Type-Specialized Staged Programming with Process Separation

Yu David Liu
SUNY Binghamton
davidl@cs.binghamton.edu

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

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

Abstract

Staging is a powerful language construct that allows a program at one stage to manipulate and specialize a program at the next. We propose $\langle \text{ML} \rangle$ as a new staged calculus designed with novel features for staged programming in modern computing platforms such as embedded systems. A distinguishing feature of $\langle \text{ML} \rangle$ is a model of process separation, whereby different stages of computation are executed in different process spaces. Our language also supports dynamic type specialization via type abstraction, dynamic type construction, and a limited form of type dependence. $\langle \text{ML} \rangle$ is endowed with a largely standard metatheory, including type preservation and type safety results. We discuss the utility of our language via code examples from the domain of wireless sensor network programming.

Categories and Subject Descriptors D Software [*D.3 Programming Languages*]: D.3.3 Language Constructs and Features

General Terms Design, Languages, Theory.

Keywords Staged Programming, Type Specialization, Polymorphism.

1. Introduction

The power of generic programming is realized through formal mechanisms of abstraction. From features as fundamental as function parameterization to more recent schemes such as metaprogramming, generic programming produces code that is more flexible, safe, and efficient by allowing principled generalization over and recombination of program elements [13]. In this paper, we explore a programming language design that combines generic programming mechanisms to obtain code efficiency and to support useful design patterns for programming embedded systems, especially software for wireless sensor networks (WSNs). In particular we explore *staged programming* and *type genericity* as principled techniques to organize and optimize program code through deployment steps in embedded systems.

1.1 Staging Deployment Steps with Process Separation

In [13] staged programming is identified as a so-called *metaprogramming* species of generic programming. There is a long history of explicit support for staging in programming languages [35, 9, 8, 31, 3, 26, 25, 5, 34]. These language designs all admit program code itself as a data type, and support generalization and composition/specialization of code via some form of code abstraction. Since running of code-as-data is also typically supported, program staging allows a principled definition of program generation. The language we present in this paper, $\langle \text{ML} \rangle$, is an extension of a core-ML calculus with support for program staging. It borrows ideas from previous systems but provides unique support for using staging to define *deployment cycles* in a multi-tier embedded systems architectures. WSNs in particular are composed of a vast number of sensor nodes (so-called *motes*) of limited resources connected to one or more *hubs* – larger computers running e.g. Linux, and the typical sensor network deployment occurs in two stages, with the first stage running on the hub controlling the deployment of mote code at the second stage [15, 11, 18, 19]. Hence, along with previous authors [32] we argue that staging abstractions provide a principled means to express typical design patterns in embedded systems such as WSNs. Furthermore, staging offers significant efficiency benefits for WSNs since it allows inlining of pre-computed (on the hub) data and functionality in specialized "later stage" code (for mote deployment). This is important since energy and computational resources in such a power-constrained environment are precious.

Since we view stages as models of deployment steps in a multi-tiered hardware setting, it follows that each stage must be envisioned as executing within a distinct process space. While the $\langle \text{ML} \rangle$ model is closely related to MetaML [35], a well known extension of ML with support for staging, it differs fundamentally from that system in part because *cross-stage persistence* is disallowed in $\langle \text{ML} \rangle$. In essence, cross-stage persistence is a feature that allows values to migrate freely between stages through standard function abstraction and application. This is especially problematic in a multi-tiered embedded system with mutable state, since sharing memory between processes is not feasible and hence memory references cannot be sensibly interpreted between process spaces (i.e. stages). We prevent cross-stage persistence through a novel static type analysis. At the same time, we allow composition and specialization of stateful code by allowing values to be "lifted" between stages in a principled manner that incorporates a form of data serialization (*a.k.a.* marshalling). It is important to consider state in this setting since embedded systems languages such as nesC [11] make heavy use of it.

1.2 Type Genericity

Another sort of genericity we explore in this paper is genericity by type as defined in [13]. We support type genericity in two related dimensions: *first*, we allow specialization of the types of declared variables through a form of parametric polymorphism, and *second*, we allow dynamically construction of types of programs by introducing type-indexed terms, i.e. by treating types as first class values.

The usefulness of these features is determined by common design patterns in embedded systems such as WSNs. For example, upon deployment, WSNs can initially refine node address sizes via

WGP'09, August 30, 2009, Edinburgh, Scotland, UK.

locally unique address assignment protocols that aim to minimize node address sizes. This allows bandwidth reduction during communications, by reducing message sizes [29]. We envision that type genericity will work in conjunction with program staging in ⟨ML⟩ to maximize program efficiency in this scenario. We can leverage our ⟨ML⟩ staging abstraction to formalize the interaction between address assignment protocols and subsequent network communications, while our support for type specialization will allow message structure types to be dynamically defined by address assignment and passed as type parameters to network communication stages. We use this scenario as a running example throughout the remainder of the paper.

The work presented here is foundational. While our ultimate goal is to port these ideas to realistic languages for programming embedded systems such nesC, our current goal is to explore them in a theoretical setting comprising a simple core calculus and associated metatheory. We are especially concerned with establishing type safety results in the core language model. Our language, type theory, and metatheory are presented in the following Sections 2 through 5. To illustrate how our proposed system can support WSN applications programming we present and discuss an extended example in Sec. 6.

2. The Core Language

In this Section we define and discuss the core ⟨ML⟩ syntax and semantics. We will later discuss the type system in Sec. 3 and then the addition of mutation and state in Sec. 4.

2.1 ⟨ML⟩ Syntax and Semantics

The ⟨ML⟩ language syntax is defined in Fig. 1, including *values* v, *expressions* e, *evaluation contexts* E, *types* τ, *type coercions* Δ, and *type environments* Γ. Expression forms directly related to type genericity, including type abstraction $\Lambda t \preccurlyeq \tau.e$, a "type let" **tlet**, type-as-terms and casting, are discussed more in Sec. 3. Our initial focus is on our three expression forms for staged computation. The form $\langle e \rangle$ represents the *code* e, which is treated as a first class value. The form **run** e evaluates e to code and then runs that code (in its own process space). The form **lift** e evaluates e to a value, and turns that value into code, i.e. "lifts" it to a later stage. We omit the "escape" operator, *e.g.* ~e of MetaML; since this form is common in staged programming languages we discuss this design choice more below.

In order to prevent cross-stage persistence, central to our approach is the definition of term substitution. Our substitution should ensure "stage conformity", i.e. we can only substitute code into code, and code stage levels should be coordinated in substitution. To achieve this we define $\langle e \rangle[\langle e' \rangle/x] = \langle e[e'/x] \rangle$, and make $\langle e \rangle[e'/x]$ be undefined if e' is not code. This definition forces free variables in $\langle e' \rangle$ to be instantiated with code only, and as a consequence bound and free variables in e' have a different meaning: the bound variables range over non-code expressions and the free variables range over code expressions. More completely, substitution $e[e'/x]$ can be defined by case analysis on e, with interesting cases as follows:

$$
\begin{array}{rcll}
x[e'/x] & = & e' & \\
y[e'/x] & = & y & \text{if } x \neq y \\
\langle e \rangle[\langle e' \rangle/x] & = & \langle e[e'/x] \rangle & \\
(e_1 e_2)[e'/x] & = & (e_1[e'/x])(e_2[e'/x]) & \\
(\lambda x : \tau.e)[e'/x] & = & \lambda x : \tau.e & \\
(\lambda y : \tau.e)[e'/x] & = & \lambda y : \tau.e[e'/x] & \text{if } x \neq y \\
(\Lambda t \preccurlyeq \tau.e)[e'/x] & = & \Lambda t \preccurlyeq \tau.(e[e'/x]) & \\
 & \vdots & &
\end{array}
$$

We can similarly define type substitutions $\tau[\tau'/t]$ in a standard manner.

