Noname manuscript No.
(will be inserted by the editor)

Type-Specialized Staged Programming with Process Separation

Yu David Liu · Christian Skalka · Scott F. Smith

the date of receipt and acceptance should be inserted later

Abstract Staging is a powerful language construct that allows a program at one stage of evaluation to manipulate and specialize a program to be executed at a later stage. We propose a new staged language calculus, ⟨ML⟩, which extends the programmability of staged languages in two directions. First, ⟨ML⟩ supports *dynamic type specialization*: types can be dynamically constructed, abstracted, and passed as arguments, while preserving decidable typechecking via a System $F_{\leq}$-style semantics combined with a restricted form of λ_{ω}-style runtime type construction. With dynamic type specialization the data structure layout of a program can be optimized via staging. Second, ⟨ML⟩ works in a context where different stages of computation are executed in different process spaces, a property we term *staged process separation*. Programs at different stages can directly communicate program data in ⟨ML⟩ via a built-in serialization discipline. The language ⟨ML⟩ is endowed with a metatheory including type preservation, type safety, and decidability as demonstrated constructively by a sound type checking algorithm. While our language design is general, we are particularly interested in future applications of staging in resource-constrained and embedded systems: these systems have limited space for code and data, as well as limited CPU time, and specializing code for the particular deployment at hand can improve efficiency in all of these dimensions. The combination of dynamic type specialization and staging across processes greatly increases the utility of staged programming in these domains. We illustrate this via wireless sensor network programming examples.

Christian Skalka's work was supported by a grant from the Air Force Office of Scientific Research Young Investigator Program (AFOSR YIP).

Y.D. Liu
SUNY Binghamton
davidl@cs.binghamton.edu

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

S.F. Smith
The Johns Hopkins University
scott@cs.jhu.edu

1 Introduction

There is a rich history of explicit, principled support for dynamic program generation and execution in programming languages [56,54,15,49,6,42,41,9,55]. Such systems are said to support program *staging*, where program code is admitted as a data type, and features are provided for the generalization, composition, and specialization of code-as-data, and also to execute code-as-data.

Program staging has been effective as a means to formalize and implement features such as macros [20,30], and to increase the efficiency, programmability, and reliability of systems involving code generation, such as compiler compilers [45]. In this paper we present the language ⟨ML⟩, an extension of a core ML calculus with explicit support for program staging. The design of ⟨ML⟩ draws motivations from real-world deployment of embedded systems, in particular wireless sensor networks (WSNs). Our main contribution is to offer a flexible yet type-safe solution to achieve *dynamic type specialization*: a program at one stage can compose and specialize the types occurring in the code of the next stage. In subsequent discussion we illustrate why type specialization is important in the WSN application space. In this space there is also a need for other features not offered by previous systems, namely separation of staged runs by process boundaries and protocols for type-safe data passing between stages. Due to the novelty and complexity of these features, the exploration in this paper is foundational and carried out in an ML-style setting.

1.1 Type Specialization

The need for type specialization in staged programming is illustrated by a common use of macros, which are a simple form of staged computation [20,30]. In particular, it is common to use C conditional macros in type definitions, such as

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

Here two needs are expressed by macro users: 1) the macro stage should specialize type `T`, to be used at the C code execution stage and 2) `T` at the macro stage is dynamically constructed, here with a conditional expression. The challenging problem here is *type safety* of staged execution in the presence of type specialization: if `T` is used in the rest of the C program, can we guarantee the C program after macro expansion will not produce a type error? Existing work in staged programming either does not support rich type specialization [56,54,15,49], or does so in a manner that is not statically type safe [36].

We address the first need by supporting type genericity on staged types in ⟨ML⟩. The language allows specialization of the types of declared variables through bounded parametric polymorphism *à la* System $F_{\leq}$. As in System $F_{\leq}$, type bounds represent the static upper bound of a type parameter, while runtime instantiations of the parameter can in fact subtype the given upper bound. The $F_{\leq}$ combination of both subtyping and polymorphism is what is needed here – the supertype bound of $F_{\leq}$ quantification allows code to be written over a type which is partly generic and partly fixed: the fixed bound allows the type to be constructively used under the quantifier, and the generic aspect allows it to still be flexibly instantiated.

To address the second need, we support a weak form of *type computation*. This functionality is expressed in pure form as $\lambda_{\omega-}$ the edge of the λ cube [2] where types

depend on types, and type constructors are endowed with a semantics that is essentially a simply typed λ calculus. Other type theories such as the Calculus of Constructions [12] and Martin-Löf type theory [37] subsume the full expressivity of λ_ω, but here we propose a simpler, first-order form of type construction that is adequate for the application space of interest and yields a more manageable metatheory.

This novel combination of System $F_{\leq}$ and a restricted λ_ω (which we will informally call λ_ω^-) can be used to support type specialization for staged programming. Furthermore, leveraging these constructs yields a well-founded system that is amenable to metatheoretic analysis, and we will illustrate both type safety for and decidability (constructively) of $\langle$ML$\rangle$. In addition to practical relevance, there is a theoretical contribution here in how we show System $F_{\leq}$ and λ_ω^- can be combined into a single Turing complete language with a decidable type checking property. This is analogous to other systems which have considered leveraging other edges of the λ-cube (*e.g.* λ_P, *a.k.a.* dependent types) for practical programming [59, 10, 4, 19].

Our type specialization technique is critical for WSNs. WSNs are composed of a potentially large number of sensor nodes (so-called *motes*) of limited resources connected to one or more *hubs* – larger computers running e.g. Linux. Type specialization allows WSN hubs to dynamically refine node address sizes and other mote data structures to minimize their memory footprint. This reduces message sizes, leading to bandwidth reduction during communications [48]. The example in Sec. 7 below illustrates this point in detail.

1.2 Staging Deployment Steps with Process Separation

A major contrast of $\langle$ML$\rangle$ with existing staged languages such as MetaML [56] is the intended application space. We consciously align *stages* in $\langle$ML$\rangle$ with *steps* in the deployment cycle of a multi-tier architecture: each stage is an independent process that deploys the successive stage. Unlike macro systems which are also independent processes (the macros expand at compile-time and the program runs at run-time), we aim for a more generalized framework where the earlier stages can keep running and help monitor/stop/optimize/redeploy later stages. We believe this more general view has widespread applicability across different modes of deployment, but has yet to be popularized. In order to focus on this concept, we will hereafter term it *staged process separation*, or SPS.

We here build on observations of previous authors [53, 29, 51] that staged programing is a useful tool to aid in the construction of resource-bounded systems. These previous papers show the application of staged computation to both embedded systems programming and to custom circuit specification. As was pointed out previously [53], while resources may be highly constrained on the embedded device, the stage which constructs such resource-bounded programs need not be constrained. So, the larger the amount of computation that can be put in the first stage the more resource-friendly it will be for code deployed on embedded systems.We believe that WSN applications can be a prime beneficiary of staging features that realize this notion of process separation. 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 [26, 21, 35, 36]. The hub stage can specialize/deploy/stop/monitor/redeploy the mote stage, and each of these operations has important utility. For example, since each mote is highly

power-, memory-, and speed-limited, the specialization of mote behavior by the hub can yield crucial performance improvements.

Our ⟨ML⟩ model is related to MetaML [56], but it is fundamentally different because our SPS requirement is directly at odds with MetaML's support for *cross-stage persistence* (CSP). CSP is a MetaML feature that allows values to migrate freely between stages through standard function abstraction and application. This is highly problematic in a multi-tiered embedded system with mutable state, since sharing memory between processes/stages is not feasible – and embedded systems languages make heavy use of mutation and state. Our type system is constructed in a manner that implicitly rules out the possibility of CSP. At the same time, we do allow composition and specialization of stateful code by allowing values to be explicitly "lifted" between stages in a principled manner via a form of data serialization (*a.k.a.* marshaling). Our need for SPS is reflected is not unique, in that there are other settings where staging is used to implement deployment cycles across independent processes running on different hardware [49, 53, 29]. Although ⟨ML⟩ has a functional core and is therefore different than many production languages for embedded systems and WSNs such as nesC [21], the current exploration establishes a sound theoretical foundation that is applicable to other paradigms, and plan to study this in future work as discussed in Sec. 9.

1.3 The Structure of the Paper

The work presented here is foundational. While our ultimate goal is in fact to port these ideas to realistic languages for programming embedded systems, 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 given the novel interaction of staging with $\lambda_{\omega-}$ and System $F_{\leq}$-style featuresOur language, type theory, and metatheory are presented in Sec. 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 WSN example in Sec. 7, exploring the bandwidth reduction via address minimization mentioned above.

2 The Core Language

In this Section we define and discuss the core ⟨ML⟩ syntax and dynamic semantics. At certain points in the text we will refer to the type system, e.g. when discussing type casting, but we delay our full presentation of it until Sec. 3. The calculus introduced in this section is purely functional, but we will extend it with mutation and state in Sec. 4. We assume some familiarity with the basic concepts of staged programming; readers with no background may want to consult *e.g.* [50] for background and examples.

2.1 Core ⟨ML⟩ Syntax and Semantics

The ⟨ML⟩ language syntax is defined in Fig. 1, including *values* v, *expressions* e, *evaluation contexts* E, *types* τ, *type coercions* Δ, and *type environments* Γ. Discussion of

$$
\begin{array}{rll}
x \in V \subset \mathcal{V},\ t \in \mathcal{T} & & \\
v & ::= \mathbf{c} \mid x \mid \lambda x : \tau.e \mid \Lambda t \preccurlyeq \tau.e \mid \langle e \rangle \mid \tau & \textit{values} \\
e & ::= v \mid (\tau)e \mid e\,e \mid \mathbf{tlet}\ t \preccurlyeq \tau = e\ \mathbf{in}\ e \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \mid \mathbf{run}\ e \mid \mathbf{lift}\ e & \textit{expressions} \\
E & ::= [\,] \mid Ee \mid vE \mid \mathbf{tlet}\ t \preccurlyeq \tau = E\ \mathbf{in}\ e \mid \mathbf{let}\ x = E\ \mathbf{in}\ e \mid & \textit{evaluation contexts} \\
 & \quad (E)e \mid (v)E \mid \mathbf{run}\ E \mid \mathbf{lift}\ E & \\
\sigma, \tau & ::= t \mid \gamma \mid \mathbf{type}[\tau] \mid \langle \cdot\tau\cdot \rangle \mid \Pi t \preccurlyeq \tau.\tau \mid \exists t \preccurlyeq \tau.\tau \mid \tau \to \tau \mid \top & \textit{types} \\
\Delta & ::= \varnothing \mid \Delta; t \preccurlyeq \tau & \textit{type coercions} \\
\Gamma & ::= \varnothing \mid \Gamma; x : \tau & \textit{type environments}
\end{array}
$$

Fig. 1 ⟨ML⟩ Term and Type Syntax

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

$$
\mathbf{tlet}\ t \preccurlyeq \tau = \tau'\ \mathbf{in}\ e \to e[\tau'/t]\ \textsc{RTLet}
\qquad
(\Lambda t \preccurlyeq \tau.e)\tau' \to e[\tau'/t]\ \textsc{RApp}_{\Pi}
\qquad
\frac{v : \tau}{(\tau)v \to v}\ \textsc{RCast}
\qquad
\frac{e \to^{\star} v}{\mathbf{run}\ \langle e \rangle \to v}\ \textsc{RRun}
$$

$$
\mathbf{lift}\ v \to \langle v \rangle\ \textsc{RLift}
\qquad
\frac{e \to e'}{E[e] \to E[e']}\ \textsc{RContext}
$$

Fig. 2 ⟨ML⟩ Core Operational Semantics

expression forms directly related to type genericity, including type abstraction $\Lambda t \preccurlyeq \tau.e$, a "type let" **tlet**, type-as-terms, and type casting are discussed in Sec. 3. Our initial focus is on the three basic expression forms for staged computation. The form $\langle e \rangle$ represents the *code* e, which is treated as a first-class value. The form **run** e evaluates e to code and then runs that code, in its own process space. The form **lift** e evaluates e to a value, and turns that value into code, *i.e.* "lifts" it to a later stage. We omit the "escape" operator, realized for example as $\sim e$ in MetaML, a design choice we discuss in more detail below.

Central to the prevention of CSP 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 $\langle e \rangle[\langle e' \rangle/x] = \langle e[e'/x] \rangle$, and make $\langle e \rangle[e'/x]$ be undefined if e' is not code. This definition forces free variables in $\langle e' \rangle$ to be instantiated with code only. Term substitution is completely defined in Fig. 3.

The operational semantics of ⟨ML⟩ are then defined in Fig. 2 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; the need for separation will become more clear when we consider mutation and state in Sec. 4. The semantics of casting are predicated on a notion of typing defined in the following section.

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

$$
\begin{array}{rll}
t[\tau/t] &= \tau & \\
t'[\tau/t] &= t' & \text{if } t \neq t' \\
\gamma[\tau/t] &= \gamma & \\
\mathbf{type}[\tau'][\tau/t] &= \mathbf{type}[\tau'[\tau/t]] & \\
\langle \cdot\tau'\cdot \rangle[\tau/t] &= \langle \cdot\tau'[\tau/t]\cdot \rangle & \\
(\Pi t' \preccurlyeq \sigma.\tau')[\tau/t] &= \Pi t' \preccurlyeq \sigma[\tau/t].\tau'[\tau/t] & \text{if } t \neq t' \\
(\Pi t \preccurlyeq \sigma.\tau')[\tau/t] &= \Pi t \preccurlyeq \sigma[\tau/t].\tau' &
\end{array}
$$

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

Fig. 3 Term and Type Substitutions in ⟨ML⟩

2.1.1 User-Defined Constants

The language ⟨ML⟩ may be extended with user-defined constants **c**. Any functional constants must be accompanied by specification of their semantics via the function δ. For example, suppose it is desired to add numerical constants to the language, including 8-bit integers, 16-bit integers, and real numbers, and including arithmetic functions *abs* for taking the absolute value of integer constants and *floor* for taking the floor of real numbers. Formally, given $i^8 \in \mathbb{Z}_{2^8}$, $i^{16} \in \mathbb{Z}_{2^{16}}$, and $n \in \mathbb{R}$, we may define **c** as follows:

$$\mathbf{c} ::= i^8 \mid i^{16} \mid n \mid \mathit{abs} \mid \mathit{floor}$$

The functional constants are endowed with an appropriate interpretation as follows:

$$\delta(\mathit{abs}, i^{16}) = |i^{16}| \qquad\qquad \delta(\mathit{floor}, n) = \lfloor n \rfloor$$

Since $\mathbb{Z}_{2^8} \subseteq \mathbb{Z}_{2^{16}} \subseteq \mathbb{R}$, the function *floor* is defined on any sort of numeric constant, while *abs* is defined only on integral constants in $\mathbb{Z}_{2^{16}}$.

2.2 Discussion

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

Comparison with MetaML-style staging The MetaML bracket expression $\langle e \rangle$ and execution expression **run** e are directly reflected in ⟨ML⟩. The canonical MetaML example of a staged list membership testing function, *member*, can be written in ⟨ML⟩ as in Fig. 4(a). The gist of this example is that first-stage execution of *member* applications yields a piece of code where membership tests are inlined rather than executed in recursive calls, which is typically more efficient.

The code so produced is also specialized in that membership tests are partly instantiated. 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 test membership of the result of some value v_{sense} over 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 recursion and **if**...**then**...**else**, we can execute the program in Fig. 4(a) on the hub, a computer with fewer resource constraints. What will be deployed to individual sensor nodes is only the argument of the **run** expression, *member* $\langle v_{sense} \rangle$ $[0, 1]$, which will be evaluated on the hub to:

$$\langle v_{sense} = 0 \;||\; v_{sense} = 1 \;||\; \mathbf{false} \rangle$$

However, there is a drawback to this approach in a ⟨ML⟩ context: the expression v_{sense} must be closed, and so can't reference variables in the larger program context. Function *member*′ in Fig. 4(b) gives a different version of code specialization showing how this problem can be alleviated in ⟨ML⟩. Note how any parameter within an object-level expression must be made explicit with a λ-binding in ⟨ML⟩. This is essentially because the MetaML "escape" operator is not available in ⟨ML⟩ as we discuss more below.

Value Migration via Lift and Run The semantics of ⟨ML⟩'s substitution-based term reduction itself disallows CSP; for example the (untypeable) 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 **lift**ing a value from the meta to the object language and **run**ning object code from the meta level are sufficient and indeed function as duals. For example in Fig. 4, 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 $(x = h)$.

In this foundational presentation our object and meta-stage languages have identical syntax for simplicity, but it is likely that a production language will want to restrict the syntax allowed on motes. In such a case the **lift** function would be restricted to only support lifting of meta-code which was also typeable at the object stage.

