

Type-Specialized Staged Programming with Process Separation

YU DAVID LIU

SUNY Binghamton

CHRISTIAN SKALKA

The University of Vermont

SCOTT SMITH

The Johns Hopkins University

Abstract

Staging is a powerful language construct that allows a program at one stage of evaluation to manipulate and specialize a program at a later one. We propose $\langle\text{ML}\rangle$ as a new staged calculus designed with novel features for staged programming in modern computing platforms with resource constraints such as embedded systems. A distinguishing feature of $\langle\text{ML}\rangle$ is its support for dynamic type specialization via type abstraction, dynamic type construction, and a decidable form of dependent typing. This particular combination of flexible generic types and staging brings genericity in programs to a new level that is extremely useful in our intended application domains. To further support applications in these domains, $\langle\text{ML}\rangle$ incorporates a model of process separation, whereby different stages of computation are executed in different process spaces. Despite these novelties, our language 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

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, in particular software for wireless sensor networks (WSNs). The particular notion of genericity we explore in this paper, using the classification of generic programming found in Section 2 of (Gibbons, 2007), is a novel combination of *genericity by stage* and dynamic *genericity by type* and *genericity by structure*, producing a principled technique to organize and optimize program code deployed in embedded systems.

1.1 Staging Deployment Steps with Process Separation

There is a long history of explicit support for staging in programming languages (Taha & Sheard, 1997; Taha & *et. al.*, n.d.; Davies & Pfenning, 2001; Shi *et al.*,

2006; Calcagno *et al.*, 2000; Nanevski, 2002; Murphy VII *et al.*, 2004; Chen & Xi, 2003; Taha & Nielsen, 2003). 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. Program staging allows a principled method for dynamic program generation and execution, via program syntax to *run* the constructed code-as-data. 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 multi-tier architectures. WSNs in particular are composed of a potentially 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 (Hill *et al.*, 2000; Gay *et al.*, 2003; Madden *et al.*, 2002; Mainland *et al.*, 2008). Hence, along with previous authors (Taha, 2004) 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, each stage must be envisioned as executing within a distinct process space. Our $\langle \text{ML} \rangle$ model is closely related to MetaML (Taha & Sheard, 1997), a well known extension of ML with support for staging, but 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 (Gay *et al.*, 2003) make heavy use of it.

1.2 Type and Structure Genericity

Two other sorts of genericity we explore in this paper are genericity by type and genericity by structure (Gibbons, 2007). 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 dynamic construction of types by treating types as first class values (the latter also leads to genericity by function, since a function may take a dynamically-constructed type as argument). We additionally allow genericity by structure via record subtyping, and putting this

$ \begin{aligned} x &\in V \subset \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 ee \mid \mathbf{tlet} \ t \preccurlyeq \tau = e \ \mathbf{in} \ e \mid \mathbf{run} \ e \mid \mathbf{lift} \ e \\ E &::= [] \mid Ee \mid vE \mid \mathbf{tlet} \ t \preccurlyeq \tau = E \ \mathbf{in} \ e \mid (E)e \mid (v)E \\ \tau &::= t \mid \gamma \mid \mathbf{type}[\tau] \mid \langle \cdot \tau \cdot \rangle \mid \mathbf{II}t \preccurlyeq \tau. \tau \mid \mathbf{\exists}t \preccurlyeq \tau. \tau \mid \tau \rightarrow \tau \\ \Delta &::= \emptyset \mid \Delta; t \preccurlyeq \tau \\ \Gamma &::= \emptyset \mid \Gamma; x : \tau \end{aligned} $

Fig. 1. $\langle \text{ML} \rangle$ Term and Type Syntax

together with type genericity allows a static declaration of a record type with “at least” some fields, where at runtime the dynamically constructed record type may in fact have more fields than was statically declared.

Our focus on these features was determined by studying common design patterns in WSN programming applications. For example, upon deployment, WSNs can refine node address sizes to minimize their memory footprint. This reduces message sizes, leading to significant bandwidth reduction during communications (Schurgers *et al.*, 2002). Type and structure genericity can work in conjunction with program staging in $\langle \text{ML} \rangle$ to maximize program efficiency in this scenario.

The work presented here is foundational. While our ultimate goal is to port these ideas to realistic languages for programming embedded systems such as 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 and decidability results in the core language model. Our language, type theory, and metatheory are presented in the following Sections 2 through 5. We then define a type checking algorithm that is demonstrably sound with respect to the type theory specification in Sec. 6. To illustrate how our proposed system can support WSN applications programming we present and discuss an extended example in Sec. 7, in particular we explore bandwidth reduction via address minimization as mentioned above.

2 The Core Language

In this Section we define and discuss the core $\langle \text{ML} \rangle$ 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 $\langle \text{ML} \rangle$ Syntax and Semantics

The $\langle \text{ML} \rangle$ 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” \mathbf{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 $\langle e \rangle$

$\frac{\text{RCONST}}{\delta(\mathbf{c}, v) = e} \quad \frac{\mathbf{c} \ v \rightarrow e}{\mathbf{c} \ v \rightarrow e}$	$\text{RAPP} \quad (\lambda x : \tau.e)v \rightarrow e[v/x]$	$\text{RTLIFT} \quad \mathbf{tlet} \ t \preccurlyeq \tau = \tau' \ \mathbf{in} \ e \rightarrow e[\tau'/t]$	$\text{RCAST} \quad \frac{v : \tau}{(\tau)v \rightarrow v}$
$\text{RAPP}_{\Pi} \quad (\Lambda t \preccurlyeq \tau.e)\tau' \rightarrow e[\tau'/t]$	$\text{RRUN} \quad \frac{e \rightarrow^* v}{\mathbf{run} \ \langle e \rangle \rightarrow v}$	$\text{RLIFT} \quad \mathbf{lift} \ v \rightarrow \langle v \rangle$	$\text{RCONTEXT} \quad \frac{e \rightarrow e'}{E[e] \rightarrow E[e']}$

Fig. 2. $\langle \text{ML} \rangle$ Core Operational Semantics

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.* $\sim e$ of MetaML; since this form is common in staged programming languages we discuss this design choice in detail below.

Central to the prevention of cross-stage persistence is our definition of term substitution. 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 $\langle e \rangle[e'/x] = \langle e[e'/x] \rangle$, 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' to 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:

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

We can similarly define type substitutions $\tau[\tau'/t]$, except the definition here is even more standard, in particular $\langle \cdot \tau \cdot \rangle[\tau'/t] = \langle \cdot \tau[\tau'/t] \cdot \rangle$. This is because types, being static entities, should be allowed to persist across stages.

The operational semantics of $\langle \text{ML} \rangle$ 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^* . 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 axiomitizes our interpretation of program constants \mathbf{c} . The semantics of casting are predicated on a notion of typing defined in the following section.

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. $\langle \text{ML} \rangle$ in fact supports arbitrary stages.

