A Type System for Resource Protocol Verification and its Correctness Proof

Corneliu Popeea and Wei-Ngan Chin
Department of Computer Science
School of Computing, National University of Singapore
{popeeaco,chinwn}@comp.nus.edu.sg

ABSTRACT

We present a new method, based on a form of dependent typing, to verify the correct usage of resources in a program. Our approach allows complex resources to be specified, whose properties are captured by annotated types and conditions on *invariance* and *final states*. The protocol itself is specified through a set of pre-defined methods, whose pre-condition and post-condition together, enforce the correct temporal usage of each resource type. We design a simple language together with a type system that shows how resource protocol verification can be achieved. We formalise an operational semantics for the language and provide a correctness proof which confirms that well-typed programs conform to the specified protocol of each resource type.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification - Correctness proofs; D.3.3 [**Programming Languages**]: Language Constructs and Features - Abstract data types; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs - Pre- and post-conditions; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages - Program analysis

General Terms

Languages, Verification, Theory

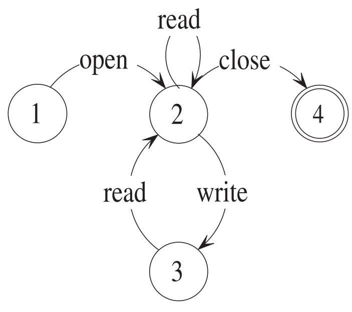
Keywords

Resource Specification, Protocol Verification, Dependent Type System, Path-sensitive Analysis, Correctness Proof

1. INTRODUCTION

In recent years, there have been increasing interests on tools (and techniques) that could verify important safety properties of software, for the purpose of eliminating bugs at compile-time [12, 11, 4, 13, 14]. An important domain for such software verification is in the realm of protocols for resources (such as those of device-drivers or memory management) where bugs could have a crippling effect on systems software.

An example of protocol (expressed as a finite automata) is shown in figure 1(a) for a file system with audit checks. Each file has a state and must be opened before it is accessed via either read or write. Each write operation must be followed by a read operation for the purpose of verifying the previous write. At the end of the operations, we expect each file to be closed, with a node (labeled 4) to denote this final state.



(a)

$$
\begin{array}{l}
\texttt{resource File}\langle\texttt{s}\rangle\ \texttt{st}\ (1 \leq \texttt{s} \leq 4),\ (\texttt{s} = 4),\ \{\texttt{s}\} \\
\texttt{newFile} :: () \rightarrow \texttt{File}\langle\texttt{s}\rangle\ \texttt{st}\ (\texttt{s} = 1) \\
\texttt{open} :: \texttt{File}\langle\texttt{s}\rangle \rightarrow ()\ \texttt{st}\ (\texttt{s} = 1 \wedge \texttt{s}' = 2) \\
\texttt{close} :: \texttt{File}\langle\texttt{s}\rangle \rightarrow ()\ \texttt{st}\ (\texttt{s} = 2 \wedge \texttt{s}' = 4) \\
\texttt{read} :: \texttt{File}\langle\texttt{s}\rangle \rightarrow \texttt{Int}\langle\texttt{r}\rangle\ \texttt{st}\ (\texttt{s} = 2 \vee \texttt{s} = 3) \wedge \texttt{s}' = 2 \\
\texttt{write} :: (\texttt{File}\langle\texttt{s}\rangle, \texttt{Int}\langle\texttt{r}\rangle) \rightarrow ()\ \texttt{st}\ (\texttt{s} = 2 \wedge \texttt{s}' = 3) \\
\texttt{getState} :: \texttt{File}\langle\texttt{s}\rangle \rightarrow \texttt{Int}\langle\texttt{n}\rangle\ \texttt{st}\ (\texttt{n} = \texttt{s} \wedge \texttt{s}' = \texttt{s})
\end{array}
$$

(b)

Figure 1: Audit file protocol

Various approaches, based on static analysis [9, 4, 19], have been advocated for ensuring that user programs satisfy stated resource protocols. Of particular interests are type-based approaches [11, 13] where users are expected to provide type annotations that can assist in the verification process. However, current type-based approaches suffer from two main problems. Firstly, they are usually path-insensitive, as they do not take the conditions of branches into account. Secondly, only simple resources with finite states have been considered.

PEPM'04, August 24–26, 2004, Verona, Italy.

In this paper, we propose the use of an advanced type system, based on dependent typing, to model both resource protocols and the verification of user programs that use them. In the case of audit file, an annotated type $\mathtt{File}\langle\mathtt{s}\rangle$ denotes the state of its resource, while its protocol can be specified through dependent types for each of its primitive operations, as shown in figure 1(b). This essentially captures the pre-condition and post-condition of each method, allowing user programs to be checked for protocol conformance. Take note that we use the prime notation to capture the post-state. For example, the size variables $\mathtt{s}$ and $\mathtt{s}'$ in the constraint of a given method will denote the state of a file of type $\mathtt{File}\langle\mathtt{s}\rangle$, *before* and *after* the execution of the method. In the case of `open`, we require the pre-state of the file be $\mathtt{s} = 1$ and the post-state to be $\mathtt{s}' = 2$. For the `getState` primitive, the integer output captures the state of the file via $\mathtt{n} = \mathtt{s}$. As this is a query, the state of the file is unchanged, denoted explicitly by $\mathtt{s}' = \mathtt{s}$. Note that primed versions of size variables are not needed for output (e.g. $\mathtt{File}\langle\mathtt{s}\rangle$ of `newFile`) nor immutable parameter values (e.g. $\mathtt{Int}\langle\mathtt{r}\rangle$ of `write`), as these values have only a single state each.

The main motivation for proposing such a type system is to provide an *expressive* and *precise* means for specifying resource protocols and to verify that programs which use these resources conform to the specified protocols. We shall highlight a correctness proof which shows that such a verification method is sound with respect to an operational semantics for our programs. Our main contributions are:

- **High precision:** We propose a new approach to protocol specification and usage verification, based on a dependent type system that is *context*-, *flow*- and *path-sensitive*. Together with a relational size analysis in the Presburger arithmetic domain, they add considerably to the precision of the proposed verification process.
- **Resources as ADTs:** We model each resource (and its protocol) as an abstract data type (ADT) with a size-annotated type, together with corresponding conditions for *invariance*, *finality* and *mutability*. This specification allows both simple and complex resource types to be elegantly expressed.
- **Soundness:** Our type system has been proven sound. We provide an operational semantics for our language and prove that each well-typed program never violates the protocols of the resources used.

The remainder of this paper is organized as follows. Section 2 elaborates on a specification mechanism for modelling each resource type and its protocol, followed by a simple alias-free language which we have adopted for our study. Section 3 proposes a set of type rules that can be used to verify user programs to ensure the correct usage of each resource type in accordance with the specified protocols. The semantics of our language is introduced in section 4. In section 5, we highlight the correctness of our type rules by proving that each well-typed program is guaranteed to be free of protocol errors. Section 6 discusses how aliasing of resources can be handled, while section 7 presents some related works. We provide some concluding remarks in the last section.

2. RESOURCE SPECIFICATION

We propose to model each resource as an ADT with a set of pre-defined methods. These methods may change the state of their resources, and must be executed according to given resource protocols. Several aspects of each resource type and its protocol can be captured, including:

- resource *invariance* that has to be maintained at all times.
- resource *finality* that has to be satisfied whenever a given resource becomes inaccessible (dead).
- method *pre-condition* which captures the requirement *before* each method invocation.
- method *post-condition* which captures the expected state *after* each method invocation.

Each new resource type can be specified using the following construct:

$$\textbf{resource}\ r\langle n_{1..p}\rangle\ \textbf{st}\ \phi_{inv},\ \phi_{final},\ \mathit{ISet}$$

where $\{n_{1..p}\}$ denotes a non-empty set of integer-valued size variables representing the state for resource type r, while ϕ_{inv} and ϕ_{final} are the invariance and finality constraint for each object of the resource type. The invariance condition essentially limits the allowable state of each resource type, while the finality condition captures a mandated state of a resource prior to its disposal. Also, *ISet* denotes a set of size variables that are subject to imperative changes. Conversely, $(\{n_{1..p}\} - \mathit{ISet})$ denotes the set of immutable size-variables whose values do not change.

```
resource Lock⟨s⟩ st (0 ≤ s ≤ 1), (s = 0), {s}
newLock :: Int⟨i⟩ → Lock⟨s⟩ st (s = 0)
lock :: Lock⟨s⟩ → () st (s = 0 ∧ s′ = 1)
unlock :: Lock⟨s⟩ → () st (s = 1 ∧ s′ = 0)
getState :: Lock⟨s⟩ → Int⟨n⟩ st (s′ = s ∧ n = s)
```

(a)

```
resource Buffer⟨s,c⟩ st (0 ≤ s ≤ c ∧ c > 0), (s = 0), {s}
newBuffer :: Int⟨n⟩ → Buffer⟨s,c⟩ st (n > 0 ∧ c = n ∧ s = 0)
get :: Buffer⟨s,c⟩ → Int⟨n⟩ st (s > 0 ∧ s′ = s − 1)
add :: (Buffer⟨s,c⟩, Int⟨n⟩) → () st (s < c ∧ s′ = s + 1)
getNum :: Buffer⟨s,c⟩ → Int⟨n⟩ st (n = s ∧ s′ = s)
getCap :: Buffer⟨s,c⟩→ Int⟨n⟩ st (n = c ∧ s′ = s)
```

(b)

```
resource Array⟨s⟩ st (s > 0), True, {}
newArray :: Int⟨n⟩ → Array⟨s⟩ st (n > 0 ∧ s = n)
assign :: (Array⟨s⟩, Int⟨i⟩, Int⟨k⟩) → () st (0 ≤ i < s)
get :: (Array⟨s⟩, Int⟨i⟩) → Int⟨k⟩ st (0 ≤ i < s)
length :: Array⟨s⟩ → Int⟨k⟩ st (k = s)
```

(c)

Figure 2: Resource protocol specifications

An example of resource declaration is the *mutex* lock type, declared in figure 2(a). Take note that each lock's state could either be 0 (unlocked) or 1 (locked). Its sole size-variable $\mathtt{s}$ may be changed. The finality constraint $(\mathtt{s} = 0)$ ensures that the lock is released in the end. Associated with this resource declaration are four predefined methods. Both the `lock` and `unlock` operations require each mutex lock to be in the opposite state of its intended operation. The post-conditions mirror the flip operations performed. We also provide a query function, `getState`, which allows programmers to determine the status of a given mutex. Lastly, `newLock` returns a new mutex (for a specified identifier) in the unlocked state.

