockGrant))
delta Unlocked (_, _)
  = (Unlocked, (NullRsp,NullRsp))
delta LeftLocked (Release, _)
  = (Unlocked, (Ack,NullRsp))
delta LeftLocked (_, _)  
  = (LeftLocked, (LockGrant,NullRsp))
delta RightLocked (_, Release)
  = (Unlocked, (NullRsp,Ack))
delta RightLocked (_, _)  
  = (RightLocked, (NullRsp,LockGrant))

**ReWire Device**

mutex :: Dev (Req, Req) (Rsp, Rsp)
mutex = iterS delta (Unlocked, (NullRsp,NullRsp))
Simple Triple Modular Redundancy

The Rule of Three

```haskell
vote :: (a,a,a) -> a
vote (a1,a2,a3) | a1 == a2 = a1
| a1 == a3 = a1
| a2 == a3 = a2
| otherwise = a1

fan :: a -> i -> (i,i,i)
fan _ i = (i,i,i)

tmr :: Dev i o -> Dev i o
tmr dev = refold vote fan (dev<&>dev<&>dev)
```
Simple Triple Modular Redundancy

The Rule of Three

\[
\text{vote} :: (a,a,a) \rightarrow a
\]

\[
\text{vote} (a_1, a_2, a_3) \mid a_1 == a_2 = a_1
\]
\[
| a_1 == a_3 = a_1
\]
\[
| a_2 == a_3 = a_2
\]
\[
| \text{otherwise} = a_1
\]

\[
\text{fan} :: a \rightarrow i \rightarrow (i,i,i)
\]

\[
\text{fan} \_ i = (i,i,i)
\]

\[
\text{tmr} :: \text{Dev} i o \rightarrow \text{Dev} i o
\]

\[
\text{tmr} \ \text{dev} = \text{refold} \ \text{vote} \ \text{fan}
\]

\[
\quad (\text{dev} \&\& \ \text{dev} \&\& \ \text{dev})
\]
Programming Synchronization

Barriers

\[
\begin{align*}
\text{d1} & \quad \text{Busy} \\
\text{d2} & \quad \text{Complete} \\
\text{Barrier} \\
\text{Continue}
\end{align*}
\]
Programming Synchronization

Barriers

```haskell
data Status a = Busy | Complete a

barrier :: Dev i1 (Status o1) ->
    Dev i2 (Status o2) ->
    Dev (i1,i2) (Status (o1,o2))

barrier d1 d2 =
    refold out inp
        (makeStall d1 <&> makeStall d2)

where
    inp (Busy,Busy) (i1,i2) = (Continue i1,Continue i2)
    inp (Complete l,Busy) (i1,i2) = (Stall, Continue i2)
    inp (Busy,Complete r) (i1,i2) = (Continue i1,Stall)
    inp (Complete l,Complete r) (i1,i2) = (Continue i1,Continue i2)

    out (Busy,_) = Busy
    out (_,Busy) = Busy
    out (Complete a,Complete b) = Complete (a,b)
```
A Dual Core System realized in ReWire

```
dlx_\ell \quad :: \quad \text{Dev} \ (\text{Instr}_\ell, \text{Rsp}_\ell) \ (\text{Next}_\ell, \text{Req}_\ell)
memCtrl \quad :: \quad \text{Dev} \ (\text{Data}, \text{Req}_H, \text{Req}_L) \ (\text{Req}, \text{Rsp}_H, \text{Rsp}_L)
memory \quad :: \quad \text{Dev} \ \text{Req} \ \text{Data}

\text{system} \quad :: \quad \text{Dev} \ (\text{Instr}_H, \text{Instr}_L) \ (\text{Next}_H, \text{Next}_L)

\text{system} = \text{refold}

\quad \text{systemOut}
\quad \text{systemIn}
\quad \text{(dlx}_H <\&> \text{dlx}_L <\&> \text{memCtrl} <\&> \text{memory})
```
The Memory Controller Pattern

```
reqMaster = reqMaster_policyH policyL
reqMaster_ :: Policy -> Policy -> Dev (Req,Req) (Req,(Mask,Mask))
```

Memory Controller Device

```
memCtrl :: Dev (Data,(Req,Req)) (Req,(Rsp,Rsp))
memCtrl = refold outputSelect inputSelect (reqMaster <&> rspMaster)
```
The Memory Controller Pattern

Access Policies as Functions

\[
\text{reqMaster} = \text{reqMaster}_- \text{ policy}_H \text{ policy}_L
\]

\[
\text{reqMaster}_- ::
\begin{align*}
\text{Policy} & \to \\
\text{Policy} & \to \\
\text{Dev} \ (\text{Req}, \text{Req}) & \ (\text{Req}, \ (\text{Mask}, \text{Mask}))
\end{align*}
\]
The Memory Controller Pattern

Access Policies as Functions

```
reqMaster = reqMaster_ policyH policyL
reqMaster_ ::
  Policy ->
  Policy ->
  Dev (Req,Req) (Req,(Mask,Mask))
```

Memory Controller Device

```
memCtrl :: Dev (Data,(Req,Req))
        (Req,(Rsp,Rsp))
memCtrl = refold
  outputSelect
  inputSelect
  (reqMaster <&> rspMaster)
```
Related Work

- **HW Synthesis from DSLs**
  - Delite [Olukotun, Ienne, et al.]
  - DSLs and Language Virtualization
  - The “Three P’s” + Provability

- **Functional HDLs**
  - Chisel, Bluespec, Lava
  - ReWire project motivated by formal methods & security
  - ReWire: functional concurrent language

  - [Procter et al., 2015;2016] produce a verified secure dual-core processor in ReWire

- **Cryptol**
Summary, Conclusions & Future Work

- FPGA Programmability: [Andrews15] argues SE virtues precondition for wider adoption of Reconfigurable Tech
  - to enable productivity, reuse, scalability
- Encapsulated a wide variety of concurrency templates
  - Synchronization, Memory Protection, Voting
  - Each of which displays Abstraction, Modularity and Comprehensibility
    - Enabled by functional HDL ReWire
- Approach relies on semantically-faithful compiler
  - Mechanization in Coq; Compiler Verification
- Rewire is open source:
  https://github.com/mu-chaco/ReWire
THANKS!

* This research supported by the US National Science Foundation CAREER Award #0746509 and the US Naval Research Laboratory.