Scalaness/nesT: Type Specialized Staged Programming for Sensor Networks

Peter Chapin
University of Vermont
pchapin@cs.uvm.edu

Christian Skalka
University of Vermont
skalka@cs.uvm.edu

Scott Smith
The Johns Hopkins University
scott@cs.jhu.edu

Michael Watson
University of Vermont
mpwatson@cs.uvm.edu

Abstract

Programming wireless embedded networks is challenging due to severe limitations on processing speed, memory, and bandwidth. Staged programming can help bridge the gap between high level code refinement techniques and efficient device level programs by allowing a first stage program to specialize device level code. Here we introduce a two stage programming system for wireless sensor networks. The first stage program is written in our extended dialect of Scala, called Scalaness, where components written in our type safe dialect of nesC, called nesT, are composed and specialized. Scalaness programs can dynamically construct TinyOS-compliant nesT device images that can be deployed to motes. A key result, called cross-stage type safety, shows that successful static type checking of a Scalaness program means no type errors will arise either during programmatic composition and specialization of WSN code, or later on the WSN itself. Scalaness has been implemented through direct modification and plug-in extension of the Scala compiler. Implementation of a staged public-key cryptography calculation shows the sensor memory footprint can be significantly reduced by staging.

1. Introduction

Programming wireless embedded networks is challenging because their architectures are severely resource constrained in terms of memory and processor speed. This paper describes a programming language designed to support the automatic generation of more runtime-efficient code for wireless sensor network (WSN) devices. The language enables *dynamic specialization* of device code on a nearby hub or other more resource-rich device, allowing adaptation to properties of a device's deployment environment such as neighborhood characteristics, network interference factors, *etc*.

Our programming language system supports dynamic generation of TinyOS programs, a popular platform for WSNs. It features programming abstractions for specializing WSN code, allowing on-the-fly adaptation to current WSN deployment conditions. The system has been implemented as an extension to Scala [29], through modification of the Scala compiler. We use a restricted form of *staging* [8, 30, 31] to achieve well founded dynamic program generation. *First stage* code is written in an extended version of Scala, called Scalaness, that includes high level abstractions to ease program development. Scalaness program execution yields a residual *second stage* WSN node program written in nesT, a variant of the popular nesC WSN programming language [12] with a stronger type checking analysis. The second stage program is constructed from module components treated as first class values, which may be *type and value specialized* during the course of first stage computation to yield more compact and efficient code. A code rewriting strategy in the implementation transforms nesT code into nesC code, which can be compiled using standard TinyOS tools.

While staging is well-studied and has been explored in a WSN context [23], our work is novel in that we achieve stronger static safety guarantees than previous work. At the point of Scalaness program compilation, our compiler can statically verify that any nesT program produced by the Scalaness runtime will be statically type-safe when deployed and run on a network device, even if module parameters are specialized during the course of nesT module composition. We call this property *cross-stage type safety*, which has been previously studied in a foundational language context [19]. In this paper we apply these concepts to the more practical Scalaness/nesT language, and illustrate how they support the implementation and efficiency of real WSN applications.

1.1 Application Setting and Contributions

The diagram in Fig. 1 provides an overview of the Scalaness/nesT language architecture. Scalaness source code is compiled in a modified Scala compiler to Java bytecode, and run in a standard JVM. At runtime this Scalaness program may generate nesT code, which is subsequently rewritten to nesC and compiled using the standard TinyOS compiler (ncc). The resulting image can then be installed on nodes in a WSN. Observe that more than one code image can be generated during program execution, so code can be specialized for each node in a network, or a single Scalaness program can refine and redeploy network code, allowing programmatic specification of evolving network behavior.

Another interesting feature of our intended application setting, captured in Fig. 1, is the physical platform on which different elements of the Scalaness/nesT "workflow" may be executed. Scalaness source code will typically be compiled in the lab, prior to deployment. There are two distinct deployment scenarios where compiled bytecode execution, TinyOS image generation, and mote (re)programming (the rightmost two boxes in Fig. 1) can occur. Clearly these activities can take place in the lab, where WSN motes can be easily imaged over e.g. USB connections prior to deployment. But the more interesting scenario we aim to support is generation of TinyOS images on a "hub" device *in situ*, and then to automatically reprogram WSN nodes over the air (OTA) from the nearby hub.

In WSN applications such as our Snowcloud snow telemetry system [11, 25], sensor motes report data to higher powered hubs, pictured in Fig. 2. The hub device in the figure uses an ARM pro-

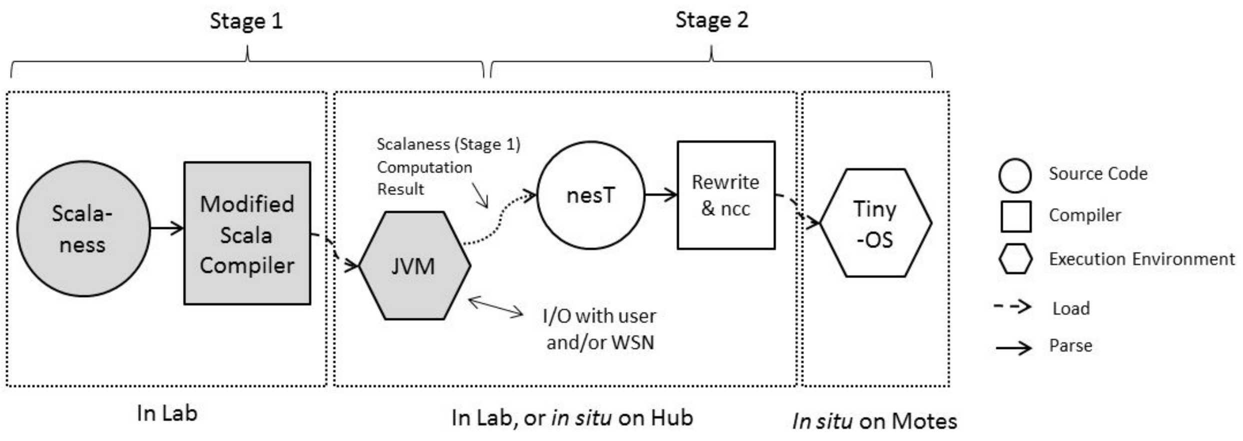

Figure 1. Scalaness/nesT Compilation and Execution Model

cessor, runs full-featured Debian Linux, and is in direct radio communication with the sensor network via a physically attached mote. Such a system can execute bytecode compiled from Scalaness source code, at run-time generate and compile TinyOS images, and deploy them to nearby motes. Since the hub is in communication with the network, Scalaness/nesT is uniquely positioned to evolve network behavior based on a global view of observed data, a technique called *backcasting* [32] when used specifically for network control. In this context the benefit of cross-stage type safety is clear: type-checked Scalaness compilation in the lab ensures type safety of bytecode execution on the hub, *and* type-safety of dynamically generated TinyOS image execution on the WSN. Manual correction of type errors in generated TinyOS images in this scenario is infeasible since these systems run automatically in remote settings.

Figure 2. A Sensor Node (L,C) and Hub Device (R).

Paper Outline. The main contributions of the work presented here are the specification, implementation, and prototype application of the Scalaness and nesT languages, including their syntax, semantics, and type analysis. In Sect. 2 we summarize Scalaness/nesT via discussion of an extended example. Formal specifications of the nesT and Scalaness languages are presented in Sect. 3 and Sect. 4, respectively. Their semantics and type theory possess several novel and challenging features, which we show in Sect. 5 are grounded in principles studied in a previous foundational setting [19]. We describe our implementation and present an extended example application of our system to resource access control in WSNs in Sect. 6 and Sect. 7, along with some empirical results demonstrating efficiency benefits of our approach. We conclude with remarks on related work in Sect. 8

2. An Example: Authenticated Messaging

In this section we provide a high-level overview of Scanaless/nesT usage and applications via an example. (This example is in fact written in DScalaness/DnesT, a simplified formalization of the implemented Scalaness/nesT, which is defined later in this paper.) The example illustrates both the type and value specializations that can occur in our system.

Program description. To illustrate type specialization, we refine address bitsize. It is well-known that minimizing address space size in WSN message packets can obtain significant energy savings by reducing message sizes, since each bit of transmission is known to consume energy similar to 800 instructions [21]. However, WSNs are "ad hoc" precisely in the sense that positions and densities of nodes in space are unpredictable, so "minimal" address space is an environmental property, where minimality may need to be determined *in situ*.