Our next example is a more sophisticated buffer resource, specified in figure 2(b). In general, this resource cannot be captured using a finite state model, as the number of states is dependent on its capacity (which is unknown at compile-time). Two size variables, `c` and `s`, are used to denote respectively, the *capacity* and *current size* of the buffer. Its invariance, namely ($0 \leq s \leq c \wedge c > 0$), is guaranteed at all times. Before each buffer becomes inaccessible, we require it be cleared with ($s = 0$) as its finality constraint. Lastly, `s` is imperative, while `c` is immutable. The given set of predefined functions must ensure that the invariance of each buffer is maintained at all times.

Our specification mechanism for resource protocol using ADT is quite general. Though state changes are typically expected for resource protocols, our approach also supports as a special case resource protocol whose state never change after it has been created. A well-known example that falls under this degenerate category is the array data type itself. We could model such an array resource by the ADT from figure 2(c), effectively allowing the array bound check safety problem to be treated as a special case of protocol verification.

2.1 Language

The primary focus of this paper is an advanced type system for resource protocol and its corresponding soundness proof. For simplicity, we shall focus on a first-order functional language with resources, called RESFP. Its syntax is given in figure 3.

$$
\begin{array}{rl}
\mathcal{P} ::= & \textit{rdecl}^* \ \textit{pdef}^* \ \textit{fdef}^* \\
\textit{rdecl} ::= & \textbf{resource} \ \textit{rt} \ \textbf{st} \ \phi_{inv}, \ \phi_{final}, \ \{n^*\} \\
\textit{pdef} ::= & f :: (t_1, .., t_m) \ \rightarrow \ t \ \textbf{st} \ \phi \\
\textit{fdef} ::= & f :: (t_1, .., t_m) \ \rightarrow \ t \ \textbf{st} \ \phi \ ; \ f(v_1, .., v_m) = e \\
t ::= & \textit{rt} \ | \ b \\
\textit{rt} ::= & r\langle n^+ \rangle \\
b ::= & \textbf{Int}\langle n \rangle \ | \ \textbf{Bool}\langle n \rangle \ | \ \textbf{Void}\langle\rangle \ | \ \textbf{List}\langle n \rangle(b) \\
e ::= & k \ | \ v \ | \ f(v_1, .., v_m) \\
 & | \ \textbf{if} \ v \ \textbf{then} \ e_1 \ \textbf{else} \ e_2 \ | \ \textbf{let} \ v \ = \ e_1 \ \textbf{in} \ e_2 \\
\phi \in & \mathbf{F} \qquad \textit{(Presburger Size Constraint)} \\
\phi ::= & \beta \ | \ \phi_1 \wedge \phi_2 \ | \ \phi_1 \vee \phi_2 \ | \ \neg\phi \ | \ \exists n \cdot \phi \ | \ \forall n \cdot \phi \\
\beta \in & \mathbf{BExp} \qquad \textit{(Boolean Expression)} \\
\beta ::= & \textit{True} \ | \ \textit{False} \ | \ \alpha_1 = \alpha_2 \ | \ \alpha_1 < \alpha_2 \ | \ \alpha_1 \leq \alpha_2 \\
\alpha \in & \mathbf{AExp} \qquad \textit{(Arithmetic Expression)} \\
\alpha ::= & c \ | \ n \ | \ c * \alpha \ | \ \alpha_1 + \alpha_2 \ | \ -\alpha \\
 & \textit{where } c \textit{ is an integer constant} \\
 & \qquad n \textit{ is a size variable} \\
 & \qquad v \textit{ is a program variable}
\end{array}
$$

Figure 3: Syntax for the RESFP language

Each program contains declarations for resources, primitives and user-defined methods. As already illustrated in our examples, each primitive method declaration has the form:

$$f :: (t_1, .., t_m) \ \rightarrow \ t \ \textbf{st} \ \phi$$

where $t_1, .., t_m$ and t are annotated types for parameters and result, respectively. The size constraint ϕ captures both the pre-condition and post-condition of the method.

To support dependent typing, our types and methods are augmented with size variables and size constraints. For size constraint, we restrict it to Presburger form, as decidable (and practical) constraint solvers (e.g. [23]) are available.

Note that the suffix notation y^* denotes a list of zero or more distinct syntactic terms. For convenience, we use () to denote both the value of and as a shorthand for the `Void`⟨⟩ type.

Though functional, the language has imperative effects through its resources. In order to avoid aliasing of resources, new (unique) resources are only returned through primitive methods. Further restrictions to prevent aliasing are enforced by our type system. User-defined methods cannot return values of resource type and we neither allow the same resource to be passed to different parameters of each method call, nor allow a resource variable to be used otherwise than as a method parameter.

Alias analysis can often be supported separately as an add-on module. An initial proposal on how to handle alias analysis is discussed in section 5.

The RESFP language can be extended with syntactic abbreviations to make programming more convenient. Some examples of equivalences are shown below.

$$
\begin{array}{l}
m(e_1, \ldots, e_n) \equiv \\
\quad \textbf{let} \ v_1 = e_1 \ \textbf{in} \ (\cdots (\textbf{let} \ v_n = e_n \ \textbf{in} \ m(v_1, \ldots, v_n)) \cdots) \\
\\
\textbf{if} \ e_1 \ \textbf{then} \ e_2 \ \textbf{else} \ e_3 \equiv \\
\quad \textbf{let} \ v = e_1 \ \textbf{in} \ (\textbf{if} \ v \ \textbf{then} \ e_2 \ \textbf{else} \ e_3) \\
\\
e_1 \ ; e_2 \equiv \quad \textbf{let} \ v = e_1 \ \textbf{in} \ e_2
\end{array}
$$

2.2 Path-Sensitive Relational Analysis

The use of Presburger arithmetic with integer domain has several benefits. Firstly, it provides for a uniform and consistent way to capture both the states of resources and values of program variables. Both bounded and unbounded states/values could be captured by the integer domain. For example, the boolean value, denoted by the annotated type `Bool`⟨`b`⟩, could have its domain bounded by the size constraint $0 \leq b \leq 1$; with $b{=}0$ to denote `False` and $b{=}1$ to denote `True`.

Capturing all states/values in the same integer domain also allows us to express relational analysis in a straightforward way. For example, the latest state of a file `File`⟨`s`⟩ may be tightly coupled with an integer program variable `Int`⟨`i`⟩ via a disjunctive formula $(s'{=}4 \wedge i{=}-1) \vee (s' {\neq} 4 \wedge i {\geq} 0)$. Disjunctive formula may arise from different branches of conditions. Such relations are directly supported by the Presburger arithmetic form.

Presburger formula also allows us to support path-sensitive analysis where each path is marked by the boolean values of the tests from conditionals taken. This correlation of program states with paths from conditional construct supports more precise relational analysis. It also allows infeasible paths to be identified whenever the size constraint evaluates to `False`.

These features of Presburger form in integer domain allow more precise program states to be captured. As we have seen with the buffer example, we are able to model the protocols of resources with *unbounded* symbolic states. Let us look at two other examples, involving conditional and recursion, that will help reiterate the utility of path-sensitive relational analysis via dependent typing.

Consider a function `f` that takes a flag and a mutex lock, as shown in figure 4(a). Based on the value of the flag, it either performs a locking or does nothing. To capture this path-sensitive behaviour, we can declare the type for `f`, as follows:

$$
\begin{array}{l}
\texttt{f} :: (\texttt{Bool}\langle \texttt{b} \rangle, \texttt{Lock}\langle \texttt{s} \rangle) \rightarrow \ () \\
\qquad \texttt{st} \ (\texttt{b} = 1 \wedge \texttt{s} = 0 \wedge \texttt{s}' = 1) \vee (\texttt{b} = 0 \wedge \texttt{s}' = \texttt{s})
\end{array}
$$

Take note that the *disjunction* captures two cases, namely: (i) when flag is true and the lock operation is performed, (ii) when flag is false and no operation is performed. Each use of this `f` function

```
f(flag, r) =   if flag then lock(r)
               else ()
```

(a)

```
addMany(b, n, val) =
    if n ≤ 0 then ()
    else
      add(b, val);
      addMany(b, n−1, val)
```

(b)

Figure 4: Two user functions

is expected to satisfy its pre-condition, which can be obtained by quantifying (existentially) the size variables of the result and post-states. For example:

$$\begin{array}{l}\exists s' \cdot ((b=1 \wedge s=0 \wedge s'=1) \vee (b=0 \wedge s'=s)) \\ \quad \equiv \ (b=1 \wedge s=0) \vee b=0\end{array}$$

Recursive functions (and therefore loops) may also be handled in a path-sensitive manner by our proposed dependent type system. In this case, the pre-condition to recursive functions must be strong enough to ensure that all calls to each resource primitive be safe. Consider the function `addMany` in figure 4(b), which would add an integer value `n` times to a given buffer. To ensure that all `add` primitive calls are safe, we require the following dependent type for `addMany`.

$$\begin{array}{l}\texttt{addMany} :: (\texttt{Buffer}\langle \texttt{s},\texttt{c}\rangle, \texttt{Int}\langle \texttt{n}\rangle, \texttt{Int}\langle \texttt{v}\rangle) \rightarrow \ () \\ \quad\quad \texttt{st} \ (\texttt{s}+\texttt{n} \leq \texttt{c} \wedge \texttt{s}' = \texttt{s}+\texttt{n})\end{array}$$

The pre-condition for this function, namely $(\texttt{s}+\texttt{n} \leq \texttt{c})$, ensures that we have enough space to add a given value `n` times into the buffer. Correspondingly, $(\texttt{s}' = \texttt{s}+\texttt{n})$ indicates a precise post-state after each successful execution of the method. Let us now look at the type rules that can help verify if user programs conform to stated resource protocols.

3. TYPE SYSTEM

There are two key areas to verify for resource protocols and their user-programs, namely:

- *Protocol verification* to ensure that each resource protocol (comprising a set of predefined methods) satisfies some stated resource properties (e.g. invariance, liveness, fairness, etc).
- *Usage verification* to ensure that user program uses the resources in accordance to the respective protocols.

Traditionally, much interest have been devoted to certifying that protocols meet some stated properties[12, 1]. In recent years, new techniques [13, 11] have been developed also for the latter. This paper is mostly concerned with usage verification, as we treat each resource as an ADT. However, as we shall see later (in the type rule for primitives), we do in fact also verify the protocols themselves by checking that the respective resource invariance is maintained by the post-conditions of each predefined method.

We shall present the proposed protocol verification mechanisms in a type-checking framework whereby type annotations are supplied at method boundary.

3.1 Notations

We begin with a review of some notations used. Let us define V to return all free size variables in a formula. For example, $V(x'=z+1 \wedge y=2) = \{x,y,z\}$. We also extend the definition of V to annotated type, as well as type environment.