Exploiting MetaML-style staging Two primitive MetaML expressions are directly reflected in $\langle \text{ML} \rangle$, 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 $\langle \text{ML} \rangle$ 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 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* $\langle \text{sense_data} \rangle [0, 1]$, which will be evaluated on the hub to:

$$\langle \text{sense_data} = 0 \parallel \text{sense_data} = 1 \parallel \text{false} \rangle$$

Value Migration via Lift and Run The semantics of $\langle \text{ML} \rangle$'s substitution-based term reduction itself disallows cross stage persistence; for example the term $(\lambda x : \mathbf{uint}. \langle x \rangle) 3$ is stuck since 3 cannot be substituted into the code $\langle x \rangle$. But on the other hand it is often necessary to allow migration of values across stages to allow the computation of a value that is then “glued” into the program-as-data. For our purposes **lifting** a value from the meta to the object language and **running** object code from the meta level are sufficient and indeed function as duals. For example in Fig. 3, the expression **lift** (*hdl*) 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 $\sim e$ that can “demote” e from the object stage back to the meta stage. For instance, rather than writing:

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

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

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

This syntax comparison shows how we *implicitly* view any free variable in a code block as a meta-variable ranging over code, in contrast with the MetaML convention

```

member : ⟨int⟩ → int list → ⟨bool⟩
member x l =
  if l = nil then ⟨false⟩ else
    let h = lift (hd l) in
      let tl = member x (tail l) in
        ⟨x = h || tl⟩
run(member⟨sense_data⟩[0, 1])

```

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

of viewing such variables as being at the object level unless explicitly escaped with \sim . In the ⟨ML⟩ syntax of implicitly escaping variables such as $x/h/tl$ above, the escape operator becomes less important; in general, a MetaML expression $\langle C[\sim e] \rangle$ can often be re-written as ⟨ML⟩ expression $(\lambda x. \langle C[x] \rangle)(e)$ where C is a program context. The only case where this rewriting fails is when e contains free variables at the object (not meta) level that should be captured by the code in C ; in ⟨ML⟩ those free object variables must be explicitly parameterized with λ -abstraction and later passed explicit arguments, making programs a bit more complex.

So supporting escape does increase programmability. Nevertheless, we choose to avoid it as it leads to significant complexities for static type checking (Nanevski, 2002; Chen & Xi, 2003; Taha & Nielsen, 2003), and in the WSN examples we have studied we have not run up against expressivity constraints using the escape-free ⟨ML⟩ syntax. Thus we obtain a simple and elegant type system, even when mutable state is added (state is a particularly pernicious problem for the interpretation of escape).

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 $\Delta t \preceq \tau.e$; 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 **tle** construct for

$\text{REFLS} \quad \frac{}{\Delta \vdash \tau \preceq \tau}$	$\text{COERCES} \quad \frac{\Delta(t) = \tau}{\Delta \vdash t \preceq \tau}$	$\text{CODES} \quad \frac{\Delta \vdash \tau_1 \preceq \tau_2}{\Delta \vdash \langle \cdot \tau_1 \cdot \rangle \preceq \langle \cdot \tau_2 \cdot \rangle}$	$\text{TRANS} \quad \frac{\Delta \vdash \tau_1 \preceq \tau_2 \quad \Delta \vdash \tau_2 \preceq \tau_3}{\Delta \vdash \tau_1 \preceq \tau_3}$
$\text{FNS} \quad \frac{\Delta \vdash \tau'_1 \preceq \tau_1 \quad \Delta \vdash \tau_2 \preceq \tau'_2}{\Delta \vdash \tau_1 \rightarrow \tau_2 \preceq \tau'_1 \rightarrow \tau'_2}$		$\text{TYPES} \quad \frac{\Delta \vdash \tau \preceq \tau'}{\Delta \vdash \mathbf{type}[\tau] \preceq \mathbf{type}[\tau']}$	
$\text{PI}S \quad \frac{\Delta; t \preceq \tau_0 \vdash \tau \preceq \tau'}{\Delta \vdash (\Pi t \preceq \tau_0. \tau) \preceq (\Pi t \preceq \tau_0. \tau')}$		$\text{EXISTS}S \quad \frac{\Delta; t \preceq \tau \vdash \tau' \preceq \tau''}{\Delta \vdash (\exists t \preceq \tau. \tau') \preceq (\exists t \preceq \tau. \tau'')}$	

Fig. 4. Subtyping Rules

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. 3.4 and Sec. 7 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 (McKinna, 2006), 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 (Crary *et al.*, 2002; Brady & Hammond, 2006).

3.2 Type Forms and Subtyping

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 $\mathbf{type}[\tau]$, that represents the type of dynamically constructed type values. Intuitively, $\mathbf{type}[\tau]$ 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.

Additionally, we introduce an existential type form for typing \mathbf{tlet} expression forms. Elimination and introduction rules for this form will ensure the \exists -bound variables are “eigenvariables”, i.e. they are distinct outside their scope. This allows “hiding” the static type of the \mathbf{tlet} -bound value whose type is inherently dynamic. \exists types are discussed in more detail upon presentation of the type judgement rules in Sec. 3.3.1. Both \exists and Π types are defined to be equivalent up to α -renaming.

We define the subtyping relation $\Delta \vdash \tau \preceq \tau'$ in Fig. 4; it is predicated on type *coercions* Δ . 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; the relation is mostly standard except for covariant extension of subtyping to **type** and code types.

Definition 3.1

A *coercion* Δ is a function from type variables to types. We write $\Delta; t \preceq \tau$ to denote the coercion that maps t to τ' but agrees with Δ on all other points.

In general, we will require that coercions and subtyping judgements be closed in the following sense:

Definition 3.2

A coercion Δ is *closed* iff every free type variable t in the codomain of Δ is also in its domain. A subtyping judgement $\Delta \vdash \tau_1 \preceq \tau_2$ is *closed* iff $\text{fv}(\tau_1, \tau_2) \subseteq \text{dom}(\Delta)$.

The reader will note that bound coercions must be equivalent to compare Π and \exists 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 (Ghelli & Pierce, 1998).

3.3 Type Judgements and Validity

Type judgements in our system are of the form $\Gamma, \Delta \vdash e : \tau$, where Γ is defined as follows.

Definition 3.3

An *environment* Γ is a function from term variables to types. We write $\Gamma; x : \tau$ to denote the coercion that maps x to τ' but agrees with Γ on all other points.

Derivability of type judgements is defined in terms of type derivation rules in Fig. 5. This type discipline disallows cross-stage persistence, in particular the CODE rule ensures that variables occurring within code are implicitly treated as code values at the same or greater stage; for this purpose we define

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

A weakening rule WEAKEN is included to be used in conjunction with the CODE rule: we wish to allow term variables to occur outside of code brackets within their scope.

Note that application of type abstraction in the APP Π 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. Type validity is then defined as follows:

Definition 3.4

A type judgement $\Gamma, \Delta \vdash e : \tau$ is *valid* iff it is derivable. We write $e : \tau$ iff $\emptyset, \emptyset \vdash e : \tau$.