To illustrate value specialization, we define a DnesT code template that can be instantiated with specific session keys for secure communication in a WSN. We imagine that the template is instantiated on high powered hub or lab device, where session keys are generated. In previous work it has been shown how symmetric key signatures can be used to support language based resource authorization in WSNs [4, 5]. In particular, communication between security domains in a WSN is mediated by credentials implemented as keys, and nodes lying at domain frontiers can use different keys to send (to the other domain) and receive (to their own) over secured link layer channels. Since it is unpredictable where nodes will be physically distributed in space, appropriate keys for each node need to be established *in situ*. Defining node functionality using generic code that must be instantiated with specific keys allows adaptation to a deployment environment, and allows expensive computations for establishing session keys to be offloaded from the WSN to a higher powered device. Experience with an actual implementation of this application is discussed Sect. 7.

The Code. To distinguish Scalaness and nesT code in examples we will use a darker font for Scalaness code and a lighter font for nesT code, and line numbers for reference. We begin with the definition of a parameterized type `mesgT(t)`, where an instance `mesgT(`τ`)` is just an abbreviation for the specified record type with a type τ substituted for `t`.

```
1 typedef mesgT(t) = { src : t; dest : t; data : uint8[] };
```

Next, we define a type `radioT`, which is the type of nesT modules that provide an API to the radio.

```
2 typedef radioT  = < mt ≼ mesgT(uint) >
3                   { export radio_x(mt*);
4                     import handle_radio_r(mt*); };
```

The nesT module language is a simplified version of the nesC component language. In this example, any module of type `radioT` exports a `radio_x` function for sending messages, and imports a `handle_radio_r` function that allows received messages to be handled in a user-defined manner. Both take message references as arguments[1]. Furthermore, the module is parameterized by the type of messages `mt`, where the address type is upper-bounded by 32-bit unsigned integer. Thus, any module of type `radioT` can be dynamically specialized to a 32, 16, or 8 bit address space by type instantiation. Module type parameters are always defined with brackets $<$ ... $>$.

Now we define another type `commT` which is the type of modules providing a QOS layer over a specialized radio.

```
5 typedef commT  = (mt ≼ mesgT(uint)) ∘
6                   < >
7                   { export send(mt*);
8                     import handle_receive(mt*); };
```

Although this type is also parameterized by a bounded message type `mt`, as is `radioT`, the parameterization is subtly different syntactically and semantically, since `commT` expects a program context where the radio has been specialized. Thus, in `commT`, `mt` is understood as being "some" type with an upper bound of `mesgT(uint)` which occurs in the module signature, whereas the module itself has no parameters to be instantiated– note the empty instance parameter brackets $<>$ in the module type after the $\circ$ delimeter. This sort of type is needed in the presence of *dynamic type construction*, a useful Scalaness feature we exemplify below.

Next we define modules for sending and receiving messages that provide a layer of authentication security over the radio.

```
9  authSend   = < mt ≼ mesgT(uint); sendk : uint8[],>
10              { import radio_x(mt*),
11                export send(m : mt*)
12                       { radio_x(AES_sign(m, sendk)); }, };
13
14 authRecv   = < mt ≼ mesgT(uint); recvk : uint8[] >
15              { import handle_recv(mt*);
16                export handle_radio_r(m : mt*)
17                       { if AES_signed(m, recvk)
18                           handle_recv(m); } };
```

Observe that in the implementation of `send` in module `authSend`, messages are signed with a key `sendk`, whereas when messages are received they must be signed with a possibly different key `recvk` before being passed on to the user's receive handler, as specified in module `authRecv`. These modules are parameterized by a message type `mt`, and also the `sendk` and `recvk` key values.

To generalize a technique for composing these modules with a radio to yield a module of type `commT`, that is abstract wrt neighborhood sizes, radio implementations, and session key material, we define the Scalaness `authSpecialize` function as follows:

```
19 def authSpecialize
20   (nmax : uint16, radioM : radioT, keys : uint8[][]) : commT {
21     typedef adt ≼ uint =  if (nmax ≤ 256) uint8 else uint16;
22     val sendM = authSend⟨mesgT(adt); keys[0]⟩;
23     val recvM = authRecv⟨mesgT(adt); keys[1]⟩;
24     (sendM ⋉ radioM⟨mesgT(adt)⟩) ⋉ recvM;
25   }
```

The first-class status of nesT modules in Scalaness is apparent here. On line 20 the function is specified to take a module parameter `radioM` of type `radioT` among its arguments, and to return a module of type `commT` as a result. It also takes an array of keys as an argument, and on lines 22 and 23 it instantiates `sendMesg` and `recvMesg` with the keys in the array. It also uses the type `adt` in the instantiations, which we see in line 21 is dynamically constructed on the basis of the input variable `nmax` which defines the needed address space size. This illustrates a key novelty of our system, the ability to *dynamically* set a type to use on a mote based on a decision made in the Scalaness runtime. Since the value of `nmax` cannot be statically determined, the type analysis only knows that `adt` is some subtype of `uint`. Finally, on line 24 the instantiated radio module is composed with the instantiated send and receive modules via the Scalaness $\ltimes$ operator. The semantics of module composition here is standard [2]; in a composition aka wiring $\mu_1 \ltimes \mu_2$, the exports of μ_2 one are connected to imports of μ_1. The function result is a module of type `commT`.

To obtain a module defining a mote OS image in a program context where neighborhood size is known, a radio implementation has been provided, and session keys have been computed. We can then compose the results of an `authSpecialize` function with modules specifying top-level message send and receive behaviors, and a `main` application entry point as follows (here we assume it is known that address sizes can be limited to 8 bits, so `nmax` < 256). At line 30 a closed module is defined and a binary mote image can be produced by a call to `image`.

```
26 appMR  =
27   < > { export handle_recv(m : mesgT(uint8)*) {...} };
28 appM  =
29   < > { import send(mesgT(uint8)*); export main() {...} };
30 image(appM ⋉ (authSpecialize(nmax, radioM, keys) ⋉ appMR));
```

In DScalaness, `image` is an assertion that its argument is a *runnable* module, with no unresolved parameters or imports. In the Scalaness implementation, this is the point where nesT source code is actually generated. Successful Scalaness/nesT type checking (which occurs during stage 1 compilation as per Fig. 1) statically guarantees that specialized code generated at the point of `image` will run in a type-safe manner when it is eventually loaded and run on a mote.

3. The nesT Language Distilled

In this section we summarize a *D*istilled version of nesT, called DnesT, that isolates novel elements of nesT, specifically parametric types, subtyping, type safety, and modules. DnesT serves as a formal specification for the nesT implementation – given the novel type theory a specification is crucial as a guide for the implementation, and DnesT also serves as documentation for the language design. For lack of space in this article we summarize only the top-level structure of DnesT modules and our type checking algorithm, in order to focus more on the more central technical issues of module composition, instantiation, and typing at the Scalaness level. The DnesT language syntax is a reduced version of C which is largely standard.

The goal of nesT is to be a type-safe variant of nesC, and DnesT serves as the specification for how type safety is achieved. Our approach is another species of "safe C" language design projects such as [28]. In particular, in DnesT all array bound accesses are checked at run-time, and pointer arithmetic and casting are restricted to safe forms only. We have developed a new type checking algorithm that incorporates subtyping, which supports bounded type parameters in DnesT module definitions and a more accurate static analysis of Scalaness code in the presence of type construction and nesT module instantiation.

3.1 Syntax and Semantics of DnesT

Module definitions rely on a notion of lists aka sequences of syntactic entities, so we begin with a definition of relevant notation.