The function *prime* takes a set of size variables and returns their primed version. For example, $\mathit{prime}(\{s_1,..,s_n\}) = \{s'_1,..,s'_n\}$. We extend this to apply to annotated type (and type environment) by replacing their imperative size variables with primed counterparts, as follows: $\mathit{prime}(t) = \rho\, t$ where $\rho = [s \mapsto s' \mid s \in \mathcal{I}(t)]$ and $\mathcal{I}(t)$ denotes the set of imperative size variables from the type t. The function *prime* is also defined for substitutions as follows:

$$\mathit{prime}[x \mapsto a, y \mapsto b] = [x' \mapsto a', y' \mapsto b']$$

Often, we need to express a no-change condition on a set of imperative size variables. We define a $\mathit{no}\mathcal{X}$ operation as follows which returns a formula for which the original and prime variables are made equal.

$$\mathit{no}\mathcal{X}(\{\}) =_{df} \mathit{True} \qquad \mathit{no}\mathcal{X}(\{x\} \cup X) =_{df} (x'=x) \wedge \mathit{no}\mathcal{X}(X)$$

This is extended to types using $\mathit{no}\mathcal{X}(t) =_{df} \mathit{no}\mathcal{X}(\mathcal{I}(t))$.

We introduce a sequential composition operation, $\Delta \circ_X \phi$, to capture a size constraint Δ that is being composed with an incremental change ϕ where $X = \{s_1,..,s_n\}$ is a set of size variables that are being changed. This operation can be formally defined as follows:

$$\begin{array}{l}\Delta \circ_X \phi =_{df} \exists D \cdot \rho'(\Delta) \wedge \rho(\phi) \\ \text{where} \quad D = \{r_1,..,r_n\} \text{ are new size variables} \\ \qquad\quad \rho = [s_i \mapsto r_i]_{i=1}^n \ ; \ \rho' = [s'_i \mapsto r_i]_{i=1}^n\end{array}$$

An example of sequential composition is considered next. Assuming that the current size constraint is $(x'=5 \wedge z'=x+6)$, and the incremental change which affects variables $\{x,y\}$ is $(y'=x+1 \wedge x'=10)$, we can obtain the updated size constraint via :

$$\begin{array}{l}(x'=5 \wedge z'=x+6) \circ_{\{x,y\}} (y'=x+1 \wedge x'=10) \\ \quad \equiv \exists x_0, y_0 \cdot (x_0=5 \wedge z'=x+6) \wedge (y'=x_0+1 \wedge x'=10) \\ \quad \equiv (z'=x+6 \wedge y'=5+1 \wedge x'=10)\end{array}$$

3.2 Type Rules for Verification

Each program $\mathcal{P}$ consists of declarations for resources, primitive methods and user-defined methods. The program judgement $\vdash_{prog} \mathcal{P}$ (depicted in figure 5) checks respectively the resource declarations, primitive methods and user-defined methods by using the following three judgements:

$$\begin{array}{l}\vdash_{res}\ \mathit{rdecl}_i, \ \ i \in \{1..r\} \\ \vdash_{prim}\ \mathit{pdef}_i, \ \ i \in \{1..p\} \\ \vdash_{meth}\ \mathit{fdef}_i, \ \ i \in \{1..q\}\end{array}$$

The rule [RES] for resource declaration checks that only the size variables of the declared resource are used in its three components, namely invariance, finality and set of imperative size variables. We also provide the following functions to extract these components:

$$\frac{}{\mathit{inv}(b) =_{df} \mathit{True}} \qquad \frac{\begin{array}{c}\textbf{resource}\ r\langle m_{1..p}\rangle\ \textbf{st}\ \phi_{inv},\ \phi_{final},\ \mathit{ISet} \in \mathcal{P} \\ \rho = [m_i \mapsto n_i]_{i=1}^p\end{array}}{\mathit{inv}(r\langle n_{1..p}\rangle) =_{df} \rho\, \phi_{inv}}$$

$$\frac{}{\mathit{final}(b) =_{df} \mathit{True}} \qquad \frac{\begin{array}{c}\textbf{resource}\ r\langle m_{1..p}\rangle\ \textbf{st}\ \phi_{inv},\ \phi_{final},\ \mathit{ISet} \in \mathcal{P} \\ \rho = [m_i \mapsto n_i]_{i=1}^p\end{array}}{\mathit{final}(r\langle n_{1..p}\rangle) =_{df} \rho\, \phi_{final}}$$

$$\frac{}{\mathcal{I}(b) =_{df} \{\}} \qquad \frac{\begin{array}{c}\textbf{resource}\ r\langle m_{1..p}\rangle\ \textbf{st}\ \phi_{inv},\ \phi_{final},\ \mathit{ISet} \in \mathcal{P} \\ \rho = [m_i \mapsto n_i]_{i=1}^p\end{array}}{\mathcal{I}(r\langle n_{1..p}\rangle) =_{df} \rho\, \mathit{ISet}}$$

$$\frac{\vdash_{res}\ rdecl_i,\ \ i\in\{1..r\} \qquad \vdash_{prim}\ pdef_i,\ \ i\in\{1..p\} \qquad \vdash_{meth}\ fdef_i,\ \ i\in\{1..q\}}{\vdash_{prog}\ rdecl_{1..r}\ pdef_{1..p}\ fdef_{1..q}}\ [\underline{\textbf{PROG}}]$$

$$\frac{X = V(rt) \qquad \mathit{ISet}\subseteq X \qquad V(\phi_{inv})\subseteq X \qquad V(\phi_{final})\subseteq X}{\vdash_{res}\ \textbf{resource}\ rt\ \textbf{st}\ \phi_{inv},\ \phi_{final},\ \mathit{ISet}}\ [\underline{\textbf{RES}}]$$

$$\frac{\begin{array}{cc} X_U = \bigcup_{i=1}^{p}(V(t_i)-\mathcal{I}(t_i))\cup V(t) & V(\phi)\subseteq X_U\cup X_P \\ X_P = \bigcup_{i=1}^{p}(\mathcal{I}(t_i)\cup \mathit{prime}(\mathcal{I}(t_i))) & \psi = \bigwedge_{i=1}^{p}\mathit{inv}(t_i) \\ \psi\wedge\phi \gtrapprox_{\mathcal{I}(t_i)} \mathit{inv}(t_i),\ \ i\in 1..p & \psi\wedge\phi\Rightarrow \mathit{inv}(t)\end{array}}{\vdash_{prim}\ f :: (t_1,..,t_p)\rightarrow t\ \textbf{st}\ \phi}\ [\underline{\textbf{PRIM}}]$$

$$\frac{\begin{array}{cc} \Gamma=\{v_1::t_1,..,v_p::t_p\} & \Delta = \mathit{pre}(f)\wedge \mathit{no}\mathcal{X}(\Gamma) \\ \neg \mathit{isResourceType}(t) & \Gamma;\Delta\vdash e :: t',\Delta_f \\ \vdash t' <: t,\rho & \Delta_f \Rightarrow \rho\,\phi\end{array}}{\vdash_{meth}\ f :: (t_1,..,t_p)\rightarrow t\ \textbf{st}\ \phi; f(v_1,..,v_p)=e}\ [\underline{\textbf{METH}}]$$

$$\frac{\Delta'=\Delta}{\Gamma;\Delta\vdash () :: \mathtt{Void}\langle\rangle,\Delta'}\ [\underline{\textbf{CONS1}}] \qquad \frac{s=\mathit{fresh}() \quad \Delta'=\Delta\wedge(s=n)}{\Gamma;\Delta\vdash n :: \mathtt{Int}\langle s\rangle,\Delta'}\ [\underline{\textbf{CONS2}}]$$

$$\frac{s=\mathit{fresh}() \quad \Delta'=\Delta\wedge(s=1)}{\Gamma;\Delta\vdash \textbf{true} :: \mathtt{Bool}\langle s\rangle,\Delta'}\ [\underline{\textbf{CONS3}}] \qquad \frac{s=\mathit{fresh}() \quad \Delta'=\Delta\wedge(s=0)}{\Gamma;\Delta\vdash \textbf{false} :: \mathtt{Bool}\langle s\rangle,\Delta'}\ [\underline{\textbf{CONS4}}]$$

$$\frac{\begin{array}{cc}\Gamma(v)=t & \neg\mathit{isResourceType}(t) \\ t'=\mathit{fresh}(t) & \phi=\mathit{equate}(t',t)\end{array}}{\Gamma;\Delta\vdash v :: t',\Delta\wedge\phi}\ [\underline{\textbf{VAR}}]$$

$$\frac{\begin{array}{ccc} f :: (t_1,..,t_p)\rightarrow t\ \textbf{st}\ \phi\in\mathcal{P} & t'=\mathit{fresh}(t) & \vdash t_i' <: t_i,\rho_i \\ \Gamma(v_i)=t_i',\ \ i\in 1..p & \multicolumn{2}{c}{\rho=(\rho_i\uplus \mathit{prime}(\rho_i))_{i=1}^{p}\uplus \mathit{rename}(t,t')} \\ X=\bigcup_{i=1}^{p}\mathcal{I}(t_i') & \Delta\gtrapprox_X \rho(\mathit{pre}(f)) & \mathit{distinct}\ \{v_i \mid \mathit{isResourceType}(t_i)\}\end{array}}{\Gamma;\Delta\vdash f(v_1,..,v_p) :: t',\Delta\circ_X \rho(\phi)}\ [\underline{\textbf{MI}}]$$

$$\frac{\begin{array}{ccc}\Gamma;\Delta\vdash e_1 :: t_1,\Delta_1 & t_1'=\mathit{fresh}(t_1) & \vdash t_1 <: t_1',\rho \\ \Delta_1'=\rho^{-1}\,\Delta_1\wedge \mathit{no}\mathcal{X}(t_1') & \multicolumn{2}{c}{Y=V(t_1')\cup \mathit{prime}(\mathcal{I}(t_1'))} \\ \Gamma,v::t_1';\Delta_1'\vdash e_2 :: t_2,\Delta_2 & \multicolumn{2}{c}{\Delta_2\gtrapprox_{\mathcal{I}(t_1')} \mathit{final}(t_1')}\end{array}}{\Gamma;\Delta\vdash \textbf{let}\ v=e_1\ \textbf{in}\ e_2 :: t_2,\exists Y\cdot\Delta_2}\ [\underline{\textbf{LET}}]$$

$$\frac{\begin{array}{c}\Gamma(v)=\mathtt{Bool}\langle b\rangle \\ \Gamma;\Delta\wedge b=1\vdash e_1 :: t_1,\Delta_1 \qquad \Gamma;\Delta\wedge b=0\vdash e_2 :: t_2,\Delta_2 \\ t=\mathit{fresh}(t_1) \qquad \rho_i=\mathit{rename}(t_i,t),\ i=1,2\end{array}}{\Gamma;\Delta\vdash \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 :: t,\ \rho_1\Delta_1\vee\rho_2\Delta_2}\ [\underline{\textbf{IF}}]$$

Figure 5: Type rules

The rule [PRIM] for primitive declaration assumes that the invariant property is present for each parameter at the pre-state (through ψ). It uses this assumption to check that each resource satisfies its invariance at the post-state of the callee. Also we use X_P to represent the set of size variables that are imperative, and X_U for the set of size variables that are either immutable or appear in the return type of the method. The variables used in the size constraint ϕ should be confined to the union of these two sets, as the primed version of the variables from X_U is not allowed. The relation $\gtrapprox_X$ is used to check the validity of some condition, and is defined as follows:

$$\Delta \gtrapprox_X \phi\ =_{df}\ (\Delta\Rightarrow\rho\phi)$$

where

$$\rho=[s_1\mapsto s_1',..,s_n\mapsto s_n'] \text{ and } X=\{s_1,..,s_n\}.$$

Take note that the operator $\gtrapprox_\emptyset$ is equivalent with the usual logical implication: $\Delta\gtrapprox_\emptyset \phi\ =_{df}\ (\Delta\Rightarrow\phi)$.