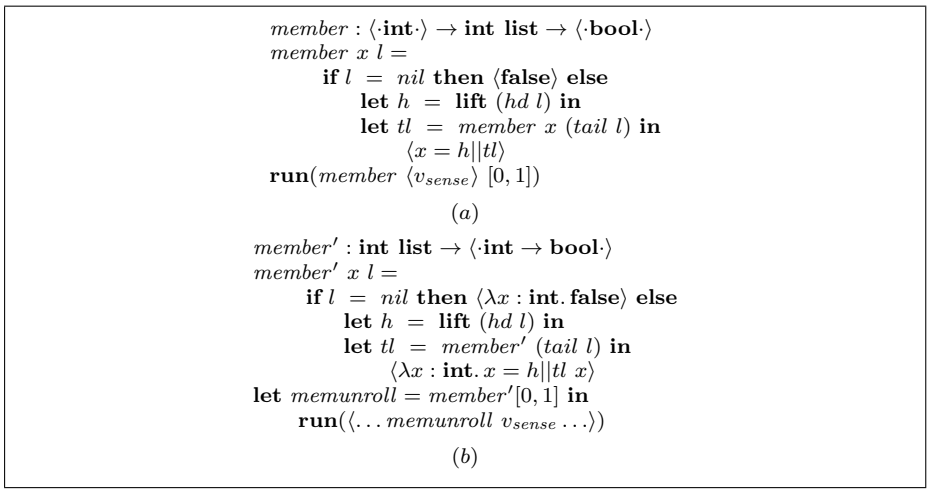

Fig. 4 ⟨ML⟩ Definitions of *member* and *member′*

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:

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

as in Fig. 4, MetaML programmers would equivalently write:

$$<\sim x = \sim h \mid\mid \sim tl>$$

This syntactic 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 of viewing such variables as being at the object level unless explicitly escaped with ~. In the ⟨ML⟩ syntax of implicitly escaping variables such as $x/h/tl$ above, the escape operator becomes less important; 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. The *member* and *member′* examples directly illustrate this point: the *member* example was taken from Section 6 of [56] and directly translated to ⟨ML⟩. In MetaML, it is in fact possible to use *member* itself inside a larger program since the code passed to the first argument may contain a free variable. Consider the following example from [56]:

$$<\lambda x.\sim(member<x> \ [1, 2, 3])>$$

The ⟨ML⟩ simulacrum of this would be:

$$(\textbf{let } m = member'[1, 2, 3] \textbf{ in } \langle \lambda x.m \ x \rangle)$$

Although MetaML escape enhances programmability, it leads to significant complexities for static type checking [42,9,55]. A canonical example is $<\lambda x.\sim(e<x>)>$. Consider the case where expression e contained code **run** $<x>$ – the type system

must statically prevent execution of open expressions such as this, but the solutions that have been applied are not elegant since it is a context-sensitive property.

The WSN examples we have studied have not revealed any expressivity constraints using the escape-free ⟨ML⟩ syntax, so we opt here to leave it out for simplicity. 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, the expression e in the above example could stash $<x>$ on the heap and allow it to escape its binding context).

3 Types and Type Specialization

In this Section we focus on the ⟨ML⟩ type system, again beginning with formalities then moving on to higher level discussion. Briefly, our goal is to support type genericity as discussed in Sec. 1.1, and to statically disallow CSP. 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. 1.1, *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 runtime 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, which supports a limited form of statically well-typed (type safe) type computation *a la* λ_ω. Note that our System $F_{\leq}$-style type abstraction $\Lambda t \preccurlyeq \tau.e$ and application forms $e\ \tau$ are too restrictive to encode an analog of the usual term encoding of **let**: syntax **tlet** $t \preccurlyeq \tau = e$ **in** e' would be analogously encoded as $(\Lambda t \preccurlyeq \tau.e')\ e$, but type application in our theory is restricted so that the argument e must in fact be a concrete type τ. Without this restriction the static type system would be undecidable, since e may not terminate. Our primitive **tlet** syntax has no such restriction and allows type variables to be bound to the results of arbitrary computations.

Further technical discussion of type computations and static typing thereof is presented in Sec. 3.3.1. We have discussed the usefulness of related forms in Sec. 1.1, and examples in Sec. 3.4 and Sec. 7 will further illustrate them. To simplify later definitions and results we take both Λ- and **tlet**-bound variables as names for their de Bruijn indices, requiring possible α-conversion of any given program to yield distinct bound type variable names at each binder.

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

$$\textsc{TopS}\ \ \Delta \vdash \tau \preccurlyeq \top \qquad \textsc{ReflS}\ \ \Delta \vdash \tau \preccurlyeq \tau$$

$$\textsc{ConstS}\ \ \frac{\gamma \preccurlyeq_{ax} \gamma'}{\Delta \vdash \gamma \preccurlyeq \gamma'} \qquad \textsc{CoerceS}\ \ \frac{\Delta(t) = \tau}{\Delta \vdash t \preccurlyeq \tau} \qquad \textsc{CodeS}\ \ \frac{\Delta \vdash \tau_1 \preccurlyeq \tau_2}{\Delta \vdash \langle\cdot\tau_1\cdot\rangle \preccurlyeq \langle\cdot\tau_2\cdot\rangle}$$

$$\textsc{TransS}\ \ \frac{\Delta \vdash \tau_1 \preccurlyeq \tau_2 \qquad \Delta \vdash \tau_2 \preccurlyeq \tau_3}{\Delta \vdash \tau_1 \preccurlyeq \tau_3} \qquad \textsc{FnS}\ \ \frac{\Delta \vdash \tau_1' \preccurlyeq \tau_1 \qquad \Delta \vdash \tau_2 \preccurlyeq \tau_2'}{\Delta \vdash \tau_1 \to \tau_2 \preccurlyeq \tau_1' \to \tau_2'} \qquad \textsc{TypeS}\ \ \frac{\Delta \vdash \tau \preccurlyeq \tau'}{\Delta \vdash \mathbf{type}[\tau] \preccurlyeq \mathbf{type}[\tau']}$$

$$\textsc{PiS}\ \ \frac{\Delta; t \preccurlyeq \tau_0 \vdash \tau \preccurlyeq \tau'}{\Delta \vdash (\Pi t \preccurlyeq \tau_0.\tau) \preccurlyeq (\Pi t \preccurlyeq \tau_0.\tau')} \qquad \textsc{ExistsS}\ \ \frac{\Delta; t \preccurlyeq \tau \vdash \tau' \preccurlyeq \tau''}{\Delta \vdash (\exists t \preccurlyeq \tau.\tau') \preccurlyeq (\exists t \preccurlyeq \tau.\tau'')}$$

Fig. 5 Subtyping Rules

forms $\tau \to \tau$ and base types γ for user-defined constants. The user-defined function *ty* maps constants **c** to their associated type γ. 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 **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 **tlet**-bound value whose type is inherently dynamic. Without use of existentials there is no possibility on having a **tlet** upper bound contain a dynamic type anywhere, so existentials are critical to make the type theory more complete. The $\exists$ types here do not have the full logical power of an existential: arbitrary types τ cannot be abstracted as "$\exists t$", because our need for $\exists$ is modest – it only serves to support proper extrusion of **tlet**-bound type variables. $\exists$ 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 $\Delta \vdash \tau \preccurlyeq \tau'$ in Fig. 5; 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, i.e. for any $t \in \boldsymbol{dom}(\Delta)$ it is not the case that $t \in \mathrm{fv}(\Delta(t))$. 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 1 A *coercion* Δ is a function from type variables to types. We write $\Delta; t \preccurlyeq \tau$ to denote the coercion that maps t to τ but agrees with Δ on all other points. We write $\Delta \oplus t \preccurlyeq \tau$ to denote $\Delta; t \preccurlyeq \tau$ where $t \in \mathrm{Dom}(\Delta)$ implies $\Delta(t) = \tau$.

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

Definition 2 Let ftv be a function returning the free type variables in a type or expression. A coercion Δ is *closed* iff for all $t \in \mathrm{Dom}(\Delta)$ it is the case that $\mathrm{ftv}(\Delta(t)) \subseteq \mathrm{Dom}(\Delta)$. A subtyping judgement $\Delta \vdash \tau_1 \preccurlyeq \tau_2$ is *closed* iff $\mathrm{ftv}(\tau_1, \tau_2) \subseteq \boldsymbol{dom}(\Delta)$.

The reader will note that constraints on Π- and $\exists$-bound type variables must be equivalent to compare these forms via the subtyping relation defined in Fig. 5. 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 constraints in this relation renders subtyping undecidable [22].

Subtyping user-defined constant types γ is accomplished via an axiomatized relation $\preccurlyeq_{ax}$, also user-defined. Note that the induced relation $\varnothing \vdash \gamma_1 \preccurlyeq \gamma_2$ is a finite lattice, which is decidable – trivially so for practical purposes, since user-defined types γ will not typically be great in number.

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 An *environment* Γ is a function from term variables to types. We write $\Gamma; x : \tau$ to denote the environment that maps x to τ but agrees with Γ on all other points. We write $\Gamma \oplus x : \tau$ to denote $\Gamma; x : \tau$ where $x \in \mathrm{Dom}(\Gamma)$ implies $\Gamma(x) = \tau$.

NB there is a distinction between "plain" environment extension, denoted ";", versus "strict" extension where a new domain element is added, denoted "$\oplus$". The same symbols are also used to denote coercion extension in the same manner as defined above. Consistency in the typing rules requires that type and term variables are not redefined at certain points, hence the need to specify strict extension.

Derivability of type judgements is defined in terms of the type derivation rules in Fig. 6. This discipline disallows CSP, 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 inductively define

$$\langle \cdot \varnothing \cdot \rangle = \varnothing$$
$$\langle \cdot \Gamma; x : \tau \cdot \rangle = \langle \cdot \Gamma \cdot \rangle; x : \langle \cdot \tau \cdot \rangle$$

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 $\textsc{App}_{\Pi}$ rule results in a type substitution. Unlike term substitution, CSP 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.

We impose sanity conditions on the structure of type judgements, ensuring that any type variables appearing in a judgement are defined, *i.e.* that the judgement is closed. This restriction ensures closure of subtyping judgements occurring in type derivations, and later will simplify the presentation of algorithmic type checking. We also require that assumed type coercions in Δ don't overlap with $\exists$-bound coercions in an ascribed type, a property reflecting our view of type variables as de Bruijn indices.

Definition 4 The pair (Δ, τ) is *well-formed* iff Δ is closed and $\mathrm{ftv}(\tau) \subseteq \mathrm{Dom}(\Delta)$ and $t \notin \mathrm{Dom}(\Delta)$ for every $\exists$-bound t in τ. The judgment $\Gamma, \Delta \vdash e : \tau$ is *well-formed* iff (Δ, τ) is, and $(\Delta, \Gamma(x))$ is well-formed for all $x \in \mathrm{Dom}(\Gamma)$, and $\mathrm{ftv}(e) \subseteq \mathrm{Dom}(\Delta)$.

Type validity is then defined as follows:

Definition 5 A type judgement $\Gamma, \Delta \vdash e : \tau$ is *valid* iff it is well-formed and derivable.

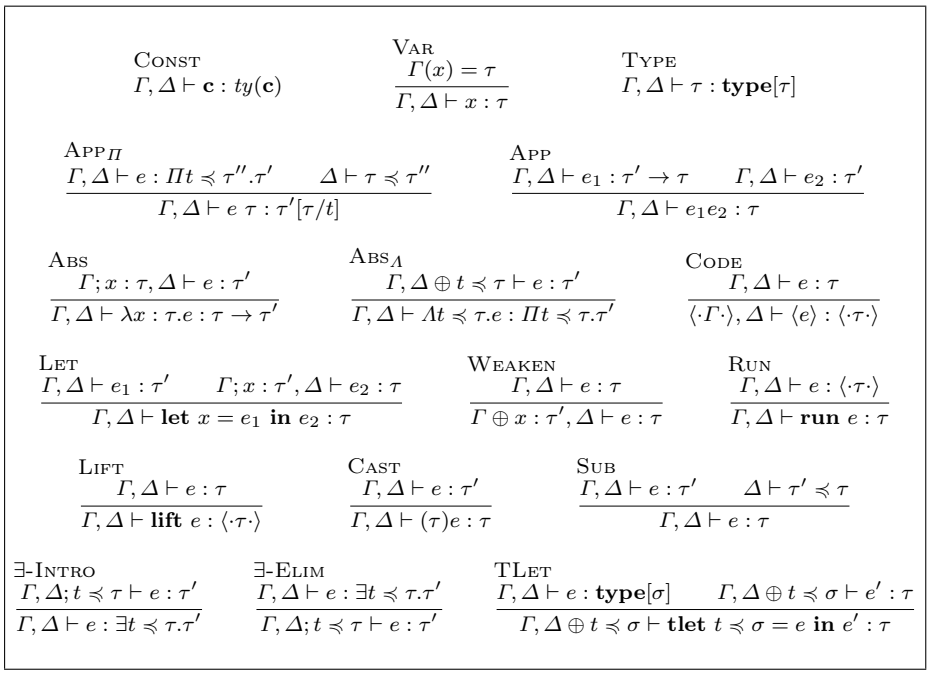

Fig. 6 Type Judgement Rules

3.3.1 On **tlet** *and* ∃ *Types*

In any **tlet** expression, a type variable is bound to a type-returning computation. But since types are first class values in ⟨ML⟩, it is not possible to statically predict the result of such a computation, hence in any **tlet** expression we allow the type variable to stand in. This means that a type variable can appear free in the **tlet** expression's type, hence its upper bound definition must be extruded in the expression's typ*ing* to close the judgement, as expressed in the TLet typing rule. For example, consider the expression **tlet** $t \preccurlyeq \sigma = e$ **in** $\lambda x : t.x$ where e is some closed expression and σ is a closed type. It can be assigned the following typing, assuming that e is appropriately typable:

$$\varnothing, t \preccurlyeq \sigma \vdash \mathbf{tlet}\ t \preccurlyeq \sigma = e\ \mathbf{in}\ \lambda x : t.x : t \to t$$

In this typing, the bound $t \preccurlyeq \sigma$ is extruded, providing a definition for the variable t occurring in the type component of the judgement. Note in general that it is not sound to replace a **tlet**-bound variable with its upper bound; in this case for example we would obtain $\sigma \to \sigma$ as the type of the expression, but $t \to t \not\preccurlyeq \sigma \to \sigma$ due to contravariance of the domain in function subtyping. Hence, use of the type t is necessary.

The typing rule ∃-Intro, on the other hand, allows extruded type variable bounds to be moved into the type itself via an ∃ binding. This is important, since in particular the upper bound of a **tlet**-bound type variable may contain a Λ-bound variable. For example, consider the following typing derivation fragment, where we assume that σ

and e are closed and e is appropriately typable:

$$\dfrac{\dfrac{\varnothing, t' \preccurlyeq \sigma; t \preccurlyeq t' \vdash \textbf{tlet } t \preccurlyeq t' = e \textbf{ in } \lambda x : t.x : t \to t}{\varnothing, t' \preccurlyeq \sigma \vdash \textbf{tlet } t \preccurlyeq t' = e \textbf{ in } \lambda x : t.x : \exists t \preccurlyeq t'.t \to t}}{\varnothing, \varnothing \vdash \Lambda t' \preccurlyeq \sigma.\textbf{tlet } t \preccurlyeq t' = e \textbf{ in } \lambda x : t.x : \Pi t' \preccurlyeq \sigma.\exists t \preccurlyeq t'.t \to t}$$

The first deduction is by $\exists$-Intro, the second by $\textsc{App}_{\Pi}$. Note that the second deduction step is not possible without the first, since t' cannot occur free in the top-level coercion; the bound $t \preccurlyeq t'$ must be moved into the type.

Indeed, 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 $\textsc{Abs}_{\Lambda}$, and always perform $\exists$-Elim just after an instance of $\textsc{App}_{\Pi}$. 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 [33], 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.

We believe that our **tlet** construct, statically typed using $\exists$ type forms and scope extrusion, is novel. In essence, it supports a well-typed first-order form of type computation. Type computation is fully embodied in λ_ω, which is subsumed e.g. by both Coq [13] and Agda [3], but those systems allow definition of higher order *type constructors* which comprise a complete λ calculus. We do not provide type constructors in our system per se; although Λ abstraction does allow an encoding of type construction, Λ application can only be performed on concrete, first order type forms.

3.3.2 Typing User-Defined Constants

To illustrate how user-defined constants may be appropriately type-axiomatized, we recall the example constants in *Sec.* 2.1.1. Appropriate base types for these constants could be:

$$\gamma ::= \textbf{int8} \mid \textbf{int16} \mid \textbf{real}$$

where the following base subtyping axioms are defined:

$$\textbf{int8} \preccurlyeq_{ax} \textbf{int16} \qquad\qquad \textbf{int16} \preccurlyeq_{ax} \textbf{real}$$

and with the type assignment function ty defined as follows:

$$ty(i^8) = \textbf{int8} \qquad \dfrac{i^{16} \notin \mathbb{Z}_{2^8}}{ty(i^{16}) = \textbf{int16}} \qquad \dfrac{n \notin \mathbb{Z}_{2^{16}}}{ty(n) = \textbf{real}} \qquad ty(abs) = \textbf{int16} \to \textbf{int16}$$

$$ty(\mathit{floor}) = \textbf{real} \to \textbf{real}$$

Given these definitions, the expression $abs(1.2)$ is not typable, whereas the expression $\mathit{floor}(2)$ is, for example, which is sound since the latter is defined whereas the latter is not.