Notation and identifiers. *Sequences* are notated $x_1, \ldots, x_n$, and are abbreviated $\overline{x}$; $\overline{x}_{(i)}$ is the i-th element, $\emptyset$ denotes the empty

[1] For brevity the return type on all commands is omitted. In all cases it is the TinyOS error type `error_t`

$$\begin{array}{llll}
\varsigma, \tau & ::= & t \mid \top \mid \mathbf{uint8} \mid \mathbf{uint16} \mid \mathbf{uint} \mid & \textit{types} \\
 & & \mathbf{uninit} \mid \{\overline{\mathsf{l} : \tau}\} \mid \tau[\,] \mid \tau\star & \\
T & ::= & \overline{t \preccurlyeq \tau} & \textit{type parameters} \\
V & ::= & \overline{x : \tau} & \textit{value parameters} \\
c & ::= & \mathsf{f}(V) : \tau = \{e\} & \textit{command definition} \\
s & ::= & \mathsf{f}(V) : \tau & \textit{command signature} \\
\iota & ::= & \overline{s} & \textit{imports} \\
\xi & ::= & \overline{c} & \textit{exports} \\
\varepsilon & ::= & \overline{s} & \textit{export types} \\
d & ::= & \tau\, x \;=\; e \mid \tau\, x \;=\; [\![\overline{e}]\!] \mid & \textit{declarations} \\
 & & \tau\, x \;=\; \{\overline{\mathsf{l} = e}\} \mid c & \\
\mu & ::= & {<}T; V{>}\{\iota; \overline{d}; \xi\} & \textit{module definitions} \\
\mu\tau & ::= & {<}T; V{>}\{\iota; \varepsilon\} & \textit{module signatures}
\end{array}$$

Figure 3. Syntax of DnesT Types and Modules

sequence, and $|\overline{x}|$ is the size. We write $x \in \overline{x}$ to denote membership in sequences, and $x\overline{x}$ denotes a sequence with head x and tail $\overline{x}$. We denote append as $\overline{x}@\overline{y}$. For relational symbols $R \in \{\preccurlyeq, =, :\}$, we use the abbreviation: $\overline{x}\,R\,\overline{y} = x_1\,R\,y_1, \ldots, x_n\,R\,y_n$. So for example, $\overline{x} : \overline{\tau} = x_1 : \tau_1, \ldots, x_n : \tau_n$.

Syntactic sorts of identifiers are partitioned as follows. We use metavariable f (of set $\mathcal{F}$) for function names, l (of set $\mathcal{L}$) for field names, x (of set $\mathcal{V}$) for term variables, t (of set $\mathcal{T}$) for type variables.

Module syntax. The syntax of DnesT modules is defined in Fig. 3. Modules μ are written ${<}T; V{>}\{\iota; \overline{d}; \xi\}$ with T and V being generic type and term parameters, $\overline{d}$ being module scope identifier declarations, including function definitions, and ι and ξ being imports and exports. In Sect. 2 and elsewhere we use the keywords `import` and `export` in module and module type definitions that do not exist in the syntactic definition, but merely make explicit the categorization of module elements.

All type parameters are assigned an upper bound, and term parameters are explicitly typed. Imports and export types are sequences of imported and exported command type signatures. Exports are sequences of command definitions. Exports are defined in terms of expressions e, the syntax of which we omit here for brevity. Declarations $\overline{d}$; are a sequence of typed variable declarations. Base values, arrays (in brackets $[\![\mathsf{l} \cdot \mathsf{l}]\!]$), structs (in braces $\{\cdot\}$), and commands may all be declared, and the scope of declared variable names is restricted to the module. Declarations are important to include in DnesT, as they support serialization of value parameters during Scalaness instantiation as we describe in Sect. 4.3.

While we have elided the specifics of DnesT syntax from this shortened presentation, we now give a high-level summary of its largely standard features. Expressions include standard C-like conditional branching, looping, sequencing of expressions, function calls, arrays, structs, numeric base datatypes and basic arithmetic operations. As in nesC, no dynamic memory allocation is possible; all memory layout is established by static variable declarations. DnesT disallows pointer arithmetic, to support stronger type safety guarantees. Type casting and array access have run time checks imposed: types may never be cast to a pointer, and array accesses are always checked to be in bounds at runtime. As in nesC, DnesT includes a $\mathbf{post}$ operation for posting tasks, although we make no syntactic distinction between tasks and commands. The meaning of $\mathbf{post}$ corresponds to the "run-to-completion" model of TinyOS tasks. Interrupts are omitted from DnesT since they do not significantly affect the typing issues we are concerned with here.

Module semantics A "runnable" module – one without imports or generic parameters – is the DnesT model of a node OS image. The declarations in the module defines a *load sequence* establishing an initial machine configuration, and the application entry point is defined in a required command `main`.

DEFINITION 3.1. *A module of the form* ${<}\varnothing; \varnothing{>}\{; \overline{d}; \xi\}$*, where* $\mathtt{main}() : \mathbf{uninit} \in \xi$*, is* called runnable.

This model is consistent with nesC, where an application is defined as a top-level component that establishes an initial configuration through variable declarations, and requires user definition of an entry point (an event handler called `Booted`). Formally speaking, type safety in nesT is a dynamic property of runnable modules.

3.2 Type Checking and Subtyping

The type system for DnesT combines a standard procedural language typing approach with subtyping techniques adapted from previous foundational work [13, 19].

At the heart of our system is a decidable subtyping judgment $T \vdash \tau_1 \preccurlyeq \tau_2$, where T is a *coercion* and defines a system of upper bounds for type variables. This establishes a subtype ordering on base types, and also allows for width subtyping of records. The relation is defined in Fig. 4. Algorithms for deciding the relation and integrating it with dynamic type construction and other Scalaness (stage 1) type features was a central topic of [19].

$$\textsc{ReflS}\quad T \vdash \tau \preccurlyeq \tau \qquad\qquad \textsc{TopS}\quad T \vdash \tau \preccurlyeq \top \qquad\qquad \textsc{TransS}\quad \frac{T \vdash \tau_1 \preccurlyeq \tau_2 \qquad T \vdash \tau_2 \preccurlyeq \tau_3}{T \vdash \tau_1 \preccurlyeq \tau_3}$$

$$\textsc{UintS}\quad T \vdash \mathbf{uint8} \preccurlyeq \mathbf{uint16} \preccurlyeq \mathbf{uint} \qquad\qquad \textsc{StructS}\quad \frac{T \vdash \overline{\tau_1 \preccurlyeq \tau_3}}{T \vdash \{\overline{\mathsf{l}_1 : \tau_1} \uplus \overline{\mathsf{l}_2 : \tau_2}\} \preccurlyeq \{\overline{\mathsf{l}_1 : \tau_3}\}}$$

Figure 4. Subtyping Rules

The type checking algorithm for DnesT expressions is a combination of standard procedural type systems and standard subtyping systems. Module typing is obtained by type checking module exports, using a coercion obtained from the module type parameters and a typing environment obtained from a combination of module value parameters, imports, and variable type declarations. A valid module type checking judgement is written as:

$${<}T, V{>}\{\iota; \overline{d}; \xi\} : {<}T, V{>}\{\iota; \varepsilon\}$$

Where ε is just the type signatures of ξ, and each of the command bodies in ξ is proven to respect its type signature.

EXAMPLE 3.1. *The module* `authSend` *defined in Sect. 2 code line 9 can be assigned the following type in DnesT:*

```
< mt ≼ mesgT(uint); sendk : uint8[] >
  { import radio_x(mt*), export send(mt*) }
```

4. The Scalaness Language Distilled

Scalaness serves as the language for nesT module composition in the same manner as nesC configurations serve to compose nesC modules, but Scalaness is a much more powerful metalanguage since modules are treated as a new category of first class values in Scalaness. Instantiation, composition (aka wiring), and imaging of modules are defined as operations on module values. Because instantiation of modules with both types and values is allowed, values and types may migrate from the Scalaness level to the nesT level after programmatic refinement, realizing a disciplined form of code specialization.