The operational semantics of ⟨ML⟩ are then defined in Fig. 2 in terms of substitutions, as a small-step reduction relation $\rightarrow$. This relation is defined in a mutually recursive fashion with its reflexive, transitive closure denoted $\rightarrow^\star$. Note that the RRun rule models process separation by treating the running of code as a separate and complete evaluation process; this separation will become more clear when we consider mutation and state in Sec. 4. The user-supplied function δ axiomatizes our interpretation of program constants $\mathbf{c}$. The semantics of casting are predicated on a notion of typing defined in the following section.

$$
\begin{array}{rcl}
x \in \mathcal{V},\ t \in \mathcal{T} & & \\
v & ::= & \mathbf{c} \mid x \mid \lambda x : \tau.e \mid \Lambda t \preccurlyeq \tau.e \mid \langle e \rangle \mid \tau \\
e & ::= & v \mid (\tau)e \mid e\,e \mid \textbf{tlet}\ t \preccurlyeq \tau = e\ \textbf{in}\ e \mid \textbf{run}\ e \mid \textbf{lift}\ e \\
E & ::= & [\,] \mid Ee \mid vE \mid \textbf{tlet}\ t \preccurlyeq \tau = E\ \textbf{in}\ e \mid (E)e \mid (v)E \\
\tau & ::= & t \mid \gamma \mid \textbf{type}[\tau] \mid \langle \cdot\tau\cdot \rangle \mid \Pi t \circ \Delta.\tau \mid \tau \rightarrow \tau \\
\Delta & ::= & \varnothing \mid \Delta; t \preccurlyeq \tau \\
\Gamma & ::= & \varnothing \mid \Gamma; x : \tau
\end{array}
$$

Figure 1. ⟨ML⟩ Term and Type Syntax

$$
\frac{\delta(\mathbf{c}, v) = e}{\mathbf{c}\ v \rightarrow e}\ \text{RConst} \qquad (\lambda x : \tau.e)v \rightarrow e[v/x]\ \text{RApp}
$$

$$
\textbf{tlet}\ t \preccurlyeq \tau = \tau'\ \textbf{in}\ e \rightarrow e[\tau'/t]\ \text{RTLet} \qquad \frac{v : \tau}{(\tau)v \rightarrow v}\ \text{RCast}
$$

$$
(\Lambda t \preccurlyeq \tau.e)\tau' \rightarrow e[\tau'/t]\ \text{RApp}_\Pi \qquad \frac{e \rightarrow^\star v}{\textbf{run}\ \langle e \rangle \rightarrow v}\ \text{RRun} \qquad \textbf{lift}\ v \rightarrow \langle v \rangle\ \text{RLift}
$$

$$
\frac{e \rightarrow e'}{E[e] \rightarrow E[e']}\ \text{RContext}
$$

Figure 2. ⟨ML⟩ Core Operational Semantics

2.2 Discussion

For the convenience of our discussion, all examples below only consider two stages, which we call the *meta stage* and the *object stage*, following standard terminology in meta programming. ⟨ML⟩ in fact supports arbitrary stages.

Exploiting MetaML-style staging Two primitive MetaML expressions are directly reflected in ⟨ML⟩, namely the bracket expression $\langle e \rangle$, and the execution expression **run** e. Indeed a canonical MetaML example, a staged list membership testing function, can be written in ⟨ML⟩ as in Fig. 3. Rather than testing membership directly, the execution of the function produces a piece of membership testing code, which is often more efficient.

The ability to specialize code is very useful for resource-constrained platforms, such as wireless sensor networks. In this particular usage, we imagine that the meta stage is on the hub that creates code to deploy and run on the sensors, and the object stage

$$
\begin{array}{l}
member : \langle \cdot \textbf{int} \cdot \rangle \rightarrow \textbf{int list} \rightarrow \langle \cdot \textbf{bool} \cdot \rangle \\
member\ x\ l = \\
\quad \textbf{if}\ l\ =\ nil\ \textbf{then}\ \langle false \rangle\ \textbf{else} \\
\quad\quad \textbf{let}\ h\ =\ \textbf{lift}\ (hd\ l)\ \textbf{in} \\
\quad\quad \textbf{let}\ tl\ =\ member\ x\ (tail\ l)\ \textbf{in} \\
\quad\quad\quad \langle x = h || tl \rangle \\
\textbf{run}(member \langle sense\_data \rangle [0,1])
\end{array}
$$

Figure 3. ⟨ML⟩ Definition of `member` Function

is the sensor node execution environment. For example, suppose each sensor node needs to frequently test membership of the result of $sense\_data$ in a fixed list of say $[0, 1]$. Rather than invoking a standard membership function at each sensor node and incurring the run-time overhead of stacks and **if** . . . **then** . . . **else**, we only need to execute the program in Fig. 3 on the hub (a computer with far fewer resource constraints). What will be deployed to individual sensor nodes is only the argument of the **run** expression, $member\ \langle sense\_data \rangle\ [0, 1]$, which will be evaluated on the hub to:

$$\langle sense\_data = 0\ ||\ sense\_data = 1\ ||\ \textbf{false} \rangle$$

Value Migration via Lift and Run The semantics of ⟨ML⟩ substitution-based term reduction itself disallows cross stage persistence; for example the term $(\lambda x : \textbf{uint}.\langle x \rangle)3$ is stuck since 3 cannot be substituted into the code $\langle x \rangle$. But it is of course necessary to allow migration of values across stages. For our purposes **lift**ing a value from the meta to the object language and **run**ning object code from the meta level are sufficient and indeed function as duals. For example in Fig. 3, the expression **lift** $(hd\ l)$ lifts the value of the meta-stage to the object-stage, which subsequently can be compared with a value in that stage $(x = h)$.

A Simple Model with No Escape MetaML has an escape expression ~e that can "demote" e from the object stage back to the meta stage. For instance, rather than writing

$$\langle x = h\ ||\ tl \rangle$$

as in Fig. 3, MetaML programmers would write the equivalent

$$\texttt{<}\sim x = \sim h\ ||\ \sim tl\texttt{>}$$

Intuitively, the call-by-value semantics of our language will enforce arguments being evaluated first before they are "spliced" together to form the object code. This is precisely what an object code with escape expressions inside would do. The previous example shows that the escape operator in MetaML is perhaps not as essential as it seems in many practical programming situations: a MetaML expression $\langle C[\sim e] \rangle$ can be re-written as $(\lambda x.\langle C[x] \rangle)(e)$ where C is a program context.

The support of an escape-like operator in a meta language – particularly the support of free variables occurring inside such an expression – is known to lead to significant complexities for static type checking [26, 5, 34]. Since most of the programs we are interested in can be written with the aforementioned encoding in mind, we choose not to support the escape operator in our language.

3. Types and Type Specialization

In this Section we focus on our type system, again beginning with formalities and then moving on to higher level discussion. Briefly, our goals in this type theory are to support type genericity as discussed in Sec. 1.2, and also to statically disallow cross-stage persistence. We delay our definition of type validity and associated metatheory until Sec. 5, aiming first to provide a basic understanding of the system.

3.1 Types in Terms

As discussed in Sec. 1.2, *type specialization* is essential for our envisioned application space. This specialization has two dimensions: first, we should be able to specialize the types of procedures, and second, we should be able to dynamically construct types of programs based on certain conditions.

For the first purpose we posit a form of bounded type abstraction, denoted $\Lambda t \preccurlyeq \tau.e$; the application of this form to a type value may result in type specialization of e. We use a bound on the abstraction to provide a closer type approximation (hence better static optimization of code) in the body of the abstraction.

For the second purpose we introduce types as values, and a **tlet** construct for dynamically constructing types. We introduce this latter form to obtain a clear separation of types and expressions and promote well-typed type construction. We have discussed the usefulness of these forms in Sec. 1.2, and examples in Sec. 3.2 and Sec. 6 will further illustrate them.

Since type abstractions can be applied to first class type values, a System-F$_{\leq}$ style approach where type instantiation arguments are statically declared is not sufficient for our system. Rather, we assign to type abstractions a restricted form of type dependence [20], hence the Π type syntax of type abstractions. Intuitively, in a call-by-value semantics our Λ abstractions are applied to "fully constructed types" at run time; statically, we have that the type of applied Λ abstractions depends on the first-class type argument. We observe that type dependence and program staging have often been used to achieve program efficiency in other contexts such as compiler optimization [7, 2].