$\frac{}{\Gamma, \Delta \vdash \mathbf{c} : \kappa(\mathbf{c})}$	$\frac{\text{VAR} \quad \Gamma(x) = \tau}{\Gamma, \Delta \vdash x : \tau}$	$\frac{}{\Gamma, \Delta \vdash \tau : \mathbf{type}[\tau]}$
$\frac{\text{APP}_{\Pi} \quad \Gamma, \Delta \vdash e : \Pi t \preccurlyeq \tau'' . \tau' \quad \Delta \vdash \tau \preccurlyeq \tau''}{\Gamma, \Delta \vdash e \tau : \tau'[\tau/t]}$	$\frac{\text{APP} \quad \Gamma, \Delta \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma, \Delta \vdash e_2 : \tau'}{\Gamma, \Delta \vdash e_1 e_2 : \tau}$	
$\frac{\text{ABS} \quad \Gamma, x : \tau, \Delta \vdash e : \tau'}{\Gamma, \Delta \vdash \lambda x : \tau . e : \tau \rightarrow \tau'}$	$\frac{\text{ABS}_{\Lambda} \quad \Gamma, \Delta; t \preccurlyeq \tau \vdash e : \tau'}{\Gamma, \Delta \vdash \Lambda t \preccurlyeq \tau . e : \Pi t \preccurlyeq \tau . \tau'}$	$\frac{\text{CODE} \quad \Gamma, \Delta \vdash e : \tau}{\langle \Gamma \cdot \rangle, \Delta \vdash \langle e \rangle : \langle \tau \cdot \rangle}$
$\frac{\text{WEAKEN} \quad \Gamma, \Delta \vdash e : \tau \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : \tau', \Delta \vdash e : \tau}$	$\frac{\text{RUN} \quad \Gamma, \Delta \vdash e : \langle \tau \cdot \rangle}{\Gamma, \Delta \vdash \mathbf{run} \ e : \tau}$	$\frac{\text{LIFT} \quad \Gamma, \Delta \vdash e : \tau}{\Gamma, \Delta \vdash \mathbf{lift} \ e : \langle \tau \cdot \rangle}$
$\frac{\text{CAST} \quad \Gamma, \Delta \vdash e : \tau'}{\Gamma, \Delta \vdash (\tau) e : \tau}$	$\frac{\text{SUB} \quad \Gamma, \Delta \vdash e : \tau' \quad \Delta \vdash \tau' \preccurlyeq \tau}{\Gamma, \Delta \vdash e : \tau}$	$\frac{\exists\text{-INTRO} \quad \Gamma, \Delta; t \preccurlyeq \tau \vdash e : \tau'}{\Gamma, \Delta \vdash e : \exists t \preccurlyeq \tau . \tau'}$
$\frac{\exists\text{-ELIM} \quad \Gamma, \Delta \vdash e : \exists t \preccurlyeq \tau . \tau'}{\Gamma, \Delta; t \preccurlyeq \tau \vdash e : \tau'}$	$\frac{\text{TLET} \quad \Gamma, \Delta \vdash e : \mathbf{type}[\tau''] \quad \Delta \vdash \tau'' \preccurlyeq \tau' \quad \Gamma, \Delta; t \preccurlyeq \tau' \vdash e' : \tau}{\Gamma, \Delta; t \preccurlyeq \tau' \vdash \mathbf{tlet} \ t \preccurlyeq \tau' = e \ \mathbf{in} \ e' : \tau}$	

Fig. 5. Type Judgement Rules

3.3.1 On \exists Types and Normal Forms

As discussed above, the purpose of \exists is to bind **tlet**-declared variables in types, to “hide” its inherently dynamic type at compile time. Observe in particular that to use an \exists -bound type the binder must be eliminated via \exists -ELIM, which always introduces a fresh instance of the type since Δ is a function. Despite this specialized intent, we have presented existential introduction and elimination in a general form, since this presentation is more standard and appropriate to this foundational exposition.

However, it is easy to imagine a complete normal-form derivation of types that uses a restricted subset of type forms. Such a normal form derivation would always perform \exists -INTRO just before an instance of ABS_{Λ} , and always perform \exists -ELIM just after an instance of APP_{Λ} . The resulting restricted type form can be shown to be in a one-to-one correspondence with the Π type form $\Pi t \circ \Delta . \tau$ defined in a preliminary presentation of this work (Liu *et al.*, 2009), where Δ contains the upper bounds for t as well as all **tlet**-bound variables in scope. This form is discussed at greater length in Sec. 6.2.1. It is possible that for practical purposes such a type form will be more user friendly.

3.4 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 $\langle \text{ML} \rangle$ this can be written as

$$\Lambda \text{addr_t}.\langle \lambda \text{addr} : \text{addr_t}.e \rangle$$

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

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

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. Observe how the type parameter *addr_t* is used within object code in the $\dots \lambda \text{addr} : \text{addr_t} \dots$ declaration, but the actual parameter **uint8** is a type and not “code that is a type” $\langle \mathbf{uint8} \rangle$. This highlights how our system supports cross-stage persistence of types (but not terms); this is sound because types “transcend” process spaces in that they can be interpreted correctly at any stage.

Type Bounds and Subtyping The benefit of static type checking for staged code has been widely discussed in recent efforts in meta programming (Moggi *et al.*, 1999; Calcagno *et al.*, 2000; Xi *et al.*, 2003; Chen & Xi, 2003; Taha & Nielsen, 2003). As in other contexts, polymorphism increases the expressivity of type systems for staged code, but note that e.g. 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 assigned a universal type within the scope of the function and hence be unusable.

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

$$\Lambda \text{addr_t} \preceq \mathbf{uint}.\langle \lambda \text{addr} : \text{addr_t}.e \rangle$$

With this bound, the type system can assume the type of *addr* is at least **uint** when *e* is typechecked, and use it as such. 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.

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 $\langle \text{ML} \rangle$, 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 (Ganz *et al.*, 2001; Krishnamurthi *et al.*, 1999), except that most people – including the

macros users themselves – complain they are not expressive enough. Treating types as values in $\langle \text{ML} \rangle$ provides programmers with a flexible way to define constructs such as the above (in fact, arbitrary $\langle \text{ML} \rangle$ programs are allowed to define \mathbb{T}), 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 $\langle \text{ML} \rangle$ program fragment:

```
tlet tcond  $\preceq$  uint32 = (if e0 then uint16 else uint32) in
let rft = ( $\Lambda$ addr_t  $\preceq$  uint32. $\langle \lambda$ addr : addr_t.e) tcond in
...
```

Here the **tlet** $\dots = \dots$ **in** expression is similar to a **let** $\dots = \dots$ **in**, except that it binds types. The binder **tlet** serves a critical purpose in the formalism: any type-valued expression such as the above conditional cannot directly appear in another type; only its **tlet**-ed name can. This means that $\langle \text{ML} \rangle$ types can only be dependent on types-as-expressions, not arbitrary expression forms. 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 $rft : \langle \cdot tcond \rightarrow \mathbf{result_t} \cdot \rangle$, under type constraint $tcond \preceq \mathbf{uint32}$. This type can then be *existentially* bound to close the type judgement.

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

$$\mathbf{type}[\tau] = \{\tau' \mid \tau' \preceq \tau\}$$

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

if *e0* **then** **uint16** **else** **uint32** : **type[uint32]**

which is straightforward since **uint16** \preceq **uint32**.

Casting To “close the loop” on runtime-dependent types as defined above we need to find a way to populate these types in spite of not knowing what value (type) they will take on 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 the running example, the *rft* 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 (Aiken *et al.*, 1994; Pottier, 2000) which are suited for this purpose, but for this presentation we opt for a simple typecast which is more expressive but incurs a runtime check. For example, in the elided part of the program fragment above, if we were to use *rft* we could write:

rft((*tcond*) 5)

which will typecast 5 to either **uint16** or **uint32** as appropriate at run-time.

s	::=	$\emptyset \mid s; e$
v	::=	$\dots \mid \{\ell_1 = v_1; \dots; \ell_n = v_n\} \mid x$
e	::=	$\dots \mid \{\ell_1 = e_1; \dots; \ell_n = e_n\} \mid e.\ell \mid \mathbf{ref} e \mid e := e \mid !e \mid s$
τ	::=	$\dots \mid \{\ell_1 : \tau_1; \dots; \ell_n : \tau_n\} \mid \mathbf{ref} \tau$
D	::=	$[\] \mid \mathbf{let} z = \mathbf{ref} v \mathbf{in} D$
m	::=	$\emptyset \mid m; z := v$
h	::=	$D[m]$

Fig. 6. Syntax for Records, Mutation, and Syntactic Stores

$\text{dom}(s)$	=	\emptyset
$\text{dom}(\mathbf{let} z = \mathbf{ref} v \mathbf{in} h)$	=	$\{z\} \cup \text{dom}(h)$
$\text{lkp} z (\mathbf{let} z' = \mathbf{ref} v \mathbf{in} h)$	=	$\text{lkp} z h \quad \text{if } z \neq z'$
$\text{lkp} z (\mathbf{let} z = \mathbf{ref} v \mathbf{in} D[m])$	=	$\text{lkp}' z v m$
$\text{lkp}' z v \emptyset$	=	v
$\text{lkp}' z v (m; z := v')$	=	v'
$\text{lkp}' z v (m; z' := v')$	=	$\text{lkp}' z v m \quad z \neq z'$

Fig. 7. Auxiliary Functions for Store Lookup

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 $\langle \text{ML} \rangle$ -style staging where we assume that distinct stages represent distinct process spaces.