Our goal in this Section is to describe the Scalaness syntax and semantics realized in our implementation. Since Scala is too

$$\begin{array}{lcll}
\mathtt{L} & ::= & \mathtt{class}\ \mathtt{C}\langle \bar{\mathtt{X}} <: \bar{\mathtt{N}}\rangle\ \mathtt{extends}\ \mathtt{N}\ \{\bar{\mathtt{T}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} & \textit{classes} \\
\mathtt{K} & ::= & \mathtt{C}(\bar{\mathtt{T}}\ \bar{\mathtt{f}})\{\mathtt{super}(\bar{\mathtt{f}});\ \mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}};\} & \textit{constructors} \\
\mathtt{M} & ::= & \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}\ \bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} & \textit{methods} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\bar{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{C}\langle\bar{\mathtt{T}}\rangle(\bar{\mathtt{e}}) \mid (\mathtt{N})\mathtt{e} \mid & \textit{expressions} \\
 & & \mathtt{e.f} = \mathtt{e} \mid \mathtt{l} \mid \mathtt{def}\ \mathtt{x} : \mathtt{T} = \mathtt{e}\ \mathtt{in}\ \mathtt{e} \mid & \\
 & & \mu \mid \mathtt{e} \ltimes \mathtt{e} \mid \mathtt{e}\langle\bar{\mathtt{e}};\bar{\mathtt{e}}\rangle \mid \mathtt{image}\ \mathtt{e} & \\
\mathtt{T} & ::= & \mathtt{X} \mid \mathtt{N} \mid T \circ \mu_T & \textit{types} \\
\mathtt{N} & ::= & \mathtt{C}\langle\bar{\mathtt{T}}\rangle & \textit{class types} \\
\mathtt{l} & ::= & (\mathtt{p}, \mathtt{N}) & \textit{references}
\end{array}$$

Figure 5. The Syntax of DScalaness

large to easily formalize, we define here a *D*istilled Scalaness, DScalaness, that extends a core typed object-oriented language to include syntax and semantics for defining and composing DnesT modules. The particular object-oriented core calculus we use is a combination of two Featherweight Java variants: Featherweight Generic Java (FGJ) [16] and Assignment Featherweight Java (AFJ) [26].

4.1 Syntax of DScalaness

The DScalaness language syntax is presented in Fig. 5. We refer the reader to [16, 26] for details on the FGJ and AFJ object oriented calculi, which are represented in the languages of class definitions, constructors, methods, and the first line of expression forms defined in Fig. 5. DScalaness extends these features with a typed variable declaration form $\mathtt{def}\ \mathtt{x} : \mathtt{T} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2$ where the scope of $\mathtt{x}$ is $\mathtt{e}_2$, a dynamic type construction form $\mathtt{typedef}\ \mathtt{x} <: \mathtt{T} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2$ with similar scoping rules (although this is defined as syntactic sugar in Definition 4.1), and several features for module definition and manipulation. First, we include DnesT modules μ in the DScalaness expression and value spaces: instantiation is obtained via the form $\mathtt{e}_1\langle\bar{\mathtt{e}}_1;\bar{\mathtt{e}}_2\rangle$, where $\bar{\mathtt{e}}_1$ are type parameters and $\bar{\mathtt{e}}_2$ are value parameters. Wiring of modules is denoted $\mathtt{e}_1 \ltimes \mathtt{e}_2$. Imaging of modules, denoted $\mathtt{image}\ \mathtt{e}$, ensures that $\mathtt{e}$ computes to a runnable DnesT module.

Syntactic sugaring. The upper bound of $\mathtt{x}$ in any `typedef` of the form $\mathtt{typedef}\ \mathtt{x} = \tau$ is implicitly τ, see for example `radioT` in Sect. 2. A type of the form $\mathtt{x}(\tau)$ in the scope of parameterized type definition $\mathtt{typedef}\ \mathtt{x}(t) = \tau'$ is just an abbreviation for $\tau'[\tau/t]$, see in particular `mesgT` in Sect. 2.

4.2 Semantics of DScalaness

The semantics of DScalaness is an extension of the semantics of AFJ and FGJ to incorporate DnesT modules and operations. Computations assume a fixed class table CT allowing access to class definitions via class names, which always decorate an object's type. A *store* ST is a function from memory locations $\mathtt{p}$ to object representations. Objects are represented in memory by lists of object references $\bar{\mathtt{l}}$, which refer to the locations of the objects stored in mutable field values. A reference $\mathtt{l}$ is a pair $(\mathtt{p}, \mathtt{N})$ where $\mathtt{p}$ is the memory location of an object representation and $\mathtt{N}$ is the nominal type of the object, including its class name. Hence, given an object reference $(\mathtt{p}, \mathtt{C}\langle\bar{\mathtt{T}}\rangle)$, we can access and mutate its fields $\bar{\mathtt{l}} = ST(\mathtt{p})$, and access and use its methods via the definition $CT(\mathtt{C})$.

Following AFJ, the semantics of DScalaness is defined as a *labeled transition system*, where transitions are of the form $\mathtt{e}{-}\{s = ST, s' = ST'\} \rightarrow \mathtt{e}'$. Intuitively, this denotes that given an initial store ST and expression $\mathtt{e}$, one step of evaluation results in a modified store ST' and contractum $\mathtt{e}'$. We write $\mathtt{e} \rightarrow \mathtt{e}'$ as an abbreviation when the store is not altered.

The primary novelty of DScalaness is the formal semantics of type and module construction. We begin with type construction, which is provided to allow programmers to dynamically construct module type instances. The appropriate behavior is obtained by treating dynamically constructed types as extensions of a basic class of objects, and declarations of DnesT level types via a `typedef` construct as syntactic sugar for ordinary object construction. We define a `LiftableType` class as the supertype of all types of objects that can be used to instantiate a module, and dynamically constructed types are defined as instances of a generic `MetaType` class.

DEFINITION 4.1. *Any DScalaness class table CT comprises the following definitions:*

$CT(\mathtt{LiftableType}) =$
$\quad\mathtt{class}\ \mathtt{LiftableType}\langle\rangle\ \mathtt{extends}\ \mathtt{Object}\ \{\ldots\}$
$CT(\mathtt{MetaType}) =$
$\quad\mathtt{class}\ \mathtt{MetaType}\langle\mathtt{X} <: \mathtt{LiftableType}\rangle\ \mathtt{extends}\ \mathtt{Object}\ \{\ldots\}$

And we take as given the following syntactic sugar:

$$\mathtt{typedef}\ \mathtt{x} <: \mathtt{T} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 \triangleq \mathtt{def}\ \mathtt{x} : \mathtt{MetaType}\langle\mathtt{T}\rangle = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2$$

Class type `MetaType` *is generalized on a single type variable. For brevity of notation, we define:*

$$\mathtt{MetaType}\langle\bar{\mathtt{T}}\rangle \quad \triangleq \quad \overline{\mathtt{MetaType}\langle\mathtt{T}\rangle}$$

A crucial fact of DScalaness type construction is that any dynamically constructed type cannot be treated as a type at the DScalaness level. This is a more restrictive mechanism than envisioned in our foundational model [19], however it allows us to define DScalaness as a straightforward extension to Scala, especially in terms of type checking.

Module instantiation is the only point where specialization of DnesT modules is allowed. Since DScalaness and DnesT are two different language spaces, some sort of transformation must occur when values migrate from DScalaness to DnesT via module instantiation. This *lifting* transformation involves both data mapping and serialization since the process spaces also differ. We aim to be flexible and allow the user to specify how values are lifted and how types are transformed. We only require that lifting and type transformation are coherent, in the sense that the lifting of an object should be typeable at the object's type transformation. We formalize this in the following definition.

DEFINITION 4.2. *We assume given a relation $\overset{lift}{\hookrightarrow}$ which transforms a DScalaness reference* $\mathtt{l}$ *into DnesT declarations $\overline{d}$ and expression e. We also assume given a DScalaness-to-DnesT transformation of types $[\![\cdot]\!]$. To preserve type safety, we require in all cases that $(p, \mathtt{N}) \overset{lift}{\hookrightarrow} \overline{d}, e$ implies both of the following for some type environment G:*

$$\varnothing, \varnothing \vdash \overline{d} : G \qquad \textit{and} \qquad G, \varnothing \vdash e : [\![\mathtt{N}]\!]$$

The full definition of serialization and an example are given and discussed below in Sect. 4.3.

Module wiring is given a standard component composition semantics. We only allow wiring of instantiated modules, which is consistent with nesC and simpler to implement. In a wiring $\mathtt{e}_1 \ltimes \mathtt{e}_2$, the exports of $\mathtt{e}_1$ are wired to the imports of $\mathtt{e}_2$. This is specified in the MODWIRE rule in Fig. 6, which relies on the following auxiliary definition of operations for combining mappings.

