To Memory Safety through Proofs*

Hongwei Xi and Dengping Zhu

Computer Science Department
Boston University

October 29, 2018

Abstract

We present a type system capable of guaranteeing the memory safety of programs that may involve (sophisticated) pointer manipulation such as pointer arithmetic. With its root in a recently developed framework *Applied Type System* ($\mathcal{ATS}$), the type system imposes a level of abstraction on program states through a novel notion of recursive stateful views and then relies on a form of linear logic to reason about such stateful views. We consider the design and then the formalization of the type system to constitute the primary contribution of the paper. In addition, we also mention a running implementation of the type system and then give some examples in support of the practicality of programming with recursive stateful views.

Keywords: Stateful View, Viewtype, Applied Type System, ATS

*Partially supported by NSF grant no. CCR-0229480

1 Introduction

The need for direct memory manipulation through pointers is essential in many applications and especially in those that involve systems programming. However, it is also commonly understood that the use (or probably misuse) of pointers is often a rich source for program errors. In safe programming languages such as ML and Java, it is completely forbidden to make explicit use of pointers and memory manipulation is done through systematic allocation and deallocation. In order to cope with applications requiring direct memory manipulation, these languages often provide a means to interface with functions written in unsafe languages such as C. While this is a workable design, the evident irony of this design is that probably the most difficult part of programming must be done in a highly error-prone manner with little, if there is any, support of types. This design certainly diminishes the efforts to promote the use of safe languages such as ML and Java.

We have previously presented a framework *Applied Type System* ($\mathcal{ATS}$) to facilitate the design and formalization of type systems in support of practical programming. It is already demonstrated that various programming styles (e.g., modular programming [Xi09], object-oriented programming [XCC03, CSX04], meta-programming [CX03]) can be directly supported *within* $\mathcal{ATS}$ without resorting to *ad hoc* extensions. In this paper, we extend $\mathcal{ATS}$ with a novel notion of recursive stateful views, presenting the design and then the formalization of a type system *ATS/SV* that can effectively support the use of types in capturing program invariants in the presence of pointers. For instance, the interesting invariant can be readily captured in *ATS/SV* that each node in a doubly linked binary tree points to its children that point back to the node itself, and this is convincingly demonstrated in an implementation of AVL trees and splay trees[Xi08]. Also, we have presented previously a less formal introduction to *ATS/SV* [ZX05], where some short examples involving stateful views can be found.

There are a variety of challenging issues that we must properly address in order to effectively capture invariants in programs that may make (sophisticated) use of pointers such as pointer arithmetic. First and foremost, we employ a notion of stateful views to model memory layouts. For instance, given a type T and an address L, we can form a (primitive) stateful view $T@L$ to mean that a value of type T is stored at the address L. We can also form new stateful views in terms of primitive stateful views. For instance, given types T_1 and T_2 and an address L, we can form a view $(T_1@L) \otimes (T_2@L+1)$ to mean that a value of type T_1 and another value of type T_2 are stored at addresses L and $L + 1$, respectively, where $L + 1$ stands for the address immediately after L. Intuitively, a view is like a type, but it is linear. Given a term of some view V, we often say that the term proves the view V and thus refer to the term as a *proof* (of V). It will soon become clear that proofs of views cannot affect the dynamics of programs and thus are all erased before program execution.

In order to model more sophisticated memory layouts, we need to form recursive stateful views. For instance, we may use the concrete syntax in Figure 1 to declare a (dependent) view constructor *arrayView*: Given a type T, an integer I and an address L, *arrayView*(T, I, L) forms a view stating that there are I values of type T stored at addresses $L, L+1, \ldots, L+I-1$. There are two proof constructors *ArrayNone* and *ArraySome* associated with *arrayView*, which are formally assigned the following views:

```
dataview arrayView (type, int, addr) =
  | {a:type, l:addr} ArrayNone (a, 0, l)
  | {a:type, n:int, l:addr | n >= 0}
      ArraySome (a, n+1, l) of (a @ l, arrayView (a, n, l+1))
```

Figure 1: An example of recursive stateful view

$$\begin{array}{lcl} \textit{ArrayNone} & : & \forall\lambda.\forall\tau.() \multimap \textit{arrayView}(\tau, 0, \lambda) \\ \textit{ArraySome} & : & \forall\lambda.\forall\tau.\forall\iota.\iota \geq 0 \supset (\tau@\lambda \otimes \textit{arrayView}(\tau, \iota, \lambda+1) \multimap \textit{arrayView}(\tau, \iota+1, \lambda)) \end{array}$$

Note that we use $\otimes$ and $\multimap$ for linear (multiplicative) conjunction and implication and τ, ι and λ for variables ranging over types, integers and addresses, respectively. Intuitively, *ArrayNone* is a proof of *arrayView*$(T, 0, L)$ for any type T and address L, and *ArraySome*(pf_1, pf_2) is a proof of *arrayView*$(T, I+1, L)$ for any type T, integer I and address L if pf_1 and pf_2 are proofs of views $T@L$ and *arrayView*$(T, I, L+1)$, respectively.

Given a view V and a type T, we can form a *viewtype* $V \wedge T$ such that a value of the type $V \wedge T$ is a pair (pf, v) in which pf is a proof of V and v is a value of type T. For instance, the following type can be assigned to a function $read_L$ that reads from the address L:

$$(T@L) \wedge \mathbf{ptr}(L) \rightarrow (T@L) \wedge T$$

Note that $\mathbf{ptr}(L)$ is the singleton type for the only pointer pointing to the address L. When applied to a value (pf_1, L) of type $(T@L) \wedge \mathbf{ptr}(L)$, the function $read_L$ returns a value (pf_2, v), where v is the value of type T that is supposed to be stored at L. Both pf_1 and pf_2 are proofs of the view $T@L$, and we may think that the call to $read_L$ consumes pf_1 and then produces pf_2. Similarly, the following type can be assigned to a function $write_L$ that writes a value of type T_2 to the address L where a value of type T_1 is originally stored:

$$(T_1@L) \wedge (\mathbf{ptr}(L) * T_2) \rightarrow (T_2@L) \wedge \mathbf{1}$$

Note that $\mathbf{1}$ stands for the unit type. In general, we can assign the read (*getPtr*) and write (*setPtr*) functions the following types:

$$\begin{array}{lcl} \textit{getPtr} & : & \forall\tau.\forall\lambda.(\tau@\lambda) \wedge \mathbf{ptr}(\lambda) \rightarrow (\tau@\lambda) \wedge \tau \\ \textit{setPtr} & : & \forall\tau_1.\forall\tau_2.\forall\lambda.(\tau_1@\lambda) \wedge (\mathbf{ptr}(\lambda) * \tau_2) \rightarrow (\tau_2@\lambda) \wedge \mathbf{1} \end{array}$$

In order to effectively support programming with recursive stateful views, we adopt a recently proposed design that combines programming with theorem proving [CX05]. While it is beyond the scope of the paper to formally explain what this design is, we can readily use some examples to provide the reader with a brief overview as to how programs and proofs are combined in this design. Also, these examples are intended to provide the reader with some concrete feel as to what can actually be accomplished in *ATS/SV*. Of course, we need a process to elaborate programs written in the concrete syntax of ATS into the (kind of) formal syntax of $\lambda^{\forall,\exists}_{view}$ (presented in Section 3).

```
fun getFirst {a:type, n:int, l:addr | n > 0} (pf: arrayView (a,n,l) | p: ptr(l))
  : '(arrayView (a,n,l) | a) =
  let
     prval ArraySome (pf1, pf2) = pf // pf1: a@l and pf2: arrayView (a,n-1,l+1)
     val '(pf1' | x) = getPtr (pf1 | p) // pf1: a@l
  in
     '(ArraySome (pf1', pf2) | x)
  end

// The following is a proof function and thus is required to be total
prfun splitLemma {a:type, n:int, i:int, l:addr | 0 <= i, i <= n} .<i>.
     (pf: arrayView (a, n, l)): '(arrayView (a, i, l), arrayView (a, n-i, l+i)) =
  sif i == 0 then '(ArrayNone (), pf) // [sif]: static [if] for forming proofs
  else
    let
       prval ArraySome (pf1, pf2) = pf // this cannot fail as [i > 0] holds
       prval '(pf21, pf22) = splitLemma {a,n-1,i-1,l+1} (pf2)
    in
       '(ArraySome (pf1, pf21), pf22)
    end

fun get {a:type, n:int, i:int, l:addr | 0 <= i, i < n}
   (pf: arrayView (a, n, l) | p: ptr l, offset: int i): '(arrayView (a, n, l) | a) =
  let
     // pf1: arrayView (a,i,l) and pf2: arrayView (a,n-i,l+i)
     prval '(pf1, pf2) = splitLemma {a,n,i,l} (pf)
     val '(pf2 | x) = getFirst (pf2 | p + offset)
  in
     '(unsplitLemma (pf1, pf2) | x)
  end
```

Figure 2: A programming example involving recursive stateful views

This is a rather involved process, and we unfortunately could not formally describe it in this paper and thus refer the interested reader to [Xi04b] for further details. Instead, we are to provide some (informal) explanation to facilitate the understanding of the concrete syntax we use.

