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 ⟨ML⟩ as a new staged calculus designed with novel features for staged programming in modern computing platforms such as embedded systems. A distinguishing feature of ⟨ML⟩ 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. ⟨ML⟩ 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.

1. Introduction

Modern software often evolves through a lifecycle of deployment stages [12], each of which executes within a distinct deployment environment with distinct requirements of speed, memory usage, and power consumption. The focus of this paper is using staged programming techniques to better support the modern evolving deployment context.

There has been a long history of explicit support for staging in programming languages [24, 6, 5, 22, 2, 19, 18, 4, 23]. These language designs all admit program code itself as a data type, and support composition, specialization, and running of code-as-data. Traditional staged languages are designed for partial evaluation and optimized code generation. The new language presented here, ⟨ML⟩, borrows ideas from many existing systems such as MetaML [24], but is designed with a new and different set of practical goals in mind, as we explain now.

Staging as Deployment Steps We believe the concept of staging is beneficial in modern software applications where deployment is an important part of the software lifecycle. When staging is explicitly supported by a programming language, each stage can represent a distinct deployment step, and the code for preparing code for the next stage can be viewed as writing a "deployment script". It would be desirable to directly use existing staged languages such as MetaML to model deployment processes. However, MetaML supports *cross-stage persistence*, a feature that allows programs to use variables bound to values at one stage to be used in "later" stages as if it were a variable defined at a later stage, and thus allows values to migrate freely from stage to stage. One example that justifies the use of cross-stage persistence is the use of library functions defined at the bootstrapping stage to be freely used at any later stage. While cross-stage persistence was originally designed for convenience, it conflicts with our intended view of each stage as a deployment step. One type of software that exemplifies the modern traits of staging are those running on wireless sensor networks [10, 8, 13, 14]. These networks 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. 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. Cross-stage persistence is unfeasible in this setting because problems of identity and value equality across different network node spaces are too great.

⟨ML⟩ provides a framework which assumes the different stages run in different processes, unlike MetaML which is a single process. On top of this, our ⟨ML⟩ supports a principled approach to cross-stage value migration in spite of the separation of processes; for this a simple serialization/marshalling protocol is designed.

Type Specialization for Efficiency A second benefit of applying staging to modern software is more in line with what traditional staging languages are good at: producing more efficient code for the next stage. But, what sort of efficiency really matters? The vast majority of existing meta-programming systems are motivated by goals of how to inline/pre-compute *code* as much as possible at the meta-stage, so that the code for the object stage will require fewer computation steps.

Some modern deployment contexts operate over a different metric of efficiency. Again taking sensor networks as an example, resources on motes are extremely constrained, but the constraint does not primarily come from the CPU, but from memory and power consumption. Shortening computation time does not reduce memory usage significantly, and may in some cases increase it. Shortening computation time also has a limited effect on sensor energy consumption, as it has been shown in experiments that the energy consumed by transmitting one bit over the radio is equivalent to executing 800 instructions [13]. Thus, given a $send$ function that is going to be executed on the a sensor node

$$send = \lambda addr : \textbf{uint64}.e$$

the way to significantly improve system efficiency (i.e. memory usage and energy consumption) is not to specialize the code e to include as fewer computation steps as possible, but instead to attempt to specialize the *type* of $addr$ so that it does not always need to be represented using 64 bits. If we can specialize the type of $addr$ to **uint8**, we are saving 7 bytes of, say, 4K, of sensor node memory, and more importantly 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.

A centerpiece of ⟨ML⟩ is it allows the types used in the staged code to be specialized. Note that in this situation, the type annotations we ultimately want on the mote code are based on the *runtime* behavior of hub code: the hub will dynamically calculate how much memory is needed for each $addr$ in a mote it is about to deploy. To solve this requires types to be treated as fully first-class objects that can be passed around freely. So, a primary contribution of this paper is a staged type theory which allows types to be treated as first-class entities, whilst maintaining decidability of typechecking. It is challenging to make a type system decidable in the presence of first-class types; we preserve decidability by greatly limiting how

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

Figure 1. A ⟨ML⟩ Code Snippet for a `member` Function

the runtime types can be used to improve typechecking. ⟨ML⟩ is a provably sound strongly typed staged programming language with type abstraction, bounded subtyping, and which supports types as first-class values. Our soundness result is strong in the sense that the code fragments assembled and specialized by the hub will not produce a runtime error either when being compiled to ship to the mote or when running on the mote.