As another realistic example, consider adding booleans expressions to the language along with a condition operator *ite*. The typing of boolean values and operators using an introduced type **bool** is straightforward. To interpret conditional expressions, we provide an appropriate semantics of *ite* to ensure full generality, in particular it comprises type abstraction:

$$\delta(ite, \text{true}) = \Lambda t \preccurlyeq \top.\lambda x : t.\lambda y : t.x \qquad \delta(ite, \text{false}) = \Lambda t \preccurlyeq \top.\lambda x : t.\lambda y : t.y$$

The appropriate type for this constant is then clear:

$$ty(ite) = \Pi t \preccurlyeq \top.\mathbf{bool} \to t \to t \to t$$

3.4 Discussion

Type Abstraction and Application for Staged Code Our running example introduced in Sec. 1.1 is that of object level code parameterized by a pre-computed address type. In ⟨ML⟩ this can be written as

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

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

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

This code will be executed on the meta stage, so that when this code is executed on sensor nodes, variable *addr* will have **uint8** type. Observe how the type parameter $addr\_t$ is used within object code in the $\ldots \lambda addr : addr\_t. \ldots$ declaration, but the actual parameter **uint8** is a type and not "code that is a type" ⟨**uint8**⟩. This highlights how our system supports CSP 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 [39, 6, 58, 9, 55]. 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 only be usable as a completely generic object.

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

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

With this bound, the type system can assume the type of *addr* is at least **uint** when e is typechecked, and use it as such. Our form of type abstraction is related to standard bounded polymorphism of System $F_{\leq}$; our bounds cannot be recursive, but we allow types to be constructed dynamically whereas System $F_{\leq}$ does not.

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 ⟨ML⟩, and can be assigned, passed around, stored in memory, compared, *etc.*

Treating types as values in ⟨ML⟩ provides programmers with a flexible way to define constructs such as the type macros discussed in Sec. 1.1 (in fact, arbitrary ⟨ML⟩ programs are allowed to define T), while 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. For example, consider the following ⟨ML⟩ program fragment:

$$\textbf{tlet } tcond \preccurlyeq \textbf{uint32} = (\textbf{if } e0 \textbf{ then uint16 else uint32}) \textbf{ in}$$
$$\textbf{let } rtt = (\Lambda addr\_t \preccurlyeq \textbf{uint32}.\langle \lambda addr : addr\_t.e_0;\ send(e_1)\rangle)\ tcond \textbf{ in}$$
$$\ldots$$

Here the **tlet** $\ldots = \ldots$ **in** expression is similar to a **let** $\ldots = \ldots$ **in** , except that it binds types. The binder **tlet** serves a critical purpose in the formalism: any type-valued expression such as the above conditional cannot directly appear in another type; only its **tlet**-ed name can. This means that ⟨ML⟩ types can only be dependent on types-as-expressions (as in λ_ω on the λ cube), not arbitrary expression forms (as in λ_P). Assuming the return type of a typical *send* function is an ACK of fixed **result_t** type, our language will type the example above as $rtt : \langle \cdot tcond \rightarrow \textbf{result\_t} \cdot \rangle$, under type constraint $tcond \preccurlyeq \textbf{uint32}$. This type can then be *existentially* bound to close the type judgement, giving $rtt : \exists tcond \preccurlyeq \textbf{uint32}.\langle \cdot tcond \rightarrow \textbf{result\_t} \cdot \rangle$.

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

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

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

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

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

Casting To "close the loop" on runtime-dependent types as defined above we need to find a way to 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 *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 [1,46] 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 *rtt* we could write:

$$rtt((tcond)\ 5)$$

which will typecast 5 to either **uint16** or **uint32** as appropriate at run-time. Typecasts are fundamentally dynamic and may fail at runtime, and such failures are not counted as typing failures in the type safety result, Theorem 2 below.

4 Records, State, and Serialization

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. Mutable features are essential to consider here because they are prevalent in languages for programming embedded systems such as nesC. Furthermore, state poses some unique technical challenges in the presence of ⟨ML⟩-style staging where we assume that distinct stages represent distinct process spaces.

$$
\begin{array}{rcl}
s & ::= & v \mid s; e \\
v & ::= & \ldots \mid \{\ell_1 = v_1; \ldots; \ell_n = v_n\} \mid () \mid x \\
e & ::= & \ldots \mid \{\ell_1 = e_1; \ldots; \ell_n = e_n\} \mid e.\ell \mid \mathbf{ref}\ e \mid e := e \mid !e \mid s \\
\tau & ::= & \ldots \mid \{\ell_1 : \tau_1; \ldots; \ell_n : \tau_n\} \mid \mathbf{ref}\ \tau \mid \mathbf{unit} \\
E & ::= & \ldots \mid \{\ell_1 = v_1; \ldots; \ell_n = v_n; \ell = E; \ell'_1 = e_1; \ldots; \ell'_m = e_m\} \\
 & & \mid E.\ell \mid \mathbf{ref}\ E \mid E := e \mid v := E \mid !E \\
 & & \\
D & ::= & [\,] \mid \mathbf{let}\ z = \mathbf{ref}\ v\ \mathbf{in}\ D \\
m & ::= & () \mid m; z := v \\
h & ::= & D[m]
\end{array}
$$

Fig. 7 Syntax for Records, Mutation, and Syntactic Stores

$$
\begin{array}{rcll}
\boldsymbol{dom}(s) & = & \varnothing & \\
\boldsymbol{dom}(\mathbf{let}\ z = \mathbf{ref}\ v\ \mathbf{in}\ h) & = & \{z\} \cup \boldsymbol{dom}(h) & \\
 & & & \\
\boldsymbol{lkp}\ z\ (\mathbf{let}\ z' = \mathbf{ref}\ v\ \mathbf{in}\ h) & = & \boldsymbol{lkp}\ z\ h & \text{if } z \neq z' \\
\boldsymbol{lkp}\ z\ (\mathbf{let}\ z = \mathbf{ref}\ v\ \mathbf{in}\ D[m]) & = & \boldsymbol{lkp'}\ z\ v\ m & \\
 & & & \\
\boldsymbol{lkp'}\ z\ v\ \varnothing & = & v & \\
\boldsymbol{lkp'}\ z\ v\ (m; z := v') & = & v' & \\
\boldsymbol{lkp'}\ z\ v\ (m; z' := v') & = & \boldsymbol{lkp'}\ z\ v\ m & z \neq z'
\end{array}
$$

Fig. 8 Auxiliary Functions for Store Lookup

4.1 Language Model

Since we aim to focus on foundational issues, our language model is intended to capture essential programming features as simply as possible. Previous work has demonstrated that the addition of state and records to a core functional calculus allows expression of a wide variety of side-effecting language idioms [17] , so we begin there. There remains the question of how to model state, where again we are guided by simplicity and generality. The *syntactic store* model uses a restricted subset of pure term syntax to denote a mutable store, and has been shown to foundationally capture many mutable programming idioms [27]. For our purposes, this technique allows clear and succinct expression of serialization, and the model is no less expressive than e.g. models of the store as a partial function from references to values, as illustrated by the presence of lookup and update operations and a domain check. These simple definitions allow us to focus on the fundamentals of how values move between stages.

4.2 Mutable ⟨ML⟩ Syntax and Semantics

In Fig. 7 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 [27]. Syntactic stores may be interpreted as a mapping from variables to values via the ***dom*** and ***lkp*** functions defined in Fig. 8.

$$
\begin{array}{rcll}
\textbf{\textit{project}}\ D[m]\ V & = & \textbf{\textit{project}}\ D\ V\ [\textbf{\textit{project}}\ m\ V] & \\
 & & & \\
\textbf{\textit{project}}\ [\,]\ V & = & [\,] & \\
\textbf{\textit{project}}\ (\textbf{let}\ z = \textbf{ref}\ v\ \textbf{in}\ D)\ V & = & \textbf{\textit{project}}\ D\ V & \text{if } z \notin V \\
\textbf{\textit{project}}\ (\textbf{let}\ z = \textbf{ref}\ v\ \textbf{in}\ D)\ V & = & (\textbf{let}\ x = \textbf{ref}\ v\ \textbf{in}\ (\textbf{\textit{project}}\ D\ V)) & \text{if } z \in V \\
 & & & \\
\textbf{\textit{project}}\ \varnothing\ V & = & \varnothing & \\
\textbf{\textit{project}}\ (m; z := v)\ V & = & \textbf{\textit{project}}\ m\ V & \text{if } z \notin V \\
\textbf{\textit{project}}\ (m; z := v)\ V & = & \textbf{\textit{project}}\ m\ V; z := v & \text{if } z \in V
\end{array}
$$

Fig. 9 Projecting a Sub-Store

$$
\textsc{RRun}\ \frac{(e, \varnothing) \rightarrow^{\star} (v, h')}{(\textbf{run}\ \langle e \rangle, h) \rightarrow (\textbf{\textit{serialize}}\ v\ h', h)}
\qquad
\textsc{RRef}\ \frac{z \notin \textbf{\textit{dom}}(D[m])}{(\textbf{ref}\ v, D[m]) \rightarrow (z, D[\textbf{let}\ z = \textbf{ref}\ v\ \textbf{in}\ m])}
$$

$$
\textsc{RDeref}\ (!z, h) \rightarrow (\textbf{\textit{lkp}}\ z\ h, h)
\qquad
\textsc{RAssign}\ \frac{z \in \textbf{\textit{dom}}(D[m])}{(z := v, D[m]) \rightarrow ((), D[m; z := v])}
\qquad
\textsc{RSeq}\ (v; e, h) \rightarrow (e, h)
$$

$$
\textsc{RLift}\ (\textbf{lift}\ v, h) \rightarrow (\langle \textbf{\textit{serialize}}\ v\ h \rangle, h)
\qquad
\textsc{RContext}\ \frac{(e, h) \rightarrow (e', h')}{(E[e], h) \rightarrow (E[e'], h')}
$$

Fig. 10 Semantics of ⟨ML⟩ Mutable Features

We write $\textbf{\textit{dom}}(h)$ to denote the domain of a store h, and $\textbf{\textit{lkp}}\ z\ h$ 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 results in a closed expression (a fact proved in Lemma 14 below). Formally:

Definition 6 *Serialization* of a value v given a store h is defined via the following function:

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

where ***project*** is defined in Fig. 9 and $\textbf{\textit{reachable}}\ v\ h = V$ iff V contains all store locations reachable from v in h. (Note that $D[m; v]$ is not a syntactic store in the above, we are using the fact that syntactic stores are defined as a subset of program syntax to form an expression.)

Now, we can define the operational semantics via a small-step relation $\rightarrow$ on *closed* configurations (e, h), where (e, h) is closed iff $\text{fv}(e) \subseteq \textbf{\textit{dom}}(h)$. 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. 10. 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.

$$\textsc{RecS}\quad \frac{\Delta \vdash \tau_1 \preccurlyeq \tau_1' \ldots \qquad \Delta \vdash \tau_n \preccurlyeq \tau_n'}{\Delta \vdash \{\ell_1 : \tau_1; \ldots; \ell_n : \tau_n\} \preccurlyeq \{\ell_1 : \tau_1'; \ldots; \ell_n : \tau_n'; \ldots; \ell_{n+m} : \tau_{n+m}\}}$$

$$\textsc{RefS}\quad \frac{\Delta \vdash \tau \preccurlyeq \tau' \qquad \Delta \vdash \tau' \preccurlyeq \tau}{\Delta \vdash \mathbf{ref}\ \tau \preccurlyeq \mathbf{ref}\ \tau'}$$

Fig. 11 Additional Record and State Subtyping Rules

$$\textsc{Rec}\quad \frac{\Gamma, \Delta \vdash e_1 : \tau_1 \qquad \ldots \Gamma, \Delta \vdash e_n : \tau_n}{\Gamma, \Delta \vdash \{\ell_1 = e_1; \ldots; \ell_n = e_n\} : \{\ell_1 : \tau_1; \ldots; \ell_n : \tau_n\}} \qquad \textsc{Ref}\quad \frac{\Gamma, \Delta \vdash e : \tau}{\Gamma, \Delta \vdash \mathbf{ref}\ e : \mathbf{ref}\ \tau}$$

$$\textsc{Set}\quad \frac{\Gamma, \Delta \vdash e : \mathbf{ref}\ \tau \qquad \Gamma, \Delta \vdash e' : \tau}{\Gamma, \Delta \vdash e := e' : \mathbf{unit}} \qquad \textsc{Get}\quad \frac{\Gamma, \Delta \vdash e : \mathbf{ref}\ \tau}{\Gamma, \Delta \vdash !e : \tau} \qquad \textsc{Unit}\quad \Gamma, \Delta \vdash () : \mathbf{unit}$$

$$\textsc{Seq}\quad \frac{\Gamma, \Delta \vdash e_1 : \tau' \qquad \Gamma, \Delta \vdash e_2 : \tau}{\Gamma, \Delta \vdash e_1; e_2 : \tau}$$

Fig. 12 Additional Record and State Type Judgement Rules

The subtyping and typing rules for records and references are standard, and are given in Fig. 11 and Fig. 12 as extensions to Fig. 5 and Fig. 6, respectively.

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 $\langle \text{ML} \rangle$.

5.1 Type Preservation in **tlet** Reduction

Our argument for type safety employs a mostly standard subject reduction aka type preservation strategy. However, the argument is complicated by the **tlet** expression form reduction case, since the type variable used to "stand in" for the **tlet**-defined type value is instantiated– in both the reduced term and its type. For example, suppose we have a reduction:

$$(\mathbf{tlet}\ t \preccurlyeq \top = \mathbf{unit}\ \mathbf{in}\ \lambda x : t.x) \to (\lambda x : \mathbf{unit}.x)$$

The redex e in this example can be typed as $\varnothing, \varnothing \vdash e : \exists t \preccurlyeq \top.t \to t$, or as $\varnothing, t \preccurlyeq \top \vdash e : t \to t$ whereas the reduced term can be assigned the more precise type $\varnothing, \varnothing \vdash e : \mathbf{unit} \to \mathbf{unit}$. The general form of this instantiation is still more complicated since reduction can take place within evaluation contexts. Also, $\exists$-Elim and $\exists$-Intro

are not syntax-directed, and though closure of typings is required, significant type bound "shuffling" can occur in perverse examples. For example, consider the expression e defined as:

$$\textbf{tlet } t_1 \preccurlyeq \top = \textbf{unit in tlet } t_2 \preccurlyeq \top = \textbf{unit in } \lambda x : t_1.x$$

This expression has the following valid typings:

$$\varnothing, \varnothing \vdash e : \exists t_1 \preccurlyeq \top.\exists t_2 \preccurlyeq \top.t_1 \to \textbf{unit} \qquad \varnothing, \varnothing \vdash e : \exists t_2 \preccurlyeq \top.\exists t_1 \preccurlyeq \top.t_1 \to \textbf{unit}$$

$$\varnothing, t_2 \preccurlyeq \top \vdash e : \exists t_1 \preccurlyeq \top.t_1 \to \textbf{unit} \qquad \varnothing, t_1 \preccurlyeq \top \vdash e : \exists t_2 \preccurlyeq \top.t_1 \to \textbf{unit}$$

$$\varnothing, t_1 \preccurlyeq \top; t_2 \preccurlyeq \top \vdash e : t_1 \to \textbf{unit}$$

This example illustrates that type instantiation via **tlet** unrolling and non-syntax directed rules must be treated delicately when proving type preservation. Our approach is to define a general form of instantiation, and show that typings are preserved modulo this notion of instantiation during reduction.

We begin by introducing syntactic sugar to more easily discuss types with multiple $\exists$ bindings; *NB* how the syntax of Δ has a different interpretation in this context (that is, not as a function, as is the case when Δ appears "bare" in judgements):

Definition 7 The syntactic sugaring $\exists\Delta.\tau$ is defined inductively as follows:

$$\exists\varnothing.\tau \triangleq \tau$$
$$\exists\Delta; t \preccurlyeq \sigma.\tau \triangleq \exists\Delta.\exists t \preccurlyeq \sigma.\tau$$

We then formalize an appropriate notion of instantiation as follows. First a small step instantiation relation $\trianglelefteq$ is defined, the transitive closure of which obtains a multi-step instantiation relation $\leq$. We let $(\Delta, \tau)[\sigma/t]$ denote $(\Delta[\sigma/t], \tau[\sigma/t])$ in the following.

Definition 8 The one-step instantiation relation $\trianglelefteq$ is defined via the following rules:

$\trianglelefteq$-Ref
$$(\Delta, \tau) \trianglelefteq (\Delta, \tau)$$

$\trianglelefteq$-Coerce
$$\frac{\Delta' \vdash \sigma' \preccurlyeq \sigma \qquad \Delta' \subseteq \Delta}{(\Delta, \tau)[\sigma'/t] \trianglelefteq (\Delta; t \preccurlyeq \sigma, \tau)}$$

$\trianglelefteq$-Exists
$$\frac{\Delta \vdash \sigma' \preccurlyeq \sigma}{(\Delta, (\exists\Delta'.\tau)[\sigma'/t]) \trianglelefteq (\Delta, \exists\Delta'; t \preccurlyeq \sigma.\tau)}$$

Then we define $\leq$ as the transitive closure of $\trianglelefteq$, and we say that (Δ', τ') *is an instance of* (Δ, τ) iff $(\Delta', \tau') \leq (\Delta, \tau)$.