We have so far finished a running implementation of ATS [Xi08], a programming language with its type system rooted in the framework $\mathcal{ATS}$, and *ATS/SV* is a part of the type system of ATS. In Figure 2, we present some code in ATS. We use $'(\ldots)$ to form tuples in the concrete syntax, where the quote symbol ($'$) is solely for the purpose of parsing. For instance, $'()$ stands for the unit (i.e., the tuple of length 0). Also, the bar symbol ($|$) is used as a separator (like the comma symbol (,)). It should not be difficult to relate the concrete syntax to the formal syntax of $\lambda_{view}^{\forall,\exists}$ introduced later in Section 3 (assuming that the reader is familiar with the SML syntax). The header of the function *getFirst* in Figure 2 indicates that the following type is assigned to it:

$$\forall\tau.\forall\iota.\forall\lambda.\iota > 0 \supset (\mathit{arrayView}(\tau, \iota, \lambda) \wedge \mathbf{ptr}(\lambda) \rightarrow \mathit{arrayView}(\tau, \iota, \lambda) \wedge \tau)$$

where $\iota > 0$ is a guard to be explained later. Intuitively, when applied to a pointer that points to a nonempty array, *getFirst* takes out the first element in the array. In the body of *getFirst*, *pf* is a proof of the view $\mathit{arrayView}(a, n, l)$, and it is guaranteed to be of the form $\mathit{ArraySome}(\mathit{pf}_1, \mathit{pf}_2)$, where pf_1 and pf_2 are proofs of views $a@l$ and $\mathit{arrayView}(a, n-1, l+1)$, respectively; thus pf'_1 is also a proof of $\tau@\lambda$ and $\mathit{ArraySome}(\mathit{pf}'_1, \mathit{pf}_2)$ is a proof of $\mathit{arrayView}(a, n, l)$. In the definition of *getFirst*, we have both code for dynamic computation and code for static manipulation of proofs of views, and the latter is to be erased before dynamic computation starts. For instance, the definition of *getFirst* turns into:

```
fun getFirst (p) = let val x = getPtr p in x end
```

after the types and proofs in it are erased; so the function can potentially be compiled into one load instruction after it is inlined.

We immediately encounter an interesting phenomenon when attempting to implement the usual array subscripting function *get* of the following type:

$$\begin{array}{l}\forall\tau.\forall n : \mathit{int}.\forall i : \mathit{int}.\forall\lambda. \\ \quad 0 \leq i \wedge i < n \supset (\mathit{arrayView}(\tau, n, \lambda) \wedge (\mathbf{ptr}(\lambda) * \mathbf{int}(i)) \rightarrow \mathit{arrayView}(\tau, n, \lambda) \wedge \tau)\end{array}$$

where $\mathbf{int}(I)$ is a singleton type for the integer equal to I. This type simply means that *get* is expected to return a value of type T when applied to a pointer and a natural number such that the pointer points to an array whose size is greater than the natural number and each element in the array is of type T. Obviously, for any $0 \leq i \leq n$, an array of size n at address L can be viewed as two arrays: one of size i at L and the other of size $n - i$ at $L + i$. This is what we call *view change*, which is often done implicitly and informally (and thus often incorrectly) by a programmer. In Figure 2, the proof function *splitLemma* is assigned the following functional view:

$$\begin{array}{l}\forall\tau.\forall n : \mathit{int}.\forall i : \mathit{int}.\forall\lambda. \\ \quad 0 \leq i \wedge i \leq n \supset (\mathit{arrayView}(\tau, n, \lambda) \multimap \mathit{arrayView}(\tau, i, \lambda) \otimes \mathit{arrayView}(\tau, n - i, \lambda + i))\end{array}$$

Note that *splitLemma* is recursively defined and the termination metric $\langle i \rangle$ is used to verify that *splitLemma* is terminating. Please see [Xi02] for details on such a termination verification technique. To show that *splitLemma* is a total function, we also need to verify the following pattern matching in its body:

```
prval ArraySome (pf1, pf2) = pf
```

can never fail. Similarly, we can also define a total function *unsplitLemma* that proves the following view:

$$\begin{array}{l} \forall \tau. \forall n : \textit{int}. \forall i : \textit{int}. \forall \lambda. \\ \quad 0 \leq i \wedge i \leq n \supset (\textit{arrayView}(\tau, i, \lambda) \otimes \textit{arrayView}(\tau, n-i, \lambda+i) \multimap \textit{arrayView}(\tau, n, \lambda)) \end{array}$$

With both *splitLemma* and *unsplitLemma* to support view changes, an $O(1)$-time array subscripting function is implemented in Figure 2. The definition of *get* turns into:

```
fun get (p, i) = let val x = getFirst (p + i) in x end
```

after the types and proofs in it are erased; so the function can potentially be compiled into one load instruction after it is inlined.

We organize the rest of the paper as follows. In Section 2, we formalize a language λ_{view} in which views, types and viewtypes are all supported. We then briefly mention in Section 3 an extension $\lambda_{view}^{\forall,\exists}$ of λ_{view} in which we support dependent types as well as polymorphic types, and we also present some examples to show how views can be used in practical programming. In Section 4, we present an overview of the notion of persistent stateful views, which is truly indispensable in practical programming. We use *ATS/SV* essentially for the type system that extends $\lambda_{view}^{\forall,\exists}$ with persistent stateful views. Lastly, we mention some related work and conclude.

2 Formal Development

In this section, we formally present a language λ_{view} in which the type system supports views, types and viewtypes. The main purpose of formalizing λ_{view} is to allow for a gentle introduction to unfamiliar concepts such as view and viewtype. To some extent, λ_{view} can be compared to the simply typed lambda-calculus, which forms the core of more advanced typed lambda-calculi. We will later extend λ_{view} to $\lambda_{view}^{\forall,\exists}$ with dependent types as well as polymorphic types, greatly facilitating the use of views and viewtypes in programming.

The syntax of λ_{view} is given in Figure 3. We use V for views and L for addresses. We use $\mathbf{l}_0, \mathbf{l}_1, \ldots$ for infinitely many distinct constant addresses, which one may assume to be represented as natural numbers. Also, we write l for a constant address. We use $\underline{x}$ for proof variables and $\underline{t}$ for proof terms. For each address l, $\underline{l}$ is a constant proof term, whose meaning is to become clear soon. We use Π for a proof variable context, which assigns views to proof variables.