In Fig. 6 we introduce new record and state expression forms that extend the syntactic definitions in Fig. 1, 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 (Honsell *et al.*, 1993). Syntactic stores may be interpreted as a mapping from variables to values via the `dom` and `lkp` functions defined in Fig. 7. We write $\text{dom}(h)$ to denote the domain of a store h , and $\text{lkp} z h$ to denote the value associated with variable z in a syntactic store h .

To define serialization, we will just project 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:

$\text{project } D[m] \ V = \text{project } D \ V [\text{project } m \ V]$	
$\text{project } [] \ V = []$	
$\text{project } (\text{let } z = \text{ref } v \text{ in } D) \ V = \text{project } D \ V$	$\text{if } z \notin V$
$\text{project } (\text{let } z = \text{ref } v \text{ in } D) \ V = (\text{let } x = \text{ref } v \text{ in } (\text{project } D \ V))$	$\text{if } z \in V$
$\text{project } \emptyset \ V = \emptyset$	
$\text{project } (m; z := v) \ V = \text{project } m \ V$	$\text{if } z \notin V$
$\text{project } (m; z := v) \ V = \text{project } m \ V; z := v$	$\text{if } z \in V$

Fig. 8. Projecting a Sub-Store

$\frac{\text{RRUN} \quad (e, \emptyset) \rightarrow^* (v, h')}{(\text{run } \langle e \rangle, h) \rightarrow (\text{serialize } v \ h', h)}$	$\frac{\text{RREF} \quad z \notin \text{dom}(D[m])}{(\text{ref } v, D[m]) \rightarrow ((), D[\text{let } z = \text{ref } v \text{ in } m])}$
$\frac{\text{RDEREF} \quad (!z, h) \rightarrow (\text{lkp } z \ h, h)}$	$\frac{\text{RASSIGN} \quad z \in \text{dom}(D[m])}{(z := v, D[m]) \rightarrow ((), D[m; z := v])}$
$\frac{\text{RLIFT} \quad (\text{lift } v, h) \rightarrow (\langle \text{serialize } v \ h \rangle, h)}$	$\frac{\text{RCONTEXT} \quad (e, h) \rightarrow (e', h')}{(E[e], h) \rightarrow (E[e'], h')}$

Fig. 9. Semantics of $\langle \text{ML} \rangle$ with Mutation and State*Definition 4.1*

Serialization of a value v given a store h is defined via the following function:

$$\text{serialize } v \ h = \text{let } D[m] = (\text{project } h \ (\text{reachable } v \ h)) \text{ in } D[m; v]$$

where project is defined in Fig. 8 and $\text{reachable } v \ h = V$ iff V contains all store locations reachable from v in h .

Now, we can define the operational semantics via a small-step relation \rightarrow 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.

The interesting rules are specified in Fig. 9. 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 . The subtyping and typing rules for records and references are standard, and are given in figures Fig. 10 and Fig. 11 as extensions to Fig. 4 and Fig. 5, respectively.

$$\begin{array}{c}
\text{RECS} \\
\frac{\Delta \vdash \tau_1 \preceq \tau'_1 \dots \quad \Delta \vdash \tau_n \preceq \tau'_n}{\Delta \vdash \{\ell_1 : \tau_1; \dots; \ell_n : \tau_n\} \preceq \{\ell_1 : \tau'_1; \dots; \ell_n : \tau'_n; \dots; \ell_{n+m} : \tau_{n+m}\}} \\
\\
\text{REFS} \\
\frac{\Delta \vdash \tau \preceq \tau' \quad \Delta \vdash \tau' \preceq \tau}{\Delta \vdash \mathbf{ref} \tau \preceq \mathbf{ref} \tau'}
\end{array}$$

Fig. 10. Additional Record and State Subtyping Rules

$$\begin{array}{c}
\text{REC} \qquad \qquad \qquad \text{REF} \\
\frac{\Gamma, \Delta \vdash e_1 : \tau_1 \quad \dots \quad \Gamma, \Delta \vdash e_n : \tau_n}{\Gamma, \Delta \vdash \{\ell_1 = e_1; \dots; \ell_n = e_n\} : \{\ell_1 : \tau_1; \dots; \ell_n : \tau_n\}} \qquad \frac{\Gamma, \Delta \vdash e : \tau}{\Gamma, \Delta \vdash \mathbf{ref} e : \mathbf{ref} \tau} \\
\\
\text{SET} \qquad \qquad \qquad \text{GET} \\
\frac{\Gamma, \Delta \vdash e : \mathbf{ref} \tau \quad \Gamma, \Delta \vdash e' : \tau}{\Gamma, \Delta \vdash e := e' : \tau} \qquad \frac{\Gamma, \Delta \vdash e : \mathbf{ref} \tau}{\Gamma, \Delta \vdash !e : \tau}
\end{array}$$

Fig. 11. Additional Record and State Type Judgement Rules

5 Type Safety

In this section we prove a formal type safety result for our system. Along with standard properties this result ensures that programs respect process separation, since process separation is enforced by the dynamic semantics of $\langle \text{ML} \rangle$.

Our argument for type safety employs a mostly standard subject reduction strategy. However, the types of **tlet** expression forms present a twist since the scope of the bound type variable is extruded in the judgement coercion, or the type of the expression is \exists bound. Either way, the types of the reductions may be instances of the types of the **tlet**-form redices. We formalize this notion of instantiation as follows, and wlog impose a canonical form on type judgements to disallow top-level \exists types.

Definition 5.1

We say that (Δ', τ') is an instance of (Δ, τ) iff $\Delta = \Delta'; \overline{\{t \preceq \tau\}}$ and $\tau' = \tau[\overline{\tau'/t}]$ where $\Delta' \vdash \overline{\tau'} \preceq \tau$.

Definition 5.2

A judgement $\emptyset, \Delta \vdash e : \tau$ is *canonical* iff it is valid and τ is not an existential type.

In our statement of type preservation (Theorem 5.1), we will show that evaluation preserves typings modulo instantiation.

To begin the argument proper, 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 $\tau = \mathbf{type}[\tau']$ for some τ' then $v = \tau''$ for some τ'' .
3. if $\tau = \Pi t \circ \Delta'.\tau'$ for some t, Δ', τ' then $v = \Lambda t \preceq \tau''.e$ 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 $\Gamma, \Delta; t \preceq \tau'_0 \vdash e : \tau_0$ and $\Gamma, \Delta \vdash \tau_1 : \mathbf{type}[\tau'_1]$ with $\Delta \vdash \tau'_1 \preceq \tau'_0$, 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 $\Delta \vdash \tau_1 \preceq \tau'_0$, 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:

$$e = \langle e' \rangle \quad \tau'_0 = \langle \cdot \tau' \cdot \rangle \quad \Gamma = \langle \cdot \Gamma' \cdot \rangle \quad \tau_0 = \langle \cdot \tau \cdot \rangle$$

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 $\Delta \vdash \tau'' \preceq \tau'$, 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`. \square

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 $\emptyset, \Delta \vdash D[m; e] : \tau$ is.