The rule [METH] for method declaration first builds Δ, an initial size constraint that is used to derive Δ_f as the post-state of the method body. The initial size constraint contains the assumption that the pre-condition of the current method is satisfied. Subsequently, the post-state that is derived must satisfy the declared size constraint, ϕ, of its method. The pre-condition of each method can be extracted by the function *pre*, by existentially quantifying both primed and size variables that appear in the return type.

$$\frac{f :: (t_1,..,t_p)\rightarrow t\ \textbf{st}\ \phi\in\mathcal{P} \qquad X=\bigcup_{i=1}^{p}\mathit{prime}(\mathcal{I}(t_i))\cup V(t)}{\mathit{pre}(f)=_{df}\exists X\cdot\phi}$$

As described earlier, we avoid aliasing by imposing restrictions on the usage of resource variables. These restrictions are enforced in the type system by the function *isResourceType*. We also make use of a subtype relation $\vdash t_1 <: t_2,\rho$, which forces same underlying types for t_1 and t_2, and returns a mapping ρ from the size variables of the supertype t_2 to their respective counterparts from the subtype t_1. Formally, the subtype relation is defined as follows:

$$\frac{\rho=[n\mapsto m]}{\vdash b\langle m\rangle <: b\langle n\rangle,\rho} \qquad \frac{\rho=[n_i\mapsto m_i]_{i=1}^{p}}{\vdash r\langle m_{1..p}\rangle <: r\langle n_{1..p}\rangle,\rho}$$

The type judgement for expressions is of the form:

$$\Gamma;\Delta\vdash e :: t,\Delta'$$

where Γ is a variable environment mapping program variables to their respective annotated types; $\Delta(\Delta')$ denotes the size constraint, which holds for the size variables associated with Γ (Γ and t) for expression e before (after) its evaluation; t is an annotated type.

Our type judgement generates types with fresh size variables. This facilitates the quantification of dead size variables at suitable junctures. (Dead size variables belong to intermediate expressions, neither from type environment nor from the result, of each type judgement.) In the above, we use the function *fresh*() to generate new size variables. We extend this to annotated type: the result of $\mathit{fresh}(t)$ has the same underlying type as t and is annotated with fresh size variables.

The rule [VAR] uses the function $\mathit{equate}(t_1,t_2)$ to generate equality constraints for the corresponding size variables of its two arguments. For example, we have $\mathit{equate}(\mathtt{Int}\langle\mathtt{r}\rangle,\mathtt{Int}\langle\mathtt{s'}\rangle)=(\mathtt{r}=\mathtt{s'})$. Occasionally, we may return a substitution using the following function: $\mathit{rename}(\mathtt{Int}\langle\mathtt{r}\rangle,\mathtt{Int}\langle\mathtt{s'}\rangle)=[\mathtt{r}\mapsto\mathtt{s'}]$. Note that both $\mathit{equate}(t_1,t_2)$ and $\mathit{rename}(t_1,t_2)$ succeed only in the case that t_1 and t_2 share the same underlying type. This check is necessary for the source program to be well-normal typed.

The rule [MI] is used for invocations of user-defined or primitive methods, whereby their type information is retrieved from the program (available globally as $\mathcal{P}$). Subtyping is used to generate a substitution on size variables from formal to actual arguments. Each method invocation includes as a safety check on the pre-condition of ϕ. It also ensures that all arguments of resource types are distinct to prevent aliasing using

$$\textit{distinct}\ \{x_1, .., x_m\} =_{df} (\forall i, j \in 1..m \cdot i \neq j \Rightarrow x_i \neq x_j)$$

The rule [LET] extends the type environment with a new local variable for the next expression, e_2. The subtyping rule $\vdash t_1 <: t_1', \rho$ returns a size variable substitution ρ from the type of v to e_1's type. We apply the inverse substitution ρ^{-1} to the size constraint collected from e_1. Note that we have added a check at the end of the scope to ensure that each local variable satisfies its finality constraint when it becomes inaccessible.

The rule [IF] for conditional expression introduces a path-sensitive analysis for the size constraints at the two branches, which are then combined with a disjunction.

3.3 How Is Usage Verification Achieved?

Our type-checking rules make use of the current size constraint, Δ, to perform necessary checks for protocol conformance. This constraint captures the states of both the resources and the other variables in relational form. If type-checking succeeds, we also derive an expected post-condition for the given expression. To illustrate this idea, consider the type-checking of an expression for which `x` is unlocked, but nothing is known about `y`, and where $\Gamma = \{x :: Lock\langle s\rangle, y :: Bool\langle b\rangle\}$ and $\Delta = (s = s' \wedge s' = 0)$. Under this scenario, type-checking fails as we may perform another unlock on a mutex that is already in the unlocked state, as illustrated below.

$$\frac{\Gamma; \Delta \wedge b = 1 \vdash \texttt{unlock(x)} :: \textbf{ILL-TYPED!} \qquad \Gamma; \Delta \wedge b = 0 \vdash \texttt{lock(x)} :: (), s = 0 \wedge s' = 1 \wedge b = 0}{\Gamma; \Delta \vdash \texttt{if y then unlock(x) else lock(x)} :: \textbf{ILL-TYPED!}}$$

However, if we would use a stronger context $\Delta_1 = (s = s' \wedge s' = 0 \wedge b = 0$ where `y` is known to be false, the judgement would succeed. Furthermore, our type-checking rule would compute a new post-condition which indicates that `x` will become locked, as indicated by the resulting context $\Delta_2 = (s = 0 \wedge s' = 1 \wedge b = 0)$. Note that *False* denotes the context of an infeasible path (or dead code).

$$\frac{\Gamma; \Delta_1 \wedge b = 1 \vdash \texttt{unlock(x)} :: (), \textit{False} \qquad \Gamma; \Delta_1 \wedge b = 0 \vdash \texttt{lock(x)} :: (), \Delta_2}{\Gamma; \Delta_1 \vdash \texttt{if y then unlock(x) else lock(x)} :: (), \Delta_2}$$

As another example, consider the following where $\Gamma = \{x :: File\langle s\rangle\}$, and the initial context is $(s' = 1)$. Note how the size context is being updated/propagated flow-sensitively through the sequence of expressions.

$$\frac{\Gamma;\ s' = 1 \vdash \texttt{open(f)} :: (), s' = 2 \qquad \dfrac{\Gamma;\ s' = 2 \vdash \texttt{read(f)} :: \texttt{Int}\langle r\rangle, s' = 2 \qquad \Gamma, v :: \texttt{Int}\langle r\rangle;\ s' = 2 \vdash \texttt{close(f)} :: (), s' = 4}{\Gamma;\ s' = 2 \vdash \texttt{let v = read(f) in close(f)} :: (), s' = 4}}{\Gamma;\ s' = 1 \vdash \texttt{open(f);(let v = read(f) in close(f))} :: (), s' = 4}$$

4. INSTRUMENTED SEMANTICS

We shall now define for our language an operational semantics that has been instrumented with the state of size variables. Instrumentation has been added to facilitate the correctness proof of our type system given in the next section. Its addition does not affect the underlying semantics of our language, as we can show net equivalence between the instrumented and underlying semantics via a bisimulation.

Notations used are defined below with *Var* and *SVar* to denote the domains of *program variables* and *size variables*, respectively.

Locations :	ι	$\in$	*Location*
Primitive values :	k	$\in$	$prim = Int \uplus Bool \uplus Void$
Values :	δ	$\in$	$Value = prim \uplus Location$
Resource types :	rt	$\in$	$ResType = (name, \langle SVar^*\rangle)$
Resource values :	r	$\in$	$ResVal = (ResType, SVar \rightarrow_{fin} Int, AbsVal)$
Variable Env :	Π	$\in$	$VEnv = Var \rightarrow_{fin} Value$
Store :	ϖ	$\in$	$Store = Location \rightarrow_{fin} ResVal$

Note that $f : A \rightarrow_{fin} B$ denotes a finite mapping from A to B. The variable environment Π is such a mapping. We write $\Pi[\delta/v]$ to denote an update of the variable v in Π to δ. We write $\Pi + \{v \mapsto \delta\}$ to denote an enhancement of Π to include a new binding of δ to v. Similar notations are used for the update and enhancement of resource value and the store.

The dynamic evaluation rules are of the following form.

$$\langle \Pi, \varpi\rangle\ [e] \hookrightarrow \langle \Pi', \varpi'\rangle\ [e']$$

Except for the primitive call, the other evaluation rules are standard and we include them in the appendix.

$\mathbf{[D-PrimCall]}$

$$\frac{f :: (t_1, .., t_p) \rightarrow t\ \textbf{st}\ \phi \in \mathcal{P} \qquad isPrim(f) \qquad (\varpi', \delta, flag) = primOp(f, [\Pi(v_1), .., \Pi(v_p)], \varpi) \qquad flag = \texttt{true}}{\langle \Pi, \varpi\rangle\ [f(v_1, .., v_p)] \hookrightarrow \langle \Pi, \varpi'\rangle\ [\delta]}$$

$\mathbf{[D-PrimCall-Error]}$

$$\frac{f :: (t_1, .., t_p) \rightarrow t\ \textbf{st}\ \phi \in \mathcal{P} \qquad isPrim(f) \qquad flag = \texttt{false} \qquad (\varpi', \delta, flag) = primOp(f, [\Pi(v_1), .., \Pi(v_p)], \varpi)}{\langle \Pi, \varpi\rangle\ [f(v_1, .., v_p)] \hookrightarrow Error\_Usage}$$

Take note that *primOp* is assumed to include checks which ensure that $pre(f)$ is satisfied, and that the necessary changes have been made to store, ϖ', to also satisfy ϕ. If the primitive's precondition is not satisfied, *primOp* is expected to return a `false` flag to signify that an error has occurred. Correspondingly, the evaluation becomes stuck with *Error_Usage* value. We will prove that for a well-typed program this situation cannot happen.

As an example, we also specify the *primOp* definition for two primitives from the buffer protocol: `newBuffer` and `add`. We do not provide any type rules to check such primitive definitions of our operational semantics, but shall assume that they adhere to the requirement stated above. Each resource value is made up of a resource type, its state, and an abstract value that is dependent on the actual implementation of the resource. The state of each resource is a mapping for its size variables, which essentially provides an instrumented semantics for resources.