DEFINITION 4.3 (Special Mapping Operations). *Let m range over vectors with mapping interpretations, in particular T, V, ι, and ξ. Binary operator $m_1 \curlyvee m_2$ represents (non-exclusive) map merge, i.e. $m_1 \curlyvee m_2 = m_1 @ m_2$ with the requirement that $id \in \mathrm{Dom}(m_1) \cap \mathrm{Dom}(m_2)$ implies $m_1(id) = m_2(id)$. The*

MODINST

$$\frac{\mu = {<}\bar{t} \preccurlyeq \bar{\tau}; \bar{x} : \bar{\varsigma}{>}\{\iota; \bar{d}; \xi\} \qquad \text{serialize}(\bar{x}, \bar{\varsigma}, \bar{\mathtt{l}}) = \overline{d'}}{\mu\langle(\bar{\mathtt{p}}, \mathtt{MetaType}\langle\bar{\mathtt{T}}\rangle);\ \bar{\mathtt{l}}\rangle \to {<>}\{\iota; \overline{d'}@\bar{d}; \xi\}[[\![\bar{\mathtt{T}}]\!]/\bar{t}]}$$

MODWIRE

$$\frac{\iota = (\iota_1/\mathrm{Dom}(\xi_2))@\iota_2 \qquad \bar{d} = \bar{d}_2@\xi_2|_{\mathrm{Dom}(\iota_1)}}{{<>}\{\iota_1; \bar{d}_1; \xi_1\} \ltimes {<>}\{\iota_2; \bar{d}_2; \xi_2\} \to {<>}\{\iota; \bar{d} \curlyvee \bar{d}_1; \xi_1\}}$$

MODIMAGE

$$\frac{\texttt{main} \text{ defined in } \xi}{\texttt{image}\ ({<>}\{; \bar{d}; \xi\}) \to {<>}\{; \bar{d}; \xi\}}$$

Figure 6. DScalaness Module Semantics

mapping m/S is the same as m except undefined on domain elements in set S, and the mapping $m \mid_S$ is the same as m except undefined on elements not in S.

Finally, the MODIMAGE rule in Fig. 6 shows that imaging it is an assertion requiring its arguments to be a runnable module.

EXAMPLE 4.1. *Given code definitions in Sect. 2 and an invocation:*

$$\texttt{authSpecialize}(50,\ \texttt{radioM},\ [|\,\mathtt{k}_1, \mathtt{k}_2\,|])$$

where $\texttt{radioM} : \texttt{radioT}$*, and* $\mathtt{k}_1, \mathtt{k}_2$ *are keys, the evaluation of the expression* $\texttt{sendM} \ltimes \texttt{radioM}\langle\texttt{mesgT}(\texttt{adt})\rangle$ *on line 24 will evaluate to the following module:*

```
< > (
{ import handle_radio_r(mesgT(uint8)*);
  ...;
  export send(m : mesgT(uint8)*)
        { radio_x(AES_sign(m, k1)); } }
```

where the elided declarations include a definition of a command `radio_x` *imported from* `radioM` *also with argument type* `mesgT(uint8)*`.

4.3 Serialization and Lifting

Serialization generates a flattened DnesT source code version of a DScalaness object in memory. At the top level, serialization binds the value parameters of a module to the results of flattening, aka lifting, via a sequence of declarations. Here is the precise definition.

DEFINITION 4.4 (Serialization). *Assume given a store ST which is implicit in the following definitions. We define serialization of DScalaness references as follows, along with an extension of the user defined lifting relation to sequences of references:*

$$\frac{\bar{\mathtt{l}} \overset{\textit{lift}}{\hookrightarrow} \bar{d}, \bar{e}}{\text{serialize}(\bar{x}, \bar{\tau}, \bar{\mathtt{l}}) = \bar{d}@\ \bar{\tau}\ \bar{x} = \bar{e}} \qquad \varnothing \overset{\textit{lift}}{\hookrightarrow} \varnothing, \varnothing$$

$$\frac{\mathtt{l} \overset{\textit{lift}}{\hookrightarrow} \bar{d}, e \qquad \bar{\mathtt{l}} \overset{\textit{lift}}{\hookrightarrow} \overline{d'}, \bar{e}}{\mathtt{l}\bar{\mathtt{l}} \overset{\textit{lift}}{\hookrightarrow} \bar{d}@\overline{d'}, e\bar{e}}$$

Although lifting is user defined, a standard strategy is to introduce a new declared variable for each memory reference in the lifted object, and bind the variable to the lifted referent. Hence, lifting will typically be defined recursively. In our implementation, we have adapted a "default" lifting which follows this strategy, and also transforms objects by just transforming the fields into a representative struct, and ignoring methods. We will illustrate this with an example in Sect. 6. We can formally capture the essence of this transformation with the following definitions. It is easy to see that these definitions will satisfy the requirements of Definition 4.2.

EXAMPLE 4.2. *In this example we allow lifting of any object references, and transform the object o into a structure containing the transformed fields of o. Methods are disregarded by the transformation. Here is the specification of the type transformation:*

$$\frac{CT(\mathtt{C}) = \texttt{class}\ \mathtt{C}\langle\bar{\mathtt{X}} <: \bar{\mathtt{S}}\rangle\ \texttt{extends}\ \mathtt{N}\ \{\bar{\mathtt{R}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\}}{[\![\mathtt{C}\langle\bar{\mathtt{T}}\rangle]\!] = \{\bar{\mathtt{f}} : [\![\bar{\mathtt{R}}[\bar{\mathtt{T}}/\bar{\mathtt{X}}]]\!]\}}$$

and here is the specification of lifting.

$$\frac{ST(\mathtt{p}) = \bar{\mathtt{l}} \qquad \textit{fields}(\mathtt{C}) = \bar{\mathtt{T}}\ \bar{\mathtt{f}} \qquad \bar{\mathtt{l}} \overset{\textit{lift}}{\hookrightarrow} \bar{d}, \bar{e} \qquad x\ \textit{fresh}}{(p, \mathtt{C}\langle\bar{\mathtt{R}}\rangle) \overset{\textit{lift}}{\hookrightarrow} \bar{d}@([\![\mathtt{C}\langle\bar{\mathtt{R}}\rangle]\!]\ x\ =\ \{\bar{\mathtt{f}} = \bar{e}\}), x}$$

4.4 DScalaness Type Checking

The primary novelty of DScalaness are the rules for DnesT module typing and composition, and that is the focus of this section. We adopt the typing rules of FGJ in their entirety, and refer the reader to [16] for relevant details.

DScalaness syntax for expressing DnesT module types is $T \circ \mu\tau$, where $\mu\tau$ is a DnesT module type. The T in this form represents the type bounds of dynamically constructed types that have been used to instantiate the module; we refer to this part of the type as the *instance coercion*. Because these types are dynamically constructed, their identity is not known statically, hence the need to treat them as upper-bounded type names in the static type analysis. This subtle technical point of our type system is discussed at more length in Sect. 5. It is important to note that the type names in T will be fully resolved at run time, so that any module generated by a DScalaness program execution will have a fully reified DnesT type.

This is reflected in the MODT rule in Fig. 7, which connects the DnesT typing system with the DScalaness type system. Since in this case we are typing an uninstantiated module definition its instance coercion is empty. An instance coercion in a module type is directly populated when a module is instantiated, as in the MODINSTT rule. Here, the type instances $\bar{\mathtt{e}}_1$ are all dynamically constructed, so they define the upper bounds of the instantiated module's instance coercion. We also expect all type and value parameters to respect the typing bounds specified in the module definition. The MODWIRET typing rule for module wiring is a straightforward reflection of the operational rule for module wiring, as is the MODIMAGET rule for module runnability imaging.

EXAMPLE 4.3. *Returning to the code and type examples in Sect. 2, we may assign the following typing:*

$$G \vdash \texttt{authSpecialize}(50,\ \texttt{radioM},\ [|\,\mathtt{k}_1, \mathtt{k}_2\,|])\ : \texttt{commT}$$

Given $\texttt{radioM} : \texttt{radioT}, \mathtt{k}_1 : \texttt{uint8}[], \mathtt{k}_2 : \texttt{uint8}[] \in G$.

5. Scalaness/nesT Foundations

The Scalaness/nesT type system design is based on principles studied in the foundational calculus $\langle\text{ML}\rangle$[2] [19]. $\langle\text{ML}\rangle$ comprises $F_{\leq}$, state, dynamic type construction, and staging features. In this section we describe how the design of modules and module operations in Scalaness can be modeled in $\langle\text{ML}\rangle$. Although the correspondance is informal, these models directed the design of Scalaness semantics and type checking, and provide confidence in its soundness. While our choice of modules as the basic unit of nesT code is based on obvious software engineering concerns and the need for a tight relation with nesC, Scalaness modules are well correlated with certain structures in $\langle\text{ML}\rangle$ and so are also technically appealing.