Given this definition, and returning to our previous example, observe that:

$$(\varnothing, \textbf{unit} \to \textbf{unit}) \leq (t \preccurlyeq \top, t \to t) \qquad (\varnothing, \textbf{unit} \to \textbf{unit}) \leq (\varnothing, \exists t \preccurlyeq \top.t \to t)$$

We now need to prove auxiliary Lemmas relating $\trianglelefteq$ and non-syntax directed rules for proving type preservation. We begin by stating two obvious properties of subtyping:

Lemma 1 $\Delta \vdash \exists t \preccurlyeq \sigma.\tau_1 \preccurlyeq \exists t \preccurlyeq \sigma.\tau_2$ *iff* $\Delta; t \preccurlyeq \sigma \vdash \tau_1 \preccurlyeq \tau_2$.

Lemma 2 *If* $\Delta \vdash \tau_1 \preccurlyeq \tau_2$ *then* $\Delta \oplus t \preccurlyeq \sigma \vdash \tau_1 \preccurlyeq \tau_2$.

Now we demonstrate that instantiation of the appropriate form preserves subtyping.

Lemma 3 *If* $\Delta \oplus t \preccurlyeq \sigma \vdash \tau \preccurlyeq \tau'$ *and* $\Delta' \vdash \sigma' \preccurlyeq \sigma$ *for some* $\Delta' \subseteq \Delta$ *then* $\Delta[\sigma'/t] \vdash \tau[\sigma'/t] \preccurlyeq \tau'[\sigma'/t]$.

Proof By induction on the derivation of $\Delta \oplus t \preccurlyeq \sigma \vdash \tau \preccurlyeq \tau'$ and case analysis on the last step. Observe that Δ' must be closed by sanity conditions on coercions.
Case CoerceS. In this case we have that $\tau = t'$ for some t', and $(\Delta \oplus t \preccurlyeq \sigma)(t') = \tau'$. Suppose on the one hand that $t = t'$. Then $\tau' = \sigma$, and $\Delta' \vdash \sigma' \preccurlyeq \sigma$ is equivalent to $\Delta' \vdash \tau[\sigma'/t] \preccurlyeq \tau'[\sigma'/t]$ since t cannot occur in σ, so the Lemma follows by Lemma 2 since $\Delta'[\sigma'/t] = \Delta'$. Suppose on the other hand that $t \neq t'$, thus $\tau' = \Delta(t')$. This $t'[\sigma'/t] = t'$, and $\Delta[\sigma'/t] \vdash t' \preccurlyeq \tau'[\sigma'/t]$ by CoerceS.

The other cases follow in a relatively straightforward manner, with the assistance of the induction hypothesis in non base cases. □

In many cases we will want to apply this result in a context where the coercion Δ' is closed, so we state the following Corollary for convenience.

Corollary 1 *If* $\Delta \oplus t \preccurlyeq \sigma \vdash \tau \preccurlyeq \tau'$ *and* $\Delta \vdash \sigma' \preccurlyeq \sigma$ *then* $\Delta \vdash \tau[\sigma'/t] \preccurlyeq \tau'[\sigma'/t]$.

Now we can demonstrate an auxiliary Lemma for the Sub case of type preservation.

Lemma 4 *If* $(\Delta_1, \tau_1) \trianglelefteq (\Delta_2, \tau_2)$ *and* $\Delta_2 \vdash \tau_2 \preccurlyeq \tau_2'$, *then there exists* τ_1' *such that* $\Delta_2 \vdash \tau_1 \preccurlyeq \tau_1'$ *and* $(\Delta_1, \tau_1') \trianglelefteq (\Delta_2, \tau_2')$.

Proof Proceed by considering cases as formulated in the definition of $\trianglelefteq$. Case $\trianglelefteq$-Ref is immediate. Case $\trianglelefteq$-Coerce follows by definition and Lemma 3. In case $\trianglelefteq$-Exists, we have that $\Delta_1 = \Delta_2$ and $\tau_2 = \exists \Delta; t \preccurlyeq \sigma.\tau$ and $\tau_1 = (\exists \Delta.\tau)[\sigma'/t]$ where $\Delta \vdash \sigma' \preccurlyeq \sigma$. But since we assume that $(\Delta_1, \exists \Delta; t \preccurlyeq \sigma.\tau)$ is well-formed we have that $\Delta_1[\sigma'/t] = \Delta_1$ by Definition 4. The result follows by Lemma 1 and Corollary 1. □

We may also state the following Lemma for proving the ∃-Elim case of type preservation. Note that conditions (iii) and (iv) arise due to possible type bound reshufflings exemplified at this beginning of this subsection; they will apply when the ∃-Elim rule instance manipulates a type variable other than the one instantiated by a **tlet**-unrolling. The result follows by definition of $\trianglelefteq$.

Lemma 5 *Given* $(\Delta', \tau') \trianglelefteq (\Delta, \exists t \preccurlyeq \sigma.\tau)$, *exactly one of the following conditions holds:*

(i) $\Delta' = \Delta$ *and* $\tau' = \exists t \preccurlyeq \sigma.\tau$
(ii) $\Delta' = \Delta$ *and* $\tau' = \tau[\sigma'/t]$, *where* $\Delta \vdash \sigma' \preccurlyeq \sigma$
(iii) $\Delta = \Delta'; t' \preccurlyeq \sigma'$ *and* $\tau' = (\exists t \preccurlyeq \sigma.\tau)[\sigma''/t']$, *where* $(\Delta'; t \preccurlyeq \sigma, \tau)[\sigma''/t'] \trianglelefteq (\Delta; t \preccurlyeq \sigma, \tau)$
(iv) $\Delta' = \Delta$ *and* $\tau = \exists \Delta''; t' \preccurlyeq \sigma'.\tau''$ *and* $\tau' = (\exists t \preccurlyeq \sigma.\exists \Delta''.\tau'')[\sigma''/t']$, *where* $(\Delta'; t \preccurlyeq \sigma, (\exists \Delta''.\tau'')[\sigma''/t']) \trianglelefteq (\Delta; t \preccurlyeq \sigma, \tau)$

A similar result can be stated to apply to the ∃-Intro case of type preservation.

5.2 Substitution and Contextual Lemmas

Both type and term substitution can occur during evaluation, and reduction in evaluation contexts must take into account term and type substructure. Here we provide relevant Lemmas. 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 6 (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 $\tau = \Pi t \preccurlyeq \sigma . \tau'$ for some t, σ, τ' then $v = \Lambda t \preccurlyeq \sigma . e$ for some e.*
4. *if $\tau = \tau_0 \to \tau_0'$, then $e = \lambda x : \tau' . e'$ for some x, τ', e'.*

Next, a term substitution Lemma will apply to the β reduction case of type preservation, following standard tactics. We present here a novel case of term substitution that is central to our system design, where code is substituted into code. The result illustrates how our type system preserves typings in this latter case.

Lemma 7 (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[1] of Code we have:

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

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

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

But $\langle \cdot \Gamma' \cdot \rangle, \Delta \vdash v : \langle \cdot \tau' \cdot \rangle$ by assumption, so by Lemma 6 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. □

Now, because types are treated as first-class values for **tlet** constructs and type abstractions, in type preservation we need to apply a type substitution lemma that is similar to term substitution. Note that the type substitution result itself requires that instantiated type variables must not be **tlet**-bound within the typed term, as such a condition falsifies the result due to possible extrusion of such variables. For example, the judgement (1) is valid, whereas (2) is not derivable:

$$\varnothing, t \preccurlyeq \mathbf{uint} \vdash \mathbf{tlet}\ t \preccurlyeq \mathbf{uint} = \mathbf{uint}\ \mathbf{in}\ t : t \tag{1}$$

$$\varnothing, \varnothing \nvdash \mathbf{tlet}\ t \preccurlyeq \mathbf{uint} = \mathbf{uint}\ \mathbf{in}\ t : \mathbf{uint} \tag{2}$$

This requirement is naturally met in the context of application within the type preservation argument.

Lemma 8 (Type Substitution) *Given that t is not* **tlet***-bound in e, if $\Gamma, \Delta \oplus t \preccurlyeq \tau_0' \vdash e : \tau_0$ and $\Delta \vdash \tau_1 \preccurlyeq \tau_0'$, then $\Gamma, \Delta \vdash e[\tau_1/t] : \tau_0[\tau_1/t]$.*

[1] In this proof and later, when given a judgement obtained as an instance of a particular derivation rule, we may assert validity of the precedents of that derivation rule instance in the standard manner; we refer to this as an *inversion* of the known derivation rule instance.

Proof By induction on the derivation of $\Gamma, \Delta \oplus t \preccurlyeq \tau_0' \vdash e : \tau_0$ and case analysis on the last step.
Case Sub. In this case the last derivation step is of the form:

$$\frac{\Gamma, \Delta \oplus t \preccurlyeq \tau_0' \vdash e : \tau \qquad \Delta \oplus t \preccurlyeq \tau_0' \vdash \tau \preccurlyeq \tau_0}{\Gamma, \Delta \oplus t \preccurlyeq \tau_0' \vdash e : \tau_0}$$

By assumptions of this Lemma and by Corollary 1 we have that $\Delta \vdash \tau[\tau_1/t] \preccurlyeq \tau_0[\tau_1/t]$, and by the induction hypothesis the judgement $\Gamma, \Delta \vdash e[\tau_1/t] : \tau[\tau_1/t]$ is derivable. Hence we may construct:

$$\frac{\Gamma, \Delta \vdash e[\tau_1/t] : \tau[\tau_1/t] \qquad \Delta \vdash \tau[\tau_1/t] \preccurlyeq \tau_0[\tau_1/t]}{\Gamma, \Delta \vdash e[\tau_1/t] : \tau_0[\tau_1/t]}$$

which was to be proven.
Case Abs_Λ. In this case $\tau_0 = \Pi t' \preccurlyeq \sigma.\tau$ and $e = \Lambda t' \preccurlyeq \sigma.e'$ and the last derivation step is of the form:

$$\frac{\Gamma, \Delta \oplus t \preccurlyeq \tau_0' \oplus t' \preccurlyeq \sigma \vdash e' : \tau}{\Gamma, \Delta \oplus t \preccurlyeq \tau_0' \vdash \Lambda t' \preccurlyeq \sigma.e' : \Pi t' \preccurlyeq \sigma.\tau}$$

Now, $\Delta \vdash \tau_1 \preccurlyeq \tau_0'$ by assumption, and since Δ is closed and $t' \notin \text{Dom}(\Delta)$ therefore t' does not occur in Δ, so it is easy to show that $\Delta \oplus t' \preccurlyeq \sigma \vdash \tau_1 \preccurlyeq \tau_0'$. Hence the induction hypothesis yields derivability of $\Gamma, \Delta \oplus t' \preccurlyeq \sigma \vdash e'[\tau_1/t] : \tau[\tau_1/t]$, so we may reconstruct:

$$\frac{\Gamma, \Delta \oplus t' \preccurlyeq \sigma \vdash e'[\tau_1/t] : \tau[\tau_1/t]}{\Gamma, \Delta \vdash \Lambda t' \preccurlyeq \sigma.e'[\tau_1/t] : \Pi t' \preccurlyeq \sigma.\tau[\tau_1/t]}$$

where the consequent is equivalent to $\Gamma, \Delta \vdash e[\tau_1/t] : \tau_0[\tau_1/t]$, which was to be proven.

The remaining cases follow in a similar fashion via the induction hypothesis. □

Since reduction may take place within an evaluation context, we provide the following lemmas to "do the dirty work" in proving type preservation in this case. These Lemmas will allow us to apply the result obtained for pure redices via induction. Proof sketches follow the statement of each.

Lemma 9 *If $\Gamma, \Delta \vdash E[e] : \tau$ is valid, its derivation contains a subderivation with consequent $\Gamma, \Delta \vdash e : \tau'$ for some τ'.*

The preceding Lemma follows in a straighforward manner by induction on the form of E. In the base case, where $E = [\,]$, wherein e is shown to be typable, a necessary condition of typability of $E[e]$. Inductive cases never need to modify Γ due to the possible forms of E, so the result follows.

Lemma 10 *If a derivation of $\Gamma, \Delta \vdash E[e] : \tau_0$ contains a subderivation with consequent $\Gamma, \Delta \vdash e : \tau_1$, and $\Gamma', \Delta' \vdash e' : \tau_1'$ is valid where Γ' extends[2] Γ and $(\Delta', \tau_1') \trianglelefteq (\Delta, \tau_1)$, then $\Gamma', \Delta' \vdash E[e'] : \tau_0'$ where $(\Delta', \tau_0') \trianglelefteq (\Delta, \tau_0)$.*

The preceding Lemma follows by induction on the form of E. In the base case, where $E = [\,]$, the result is easily obtained via assumptions, Lemma 8, and Weaken. In the inductive cases, the result follows since the induction hypothesis ensures the desired property holds essentially because τ_1' replaces occurrences of τ_1 in τ_0 to yield τ_0'.

[2] In general, a partial function f *extends* a partial function g iff $f(x) = g(x)$ for all $x \in \text{Dom}(g)$.

5.3 Configuration Typings

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

Definition 9 (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 consequence 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.

Lemma 11 *If $(E[\langle e \rangle], D[m])$ has a valid typing then $\langle e \rangle$ is closed.*

In proving type preservation we will typically want to focus on the term component of configurations, but within a typing environment dictated by variables defined in the store. To this end we introduce the notion of coverage, and provide a couple of lemmas that will allow moving between configuration and expression typing judgements.

Definition 10 We say that Γ *covers* $D[m]$ iff $\varnothing, \Delta \vdash D[m] : \tau$ is valid and contains a subderivation with consequent $\Gamma, \Delta \vdash m : \tau$.

Lemma 12 *If $(e, D[m]) : \tau \circ \Delta$ is valid then it contains a subderivation with consequent $\Gamma, \Delta \vdash e : \tau$ for some Γ that covers $D[m]$.*

Lemma 13 *If $\Gamma, \Delta \vdash e : \tau$ is valid and Γ covers $D[m]$ then $(e, D[m]) : \tau \circ \Delta$ is also valid.*

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

5.4 Failure Semantics

Type preservation is the core of our type safety argument, but there remain some subtleties in the statement of type safety. In particular, it is not satisfactory to equate the class of irreducible non-value expression forms with semantically ill-formed expressions as in the usual formulation of "stuck" expressions. There are two problematic cases; the one where an illegal typecast is attempted, and the one where code is run and diverges in its separate process space. In the former case there is a run-time check in place to catch the error, and the latter case is just a form of divergence. Neither case should be included in the class of semantically ill-formed expressions.

To address this, we introduce an untyped value **fail**, along with a failure semantics in Fig. 13. This failure semantics clearly delineates the class of semantically ill-formed expressions and allows a succinct statement of type safety.

Now, we need to show that semantically ill-formed expressions are untypable. A basic sanity condition requires that user-defined constants are type-safe:

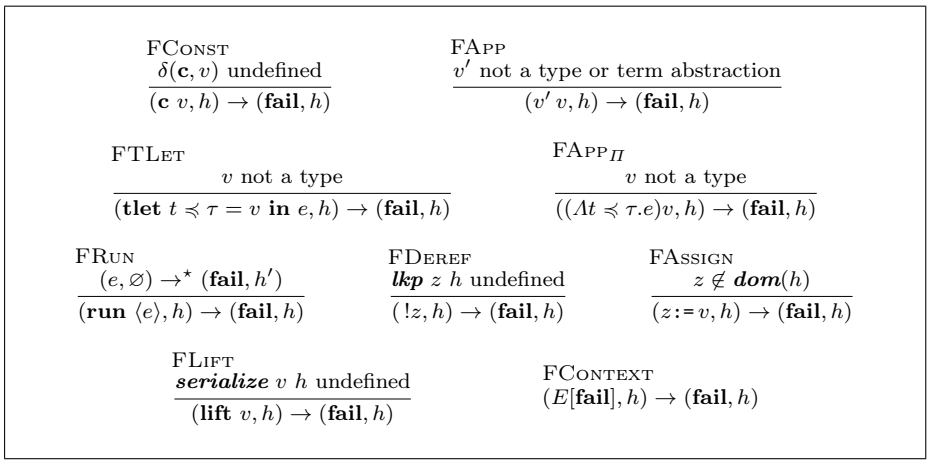

Fig. 13 ⟨ML⟩ Failure Semantics

Definition 11 We require that δ is *typable* in the following sense: if $(\mathbf{c}\ v, h) : \tau \circ \Delta$ is valid, then $\delta(\mathbf{c}, v)$ is defined.

Basic properties of typing establish that typable side-effecting operations are well-formed, in that functions underlying their semantics are defined. To guarantee the following result, we hereafter require that variables used as references are distinct from variables used as value identifiers.

