Scalaness/nesT: Type Specialized Staged Programming for Sensor Networks

Peter Chapin*, Christian Skalka*, Scott Smith†, Michael Watson*

* Department of Computer Science, University of Vermont
† Department of Computer Science, The Johns Hopkins University

GPCE’13, Indianapolis, IN
October 28, 2013
The Problem Setting: Embedded Sensor Networks

- Distributed data-gathering systems for earth and agricultural sciences.
- At UVM, focus on alpine snow hydrology.
  - Deployments in California, New Hampshire, Arctic Norway.
Challenges of Programming Sensor Networks

- Heavily resource constrained—RAM, ROM, clock cycles, power.
- e.g., Crossbow TelosB: 4 MHz, 10 KiB RAM, 48 KiB ROM
- ...yet complex, distributed algorithms used.

State of the art:

- nesC and TinyOS: Optimized for efficiency, widely used.
nesC Modules

```c
#include "Message.h"

module SendC {
    uses error_t radio_x(Msg*);
}

implementation {
    ...
}
```

```c
module RadioC {
    provides error_t radio_x(Msg*);
    uses error_t handle_radio_r(Msg*);
}

implementation {
    ...
}
```

- Modules consist of a specification and implementation.
- Specification lists used and provided commands.
- Implementation is a C-like translation unit.
nesC Configurations

configuration AppC { }
implementation {
components SendC, RadioC;
SendC.radio_x -> RadioC.radio_x;
}

- Application formed by *wiring* components together.
- Component wiring is entirely static.
- Example above incomplete: unresolved import.
Our Approach

- **Staging with two stages.** Scala at metalevel, nesC residuum. Modules are the smallest unit of code manipulation.

- Technical features: *Type specialization* with *dynamic type construction, process separation*.

- **Cross-stage type safety:** Type checking at Scala level ensures type safety of nesC residuum.

- **Well-founded language design.**
• **In the lab**: First stage program specializes and composes modules of second stage code.

• **In the field**: Generated second stage program accounts for field conditions. Deployed to nodes (over the air).
Example: Introducing Some Type Abbreviations

abbrvt mesgT(t) =
{ src : t; dest : t; data : uint8[] };

abbrvt radioT =
< at ≼ uint32 >
{ export error_t radio_x(mesgT(at)*);
  import error_t handle_radio_r(mesgT(at)*); };

• A record type parameterized by a type t.
• nesT modules parameterized by types and values.
Example: Introducing Some Type Abbreviations

abbrvt mesgT(t) =
{ src : t; dest : t; data : uint8[] };

abbrvt radioT =
< at ≼ uint32 >
{ export error_t radio_x(mesgT(at)*);
import error_t handle_radio_r(mesgT(at)*); };

• A record type parameterized by a type t.

• nesT modules parameterized by types and values.
Example: nesT Modules

```plaintext
val authSend =
  < at ≼ uint32; sendk : uint8[] >
  { import error_t radio_x(mesgT(at)*);
    export error_t send(m : mesgT(at)*)
      { radio_x(AES_sign(m, sendk)); } }
```

- First stage manipulates entire nesT modules.
Example: Scalaness Method

```python

    typedef adt ≼ uint32 =
        if (nmax <= 256) uint8 else uint16;

    val sendM = authSend⟨adt;keys(0)⟩;
    val recvM = authRecv⟨adt;keys(1)⟩;
    sendM ⋉ radioM⟨adt⟩ ⋉ recvM;
}
```

- Types constructed during first stage execution.
- Values lifted from one stage to the next only at module instantiation.
- Wiring operator composes fully instantiated modules.
Example: Scalaness Method

```scala
def authSpecialize
  (nmax : Int,
   radioM : radioT,
   keys : Array[Array[uint8]]) : commT {

  typedef adt ≼ uint32 =
    if (nmax <= 256) uint8 else uint16;

  val sendM = authSend⟨adt;keys(0)⟩;
  val recvM = authRecv⟨adt;keys(1)⟩;
  sendM ⊙ radioM⟨adt⟩ ⊙ recvM;
}
```

- Types constructed during first stage execution.
- Values lifted from one stage to the next only at module instantiation.
- Wiring operator composes fully instantiated modules.
Example: Scalaness Method

```scala
def authSpecialize
  (nmax : Int,
   radioM : radioT,
   keys : Array[Array[uint8]]) : commT {

  typedef adt ⇝ uint32 =
    if (nmax <= 256) uint8 else uint16;

  val sendM = authSend⟨adt;keys(0)⟩;
  val recvM = authRecv⟨adt;keys(1)⟩;
  sendM ⇉ radioM⟨adt⟩ ⇉ recvM;
}
```