$\mathbf{[newBuffer]}$

$$\frac{\iota = fresh() \qquad \alpha = \texttt{newBuffer}(n) \qquad \varpi' = \varpi + [\iota \mapsto (\texttt{Buffer}\langle s, c\rangle, [s \mapsto 0, c \mapsto n], \alpha)]}{primOp\ (\texttt{newBuffer}, [n], \varpi) =_{df} (\varpi', \iota, (n > 0))}$$

$\mathbf{[add]}$

$$\frac{\varpi(\iota) = (\texttt{Buffer}\langle s, c\rangle, [s \mapsto n_s, c \mapsto n_c], \alpha) \qquad \alpha' = \texttt{add}(\alpha, n) \qquad \varpi' = \varpi[\iota \mapsto (\texttt{Buffer}\langle s, c\rangle, [s \mapsto n_s + 1, c \mapsto n_c], \alpha')]}{primOp\ (\texttt{add}, [\iota, n], \varpi) =_{df} (\varpi', (), (n_s < n_c))}$$

5. SOUNDNESS OF TYPE SYSTEM

We shall now provide a correctness proof for our type rules. Given a well-typed program, our safety theorem guarantees that usage violations never occur.

For this purpose, we extend the static semantics of the language with the introduction of *store typing* to describe resource types at each store location. This ensures that objects created in the store during runtime are type-wise consistent with those captured by the static semantics. In our case, it is denoted by:

$$\Sigma \in \mathit{StoreType} = \mathit{Location} \rightarrow \mathit{ResType}$$

Store typing is conventionally used to link static and dynamic semantics [22]. We also introduce a consistency relation to say that the runtime environment (Π, ϖ) is *consistent* with the type environment (Γ, Σ, Δ), written $\Gamma; \Sigma; \Delta \models \langle \Pi, \varpi \rangle$, if the following judgement holds:

$$\frac{\begin{array}{c} dom(\Pi) = dom(\Gamma) \qquad dom(\Sigma) = dom(\varpi) \qquad X = \mathcal{I}(\Gamma) \\ \forall v \in dom(\Pi) \cdot \Gamma; \Sigma; \mathit{True} \vdash value(\Pi(v), \varpi) :: t_v, \Delta_v \\ C = \bigwedge_{v \in dom(\Pi)} (\Delta_v \wedge equate(prime(\Gamma(v)), t_v) \\ \exists X.C \Rightarrow \exists X.\Delta \end{array}}{\Gamma; \Sigma; \Delta \models \langle \Pi, \varpi \rangle}$$

The judgement captures the fact that the latest values of stack and store are consistent with the final values captured by the static context, Δ. Function $value(\delta)$ is defined as follows:

$$\begin{array}{ll} value(\Pi(v), \varpi) = k, & \mathit{if}\ \Pi(v) = k \\ value(\Pi(v), \varpi) = \varpi(\iota), & \mathit{if}\ \Pi(v) = \iota \end{array}$$

Type-checking rules will be extended to use the store typing, as follows: $\Gamma; \Sigma; \Delta \vdash e :: t, \Delta'$.

We require additional intermediate expressions for use by the dynamic semantics. The syntax of intermediate expressions is thus extended from the original expression syntax as follows.

$$e ::= \ldots \mid \mathtt{ret}(v^*, b, e) \mid \iota$$

The expression $\mathtt{ret}(v^*, b, e)$ is used for capturing expression in a local block, or in a method invocation, after the stack has been extended. The list of variables associated with `ret` contains the local variables declared and already allocated in the stack. The newly introduced expression allows us to unframe the stack when the block has been completely evaluated. The flag b is used to indicate whether the `ret` expression originates from a local block. In this case, if the variable denotes a resource, the semantics will check whether the finality constraint is satisfied when the scope of the variable is terminated. The type-checking rule for $\mathtt{ret}(v^*, b, e)$ expression follows.

Return: [RET]

$$\frac{\begin{array}{c} \Gamma(v_i) = t_i \qquad X = \bigcup_{i=1}^{n} \{V(t_i) \cup prime(t_i)\} \\ \Gamma; \Sigma; \Delta \vdash e :: t, \Delta_1 \quad \Delta_2 = \exists X.\Delta_1 \end{array}}{\Gamma; \Sigma; \Delta \vdash \mathbf{ret}([v_1, .., v_n], b, e) :: t, \Delta_2}$$

We also provide type-checking rules for locations and resource values. The latter one will be used for the consistency relation of the runtime environment with the type environment.

Location: [LOC]

$$\frac{\Sigma(\iota) = r\langle n_{1..p}\rangle \quad s_i = \mathit{fresh}() \quad \rho = [n_i \mapsto s_i]_{i=1}^{p}}{\Gamma; \Sigma; \Delta \vdash \iota :: r\langle s_{1..p}\rangle, \Delta \wedge \rho(\mathit{inv}(r\langle s_{1..p}\rangle))}$$

Because location expressions are not included in the source language, they can only appear as a result of reduction rules, specifically when a primitive call reduces to a location. In this case, the location corresponds to a new, unique resource, and its invariant is added to the contextual constraint.

Resource Value: [RES−VAL]

$$\frac{s_i = \mathit{fresh}() \qquad \phi = \bigwedge_{i=1}^{p} (n_i = s_i)}{\Gamma; \Sigma; \Delta \vdash (r\langle n_{1..p}\rangle, \rho, \alpha) :: r\langle s_{1..p}\rangle, \Delta \wedge \rho\phi}$$

Note that the derived constraint is precise, as it uses the values from the runtime environment.

After introducing the additional type rules, we formulate the soundness of our type system by proving two key properties, formulated as preservation and progress theorems.

THEOREM 1 (PRESERVATION).

(a) (Value) If

$$\begin{array}{c} \Gamma; \Sigma; \Delta \vdash \delta : t, \Delta_1 \\ \Gamma; \Sigma; \Delta \models \langle \Pi, \varpi \rangle \end{array}$$

then the following holds where x is fresh:

$$\Gamma + \{x :: t\}; \Sigma; \Delta_1 \models \langle \Pi + \{x \mapsto \delta\}, \varpi \rangle.$$

(b) (Expression) If

$$\begin{array}{c} \Gamma; \Sigma; \Delta \vdash e : t, \Delta_1 \\ \Gamma; \Sigma; \Delta \models \langle \Pi, \varpi \rangle \\ \langle \Pi, \varpi \rangle\ [e] \hookrightarrow \langle \Pi', \varpi' \rangle\ [e'] \end{array}$$

then there exists $\Sigma' \supseteq \Sigma$, Γ', and Δ', such that

$$\begin{array}{c} \Gamma' \text{ - } local(e') = \Gamma \text{ - } local(e) \\ \Gamma'; \Sigma'; \Delta' \vdash e' : t, \Delta_1 \\ \Gamma'; \Sigma'; \Delta' \models \langle \Pi', \varpi' \rangle. \end{array}$$

The preservation theorem characterizes the result of evaluation: if a well-typed term takes a step of evaluation, then the resulting term has the same well-typedness (modulo renaming of size variables). Furthermore, the resulting runtime environment is consistent with the static context captured by the type judgement. The theorem uses the function $local(e)$ which returns the set of variables from the part of the stack that will be de-allocated during e's evaluation. Its definition follows:

$$\begin{array}{lll} local(e) = \mathit{case}\ e\ \mathit{of} & & \\ \quad \mathbf{ret}(v^*, b, e) & \rightarrow & \{v^*\} \cup local(e) \\ \quad \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 & \rightarrow & local(e_1) \cup local(e_2) \\ \quad \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 & \rightarrow & local(e_1) \cup local(e_2) \\ \quad \delta \mid v \mid f(v^*) & \rightarrow & \emptyset \end{array}$$

THEOREM 2 (PROGRESS). *If $\Gamma; \Sigma; \Delta \vdash e : t; \Delta_1$ and $\Gamma; \Sigma; \Delta \models \langle \Pi, \varpi \rangle$, then e is either a value or there exist Π' and ϖ' such that $\langle \Pi, \varpi \rangle\ [e] \hookrightarrow \langle \Pi', \varpi' \rangle\ [e']$.*

The progress theorem states that a well-typed term is either a value or it can take a step according to the evaluation rules.

Using the above theorems (proved in the appendix), we can prove that a well-typed term can never reach a stuck state during its evaluation (or reduce to an $\mathit{Error\_Usage}$ value). Specifically, the precondition of each primitive is satisfied by the states of the resources maintained in the runtime environment. Also when resources become inaccessible, they satisfy the finality constraint according to their type.

$$\underline{[\textbf{consumeU}]}$$

$$\frac{v \notin \Theta \qquad d = (\{v\} \lhd ann(\tau) = \mathbf{U} \rhd \emptyset)}{\Theta \vdash consumeU(v, \tau), \Theta \cup d}$$

$$\underline{[\textbf{conUL}]}$$

$$\frac{\begin{array}{c} \vdash \tau' {<:}\, \tau, \rho \qquad v \notin \Theta \cup \Lambda \\ d = (\{v\} \lhd ann(\tau') = \mathbf{U} \wedge ann(\tau) \neq \mathbf{L} \rhd \emptyset) \\ g = (\{v\} \lhd ann(\tau') = \mathbf{U} \wedge ann(\tau) = \mathbf{L} \rhd \emptyset) \end{array}}{\Theta, \Lambda, \Psi \vdash conUL(v, \tau', \tau), \Theta \cup d, \Lambda \cup g, \Psi \uplus \rho}$$

$$\underline{[\textbf{A}-\textbf{METH}]}$$

$$\frac{\begin{array}{c} \Gamma = \{v_1 :: \tau_1, .., v_p :: \tau_p\} \qquad \Delta = \mathit{pre}(f) \wedge \mathit{no}\mathcal{X}(\Gamma) \\ \Gamma; \Delta; \emptyset \vdash e :: \tau', \Delta_f, \Theta \qquad \vdash \tau' {<:}\, \tau, \rho \qquad \Delta_f \Rightarrow \rho\, \phi \\ ann(\tau) \neq \mathbf{L} \qquad \forall i \in 1..p.(ann(\tau_i) = \mathbf{L}) \Rightarrow v_i \notin \Theta \\ test_i = (\mathit{final}(\tau_i) \lhd ann(\tau_i) = \mathbf{U} \wedge v_i \notin \Theta \rhd \mathit{True}), \quad i \in 1..p \\ \Delta_f \mathrel{\approx\!>}_{\mathcal{I}(\Gamma)} \bigwedge_{i=1}^{p} test_i \end{array}}{\vdash_{func} f :: (\tau_1, .., \tau_p) \rightarrow \tau' \ \mathbf{st}\ \phi; f(v_1, .., v_p) = e}$$