An important corollary of this definition is that code values at run-time 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 $\emptyset, \Delta \vdash \text{serialize } v \ h : \tau$.

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 5.4

The *length of an evaluation relation* $(e, h) \rightarrow^* (e', h')$ 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. The requirement that reduced expressions typings may be instances of initial typings was anticipated at the beginning of this section; note in the **tle**-form case this property will fall out as a result of the relevant reduction rule and Lemma 5.2.

Theorem 5.1 (Type Preservation)

If $(e_0, h_0) : \tau_0 \circ \Delta_0$ is valid and $(e_0, h_0) \rightarrow^* (e_n, h_n)$, then there exists (Δ_n, τ_n) which is an instance of (Δ_0, τ_0) such that $(e_n, h_n) : \tau_n \circ \Delta_n$.

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 $(e_0, h_0) \rightarrow^* (e_1, h_1)$ where (e_1, h_1) is irreducible and e_1 is not a value.

6 Type Checking

We now define a type checking algorithm that is demonstrably sound with respect to the logical system. We also conjecture, though do not prove, that it is complete. We break up this task into subcomponents, one is the realization of the subtyping relation as an algorithm, and the other is the same for typing judgements. We note that any sort of type unification or constraint solution is unnecessary, since ours is inherently a type *checking*, not *reconstruction*, system.

6.1 Algorithmic Subtyping

To define a subtyping algorithm we mostly follow a technique invented in previous work on bounded existential type checking (Ghelli & Pierce, 1998). We begin by defining type *promotion*, whereby structure can be imposed on type variables by relating them to their least structured upper bound in a given Δ .

Definition 6.1

$\frac{\text{REFLWS}}{\Delta \vdash_W \tau \preceq \tau}$	$\frac{\text{CODEWS}}{\Delta \vdash_W \tau_1 \preceq \tau_2} \quad \frac{\Delta \vdash_W \tau_1 \preceq \tau_2}{\Delta \vdash_W \langle \cdot \tau_1 \cdot \rangle \preceq \langle \cdot \tau_2 \cdot \rangle}$	$\frac{\text{FNWS}}{\Delta \vdash_W \tau_1 \rightarrow \tau_2 \preceq \tau'_1 \rightarrow \tau'_2} \quad \frac{\Delta \vdash_W \tau'_1 \preceq \tau_1 \quad \Delta \vdash_W \tau_2 \preceq \tau'_2}{\Delta \vdash_W \tau_1 \rightarrow \tau_2 \preceq \tau'_1 \rightarrow \tau'_2}$
$\frac{\text{TYPEWS}}{\Delta \vdash_W \mathbf{type}[\tau] \preceq \mathbf{type}[\tau']}$	$\frac{\text{PIWS}}{\Delta \vdash_W (\Pi t \preceq \tau_0. \tau) \preceq (\Pi t \preceq \tau_0. \tau')}$	
$\frac{\text{EXISTSWWS}}{\Delta \vdash_W (\exists t \preceq \tau. \tau_0) \preceq (\exists t \preceq \tau. \tau_1)} \quad \frac{\Delta; t \preceq \tau \vdash \tau_0 \preceq \tau_1}{\Delta; t \preceq \tau \vdash \tau_0 \preceq \tau_1}$	$\frac{\text{PROMOTIEWS}}{\Delta \vdash_W \tau_0 \preceq \tau_1} \quad \frac{\Delta \vdash \tau_0 \ll \tau'_0 \quad \Delta \vdash \tau_1 \ll \tau'_1 \quad \Delta \vdash_W \tau'_0 \preceq \tau'_1}{\Delta \vdash_W \tau_0 \preceq \tau_1}$	

Fig. 12. Algorithmic Subtyping Rules

The relation \ll promotes a type variable to the structured type which is its lub given a coercion:

$$\frac{\Delta \vdash \Delta(t) \ll \tau}{\Delta \vdash t \ll \tau} \quad \frac{\neg \exists t. \tau = t}{\Delta \vdash \tau \ll \tau}$$

Next, we define a relation $\Delta \vdash_W \tau \preceq \tau'$ which is the algorithmic version of subtyping. The derivation rules for the relation are given in Fig. 12. We state the correctness of this relation in the next Lemma. The result follows by straightforward induction on the derivation of either relation in the equivalence:

Lemma 6.1

$$\Delta \vdash_W \tau \preceq \tau' \text{ iff } \Delta \vdash \tau \preceq \tau'$$

We note that in keeping with the logical specification of our type theory, in type checking we take Π and \exists types to be equivalent up to α -renaming, so implementations of the algorithmic subtyping relation must perform explicit α -renaming when checking subtyping of these forms. But this is straightforward, and it is easy to implement this relation, so we have:

Lemma 6.2

The relation $\Delta \vdash_W \tau \preceq \tau'$ induces a decision procedure: there is an algorithm which given Δ , τ and τ' returns *true* if $\Delta \vdash_W \tau \preceq \tau'$ and *false* otherwise.

6.2 Algorithmic Type Derivation

We here define an algorithmic typing relation $\Gamma, \Delta \vdash_W e : \tau$ that defines an algorithm which given Γ , Δ , and e , returns a type τ , such that $\Gamma, \Delta \vdash_W e : \tau$ implies $\Gamma, \Delta \vdash e : \tau$. Type checking is predicated on several preliminary definitions. In order to move between code levels in static typing, we define a function *peel* as follows:

Definition 6.2

The function *peel* removes a layer of code type from an environment:

$$\begin{aligned} \text{peel}(\emptyset) &= \emptyset \\ \text{peel}(\Gamma; x : \langle \cdot \tau \cdot \rangle) &= \text{peel}(\Gamma); x : \tau \end{aligned}$$

$\frac{\text{CONSTW}}{\Gamma, \Delta \vdash_W \mathbf{c} : \kappa(\mathbf{c})}$	$\frac{\text{VARW}}{\Gamma(x) = \tau}$ $\frac{\Gamma(x) = \tau}{\Gamma, \Delta \vdash_W x : \tau}$	$\frac{\text{TYPEW}}{\Gamma, \Delta \vdash_W \tau : \mathbf{type}[\tau]}$
$\frac{\text{APP}_{\Pi} \text{W}}{\Gamma, \Delta \vdash_W e : \exists \Delta'. \tau_0 \quad \Delta; \Delta' \vdash \tau_0 \ll \Pi t \approx \tau''. \tau' \quad \Delta; \Delta' \vdash_W \tau \approx \tau''}{\Gamma, \Delta \vdash_W e \tau : \exists \Delta'. \tau'[\tau/t]}$		
$\frac{\text{APPW}}{\Delta; \Delta' \vdash \tau_0 \ll \tau' \rightarrow \tau \quad \Gamma, \Delta \vdash_W e_1 : \exists \Delta'. \tau_0 \quad \Gamma, \Delta \vdash_W e_2 : \tau'' \quad \Delta; \Delta' \vdash_W \tau'' \approx \tau'}{\Gamma, \Delta \vdash_W e_1 e_2 : \exists \Delta'. \tau}$		
$\frac{\text{ABSW}}{\Gamma; x : \tau, \Delta \vdash_W e : \exists \Delta'. \tau'}{\Gamma, \Delta \vdash_W \lambda x : \tau. e : \exists \Delta'. \tau \rightarrow \tau'}$	$\frac{\text{ABS}_{\Lambda} \text{W}}{\Gamma, \Delta; t \approx \tau \vdash_W e : \tau'}{\Gamma, \Delta \vdash_W \Lambda t \approx \tau. e : \Pi t \approx \tau. \tau'}$	
$\frac{\text{CODEW}}{\text{peel}(\Gamma _{\text{fv}(e)}), \Delta \vdash_W e : \exists \Delta'. \tau}{\Gamma, \Delta \vdash_W \langle e \rangle : \exists \Delta'. \langle \cdot \tau \cdot \rangle}$	$\frac{\text{RUNW}}{\Gamma, \Delta \vdash_W e : \exists \Delta'. \tau_0 \quad \Delta; \Delta' \vdash \tau_0 \ll \langle \cdot \tau \cdot \rangle}{\Gamma, \Delta \vdash_W \mathbf{run} e : \exists \Delta'. \tau}$	
$\frac{\text{LIFTW}}{\Gamma, \Delta \vdash_W e : \exists \Delta'. \tau}{\Gamma, \Delta \vdash_W \mathbf{lift} e : \exists \Delta'. \langle \cdot \tau \cdot \rangle}$		$\frac{\text{CASTW}}{\Gamma, \Delta \vdash_W e : \tau'}{\Gamma, \Delta \vdash_W (\tau) e : \tau}$
$\frac{\text{TLETW}}{\Delta; \Delta' \vdash \tau_0 \ll \mathbf{type}[\tau''] \quad \Gamma, \Delta \vdash_W e : \exists \Delta'. \tau_0 \quad \Delta; \Delta' \vdash_W \tau'' \approx \tau' \quad \Gamma, \Delta; t \approx \tau' \vdash_W e' : \exists \Delta''. \tau}{\Gamma, \Delta \vdash_W \mathbf{tlet} t \approx \tau' = e \mathbf{in} e' : \exists (\Delta'; \Delta''; t \approx \tau'). \tau}$		