Lemma 15 *Each of the following conditions hold:*

1. *If* $(v, h) : \tau \circ \Delta$ *is valid then* $(\boldsymbol{serialize}\ v\ h)$ *is defined.*
2. *If* $(!z, h) : \tau \circ \Delta$ *is valid then* $(\boldsymbol{lkp}\ z\ h)$ *is defined.*
3. *If* $(z := v, h) : \tau \circ \Delta$ *is valid then* $z \in \boldsymbol{dom}(h)$.

The preceding results allow us to demonstrate that semantically ill-formed expressions are untypable, except for the FRun case that needs an induction to be addressed. This will by done in the proof of Type Preservation.

Lemma 16 *If* $(e_0, h_0) \to (\mathbf{fail}, h_1)$ *for some* h_1 *by a rule other than* FRun*, then* (e_0, h_0) *is untypable.*

Proof The proof proceeds by case analysis on the failure semantics of Fig. 13. Case FConst follows by Definition 11. Cases FApp, FTLet, and FApp$_\Pi$ follow by Lemma 6. Cases FDeref, FAssign, and FLift follow by Lemma 15. Case FContext follows since **fail** is untypable, whereas all subexpressions of typable expressions are necessarily typable. □

5.5 Typings and Structural Induction

Not all typing rules are syntax-directed, but structural inductions in type preservation require typings of subterms to be known in order for induction hypotheses to apply. Thus, when something is known about the structure of typed terms, the following Lemmas allows judgements to be deconstructed in a syntax directed manner. In particular, we first show how to manipulate non-syntax directed rules wlog.

Lemma 17 *Each of the following properties hold:*

1. *Any* Sub *application sequence (including one of length 0) can be expressed in a single application of* Sub.
2. *Consecutive application of* ∃-Intro *followed by* ∃-Elim *can be suppressed.*
3. *Consecutive applications of* ∃-Elim *and* Sub *(in either order) can be switched.*
4. *Consecutive applications of* ∃-Intro *and* Sub *(in either order) can be switched.*

Proof Property 1 follows by reflexivity and transitivity of $\preccurlyeq$. Property 2 follows immediately by definition of typing. To prove property 3, we note that in general $\Delta; t \preccurlyeq \sigma \vdash \tau_1 \preccurlyeq \tau_2$ iff $\Delta \vdash (\exists t \preccurlyeq \sigma.\tau_1) \preccurlyeq (\exists t \preccurlyeq \sigma.\tau_2)$, and $\Delta \vdash (\exists t_1 \preccurlyeq \tau_1.\tau) \preccurlyeq (\exists t_1 \preccurlyeq \tau_2.\tau')$ implies that $t_1 = t_2$ and $\tau_1 = \tau_2$, so the following proof fragments are equivalent and illustrate the desired property in general:

$$\dfrac{\dfrac{\Gamma, \Delta; t \preccurlyeq \sigma \vdash e : \tau}{\Gamma, \Delta \vdash e : \exists t \preccurlyeq \sigma.\tau} \qquad \Delta \vdash \exists t \preccurlyeq \sigma.\tau \preccurlyeq \exists t \preccurlyeq \sigma.\tau'}{\Gamma, \Delta \vdash e : \exists t \preccurlyeq \sigma.\tau'}$$

and:

$$\dfrac{\dfrac{\Gamma, \Delta; t \preccurlyeq \sigma \vdash e : \tau \qquad \Delta; t \preccurlyeq \sigma \vdash \tau \preccurlyeq \tau'}{\Gamma, \Delta; t \preccurlyeq \sigma \vdash e : \tau'}}{\Gamma, \Delta \vdash e : \exists t \preccurlyeq \sigma.\tau'}$$

Similarly for property 4. □

Now, we show how to relate various syntactic forms with their corresponding syntax-directed typing rules, in a manner that will allow structural induction on type derivations to apply.

Lemma 18 *Each of the following properties holds:*

1. $\Gamma, \Delta \vdash \Lambda t \preccurlyeq \sigma.e : \Pi t \preccurlyeq \sigma.\tau$ *implies* $\Gamma, \Delta \oplus t \preccurlyeq \sigma \vdash e : \tau'$*, where* $\Delta; t \preccurlyeq \sigma \vdash \tau' \preccurlyeq \tau$.
2. $\Gamma, \Delta \vdash \lambda x : \sigma.e : \tau' \to \tau$ *implies* $\Gamma; x : \sigma, \Delta \vdash e : \tau$*, where* $\Delta \vdash \tau' \preccurlyeq \sigma$.
3. $\Gamma, \Delta \vdash \tau : \mathbf{type}[\tau']$ *implies* $\Delta \vdash \tau \preccurlyeq \tau'$.
4. $\Gamma, \Delta \vdash \langle e \rangle : \langle \cdot \tau \cdot \rangle$ *implies* $\Gamma, \Delta \vdash e : \tau$.

Furthermore, the judgements in the consequents of each of the above implications follow by strict subderivations of the antecedent judgement derivations.

Proof In each of the above implications, the antecedent must be derived by some syntax directed rule instance $\mathcal{R}$ followed by some sequence of non-syntax directed rules, i.e. Sub, ∃-Intro, and ∃-Elim. Lemma 17, properties 3,4, and 1 allow all instances of Sub to be migrated down below all ∃ rule instances, and collapsed into a single Sub rule instance $\mathcal{R}'$. Furthermore, the remaining sequence of ∃-Intro and ∃-Elim will be "nested", in the sense that the first instance of ∃-Elim must be preceded by a ∃-Intro of the same subtyping coercion, allowing this two rule instance sequence to be suppressed by Lemma 17, property 2. This process can be continued for all remaining ∃-Intro rule instances, eliminating all ∃-Intro and ∃-Elim instances between $\mathcal{R}$ and $\mathcal{R}'$. The result follows by syntactic case analysis of the property under consideration and the corresponding form of $\mathcal{R}$. □

5.6 Type Preservation and Type Safety

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 12 The *length of an evaluation relation* $(e, h) \rightarrow^\star (e', h')$ is inductively defined as the sum of all single reduction steps in the evaluation, plus the lengths of all evaluation relations occurring as precedents of Run instances in the derivation of $(e, h) \rightarrow^\star (e', h')$.

Definition 13 A *computational* reduction $(e, h) \rightarrow (e', h')$ is one that holds by a rule other than Context or FContext.

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 expression typings may be instances of initial typings was anticipated and briefly discussed at the beginning of this section.

Theorem 1 (Type Preservation) *If* $(e_0, h_0) : \tau_0 \circ \Delta_0$ *is valid and* $(e_0, h_0) \rightarrow^\star (e_n, h_n)$*, then there exists* (Δ_n, τ_n) *which is an instance of* (Δ_0, τ_0) *such that* $(e_n, h_n) : \tau_n \circ \Delta_n$*.*

Proof Assuming that $(e_0, h_0) \rightarrow^\star (e_n, h_n)$ has length n, the proof follows by induction on n. The result follows trivially for $n = 0$. For the induction step in case $n > 0$, we let $h_n = D_n[m_n]$ and focus on the last step in the reduction, of the form $(e, D[m]) \rightarrow (e_n, h_n)$. We first observe that assumptions of the theorem and the induction hypothesis imply the existence of (Δ, τ) which is an instance of (Δ_0, τ_0) such that $(e, D[m]) : \tau \circ \Delta$. Therefore by Lemma 12 there exists Γ covering $D[m]$ such that $\Gamma, \Delta \vdash e : \tau$ is valid.

Since $\leq$ is the transitive closure of $\trianglelefteq$, it now suffices to show that there exists Γ_n, Δ_n, and τ_n such that: $(\Delta_n, \tau_n) \trianglelefteq (\Delta, \tau)$, and Γ_n extends Γ and covers $D_n[m_n]$, and $\Gamma_n, \Delta_n \vdash e_n : \tau_n$ is valid. Given this, the Theorem follows by Lemma 13. To proceed, we induct on the derivation of $\Gamma, \Delta \vdash e : \tau$ and split the proof into two subclaims, (1) where we assume that this reduction step is computational and (2) where it follows as an instance of Context. These claims cover all cases of the Theorem since reduction by FContext is ruled out by Lemma 16.

Case (1): $(e, D[m]) \rightarrow (e_n, D_n[m_n])$ is a computational reduction, wherein the result follows by subcase analysis on the last step in the derivation of the judgement $\Gamma, \Delta \vdash e : \tau$.

Subcase $\textsc{App}_\Pi$. By definition of typing, evaluation, and Lemma 16 we have that $(e, D[m]) \rightarrow (e_n, h_n)$ is an instance of $\textsc{RApp}_\Pi$, meaning that e is of the form $(\Lambda t \preccurlyeq \sigma.e')\sigma'$ and τ is of the form $\tau'[\sigma'/t]$. Inversion of the last derivation step in the typing obtains:

$$\Gamma, \Delta \vdash \Lambda t \preccurlyeq \sigma.e' : \Pi t \preccurlyeq \sigma.\tau' \qquad\qquad \Delta \vdash \sigma' \preccurlyeq \sigma$$

Further, $\Gamma, \Delta \vdash \sigma' : \mathbf{type}[\sigma']$ by rules of typing so $\Gamma, \Delta \vdash \sigma' : \mathbf{type}[\sigma]$ by Sub. These facts and Lemma 18 also obtain:

$$\Gamma, \Delta \oplus t \preccurlyeq \sigma \vdash e' : \tau'' \qquad\qquad \Delta; t \preccurlyeq \sigma \vdash \tau'' \preccurlyeq \tau'$$

Now, $\Gamma, \Delta \vdash e'[\sigma'/t] : \tau''[\sigma'/t]$ by Lemma 8 since t is not **tlet**-bound in e' (recall that we treat type variables as de Bruijn indices). Further, by Corollary 1 it is the case that $\Delta \vdash \tau''[\sigma'/t] \preccurlyeq \tau'[\sigma'/t]$, so by an application of Sub we have $\Gamma, \Delta \vdash e'[\sigma'/t] : \tau'[\sigma'/t]$. But $(e_n, h_n) = (e'[\sigma'/t], D[m])$ in this case by definition of evaluation; taking $\Gamma_n = \Gamma$, $\Delta_n = \Delta$, and $\tau_n = \tau'[\sigma'/t]$ the result follows.

Subcase App. By definition of typing, evaluation, and Lemma 16 we have that $(e, D[m]) \rightarrow (e_n, h_n)$ is an instance of RApp, meaning that e is of the form $(\lambda x : \sigma.e')v$, and $e_n = e'[v/x]$ and $h_n = D[m]$ for some x, σ, e', and v by definition of evaluation. Further, by inversion of the last step in the typing derivation we obtain:

$$\Gamma, \Delta \vdash \lambda x : \sigma.e' : \tau' \rightarrow \tau \qquad\qquad \Gamma, \Delta \vdash v : \tau'$$

Now, by Lemma 18 we have $\Gamma; x : \sigma, \Delta \vdash e' : \tau$ where $\Delta \vdash \tau' \preccurlyeq \sigma$. Thus by Lemma 7 we also have $\Gamma, \Delta \vdash e'[v/x] : \tau$. Taking $\Gamma_n = \Gamma$, $\Delta_n = \Delta$, and $\tau_n = \tau$, the result follows in this case.

Subcase TLet. By definition of typing, evaluation, and Lemma 16 we have that $(e, D[m]) \rightarrow (e_n, h_n)$ is an instance of RTLet, meaning that e is of the form **tlet** $t \preccurlyeq \sigma = \tau'$ **in** e' and $\Delta = \Delta'; t \preccurlyeq \sigma$ for some Δ' by definition of typing, and $e_n = e'[\tau'/t]$. Also, by inversion in the last step of the typing rule in this case we have:

$$\Gamma, \Delta' \vdash \tau' : \mathbf{type}[\sigma] \qquad\qquad \Gamma, \Delta' \oplus t \preccurlyeq \sigma \vdash e' : \tau$$

and consequently by Lemma 18 we also have $\Delta' \vdash \tau' \preccurlyeq \sigma$. But then by Lemma 8 we may assert that $\Gamma, \Delta' \vdash e'[\tau'/t] : \tau[\tau'/t]$ since t is not **tlet**-bound in e', and also $(\Delta', \tau[\tau'/t]) \trianglelefteq (\Delta'; t \preccurlyeq \sigma, \tau)$ by definition. Taking $\Delta_n = \Delta'$, $\tau_n = \tau[\tau'/t]$, and $\Gamma_n = \Gamma$, the result follows in this case.

Subcase Lift. By definition of typing, evaluation, and Lemma 16 we have that $(e, D[m]) \rightarrow (e_n, h_n)$ is an instance of RLift, meaning that e is of the form **lift** v for some v and $\tau = \langle \cdot\tau' \cdot \rangle$ for some τ' by definition of typing. And by typing inversion in this case we have that $\Gamma, \Delta \vdash v : \tau'$. But then by Lemma 14 and applications of Weaken we have that $\Gamma, \Delta \vdash \textbf{\textit{serialize}}\ v\ D[m] : \tau'$. And $(e_n, h_n = (\langle \textbf{\textit{serialize}}\ v\ D[m] \rangle, D[m])$ in this case by definition of $\rightarrow$. Taking $\Gamma_n = \Gamma$, $\tau_n = \langle \cdot\tau' \cdot \rangle$, and $\Delta_n = \Delta$, the result follows by Lemma 13 and an application of Code.

Subcase Run. In this case $e =$ **run** $\langle e \rangle$ by definition of typing. By typing inversion we have $\Gamma, \Delta \vdash \langle e \rangle : \langle \cdot\tau \cdot \rangle$, and by Lemma 11 we have that $\langle e \rangle$ is closed so $\varnothing, \Delta \vdash \langle e \rangle : \langle \cdot\tau \cdot \rangle$ by obvious properties of typing. Now by Lemma 18 we have $\varnothing, \Delta \vdash e : \tau$, so assuming $(e, \varnothing) \rightarrow^\star (v, h)$ with length j for some j, v, and h we have that $(v, h) : (\Delta', \tau')$ where $(\Delta', \tau') \leq (\Delta, \tau)$ by the first induction hypothesis of this Theorem since $j < n$ by definition. But since **fail** is untypable it cannot be that $v =$ **fail**, so by evaluation rules we have that $(e, D[m]) \rightarrow (e_n, h_n)$ can only be an instance of RRun, meaning that e' is of the form $(\textbf{\textit{serialize}}\ v\ h, D[m])$. Thus $\Gamma, \Delta' \vdash \textbf{\textit{serialize}}\ v\ h : \tau'$ by Lemma 14 and applications of Weaken, so taking $\Gamma_n = \Gamma$, $\Delta_n = \Delta'$, and $\tau_n = \tau'$ the result follows.

Subcase Ref. In this case $e =$ **ref** v and $\tau =$ **ref** τ' and $(e_n, h_n) = (z, D[$**let** $z = v$ **in** $m])$ for some τ' and v and fresh z by definition of typing and evaluation. By typing inversion in this case we have $\Gamma, \Delta \vdash v : \tau'$, so clearly $\Gamma; z :$ **ref** τ' covers $D[$**let** $z = v$ **in** $m]$ since Γ covers $D[m]$ by assumption and z is fresh. And $\Gamma; z :$ **ref** $\tau', \Delta \vdash z : \tau$ by one application of Var, so taking $\Delta_n = \Delta$, $\tau_n = \tau$, and $\Gamma_n = \Gamma; z :$ **ref** τ' the result follows.

Subcase $\exists$-Elim. In this case we have that $\Delta = \Delta'; t \preccurlyeq \sigma$ for some Δ', t, and σ by definition of typing. And by typing inversion in this case we have $\Gamma, \Delta' \vdash e : \exists t \preccurlyeq \sigma.\tau$.

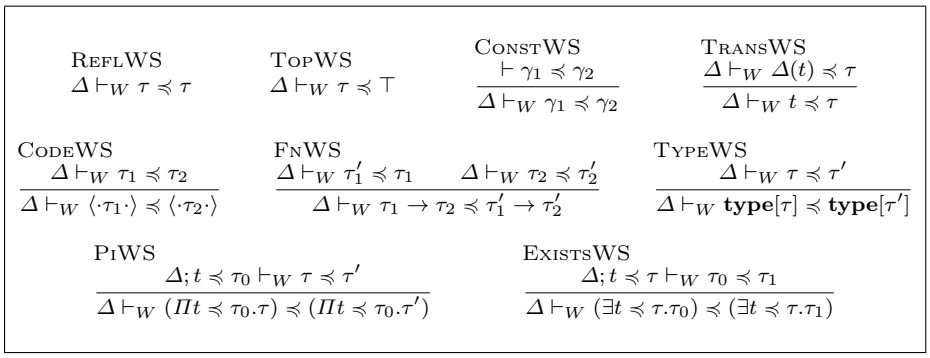

Fig. 14 Algorithmic Subtyping Rules

But by the second induction hypothesis of the theorem we then have that $\Gamma', \Delta'' \vdash e_n : \tau'$ where Γ' is some extension of Γ and $(\Delta'', \tau') \trianglelefteq (\Delta', \exists t \preccurlyeq \sigma.\tau)$. Now, we proceed by considering four possible conditions identified in Lemma 5. Assuming (i), the subcase follows by an instance of $\exists$-Elim applied to $\Gamma', \Delta'' \vdash e_n : \tau'$. Assuming (ii), the subcase follows immediately by definition of $\trianglelefteq$. Assuming (iii) or (iv), the subcase follows by an instance of $\exists$-Elim applied to $\Gamma', \Delta'' \vdash e_n : \tau'$.