2. Motivations and Examples

In this section, we informally describe the design principles and key language features of ⟨ML⟩. 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.

2.1 The Relation with MetaML

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. 1. 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 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. 1 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:

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

The Principle of Cross-Stage Process Separation Philosophically, there is no justification why code on different stages has to be working in the same memory space. In fact, we feel a main attraction of meta programming in practice is to treat each stage as a *distinct deployment context*. One of the main goals of ⟨ML⟩ is to model a clean separation of process spaces for different program stages.

Of course, it is necessary to allow a *principled* migration of values across stages, which our language does allow. A language expression for this purpose is **lift** e, which "lifts" the evaluation result of e to the next stage, and hence can be processed by the next stage. In Fig. 1, 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. 1, MetaML programmers would write the equivalent

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

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 often be re-written as $\lambda x.\langle C[x] \rangle)(e)$ where C is an evaluation 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, the "open code" problem [19, 4, 23]. 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; a more detailed discussion of this topic can be seen in Sec. 7.

Note that a design with no escape operators does not undermine support of open code per se. As the example shows, x, h, and tl are all free variables in code $\langle x = h\ ||\ tl \rangle$. Outside of the $\langle \cdot \rangle$ they are types of code, and inside they indicate code inserted into code.

2.2 Type Specialization

Type Abstraction and Application for Staged Code The primary reason why ⟨ML⟩ has type specialization is to support specialization of types in staged code. The previous code in ⟨ML⟩ can be re-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 ⟨ML⟩ type arguments can be dynamically constructed, not just statically declared. Type application can then be performed to produce staged code with a concrete type, such as

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

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

One might wonder why the code above does not need to be written as:

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

The answer is that unlike term values, types (such as **uint8**) are merely declarative information always interpreted uniformly at different deployment contexts. Thus it is appropriate to support "cross-stage persistence" of types, as evinced in the above example.

Type Bounds and Subtyping The benefit of static type checking over staged code has been widely discussed in recent efforts in meta programming [16, 2, 26, 4, 23]. Type checking the code above however would be challenging. As a universal type, $addr\_t$ can be instantiated with any concrete type. To typecheck the code above modularly, the typechecker would have to make conservative consumptions, so that practically any use of $addr$ in e would need to be assigned with a top type, and the vast majority of programs would not type check.

To tackle this problem, ⟨ML⟩ allows programmers to give a bound on the abstracted type. For instance, the $send$ code defined previously can be refined as:

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

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 $F_<$, 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 $F_<$ where types and terms do not mix, and all type instantiation occurs statically, types are first-class citizens in ⟨ML⟩, 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 [7], except that most people – including the macros users themselves – complain they are not expressive enough. Treating types as values in ⟨ML⟩ provides programmers with a flexible way to define constructs like above (so much so that arbitrary programs are allowed to define how the T above can be typedef-ed), at the same time *preserving static type safety*. As a result of this design choice, the static type system of our language is very different from System $F_<$ and its descendants such as Java generics; our types are a more general form of dependent type and we use notation $\Pi t.\tau$ for them to reflect this.

As an example, the following code is an ⟨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. **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 $rtt : \langle tcond \rightarrow result\_t \rangle$, under type constraint $tcond \preccurlyeq$ **if** e **then uint16 else uint32**. Here notation $\langle \tau \rangle$ represents the type for a piece of staged code which has a τ type at its stage.

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

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

Many dependent type systems are known to be undecidable since some types will be inextricably tied to runtime behavior. Our particular formalism however is decidable due to the limited runtime presence of types.

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 upper bounds and the type parametricity provide ways to get non-initial values in and out of the type so it is the initial construction that is the problem). To define a member of a runtime-decided type in the code, the runtime condition, for example the $e0$ condition in the above example, is pivotal. For the above 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, 20] which are suited for this purpose, but for this simple presentation we opt for a typecast which is more general but incurs a runtime check. For this particular example we could write say

$$rtt((tcond)5)$$

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

State and Serialization Our ultimate goal is a sensor language but the scope of such a language project is large: it needs to include remote code loading, concurrent execution of the global sensors-hub system, and messaging protocols. Those features are largely orthogonal to the type and staging system design however and so we leave them out of this presentation. We do extend our pure language to include state and a method for serializing state. This is important for sensor programming: the hub can build specialized data structures (such as routing tables) and **lift** them to hub code so they become hardwired in the ROM of the mote. Since RAM is much more scarce than ROM on a mote this is a very useful operation.