$$\underline{[\textbf{A}-\textbf{MI}]}$$

$$\frac{\begin{array}{c} f :: (\tau_1, .., \tau_p) \rightarrow \tau \ \mathbf{st}\ \phi \in \mathcal{P} \qquad \tau' = \mathit{fresh}(\tau) \\ \Gamma(v_i) = \tau_i', \ \ i \in 1..p \qquad X = \bigcup_{i=1}^{p} \mathcal{I}(\tau_i') \qquad \Lambda_0 = \emptyset \qquad \rho_0 = [\,] \\ \Theta_{i-1}, \Lambda_{i-1}, \rho_{i-1} \vdash conUL(v_i, \tau_i', \tau_i), \Theta_i, \Lambda_i, \rho_i \ \ i \in 1..p \\ \rho = \rho_p \uplus \mathit{prime}(\rho_p) \uplus \mathit{rename}(\tau, \tau') \qquad \Delta \mathrel{\approx\!>}_X \rho(\mathit{pre}(f)) \end{array}}{\Gamma; \Delta; \Theta_0 \vdash f(v_1, .., v_p) :: \tau, \Delta \circ_X \rho(\phi), \Theta_p}$$

$$\underline{[\textbf{A}-\textbf{VAR}]}$$

$$\frac{\begin{array}{c} \Gamma(v) = \tau \qquad \Theta \vdash consumeU(v, \tau), \Theta' \\ \tau' = \mathit{fresh}(\tau) \qquad \phi = \mathit{equate}(\tau', \tau) \end{array}}{\Gamma; \Delta; \Theta \vdash v :: \tau', \Delta \wedge \phi, \Theta'}$$

$$\underline{[\textbf{A}-\textbf{LET}]}$$

$$\frac{\begin{array}{c} \Gamma; \Delta; \Theta \vdash e_1 :: \tau_1, \Delta_1, \Theta_1 \qquad \tau_1' = \mathit{freshA}(\tau_1, A) \qquad \vdash \tau_1 {<:}\, \tau_1', \rho \\ \Delta_1' = (\exists V(\tau_1) \cdot \rho^{-1}\, \Delta_1) \wedge \mathit{no}\mathcal{X}(\tau_1') \qquad Y = V(\tau_1') \cup \mathit{prime}(\mathcal{I}(\tau_1')) \\ \Gamma, v :: \tau_1'; \Delta_1'; \Theta_1 \vdash e_2 :: \tau_2, \Delta_2, \Theta_2 \\ test = (\mathit{final}(\tau_1') \lhd ann(\tau_1) = \mathbf{U} \wedge A = \mathbf{U} \wedge v \notin \Theta_2 \rhd \mathit{True}) \\ \Delta_2 \mathrel{\approx\!>}_{\mathcal{I}(\tau_1')} test \end{array}}{\Gamma; \Delta; \Theta \vdash \mathbf{let}\ v@A = e_1\ \mathbf{in}\ e_2 :: \tau_2, \exists Y \cdot \Delta_2, \Theta_2}$$

Figure 6: Alias-annotated type rules

6. DISCUSSION

For simplicity, in the previous sections we have applied language restriction to ensure that every resource is unique. While this simplifies our type system and its correctness proof, it restricts the set of programs that could be accepted. For a more practical system, we may instead rely to an alias analysis technique to help distinguish unique resources from possibly shared ones.

A common technique for handling shared resource is to use *weak updates* on the states of these potentially aliased resources [11, 13, 24]. (A weak update occurs when the state of a resource is being changed from s_1 to s_2 such that $s_1 {<:} s_2$. In other words, the state of such a shared resource may become less precise and is not allowed to be arbitrarily changed.) However, this approach limits the kinds of operations that shared resources may be subjected to. To compensate for this, we describe runtime mechanisms to check for safety preconditions on shared resources and their finality constraints.

In this section, we explore the use of a simpler alias type system, inspired by the work of Aldrich et al [2]. Consequently, we use **U**, **S** and **L** to denote unique, shared and lent resources respectively. Unique resources are not aliased and can be accurately tracked, while shared resources may have global aliases. Resources in lent mode are restricted to formal parameters whose references do not escape their methods.

In order to track the state changes for lent resources, we use a restricted form of lending named *lent-once* (or limited unique), whereby each unique (or lent-once) resource is only allowed to be passed to a single lent-once parameter for each method call [5, 7]. This restriction is meant to facilitate state change without aliasing problem, and can be statically checked.

For each resource type, we annotate it with an alias A as follows $r\langle n_{1..p}\rangle@A$ where $A = \mathbf{U} \mid \mathbf{S} \mid \mathbf{L}$. Since imperative effects are present only for variables of resource type, other types can be annotated as shared without any loss in precision. Alias-annotated types are denoted by τ, to distinguish them from types without alias annotations used in previous sections (denoted by t). To extract the alias of an annotated type, we use $\mathit{ann}(t@A) = A$. Furthermore, we use the following partial ordering among annotations:

$$A {<:} A \qquad \mathbf{U} {<:} \mathbf{S} \qquad \mathbf{U} {<:} \mathbf{L}$$

Note that this ordering allows each unique resource to be transferred to a location that is in either shared or lent mode, but not vice versa. To track unique objects whose references may have been consumed by such transfers, we require our type judgement to be augmented with a set of unique resources whose uniqueness have been consumed. We call this the *consumed set*, denoted by Θ, and track it in a flow-sensitive manner using the following judgement:

$$\Gamma; \Delta; \Theta \vdash e :: \tau, \Delta', \Theta'$$

Each unique resource may only be consumed once, but could be temporarily (and separately) borrowed out multiple times by lent-once parameters. Our extended type rules presented in figure 6 use two auxiliary functions to check that these conditions are satisfied. The first function, $consumeU$, ensures that uniqueness is not consumed twice, while $conUL$ function checks that, at each time instance, a unique argument can be lent at most once.

Resource parameters of primitive methods are usually annotated as lent: this allows the primitive method to perform state changes, while assuming that no aliases are created during its execution. Since state changes should be allowed only on unique resources, the alias subtyping relation disallows a shared resource to be passed as an argument to a lent parameter.

To accommodate with shared resource arguments, a *controlled* form of primitive operations is used. Primitive operations cannot make any assumption about the imperative size variables of their shared arguments. For this case, controlled primitives include runtime checks as an alternative for compile-time preconditions. Following the example from figure 2(b), we introduce the following primitive operations:

```
get :: Buffer⟨s, c⟩@L → Int⟨n⟩@S st (s > 0 ∧ s' = s − 1)
getCHK :: Buffer⟨s, c⟩@S → Int⟨n⟩@S st True
add :: (Buffer⟨s, c⟩@L, Int⟨n⟩@S) → () st (s < c ∧ s' = s + 1)
addCHK :: (Buffer⟨s, c⟩@S, Int⟨n⟩@S) → () st True
```

Take note that the post-state of both `getCHK` and `addCHK` primitives do not capture the state change for their shared argument, as denoted by the size constraint `True`.

To handle aliasing annotations, type rules are changed accordingly: rules for method declaration, variable, method invocation and let block are presented in figure 6. Similarly, the subtyping relation is extended to handle possibly shared resources.

$$\frac{\rho = [n \mapsto m]}{\vdash b\langle m\rangle@\mathbf{S} <: b\langle n\rangle@\mathbf{S}, \rho} \qquad \frac{A_1 <: A_2 \quad \rho = (\rho_F \lhd A_2 = \mathbf{S} \rhd \rho_F \uplus \rho_{\mathcal{I}})}{\vdash r\langle m_{1..p}\rangle@A_1 <: r\langle n_{1..p}\rangle@A_2, \rho}$$

where

$$\begin{array}{l} \rho_F = [m_i \mapsto n_i],\ \ n_i \in V(r\langle n_{1..p}\rangle) - \mathcal{I}(r\langle n_{1..p}\rangle) \\ \rho_{\mathcal{I}} = [m_i \mapsto n_i],\ \ n_i \in \mathcal{I}(r\langle n_{1..p}\rangle) \end{array}$$

This relation returns a substitution that links size variables from subtype with those from the supertype. Intuitively, if the supertype is annotated as shared, only immutable size variables can be retrieved from the subtype (as indicated by ρ_F). On the other hand, if the supertype is annotated as unique or lent, both immutable and imperative size variables are equated with those of the subtype (as indicated by the substitution $\rho_F \uplus \rho_{\mathcal{I}}$.) We use the following functions to generate types with fresh size variables:

$$\begin{array}{lcl} \mathit{fresh}(t@A) & = & (\mathit{fresh}(t))@A \\ \mathit{freshA}(t@A, A_1) & = & (\mathit{fresh}(t))@A_1 \end{array}$$

Also, conditional is expressed as $\xi_1 \lhd b \rhd \xi_2 =_{df} \begin{cases} \xi_1, & \mathit{if}\ b; \\ \xi_2, & \mathit{otherwise}. \end{cases}$

Other functions previously defined in the section 3.1 can be extended easily to handle alias-annotated types.

Compared to the previous section, the rule [A-METH] for method declaration checks additionally the finality of **U** arguments. Also, **L** arguments cannot be consumed, since their uniqueness is to be returned to the caller.

The rule [A-VAR] allows variables of resource type and primitive type to be treated similarly without restrictions.

In the rule [A-MI], arguments of resource type are checked for consumed uniqueness (this check subsumes the check for distinctness), and must adhere to the alias subtype relation.

In our protocol specification, we require each resource to satisfy its finality constraint prior to its disposal. Previously, this was checked only at the rule for `let` construct, where the (unique) resource becomes inaccessible.

In the presence of aliasing, for each possibly shared resource, we must now also allow the finality constraint to be checked at runtime, since statically it is not possible to determine when the resource becomes inaccessible. Thus, if a local declaration denotes a shared resource, we require at runtime a finality test (in the `let` construct) to check if the shared variable is the last active reference (a runtime mechanism could employ reference counting). Consequently, any non-compliance of such a finality check will be reported as a runtime error.

For a unique resource, we will continue to perform its finality check at compile-time. A unique resource may become inaccessible at three possible places, namely (i) `let` construct, (ii) method declaration or (iii) at a branch of conditional. Here, each unique resource that is not already consumed, will become dead thereafter and their finality must be correspondingly checked. On the other hand, if the unique resource is consumed, the obligation to satisfy its finality is transferred together with its uniqueness.

To highlight our technique, we introduce two examples that use bounded buffer resources. For the first example from figure 7(a), the finality constraint can be checked at compile-time because the resource maintains its uniqueness. This example is ill-typed, since the finality of the resource is not satisfied at the end of the let scope: when the buffer becomes inaccessible, it is not empty.

```
let b@U = newBuffer(10) in
  add(b,1);  add(b,2);  get(b)
```

(a)

```
let b@U = newBuffer(10) in
  add(b,1);  add(b,2);
  let b1@S = b in
    ( let b2@S = b1 in
      getCHK(b2) );  getCHK(b1)
```

(b)

Figure 7: Finality checks

The second example uses a combination of static and dynamic checks to ensure that the code in figure 7(b) satisfies the buffer protocol. A reference-counting mechanism is used to detect where the last reference to the resource becomes inaccessible, to enable a runtime check for the resource finality. The initial unique reference `b` becomes inaccessible when its uniqueness is consumed. However, two references to the buffer resource are shared via variables `b1` and `b2` and, when both of them become inaccessible, the finality constraint is checked and satisfied (the buffer is empty).