Subcase $\exists$-Intro follows in a manner similar to subcase $\exists$-Elim.

Subcase Sub. In this case we have that $\Gamma, \Delta \vdash e : \tau_1$ with $\Delta \vdash \tau_1 \preccurlyeq \tau$ by typing inversion. Thus $\Gamma', \Delta' \vdash e_n : \tau_2$ with Γ' some extension of Γ and $(\Delta', \tau_2) \trianglelefteq (\Delta, \tau_1)$ by the induction hypothesis. But then by Lemma 4 there exists τ_2' with $\Delta' \vdash \tau_2 \preccurlyeq \tau_2'$ and $(\Delta', \tau_2') \trianglelefteq (\Delta, \tau)$, hence $\Gamma', \Delta' \vdash e_n : \tau_2'$ by Sub, so this subcase follows.

The other subcases of Case (1) follow in a relatively straightforward manner. What remains to be shown is the second Case.

Case (2). Suppose on the other hand that $(e, D[m]) \to (e_n, h_n)$ holds as an instance of Context, so that e is of the form $E[e']$ and e_n is of the form $E[e_n']$ and $(e', D[m]) \to (e_n', h_n)$. If $E = [\,]$, the result follows by Case (1) above. Otherwise, by Lemma 9 we may assert the existence of some τ' such that $\Gamma, \Delta \vdash e' : \tau'$, so that by the induction hypothesis we may assert the existence of Γ_n which is an extension of Γ that covers h_n and the existence of $(\Delta_n, \tau_n') \leq (\Delta, \tau')$ such that $\Gamma_n, \Delta_n \vdash e_n' : \tau_n'$. But then by Lemma 10 we have that $\Gamma_n, \Delta_n \vdash E[e_n'] : \tau_n$ where $(\Delta_n, \tau_n) \leq (\Delta, \tau)$ for some τ_n, obtaining the result. □

Finally, our main result follows easily by Type Preservation.

Theorem 2 (Type Safety) *If* $(e_0, h_0) : \tau_0 \circ \Delta$ *is valid then it is not the case that* $(e_0, h_0) \to^\star (\mathbf{fail}, h_1)$ *for some* h_1.

Proof The result follows by Theorem 1, since **fail** is untypable. □

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 [22]. 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. This function is useful during type checking (Fig. 15) to convert type variables to structured types for syntax direction of the algorithm; it is not used by the subtyping algorithm itself.

Definition 14 The relation $\ll$ promotes a type variable to the structured type which is its ***lub*** given a coercion:

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

Next, we define a relation $\Delta \vdash_W \tau \preccurlyeq \tau'$ which is the algorithmic version of subtyping. The derivation rules for the relation are given in Fig. 14. This relation in fact subsumes the algorithmic subtyping relation defined in [22] with trivial extensions to accomodate code types; we may therefore use their main result, which equates the logical and algorithmic subtyping relations:

Lemma 19 (Equivalence of Algorithmic and Logical Subtyping) $\Delta \vdash_W \tau \preccurlyeq \tau'$ *iff* $\Delta \vdash \tau \preccurlyeq \tau'$.

It is also easy to see that promotion is subsumed by the algorithmic subtyping relation, hence we can assert the following:

Lemma 20 $\Delta \vdash \tau \ll \tau'$ *implies* $\Delta \vdash_W \tau \preccurlyeq \tau'$.

Finally, we note that 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 21 *The relation* $\Delta \vdash_W \tau \preccurlyeq \tau'$ *induces a decision procedure: there is an algorithm which given* Δ, τ *and* τ' *returns* true *if* $\Delta \vdash_W \tau \preccurlyeq \tau'$ *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 $\Gamma, \Delta \vdash e : \tau$. We begin with relevant definitions, and then discuss type checking as a sort of normal-form type derivation.

In the computation of any type checking judgement $\Gamma, \Delta \vdash_W e : \tau$, our strategy is to pass Λ-defined coercions down via Δ, and return extruded **tlet**-defined coercions up via existential bindings in the type τ. Because there may be multiple such bindings,

$$\textsc{ConstW}\quad \Gamma, \Delta \vdash_W \mathbf{c} : ty(\mathbf{c})$$

$$\textsc{VarW}\quad \frac{\Gamma(x) = \tau}{\Gamma, \Delta \vdash_W x : \tau}$$

$$\textsc{TypeW}\quad \Gamma, \Delta \vdash_W \tau : \mathbf{type}[\tau]$$

$$\textsc{App}_{\Pi}\textsc{W}\quad \frac{\Gamma, \Delta \vdash_W e : \exists \Delta'.\varsigma \qquad \Delta; \Delta' \vdash \varsigma \ll \Pi t \preccurlyeq \tau''.\tau' \qquad \Delta; \Delta' \vdash_W \tau \preccurlyeq \tau''}{\Gamma, \Delta \vdash_W e\ \tau : \exists \Delta'.(\tau'[\tau/t])}$$

$$\textsc{AppW}\quad \frac{\begin{array}{c}\Gamma, \Delta \vdash_W e_1 : \exists \Delta_1.\varsigma_1 \\ \Gamma, \Delta \vdash_W e_2 : \exists \Delta_2.\varsigma_2\end{array} \qquad \Delta; \Delta_1; \Delta_2 \vdash \varsigma_1 \ll \tau' \to \tau \qquad \Delta; \Delta_1; \Delta_2 \vdash_W \varsigma_2 \preccurlyeq \tau'}{\Gamma, \Delta \vdash_W e_1 e_2 : \exists \Delta_1; \Delta_2.\tau}$$

$$\textsc{AbsW}\quad \frac{\Gamma; x : \tau, \Delta \vdash_W e : \exists \Delta'.\varsigma}{\Gamma, \Delta \vdash_W \lambda x : \tau.e : \exists \Delta'.\tau \to \varsigma}$$

$$\textsc{Abs}_{\Lambda}\textsc{W}\quad \frac{\Gamma, \Delta; t \preccurlyeq \tau \vdash_W e : \tau'}{\Gamma, \Delta \vdash_W \Lambda t \preccurlyeq \tau.e : \Pi t \preccurlyeq \tau.\tau'}$$

$$\textsc{CodeW}\quad \frac{peel(\Gamma|_{\mathrm{fv}(e)}), \Delta \vdash_W e : \exists \Delta'.\varsigma}{\Gamma, \Delta \vdash_W \langle e \rangle : \exists \Delta'.\langle \cdot \varsigma \cdot \rangle}$$

$$\textsc{RunW}\quad \frac{\Gamma, \Delta \vdash_W e : \exists \Delta'.\varsigma \qquad \Delta; \Delta' \vdash \varsigma \ll \langle \cdot \tau \cdot \rangle}{\Gamma, \Delta \vdash_W \mathbf{run}\ e : \exists \Delta'.\tau}$$

$$\textsc{LiftW}\quad \frac{\Gamma, \Delta \vdash_W e : \exists \Delta'.\varsigma}{\Gamma, \Delta \vdash_W \mathbf{lift}\ e : \exists \Delta'.\langle \cdot \varsigma \cdot \rangle}$$

$$\textsc{CastW}\quad \frac{\Gamma, \Delta \vdash_W e : \tau'}{\Gamma, \Delta \vdash_W (\tau)e : \tau}$$

$$\textsc{LetW}\quad \frac{\Gamma, \Delta \vdash_W e_1 : \exists \Delta_1.\varsigma_1 \qquad \Gamma; x : \varsigma_1, \Delta; \Delta_1 \vdash_W e_2 : \tau}{\Gamma, \Delta \vdash_W \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \exists \Delta_1.\tau}$$

$$\textsc{TLetW}\quad \frac{\begin{array}{c}\Gamma, \Delta \vdash_W e : \exists \Delta'.\varsigma \\ \Delta; \Delta' \vdash \varsigma \ll \mathbf{type}[\tau''] \qquad \Delta; \Delta' \vdash_W \tau'' \preccurlyeq \tau' \qquad \Gamma, \Delta; t \preccurlyeq \tau' \vdash_W e' : \tau\end{array}}{\Gamma, \Delta \vdash_W \mathbf{tlet}\ t \preccurlyeq \tau' = e\ \mathbf{in}\ e' : \exists \Delta'.\exists t \preccurlyeq \tau'.\tau}$$

Fig. 15 Type Checking Rules

we make heavy use of the syntactic sugaring $\exists \Delta.\tau$ specified in Definition 7. It is also important to distinguish structured types from existential types in the definition of the algorithm:

Definition 15 We let ς range over *structured types*, i.e. any type not of the form $\exists t \preccurlyeq \sigma.\tau$.

Thus, every type checking judgement is in fact of the form $\Gamma, \Delta \vdash_W e : \exists \Delta'.\varsigma$, with Δ' containing extruded type coercions. It will often be necessary to append these extruded coercions to top-level coercions, both in the computation of subtypes and also in the course of proving soundness. Hence we define coercion append as follows:

Definition 16 Letting $\Delta = \varnothing; t_1 \preccurlyeq \tau_1; \ldots; t_n \preccurlyeq \tau_n$, we define $\Delta'; \Delta \triangleq \Delta'; t_1 \preccurlyeq \tau_1; \ldots; t_n \preccurlyeq \tau_n$.

Further, in order to move between code levels in static typing, we define a function *peel* as follows:

Definition 17 The function *peel* removes a layer of code type from an environment:

$$peel(\varnothing) = \varnothing$$
$$peel(\Gamma; x : \langle \cdot \tau \cdot \rangle) = peel(\Gamma); x : \tau$$

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

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

Type derivation rules for the algorithmic system are given in Fig. 15; the reader will observe that these rules are entirely syntax-directed. Given the above definitions, and also decidability of subtyping as noted above, the syntax-directed nature of type checking immediately implies decidability of type checking:

Lemma 22 (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 `fail`s otherwise.*

6.2.1 Type Checking as Normal Form Derivation

As discussed above, type judgements in our system are of the form $\Gamma, \Delta \vdash_W e : \exists \Delta'.\varsigma$ 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 [33].

6.3 Soundness of Type Checking

Now we prove that type checking is sound with respect to the logical system that was shown to enjoy type safety. In the course of this presentation we will refine the algorithm slightly to ensure consistency of typing coercions during computation. In particular, note that when computing types for expressions with multiple subexpression, the computation of each subexpression type will return an existential type carrying assumptions about type variables within the subexpressions. These assumptions need

to be "fused" and percolated upward for the type computation of the root expression; therefore the assumptions must be consistent. We begin by formalizing an adequate notion of coercion consistency as follows:

Definition 19 Coercions Δ and Δ' are *consistent*, written $\Delta \bowtie \Delta'$, iff $t \in \mathrm{Dom}(\Delta) \cap \mathrm{Dom}(\Delta')$ implies $\Delta(t) = \Delta(t')$.

In order to demonstrate soundness, we first show that the strategy of reconstructing coercions from leaves towards the root in typing derivations, checking consistency, is fundamentally sound wrt the logical system, as follows. Both stated properties hold by a straightforward induction on derivations; consistent weakening of coercions is monotonic wrt subtyping derivability.

Lemma 23 (Coercion Weakening) *Given $\Delta \bowtie \Delta'$, both of the following properties hold:*

1. *$\Delta \vdash \tau \preccurlyeq \sigma$ implies $\Delta; \Delta' \vdash \tau \preccurlyeq \sigma$*
2. *$\Gamma, \Delta \vdash e : \tau$ implies $\Gamma, \Delta; \Delta' \vdash e : \tau$*

When proving soundness we will typically make multiple applications in a row of either $\exists$ introduction and elimination in the logical system, so we introduce shorthand to simplify this part of the presentation.

Definition 20 We add to the logical type system the following sound sugarings for sequences of existential intros and elims:

$$\exists\text{-}\textsc{Intro}^* \quad \frac{\Gamma, \Delta; \Delta' \vdash e : \tau'}{\Gamma, \Delta \vdash e : \exists \Delta'.\tau'} \qquad \exists\text{-}\textsc{Elim}^* \quad \frac{\Gamma, \Delta \vdash e : \exists \Delta'.\tau'}{\Gamma, \Delta; \Delta' \vdash e : \tau'}$$

Next we must establish some sanity and hygiene conditions on type checking inputs and results. In the logical system we defined well-formedness of judgements, whereas *a priori* we impose well-formedness conditions only on *inputs* to type checking; type soundness will demonstrate that well-formedness of the resulting judgement is obtained by the algorithm. We note that at the top level these conditions are trivially satisfied given a closed expression and an empty environment and coercion; they are subsequently maintained as a representation invariant during computation.

Definition 21 The tuple (Δ, Γ, e) is *well-formed* iff each of the following properties hold:

1. $(\Delta, \Gamma(x))$ is well formed for all $x \in \mathrm{Dom}(\Gamma)$
2. $\mathrm{ftv}(e) \subseteq \mathrm{Dom}(\Delta)$
3. All **tlet**$-$ and Λ-bound variables in e are not in $\mathrm{Dom}(\Delta)$.

Proving type soundness requires to observe that well-formedness of coercion, type pairs is preserved by algorithmic subtyping:

Lemma 24 *If (Δ, σ) is well-formed and $\Delta \vdash_W \sigma \preccurlyeq \tau$ then (Δ, τ) is well-formed.*

Finally, we refine the type checking algorithm slightly with hygiene conditions that will ensure consistency of coercions returned by the algorithm. Although α-renaming of Λ- and **tlet**-bound terms will ensure "freshness" of those variables, type annotations on λ-bound variables, type casts, and type assigments in the environment may mention $\exists$-bound types as well. The following hygiene condition will ensure the consistency of returned coercions.

Definition 22 The notion of a *hygienic* type is defined inductively as follows: any structured type ς is hygienic, and $\exists t \preccurlyeq \sigma.\exists\Delta.\varsigma$ is hygienic iff $\exists\Delta.\varsigma$ is and $t \in \mathrm{Dom}(\Delta)$ implies $\Delta(t) = \sigma$.

A check for this hygiene condition is easily implemented and incorporated into a type checking implementation. To increase efficiency we observe that such a check need not be performed at every step of type checking, but rather at strategic points; hygiene is maintained by the so-called *canonical* algorithm as demonstrated by type soundness:

Definition 23 A judgement of the form $\Gamma, \Delta \vdash_W e : \exists\Delta'.\varsigma$ where $\exists\Delta'.\varsigma$ is hygienic and $\Delta \bowtie \Delta'$ is called *hygienic*. An algorithmic derivation is *canonical* iff the consequent of any instance of the following inference rules it contains is hygienic: ConstW,VarW,App$_\Pi$W,AppW, RunW,CastW, TletW.

Now the pieces are in place to demonstrate our main result for type checking, which follows by induction on derivations. The statement of the result mentions well-formedness and hygiene conditions so that the induction hypothesis is sufficiently strengthened to obtain *validity*, not just derivability, of the analagous derivation in the logical system.

Lemma 25 (Soundness of Type Checking) *Given well-formed (Δ, Γ, e), if the judgment $\Gamma, \Delta \vdash_W e : \tau$ is canonically derivable, then it is hygienic with (Δ, τ) well-formed and $\Gamma, \Delta \vdash e : \tau$ is valid.*

Proof The result follows by induction on the derivation of $\Gamma, \Delta \vdash_W e : \tau$ and case analysis on the last step thereof.
Case App$_\Pi$W. In this case by definition of type checking we have $e = e'\sigma$ and $\tau = \exists\Delta'.\tau'[\sigma/t]$, where by inversion of the last step there exist derivable judgements of the following form:

$$\Gamma, \Delta \vdash_W e' : \exists\Delta'.\varsigma' \qquad \Delta; \Delta' \vdash \varsigma' \ll \Pi t \preccurlyeq \tau''.\tau' \qquad \Delta; \Delta' \vdash_W \sigma \preccurlyeq \tau''$$

Since (Γ, Δ, e') must be well-formed, therefore by the induction hypothesis we may assert validity of $\Gamma, \Delta \vdash e' : \exists\Delta'.\varsigma'$, and $\Delta; \Delta' \vdash \varsigma' \preccurlyeq (\Pi t \preccurlyeq \tau''.\tau')$ by Lemma 20 and Lemma 19. Also by the induction hypothesis we have that $(\Delta, \exists\Delta'.\varsigma')$ is well-formed, hence $\Delta; \Delta'$ is closed. Therefore the following valid derivation fragment can be constructed[3]:

$$\dfrac{\dfrac{\Gamma, \Delta \vdash e' : \exists\Delta'.\varsigma'}{\Gamma, \Delta; \Delta' \vdash e' : \varsigma'}\,\exists\text{-Elim}^* \qquad \Delta; \Delta' \vdash \varsigma' \preccurlyeq (\Pi t \preccurlyeq \tau''.\tau')}{\Gamma, \Delta; \Delta' \vdash e' : \Pi t \preccurlyeq \tau''.\tau'}\,\textsc{Sub}$$

But we have also $\Delta; \Delta' \vdash \sigma \preccurlyeq \tau''$ by Lemma 19, so the following can be constructed:

$$\dfrac{\dfrac{\Gamma, \Delta; \Delta' \vdash e' : \Pi t \preccurlyeq \tau''.\tau' \qquad \Delta; \Delta' \vdash \sigma \preccurlyeq \tau''}{\Gamma, \Delta; \Delta' \vdash e'\sigma : \tau'[\sigma/t]}\,\textsc{App}_\Lambda}{\Gamma, \Delta \vdash e'\sigma : \exists\Delta'.\tau'[\sigma/t]}\,\exists\text{-Intro}^*$$

Finally, we have that $(\Delta; \Delta', \Pi t \preccurlyeq \tau''.\tau')$ is well-formed by Lemma 24, so that also $(\Delta; \Delta', \tau'[\sigma/t])$ is well-formed since $\mathrm{ftv}(\sigma) \subseteq \mathrm{Dom}(\Delta)$ by assumptions of the Lemma. Since $\Gamma, \Delta \vdash_W e : \tau$ is hygienic by definition in this case, the result follows.