3. Core Language and Type System

As discussed in Sect. 2, we aim to establish foundations for staged computation with embedded systems program specialization as an envisioned application domain. To this end we define the ⟨ML⟩ language syntax, operational semantics, and type system in this Section, focusing on the side-effect-free fragment of the language at first for ease of discussion; in Sect. 4 we will add mutation and state and records to obtain a more expressive language model.

3.1 ⟨ML⟩ Syntax and Semantics

$x \in \mathcal{V},\ t \in \mathcal{T}$

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

Figure 2. ⟨ML⟩ Term and Type Syntax

The ⟨ML⟩ language syntax is defined in Fig. 2, including *values* v, *expressions* e, *evaluation contexts* E, *types* τ, *type coercions* Δ, and *type environments* Γ. In addition to core functional calculus terms and arbitrary (user-defined) program constants $\mathbf{c}$, we include a form for type abstractions $\Lambda t \preccurlyeq \tau.e$, a "type let" **tlet**, and also include types as program terms; these forms are discussed more in Sect. 3.2. A casting form $(\tau)e$ is included, the motivations for which were discussed in Sect. 2. We also include three 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.

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

$$
\begin{array}{rcll}
t[\tau/t] & = & \tau & \\
t'[\tau/t] & = & t' & \text{if } t \neq t' \\
\gamma[\tau/t] & = & \gamma & \\
\mathbf{type}[\tau'][\tau/t] & = & \mathbf{type}[\tau'[\tau/t]] & \\
\langle \tau' \rangle[\tau/t] & = & \langle \tau'[\tau/t] \rangle & \\
(\Pi t' \circ \Delta.\tau')[\tau/t] & = & \Pi t' \circ \Delta[\tau/t].\tau'[\tau/t] & \text{if } t \neq t' \\
(\Pi t \circ \Delta.\tau')[\tau/t] & = & \Pi t \circ \Delta[\tau/t].\tau'[\tau/t] &
\end{array}
$$

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

Figure 3. Type and Term Subsitutions in $\langle$ML$\rangle$

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

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

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

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

Figure 4. $\langle$ML$\rangle$ Core Operational Semantics

As discussed in the previous section, we omit a "quote" or "escape" form, typically denoted ~e, because we use here a much simplified form that is adequate for our envisioned application, and indeed adequate to express what escape is typically used for, that is dynamic construction and splicing of code. 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, in particular we should have that $\langle e \rangle[\langle e' \rangle/x] = \langle e[e'/x] \rangle$, and $\langle e \rangle[e'/x]$ is undefined if e' is not code. This is specified in our formal definition of term substitution in Fig. 3.

The operational semantics of $\langle$ML$\rangle$ are then defined in Fig. 4 in terms of substitutions, as a small-step reduction relation $\to$. This relation is defined in a mutually recursive fashion with its reflexive, transitive closure denoted $\to^\star$. 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 Sect. 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.

3.2 Type Specialization and Subtying Theory

As discussed in Sect. 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. Examples in Sect. 2 and Sect. 6 illustrate the usefulness of this. In order to obtain a clear separation of types and expressions and promote well-typed type construction, we introduce the **tlet** form.

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, 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.

3.2.1 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 $\tau \to \tau$ 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. In a type $\Pi t \circ \Delta.\tau$ the coercion Δ expresses the type variable bound on the type abstraction, and also expresses bounds related to **tlet**-declared type variables in the body of the abstraction (that may escape their static scope). Clearly, subtyping plays a central role in our type system, so associated formalisms must be specified.

$$\Delta \vdash \tau \preccurlyeq \tau \qquad \frac{t \preccurlyeq \tau \in \Delta}{\Gamma \vdash t \preccurlyeq \tau} \qquad \frac{\Delta \vdash \tau_1 \preccurlyeq \tau_2}{\Delta \vdash \langle\tau_1\rangle \preccurlyeq \langle\tau_2\rangle}$$

$$\frac{\Delta \vdash \tau_1 \preccurlyeq \tau_2 \qquad \Delta \vdash \tau_2 \preccurlyeq \tau_3}{\Delta \vdash \tau_1 \preccurlyeq \tau_3}$$

$$\frac{\Delta \vdash \tau_1' \preccurlyeq \tau_1 \qquad \Delta \vdash \tau_2 \preccurlyeq \tau_2'}{\Delta \vdash \tau_1 \to \tau_2 \preccurlyeq \tau_1' \to \tau_2'} \qquad \frac{\Delta \vdash \tau \preccurlyeq \tau'}{\Delta \vdash \textbf{type}[\tau] \preccurlyeq \textbf{type}[\tau']}$$

$$\frac{\Delta; t \preccurlyeq \tau_0 \vdash \tau \preccurlyeq \tau'}{\Delta \vdash (\Pi t \preccurlyeq \tau_0.\tau) \preccurlyeq (\Pi t \preccurlyeq \tau_0.\tau')}$$