[2] Pronounced "framed ML".

$$\textsc{ModT}\quad \frac{\mu : \mu\tau \text{ in DnesT type checking}}{\Gamma \vdash \mu : \varnothing \circ \mu\tau}$$

$$\textsc{ModImageT}\quad \frac{\Gamma \vdash \texttt{e} : T \circ <\!>\{\iota; \varepsilon\} \qquad \text{main}() : \tau \in \varepsilon}{\Gamma \vdash \texttt{image e} : T \circ <\!>\{\iota; \varepsilon\}}$$

$$\textsc{ModInstT}\quad \frac{\begin{array}{c}\Gamma \vdash \texttt{e} : \varnothing \circ <\bar{t} \preccurlyeq \overline{\tau}_1; \overline{x} : \overline{\tau}_2>\{\iota; \varepsilon\} \qquad \Gamma \vdash \bar{\texttt{e}}_1 : \texttt{MetaType}\langle\bar{\texttt{T}}_1\rangle \\ \Gamma \vdash \bar{\texttt{e}}_2 : \bar{\texttt{T}}_2 \qquad \vdash [\![\bar{\texttt{T}}_1]\!] \preccurlyeq \overline{\tau}_1 \qquad \vdash [\![\bar{\texttt{T}}_2]\!] \preccurlyeq \overline{\tau}_2\end{array}}{\Gamma \vdash \texttt{e}\langle\bar{\texttt{e}}_1; \bar{\texttt{e}}_2\rangle : \bar{t} \preccurlyeq [\![\bar{\texttt{T}}_1]\!] \circ <\!>\{\iota; \varepsilon\}}$$

$$\textsc{ModWireT}\quad \frac{\begin{array}{c}\Gamma \vdash \texttt{e}_1 : T_1 \circ <\!>\{\iota_1; \varepsilon_1\} \\ \Gamma \vdash \texttt{e}_2 : T_2 \circ <\!>\{\iota_2; \varepsilon_2\} \qquad \iota = (\iota_1/\text{Dom}(\varepsilon_2))@\iota_2\end{array}}{\Gamma \vdash \texttt{e}_1 \ltimes \texttt{e}_2 : T_1 \curlyvee T_2 \circ <\!>\{\iota; \varepsilon_1\}}$$

Figure 7. DScalaness Module Typing Rules

The model of a module. Code as a datatype is available in $\langle\text{ML}\rangle$ as expressions of the form $\langle e\rangle$. While code as a datatype is a standard feature of staged/generative programming, $\langle\text{ML}\rangle$ has adapted staged programming to a setting where different code levels are intended for execution on different machines with distinct process spaces. In particular, values, including code values, must be closed. If a type or term variable occurs free in $\langle e\rangle$, it must be Λ or λ bound, respectively, for closure. Hence, if a type variable t is free in $\langle e\rangle$, then $\Lambda t \preccurlyeq \tau.\langle e\rangle$ binds it, and provides parametric subtyping polymorphism for $\langle\text{ML}\rangle$ terms.

If the term variable x is free in $\langle e\rangle$, then $\lambda x : \tau.\langle e\rangle$ binds it. Furthermore, the type τ in the term $\lambda x : \tau.\langle e\rangle$ *must* be of the form $\langle\varsigma\rangle$, because the type discipline requires that x is of code type, since it occurs within code. If the programmer wishes to pass a value residing at the current execution stage to such a function, it must be explicitly "lifted" in the now-standard style of [31]. However, in $\langle\text{ML}\rangle$, lifting a value entails serialization of it, which is non-trivial in case the value is stateful.

We use $\langle\text{ML}\rangle$ type and term bindings to model Scalaness type and term parameters. This is a standard strategy, in fact FGJ typing [16] is based on it as well. Hence the basic analog of a module is:

$$\Lambda t \preccurlyeq \tau.\lambda x : \langle\varsigma\rangle.\langle e\rangle$$

where t is a bounded type parameter and x is a value parameter.

The model of instantiation. Most of the interesting parts of Scalaness typing happen at instantiation. Given the above model of a module, the $\langle\text{ML}\rangle$ analog of instantiation is a term of the form:

$$(\Lambda t \preccurlyeq \tau.\lambda x : \langle\varsigma\rangle.\langle e\rangle)(\tau')(\text{lift } v)$$

where all parameters are instantiated. Note in particular that the value parameter v must be explicitly lifted, since the model must reflect that values passed in to modules are always constructed at the first stage in a Scalaness program. This means that v must be assumed to not be a code value, while the type annotation on x requires that it be lifted. There is no explicit lift operation in Scalaness, but the DSCalaness semantics (Fig. 6) specifies that serialization is always implicit at module instantiation. Scalaness typing of instantiation thus treats value instantion as λ application with implicit lifting of the argument, and type instantiation as Λ application, i.e. a form of bounded $\forall$-elimination.

Type construction and variable escape. A central technical novelty and core feature of DScalaness is dynamic type construction for module instantiation. As we discussed in Sect. 2, this feature is technically challenging since constructed types can escape their scope of declaration. Similarly, in $\langle\text{ML}\rangle$, types may be dynamically constructed that can escape their declaration scope, in particular if they are used as function type annotations. An $\exists$ type binder was introduced in $\langle\text{ML}\rangle$ for this purpose; intuitively a type of the form $\exists t \preccurlyeq \tau.\varsigma$ is a type containing a dynamically constructed type term t with upper bound τ. $\langle\text{ML}\rangle$ includes a "tlet" expression form for constructing types, so for example:

$$\text{tlet } t \preccurlyeq \textbf{uint16} = \text{if } e \text{ then } \textbf{uint8} \text{ else } \textbf{uint16} \text{ in } (\lambda x : t.x)$$
$$:$$
$$\exists t \preccurlyeq \textbf{uint16}.t \rightarrow t$$

Here a type t is dynamically constructed to be either **uint8** or **uint16**, and then used in the type annotation of a type-specialized identity function. Furthermore, t escapes its declaration scope since it annotates a function argument. Since e is some arbitrary computation, we cannot statically predict what t will be, other than "some type with upper bound **uint16**". Note also that since t can appear in contravariant positions, it is unsound to perform a covariant substitution of **uint16** for t, so the $\exists$ bound is needed. Although this usage of $\exists$ types is somewhat non-standard, an egenvariable interpretation of the bound type variable is sound and also consistent with standard existential type interpretations.

Inspired by these foundations, in DScalaness the type form:

$$T_1 \circ <T_2; V>\{\iota; \varepsilon\}$$

captures the same typing mechanisms, in particular the instance coercion T_1 is the analog of $\exists$ bound type variables, in contrast to the type parameters T_2 which are implictly $\forall$ bound, as discussed above. The static semantics of T_1 and T_2 are distinguished appropriately, especially in the treatment of the typing rules for module instantiation and module wiring in Fig. 7.

6. Implementation

Scalaness is implemented as a modification to the open source Scala compiler. Although the Scala compiler supports a plug-in architecture, Scalaness is not implemented as a plug-in since the needed modifications to the type checker can only be made by direct modification of the compiler code. In addition to static type checking, runtime support is needed to support Scalaness module operations. Also, facilities are required to read nesT modules from the file system and parse them into ASTs, and to write TinyOS image source code files defined by constructed nesT modules at `image` invocations.

6.1 Online Repository and Examples

The Scalaness/nesT compiler and several code examples are available for download from a GitHub repository, accessible via an informative webpage at the following URL:

`http://tinyurl.com/a85z8cu`

The examples include code for applications discussed in Sect. 2 and Sect. 7.

6.2 nesT Type Checking and Program Transformation

The nesT language is treated by two major components in the implementation, the type checker and the nesT-to-nesC rewriting transformation. The nesT type checker was written from the ground up, in contrast to Scalaness type checker which was defined as an extension to the Scala type checker. The rewriting transformation yields TinyOS2-compliant source code, which can be separately compiled.

The nesT language is defined as a subset of the nesC language. An AST yielded by parsing is type checked by our algorithm, which incorporates subtyping and other features not present in nesC type checking. This algorithm is a nearly direct encoding of the type discipline described in Sect. 3. Following type checking, the AST is submitted to a rewriting transformation that imposes semantic disciplines discussed in Sect. 3, in particular type safe casting and array bounds checks, also in nesC. For example, a statement of the form $\mathtt{x} = \mathtt{a}[\mathtt{e}]$ will be rewritten to:

```
int _x = e; if (_x >= a_SIZE) fail(); x = a[_x];
```

where a_SIZE is an automatically generated variable containing the size of a and fail is some user-defined function that handles array bounds check failure.