[3] Operative rules label consequents in all reconstructed derivation fragments.

Case TLetW. In this case we have $e = (\mathbf{tlet}\ t \preccurlyeq \sigma = e_0\ \mathbf{in}\ e_1)$ and $\tau = \exists \Delta_0.t \preccurlyeq \sigma.\tau_1$, where by inversion we have:

$$\Gamma, \Delta \vdash_W e_0 : \exists \Delta_0.\varsigma \qquad \Delta; \Delta_0 \vdash \varsigma \ll \mathbf{type}[\sigma'] \qquad \Delta; \Delta_0 \vdash_W \sigma' \preccurlyeq \sigma$$
$$\Gamma, \Delta; t \preccurlyeq \sigma \vdash_W e_1 : \tau_1$$

Now, by the induction hypothesis we have that $\Gamma, \Delta \vdash e_0 : \exists \Delta_0.\varsigma$ is valid, and $\Delta; \Delta_0 \vdash \varsigma \preccurlyeq \mathbf{type}[\sigma']$ by Lemma 20 and Lemma 19. Also by the induction hypothesis we have that $(\Delta, \exists \Delta_0.\varsigma)$ is well-formed so $\Delta; \Delta_0$ is closed. And $\Delta; \Delta_0 \vdash \sigma' \preccurlyeq \sigma$ by Lemma 19, wence we can construct the following valid derivation fragment:

$$\dfrac{\dfrac{\dfrac{\Gamma, \Delta \vdash e_0 : \exists \Delta_0.\varsigma}{\Gamma, \Delta; \Delta_0 \vdash e_0 : \varsigma}\ \exists\text{-Elim}^* \qquad \Delta; \Delta_0 \vdash \varsigma \preccurlyeq \mathbf{type}[\sigma']}{\Gamma, \Delta; \Delta_0 \vdash e_0 : \mathbf{type}[\sigma']}\ \textsc{Sub} \qquad \Delta; \Delta_0 \vdash \mathbf{type}[\sigma'] \preccurlyeq \mathbf{type}[\sigma]}{\Gamma, \Delta; \Delta_0 \vdash e_0 : \mathbf{type}[\sigma]}\ \textsc{Sub}$$

Now, since $\Gamma, \Delta \vdash_W e : \tau$ is hygienic by construction in this case, therefore also $\exists \Delta_0.\exists t \preccurlyeq \sigma.\tau_1$ is hygienic. Since $t \notin \mathrm{Dom}(\Delta)$ this means that $\Delta; t \preccurlyeq \sigma \bowtie \Delta_0$. And since $\mathrm{ftv}(\sigma) \subseteq \mathrm{Dom}(\Delta)$ by well-formedness therefore $\Delta; \Delta_0; t \preccurlyeq \sigma$ is closed, so by the induction hypothesis and Lemma 23 we have that $\Gamma, \Delta; \Delta_0 \oplus t \preccurlyeq \sigma \vdash e_1 : \tau_1$ is valid. Hence the following derivation fragment can be constructed:

$$\dfrac{\dfrac{\Gamma, \Delta; \Delta_0 \vdash e_0 : \mathbf{type}[\sigma] \qquad \Gamma, \Delta; \Delta_0 \oplus t \preccurlyeq \sigma \vdash e_1 : \tau_1}{\Gamma, \Delta; \Delta_0 \oplus t \preccurlyeq \sigma \vdash \mathbf{tlet}\ t \preccurlyeq \sigma = e_0\ \mathbf{in}\ e_1 : \tau_1}\ \textsc{TLet}}{\Gamma, \Delta \vdash \mathbf{tlet}\ t \preccurlyeq \sigma = e_0\ \mathbf{in}\ e_1 : \exists \Delta_0.t \preccurlyeq \sigma.\tau_1}\ \exists\text{-Intro}^*$$

The result follows in this case.

Case AppW In this case $e = e_1 e_2$ and $\tau = \exists \Delta_1; \Delta_2.\tau_1$, where by inversion we have:

$$\Gamma, \Delta \vdash_W e_1 : \exists \Delta_1.\varsigma_1 \qquad \Delta; \Delta_1 \vdash \varsigma_1 \ll \tau_0 \to \tau_1 \qquad \Gamma, \Delta \vdash_W e_2 : \exists \Delta_2.\varsigma_2$$
$$\Delta; \Delta_1; \Delta_2 \vdash_W \varsigma_2 \preccurlyeq \tau_0$$

By the induction hypothesis both $\Gamma, \Delta \vdash e_1 : \exists \Delta_1.\varsigma_1$ and $\Gamma, \Delta \vdash e_2 : \exists \Delta_2.\varsigma_2$ are derivable and $\Delta; \Delta_1; \Delta_2$ is closed, and clearly $\Delta_1; \Delta_2 = \Delta_2; \Delta_1$ since τ is hygienic. Also, both $\Delta; \Delta_1; \Delta_2 \vdash \varsigma_1 \preccurlyeq \tau_0 \to \tau_1$ and $\Delta; \Delta_2; \Delta_2 \vdash \varsigma_2 \preccurlyeq \tau_0$ are valid by Lemma 20 and Lemma 19. The following derivation fragments can therefore be constructed by Lemma 23:

$$\dfrac{\dfrac{\dfrac{\Gamma, \Delta; \Delta_2 \vdash e_1 : \exists \Delta_1.\varsigma_1}{\Gamma, \Delta; \Delta_1; \Delta_2 \vdash e_1 : \varsigma_1}\ \exists\text{-Elim}^* \qquad \Delta; \Delta_1; \Delta_2 \vdash \varsigma_1 \preccurlyeq \tau_0 \to \tau_1}{\Gamma, \Delta; \Delta_1; \Delta_2 \vdash e_1 : \tau_0 \to \tau_1}\ \textsc{Sub}$$

$$\dfrac{\dfrac{\dfrac{\Gamma, \Delta; \Delta_1; \vdash e_2 : \exists \Delta_2.\varsigma_2}{\Gamma, \Delta; \Delta_1; \Delta_2 \vdash e_2 : \varsigma_2}\ \exists\text{-Elim}^* \qquad \Delta; \Delta_1; \Delta_2 \vdash \varsigma_2 \preccurlyeq \tau_0}{\Gamma, \Delta; \Delta_1; \Delta_2 \vdash e_2 : \tau_0}\ \textsc{Sub}$$

$$\dfrac{\dfrac{\Gamma, \Delta; \Delta_1; \Delta_2 \vdash e_1 : \tau_0 \to \tau_1 \qquad \Gamma, \Delta; \Delta_1; \Delta_2 \vdash e_2 : \tau_0}{\Gamma, \Delta; \Delta_1; \Delta_2 \vdash e_1 e_2 : \tau_1}\ \textsc{App}}{\Gamma, \Delta \vdash e_1 e_2 : \exists \Delta_1; \Delta_2.\tau_1}\ \exists\text{-Intro}^*$$

Since $(\Delta; \Delta_1; \Delta_2, \tau_0 \to \tau_1)$ is well-formed by Lemma 24, the result follows in this case.

Case $\textsc{Abs}_\Lambda$W. In this case $e = \Lambda t \preccurlyeq \tau.e'$ and $\tau = \Pi t \preccurlyeq \sigma.\tau'$, where we have $\Gamma, \Delta; t \preccurlyeq \sigma \vdash_W e' : \tau'$ by inversion. But $t \notin \mathrm{Dom}(\Delta)$ by well-formedness, hence by the induction hypothesis $\Gamma, \Delta \oplus t \preccurlyeq \sigma \vdash e' : \tau'$ is derivable, with $(\Delta; t \preccurlyeq \sigma, \tau')$ well-formed. But then $(\Delta, \Pi t \preccurlyeq \sigma.\tau')$ is clearly well-formed, and $\Gamma, \Delta \vdash \Lambda t \preccurlyeq \tau.e' : \Pi t \preccurlyeq \sigma.\tau'$ is derivable by an instance of App, so the result follows in this case.

The remaining cases follow in a relatively straightforward manner by the induction hypothesis. □

7 A Programming Example

In this section, we use WSN programming as a case study to demonstrate how ⟨ML⟩ 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 [35]. Thus, given *e.g.* a radio communication function that is going to be executed on the a sensor node, the way to significantly improve system efficiency is not to shorten the code of the function, but to minimize the bandwidth it would consume by *shortening the size of the packet* transmitted by the radio.

The example we present in this section fleshes out this observation. If we can specialize the type of a node address 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 radio packet transmission*. 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 represent addresses, rather than **uint64**, and save radio power.

We will use some language constructs beyond the ⟨ML⟩ formal core in the example, including **for** loops and arrays; adding these features is not difficult technically and clarifies the presentation. The code will also assume the user-defined constant types include **uint4** subtype of **uint8** subtype of **uint16**, and so on, and that all of these integer constant forms are a subtype of **uint**. We allow **tlet** syntax to be used here without any upper bound on the variable, in which case the upper bound can be taken to be the same as the type of the expression bound to the variable.

The rest of the section describes the example in more detail. Code snippets are found in Fig. 16 and Fig. 17. Fig. 16 illustrates a type-specializable program for inter-mote packet send, and is described in Sections. 7.1-7.3. Fig. 17 contains hub code for the entire deployment process, and is described in Section. 7.4. The code here is only a small part of a full-fledged application, since packet receiving, mote synchronization, low-level hardware control, and other features are missing.

$$
\begin{array}{l}
radio = \Lambda\, addr\_t \preccurlyeq \mathbf{uint}. \\
\quad\quad \Lambda\, dlen\_t \preccurlyeq \mathbf{uint}. \\
\quad\quad \Lambda\, msg\_t \preccurlyeq \{header : \{dest :\ addr\_t\}; data : \mathbf{uint8}[dlen\_t]\}. \\
\quad\quad \lambda\, msg : \langle\cdot msg\_t\cdot\rangle. \\
\quad\quad\quad \langle \quad \ldots \text{prepare packets and physically send } msg \text{ based on } msg\_t \ldots \quad \rangle \\
\\
\\
\\
\texttt{FLEN} = \mathbf{uint4} \\
send = \lambda\, parity : \mathbf{bool}. \\
\quad\quad \Lambda\, addr\_t \preccurlyeq \mathbf{uint}. \\
\quad\quad \lambda\, self : \overline{\langle\cdot addr\_t\cdot\rangle}. \\
\quad\quad \lambda\, other : \langle\cdot addr\_t\cdot\rangle. \\
\quad\quad \Lambda\, dlen\_t \preccurlyeq \mathbf{uint}. \\
\quad\quad \lambda\, d : \langle\cdot \mathbf{uint8}[dlen\_t]\cdot\rangle. \\
\quad\quad\quad \mathbf{tlet}\ hd\_t = \{src : addr\_t;\ dest : addr\_t\}\ \mathbf{in} \\
\quad\quad\quad \mathbf{tlet}\ ms\_t = \{header : hd\_t;\ data : \mathbf{uint8}[dlen\_t]\}\ \mathbf{in} \\
\quad\quad\quad \mathbf{tlet}\ ml\_t = \left\{ \begin{array}{l} header : hd\_t;\ data : \mathbf{uint8}[dlen\_t]; \\ footer : \mathbf{uint8}[\texttt{FLEN}] \end{array} \right\}\ \mathbf{in} \\
\quad\quad\quad \mathbf{if}\ (parity) \\
\quad\quad\quad\quad radio\ addr\_t\ dlen\_t\ ml\_t \left\langle \begin{array}{l} \mathbf{let}\ ft = \ldots \text{CRC algorithm on } d \ldots\ \mathbf{in} \\ \left\{ \begin{array}{l} header = \{src = self; dest = other\}; \\ data = d; \\ footer = ft \end{array} \right\} \end{array} \right\rangle \\
\quad\quad\quad \mathbf{else} \\
\quad\quad\quad\quad radio\ addr\_t\ dlen\_t\ ms\_t \left\langle \left\{ \begin{array}{l} header = \{src = self; dest = other\}; \\ data = d \end{array} \right\} \right\rangle \\
\\
\\
\texttt{D\_BD} = \mathbf{uint8} \\
mote = \lambda\, parity : \mathbf{bool}. \\
\quad\quad \Lambda\, addr\_t \preccurlyeq \mathbf{uint}. \\
\quad\quad \lambda\, self : \overline{\langle\cdot addr\_t\cdot\rangle}. \\
\quad\quad \lambda neighbors : addr\_t[]. \\
\quad\quad \lambda neighbor\_num : \mathbf{uint16}. \\
\quad\quad\quad \mathbf{let}\ c = \mathbf{ref}\ \langle 0\rangle\ \mathbf{in} \\
\quad\quad\quad \mathbf{for}(\mathbf{uint16}\ i = 0; i < neighbor\_num; i{+}{+})\{ \\
\quad\quad\quad\quad \mathbf{let}\ addr = \mathbf{lift}\ neighbors[i]\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ sendf = send\ parity\ addr\_t\ self\ addr\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ pl\ = payload\ i\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{tlet}\ dlen\_t\ \preccurlyeq \texttt{D\_BD} = \mathbf{if}\ (pl.size < 16)\ \mathbf{then\ uint4\ else}\ \texttt{D\_BD}\ \mathbf{in} \\
\quad\quad\quad\quad c := seq\ !c\ (sendf\ dlen\_t\ (\mathbf{uint8}[dlen\_t])pl.content) \\
\quad\quad\quad \};!c \\
\quad seq = \lambda x : \langle\cdot\top\cdot\rangle.\lambda y : \langle\cdot\top\cdot\rangle.\langle x; y\rangle
\end{array}
$$

Fig. 16 Code Snippet for Mote Code Specialization

7.1 A Specializable "Radio" Snippet

In the standard TinyOS sensor network platform [26], 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 radio communication function that uses type `message_t` for messages will not necessarily be efficient. For example, 64-bit addresses are hardcoded inside this data structure; the length of data is fixed to `TOSH_DATA_LENGTH`; the message is required to carry fields such as `footer` even when there are times that they may carry no useful information. ⟨ML⟩ can effectively avoid these situations. A more flexible implementation of the *radio* function in Fig. 16 allows programmers to

- customize the type *addr_t*, used to represent the addresses
- customize the type *dlen_t*, used to represent the length of the data; as our core calculus does not have refined singleton integer types, the example here uses notation such as **uint8**[**uint4**] to represent an array of size 16.
- customize the type *msg_t*, used to represent the format of the packet

Note that the *radio* function here is different from a standard systems programming function that encapsulates the radio communication logic. It is a staged function that returns *a piece of first-class code* which in turn encapsulates the communication logic. The body of this piece of code prepares memory layouts and radio communication logic based on *msg_t*; a more customized message format (including address format, and data length) can positively contribute to power savings when motes are running after specialization. All three types are abstracted, and eventually will be instantiated at the meta-stage with the most efficient concrete types. Type *msg_t* is given a type bound of a record type with at least a *header* field with a destination address *dest* inside, and a *data* field with length *dlen_t*.

7.2 A Specializable "Send" Utility

Type specialization over *radio* is illustrated by function *send*. Overall, *send* is a "message assembler" – given message fragments such as data *d*, source address *self*, destination address *other*, and a boolean *parity* bit indicating whether a CRC-style footer is needed, the function assembles the messages into a record, and sends it to the underlying *radio* function. Like *radio*, *send* is also a staged program, in the sense that the function is meta-stage code which builds object stage code able to send a particular customized form of message through radio communication.

The bulk of this program centers on whether a *footer* field is needed for the message. If *parity* is set to be true, a footer is calculated. Note that at the execution of *send*, argument *d* is only a piece of code which *at object stage* will turn into byte-array data.