3.1.1 Type Forms and Type Coercions

As usual we must define a different type form for each class of values in our language. In addition to a Π type form for type abstractions, we have standard term function type forms $\tau \rightarrow \tau$ and base types γ for user-defined constants. We also introduce a type form **type**$[\tau]$, that represents the type of dynamically constructed type values. Intuitively, **type**$[\tau]$ represents the set of all types that are subtypes of τ, considered as values. Since we consider code a value, type-of-code has a denotation $\langle \cdot \tau \cdot \rangle$, where τ is the type of value that will be returned if the code is run.

The Π type form of type abstractions comprises a subtyping coercion Δ which is a function from type variables to types. In a type $\Pi t \circ \Delta.\tau$ the coercion Δ expresses the type variable bounds on t, and also expresses bounds related to **tlet**-declared type variables in the body of the abstraction (that may escape their static scope). In any type $\Pi t \circ \Delta.\tau$ we consider Π to bind all variables in $\text{dom}(\Delta)$, and we equate types up to α renaming of these variables.

Intuitively, a coercion Δ defines upper bounds on a set of type variables; we require that these bounds are not recursive. Any coercion induces a set of subtyping relations in a standard manner extended to comprise also types of type abstractions, code, and type values; formally we define the subtyping relation $\Delta \vdash \tau \preccurlyeq \tau'$ in Fig. 4. We write $\Delta; t \preccurlyeq \tau$ (resp. $\Gamma; x : \tau$) to denote the function that maps t to τ (resp. x to τ) and agrees with Δ (resp. Γ) on all other points. Abusing notation, we write $\Delta; \Delta'$ to denote the pointwise extension of Δ with Δ'. To clarify type substitutions, we define:

$$\frac{t' \cap \text{dom}(\Delta) = \varnothing \qquad \text{dom}(\Delta') = \text{dom}(\Delta) \qquad \forall t \in \text{dom}(\Delta').\Delta'(t) = \Delta(t)[\tau'/t']}{(\Pi t \circ \Delta.\tau)[\tau'/t'] = \Pi t \circ \Delta'.(\tau[\tau'/t'])}$$

We also make the following definition for brevity in the typing rules, which is a slight variant of the AND subtyping rule:

$$\frac{\text{ANDSUB} \quad \forall t \in \text{dom}(\Delta').\Delta \vdash t[\tau/t'] \preccurlyeq (\Delta'(t))[\tau/t']}{\Delta \vdash \Delta'[\tau/t']}$$

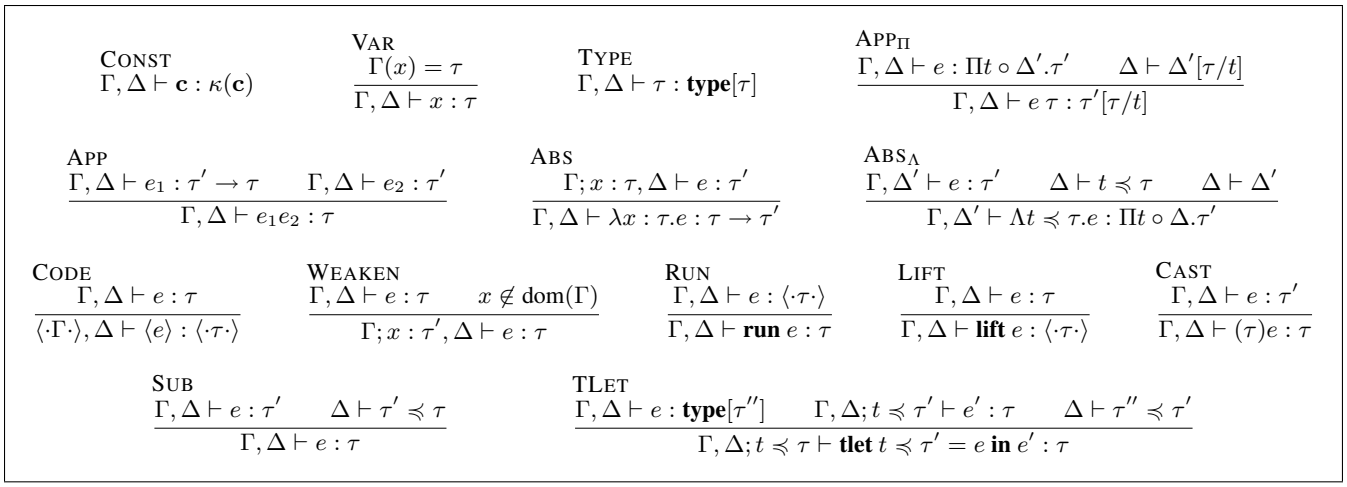

Figure 5. Type Judgement Rules

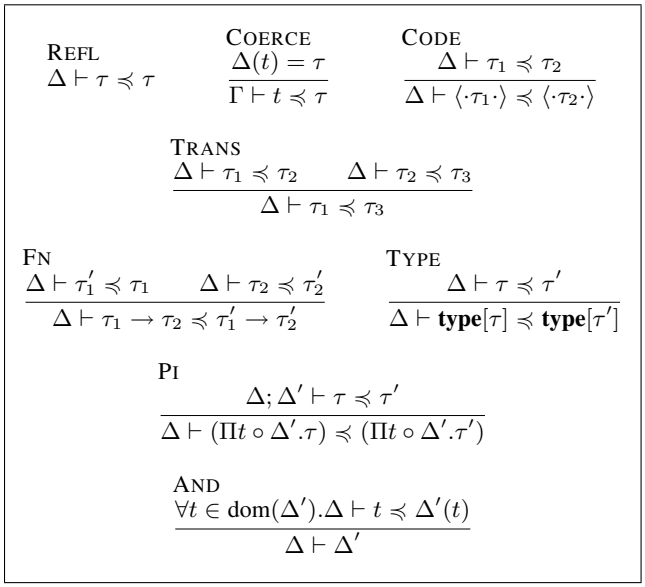

Figure 4. Subtyping Rules

The reader will note that bound coercions must be equivalent to compare Π types via subtyping as specified in Fig. 4. This restriction is imposed to support decidability of typing in the presence of bounded polymorphism; it is well-known that allowing variance of type variable bounds in this relation renders subtyping undecidable [12].

3.2 Discussion

Type Abstraction and Application for Staged Code Our running example introduced in Sec. 1.2 is that of object level code parameterized by a pre-computed address type. In $\langle$ML$\rangle$ this can be written as

$$\Lambda addr\_t.\langle \lambda addr : addr\_t.e \rangle$$

Type theoretically, the construct here is a standard type abstraction mechanism as is found in System F, with the twist that in $\langle$ML$\rangle$ type arguments can be dynamically constructed, not just statically declared. In this sense, our type abstraction and application mechanism can be viewed as a very simple form of type dependence. Type application can then be performed to produce staged code with a concrete type, such as

$$(\Lambda addr\_t.\langle \lambda addr : addr\_t.e \rangle)\mathbf{uint8}$$

This code will be executed on the meta stage, so that when this code is executed on sensor nodes, variable $addr$ will have **uint8** type.

Type Bounds and Subtyping The benefit of static type checking over staged code has been widely discussed in recent efforts in meta programming [23, 3, 37, 5, 34]. However, note that type checking the code above would be restrictive since $addr\_t$ can be instantiated with *any* concrete type, any variable of type $addr\_t$ would be effectively assigned a universal type and hence be unusable.

To address this problem in a familiar fashion, $\langle$ML$\rangle$ allows programmers to assign a bound on the abstracted type. For instance, the message send code defined previously can be refined as:

$$\Lambda addr\_t \preccurlyeq \mathbf{uint}.\langle \lambda addr : addr\_t.e \rangle$$

With this bound, the type system can assume the type of $addr$ is at least **uint** when e is typechecked.

Our form of type abstraction is related to standard bounded polymorphism of System $F_{\leq}$, except that bounds are not recursive in our system and also we allow types to be constructed dynamically, as discussed below.