Figure 5. Subtyping Rules

Intuitively, any coercion Δ is a system of upper bounds on type variables. 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. 5. In our type system, we will also require that only *one* upper bound per type variable may be defined, meaning that **tlet**-declared variables within the same static scope must be distinct, and also it disallows an overspecialization of type variables, as discussed more below.

DEFINITION 3.1. *A coercion Δ is* canonical *iff $t \preccurlyeq \tau_1 \in \Delta$ and $t \preccurlyeq \tau_2 \in \Delta$ implies $\tau_1 = \tau_2$.*

3.2.2 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. 6. 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:

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

Note that application of type abstraction in the 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 Fig. 3. The APP$_\Pi$ rule is also defined in terms of a relation between type coercions defined as follows.

DEFINITION 3.2. *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 3.3. *A type judgement $\Gamma, \Delta \vdash e : \tau$ is* valid *iff it is derivable and Δ is canonical. We write $e : \tau$ iff $\varnothing, \varnothing \vdash e : \tau$.*

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. 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 [11].

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

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

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

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

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

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

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:

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

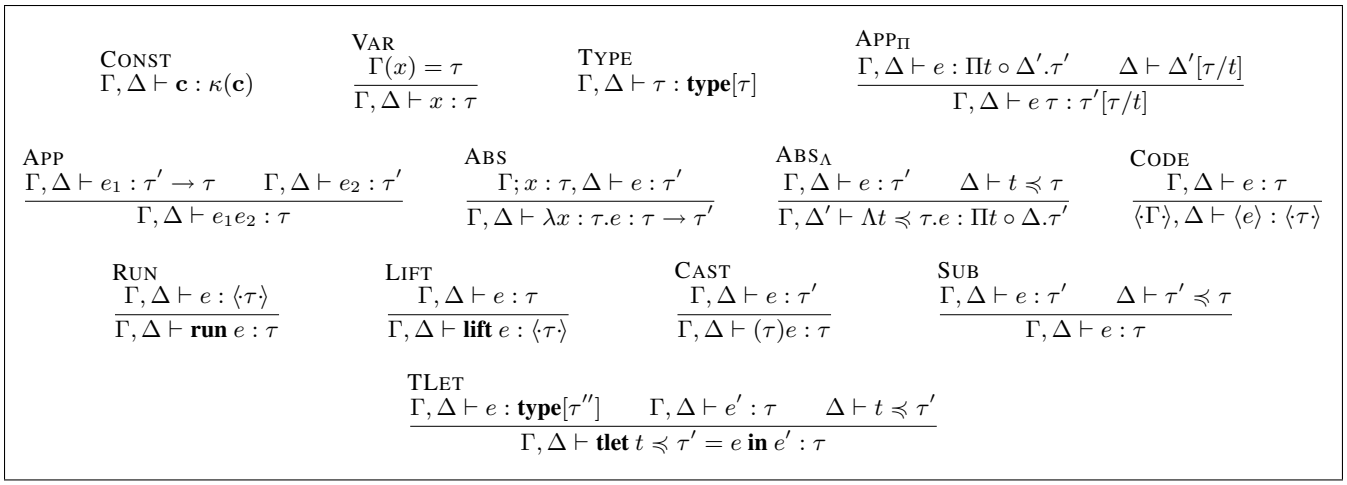

Figure 6. Type Judgement Rules

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

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

$$\begin{array}{rl} \text{project}\ [\,]\ \mathcal{V} = [\,] & \\ \text{project}\ (\mathbf{let}\ z = \mathbf{ref}\ v\ \mathbf{in}\ D)\ \mathcal{V} = \text{project}\ D\ \mathcal{V} & \text{if } z \notin \mathcal{V} \\ \text{project}\ (\mathbf{let}\ z = \mathbf{ref}\ v\ \mathbf{in}\ D)\ \mathcal{V} = (\mathbf{let}\ x = \mathbf{ref}\ v\ \mathbf{in} & \\ \qquad (\text{project}\ D\ \mathcal{V})) & \text{if } z \in \mathcal{V} \end{array}$$