Fig. 13. Type Checking Rules

Note that $\text{peel}(\Gamma)$ is defined only if all bindings in Γ are of code type, and if $\text{peel}(\Gamma)$ is defined then $\langle \cdot \text{peel}(\Gamma) \cdot \rangle = \Gamma$.

Also, when moving between code levels, it is desirable to isolate only those variables that occur free within a deeper level (which must have code type). Hence:

Definition 6.3

We write $\Gamma|_{\{x_1, \dots, x_n\}}$ to denote the environment Γ' where $\text{dom}(\Gamma') = \{x_1, \dots, x_n\}$ and $\Gamma'(x) = \Gamma(x)$ for all $x \in \{x_1, \dots, x_n\}$.

Given these definitions, and also decidability of subtyping as noted above, the completely syntax-directed nature of the type checking immediately implies decidability of type checking:

Lemma 6.3 (Decidability of Type Checking)

Relation $\Gamma, \Delta \vdash_W e : \tau$ induces a decision procedure: there is an algorithm which given Γ, Δ , and e produces τ if $\Gamma, \Delta \vdash_W e : \tau$, and fails otherwise.

6.2.1 Type Checking as Normal Form Derivation

A more interesting preliminary definition is related to the restricted type forms produced in checking. In particular, existential types get “bunched up” in the process of type checking so that it is convenient to introduce shorthand to denote sequences of existential bindings:

Definition 6.4

Given $\Delta = t_1 \preccurlyeq \tau_1; \dots; t_n \preccurlyeq \tau_n$, we define:

$$\exists\Delta.\tau \triangleq \exists t_1 \preccurlyeq \tau_1 \dots \exists t_n \preccurlyeq \tau_n.\tau$$

and:

$$\Delta'; \Delta \triangleq \Delta'; t_1 \preccurlyeq \tau_1; \dots; t_n \preccurlyeq \tau_n$$

Note that if $\Delta = \emptyset$ then $\exists\Delta.\tau = \tau$.

Type derivation rules for the algorithmic system are given in Fig. 13. Type judgements in this system are of the form $\Gamma, \Delta \vdash_W e : \exists\Delta'\tau$ where Γ keeps track of type binding annotations on term abstractions, Δ keeps track of type binding annotations on type abstractions, and Δ' contains extruded typing assumptions for **tlet**-bound variables within e . This latter trick distinguishes the two type binding forms and allows **tlet**-bound variables to be “percolated upwards”, while Λ -bound variables are passed downwards. It is sound, since in the logical typing rules any syntax-directed rule can be preceded by an instance of \exists -ELIM and followed by an instance of \exists -INTRO to replay the strategy.

Indeed, while the type checking rules are syntax-directed, it is illuminating to think of them as a normal form logical derivations, incorporating implicit uses of \exists -ELIM and \exists -INTRO. From this perspective, implicit instances of \exists -ELIM occur before every point where subtyping is checked or used in the form of promotion. The point where non-trivial \exists -INTRO implicitly occurs is at the point of Λ abstraction. By non-trivial here we mean where the resulting existential type becomes the subterm of another type, not a form that can be immediately eliminated. This is essential for soundness, since Λ -bound type variables can occur within the upper bounds of **tlet**-bindings within their scope. This results in a restricted type form, where all Π types in type inference are of the form $\Pi t \preccurlyeq \tau.\exists\Delta.\tau'$, and occurrences of t in Δ are instantiated appropriately when Π types are applied. As already noted in Sec. 3.3.1, this is equivalent to the type form presented in a preliminary version of this paper (Liu *et al.*, 2009).

In the metatheory it is straightforward to show that any type checking derivation can be transformed into a valid logical typing derivation, in particular the normal form outlined above. Soundness of type checking follows:

Lemma 6.4 (Soundness of Type Checking)

$\Gamma, \Delta \vdash_W e : \tau$ implies $\Gamma, \Delta \vdash e : \tau$.

```

send =  $\Lambda$  addr_t  $\preceq$  uint.
       $\Lambda$  message_header_t  $\preceq$  { src : addr_t
                                dest : addr_t }
       $\Lambda$  msg_t  $\preceq$  { header : message_header_t }.
       $\lambda$  psend :  $\langle$ ·msg_t  $\rightarrow$  result_t $\rangle$ .
       $\lambda$  self :  $\langle$ ·addr_t $\rangle$ .
       $\langle$   $\lambda$  addr : addr_t.
         $\lambda$  msg : msg_t.
          msg.header.src := self;
          msg.header.dest := addr;
          psend msg
       $\rangle$ 
radio =  $\Lambda$  msg_t  $\preceq$  { header : message_header_t }.  $\langle$   $\lambda$  msg : msg_t...  $\rangle$ 

```

Fig. 14. Code Snippet for *send*

```

moteCode =  $\Lambda$  addr_t  $\preceq$  uint.
           $\Lambda$  msg_t  $\preceq$  { header : { src : addr_t; dest : addr_t }; data : uint8[] }.
           $\lambda$  sendf :  $\langle$ ·addr_t  $\rightarrow$  msg_t  $\rightarrow$  result_t $\rangle$ .
           $\lambda$  neighbors :  $\langle$ ·addr_t[] $\rangle$ .
           $\lambda$  neighbor_num :  $\langle$ ·uint16 $\rangle$ .
           $\langle$  msg_t m;
            m.data = "hello";
            for(uint16 i = 0; i < neighbor_num; i++){
              sendf neighbors[i] m
            }
           $\rangle$ 

```

Fig. 15. Code for Motes

7 A Programming Example

In this section, we use sensor network programming as a case study to demonstrate how \langle ML \rangle supports programming practice. Our focus here is on highlighting 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 a 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 (Madden *et al.*, 2002). 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 speed up its code, but to minimize the bandwidth it would consume.

The example we present in this section fleshes out this observation. 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 of energy saved 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 communicate with more than 16 neighbors due to some address assignment or neighborhood discovery protocol, we can assign a `uint4` type to `addr`, rather than `uint64`, and save radio power.