We use T and *VT* for types and viewtypes, respectively. Note that a type T is just a special form of viewtype. We use t for dynamic terms (that is, programs) and v for values. We write Δ^i (Δ^l) for

$$
\begin{array}{llcl}
\text{addresses} & L & ::= & \mathbf{l}_0 \mid \mathbf{l}_1 \mid \ldots \\
\text{views} & V & ::= & T@L \mid \mathbf{1} \mid V_1 \otimes V_2 \mid V_1 \multimap V_2 \\
\text{proof terms} & \underline{t} & ::= & \underline{x} \mid \underline{l} \mid \langle \underline{t}_1, \underline{t}_2 \rangle \mid \mathbf{let}\ \langle \underline{x}_1, \underline{x}_2 \rangle = \underline{t}_1\ \mathbf{in}\ \underline{t}_2 \mid \lambda \underline{x}.\underline{t} \mid \underline{t}_1(\underline{t}_2) \\
\text{proof var. ctx.} & \Pi & ::= & \emptyset \mid \Pi, a : V \\
 & & & \\
\text{types} & T & ::= & \mathbf{Bool} \mid \mathbf{Int} \mid \mathbf{ptr}(L) \mid \mathbf{1} \mid V \supset \mathit{VT} \mid T * T \mid \mathit{VT} \to \mathit{VT} \\
\text{viewtypes} & \mathit{VT} & ::= & \mathbf{Bool} \mid \mathbf{Int} \mid \mathbf{ptr}(L) \mid \mathbf{1} \mid V \land \mathit{VT} \mid V \supset_0 \mathit{VT} \mid V \supset \mathit{VT} \mid \\
 & & & \mathit{VT}_1 * \mathit{VT}_2 \mid \mathit{VT} \to_0 \mathit{VT} \mid \mathit{VT} \to \mathit{VT} \\
\text{dyn. terms} & t & ::= & x \mid f \mid \mathit{cc}(t_1, \ldots, t_n) \mid \mathit{cf}(t_1, \ldots, t_n) \mid \\
 & & & \mathit{read}(\underline{t}, t) \mid \mathit{write}(\underline{t}, t_1, t_2) \mid \\
 & & & \mathbf{if}(t_1, t_2, t_3) \mid \mathit{read}(\underline{t}, t) \mid \mathit{write}(\underline{t}, t_1, t_2) \mid \\
 & & & \underline{t} \land t \mid \mathbf{let}\ \underline{x} \land x = t_1\ \mathbf{in}\ t_2 \mid \lambda \underline{x}.v \mid t(\underline{t}) \mid \\
 & & & \langle\rangle \mid \langle t_1, t_2 \rangle \mid \mathbf{let}\ \langle x_1, x_2 \rangle = t_1\ \mathbf{in}\ t_2 \mid \\
 & & & \mathbf{lam}\ x.t \mid \mathbf{app}(t_1, t_2) \mid \mathbf{fix}\ f.t \\
\text{values} & v & ::= & x \mid \mathit{cc}(v_1, \ldots, v_n) \mid \underline{t} \land v \mid \lambda \underline{x}.v \mid \langle v_1, v_2 \rangle \mid \mathbf{lam}\ x.d \\
\text{int. dyn. var. ctx.} & \Delta^i & ::= & \emptyset \mid \Delta^i, x : T \\
\text{lin. dyn. var. ctx.} & \Delta^l & ::= & \emptyset \mid \Delta^l, x : \mathit{VT} \\
\text{dyn. var. ctx.} & \Delta & ::= & (\Delta^i; \Delta^l) \\
 & & & \\
\text{state types} & \mu & ::= & [] \mid \mu[l \mapsto T] \\
\text{states} & \mathit{ST} & ::= & [] \mid \mathit{ST}[l \mapsto v]
\end{array}
$$

Figure 3: The syntax for λ_{view}

an intuitionistic (a linear) dynamic variable context, which assign types (viewtypes) to dynamic variables, and Δ for a (combined) dynamic context of the form $(\Delta^i; \Delta^l)$. Given $\Delta = (\Delta^i; \Delta^l)$, we may use $\Delta, x : \mathit{VT}$ for $(\Delta^i; \Delta^l, x : \mathit{VT})$; in case the viewtype VT is actually a type, we may also use $\Delta, x : \mathit{VT}$ for $(\Delta^i, x : \mathit{VT}; \Delta^l)$. In addition, given $\Delta_1 = (\Delta^i; \Delta^l_1)$ and $\Delta_2 = (\Delta^i; \Delta^l_2)$, we write $\Delta_1 \uplus \Delta_2$ for $(\Delta^i; \Delta^l_1, \Delta^l_2)$.

We use x and f for dynamic **lam**-variables and **fix**-variables, respectively, and xf for either x or f; a **lam**-variable is a value while a **fix**-variable is not. We use c for a dynamic constant, which is either a function cf or a constructor cc. Each constant is given a constant type (or c-type) of the form $(T_1, \ldots, T_n) \Rightarrow T$, where n is the arity of c. We may write cc for $\mathit{cc}()$. For instance, each address l is given the c-type $() \Rightarrow \mathbf{ptr}(l)$; each boolean value is given the c-type $() \Rightarrow \mathbf{Bool}$; each integer is given the c-type $() \Rightarrow \mathbf{Int}$; the equality function on integers can be given the c-type: $(\mathbf{Int}, \mathbf{Int}) \Rightarrow \mathbf{Bool}$. Also, we use $\langle\rangle$ for the unit and $\mathbf{1}$ for the unit type.

We use μ and ST for state types and states, respectively. A state type maps constant addresses to types while a state maps constant addresses to values. We use $[]$ for the empty mapping and $\mu[l \mapsto T]$ for the mapping that extends μ with a link from l to T. It is implicitly assumed that l is not in the domain $\mathbf{dom}(\mu)$ of μ when $\mu[l \mapsto T]$ is formed. Given two state types μ_1 and μ_2 with disjoint domains, we write $\mu_1 \otimes \mu_2$ for the standard union of μ_1 and μ_2. Similar notations are also applicable to states. In addition, we write $\models \mathit{ST} : \mu$ to mean that $\mathit{ST}(l)$ can be assigned the type $\mu(l)$ for each $l \in \mathbf{dom}(\mathit{ST}) = \mathbf{dom}(\mu)$.

We now present some intuitive explanation for certain unfamiliar forms of types and viewtypes. In λ_{view}, types are just a special form of viewtypes. If a dynamic value v is assigned a type, then it consumes no resources to construct v and thus v can be duplicated. For instance, an integer constant i is a value.[1] On the other hand, if a value v is assigned a viewtype, then it may consume some resources to construct v and thus v is not allowed to be duplicated. For instance, the value $(\underline{l}, l)$ can be assigned the viewtype $(\mathbf{Int}@l) \wedge \mathbf{ptr}(l)$, which is essentially for a pointer pointing to an integer; this value contains the resource $\underline{l}$ and thus cannot be duplicated.

- The difference between $V \supset_0 \mathit{VT}$ and $V \supset \mathit{VT}$ is that the former is a viewtype but not a type while the latter is a type (and thus a viewtype as well). For instance, the following type

  $$(T_1@L_1) \wedge ((T_2@L_2) \wedge (\mathbf{ptr}(L_1) * \mathbf{ptr}(L_2))) \to (T_1@L_2) \wedge ((T_1@L_2) \wedge \mathbf{1})$$

  can be assigned to the function that swaps the contents stored at L_1 and L_2. This type is essentially equivalent to the following one:

  $$(T_1@L_1) \supset ((T_2@L_2) \supset_0 (\mathbf{ptr}(L_1) * \mathbf{ptr}(L_2) \to (T_1@L_2) \wedge ((T_1@L_2) \wedge \mathbf{1})))$$

  where both $\supset_0$ and $\supset$ are involved.

- The difference between $\mathit{VT} \to_0 \mathit{VT}$ and $\mathit{VT} \to \mathit{VT}$ is rather similar to that between $V \supset_0 \mathit{VT}$ and $V \supset \mathit{VT}$.