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

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

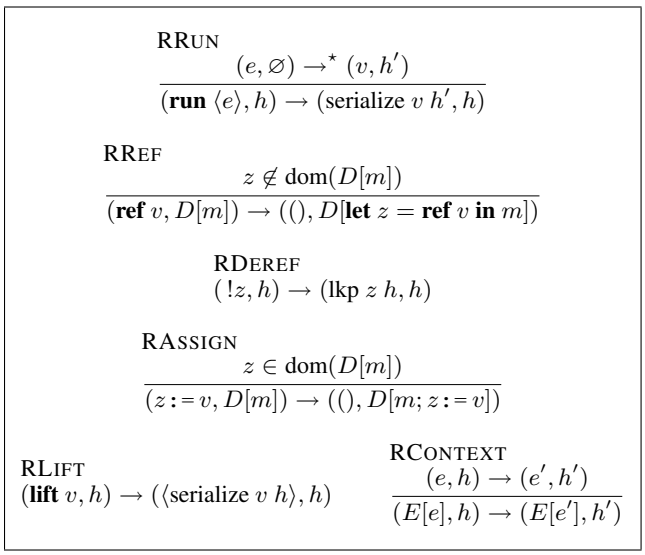

Figure 7. Semantics of $\langle$ML$\rangle$ with Mutation and State

The interesting rules are specified in Fig. 7. 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.

5. Properties

In this section we 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 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* $\tau = \langle\tau'\rangle$ *for some* τ' *then* $v = \langle e\rangle$ *for some* e.
2. *if* $\tau = \mathbf{type}[\tau']$ *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* $\Gamma, \Delta \vdash \tau_1 : \mathbf{type}[\tau_1']$ *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:

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

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

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

But $\langle \Gamma' \rangle, \Delta \vdash v : \langle \tau' \rangle$ 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 5.1 (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.

DEFINITION 5.2. *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 $(e_0, h_0) \rightarrow^\star (e_0, h_0)$ where (e_0, h_0) is irreducible and e_0 is not a value.*

$$\begin{array}{rcl}
send & = & \Lambda\, addr_t \preccurlyeq \textbf{uint}. \\
 & & \Lambda\, message_header_t \preccurlyeq \left\{ \begin{array}{l} src : addr_t \\ dest : addr_t \end{array} \right\} \\
 & & \Lambda\, msg_t \preccurlyeq \{header : message_header_t\}. \\
 & & \lambda\, psend :< .msg_t \rightarrow \textbf{result_t}. > . \\
 & & \lambda\, self :< .addr_t. > \\
 & & < \quad \lambda\, addr : addr_t. \\
 & & \quad \lambda\, msg : msg_t. \\
 & & \quad msg.header.src := self; \\
 & & \quad msg.header.dest := addr; \\
 & & \quad psend\ msg \\
 & & > \\
radio & = & \Lambda\, msg_t \preccurlyeq \textbf{Type}. < \lambda\, msg : msg_t. \cdots >
\end{array}$$

Figure 8. Code Snippet for *send*

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. The specific issue we address is to allow mote addresses to be of variant length depending on the deployment context, rather than a fixed type such as the **uint64** used in the *send* example of Sec. 1. If in a particular network deployment we know there is no need for a sensor node to talk to more than 16 neighbors, we can assign a 4-bit integer as the type for network packet addresses, rather than **uint64**, and save radio power. This example does not show the full scope of runtime type specialization since 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 information in some particular specializations.