We will use some language constructs beyond the $\langle ML \rangle$ formal core in the example, including `for` loops and arrays; adding these features is not difficult technically and clarifies the presentation. 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, and that all of these integer sizes are a subtype of `uint`. Subtyping relations defined as such may lead to memory layout conversions when a subtype value is 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 of a `tlet`-bound variable is not given in our example it can be assumed to be the same as the type of the expression bound to the variable.

7.1 A Specializable “Send” Snippet

In the standard TinyOS sensor network platform (Hill *et al.*, 2000), 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 uses type `message_t` for messages will not necessarily be efficient: 64-bit addresses are hardcoded inside this data structure. This situation can be avoided in $\langle ML \rangle$, using the implementation of the `send` function given in Fig. 14. Function `send` is a staged program intended to build a piece of code implementing the message send functionality for a mote; it takes several type and code parameters and builds the underlying send function code. The first argument

of the underlying send, *addr*, denotes 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 tied to the power consumption of the final sensor network: when the *send* function is defined, it is abstracted to work on any subtype of **uint**. Depending on how *send* is instantiated, the code deployed on motes may either send messages with short addresses such as **uint4**, or long ones such as **uint64**.

Note that *send* ultimately must invoke some function on the physical layer to send the actual message by the radio. The particular physical-layer send can be customized, and that is achieved here by passing it in as an argument, *psend*. The signature of that argument indicates that it is another piece of staged code. The *psend* example more generally illustrates how library functions can be used in the context of staged code fragments. Note that the *send* definition above is applied at the meta-stage to produce the 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 using the cross-stage persistence of MetaML to implicitly take *psend* to be the function defined at the meta level, our calculus avoids a potential library version incompatibility that could arise in the MetaML view if a mote tried to use the hub *psend* code: that would be wrong because the hub is of a different architecture than the mote.

With this function defined, one way to produce a send function with all addresses being **uint4** 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 send here is *radio*, which we leave under-defined in this high-level presentation. A typecast is needed in the definition of *self*, as was explained in Sec. 3.4. *DATA_LEN* is an integer constant.

7.2 A Specializable Toy Program on Motes

The *send* code we have described in Fig. 14 is one function that could be used in a full application deployed to the motes by the hub. We now define a complete “hello world” mote application, in Fig. 15. The program just sends a “hello” message to the mote’s 1-hop neighbors.

The type of the message that eventually will be sent to all neighbors, *msg_t*, is generic and can be specialized. It can be of any record type with a *header* field and a *data* field which is a **uint8** array. The header contains at least 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”. Supporting hardcoding of neighbor information may seem 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 $\langle ML \rangle$ 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.)

```

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])(getNeigh 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)
                }
      getNeigh = 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
        }

```

Fig. 16. Bootstrapping Code for Hub

The *moteCode* expression takes another piece of staged code, *sendf*, as one of its arguments. Thus, on the hub, running the following code will deploy a specialization 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 of Sec. 7.1. In the fifth line, a (trivial) 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 the **run** will not necessarily run the code on a mote, only in a different deployment context. Including mote location in **run** is left to future work.

7.3 A Metaprogram on the Hub

Fig. 16 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 large, 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

$$coloring :: topology_t \rightarrow \mathbf{uint32}$$

to color the topology graph. The idea here is that sensors only talk to their immediate neighbors, so the unique addresses needed are the number of colors computed by the classic *n*-coloring algorithm. 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 in the range $[0..colors-1]$.

The rest of the function is largely copied from the code fragment deploying the motes, explained in Sec. 7.1 and Sec. 7.2. Note 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 *addr_t*. The neighbor information is also computed at the meta level based on the topology information *topo*. This is achieved by function *getNeigh*, which is purely a hub execution.

7.4 Discussion of the Example

The simplified example presented in this section does not cover 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 fields in 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. Refinements 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 neighbors of a node fluctuate between 12 and 120 for example. In this case, a costly redeployment of code may need to be performed. However, for many applications neighborhood sizes will remain within certain bounds for a sufficient duration to make this tradeoff more than favorable.

8 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 (Brady & Hammond, 2006) and interpreters (Pasalic *et al.*, 2002). Also related is work on program generation formalisms for compiler construction that leverage first class types and intensional polymorphism (Crary *et al.*, 2002). Tempo (Consel *et al.*, 1998) is a related system that integrates partial evaluation and type specialization for increasing efficiency of systems applications. Tempo is especially interesting to us because it is intended for application to C, which is a foundation of nesC. Perhaps the system most closely related to ours is Monnier and Shao's (Monnier & Shao, 2003), where type abstraction as a language construct is supported in a staged program calculus albeit following a standard System F_{\leq} style (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 (Kameyama *et al.*, 2009). Moggi and Fagorzi have established a monadic foundation for integrating staging with arbitrary side effects in a highly general and mathematically rigorous fashion (Moggi & Fagorzi, 2003). 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 (Taha & Sheard, 1997; Moggi *et al.*, 1999) and its more recent and robust implementation, MetaOCaml (Taha & *et. al.*, n.d.). MetaML has also been promoted as an effective foundation for embedded systems programming (Taha, 2004) and enjoys type safety results of the sort presented here (Taha *et al.*, 1998). 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 (Davies & Pfenning, 2001), higher-order abstract syntax (Xi *et al.*, 2003), and first-order abstract syntax with deBruijn indices (Chen & Xi, 2003). 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 (Nanevski, 2002; Chen & Xi, 2003; Calcagno *et al.*, 2000; Taha & Nielsen, 2003). 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} (Cardelli & Wegner, 1985), 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 (Harper & Morrisett, 1995), 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 (Mainland *et al.*, 2008). The main motivation of designing Flask is to allow FRP-based (Wan & Hudak, 2000) 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 $\langle e \rangle$ combined with an escape operator \tilde{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` (mig, n.d.). 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 (Aiken *et al.*, 1994; Pottier, 2000) or conditionally tagged type unions (Shapiro & Sridhar, n.d.) to avoid some of our need for typecasts and thus to gain more static type safety. It is possible to use much more expressive dependently-typed system such as Agda (Norell, 2007), but our focus is on a practical systems programming language and as such we require that typechecking be decidable.

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 marshal data to bit strings. These features will be important when we build a domain-specific language upon the foundation of $\langle ML \rangle$.

The $\langle ML \rangle$ calculus only represents an exploration of concepts. As a next step, we are interested both in porting the ideas presented here to more popular sensor network languages such as `nesC` (Gay *et al.*, 2003), and in directly implementing them in a functional language setting. The second route may appear difficult at first glance, since a functional language with first-class functions and dynamic allocation generally has more runtime overhead than typical sensor networks expect. This however is largely a non-issue for the meta-stage of a staged programming language, because meta-stage code is always executed on the resource-rich hub where efficiency is not a concern, and while here the hub and mote programming languages are identical, there is no reason that this must be the case. As for the specialized mote-stage code, there are precedent functional languages such as `Regiment` (Newton *et al.*, 2007) designed for sensor networks. Experiments have shown competitive performance can be achieved by placing restrictions on the mote language; for example, only statically bounded recursion is allowed in `Regiment`.

References

- Aiken, A., Wimmers, E. L., & Lakshman, T. K. (1994). Soft typing with conditional types. *Pages 163–173 of: Conference record of the twenty-first annual acm symposium on principles of programming languages.*
- Brady, Edwin, & Hammond, Kevin. (2006). A verified staged interpreter is a verified compiler. *Pages 111–120 of: GPCE '06: Proceedings of the 5th international conference on generative programming and component engineering.* New York, NY, USA: ACM.
- Calcagno, Cristiano, Moggi, Eugenio, & Taha, Walid. (2000). Closed types as a simple approach to safe imperative multi-stage programming. *Pages 25–36 of: ICALP '00: Proceedings of the 27th international colloquium on automata, languages and programming.* London, UK: Springer-Verlag.