[1] In particular, we emphasize that there are simply no "linear" integer values in λ_{view}.

$$\frac{}{\emptyset \vdash_{[l \mapsto T]} \underline{l} : T@l}\ \textbf{(vw-addr)} \qquad \frac{}{\emptyset, \underline{x} : V \vdash_{[]} \underline{x} : V}\ \textbf{(vw-var)}$$

$$\frac{}{\emptyset \vdash_{[]} \langle\rangle : \mathbf{1}}\ \textbf{(vw-unit)} \qquad \frac{\Pi_1 \vdash_{\mu_1} \underline{t}_1 : V_1 \quad \Pi_2 \vdash_{\mu_2} \underline{t}_2 : V_2}{\Pi_1, \Pi_2 \vdash_{\mu_1 \otimes \mu_2} \langle \underline{t}_1, \underline{t}_2 \rangle : V_1 \otimes V_2}\ \textbf{(vw-tup)}$$

$$\frac{\Pi_1 \vdash_{\mu_1} \underline{t}_1 : V_1 \otimes V_2 \quad \Pi_2, \underline{x}_1 : V_1, \underline{x}_2 : V_2 \vdash_{\mu_2} \underline{t}_2 : V}{\Pi_1, \Pi_2 \vdash_{\mu_1 \otimes \mu_2} \textbf{let } \langle \underline{x}_1, \underline{x}_2 \rangle = \underline{t}_1 \textbf{ in } \underline{t}_2 : V}\ \textbf{(vw-let)}$$

$$\frac{\Pi, \underline{x} : V_1 \vdash_{\mu} \underline{t} : V_2}{\Pi \vdash_{\mu} \lambda \underline{x}.\underline{t} : V_1 \multimap V_2}\ \textbf{(vw-lam)} \qquad \frac{\Pi_1 \vdash_{\mu_1} \underline{t}_1 : V_1 \multimap V_2 \quad \Pi_2 \vdash_{\mu_2} \underline{t}_2 : V_1}{\Pi_1, \Pi_2 \vdash_{\mu_1 \otimes \mu_2} \underline{t}_1(\underline{t}_2) : V_2}\ \textbf{(vw-app)}$$

Figure 4: The rules for assigning views to proof terms

- In the current implementation of ATS, viewtypes of either the form $V \supset_0 T$ or the form *VT* $\rightarrow_0$ *VT* are not directly supported (though they may be in the future). However, as far as formalization of viewtypes is concerned, we feel that eliminating such viewtypes would seem rather *ad hoc*.

The rules for assigning views to proofs are given in Figure 4. So far only logic constructs in the multiplicative fragment of intuitionistic linear logic are involved in forming views, and we plan to handle logic constructs in the additive fragment of intuitionistic linear logic in future if such a need, which we have yet to encounter in practice, occurs. A judgment of the form $\Pi \vdash_{\mu} \underline{t} : V$ means that $\underline{t}$ can be assigned the view V if the variables and constants in $\underline{t}$ are assigned views according to Π and μ, respectively.

The rules for assigning viewtypes (which include types) are given in Figure 5. We use $\supset_?$ for $\supset$ or $\supset_0$ in the rule **(ty-vapp)** and $\rightarrow_?$ for $\rightarrow$ or $\rightarrow_0$ in the rule **(ty-app)**. Intuitively, a type of the form $V \supset_?$ *VT* is for functions from proofs of view V to values of viewtype *VT*. Similarly, a type of the form $VT_1 \rightarrow_? VT_2$ is for functions from values of viewtype VT_1 to values of viewtype VT_2.

Proposition 2.1 *Assume that* $\emptyset; (\emptyset; \emptyset) \vdash_{\mu} v : T$ *is derivable. Then the state type* μ *must equal* $[]$.

Proof By a careful inspection of the rules in Figure 5. ■

Though simple, Proposition 2.1 is of great importance. Intuitively, the proposition states that if a closed value is assigned a type T, then the value can be constructed without consuming resources (in the sense of proof constants $\underline{l}$) and thus is allowed to be duplicated.

We use a judgment of the form $ST \models V$ to mean that the state ST entails the view V. The rules for deriving such judgments are given below:

$$\frac{\emptyset; (\emptyset; \emptyset) \vdash_{[]} v : T}{[l \mapsto v] \models T@l} \qquad \frac{ST_1 \models V_1 \quad ST_2 \models V_2}{ST_1 \otimes ST_2 \models V_1 \otimes V_2} \qquad \frac{ST_0 \otimes ST \models V_2 \text{ for each } ST_0 \models V_1}{ST \models V_1 \multimap V_2}$$

Lemma 2.2 *Assume* $\Pi \vdash_{\mu} \underline{t} : V$ *is derivable for* $\Pi = \underline{x}_1 : V_1, \ldots, \underline{x}_n : V_n$. *If* $ST = ST_0 \otimes ST_1 \otimes \ldots \otimes ST_n$ *and* $\models ST_0 : \mu$ *and* $ST_i \models V_i$ *for* $1 \leq n$, *then* $ST \models V$ *holds.*

$$\frac{}{\emptyset; (\Delta^i; \emptyset), x : \mathit{VT} \vdash_{[]} x : \mathit{VT}} \textbf{ (ty-var)}$$

$$\frac{\vdash c : (T_1, \ldots, T_n) \Rightarrow T \quad \Pi_i; \Delta_i \vdash_{\mu_i} t_i : T_i \text{ for } 1 \leq i \leq n}{\Pi_1, \ldots, \Pi_n; \Delta_1 \uplus \ldots \uplus \Delta_n \vdash_{\mu_1 \otimes \ldots \otimes \mu_n} c(t_1, \ldots, t_n) : T} \textbf{ (ty-cst)}$$

$$\frac{\Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : \textbf{Bool} \quad \Pi_2; \Delta_2 \vdash_{\mu_2} t_2 : \mathit{VT} \quad \Pi_2; \Delta_2 \vdash_{\mu_2} t_3 : \mathit{VT}}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \textbf{if}(t_1, t_2, t_3) : \mathit{VT}} \textbf{ (ty-if)}$$

$$\frac{\Pi_1 \vdash_{\mu_1} \underline{t} : V \quad \Pi_2; \Delta \vdash_{\mu_2} t : \mathit{VT}}{\Pi_1, \Pi_2; \Delta \vdash_{\mu_1 \otimes \mu_2} \langle \underline{t}, t \rangle : V \wedge \mathit{VT}} \textbf{ (ty-vtup)}$$

$$\frac{}{\emptyset; (\Delta^i; \emptyset) \vdash_{[]} \langle\rangle : \mathbf{1}} \textbf{ (ty-unit)} \qquad \frac{\Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : \mathit{VT}_1 \quad \Pi_2; \Delta_2 \vdash_{\mu_2} t_2 : \mathit{VT}_2}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \langle t_1, t_2 \rangle : \mathit{VT}_1 * \mathit{VT}_2} \textbf{ (ty-tup)}$$

$$\frac{\Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : \mathit{VT}_1 * \mathit{VT}_2 \quad \Pi_2; \Delta_2, x_1 : \mathit{VT}_1, x_2 : \mathit{VT}_2 \vdash_{\mu_2} t_2 : \mathit{VT}}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \textbf{let } \langle x_1, x_2 \rangle = t_1 \textbf{ in } t_2 : \mathit{VT}} \textbf{ (ty-let)}$$

$$\frac{\Pi, \underline{x} : V; \Delta \vdash_{\mu} v : \mathit{VT}}{\Pi; \Delta \vdash_{\mu} \lambda \underline{x}.v : V \supset_0 \mathit{VT}} \textbf{ (ty-vlam0)} \qquad \frac{\emptyset, \underline{x} : V; (\Delta^i; \emptyset) \vdash_{[]} v : \mathit{VT}}{\emptyset; (\Delta^i; \emptyset) \vdash_{[]} \lambda \underline{x}.v : V \supset \mathit{VT}} \textbf{ (ty-vlam)}$$

$$\frac{\Pi_1; \Delta \vdash_{\mu_1} t : V \supset_? \mathit{VT} \quad \Pi_2 \vdash_{\mu_2} \underline{t} : V}{\Pi_1, \Pi_2; \Delta \vdash_{\mu_1 \otimes \mu_2} t(\underline{t}) : \mathit{VT}} \textbf{ (ty-vapp)}$$

$$\frac{\Pi; \Delta, x : \mathit{VT}_1 \vdash_{\mu} t : \mathit{VT}_2}{\Pi; \Delta \vdash_{\mu} \textbf{lam } x.t : \mathit{VT}_1 \to_0 \mathit{VT}_2} \textbf{ (ty-lam0)} \qquad \frac{\emptyset; (\Delta^i; \emptyset), x : \mathit{VT}_1 \vdash_{[]} t : \mathit{VT}_2}{\emptyset; (\Delta^i; \emptyset) \vdash_{[]} \textbf{lam } x.t : \mathit{VT}_1 \to \mathit{VT}_2} \textbf{ (ty-lam)}$$

$$\frac{\Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : \mathit{VT}_1 \to_? \mathit{VT}_2 \quad \Pi_2; \Delta_2 \vdash_{\mu_2} t_2 : \mathit{VT}_1}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \textbf{app}(t_1, t_2) : \mathit{VT}_2} \textbf{ (ty-app)}$$