Source code for nesT module definitions is written in separate files that are included in Scalaness code, as discussed below. This separation is mainly for software engineering purposes, since we imagine that module definitions will be reused in various Scalaness programs.

6.3 Scalaness Module Language Syntax

In order to limit modifications of the Scala compiler and reduce engineering problems in our implementation, we have avoided modifying Scala syntax to represent Scalaness features. Hence, modules are represented as class instances, which must satisfy the following trait:

```
trait NesTModule {
  def image(): Unit    // Generates residual nesC program.
  def +>(m: NesTModule): NesTModule  // Wires this to m.
}
```

This trait is implemented by a NesTModule class that provides the appropriate semantics for wiring and TinyOS image generation, including translation to nesC and file output. This class also manages parsing and storage of nesT ASTs from source code files, and type checking of nesT ASTs.

Any nesT module definition is a subclass of NesTModule. Some subtleties are involved in supporting first class *generic* modules. Instantiation is implemented by method call, but since type and value parameters vary per module, particular modules must define their own parameters and instantiation methods. For example, we would represent the authSend component definition from Sect. 2, line 9 as follows:

```
class authSend extends NesTModule {
  var mt     : MetaType[LiftableType] = _
  var sendk  : LiftableType = _
  def instantiate(m: MetaType[LiftableType], k: LiftableType) =
   { val result = new nodeC; result.mt = m; result.sendk = k }
  "authSend.nt"
}
```

Although the instantiate method and parameter fields must be defined in the implementation at the time of this writing, compiler generation of these definitions is a topic for future work; any modules instantiate method can be easily inferred from its type annotation. Note that the types at which parameters are declared are as general as possible (e.g. s and n are not declared as uints but as LiftableTypes. This is because class definitions support the semantics of Scalaness, not Scalaness type checking (discussed below), and declaring generic parameters at a maximally general type removes interference related to Scala type checking. Finally, note the string literal ″nodeC.nt″ at the end of the definition. This is the file containing the nesT source code definition of the module. The Scala compiler has been modified to input and parse the specified source code when this literal is encountered during the Scala type checking phase.

6.4 Type Annotation and Checking

Scalaness typing relies on native Scala syntax for terms, specifically Scala annotations and singleton types are utilized. Scala annotations allow metadata to be associated with definitions. A module type annotation is of the form @ModuleType(″$\mu\tau$″), where $\mu\tau$ is defined using the syntax of Fig. 3. The compiler-defined ModuleType class automatically associates the type with the identifier immediately following it. In the case of module class definitions, the type is assigned as a class field. In the case of variable definitions, the type (in string literal form) is assigned as a Scala singleton type of the object. For example, the declaration of authSend on line 9 in as in Sect. 2 would be preceded by such an annotation where $\mu\tau$ is the type specified in Example 3.1, and sendM as on line 22 would be annotated with an instance of that type. Similarly, annotations are required on method parameter and result types, if those methods expect nesT modules as arguments or return them, as for the radioC parameter of the authSpecialize method defined in Sect. 2, and the method's commT return type. These requirements reflect the type discipline in Scalaness as specified in Sect. 4, which requires module type annotations at these points.

Scalaness type checking has been implemented as an analysis of these annotations during Scala type checking, piggybacking on that process. When type checking a class that extends NesTModule, the compiler uses its type annotation to perform nesT type checking on the underlying AST representation of the module. When type checking module operations (i.e. at invocations of instantiate, +>, or image), the Scala compiler has been modified to examine operand types for Scalaness type annotations, and to decorate resultant singleton types of these operations with new Scalaness annotations, reflecting the typing rules in Fig. 7. A type checking exception is raised in case this analysis fails. Scalaness type checking does not modify Scala type checking in any other way, so it is a conservative extension of Scala typing.

6.5 Importing nesC Libraries

Our preliminary experiments with nesT show that it is expressive enough to write useful program components. However, any realistic application will need to interact with various libraries written in nesC. One library of critical importance is the TinyOS operating system itself. Our current solution is to allow non-generic nesC components to be treated as nesT modules as long as they only use or provide commands, which are interpreted as nesT imports and exports. Support for specializable generic nesC library components is a topic for future work. Events can be accessed through "shim" modules provided by the user, since used or provided events are really just syntactic sugar for provided or used commands respectively. A library component defined in a file LibraryC.nc can be defined as a nesT module as follows:

```
object LibraryC extends NesCModule { external("LibraryC.nc") }
```

Note that nesC code imported in this way is not type checked by the Scalaness/nesT compiler, since nesT is a strict subset of nesC. Rather, the programmer type annotates the shimmed module using a @ModuleType annotation as for other module definitions, and the compiler trusts that the annotation is correct. This introduces a possibility for type safety failure in our system, if the imported code contains a type error. A possible long term goal would be a complete re-write of TinyOS in nesT, yielding full type safety of all sensor code, but this is well beyond our current scope.

7. Application Example: Staged Authorization and Access Control

In [5] the SpartanRPC architecture for link-layer resource authorization in TinyOS-based WSNs is developed (as an extension of [4]). In SpartanRPC, resources are accessed by link-layer remote procedure calls (RPC) which require authorization for use. Users

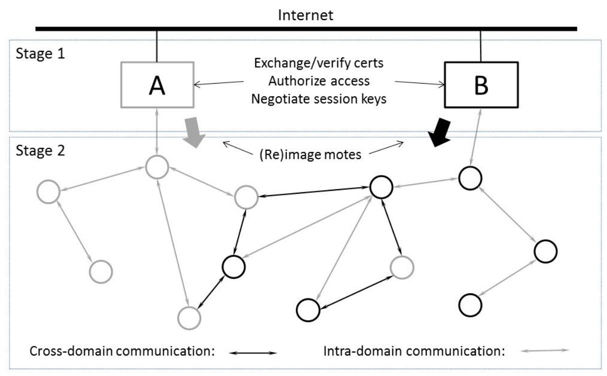

Figure 8. Staging Authorization and Authorized Access in a Multi-Domain WSN.

are authorized by communicating credentials to the provider, expressed in an authorization logic based on RT [17] and implemented using TinyECC [18] public key signatures. SpartanRPC supports an "open world" security model, allowing WSNs in different security domains to interact without sharing secrets *a priori*. However, public key encryption and signature verification is very expensive in a WSN: a single signature may take several minutes to verify. Hence, session keys are negotiated for ongoing resource access (using a TinyECC-based Diffie-Hellman protocol).

In this section we describe a re-implementation of the SpartanRPC protocol in Scalaness/nesT that addresses several shortcomings of SpartanRPC, and will thus serve to illustrate the power of Scalaness/nesT. The central idea, illustrated in Fig. 8, is that responsibility for authorization on the basis of public key credentials is offloaded from the WSN to a Scalaness program running on a hub device or lab computer. We assume a WSN comprising two subnetworks under control of distinct security domains A and B. Each domain also controls a lab or hub device which is in communication with WSN nodes in their domain, either prior to or during deployment. These devices are in communication with each other over the Internet, and exchange authorization credentials for their domain over that medium in the first-stage Scalaness program. Each device then confirms authorization for resource access according to their own domain's policies, and subsequently they negotiate session keys over the Internet. These keys are then used to specialize nesT code for imaging on WSN nodes. The overall architecture of this application represents a concrete realization of the ideas of Fig. 1, and also expands on and implements the idealized example presented in Sect. 2.

Note that our current implementation assumes nodes are programmed in the lab since we have yet to implement a secure OTA program dissemination library; the Deluge protocol has a secure OTA reprogramming extension [9] that we plan on using to guarantee code dissemination is itself secure.

Evaluation on Snowcloud To empirically evaluate the staged implementation of SpartanRPC in Scalness/nesT, we have implemented and tested both the original SpartanRPC as well as the Scalaness/nesT staged version in our deployed Snowcloud WSN system architecture. Mobile gateway devices as pictured in Fig. 2 are provided to Snowcloud system users for data gathering, and are also used by system administrators for controlling sampling rates. The hardware both of these so-called "harvester" devices, the same for users and administrators, is equipped with a mote for establishing network communication. When the device is introduced to the sensor network, the two together comprise a single network with two distinct security domains – the sensor node subnetwork, and the subnetwork of the single device mote. The mote on harvester devices provided to system users is supplied with credentials for collecting data, but not modifying network control, whereas system administration harvester motes are supplied with stronger credentials for both functions.