- Cardelli, Luca, & Wegner, Peter. (1985). On understanding types, data abstraction, and polymorphism. *Acm comput. surv.*, **17**(4), 471–523.
- Chen, Chiyan, & Xi, Hongwei. (2003). Meta-programming through typeful code representation. *ICFP'03*.
- Consel, C., Hornof, L., Marlet, R., Muller, G., Thibault, S., Volanschi, E.-N., Lawall, J., & Noyé, J. (1998). Tempo: specializing systems applications and beyond. *Acm comput. surv.*, 19.
- Crary, Karl, Weirich, Stephanie, & Morrisett, Greg. (2002). Intensional polymorphism in type-erasure semantics. *J. funct. program.*, **12**(6), 567–600.
- Davies, Rowan, & Pfenning, Frank. (2001). A modal analysis of staged computation. *J. acm*, **48**(3), 555–604.
- Ganz, Steven E., Sabry, Amr, & Taha, Walid. (2001). Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. *Pages 74–85 of: ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on functional programming*. New York, NY, USA: ACM.
- Gay, David, Levis, Philip, von Behren, Robert, Welsh, Matt, Brewer, Eric, & Culler, David. (2003). The nesC language: A holistic approach to networked embedded systems. *Pages 1–11 of: PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on programming language design and implementation*. New York, NY, USA: ACM.
- Ghelli, Giorgio, & Pierce, Benjamin. (1998). Bounded existentials and minimal typing. *Theoretical computer science*, **193**(1-2), 75 – 96.
- Gibbons, Jeremy. (2007). *Datatype-Generic programming*. Pages 71, 1.
- Harper, Robert, & Morrisett, Greg. (1995). Compiling polymorphism using intensional type analysis. *Pages 130–141 of: In twenty-second acm symposium on principles of programming languages*. ACM Press.
- Hill, Jason, Szewczyk, Robert, Woo, Alec, Hollar, Seth, Culler, David E., & Pister, Kristofer S. J. (2000). System architecture directions for networked sensors. *Pages 93–104 of: Architectural support for programming languages and operating systems*.
- Honsell, Furio, Mason, Ian A., Smith, Scott, & Talcott, Carolyn. (1993). A variable typed logic of effects. *Information and computation*, **119**, 55–90.
- Kameyama, Yuki Yoshi, Kiselyov, Oleg, & Shan, Chung-chieh. (2009). Shifting the stage: staging with delimited control. *Pages 111–120 of: PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on partial evaluation and program manipulation*. New York, NY, USA: ACM.
- Krishnamurthi, Shriram, Felleisen, Matthias, & Duba, Bruce F. (1999). From macros to reusable generative programming. *Pages 105–120 of: In international symposium on generative and component-based software engineering, number 1799 in lecture notes in computer science*. Springer-Verlag.
- Liu, Yu David, Skalka, Christian, & Smith, Scott. (2009). Type-specialized staged programming with process separation. *Workshop on generic programming (WGP09)*.
- Madden, Samuel, Franklin, Michael J., Hellerstein, Joseph M., & Hong, Wei. (2002). TAG: a Tiny AGgregation service for ad-hoc sensor networks. *Sigops oper. syst. rev.*, **36**(SI), 131–146.
- Mainland, Geoffrey, Morrisett, Greg, & Welsh, Matt. 2008 (September). Flask: Staged functional programming for sensor networks. *13th ACM SIGPLAN international conference on functional programming (ICFP 2008)*.
- McKinna, James. (2006). Why dependent types matter. *Sigplan not.*, **41**(1), 1–1.
- `mig`. *mig: message interface generator for nesC*, available online at <http://www.tinyos.net/tinyos-1.x/doc/nesc/mig.html>.

- Moggi, Eugenio, & Fagorzi, Sonia. (2003). A monadic multi-stage metalanguage. *Pages 358–374 of: Gordon, Andrew D. (ed), FoSSaCS. Lecture Notes in Computer Science*, vol. 2620. Springer.
- Moggi, Eugenio, Taha, Walid, abidine Benaïssa, Zine El, & Sheard, Tim. (1999). An idealized MetaML: Simpler, and more expressive. *Pages 193–207 of: In european symposium on programming (ESOP)*. Springer-Verlag.
- Monnier, Stefan, & Shao, Zhong. (2003). Inlining as staged computation. *J. funct. program.*, **13**(3), 647–676.
- Murphy VII, Tom, Crary, Karl, Harper, Robert, & Pfenning, Frank. (2004). A symmetric modal lambda calculus for distributed computing. *Pages 286–295 of: LICS '04: Proceedings of the 19th annual IEEE symposium on logic in computer science*. Washington, DC, USA: IEEE Computer Society.
- Nanevski, Aleksandar. (2002). Meta-programming with names and necessity. *Pages 206–217 of: ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on functional programming*. New York, NY, USA: ACM.
- Newton, Ryan, Morrisett, Greg, & Welsh, Matt. (2007). The regiment macroprogramming system. *Pages 489–498 of: IPSN '07: Proceedings of the 6th international conference on information processing in sensor networks*.
- Norell, Ulf. (2007). *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology and Göteborg University.
- Pasalic, Emir, Taha, Walid, Sheard, Tim, & S, Tim. (2002). Tagless staged interpreters for typed languages. *Pages 218–229 of: In the international conference on functional programming (ICFP'02)*. ACM.
- Pottier, François. (2000). A versatile constraint-based type inference system. *Nordic journal of computing*, **7**(4), 312–347.
- Schurgers, Curt, Kulkarni, Gautam, & Srivastava, Mani B. (2002). Distributed on-demand address assignment in wireless sensor networks. *Ieee trans. parallel distrib. syst.*, **13**(10), 1056–1065.
- Shapiro, Jonathan, & Sridhar, Swaroop. *The BitC programming language*. <http://www.bitc-lang.org/>.
- Shi, Rui, Chen, Chiyan, & Xi, Hongwei. (2006). Distributed meta-programming. *Pages 243–248 of: GPCE '06: Proceedings of the 5th international conference on generative programming and component engineering*.
- Taha, Walid. (2004). Resource-aware programming. *Pages 38–43 of: Wu, Zhaohui, Chen, Chun, Guo, Minyi, & Bu, Jiajun (eds), ICSSS. Lecture Notes in Computer Science*, vol. 3605. Springer.
- Taha, Walid, & et. al. *MetaOCaml: A compiled, type-safe multi-stage programming language*. <http://www.metaocaml.org/>.
- Taha, Walid, & Nielsen, Michael Florentin. (2003). Environment classifiers. *POPL'03*.
- Taha, Walid, & Sheard, Tim. (1997). Multi-stage programming with explicit annotations. *Pages 203–217 of: PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation*.
- Taha, Walid, el-abidine Benaïssa, Zine, & Sheard, Tim. (1998). Multi-stage programming: Axiomatization and type safety (extended abstract). *Pages 918–929 of: In 25th international colloquium on automata, languages, and programming*. Springer-Verlag.
- Wan, Zhanyong, & Hudak, Paul. (2000). Functional reactive programming from first principles. *Pages 242–252 of: PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on programming language design and implementation*. New York, NY, USA: ACM.
- Xi, Hongwei, Chen, Chiyan, & Chen, Gang. (2003). Guarded recursive datatype construc-

tors. *Pages 224–235 of: POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on principles of programming languages.* New York, NY, USA: ACM.