$$\frac{\Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : V \wedge \mathit{VT}_1 \quad \Pi_2, \underline{x} : V; \Delta_2, x : \mathit{VT}_1 \vdash_{\mu_2} t_2 : \mathit{VT}_2}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \textbf{let } \underline{x} \wedge x = t_1 \textbf{ in } t_2 : \mathit{VT}_2} \textbf{ (ty-vlet)}$$

$$\frac{\emptyset; (\Delta^i, f : T; \emptyset) \vdash_{[]} t : T}{\emptyset; (\Delta^i; \emptyset) \vdash_{[]} \textbf{fix } f.t : T} \textbf{ (ty-fix)}$$

$$\frac{\Pi_1 \vdash_{\mu_1} \underline{t} : T@L \quad \Pi_2; \Delta \vdash_{\mu_2} t : \textbf{ptr}(L)}{\Pi_1, \Pi_2; \Delta \vdash_{\mu_1 \otimes \mu_2} \mathit{read}(\underline{t}, t) : (T@L) \wedge T} \textbf{ (ty-read)}$$

$$\frac{\Pi_1 \vdash_{\mu_1} \underline{t} : T@L \quad \Pi_2; \Delta \vdash_{\mu_2} t_1 : \textbf{ptr}(L) \quad \Pi_3 \vdash_{\mu_3} t_2 : T'}{\Pi_1, \Pi_2, \Pi_3; \Delta \vdash_{\mu_1 \otimes \mu_2 \otimes \mu_3} \mathit{write}(\underline{t}, t_1, t_2) : (T'@L) \wedge \mathbf{1}} \textbf{ (ty-write)}$$

Figure 5: The rules for assigning viewtypes to dynamic terms

Proof By structural induction on the derivation $\mathcal{D}$ of $\Pi \vdash_\mu \underline{t} : V$. ■

We use $[\underline{x}_1, \ldots, \underline{x}_n \mapsto \underline{t}_1, \ldots, \underline{t}_n]$ for a substitution that maps $\underline{x}_i$ to $\underline{t}_i$ for $1 \leq i \leq n$. Similarly, we use $[\mathit{xf}_1, \ldots, \mathit{xf}_n \mapsto t_1, \ldots, t_n]$ for a substitution that maps xf_i to t_i for $1 \leq i \leq n$.

Definition 2.3 *We define redexes as follows:*

1. **let** $\langle \underline{x}, x \rangle = \langle \underline{t}, v \rangle$ **in** t *is a redex, and its reduction is* $t[\underline{x} \mapsto \underline{t}][x \mapsto v]$.
2. $(\lambda \underline{x}.v)(\underline{t})$ *is a redex, and its reduction is* $v[\underline{x} \mapsto \underline{t}]$.
3. **let** $\langle x_1, x_2 \rangle = \langle v_1, v_2 \rangle$ **in** t *is a redex, and its reduction is* $t[x_1, x_2 \mapsto v_1, v_2]$.
4. **app**(**lam** $x.t, v$) *is a redex, and its reduction is* $t[x \mapsto v]$.
5. **fix** $f.t$ *is a redex, and its reduction is* $t[f \mapsto$ **fix** $f.t]$.

We use E for evaluation contexts, which are defined as follows:

$$\begin{array}{lll} \text{evaluation context} \quad E & ::= & [\,] \mid c(v_1, \ldots, v_{i-1}, E, t_{i+1}, \ldots, t_n) \mid \\ & & \mathit{read}(\underline{t}, E) \mid \mathit{write}(\underline{t}, E, t) \mid \mathit{write}(\underline{t}, v, E) \mid \\ & & \underline{t} \wedge E \mid \mathbf{let}\ \underline{x} \wedge x = E\ \mathbf{in}\ t \mid E(\underline{t}) \mid \\ & & \langle E, t \rangle \mid \langle v, E \rangle \mid \mathbf{let}\ \langle x_1, x_2 \rangle = E\ \mathbf{in}\ t \mid \mathbf{app}(E, t) \mid \mathbf{app}(v, E) \end{array}$$

Given E and t, we write $E[t]$ for the dynamic term obtained from replacing the hole [] in E with t. Note that such a replacement can never cause a free variable to be captured. Given $\mathit{ST}_1, \mathit{ST}_2$ and t_1, t_2, we write $(\mathit{ST}_1, t_1) \rightarrow_{ev/st} (\mathit{ST}_2, t_2)$ if

1. $t_1 = E[t]$ and $t_2 = E[t']$ for some redex t and its reduction, or
2. $t_1 = E[\mathit{read}(\underline{t}, l)]$ for some $l \in \mathbf{dom}(\mathit{ST}_1)$ and $t_2 = E[\langle \underline{t}, \mathit{ST}_1(l) \rangle]$ and $\mathit{ST}_2 = \mathit{ST}_1$, or
3. $t_1 = E[\mathit{write}(\underline{t}, l, v)]$ for some $l \in \mathbf{dom}(\mathit{ST}_1)$ and $t_2 = E[\langle \underline{t}, \langle\rangle \rangle]$ and $\mathit{ST}_2 = \mathit{ST}_1[l := v]$.

We write $\mathit{ST}[l := v]$ for a state that maps l to v and coincides with ST elsewhere. Note that we implicitly assume $l \in \mathbf{dom}(\mathit{ST})$ when writing $\mathit{ST}[l := v]$.

We now state some lemmas and theorems involved in establishing the soundness of λ_{view}. Please see [XZ04] for details on their proofs.

Lemma 2.4 (Substitution) *We have the following:*

1. *Assume that both* $\Pi_1 \vdash_{\mu_1} \underline{t}_1 : V_1$ *and* $\Pi_2, \underline{x} : V_1 \vdash_{\mu_2} \underline{t}_2 : V_2$ *are derivable. Then* $\Pi_2 \vdash_{\mu_1 \otimes \mu_2} \underline{t}_2[\underline{x} \mapsto \underline{t}_1] : V_2$ *is also derivable.*
2. *Assume that both* $\Pi_1 \vdash_{\mu_1} \underline{t} : V$ *and* $\Pi_2, \underline{x} : V; \Delta \vdash_{\mu_2} t : \mathit{VT}$ *are derivable. Then* $\Pi_1, \Pi_2; \Delta \vdash_{\mu_1 \otimes \mu_2} t[\underline{x} \mapsto \underline{t}] : \mathit{VT}$ *is also derivable.*
3. *Assume that both* $\emptyset; (\emptyset; \emptyset) \vdash_{\mu_1} v : \mathit{VT}_1$ *and* $\Pi; \Delta, x : \mathit{VT}_1 \vdash_{\mu_2} t : \mathit{VT}_2$ *are derivable. Then* $\Pi; \Delta \vdash_{\mu_1 \otimes \mu_2} t[x \mapsto v] : \mathit{VT}_2$ *is also derivable.*

Proof By standard structural induction. In particular, we encounter a need for Proposition 2.1 when proving (3). ∎

As usual, the soundness of the type system of λ_{view} is built on top of the following two theorems:

Theorem 2.5 (Subject Reduction) *Assume $\emptyset; (\emptyset; \emptyset) \vdash_{\mu_1} t_1 : VT$ is derivable and $\models ST_1 : \mu_1$ holds. If $(ST_1, t_1) \rightarrow_{ev/st} (ST_2, t_2)$, then $\emptyset; (\emptyset; \emptyset) \vdash_{\mu_2} t_2 : VT$ is derivable for some store type μ_2 such that $\models ST_2 : \mu_2$ holds.*

Proof By structural induction on the derivation of $\emptyset; (\emptyset; \emptyset) \vdash_{\mu_1} t_1 : VT$. ∎

Theorem 2.6 (Progress) *Assume that $\emptyset; (\emptyset; \emptyset) \vdash_{\mu} t : VT$ is derivable and $\models ST : \mu$ holds. Then either t is a value or $(ST, t) \rightarrow_{ev/st} (ST', t')$ for some ST' and t' or t is of the form $E[cf(v_1, \ldots, v_n)]$ such that $cf(v_1, \ldots, v_n)$ is undefined.*

Proof By structural induction on the derivation $\mathcal{D}$ of $\emptyset; (\emptyset; \emptyset) \vdash_{\mu} t : VT$. Lemma 2.2 is needed when we handling the rules **(ty-read)** and **(ty-write)**. ∎

By Theorem 2.5 and Theorem 2.6, we can readily infer that if $\emptyset; (\emptyset; \emptyset) \vdash_{\mu} t : VT$ is derivable and $\models ST : \mu$ holds, then either the evaluation of (ST, t) reaches (ST', v) for some state ST' and value v or it continues forever.