Types as Expressions Unlike System $F_{\leq}$ where types and terms do not mix, and all type instantiation occurs statically, types are first-class citizens in $\langle$ML$\rangle$, and can be assigned, passed around, stored in memory, compared, *etc*.

The design choice here is driven by our application needs. In systems programming, it is not uncommon to see conditional macros used over types, such as

```
# ifdef v typedef T {...} else typedef T{...}
```

The connection between macros and staged programming is widely known [10], except that most people – including the macros users themselves – complain they are not expressive enough. Treating types as values in $\langle$ML$\rangle$ provides programmers with a flexible way to define constructs such as the above (so much so that arbitrary programs are allowed to define how the T above can be typedefed), at the same time *preserving static type safety* as demonstrated in Sec. 5. As a result, the static type system of our language differs from System $F_{\leq}$ and its descendants such as Java generics. That is, type parameters abstracted by Λ are instantiated not with static

types, but with types as first class values. In this sense the system incorporates a simple form of type dependence. For example, consider the following ⟨ML⟩ program:

$$rtt = \textbf{tlet}\ tcond \preccurlyeq \textbf{uint32} = (\textbf{if}\ e0\ \textbf{then uint16 else uint32})\ \textbf{in}\ (\Lambda addr\_t \preccurlyeq \textbf{uint32}.\langle \lambda addr : addr\_t.e \rangle)\ tcond$$

Here the **tlet** $\ldots = \ldots$ **in** expression is similar to a **let** $\ldots = \ldots$ **in** , except that it binds types. The binder **tlet** serves a critical purpose in the formalism: any type-valued expression such as the above if-then-else cannot directly appear in another type; only its **tlet**-ed name can. This keeps expressions out of the type grammar: for example, (**if** $e0$ **then uint16 else uint32**) $\rightarrow$ **uint32** is ill-formed and such types never can be written. Assuming the return type of a typical $send$ function is an ACK of fixed **result_t** type, our language will type the example above as $rtt : \langle \cdot tcond \rightarrow \textbf{result\_t} \cdot \rangle$, under type constraint $tcond \preccurlyeq$ **if** $e0$ **then uint16 else uint32**.

Notation **type**[**uint**] means any type less than **uint**; **type**[τ] in general has the following meaning:

$$\textbf{type}[\tau] = \{\tau' \mid \tau' \preccurlyeq \tau\}$$

These range types are used to type type-valued expressions; for example, in typing the above we would need to show:

$$\textbf{if}\ e0\ \textbf{then uint16 else uint32} : \textbf{type}[\textbf{uint32}]$$

which is straightforward since **uint16** $\preccurlyeq$ **uint32**.

Casting To "close the loop" on runtime-dependent types as defined above we need to find a way to put initial members in these types in spite of not knowing what value (type) they will be at runtime. The runtime condition is crucial to define a member of a runtime-decided type in the code, for example the $e0$ condition in the above example. In this example, the rtt function must take some value $v : tcond$ as argument, where $tcond$ is a type whose value depends on the runtime value of $e0$. Conditional types have been defined [1, 28] which are suited for this purpose, but for this simple presentation we opt for a typecast which is more expressive but incurs a runtime check. For this particular example we could write:

$$rtt((tcond)5)$$

which will cast 5 to $tcond$, which at the time the cast runs will either be **uint16** or **uint32** as appropriate and so will succeed.

4. Records, State, Serialization, and Semantics

In this section we extend the core functional language with records and mutable store, along with a notion of serialization that will allow mutable data to be shared between stages. The reason for this is that we aim to port the ideas presented in this paper to languages such as nesC, where state and `struct` definitions are fundamental. Furthermore, state presents interesting technical challenges in the presence of ⟨ML⟩-style staging where we assume that distinct stages represent distinct process spaces.

We introduce new record and state expression forms, as well as an expression sequence form that is a semicolon-delimited vector of expressions, a unit value $()$, and a special form of let-expression helpful for representing syntactic stores that makes subsequent definitions more succinct; this technique follows previous work such as [16].

$$\begin{array}{rcll} s & ::= & \varnothing \mid s; e & \textit{(sequences)} \\ v & ::= & \ldots \mid \{\ell_1 = v_1; \ldots; \ell_n = v_n\} \mid x & \textit{(values)} \\ e & ::= & \ldots \mid \{\ell_1 = e_1; \ldots; \ell_n = e_n\} \mid e.\ell & \\ & & \mid \textbf{ref}\ e \mid e := e \mid !e \mid s & \textit{(expressions)} \\ \tau & ::= & \ldots \mid \{\ell_1 : \tau_1; \ldots; \ell_n : \tau_n\} \mid \textbf{ref}\ \tau & \textit{(types)} \\ \\ D & ::= & [\,] \mid \textbf{let}\ z = \textbf{ref}\ v\ \textbf{in}\ D & \textit{(decl. contexts)} \\ m & ::= & \varnothing \mid m; z := v & \textit{(mutations)} \\ h & ::= & D[m] & \textit{(syntactic stores)} \end{array}$$

Syntactic stores may be interpreted as a mapping from variables to values via the dom and lkp functions defined as follows. We write $\mathrm{dom}(h)$ to denote the domain of a store h:

$$\begin{array}{rcl} \mathrm{dom}(s) & = & \varnothing \\ \mathrm{dom}(\textbf{let}\ z = \textbf{ref}\ v\ \textbf{in}\ h) & = & \{z\} \cup \mathrm{dom}(h) \end{array}$$

We write $\mathrm{lkp}\ z\ h$ to denote the value associated with variable z in a syntactic store h:

$$\begin{array}{rcll} \mathrm{lkp}\ z\ (\textbf{let}\ z' = \textbf{ref}\ v\ \textbf{in}\ h) & = & \mathrm{lkp}\ z\ h & \text{if } z \neq z' \\ \mathrm{lkp}\ z\ (\textbf{let}\ z = \textbf{ref}\ v\ \textbf{in}\ D[m]) & = & \mathrm{lkp}'\ z\ v\ m & \end{array}$$

$$\begin{array}{rcll} \mathrm{lkp}'\ z\ v\ \varnothing & = & v & \\ \mathrm{lkp}'\ z\ v\ (m; z := v') & = & v' & \\ \mathrm{lkp}'\ z\ v\ (m; z' := v') & = & \mathrm{lkp}'\ z\ v\ m & z \neq z' \end{array}$$

To define serialization, we will just "slice out" that part of the store that is relevant to a particular value and "wrap" the serialized value in that part of the store. That part is the sub-store that defines all references reachable from that value; serialization will result in a closed expression as demonstrated in Lemma 5.4. Formally:

$$\begin{array}{l} \mathrm{serialize}\ v\ h = \\ \quad \textit{let}\ D[m] = (\mathrm{project}\ h\ (\mathrm{reachable}\ v\ h))\ \textit{in}\ D[m; v] \end{array}$$

Here, $\mathrm{reachable}\ v\ h = \mathcal{V}$ iff $\mathcal{V}$ contains all store locations reachable from v in h. Further, we define project as follows:

$$\mathrm{project}\ D[m]\ \mathcal{V} = \mathrm{project}\ D\ \mathcal{V}\ [\mathrm{project}\ m\ \mathcal{V}]$$

$$\begin{array}{rcll} \mathrm{project}\ [\,]\ \mathcal{V} & = & [\,] & \\ \mathrm{project}\ (\textbf{let}\ z = \textbf{ref}\ v\ \textbf{in}\ D)\ \mathcal{V} & = & \mathrm{project}\ D\ \mathcal{V} & \text{if } z \notin \mathcal{V} \\ \mathrm{project}\ (\textbf{let}\ z = \textbf{ref}\ v\ \textbf{in}\ D)\ \mathcal{V} & = & (\textbf{let}\ x = \textbf{ref}\ v\ \textbf{in} & \\ & & \quad (\mathrm{project}\ D\ \mathcal{V})) & \text{if } z \in \mathcal{V} \end{array}$$

$$\begin{array}{rcll} \mathrm{project}\ \varnothing\ \mathcal{V} & = & \varnothing & \\ \mathrm{project}\ (m; z := v)\ \mathcal{V} & = & \mathrm{project}\ m\ \mathcal{V} & \text{if } z \notin \mathcal{V} \\ \mathrm{project}\ (m; z := v)\ \mathcal{V} & = & \mathrm{project}\ m\ \mathcal{V}; z := v & \text{if } z \in \mathcal{V} \end{array}$$

Now, we can define the operational semantics via a small-step relation $\rightarrow$ on *closed* configurations (e, h), where (e, h) is closed iff $\mathrm{fv}(e) \subseteq \mathrm{dom}(h)$. In our metatheory we will assume that the semantics of ref cell creation will create a globally "fresh" variable reference every time.