6.1 A Specializable "Send" Snippet

In the standard TinyOS sensor network platform [10], 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;
```

It is evident that any *send* function that is written with type `message_t` being the type for messages will not 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. 8. 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 is going to be sent.

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

Figure 9. Code for Motes

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 of **uint4**) or long ones (such as of **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 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 examples 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, it is obvious that 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. 8. To simplify the presentation, we have assumed it can be of any type (with top bound **Type**). This can certainly be refined in a realistic context. Note that to make this program type check, we have assumed $inthalf$ is a subtype of **uint**. This base subtyping rule can be easily augmented to the core calculus. The typecast is needed in the first line as we have explained in Sec. 2.2.

6.2 A Specializable Toy Program on Motes

The *send* code we have described in Fig. 8 is one function that would be deployed to the motes by the hub. We now define a complete toy application to run on motes, in Fig. 9. All this example does is to send a "hello" message to its "neighbors", other motes that can be reached in a 1-hop range. To make this example not too contrived, we have used 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 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 above. At the fifth line, a one-neighbor array is created and lifted to the mote stage as *contacts_info*. 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 Meta Program on the Hub

Fig. 10 gives the bootstrapping code to be executed on the hub. (Notation: if the upper bound type of a **tlet** expression is not given it can be assumed to be the same as the tletted type.) The code has

also assumed that **uint4** is a subtype of **uint8**, **uint8** is a subtype of **uint16**, and so on. In general, 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.

The general idea here is the hub will first execute function

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

to get the entire global connectivity information of the entire sensor network, and store the result in a hub data structure (the *topo* variable in the example). This 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 $\{n1 : 3; n2 : 2\}$ is an edge in the graph, then $\{n1 : 2; n2 : 3\}$ is not redundantly put in the same graph.

The hub then invokes function

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

to compute the minimal numbers to color this topology graph. The idea here is that sensors only talks to their neighbors, so the unique addresses that are needed are basically the number of colors in the classic coloring problem. Function

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

is an immutable function that takes a *topo* data structure and returns a new one with all colors filled, in the *color* field of each entry of that data structure. When the second argument of the *coloring* function is *colors*, the colors being used to fill the fields are represented by integers [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 that *send* is specialized twice, as is *moteCode*. At a high level, the two send deployments reflect two different send protocols, before specialization and after specialization. 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. As we explained above, the current language does not support locations, concurrency or messaging, so the code above is not complete; still it fully shows the types and how they can be specialized and code deployed.

7. Related and Future Work

The centerpiece of this paper – type-safe runtime type specialization for staged programming – does not appear to overlap significantly with existing work. The closest work we know of is [17]. In that work, type abstraction as a language construct is supported in a staged program calculus following a standard standard System $F_<$ route; types are not treated as expressions.

Type-safe code specialization has been the focus of MetaML [24, 16] and its more recent and robust implementation, MetaOCaml [6]. 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 [5], higher-order abstract syntax [26], and first-order abstract syntax with deBruijn indices [4]. 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. 2.1, Our calculus does not support arbitrary escape expressions, and so the open code problem does not appear, simplifying our formal development. The advantage of supporting open code is it allows programmers to escape code under a variable binding which manipulates that variable, for example as in the MetaML example <function x -> ~ (member <x> [1..100])>; we miss this opportunity in ⟨ML⟩. The added expressiveness of MetaML here comes at the price of having to deal with significant additional type system complexities [19, 4, 2, 23]. We have thus far not found this added expressiveness useful for our sensor programming examples and so we have left it out.

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 $F_<$ [3], 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 [9], 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 meta-programming to sensor networks was recently explored by Flask [14]. The main motivation of designing Flask is to allow FRP-based [25] 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 address. 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` [15]. 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, 20] or conditionally tagged type unions [21] to avoid some of our need for typecasts and thus to gain more static type safety.

Even though the design of ⟨ML⟩ 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 (re-)deployment are important. For this reason, the calculus leaves out language abstractions that are needed for sensor network programming specifically. For instance, ⟨ML⟩ 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 ⟨ML⟩.

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

```
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 = (uint64)lift 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 = minColors topo in
               tlet addrt ≼ uint8 = if (colors <= 16) then uint4 else uint8 in
               let psize = if (colors <= 16)  then DATA_LEN
                                              else DATA_LEN + 6 in
               let topo = coloring topo colors in
               tlet ht2 = {flag : uint8; src : addrt; dest : addrt} in
               tlet mt2 = {header : ht2; data : uint8[psize]} in
               for(uint16 i = 0; i < NODE_NUM; i++){
                 let lift self = (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 10. Bootstrapping Code for Sensor Head Node

163–173, 1994.

[2] 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.

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Furio Honsell, Ian A. Mason, Scott Smith, and Carolyn Talcott. A variable typed logic of effects. *Information and Computation*, 119:55–90, 1993.

[12] Yu David Liu and Scott F. Smith. A formal framework for component deployment. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 325–344, New York, NY, USA, 2006. ACM.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

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

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

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.