Clearly, we can define a function $|\cdot|$ that erases all the proof terms in a given dynamic term. For instance, some key cases in the definition of the erasure function are given as follows:

$$
\begin{array}{rcl}
|\mathbf{let}\ \underline{x} \wedge x = t_1\ \mathbf{in}\ t_2| & = & \mathbf{let}\ x = |t_1|\ \mathbf{in}\ t_2 \\
|\underline{t} \wedge t| & = & |t| \\
|\lambda \underline{x}.v| & = & |v| \\
|t(\underline{t})| & = & |t| \\
|\mathit{read}(\underline{t}, t)| & = & \mathit{read}(|t|) \\
|\mathit{write}(\underline{t}, t_1, t_2)| & = & \mathit{write}(|t_1|, |t_2|)
\end{array}
$$

It is then straightforward to show that a dynamic term evaluates to a value if and only if the erasure of the dynamic term evaluates to the erasure of the value. Thus, there is no need to keep proof terms at run-time: They are only needed for the purpose of type-checking. Please see [CX05] for more details on the issue of proof erasure.

3 Extension

While it supports both views and viewtypes, λ_{view} is essentially based on the simply typed language calculus. This makes it difficult to truly reap the benefits of views and viewtypes. In this section, we outline an extension from λ_{view} to $\lambda_{view}^{\forall,\exists}$ to include universally quantified types as well as existentially quantified types, greatly facilitating the use of views and viewtypes in programming. For brevity, most of technical details are suppressed in this presentation, which is primarily for the reader to relate the concrete syntax in the examples we present to some form of formal syntax.

sorts	σ	$::=$	*addr* | *bool* | *int* | *view* | *type* | *viewtype*
static contexts	Σ	$::=$	$\emptyset \mid \Sigma, a : \sigma$
static addr.	L	$::=$	$a \mid l \mid L + I$
static int.	I	$::=$	$a \mid i \mid c_I(s_1, \ldots, s_n)$
static prop.	B	$::=$	$a \mid b \mid c_B(s_1, \ldots, s_n) \mid \neg B \mid B_1 \wedge B_2 \mid B_1 \vee B_2 \mid B_1 \supset B_2$
views	V	$::=$	$\ldots \mid B \supset V \mid \forall a : \sigma.V \mid B \wedge V \mid \exists a : \sigma.V$
types	T	$::=$	$\ldots \mid a \mid \mathbf{bool}(B) \mid \mathbf{int}(I) \mid B \supset T \mid \forall a : \sigma.T \mid B \wedge T \mid \exists a : \sigma.T$
viewtypes	VT	$::=$	$\ldots \mid B \supset \mathit{VT} \mid \forall a : \sigma.\mathit{VT} \mid B \wedge \mathit{VT} \mid \exists a : \sigma.\mathit{VT}$

Figure 6: The syntax for the statics of $\lambda^{\forall,\exists}_{view}$

Like an applied type system [Xi04a], $\lambda^{\forall,\exists}_{view}$ consists of a static component (statics) and a dynamic component (dynamics). The syntax for the statics of $\lambda^{\forall,\exists}_{view}$ is given in Figure 6. The statics itself is a simply typed language and a type in it is called a *sort*. We assume the existence of the following basic sorts in $\lambda^{\forall,\exists}_{view}$: *addr*, *bool*, *int*, *type*, *view* and *viewtype*; *addr* is the sort for addresses, and *bool* is the sort for boolean constants, and *int* is the sort for integers, and *type* is the sort for types, and *view* is the sort for views, and *viewtype* is the sort for viewtypes. We use a for static variables, l for address constants $\mathbf{l}_0, \mathbf{l}_1, \ldots$, b for boolean values *true* and *false*, and i for integers $0, -1, 1, \ldots$. We may also use $\mathbf{0}$ for the null address $\mathbf{l}_0$. A term s in the statics is called a static term, and we use $\Sigma \vdash s : \sigma$ to mean that s can be assigned the sort σ under Σ. The rules for assigning sorts to static terms are all omitted as they are completely standard.

We may also use $L, B, I, T, V, \mathit{VT}$ for static terms of sorts *addr*, *bool*, *int*, *type*, *view*, and *viewtype*, respectively. We assume some primitive functions c_I when forming static terms of sort *int*; for instance, we can form terms such as $I_1 + I_2$, $I_1 - I_2$, $I_1 * I_2$ and I_1/I_2. Also we assume certain primitive functions c_B when forming static terms of sort *bool*; for instance, we can form propositions such as $I_1 \leq I_2$ and $I_1 \geq I_2$, and for each sort σ we can form a proposition $s_1 =_\sigma s_2$ if s_1 and s_2 are static terms of sort σ; we may omit the subscript σ in $=_\sigma$ if it can be readily inferred from the context. In addition, given L and I, we can form an address $L + I$, which equals $\mathbf{l}_{n+i}$ if $L = \mathbf{l}_n$ and $I = i$ and $n + i \geq 0$.

We use $\overline{B}$ for a sequence of propositions and $\Sigma; \overline{B} \models B$ for a constraint that means for any $\Theta : \Sigma$, if each proposition in $\overline{B}[\Theta]$ holds then so does $B[\Theta]$.

In addition, we introduce two type constructors **bool** and **int**; given a proposition B, $\mathbf{bool}(B)$ is the singleton type in which the only value is the truth value of B; similarly, given an integer I, $\mathbf{int}(I)$ is the singleton type in which the only value is the integer I. Obviously, the previous types **Bool** and **Int** can now be defined as $\exists a : \mathit{bool}.\mathbf{bool}(a)$ and $\exists a : \mathit{int}.\mathbf{int}(a)$, respectively.

Some (additional) syntax for the dynamics of $\lambda^{\forall,\exists}_{view}$ is given in Figure 7. The markers $\supset^+ (\cdot)$, $\supset^- (\cdot)$, $\forall^+(\cdot)$, $\forall^-(\cdot)$, $\wedge(\cdot)$ and $\exists(\cdot)$ are primarily introduced to prove the soundness of the type system of $\lambda^{\forall,\exists}_{view}$, and please see [Xi04a] for explanation.

We can now also introduce the (built-in) memory access functions *getPtr* and *setPtr*[2] as well as

[2]The type assigned to *setPtr* is slightly different from the one in Section 1.

$$
\begin{array}{llcl}
\text{proof terms} & \underline{t} & ::= & \ldots \mid \supset^{+}(\underline{t}) \mid \supset^{-}(\underline{t}) \mid \forall^{+}(\underline{t}) \mid \forall^{-}(\underline{t}) \mid \\
 & & & \wedge(\underline{t}) \mid \textbf{let } \wedge(\underline{x}) = \underline{t}_1 \textbf{ in } \underline{t}_2 \mid \exists(\underline{t}) \mid \textbf{let } \exists(\underline{x}) = \underline{t}_1 \textbf{ in } \underline{t}_2 \\
\text{dynamic terms} & t & ::= & \ldots \mid \textbf{let } \wedge(\underline{x}) = \underline{t} \textbf{ in } t \mid \textbf{let } \exists(\underline{x}) = \underline{t} \textbf{ in } t \mid \\
 & & & \supset^{+}(v) \mid \supset^{-}(t) \mid \forall^{+}(v) \mid \forall^{-}(t) \mid \\
 & & & \wedge(t) \mid \textbf{let } \wedge(x) = t_1 \textbf{ in } t_2 \mid \exists(t) \mid \textbf{let } \exists(x) = t_1 \textbf{ in } t_2 \\
\text{values} & v & ::= & \ldots \mid \supset^{+}(v) \mid \forall^{+}(v) \mid \wedge(v) \mid \exists(v)
\end{array}
$$