The two invocations of `add` primitive can be proven correct by our type system. The flow sensitivity of the alias type rules, allows the same resource to be viewed as unique for the first part of its lifetime (via variable `b`) and as shared afterwards (via variables `b1` and `b2`).

7. RELATED WORK

In recent years, several type-based approaches [11, 13] have been advocated for verifying user programs conformance to resource protocols through tracking the states of resources. They specifically cater to protocols expressible as finite state models and do not allow checking for resource finality. In CQual [13], qualifiers were added to C-type to track their states through the operations of selected protocols. This tracking is done automatically for user programs in a flow-sensitive manner. In Vault[11], programmers are expected to add annotations to help check that device drivers are being used correctly. Annotations include alias and linearity information and also a special variant type to provide path-sensitivity to the analysis. As a downside, the programming style is less flexible, since both branches of a conditional have to agree on the resource states. Both CQual and Vault are to be supported by alias analysis, where shared objects are limited to *weak updates* on their states.

Another approach to the problem of resource verification is to use dataflow analysis. ESP [10] was designed to analyse file errors in C programs and does not require user annotations. Their analysis is path-sensitive and has polynomial complexity, analyzing only relevant branches taken by the state automaton transitions. On the downside, ESP is able to track only one resource at a time, limiting the set of properties that can be verified.

Igarashi and Kobayashi [18, 19] proposed a general framework for resource usage analysis to infer usage patterns of resources. The usage pattern is expressed as a trace that is checked against the declared resource protocol (also specified via a trace.) Similar to the above mentioned type-based methods [11, 13], but unlike our approach, resource usage analysis is not designed to capture value-dependent behavior of programs. We found this to be crucial in refining our path-sensitive analysis.

Other trace-based approaches [9, 21] aim to enforce a safety property onto a source program by adding dynamic checks where other static analyses would reject a program as unsafe. Fradet et al [9] analyze the usage traces at compile-time to minimize the number of required runtime tests. Similarly, Marriott et al [21] employed context free languages to capture usage patterns which are then checked against protocols expressed via regular automata. However, by mapping entire programs to automata, and performing transformations and analyses on these automata (in a manner similar to whole program analysis) these approaches lack modularity.

To allow more properties to be checked at compile time, Mandelbaum et al [20] devised a general theory of type refinements. They use a fragment of intuitionistic linear logic for local reasoning on program state. The properties that they reason about are divided in persistent and ephemeral facts which are similar to our immutable and imperative sizes of resources, respectively. However, values are given singleton types and there is no provision for handling shared resources. Their predicate logic is expressive, but their system is unable to express protocols with unbounded (or symbolic) number of states like the buffer resource protocol. Furthermore, relational analysis between program variables and resource states is not directly supported.

8. CONCLUDING REMARKS

Traditionally, dependent type [25, 17, 6] has been advocated for size analysis and used in applications, such as termination analysis[27, 3], array bound checks elimination[28, 8] and memory space analysis[16, 15]. Through Xanadu[26], it has recently been extended to imperative languages where variables may change values, but objects created are presently captured in a *size-immutable* way so that aliasing is not an issue. However, resources are inherently stateful. We have shown that this aspect can be tamed with the help of language restriction, so that aliasing is not an issue in the presence of mutability. Nevertheless, we also highlight a more sophisticated solution which tracks aliasing and uses a runtime checking mechanism to allow shared resources to be properly handled. Furthermore, our approach can cover a wide range of resource protocols, including those with finality requirement prior to the death of its resource.

With this enhanced dependent type system, we have designed a new solution to the protocol usage verification problem, and have also proven its correctness. Our solution is both precise and expressive as it is able to capture a wide range of resources and their protocols. We have presented our approach within a type-checking framework. Extension to an inference framework should be feasible, and could follow the approach taken for array check optimization that was recently proposed in [8]. Under this approach, we may introduce runtime tests into locations where protocol safety cannot be guaranteed, further extending the scope of our method. Our adopted language is simple, and we have exploited this to formalize a provably correct resource protocol verification technology.

Acknowledgments

We gratefully acknowledge helpful and encouraging suggestions from Florin Craciun, Siau-Cheng Khoo, Shengchao Qin, Martin Sulzmann and Dana Xu. The anonymous referees provided many valuable comments that helped streamline the presentation of the paper.

9. REFERENCES

[1] Martín Abadi and Bruno Blanchet. Computer-Assisted Verification of a Protocol for Certified Email. In Radhia Cousot, editor, *Proceedings of the International Static Analysis Symposium (SAS)*, volume 2694 of *Lecture Notes on Computer Science*, pages 316–335, San Diego, California, June 2003. Springer Verlag.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotation for Program Understanding. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Seattle, Washington, November 2002.

[3] Hugh Anderson and Siau-Cheng Khoo. Affine-based size-change termination. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*, June 2003.

[4] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the ACM Symposium on the Principles of Programming Languages (POPL)*, pages 1–3. ACM Press, 2002.

[5] E. C. Chan, J. Boyland, and W. L. Scherlis. Promises: Limited Specifications for Analysis and Manipulation. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE)*, pages 167–176, Kyoto, Japan, April 1998.

[6] W.N. Chin and S.C. Khoo. Calculating sized types. In *Proceedings of the ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 62–72, Boston, Massachusetts, United States, January 2000.

[7] W.N. Chin, S.C. Khoo, and S.C. Qin. A Sized Type System for Objects with Alias Controls. Technical report, SoC, Natl Univ. of Singapore, January 2004. avail. at http://www.comp.nus.edu.sg/~qinsc/papers/sizedtype.ps.gz.

[8] W.N. Chin, S.C. Khoo, and Dana N. Xu. Deriving pre-conditions for array bound check elimination. In *Programs as Data Objects II*, pages 2–24, Aarhus, Denmark, May 2001. Springer Verlag.

[9] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *Proceedings of the ACM Symposium on the Principles of Programming Languages (POPL)*, January 2000.

[10] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation (PLDI)*, June 2002.

[11] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation (PLDI)*, June 2001.

[12] Amy P. Felty, Douglas J. Howe, and Frank A. Stomp. Protocol verification in Nuprl. In *Tenth International Conference on Computer Aided Verification*, pages 428–439. Springer-Verlag Lecture Notes in Computer Science, 1998.

[13] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation (PLDI)*, June 2002.

[14] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with Blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239. Lecture Notes

in Computer Science 2648, Springer-Verlag, 2003.

[15] Martin Hofmann. The strength of non-size increasing computation. In *Proceedings of the ACM Symposium on the Principles of Programming Languages (POPL)*, pages 260–269. ACM Press, 2002.

[16] J. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space: Towards Embedded ML Programming. In *Proceedings of the ACM Conference on Functional Programming (ICFP)*, September 1999.

[17] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the ACM Symposium on the Principles of Programming Languages (POPL)*, pages 410–423. ACM Press, January 1996.

[18] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proceedings of the ACM Symposium on the Principles of Programming Languages (POPL)*, January 2002.

[19] Naoki Kobayashi. Time regions and effects for resource usage analysis. Technical report, Tokyo Inst. of technology, 2003.

[20] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proceedings of the ACM Conference on Functional Programming (ICFP)*, Uppsala, Sweden, 2003.

[21] Kim Marriott, Peter Stuckey, and Martin Sulzmann. Resource usage verification. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*, Beijing, China, November 2003.

[22] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[23] W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.

[24] F. Smith, D. Walker, and G. Morrisett. Alias Types. In *Proceedings of the 9th European Symposium on Programming*, Berlin, Germany, March 2000.

[25] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[26] H. Xi. Imperative Programming with Dependent Types. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*, Santa Barbara, June 2000.

[27] H. Xi. Dependent Types for Program Termination Verification. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*, Boston, June 2001.

[28] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation (PLDI)*, pages 249–257. ACM Press, June 1998.

APPENDIX

A. EVALUATION RULES

We specify the evaluation rules, as follows.

[D−Var]

$$\frac{\Pi(v) = \delta}{\langle \Pi, \varpi \rangle\ [v] \hookrightarrow \langle \Pi, \varpi \rangle\ [\delta]}$$

[D−If−true] and [D−If−false]

$$\frac{\Pi(v) = \mathtt{true}}{\langle \Pi, \varpi \rangle\ [\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2] \hookrightarrow \langle \Pi, \varpi \rangle\ [e_1]}$$

$$\frac{\Pi(v) = \mathtt{false}}{\langle \Pi, \varpi \rangle\ [\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2] \hookrightarrow \langle \Pi, \varpi \rangle\ [e_2]}$$

[D−Let−1] and [D−Let−2]

$$\frac{\langle \Pi, \varpi \rangle\ [e_1] \hookrightarrow \langle \Pi', \varpi' \rangle\ [e_1']}{\langle \Pi, \varpi \rangle\ [\mathbf{let}\ v = e_1\ \mathbf{in}\ e_2] \hookrightarrow \langle \Pi', \varpi' \rangle\ [\mathbf{let}\ v = e_1'\ \mathbf{in}\ e_2]}$$

$$\frac{\Pi' = \Pi + \{x \mapsto \delta\} \quad x = \mathit{fresh}()}{\langle \Pi, \varpi \rangle\ [\mathbf{let}\ v = \delta\ \mathbf{in}\ e_2] \hookrightarrow \langle \Pi', \varpi \rangle\ [\mathbf{ret}(x, \mathtt{true}, [v \mapsto x]e_2]}$$

We use $x = \mathit{fresh}()$ to obtain new program variables, similar to the function used to generate new size variables in the static semantics.

[D−Ret−1] and [D−Ret−2]

$$\frac{\langle \Pi, \varpi \rangle\ [e] \hookrightarrow \langle \Pi', \varpi' \rangle\ [e']}{\langle \Pi, \varpi \rangle\ [\mathbf{ret}([v_1, .., v_n], \mathit{cflag}, e)] \hookrightarrow \langle \Pi', \varpi' \rangle\ [\mathbf{ret}([v_1, .., v_n], \mathit{cflag}, e')]}$$

$$\frac{\Pi' = \Pi - \{v_1, .., v_n\} \qquad \mathit{cflag} \Rightarrow \mathit{check\_final}(\Pi(v_1), \varpi)}{\langle \Pi, \varpi \rangle\ [\mathbf{ret}([v_1, .., v_n], \mathit{cflag}, \delta)] \hookrightarrow \langle \Pi', \varpi \rangle\ [\delta]}$$

The boolean flag *cflag* indicates whether a `ret` expression originates from a `let` construct (in [D−Let−2], *cflag* is `true`) or from a method invocation (in [D−Call], *cflag* is `false`). $\mathit{check\_final}(\delta, \varpi)$ will check if the resource δ in ϖ satisfies the finality constraint. If the value δ does not represent a resource, $\mathit{check\_final}(\delta, \varpi) =_{df} \mathit{True}$.