The interesting rules are specified in Fig. 6. Note that the semantics of run establishes a distinct process space, so there will be no cross-stage persistence. Also, observe how values are serialized whenever we move between process spaces, in particular when values are lifted, and when results are returned by run.

We lack the space to give the type rules for records and references, but they are standard; we utilize the standard "width and depth" structural subtyping rules for records.

$$\textsc{RRun}\quad \frac{(e,\varnothing) \rightarrow^{\star} (v,h')}{(\textbf{run}\ \langle e\rangle, h) \rightarrow (\text{serialize}\ v\ h', h)}$$

$$\textsc{RRef}\quad \frac{z \notin \text{dom}(D[m])}{(\textbf{ref}\ v, D[m]) \rightarrow ((), D[\textbf{let}\ z = \textbf{ref}\ v\ \textbf{in}\ m])}$$

$$\textsc{RDeref}\quad (!z, h) \rightarrow (\text{lkp}\ z\ h, h)$$

$$\textsc{RAssign}\quad \frac{z \in \text{dom}(D[m])}{(z := v, D[m]) \rightarrow ((), D[m; z := v])}$$

$$\textsc{RLift}\quad (\textbf{lift}\ v, h) \rightarrow (\langle \text{serialize}\ v\ h\rangle, h)$$

$$\textsc{RContext}\quad \frac{(e,h) \rightarrow (e',h')}{(E[e],h) \rightarrow (E[e'],h')}$$

Figure 6. Semantics of $\langle$ML$\rangle$ with Mutation and State

5. Type Validity and Type Safety

In this section we define type validity and sketch formal properties of our metatheory. Aside from illustrating properties of our system, we intend to emphasize how our approach allows standard type properties to be obtained in the metatheory. In particular we can obtain type safety (Theorem 5.2) via a familiar type preservation property (Theorem 5.1). Our approach here is similar to [33] in the context of staged programming, although we would argue simpler due to the lack of open code and escape in the $\langle$ML$\rangle$ core language.

5.1 Type Validity

Type judgements in our system are of the form $\Gamma, \Delta \vdash e : \tau$. *Derivability* of type judgements is defined in terms of type derivation rules in Fig. 5. This type discipline enforces disallowance of cross-stage persistence, in particular the CODE rule ensures that variables occurring within code are treated as code values at the same or greater stage; here we define:

$$\begin{aligned} \langle\cdot\varnothing\cdot\rangle &= \varnothing \\ \langle\cdot\Gamma; x:\tau\cdot\rangle &= \langle\cdot\Gamma\cdot\rangle; x : \langle\cdot\tau\cdot\rangle \end{aligned}$$

Note that application of type abstraction in the $\textsc{App}_\Pi$ rule results in a type substitution. Unlike term substitution, cross-stage persistence of types *should* be allowed, since once evaluated types are purely declarative entities and should be able to migrate across stage levels. This is reflected in the definition of type substitutions defined in Sec. 2. The $\textsc{App}_\Pi$ rule is also defined in terms of a relation between type coercions defined as follows.

DEFINITION 5.1. *We write $\Delta_1 \vdash \Delta_2[\tau/t]$ iff for all $t' \preccurlyeq \tau' \in \Delta_2$ we have $\Delta_1 \vdash t'[\tau/t] \preccurlyeq \tau'[\tau/t]$.*

Type validity is then defined as follows:

DEFINITION 5.2. *A type judgement $\Gamma, \Delta \vdash e : \tau$ is* valid *iff it is derivable and Δ is canonical. We write $e : \tau$ iff $\varnothing, \varnothing \vdash e : \tau$.*

5.2 Metatheory

Our argument for type safety follows a standard path. To begin, a canonical forms Lemma specifies the correspondence of types to their associated classes of values in valid type judgements. Here we consider just the interesting cases.

LEMMA 5.1 (Canonical Forms). *Given valid $\Gamma, \Delta \vdash v : \tau$ all of the following hold:*

1. *if $\tau = \langle\cdot\tau'\cdot\rangle$ for some τ' then $v = \langle e\rangle$ for some e.*
2. *if $\tau = \textbf{type}[\tau']$ for some τ' then $v = \tau''$ for some τ''.*
3. *if $\tau = \Pi t \circ \Delta'.\tau'$ for some t, Δ', τ' then $v = \Lambda t \preccurlyeq \tau''.e$ for some e and τ''.*

Next, a term substitution Lemma will apply to the β reduction case of type preservation. But in type preservation we similarly need to consider the case where type abstraction applications are reduced, so we also obtain an analogous type substitution Lemma. We sketch a case of the term substitution that is central to our system design, where code is substituted into code; the type substitution Lemma follows by a similar induction on type derivations.

LEMMA 5.2 (Type Substitution). *If $\Gamma, \Delta; t \preccurlyeq \tau_0' \vdash e : \tau_0$ and $\Gamma, \Delta \vdash \tau_1 : \textbf{type}[\tau_1']$ with $\Delta \vdash \tau_1' \preccurlyeq \tau_0'$, then $\Gamma, \Delta \vdash e[\tau_1/t] : \tau_0[\tau_1/t]$.*

LEMMA 5.3 (Term Substitution). *If $\Gamma; x : \tau_0', \Delta \vdash e : \tau_0$ and $\Gamma, \Delta \vdash v : \tau_1$ with $\Delta \vdash \tau_1 \preccurlyeq \tau_0'$, then $\Gamma, \Delta \vdash e[v/x] : \tau_0$.*

Proof. This result follows in a mostly standard manner by induction on the derivation of $\Gamma; x : \tau_0', \Delta \vdash e : \tau_0$ and case analysis on the last step in the derivation. The interesting case in our system is where the last step is an instance of CODE. In this case by inversion of CODE we have:

$$e = \langle e'\rangle \qquad \tau_0' = \langle\cdot\tau'\cdot\rangle \qquad \Gamma = \langle\cdot\Gamma'\cdot\rangle \qquad \tau_0 = \langle\cdot\tau\cdot\rangle$$

for some e', τ', τ, and Γ', and we have also a judgement of the form:

$$\frac{\Gamma'; x : \tau', \Delta \vdash e' : \tau}{\langle\cdot\Gamma'\cdot\rangle; x : \langle\cdot\tau'\cdot\rangle, \Delta \vdash \langle e'\rangle : \langle\cdot\tau\cdot\rangle}$$

But $\langle\cdot\Gamma'\cdot\rangle, \Delta \vdash v : \langle\cdot\tau'\cdot\rangle$ by assumption, so by Lemma 5.1 we have that v is a code value of the form $\langle e_1\rangle$ for some e_1. By inversion of the typing rules it is easy to show that $\Gamma', \Delta \vdash e_1 : \tau''$ where $\Delta \vdash \tau'' \preccurlyeq \tau'$, so by the induction hypothesis we have that $\Gamma', \Delta \vdash e'[e_1/x] : \tau$. And since $e[v/x] = \langle e'[e_1/x]\rangle$ in this case by definition of term substitutions, the result follows in this case by an application of CODE. □

Next we extend the notion of type validity to configurations. The definition is quite straightforward thanks to our use of syntactic stores.

DEFINITION 5.3 (Type Valid Configurations). *A configuration typing $(e, D[m]) : \tau \circ \Delta$ is valid iff $\varnothing, \Delta \vdash D[m; e] : \tau$ is.*

An important corollary of this definition is that code values at runtime are *closed*; the importance of this is that closedness ensures that references do not "cross stages", ensuring process separation between stages.