It is thus necessary that the entire CRC algorithm is part of the object stage execution as well, enclosed in the $\langle \ldots \rangle$ bracket. Observe that at specialization time, any subtype of the type bound for *msg_t* in *radio* can be applied. Both *mts* and *mtl* are such subtypes: the first has an additional *dest* field, and the second has both an additional *footer* field as well as a *dest* field.

The *addr_t* and *dlen_t* types in the *radio* function are also specialized, by real arguments of the same in the *send* function. The two arguments in turn are passed in to *send*, meaning *send* can be further specialized. Note that as a result of the dependency of *ml_t* (or *ms_t*) over *addr_t* and *dlen_t* corresponding to the dependency specified by the signature of *radio*, both type specializations typecheck.

7.3 A Specializable Toy Program on Motes

The *send* code we have described is a "utility" function that could be used in a full application deployed to the motes by the hub. We now define a complete toy mote application, as *mote* in Fig. 16. The program just sends a beacon broadcast message to the mote's 1-hop neighbors. This example is constructed to showcase the power of staged programming in two aspects: loop unrolling and data length specialization, which we describe now.

As a specializable staged function, *mote* 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 *mote* is specialized. As a result, rather than allocating the array *neighbors* in (scarce) mote memory, loop unrolling can be performed before the code is deployed. This can be illustrated in the body of the *mote* function. Here, the mote intends to send a beacon message to all neighbors one by one[4]. Note that the **for** loop is executed at the *meta-stage*, and the *neighbors* array does not exist at the object stage at all. Auxiliary binary function *seq* pieces code together via statement sequencing.

The *mote* here further demonstrates the ability to specialize on data field length. In this toy example, let us assume function

$$payload :: \mathbf{uint16} \to \{content : \mathbf{uint8}[\texttt{D\_BD}]; size : \texttt{D\_BD}\}$$

is able to pre-compute the payload information given a node ID (the argument of the function). In the return value, the real data is stored in field *content* and the size of data is in field *size*. Type constant `D_BD` is the bound of the data size; in the example, it is set to **uint8**, meaning the data is maximally 256 bytes. Here if the payload *content* turns out to be shorter than 16 bytes, a more compact message format is used by specializing the *send* function with *dlen_t*. Here the **tlet** expression is used to compute *dlen_t*. Unlike previous uses of **tlet** expressions that are nothing but convenient type abbreviations which could have been inlined, the **tlet** expression plays a critical role in

[4] In real-world WSN programs, 1-hop broadcasting does not need to specify a destination address. The example here is used to demonstrate language features, and in the real world, this programming pattern is still useful when polling all neighbors one by one with neighbor-specific requests.

```
NODE_NUM = 0xFFFF;
  main = let tp = topology () in
         let ctp = coloring tp in
         for(uint16 i = 0; i < NODE_NUM; i++){
           let cnum = tp[i].nsize in
           tlet addr_t ≼ uint16 = if (cnum <= 0xF) then uint4
                                  else if (cnum <= 0xFF) then uint8
                                  else uint16 in
           let p = (noiseRate () > 0.5) in
           let self = lift (addr_t)(ctp[i].color) in
           let nb = ctp[i].cneighbors in
           run (mote p addr_t self nb cnum)
         }
```

Fig. 17 Bootstrapping Code for Hub

the fundamentally dynamic nature of *dlen_t*: the latter is given the bound `D_BD`, but its concrete value is unknown until run time.

Process separation is illustrated here with use of **lift** to move the value of a meta-stage destination address to the object stage, via the **lift** $neighbors[i]$ expression.

7.4 A Metaprogram on the Hub

With a specializable mote code *mote* defined, Fig. 17 gives the bootstrapping code to be executed on the hub, with the primary goal of specializing the code produced by *mote* and deploying it. The general idea here is the hub will first execute function

$$topology :: () \to node\_t[\texttt{NODE\_NUM}]$$

where

$$node\_t = \left\{ \begin{array}{l} id : \textbf{uint64}; \\ neighbors : \textbf{uint64}[\texttt{NODE\_NUM}]; \\ nsize : \textbf{uint16} \end{array} \right\}$$

to obtain the global connectivity graph of the initially deployed sensor network, and store the result in a hub data structure (the *tp* variable in the example). We omit the definition of this function here. The only implementation detail that is related to the discussion here is the computed graph is represented as an array, each element of which stores the network-layer identifier represented as an eight-byte integer (the *id* field), the network-layer identifiers of its neighbors (the *neighbors* field), and the number of neighbors (the *nsize* field). This data structure may be large, but note that it is kept on the hub only – a resource-rich computer. Next, the hub invokes an effectful function

$$coloring :: node\_t[\texttt{NODE\_NUM}] \to cnode\_t[\texttt{NODE\_NUM}]$$

where

$$cnode\_t = \left\{ \begin{array}{l} color : \textbf{uint16}; \\ cneighbors : \textbf{uint16}[\texttt{NODE\_NUM}] \end{array} \right\}$$

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 to guarantee adjacent nodes are rendered with different

colors. We omit this classic algorithm here, and only note that the function computes an array that index-wise aligns with its argument array, with each element recording the color computed by the coloring algorithm (the *color* field), and the neighbors as represented in colors (the *cneighbors* field). The color is represented as a **uint16** integer, as in the worst case the number of colors is the same as the number of nodes.

The primary goal of specialization is to specialize the type of addresses. This is illustrated by *addr_t*, which can be either **uint4**, **uint8**, or **uint16** based on the number of neighbors a particular mote has, as indicated by *cnum*. The **tlet** expression is again used to compute the *addr_t* type that is more compact for the specific deployment environment. Just like the previous *dlen_t* example, the concrete value of *addr_t* is unknown until run time. Auxiliary function *noiseRate* (omitted here) is used to compute the environmental noise levels, to determine whether parity is needed or not.

Finally, the **run** expression deploys and bootstraps the specialized code (*mote p addr_t self nb cnum*) on motes. 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.

8 Related Work

While this paper contains several novel contributions such as the combination of $F_{\leq}$ and λ^{-}_{ω} for staging and the use of SPS in the sensor programming context, it builds on many threads of related work. The general topic of type specialization and program staging as a means to obtain program efficiency is a common theme across many projects in the program staging and partial evaluation literature. Several authors have explored the interaction of program staging and type dependence to support compiler construction [5] and interpreters [45]. Also related is work on program generation formalisms for compiler construction that leverage first-class types and intensional polymorphism [14]. Tempo [11] is a related system that integrates partial evaluation and type specialization for increasing efficiency of systems applications. Perhaps the system most closely related to ours is Monnier and Shao's [40], where type abstraction as a language construct is supported in a staged program calculus albeit following a standard System $F_{\leq}$ style (i.e. it lacks any λ_{ω} dimension of expressiveness). Kameyama et al. have studied staging in the presence of side effects as a way to optimize algorithms that exploit mutation [28]. Moggi and Fagorzi have established a monadic foundation for integrating staging with arbitrary side effects in a general and mathematically rigorous fashion [38].

Type-safe code specialization has been the focus of MetaML [56,39] and its more recent and robust implementation, MetaOCaml [54]. MetaML also enjoys type safety results of the sort presented here [52]. 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 [15], higher-order abstract syntax [58], and first-order abstract syntax with de Bruijn indices [9]. 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 arise in our system, simplifying our formal development. The added expressiveness of MetaML here comes at the price of having to deal with significant additional type system complexities [42,9,6,55]. We have thus far not found this added expressiveness useful for sensor network programming.

MetaML has also been promoted as an effective foundation for embedded systems programming [51]. The GeHB language is a proposal for a type-safe two-stage language for programming embedded systems [53]. The focus of this work is on how resource bounds can be guaranteed for the embedded systems code via a type system. The work in [29] also shows how type-safe staged programming can be effectively applied to circuit generation languages, and other recent work shows the utility of staging for programming hardware descriptions [23,47]. While these works do not directly overlap with ours, they provide additional supporting evidence of the utility of the general approach.

Parametric customization of type annotations is widely used today; examples include C++ templates and Java generics. The formal foundations for Java generics are the parametric type systems System F and $F_{\leq}$ [8], and our parameterized type syntax over an underlying subtyping relation 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.

Module systems are primarily designed for separate compilation and sound linking [7] – goals we are not focusing on here – but there are nonetheless some structural analogies between our work and module systems. $\langle \text{ML} \rangle$ is most closely related to module systems supporting both types and values as module components [24,34,32,18] since we analogously support types as data in our meta-stage. A "module" entity in our language is approximately equivalent to meta code of the form:

$$\textbf{tlet } ty = \tau \textbf{ in } \{sometype = ty; \langle \{somefun = \ldots; somestate = \textbf{ref } \ldots\} \rangle\}$$

There is a well-known implicit staging of module definitions into compile-time and run-time components of composition [24], and that staging is explicit in $\langle \text{ML} \rangle$. This can be seen in the above example, where the type portion is at the meta-stage and the non-type portion $\langle \ldots \rangle$ is at the object stage (but can refer to the type *ty* by our scoping rules). A "functor"-like object in $\langle \text{ML} \rangle$ is simply a meta-code λ-abstraction taking and returning such an entity. More explicit staging could potentially be a useful approach for future module system designs; for example, some of the thorny technical problems in module system designs [31,16,16] arise from implicit cross-stage interactions.

The notion of type computation is expressed in pure form in λ_ω, which is a calculus formalized as part of the λ cube [2] and allows types to depend on types. In λ_ω, type constructors are endowed with a semantics that is essentially a simply typed λ calculus over types-as-expressions. Both the Calculus of Constructions [12] and Martin Löf type theory [37] subsume λ_ω, thus so do the Coq [13] and Agda [3] frameworks which are derived from these respective theories. Coq and Agda are on the cutting edge of language theory and practice, and thus illustrate the usefulness of type computations. Our application space only needs a first-order form of λ_ω that we term $\lambda_{\underline{\omega}}$, and both Coq and Agda subsume expressivity we do not consider here, in particular type dependence (formalized on the λ cube as λ_P). Also, Coq and Agda are functional languages which are arguably too far from the imperative core needed for systems programming. Note that recent work such as Ynot [43] has helped bridge this gap, and we believe that incorporating program specifications in a Coq or Agda style is an interesting direction for future work. A number of recent advances related to type dependence in practice would support and inspire such an effort [59,10,4,19]. Runtime type information has been successfully used for the special case of a decidable type system for specializing types of polymorphic functions [25], 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. Flask [36] applies metaprogramming to sensor networks. The main motivation of designing Flask is to allow FRP-based [57] 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 $\sim e$. Since pre-computing stream combinators is the main goal of Flask, the focus of our language – computing and checking precise *type annotations* inside the object stage code at meta stage – is not a topic they focus on. In particular, at the Flask meta stage a monadic encoding is used to perform dynamic type checking of residual object code, in contrast to $\langle$ML$\rangle$ where object code enjoys static type safety at the meta stage. Also, Flask is a "macroprogramming" language, where global network behavior is specified in code, and node-level behavior is then deduced and encoded by the compiler. In contrast, our system is intended for "microprogramming", i.e. direct programming of individual nodes, albeit with the programmatic orchestration capabilities of metaprogramming.

9 Conclusion

This paper has shown how the combination of type genericity and subtyping *à la* System $F_{\leq}$, extended with a limited form of type computation *à la* λ_ω, yields important improvements in expressiveness for staged programming. The particular example we focus on here is the case of wireless sensor network programming; in that scenario there is no need for complex cross-stage persistence (CSP); instead, a model of staged process separation (SPS) is more relevant and appropriate. We have demonstrated type safety and soundness of decidable type checking for $\langle$ML$\rangle$, establishing a solid theoretical foundation for our model.

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 [21], and in directly implementing them in a functional language setting. The second route may appear difficult since a functional language with first-class functions and dynamic allocation generally has more runtime overhead than typical sensor networks expect. But this 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 [44] and Flask [36], designed for sensor networks. Competitive performance can be achieved by placing restrictions on the mote language; for example, only statically bounded recursion is allowed in Regiment.

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,

⟨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 163–173, 1994.
2. H Barendregt. Introduction to generalized type systems. *Journal of functional programming*, 1(2):125–154, 1991.
3. Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer Berlin / Heidelberg, 2009.
4. Edwin Brady. Irdis: a language with dependent types. `http://www.idris-lang.org`, 2009.
5. Edwin Brady and Kevin Hammond. A verified staged interpreter is a verified compiler. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 111–120, New York, NY, USA, 2006. ACM.
6. 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.
7. Luca Cardelli. Program fragments, linking, and modularization. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, 1997.
8. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.
9. Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. In *ICFP'03*, 2003.
10. Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 66–77, New York, NY, USA, 2005. ACM.
11. C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé. Tempo: specializing systems applications and beyond. *ACM Comput. Surv.*, page 19, 1998.
12. T Coquand and G Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.
13. T Coquand and G Huet *et al*. The Coq proof assistant. `http://coq.inria.fr`.
14. Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. *J. Funct. Program.*, 12(6):567–600, 2002.
15. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001.
16. Derek Dreyer. A type system for well-founded recursion. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 293–305, 2004.
17. Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Language Semantics (MFPS)*, volume 1 of *Electronic Lecture Notes in Computer Science*, 1995.
18. Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 236–248, 1998.
19. Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: indexed types now! In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '07, pages 112–121, New York, NY, USA, 2007. ACM.

20. 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.
21. 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.
22. Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193(1-2):75 – 96, 1998.
23. Jennifer Gillenwater, Gregory Malecha, Cherif Salama, Angela Yun Zhu, Walid Taha, Jim Grundy, and John O'Leary. Synthesizable high level hardware descriptions: using statically typed two-level languages to guarantee verilog synthesizability. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '08, pages 41–50, New York, NY, USA, 2008. ACM.
24. Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *In Seventeenth ACM Symposium on Principles of Programming Languages*, pages 341–354. ACM Press, 1990.
25. 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.
26. 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.
27. Furio Honsell, Ian A. Mason, Scott Smith, and Carolyn Talcott. A variable typed logic of effects. *Information and Computation*, 119:55–90, 1993.
28. Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage: staging with delimited control. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 111–120, New York, NY, USA, 2009. ACM.
29. Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *Proceedings of the 4th ACM international conference on Embedded software*, EMSOFT '04, pages 249–258, New York, NY, USA, 2004. ACM.
30. Shriram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. From macros to reusable generative programming. In *In International Symposium on Generative and Component-Based Software Engineering, number 1799 in Lecture Notes in Computer Science*, pages 105–120. Springer-Verlag, 1999.
31. Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *POPL*, pages 142–153, 1995.
32. Xavier Leroy. A modular module system. *J. Funct. Program.*, 10(3):269–303, 2000.
33. Yu David Liu, Christian Skalka, and Scott Smith. Type-specialized staged programming with process separation. In *Workshop on Generic Programming (WGP09)*, Edinburgh, Scotland, 2009.
34. D. MacQueen. Modules for Standard ML. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 409–423, 1984.
35. 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.
36. 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.
37. P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. F. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118, Amsterdam, 1973. North-Holland.
38. Eugenio Moggi and Sonia Fagorzi. A monadic multi-stage metalanguage. In Andrew D. Gordon, editor, *FoSSaCS*, volume 2620 of *Lecture Notes in Computer Science*, pages 358–374. Springer, 2003.
39. 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.

40. Stefan Monnier and Zhong Shao. Inlining as staged computation. *J. Funct. Program.*, 13(3):647–676, 2003.
41. 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.
42. 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.
43. Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *ICFP '08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, September 2008.
44. Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498, 2007.
45. Emir Pasalic, Walid Taha, Tim Sheard, and Tim S. Tagless staged interpreters for typed languages. In *In the International Conference on Functional Programming (ICFP'02)*, pages 218–229. ACM, 2002.
46. François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.
47. Cherif Salama, Gregory Malecha, Walid Taha, Jim Grundy, and John O'Leary. Static consistency checking for verilog wire interconnects: using dependent types to check the sanity of verilog descriptions. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '09, pages 121–130, New York, NY, USA, 2009. ACM.
48. Curt Schurgers, Gautam Kulkarni, and Mani B. Srivastava. Distributed on-demand address assignment in wireless sensor networks. *IEEE Trans. Parallel Distrib. Syst.*, 13(10):1056–1065, 2002.
49. 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.
50. Walid Taha. A gentle introduction to multi-stage programming. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 30–50. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-25935-0 3.
51. Walid Taha. Resource-aware programming. In Zhaohui Wu, Chun Chen, Minyi Guo, and Jiajun Bu, editors, *ICESS*, volume 3605 of *Lecture Notes in Computer Science*, pages 38–43. Springer, 2004.
52. Walid Taha, Zine el-abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type safety (extended abstract). In *In 25th International Colloquium on Automata, Languages, and Programming*, pages 918–929. Springer-Verlag, 1998.
53. Walid Taha, Stephan Ellner, and Hongwei Xi. Generating heap-bounded programs in a functional setting. In *In EMSOFT*, pages 340–355. Springer, 2003.
54. Walid Taha and *et. al.* MetaOCaml: A compiled, type-safe multi-stage programming language. `http://www.metaocaml.org/`.
55. Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *POPL'03*, 2003.
56. 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.
57. 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.
58. 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.
59. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 214–227, New York, NY, USA, 1999. ACM.