[D−Call]

$$\frac{\begin{array}{c} f :: (t_1, .., t_p) \rightarrow t\ \mathbf{st}\ \phi; f(v_1, .., v_p) = e \in \mathcal{P} \\ x_1, .., x_p = \mathit{fresh}() \\ \Pi' = \Pi + \{x_i \mapsto \Pi[v_i]\}_{i=1}^{p} \quad \rho = \{v_i \mapsto x_i\}_{i=1}^{p} \end{array}}{\langle \Pi, \varpi \rangle\ [f(v_1, .., v_p)] \hookrightarrow \langle \Pi', \varpi \rangle\ [\mathbf{ret}([x_1, .., x_p], \mathbf{false}, \rho e)]}$$

B. PROOFS

Proof of preservation: (a) After extending both the dynamic stack with a value δ, and the type environment with a fresh variable of δ's corresponding type, the consistency relation continues to hold assuming two hypotheses: the non-extended versions of the type environment (Γ) and dynamic stack are also consistent, and the constant k is well-typed under Γ. If the value is a location ι, the consistency relation requires that the state of ι's corresponding resource satisfies its resource type invariant. This requirement is satisfied, as all primitive declarations are checked that they preserve the resource invariance (rule [Prim]).

(b) By induction on each well-typed expression e and on each evaluation rule applicable for e. At each step of the induction, we assume that the desired property holds for each inductive derivation step, and proceed by case analysis on the next derivation step. Our induction is carried out on the expected forms of well-typed expressions, as follows.

Case [VAR]: Here, v is of base type (variables of resource types cannot appear in this position during evaluation), let us denote it by $\Gamma(v) = b\langle n\rangle$. We choose Γ', Σ' and Δ' to be respectively Γ, Σ and Δ. $\Gamma', \Sigma', \Delta' \models \langle \Pi, \varpi\rangle$ is trivially maintained. We can also conclude that if v reduces to k then $\Pi(v) = k$. From the consistency relation, it follows that $\Gamma'; \Sigma'; \Delta' \vdash k :: b\langle n\rangle, \Delta'$.

Case [MI]: From static semantics, we have:
$\Gamma; \Sigma; \Delta \vdash f(v_1, .., v_p) :: t, \Delta \circ_X \rho(\phi)$, and $\Delta \approx\!> _X \rho(pre(f))$. The second assumption of the theorem is that $\Gamma, \Sigma, \Delta \models \langle \Pi, \varpi\rangle$. There are two ways in which a call can be reduced, depending on whether it is user-defined or primitive.

Subcase (D-Call): We choose $\Gamma' = \Gamma + \{x_i : t_i\}_{i=1}^p$, $\Sigma' = \Sigma$ and $\Delta' = \Delta$. From [MI] type rule, we have $\Gamma(v_i) = t_i$, and consequently consistency relation continues to hold:
$\Gamma + \{x_i :: t_i\}_{i=1}^p; \Sigma; \Delta \models \langle \Pi + \{x_i \mapsto \Pi[v_i]\}_{i=1}^p\rangle, \varpi$. From [RET] and [METH], we can also conclude that $\Gamma'; \Sigma'; \Delta' \vdash \mathtt{ret}([x_1, .., x_p], \mathtt{true}, \rho e) :: t, \Delta \circ_X \rho(\phi)$.

Subcase (D-PrimCall): Choose $\Gamma' = \Gamma$, $\Sigma' = \Sigma$ and $\Delta' = \Delta \circ_X \rho(\phi)$. Due to the consistency of *primOp* with ϕ, it follows that $\Gamma', \Sigma', \Delta' \models \langle \Pi, \varpi'\rangle$. Furthermore, we can also conclude that $\Gamma'; \Sigma'; \Delta' \vdash \delta :: t, \Delta \circ_X \rho(\phi)$.

Case [LET]: There are two rules (D-Let-1) and (D-Let-2) by which $e \hookrightarrow e'$ can be derived. We consider each case separately.

Subcase (D-Let-1): From static semantics, we have: $\Gamma; \Sigma; \Delta \vdash e_1 :: t_1, \Delta_1$. By induction hypothesis, there exists Γ', Σ' and Δ' such that $\Gamma'; \Sigma'; \Delta' \vdash e_1' :: t_1, \Delta_1$ and $\Gamma'; \Sigma'; \Delta' \models \langle \Pi', \varpi'\rangle$. The latter relation shows that consistency relation holds also for the environment of expression e'. From the [LET] rule applied on e, we have $\Gamma' + \{v :: t\}; \Sigma'; \Delta_1' \vdash e_2 :: t_2, \Delta_2$. By applying [LET] rule on e', we obtain for e and e' same type and contextual constraint.

Subcase (D-Let-2): We choose Γ' to be $\Gamma + \{x :: t\}$, Σ' to be Σ and Δ' to be $\exists X \cdot \Delta_1 \wedge \mathit{equate}(t, t_1) \wedge \mathit{no}\mathcal{X}(t)$. From [LET] rule, we have $\Gamma + \{x :: t\}; \Sigma'; \Delta' \vdash e_2 :: t_2, \Delta_2$, such that Δ_2 implies *final(t)*. Applying [RET], we can conclude that the types of e and e' coincide. The consistency relation is also maintained: from [LET] rule, we have $\Gamma'; \Sigma'; \Delta' \vdash \delta :: t_1, \Delta''$, and consequently we can conclude this case:
$\Gamma + \{x :: t\}; \Sigma; \Delta_2 \models \langle \Pi + \{x \mapsto \delta\}, \varpi\rangle$, where $\Delta_2 = \exists X \cdot \Delta_1 \wedge \mathit{equate}(t, t_1) \wedge \mathit{no}\mathcal{X}(t)$.
Additionally, if the value that e_2 will reduce to is a location, then the corresponding resource satisfies its finality constraint. We obtained that the contextual constraint Δ_2 satisfies *final(t)* from [LET] rule. From the induction hypothesis, we can assume that, after applying [RET] rule, the consistency relation holds. Specifically the runtime environment obtained after e_2 reduces to a location satisfies Δ_2. This runtime environment will also satisfy *final(t)* when the resource will become inaccessible. As a consequence, for a well-typed program, the runtime test *check_final* is always satisfied and can be safely eliminated.

Case [IF]: Here, e must have the form **if** v **then** e_1 **else** e_2, for some v, e_1 and e_2. We also must have $\Gamma(v) = \mathtt{Bool}\langle \mathtt{b}\rangle$,
$\Gamma; \Delta \wedge b = 1 \vdash e_1 :: t_1, \Delta_1$ and $\Gamma; \Delta \wedge b = 0 \vdash e_2 :: t_2, \Delta_2$. There are two rules (D-If-true) and (D-If-false) by which $e \hookrightarrow e'$ can be derived.

Subcase (D-If-true): If $e \hookrightarrow e'$ is derived using (D-If-true), we know that $\Pi(v)$ must be $\mathtt{true}$ and the resulting expression is e_1. Choose Γ' to be Γ, Δ' to be $\Delta \wedge (b = 1)$ and Σ' to be Σ. It is obvious that $\Gamma', \Sigma', \Delta' \models \langle \Pi, \varpi\rangle$. We also have
$\Gamma'; \Sigma'; \Delta' \vdash e_1 :: t_1, \rho\, \Delta_1$ and $(\rho_1\, \Delta_1 \Rightarrow \rho_1\, \Delta_1 \vee \rho_2\, \Delta_2)$.

Subcase (D-If-false): Similar reasoning as *Subcase* (D-If-true).

Case [RET]: There are two rules (D-Ret-1) and (D-Ret-2) by which $e \hookrightarrow e'$ can be derived. We consider each case separately.

Subcase (D-Ret-1): From static semantics, we have: $\Gamma; \Sigma; \Delta \vdash e_1 :: t_1, \Delta_1$. By induction hypothesis, there exists Γ', Σ' and Δ' such that $\Gamma'; \Sigma'; \Delta' \vdash e_1' :: t_1, \Delta_1$ and $\Gamma'; \Sigma'; \Delta' \models \langle \Pi', \varpi'\rangle$. The latter relation shows that consistency relation holds also for the environment of expression e'. From the [RET] rule, and the similarity in typing for e_1 and e_1', we conclude that same type and contextual constraint are derived for e and e'.

Subcase (D-Ret-2): Let us choose Γ' to be $\Gamma - \{v_1, .., v_n\}$, Σ' to be Σ and Δ' to be Δ. The consistency relation is trivially maintained, since $\{v_1, .., v_n\}$ are excluded from both the type environment and dynamic stack. Also, from [RET] rule, we can conclude that e and e' have the same type.

Proof of progress: By induction on the derivation of $e : t; \Delta_1$. The cases in [Cons1]-[Cons4] are immediate since e is a value. For the other cases, we argue as follows.

Case [VAR]: Rule (D-Var) can be applied to reduce a variable to a primitive value.

Case [MI]: This can be reduced by (D-Call) or (D-PrimCall), depending on whether e represents a user-defined or a primitive method call. In the former case, a one-step reduction is always possible by the (D-Call) rule.

However, for primitive method there is the possibility for the failure of the associated runtime test, making the evaluation to get stuck. We will assume that the precondition specified by the user and used by the static semantics ($pre(f)$) describes faithfully the runtime test executed dynamically.

We can conclude from the static semantics that $\Delta \approx\!> _X \rho(pre(f))$. From this fact, together with the consistency relation
$\Gamma; \Sigma; \Delta \models \langle \Pi, \varpi\rangle$, it follows that the runtime test cannot fail. Hence, one-step reduction will proceed by (D-PrimCall). This further indicates that for a well-typed program the runtime tests are always satisfied and therefore can be safely eliminated.

Case [LET]: This can be reduced by either (D-Let-1) or (D-Let-2). In the case that e_1 is not a value, we rely on induction hypothesis to make a reduction $e_1 \hookrightarrow e_1'$. Otherwise, if e_1 is a value, we use (D-Let-2) to perform a one-step reduction.

Case [IF]: For e to be well-typed, v must be a boolean value, either `true` or `false`. In the first case, (D-If-true) applies, otherwise (D-If-false) applies.

Case [RET]: This can be reduced by either (D-Ret1) or (D-Ret2). In the case that e_1 is not a value, we rely on induction hypothesis to make a reduction $e_1 \hookrightarrow e_1'$. Otherwise, if e_1 is a value, rule (D-Ret2) can be used to perform a one-step reduction.