COROLLARY 5.1. *If $(E[\langle e\rangle], D[m])$ has a valid typing then $\langle e\rangle$ is closed.*

Another important property has to do with serialization, and ensuring that our definition of serialization is type-correct in the sense that serialization produces a closed value of the same type as the original, unserialized value:

LEMMA 5.4 (Serialization Typing). *If $(v, h) : \tau \circ \Delta$ is valid, then so is $\varnothing, \Delta \vdash$ serialize $v\ h : \tau$.*

Now, before proving type safety, we observe that the single-step RRUN reduction rule is predicated on a complete reduction in the next-stage process space. Because of this, in type preservation we will need to induct on the length of reduction sequences, where length takes into account the preconditions of RRUN reduction instances.

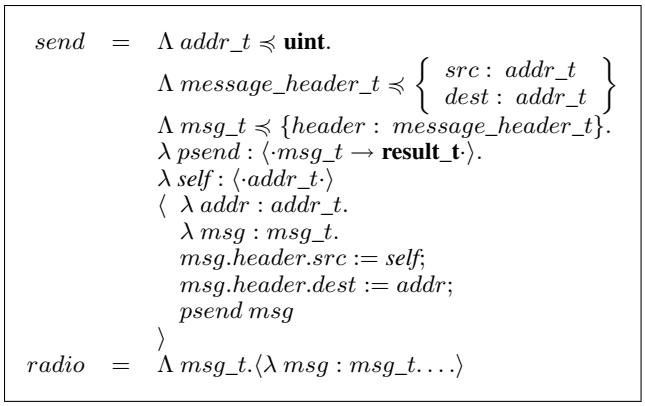

Figure 7. Code Snippet for *send*

DEFINITION 5.4. *The* length of an evaluation relation $(e, h) \rightarrow^{\star} (e', h')$ *is the sum of all single reduction steps in the evaluation, including the reduction steps required in the precedent of a* RUN *reduction.*

Now we can state type preservation, which follows by a double induction on the length of a multi-step reduction sequence and type derivations. Details are omitted here for brevity. The "shared upper bound" relation between initial and final types, rather than equality, is necessary due to subtleties of typing **tlet** expression forms.

THEOREM 5.1 (Type Preservation). *If* $(e_0, h_0) : \tau_0 \circ \Delta$ *is valid and* $(e_0, h_0) \rightarrow^{\star} (e_n, h_n)$*, then* $(e_0, h_0) : \tau_n \circ \Delta$ *is valid where* $\Delta \vdash \tau_0 \preccurlyeq \tau$ $\Delta \vdash \tau_n \preccurlyeq \tau$ *for some* τ_n *and* τ*.*

Type safety follows in a straightforward manner from type preservation, and the additional property that expressions which are irreducible but are not values have no type.

THEOREM 5.2 (Type Safety). *If* $(e_0, h_0) : \tau_0 \circ \Delta$ *is valid then it is not the case that* $(e_0, h_0) \rightarrow^{\star} (e_1, h_1)$ *where* (e_1, h_1) *is irreducible and* e_1 *is not a value.*

6. A Programming Example

In this section, we use sensor network programming as a case study to demonstrate how ⟨ML⟩ can be helpful in real-world programming. Our focus here is to highlight the crucial need for type specialization in staged programming. Existing staged programming systems often focus on how to pre-execute code as much as possible at meta-stage so that code for object-stage execution has the shortest computation time. This philosophy however does not always work well for sensor networks, as shortening computation time alone has a limited effect on the primary issue faced by WSNs – sensor energy consumption. It has been shown in experiments that the energy consumed by transmitting one bit over the radio is equivalent to executing 800 instructions [18]. Thus, given e.g. a *send* function that is going to be executed on the a sensor node, the way to significantly improve system efficiency is not to shorten its code, but to minimize network traffic it would trigger.

The example we are going to present in this section fleshes out the observation above. For example, if we can specialize the type of a node address *addr* so that its representation requires the least possible amount of bits – say **uint4** rather than **uint64**, we are saving 56 bits of each radio send, so the net effect of energy saving is equivalent to saving 56 * 800 = 44,800 instructions for *each send*. Now, if in a particular network deployment we know there is no need for a sensor node to talk to more than 16 neighbors due to some address assignement or neighborhood discovery protocol, we can assign a **uint4** type to *addr*, rather than **uint64**, and save radio power.

To make our example not too contrived, we will use several language constructs beyond the ⟨ML⟩ formal core, including **for** loops and arrays. Adding these features should not be difficult given we already have side effects. For the purpose of this presentation, array-out-of-bound access can happen, and is not considered a type error. The code will also assume that **uint4** is a subtype of **uint8**, **uint8** is a subtype of **uint16**, and so on. All of them are a subtype of **uint**. These base subtyping rules can be easily augmented to the core calculus. Subtyping relations defined as such may lead to memory layout conversions when a subtype value assigned to a supertype value, but for the purpose of type specialization, this is not a problem – the specialized code and the parameter used for specialization does not live in the same memory space. Notation-wise, if the upper bound type of a **tlet** expression is not given it can be assumed to be the same as the tletted type. Such abbreviations also apply to Λ abstractions.

6.1 A Specializable "Send" Snippet

In the standard TinyOS sensor network platform [15], the message type `message_t` has the following format:

```
typedef struct message_t {
  uint8 header[sizeof(message_header_t)];
  uint8 data[TOSH_DATA_LENGTH];
  uint8 footer[sizeof(message_footer_t)];
  uint8 metadata[sizeof(message_metadata_t)];
} message_t;
```

It contains a payload field `data` – the underlying data – together with network control information, including the `header`, the `footer`, and the `metadata`. The `header` in turn has the following type, where the `flag` field contains control information, and `dest` and `src` are destination and source addresses respectively.

```
typedef struct message_header_t {
  uint8 flag;
  uint64 dest;
  uint64 src;
} message_header_t;
```

Any *send* function that is written with type `message_t` being the type for messages will not necessarily be efficient: 64-bit addresses are hardcoded inside this data structure. This situation can be avoided in our language, using our implementation of the *send* function as is illustrated in Fig. 7. Here observe that *send* is a piece of code, defining the logic of message sending at the object stage (*i.e.* on motes). The first argument of the send function is *addr*, denoting the destination address where the packet (the second argument *msg*) is going to be sent.

Note the use of ⟨ML⟩ type specialization here: the message type *msg_t* is abstracted, and eventually will be instantiated at the meta-stage with the most efficient concrete type. It is given a type bound of a record type with at least a *header* field of type *message_header_t*. The latter in turn is also abstracted and can be specialized with any concrete type, as long as it contains a field *dest* whose type is *addr_t*. This last type is closely related to power consumption in sensor networks: when the *send* function is defined, it is abstracted to work on any type that is a subtype of **uint**. Depending on how *send* as a type abstraction is applied, the code eventually being deployed on motes will be sending messages with short addresses (such as **uint4**) or long ones (such as **uint64**).

Note that the *send* function eventually invokes some function on the physical layer to send the actual message out. The particular physical-layer send can be customized, and is passed in as argument *psend*. The signature of that argument suggests that

```
moteCode = Λ addr_t ≼ uint.
           Λ msg_t ≼ {header : {src : addr_t; dest : addr_t}; data : uint8[]}.
           λsendf : ⟨·addr_t → msg_t → result_t·⟩
           λneighbors : ⟨·addr_t[]·⟩.
           λneighbor_num : ⟨·uint16·⟩.
           ⟨ msg_t m;
             m.data = "hello";
             for(uint16 i = 0; i < neighbor_num; i++){
               sendf neighbors[i] m
             }
           ⟩
```

Figure 8. Code for Motes

it is another piece of staged code which contains a function that takes a message of *msg_t* type and returns a TinyOS ACK (of type **result_t**, which is for all practical purposes equivalent to **uint8**). The example of *psend* illustrates the case of how library functions can be used in this context. Note that the *send* definition above is likely to be applied at the meta-stage to produce the staged send code for the motes; the physical-layer function on the hub is probably not the same as the physical-layer function on the mote. By requiring such a function to be applied explicitly, rather than resorting to cross-stage persistence of MetaML to implicitly use the *psend* function defined in a previous stage, our calculus implicitly avoids the issue of accidental library version incompatibility that is common in modern software deployment.