- Types constructed during first stage execution.
- Values lifted from one stage to the next only at module instantiation.
- Wiring operator composes fully instantiated modules.
Example: Generating Residual Program

\[
\text{image(appM } \times \\
\text{ authSpecialize(nmax, radioM, keys) } \times \\
\text{ appMR);}\\n\]

- Type system ensures imaged module is “runnable.”
- \text{image} writes nesC residuum at run time.
- Values serialized across process spaces at first stage run time.
- Arbitrary nesC wrapped in special \textit{external modules}.
Implementation

Scalaness/nesT has been implemented.

- nesT defined as restricted subset of nesC, compiled as nesC with some rewriting (e.g. array bounds checks).
- Scalaness defined by extension to the Scala compiler.
- Type checking extends Scala type checker with module types, module operation typings, nesT type checking.

Web site with samples: http://tinyurl.com/a85z8cu
Application: WSN Session Key Negotiation

Currently studying authorization schemes for WSNs.

- WSN may comprise interacting security domains wishing to (partially) share resources.
- Symmetric keys provide efficient foundation for securing access.
- Public keys allow symmetric key negotiation in an “open world” model.

Public key signature verification expensive in WSNs; around 90 seconds on Crossbow TelosB.

Refactor authorization decision and session key negotiation into different stages.
Application: WSN Session Key Negotiation

Decreases WSN computational overhead, RAM and ROM consumption.
Results

<table>
<thead>
<tr>
<th></th>
<th>Unsecured</th>
<th>Unstaged*</th>
<th>Staged</th>
<th>Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensor ROM</td>
<td>36254</td>
<td>48616</td>
<td>36596</td>
<td>25%</td>
</tr>
<tr>
<td>Sensor RAM</td>
<td>2868</td>
<td>5417</td>
<td>3038</td>
<td>44%</td>
</tr>
<tr>
<td>Harvester ROM</td>
<td>24316</td>
<td>35834</td>
<td>24436</td>
<td>32%</td>
</tr>
<tr>
<td>Harvester RAM</td>
<td>2274</td>
<td>4771</td>
<td>2402</td>
<td>50%</td>
</tr>
</tbody>
</table>

- Security model: Two different Harvester “nodes”
  1. Data download only.
  2. Data download and control.

Future Work

- Clarifying “middle ground” between language borders.
- Syntactic transformations: Allowing more natural syntax in Scalaness programs.
- Incorporating network communication.
- Other applications: Backcasting and evolving control.
Questions?

Peter Chapin <pchapin@cs.uvm.edu>

http://tinyurl.com/a85z8cu
The \( \langle \text{ML} \rangle \) language* was developed to study these elements at a foundational level.

- MetaML-like syntax and semantics, but novel features to moderate interactions between separate process spaces.
- Comprises \( F_{\leq} \).
- Restricted form of type construction (not full \( \lambda_\omega \)).
- Formal metatheory includes cross-stage type safety—residue of partial evaluation of well-typed code is guaranteed to be well-typed.

Sample Scalaness Typing

\[ \Delta_1 \circ \langle \Delta_2, \Gamma \rangle \{ i; e \} \]

Module type form, where:

- \( \Delta_2, \Gamma \) type parameter bounds and term parameter types.
- \( i, e \) import and export type signatures.
- \( \Delta_1 \) bounds of types constructed externally to the module.
  - Early substitution of these types unsound due to possible contravariant use in \( i; e \).

\[ \text{MODINSTT} \]

\[
\begin{align*}
\Gamma &\vdash e : \emptyset \circ <\overline{T}_1 < \overline{\tau}_1; \bar{x} : \overline{\tau}_2 \{ i; e \} \\
\Gamma &\vdash \bar{s} : \text{MetaType} \langle \overline{T}_1 \rangle \\
\Gamma &\vdash \bar{e}_2 : \overline{T}_2 \\
&\vdash [\overline{T}_1] \leq \overline{\tau}_1 \\
&\vdash [\overline{T}_2] \leq \overline{\tau}_2
\end{align*}
\]

\[ \Gamma \vdash e \langle \bar{s}; \bar{e}_2 \rangle : \bar{s} \leq [\overline{T}_1] \circ <> \{ i[\bar{s}/\overline{T}_1]; e[\bar{s}/\overline{T}_1] \} \]