The original and Scalaness/nesT versions of this application can be compared both in terms of performance and user experience. In the unstaged version, the SpartanRPC protocol requires an initial network configuration period when credentials are exchanged and verified. Since a single TinyECC signature requires at least 90 seconds to verify on the Crossbow TelosB platform [5] with a fully dedicated processor, there is an initial network "warmup" period of at least a few minutes. Also, in the unstaged version, upon first invocation of an RPC service Diffie-Hellman is used in the network to negotiate a session key. In the staged version, credential exchange, validation, and session key negotiation are all performed on the high-powered hub. For this reason, mote code size in ROM is significantly reduced. There are differences in RAM usage as well, due to authorization overhead in the unstaged version and also the storage of key material in RAM vs. ROM, since specialization of code with key material in the staged version allows the latter. Note that this difference is intensified by scale and the number of keys (i.e. RPC services) needed by an application. Lower RAM and ROM usage can have significant performance impacts on deployed code. In the following table we summarize RAM and ROM usage for the harvester and sensor node images for three software versions: one with no security mechanisms in place, one with unstaged SpartanRPC protocols in place, and one generated by Scalaness evaluation in our staged version of the SpartanRPC protocol.

		Unsecured	Unstaged	Staged	Savings
Sensor:	ROM	36254	48616	36596	25%
	RAM	2868	5417	3038	44%
Harvester:	ROM	24316	35834	24436	32%
	RAM	2274	4771	2402	50%

The "Savings" are the percent reduction from unstaged to staged secure implementation, and these numbers show the potential for saving both RAM and ROM space is significant. From the perspective of user experience, the staged version of this application is more convenient, since no initial authorization period is needed when the harvester is first introduced to the network. The staged version also exposes the system to fewer bugs and failures that would be obstacles to the primary goal of data collection.

8. Conclusion

We have introduced Scalaness/nesT, a two stage programming system for wireless sensor networks. Our system provides a powerful programming environment for dynamically specializing and composing nesC modules in a type safe way; any type correct Scalaness program will generate only type correct residual programs.

8.1 Related Work

We do not review the broader topic of sensor network programming here; the reader is referred to [27] for a broader perspective.

We follow the foundational $\langle$ML$\rangle$ work in our language design [19]; Sect. 5 discusses how it serves as the theoretical underpinning of our approach. The primary aim of this work is to make the theoretical insights of $\langle$ML$\rangle$ more practical. We accomplish this by making a sensor language nesT that is based on the design of nesC, and by implementing Scalaness and nesT and testing the framework on examples.

The potential of applying metaprogramming to sensor networks was explored in the functional sensor language Flask [23]. The main motivation for designing Flask was to allow FRP-based stream combinators to be pre-computed before network deployment. The Flask designers did not focus on computing precise types for the object stage code at the meta stage, so cross-stage static type checking is not performed – it is possible to generate ill-typed Flask object code. Hume [15] is a DSL for real-time embedded device programming. It includes a metaprogramming layer but that layer is more like nesC's configuration files in that there is a very restricted syntax for a few special metaprogramming operations including component wiring, macros, and code templating.

MetaML [30, 31] and MetaHaskell [22] have each been promoted as effective foundations for embedded systems programming with type safety, but neither addresses type specialization or dynamic type construction. MetaHaskell does support heterogeneous language staging, the lower stage language is defined by a plug-in and several instantiations have been defined including one for a low-level C-like language.

Actor based sensor metaprogramming has been studied in [6]; this work shares our focus on high level dynamic reprogrammability but is untyped. More broadly, meta programming is known to be useful for increasing the efficiency of systems applications. One example is Tempo [8], a system that integrates partial evaluation and type specialization for increasing efficiency of systems applications. Ur [7] allows for type safe meta programming for web applications.

The units of staged code composition in nesT programming are *modules*. Countless different module systems exist, but they are primarily designed to achieve separate compilation and sound linking [2]. Our different design goal leads to different design choices in nesT modules. For example, data crossing nesT module boundaries needs to conform to the property of process separation, a non-issue in standard module system designs. In addition, nesT modules allow values/types across the boundary of modules to be flexibly constructed, including dynamic construction of types, to achieve maximal flexibility of cross-stage specialization. Module systems such as ML modules [20] and Units [10] allow types to be imported/exported as we also support; there are several features of ML modules including type hiding that we do not aim to support. nesT modules are more expressive in their support of first class modules as values and the possibility of dynamic construction of "type exports." That said, first class modules are not new [1, 24], we only claim novelty in their application to program staging and the incorporation of dynamic type construction.

The type parametricity of System F and $F_{\leq}$ [3], and the practical type systems it inspired such as Java's generics, do not treat types as first class values as we do. C++ templates support types as meta values in template expansion, but type safety of generated code is not guaranteed without full template expansion. Concepts [14] improves on this, but types are still not first class values.

References

[1] D. Ancona and E. Zucca. A calculus of module systems. *Journal of functional programming*, 11:91–132, 2002.

[2] Luca Cardelli. Program fragments, linking, and modularization. In *POPL*, pages 266–277, 1997.

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[4] Peter Chapin and Christian Skalka. SpartanRPC: Secure WSN middleware for cooperating domains. In *MASS*, November 2010.

[5] Peter Chapin and Christian Skalka. Technical report, University of Vermont, In submission, 2013. `http://www.cs.uvm.edu/~skalka/skalka-pubs/chapin-skalka-spartanrpctr.pdf`.

[6] Elaine Cheong. *Actor-Oriented Programming for Wireless Sensor Networks*. PhD thesis, University of California, Berkeley, 2007.

[7] Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *PLDI*, 2010.

[8] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé. Tempo: specializing systems applications and beyond. *ACM Comput. Surv.*, page 19, 1998.

[9] Prabal K. Dutta, Jonathan W. Hui, David C. Chu, and David E. Culler. Securing the deluge network programming system. In *IPSN*, pages 326–333, 2006.

[10] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *PLDI*, 1998.

[11] Jeffrey Frolik and Christian Skalka. Technical report, University of Vermont, 2013. `http://www.cs.uvm.edu/~skalka/skalka-pubs/frolik-skalka-snowcloudtr.pdf`.

[12] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI*, 2003.

[13] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193(1-2):75 – 96, 1998.

[14] Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Gabriel Dos Reis, Bjarne Stroustrup, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *OOPSLA*, 2006.

[15] Kevin Hammond and Greg Michaelson. Hume: A domain-specific language for real-time embedded systems. In *Conference on Generative Programming and Component Engineering (GPCE)*, pages 37–56. Springer-Verlag, 2003.

[16] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[17] Ninghui Li and John C. Mitchell. RT: A role-based trust-management framework. In *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition*, pages 201–212, 2003.

[18] An Liu and Peng Ning. Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks. In *IPSN*, pages 245–256, 2008.

[19] Yu Liu, Christian Skalka, and Scott Smith. Type-specialized staged programming with process separation. *Higher-Order and Symbolic Computation*, pages 341–385, 2011.

[20] D. MacQueen. Modules for Standard ML. In *Proceedings of ACM Conference on Lisp and Functional Programming*, 1984.

[21] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[22] Geoffrey Mainland. Explicitly heterogeneous metaprogramming with MetaHaskell. In *ICFP*, 2012.

[23] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: staged functional programming for sensor networks. In *ICFP*, 2008.

[24] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of standard ML modules with subtyping and inheritance. In *POPL*, 1991.

[25] C. David Moeser, Mark Walker, Christian Skalka, and Jeff Frolik. Application of a wireless sensor network for distributed snow water equivalence estimation. In *Western Snow Conference*, 2011.

[26] Thomas Molhave and Lars H. Petersen. Assignment Featherweight Java. Master's thesis, University of Aarhus, 2005.

[27] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks. *ACM Computing Surveys*, 43:19:1–19:51, April 2011.

[28] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *POPL*, 2002.

[29] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, second edition*. Artima, Inc, 2011.

[30] Walid Taha. Resource-aware programming. In *ICESS*, pages 38–43, 2004.

[31] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *PEPM*, pages 203–217, 1997.

[32] Rebecca Willett, Aline Martin, and Robert Nowak. Backcasting: adaptive sampling for sensor networks. In *IPSN*, pages 124–133, 2004.