With this function defined, one way to produce a send function with all addresses being 4-bits would be

```
let self = (⟨·uint4·⟩)⟨0xF⟩ in
tlet ht1 = {flag : uint8; src : uint4; dest : uint4} in
tlet mt1 = {header : ht1; data : uint8[DATA_LEN]} in
    send uint4 ht1 mt1 (radio mt1) self
```

The concrete physical-layer sending function is *radio*, which is defined in Fig. 7. To simplify the presentation, we have assumed it can be of any type. This can certainly be refined in a realistic context. The typecast is needed in the first line as we have explained in Sec. 3.2. `DATA_LEN` is an integer constant.

6.2 A Specializable Toy Program on Motes

The *send* code we have described in Fig. 7 is one function that would be deployed to the motes by the hub. We now define a complete toy application the hub will build to run on motes, in Fig. 8. All this example does is to send a "hello" message to its "neighbors", other motes that can be reached in a 1-hop range.

The type of the message that eventually will be sent to neighbors, *msg_t*, is abstracted and can be specialized. It can be of any record type, except that it must contain a *header* field and a *data* field which is a **uint8** array. The header at least contains two fields *src* and *dest*, both of which are of some *addr_t* type that can be specialized. In addition, it also allows the neighbor information of a mote to be specialized, including the entire *neighbors* array, and the number of neighbors *neighbors_num*. What this implies is the definition allows the neighbor information to be "hardcoded". At first glance, supporting hardcoding of neighbor information is unintuitive, especially in a dynamic environment like sensor networks, where neighbor information is previously not known before physical deployment. The rationale here is to promote the potential for memory savings for the case where the number of neighbors is known when *moteCode* is specialized. As a result, rather than allocating the array *neighbors* in (scarce) mote memory, a particular implementation of ⟨ML⟩ may choose to unroll the loop before the code is deployed. Our current foundational calculus does not perform such an unrolling, but this is one possible optimization in the context of embedded systems.

The *moteCode* expression takes anther piece of staged code, *sendf*, as one of its arguments. Thus, on the hub, running the following code piece will deploy a specialized version of *moteCode* on the motes as follows:

```
let self = (⟨·uint4·⟩)⟨0xF⟩ in
tlet ht1 = {flag : uint8; src : uint4; dest : uint4} in
tlet mt1 = {header : ht1; data : uint8[DATA_LEN]} in
let scode = send uint4 ht1 mt1 (radio mt1) self)
let contacts_info = lift [(uint4)0x0] in
    run (moteCode uint4 mt1 scode contact_info ⟨1⟩)
```

The first four lines above are identical to the previous instantiation in Sec. 6.1. At the fifth line, a (trivial) one-neighbor array is created and lifted to the mote stage as *contacts_info* – we will enrich the computation of this array in Sec. 6.3. The last line specializes the code and executes it. Note that we do not support location information in the calculus, so strictly speaking the code above only means "specialize *moteCode* and run it in *some* deployment context (mote)".

6.3 A Metaprogram on the Hub

Fig. 9 gives the bootstrapping code to be executed on the hub. The general idea here is the hub will first execute function

$$getTopology :: () \rightarrow topology\_t$$

to obtain the global connectivity graph of the initially deployed sensor network, and store the result in a hub data structure (the *topo* variable in the example). This graph data structure may be huge, but note that it is kept on the hub only – a resource-rich computer. We omit the definition of this function here. The only implementation detail that is related to the discussion here is the computed graph is undirected, *i.e.*, if edge $\{n1 : 3; n2 : 2\}$ is in the graph, then $\{n1 : 2; n2 : 3\}$ is not redundantly put in the same graph.

The hub then invokes an effectful function

$$coloring :: topology\_t \rightarrow \textbf{uint32}$$

to color the topology graph. The idea here is that sensors only talk to their neighbors, so the unique addresses needed are the number of colors computed by the classic coloring problem. This function mutates the argument *topo*, filling in the *color* field of each of its *nodes* entries. The return value of the function is the number of colors used to color the graph. If that value is *colors*, the colors being used to fill the fields are represented by **uint32** values ranging [0..*colors*-1].

The rest of the function is largely copied from the code fragment deploying the motes, explained in Sec. 6.1 and Sec. 6.2. Note

```
NODE_NUM   =  0xFFFF;
EDGE_NUM   =  0xFFFFFFFF;
DATA_LEN   =  110;
HEAD_NIC   =  0xFFFFFFFFFFFFFFFF;
   uint64      contacts[NODE_NUM];
   node_t   =  {nic : uint64; color : uint64}
   edge_t   =  {n1 : uint16; n2 : uint16};
topology_t  =  {nodes : node_t[NODE_NUM]; edges : edge_t[EDGE_NUM]};
   main     =  tlet ht1 = {flag : uint8; src : uint64; dest : uint64} in
               tlet mt1 = {header : ht1; data : uint8[DATA_LEN]} in
               let topo = getTopology() in
               for(uint16 i = 0; i < NODE_NUM; i++){
                 let self = lift (uint64)topo[i].nic in
                 let contacts_info = lift [(uint64) HEAD_NIC] in
                 let scode = send uint64 ht1 mt1 (radio mt1) self) in
                 run (moteCode uint64 mt1 scode contact_info ⟨1⟩)
               };
               let colors = coloring topo in
               tlet addrt ≼ uint8 = if (colors <= 16) then uint4 else uint8 in
               tlet ht2 = {flag : uint8; src : addrt; dest : addrt} in
               tlet mt2 = {header : ht2; data : uint8[DATA_LEN]} in
               for(uint16 i = 0; i < NODE_NUM; i++){
                 let self = lift (addrt) topo[i].color in
                 let contact_info = lift (addrt[colors])(getNeigbhors topo i) in
                 let contact_num_info = lift colors in
                 let scode = send addrt ht2 mt2 (radio mt2) self) in
                 run (moteCode addrt mt2 scode contact_info contact_num_info)
               }
getNeighbors   =  λgraph : topology_t. λnodei : uint16.
               k := 0;
               for(uint32 i = 0; i < EDGE_NUM; i++){
                  if (graph.edges[i].n1 == nodei) then contacts[k++] := edges[i].n2
                  if (graph.edges[i].n2 == nodei) then contacts[k++] := edges[i].n1
               }
```

Figure 9. Bootstrapping Code for Sensor Head Node

that *send* is specialized twice, as is *moteCode*. The two specializations represent two different send protocols, before coloring and after coloring. At the beginning, before the hub has computed the optimized solution for addressing, it consistently uses **uint64** to set up the network (the first **run** expression). Later, when the entire topology is known, the hub can compute the optimized size for addresses, eventually stored in *addrt*. The neighbor information is also computed at the meta level based on the topology information *topo*. This is achieved by function *getNeighbors*, which is purely a hub execution.

6.4 Discussion on the Example

The oversimplified example presented in this section does not show the full scope of expressiveness of our calculus. It changes no types in the packet other than the size of integers, but it would be easy to also change the packet type by adding additional fields in some particular specializations. The latter can be a very useful feature in sensor network applications, *e.g.* attaching rich `metadata` information only if cryptographic information is needed. The example also does not show how code specializations such as merging messages or dropping redundant radio packets can lead to greater radio efficiencies. In addition, we only focused on the specialization of address types in the example, and keep the length of the data field, `DATA_LEN`, constant. Refinement can be made by allowing the meta-program to adjust the data length, say 110 bytes of data when addresses are of 64-bits and 116 bytes of data when addresses are of 4-bits.

Our example is not robust to changes in a network deployment where, say, neighbors of a node fluctuate between 12 and 120. When that happens, a costly redeployment of code may need to be programmed. However, we hypothesize that for many applications neighborhood sizes will remain within certain bounds for a sufficient duration to make this tradeoff advantageous.