Figure 7: The syntax for the dynamics of $\lambda^{\forall,\exists}_{view}$

the (built-in) memory allocation/deallocation functions *alloc* and *free* and assign them the following constant types:

$$
\begin{array}{rcl}
\textit{getPtr} & : & \forall\lambda.\forall\tau.\tau@\lambda \wedge \textbf{ptr}(\lambda) \Rightarrow \tau@\lambda \wedge \tau \\
\textit{setPtr} & : & \forall\lambda.\forall\tau.\textbf{top}@\lambda \wedge \textbf{ptr}(\lambda) * \tau \Rightarrow \tau@\lambda \wedge \mathbf{1} \\
\textit{alloc} & : & \forall\iota.\iota \geq 0 \supset (\textbf{int}(\iota) \Rightarrow \exists\lambda.\lambda \neq \mathbf{0} \wedge (\textit{arrayView}(\mathbf{1}, \iota, \lambda) \wedge \textbf{ptr}(\lambda))) \\
\textit{free} & : & \forall\tau.\forall\iota.\iota \geq 0 \supset (\textit{arrayView}(\tau, \iota, \lambda) \wedge (\textbf{ptr}(\lambda) * \textbf{int}(\iota)) \Rightarrow \mathbf{1})
\end{array}
$$

We use **top** for the top type such that every type is considered a subtype of **top**. When applied to a natural number n, *alloc* returns a pointer (that is not null) pointing to a newly allocated array of n units; when applied to a pointer pointing to an array of size n, *free* frees the array. Note that how these two functions are implemented is inessential here as long as the implementations meets the constant types assigned to them.

A judgment for assigning a view to a proof is now of the form $\Sigma; \overline{B}; \Pi \vdash_{\mu} \underline{t} : V$, and the rules in Figure 4 need to be modified properly. Intuitively, such a judgment means that $\Pi[\Theta] \vdash_{\mu} \underline{t} : V[\Theta]$ holds for any substitution $\Theta : \Sigma$ such that each B in $\overline{B}[\Theta]$ holds. Some additional rules for assigning views to proof terms are given in Figure 8. Similarly, a judgment for assigning a viewtype to a dynamic term is now of the form $\Sigma; \overline{B}; \Pi; \Delta \vdash t : \textit{VT}$, and the rules in Figure 5 need to be modified properly. Some additional rules for assigning viewtypes to dynamic terms are given in Figure 9.

Given the development detailed in [Xi04a], it is a standard routine to establish the soundness of the type system of $\lambda^{\forall,\exists}_{view}$. The challenge here is really not in the proof of the soundness; it is instead in the formulation of the rules presented in Figure 4, Figure 8, Figure 5 and Figure 9 for assigning views and viewtypes to proof terms and dynamic terms, respectively. We are now ready to present some running examples taken from the current implementation of ATS.

A clearly noticeable weakness in many typed programming languages lies in the treatment of the allocation and initialization of arrays (and many other data structures). For instance, the allocation and initialization of an array in SML is atomic and cannot be done separately. Therefore, copying an array requires a new array be allocated and then initialized before copying can actually proceed. Though the initialization of the newly allocated array is completely useless, it unfortunately cannot be avoided. In $\lambda^{\forall,\exists}_{view}$ (extended with recursive stateful views), a function of the following type can be readily implemented that replaces elements of type T_1 in an array with

$$\frac{\Sigma; \overline{B}, B; \Pi \vdash_{\mu} \underline{t} : V}{\Sigma; \overline{B}; \Pi \vdash_{\mu} \supset^{+} (\underline{t}) : B \supset V} \ \textbf{(vw-}\supset\textbf{+)} \qquad \frac{\Sigma; \overline{B}; \Pi \vdash_{\mu} \underline{t} : B \supset V \quad \Sigma; \overline{B} \models B}{\Sigma; \overline{B}; \Pi \vdash_{\mu} \supset^{-} (\underline{t}) : V} \ \textbf{(vw-}\supset\textbf{-)}$$

$$\frac{\Sigma, a : \sigma; \overline{B}; \Pi \vdash_{\mu} \underline{t} : V}{\Sigma; \overline{B}; \Pi \vdash_{\mu} \forall^{+}(\underline{t}) : \forall a : \sigma.V} \ \textbf{(vw-}\forall\textbf{+)} \qquad \frac{\Sigma; \overline{B}; \Pi \vdash_{\mu} \underline{t} : \forall a : \sigma.V \quad \Sigma \vdash s : \sigma}{\Sigma; \overline{B}; \Pi \vdash_{\mu} \forall^{-}(\underline{t}) : V[a \mapsto s]} \ \textbf{(vw-}\forall\textbf{-)}$$

$$\frac{\Sigma; \overline{B} \models B \quad \Sigma; \overline{B}; \Pi \vdash_{\mu} \underline{t} : V}{\Sigma; \overline{B}; \Pi \vdash_{\mu} \wedge(\underline{t}) : B \wedge V} \ \textbf{(vw-}\wedge\textbf{+)}$$

$$\frac{\Sigma; \overline{B}; \Pi_1 \vdash_{\mu_1} \underline{t}_1 : B \wedge V_1 \quad \Sigma; \overline{B}, B; \Pi_2, \underline{x} : V_1 \vdash_{\mu_2} \underline{t}_2 : V_2}{\Sigma; \overline{B}; \Pi_1, \Pi_2 \vdash_{\mu_1 \otimes \mu_2} \textbf{let } \wedge(\underline{x}) = \underline{t}_1 \textbf{ in } \underline{t}_2 : V_2} \ \textbf{(vw-}\wedge\textbf{-)}$$

$$\frac{\Sigma \vdash s : \sigma \quad \Sigma; \overline{B}; \Pi \vdash_{\mu} \underline{t} : V[a \mapsto s]}{\Sigma; \overline{B}; \Pi \vdash_{\mu} \exists(\underline{t}) : \exists a : \sigma.V} \ \textbf{(vw-}\exists\textbf{+)}$$

$$\frac{\Sigma; \overline{B}; \Pi_1 \vdash_{\mu_1} \underline{t}_1 : \exists a : \sigma.V_1 \quad \Sigma, a : \sigma; \overline{B}; \Pi_2, \underline{x} : V_1 \vdash_{\mu_2} \underline{t}_2 : V_2}{\Sigma; \overline{B}; \Pi_1, \Pi_2 \vdash_{\mu_1 \otimes \mu_2} \textbf{let } \exists(\underline{x}) = \underline{t}_1 \textbf{ in } \underline{t}_2 : V_2} \ \textbf{(vw-}\exists\textbf{-)}$$

Figure 8: Some additional rules for assigning views to proof terms

elements of type T_2 when a function of type $T_1 \to T_2$ is given:

$$\begin{array}{l} \forall\tau_1.\forall\tau_2.\forall\iota.\forall\lambda. \\ \quad \iota \geq 0 \supset (\mathit{arrayView}(\tau_1, \iota, \lambda) \wedge (\mathbf{ptr}(\lambda) * \mathbf{int}(\iota) * (\tau_1 \to \tau_2)) \to \mathit{arrayView}(\tau_2, \iota, \lambda) \wedge \mathbf{ptr}(\lambda)) \end{array}$$

With such a function, the allocation and initialization of an array can clearly be separated. In Figure 10, we present an implementation of in-place array map function in ATS. Note that *arrayView* is declared as a recursive stateful view constructor in Figure 1.[3] Note that for a proof *pf* of view *arrayView*(T, I, L) for some type T, integer $I > 0$ and address L, the following syntax in Figure 10 means that *pf* is decomposed into two proofs pf_1 and pf_2 of views $T@L$ and *arrayView*$(T, I-1, L+1)$, respectively:

```
prval ArraySome (pf1, pf2) = pf
```

The rest of the syntax in Figure 10 should then be easily accessible.