7. Related and Future Work

A variety of previous authors have explored the combination of type specialization and program staging as a means to obtain program efficiency. Several authors have explored the interaction of program staging and type dependence to support compiler construction [2] and interpreters [27]. Also related is work on program generation formalisms for compiler construction that leverage first class types and intensional polymorphism [7]. Tempo [6] is a related system that integrates partial evaluation and type specialization for increasing efficiency of systems applications. Tempo is additionally interesting for us because it is intended for application to C, which is a foundation of nesC. Perhaps the system most closely is related to ours is Monnier and Shao's [24], where type abstraction as a language construct is supported in a staged program calculus albeit following a standard standard System $F_{\leq}$ route (i.e. types are not treated as expressions). The integration of staging abstractions and side effects is another dimension of our

work that has been considered by previous authors. Kameyama et al. have studied staging in the presence of side effects as a way to optimize algorithms that exploit mutation [17]. Moggi and Fagorzi have established a monadic foundation for integrating staging with arbitrary side effects in a highly general and mathematically rigorous fashion [22]. But in addition to various technical differences, these systems are contrasted with ours in that none have considered embedded systems as an application space.

Type-safe code specialization has been the focus of MetaML [35, 23] and its more recent and robust implementation, MetaOCaml [9]. MetaML has also been promoted as an effective foundation for embedded systems programming [32] and enjoys type safety results of the sort presented here [33]. On a foundational level, the problem of how to represent code of one stage in another stage has been studied in various formalisms, such as modal logic [8], higher-order abstract syntax [37], and first-order abstract syntax with deBruijn indices [5]. One particular technical issue that has triggered many recent developments in this area is known as the "open code" problem. As we described in Sec. 2.2, our calculus does not support arbitrary escape expressions, and so the open code problem does not appear, simplifying our formal development. The added expressiveness of MetaML here comes at the price of having to deal with significant additional type system complexities [26, 5, 3, 34]. We have thus far not found this added expressiveness useful for embedded systems programming.

Parametric customization of type annotations is not new; widely used examples include C++ templates and Java generics. The formal foundations for Java generics are the parametric type systems System F and $F_{\leq}$ [4], and our parameterized type syntax is similar. All of these systems however do not treat types as first-class values like we do, and this significantly limits their usefulness in the application domain we focus on here. Runtime type information has been successfully used for the special case of a decidable type system for specializing types of polymorphic functions [14], and while we are performing a different kind of type specialization this work shares with our work the desire to push the frontiers of decidable type systems using runtime type information. Many staging frameworks allow types to be customized, but the output of the customization needs to be re-type-checked from scratch and so does not have the level of type safety that we have; two examples of this are the C++ template expansion and Flask, the latter which we now cover.

The potential of applying metaprogramming to sensor networks was recently explored by Flask [19]. The main motivation of designing Flask is to allow FRP-based [36] stream combinators to be pre-computed before sensor networks are deployed. The key construct of Flask is *quasi-quoting*, which in essence is MetaML's stage operator <*e*> combined with an escape operator ~*e*. Since pre-computing stream combinators is the main goal of Flask, the focus of our language – computing precise *type annotations* inside the object-stage code at meta stage – is not a topic they focus on. In particular, cross-stage static type-checking of Flask is relatively weak; it is possible to generate ill-typed Flask object code.

The standard method TinyOS sensor programmers use to customize messages is a tool called `mig` [21]. Before the program is deployed, several experiments out of the scope of the programming system are conducted, so that calibration parameters can be obtained, and are used as the input parameters of `mig` to customize the code. The drawback of such an approach is the entire calibration process is manually conducted. Sensor programmers in our language can embed the entire calibration and code customization process as part of the main hub program.

In the future, we plan to explore the use of conditional types [1, 28] or conditionally tagged type unions [30] to avoid some of our need for typecasts and thus to gain more static type safety.

Even though the design of $\langle$ML$\rangle$ was greatly influenced by sensor network programming needs, the presentation here is a general-purpose staged calculus that can be independently used for meta programming in cases where runtime type specialization and deployment are important. For this reason, the calculus leaves out language abstractions that are needed for sensor network programming specifically. For instance, $\langle$ML$\rangle$ does not contain distributed communication primitives, locality, concurrency, or mechanisms to marshall data to bit strings. These features will be important when we build a domain-specific language upon the foundation of $\langle$ML$\rangle$.

References

[1] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[2] Edwin Brady and Kevin Hammond. A verified staged interpreter is a verified compiler. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 111–120, New York, NY, USA, 2006. ACM.

[3] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Closed types as a simple approach to safe imperative multi-stage programming. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 25–36, London, UK, 2000. Springer-Verlag.

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

[5] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. In *ICFP'03*, 2003.

[6] 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.

[7] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. *J. Funct. Program.*, 12(6):567–600, 2002.

[8] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001.

[9] Waid Taha *et. al.* MetaOCaml: A compiled, type-safe multi-stage programming language. `http://www.metaocaml.org/`.

[10] Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 74–85, New York, NY, USA, 2001. ACM.

[11] 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 '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM.

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

[13] Jeremy Gibbons. *Datatype-Generic Programming*, pages 71, 1. 2007.

[14] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *In Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141. ACM Press, 1995.

[15] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[16] Furio Honsell, Ian A. Mason, Scott Smith, and Carolyn Talcott. A variable typed logic of effects. *Information and Computation*, 119:55–

90, 1993.

[17] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage: staging with delimited control. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 111–120, New York, NY, USA, 2009. ACM.

[18] 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.

[19] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: Staged functional programming for sensor networks. In *13th ACM SIGPLAN International Conference on Functional Programming (ICFP 2008)*, September 2008.

[20] James McKinna. Why dependent types matter. *SIGPLAN Not.*, 41(1):1–1, 2006.

[21] `mig`: message interface generator for nesC, available online at `http://www.tinyos.net/tinyos-1.x/doc/nesc/mig.html`.

[22] Eugenio Moggi and Sonia Fagorzi. A monadic multi-stage meta-language. In Andrew D. Gordon, editor, *FoSSaCS*, volume 2620 of *Lecture Notes in Computer Science*, pages 358–374. Springer, 2003.

[23] Eugenio Moggi, Walid Taha, Zine El abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *In European Symposium on Programming (ESOP*, pages 193–207. Springer-Verlag, 1999.

[24] Stefan Monnier and Zhong Shao. Inlining as staged computation. *J. Funct. Program.*, 13(3):647–676, 2003.

[25] Tom Murphy VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *LICS '04: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, pages 286–295, Washington, DC, USA, 2004. IEEE Computer Society.

[26] Aleksandar Nanevski. Meta-programming with names and necessity. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 206–217, New York, NY, USA, 2002. ACM.

[27] Emir Pasalic, Walid Taha, Tim Sheard, and Tim S. Tagless staged interpreters for typed languages. In *In the International Conference on Functional Programming (ICFP02)*, pages 218–229. ACM, 2002.

[28] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.

[29] Curt Schurgers, Gautam Kulkarni, and Mani B. Srivastava. Distributed on-demand address assignment in wireless sensor networks. *IEEE Trans. Parallel Distrib. Syst.*, 13(10):1056–1065, 2002.

[30] Jonathan Shapiro and Swaroop Sridhar. The BitC programming language. `http://www.bitc-lang.org/`.

[31] Rui Shi, Chiyan Chen, and Hongwei Xi. Distributed meta-programming. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 243–248, 2006.

[32] Walid Taha. Resource-aware programming. In Zhaohui Wu, Chun Chen, Minyi Guo, and Jiajun Bu, editors, *ICESS*, volume 3605 of *Lecture Notes in Computer Science*, pages 38–43. Springer, 2004.

[33] Walid Taha, Zine el-abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type safety (extended abstract). In *In 25th International Colloquium on Automata, Languages, and Programming*, pages 918–929. Springer-Verlag, 1998.

[34] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *POPL'03*, 2003.

[35] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217, 1997.

[36] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 242–252, New York, NY, USA, 2000. ACM.

[37] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 224–235, New York, NY, USA, 2003. ACM.