The next example we present is in Figure 11, where a recursive view constructor *slsegView* is declared. Note that we write $(T_0, \ldots, T_n)@L$ for a sequence of views: $T_0@(L+0), \ldots, T_n@(L+n)$. Given a type T, an integer I, and two addresses L_1 and L_2, *slsegView*(T, I, L_1, L_2) is a view for a singly-linked list segment pictured as follows:

[3]Though the notion of recursive stateful view is not present in $\lambda^{\forall,\exists}_{view}$, it should be understood that this notion can be readily incorporated.

$$\frac{\begin{array}{c}\Sigma; \overline{B}; \Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : \textbf{bool}(B) \\ \Sigma; \overline{B}, B; \Pi_2; \Delta_2 \vdash_{\mu_2} t_2 : \textbf{\textit{VT}} \quad \Sigma; \overline{B}, \neg B; \Pi_2; \Delta_2 \vdash_{\mu_2} t_3 : \textbf{\textit{VT}}\end{array}}{\Sigma; \overline{B}; \Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \textbf{if}(t_1, t_2, t_3) : \textbf{\textit{VT}}} \textbf{(ty-if)}$$

$$\frac{\Sigma; \overline{B}, B; \Pi; \Delta \vdash_\mu t : \textbf{\textit{VT}}}{\Sigma; \overline{B}; \Pi; \Delta \vdash_\mu \supset^+(t) : B \supset \textbf{\textit{VT}}} \textbf{(ty-}\supset\textbf{+)} \qquad \frac{\Sigma; \overline{B}; \Pi; \Delta \vdash_\mu t : B \supset \textbf{\textit{VT}} \quad \Sigma; \overline{B} \models B}{\Sigma; \overline{B}; \Pi; \Delta \vdash_\mu \supset^-(t) : \textbf{\textit{VT}}} \textbf{(ty-}\supset\textbf{-)}$$

$$\frac{\Sigma, a : \sigma; \overline{B}; \Pi; \Delta \vdash_\mu t : \textbf{\textit{VT}}}{\Sigma; \overline{B}; \Pi; \Delta \vdash_\mu \forall^+(t) : \forall a : \sigma.\textbf{\textit{VT}}} \textbf{(ty-}\forall\textbf{+)} \qquad \frac{\Sigma; \overline{B}; \Pi; \Delta \vdash_\mu t : \forall a : \sigma.\textbf{\textit{VT}} \quad \Sigma \vdash s : \sigma}{\Sigma; \overline{B}; \Pi; \Delta \vdash_\mu \forall^-(t) : \textbf{\textit{VT}}[a \mapsto s]} \textbf{(ty-}\forall\textbf{-)}$$

$$\frac{\Sigma; \overline{B} \models B \quad \Sigma; \overline{B}; \Pi; \Delta \vdash_\mu t : \textbf{\textit{VT}}}{\Sigma; \overline{B}; \Pi; \Delta \vdash_\mu \wedge(t) : B \wedge \textbf{\textit{VT}}} \textbf{(ty-}\wedge\textbf{+)}$$

$$\frac{\Sigma; \overline{B}; \Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : B \wedge \textbf{\textit{VT}}_1 \quad \Sigma; \overline{B}, B; \Pi_2; \Delta_2, x : \textbf{\textit{VT}}_1 \vdash_{\mu_2} t_2 : \textbf{\textit{VT}}_2}{\Sigma; \overline{B}; \Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \textbf{let } \wedge(x) = t_1 \textbf{ in } t_2 : \textbf{\textit{VT}}_2} \textbf{(ty-}\wedge\textbf{-)}$$

$$\frac{\Sigma \vdash_\mu s : \sigma \quad \Sigma; \overline{B}; \Pi; \Delta \vdash_\mu t : \textbf{\textit{VT}}[a \mapsto s]}{\Sigma; \overline{B}; \Pi; \Delta \vdash_\mu \exists(t) : \exists a : \sigma.\textbf{\textit{VT}}} \textbf{(ty-}\exists\textbf{+)}$$

$$\frac{\Sigma; \overline{B}; \Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : \exists a : \sigma.\textbf{\textit{VT}}_1 \quad \Sigma, a : \sigma; \overline{B}; \Pi_2; \Delta_2, x : \textbf{\textit{VT}}_1 \vdash_{\mu_2} t_2 : \textbf{\textit{VT}}_2}{\Sigma; \overline{B}; \Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \textbf{let } \exists(x) = t_1 \textbf{ in } t_2 : \textbf{\textit{VT}}_2} \textbf{(ty-}\exists\textbf{-)}$$

Figure 9: Some additional rules for assigning viewtypes to dynamic terms

```
fun arrayMap {a1: type, a2: type, n: int, l: addr | n >= 0}
    (pf: arrayView (a1, n, l) | A: ptr l, n: int n, f: a1 -> a2)
  : ’(arrayView (a2, n, l) | unit) =
  if n igt 0 then // [igt]: the greater-than function on integers
    let
       prval ArraySome (pf1, pf2) = pf
       val ’(pf1 | v) = getPtr (pf1 | A)
       val ’(pf1 | _) = setPtr (pf1 | A, f v)
       // [ipred]: the predessor function on integers
       val ’(pf2 | _) = arrayMap (pf2 | A + 1, ipred n, f)
    in
       ’(ArraySome (pf1, pf2) | ’())
    end
  else let prval ArrayNone () = pf in ’(ArrayNone () | ’()) end
```

Figure 10: An implementation of in-place array map function

such that (1) each element of the segment is of type T, and (2) the length of the segment is I, and (3) the segment starts at L_1 and ends at L_2. There are two view proof constructors *SlsegNone* and *SlsegSome* associated with *slsegView*. A singly-linked list is just a special case of singly-linked list segment that ends at the null address. Therefore, *sllistView*(T, I, L) is a view for a singly-linked list pictured as follows:

such that each element in it is of type T and its length is I. To demonstrate how such a view

```
dataview slsegView (type, int, addr, addr) =
  | {a:type, l:addr} SlsegNone (a, 0, l, l)
  | {a:type, n:int, first:addr, next:addr, last:addr | n >= 0, first <> null}
    // 'first <> null' is added so that nullity test can
    // be used to check whether a list segment is empty.
    SlsegSome (a, n+1, first, last) of
      ((a, ptr next) @ first, slsegView (a, n, next, last))

viewdef sllistView (a:type, n:int, l:addr) = slsegView (a, n, l, null)

fun reverse {a:type, n:int, l:addr | n >= 0} // in-place singly-linked list reversal
   (pf: sllistView (a, n, l) | p: ptr l)
  : [l: addr] '(sllistView (a, n, l) | ptr l) =
  let
     fun rev {n1:int,n2:int,l1:addr,l2:addr | n1 >= 0, n2 >= 0}
        (pf1: sllistView (a,n1,l1), pf2: sllistView (a,n2,l2) |
         p1: ptr l1, p2: ptr l2)
       : [l:addr] '(sllistView (a, n1+n2, l) | ptr l) =
       if isNull p2 then let prval SlsegNone () = pf2 in '(pf1 | p1) end
       else let
          prval SlsegSome (pf21, pf22) = pf2
          prval '(pf210, pf211) = pf21
          val '(pf211 | next) = getPtr (pf211 | p2 + 1)
          val '(pf211 | _) = setPtr (pf211 | p2 + 1, p1)
          prval pf1 = SlsegSome ('(pf210, pf211), pf1)
       in rev (pf1, pf22 | p2, next) end
  in
     rev (SlsegNone (), pf | null, p)
  end
```

Figure 11: An implementation of in-place singly-linked list reversal

can be used in programming, we implement an in-place reversal function on singly-linked-lists in Figure 11, which is given the following type:

$$\forall\tau.\forall\iota.\forall\lambda.\iota \geq 0 \supset (\mathit{sllistView}(\tau,\iota,\lambda) \wedge \mathbf{ptr}(\lambda) \rightarrow \exists\lambda(\mathit{sllistView}(\tau,\iota,\lambda) \wedge \mathbf{ptr}(\lambda)))$$

indicating that this is a length-preserving function.

4 Persistent Stateful Views

There is so far an acute problem with $\lambda^{\forall,\exists}_{view}$ that we have not mentioned. Given the linearity of stateful views, we simply can not support pointer sharing in $\lambda^{\forall,\exists}_{view}$. For instance, the kind of references in ML, which require pointer sharing, can not be directly handled.[4] This problem would impose a crippling limitation on stateful views if it could not be properly resolved. Fortunately, We have already found a solution to the problem by introducing a notion of persistent stateful views, and *ATS/SV* is essentially the type system that extends $\lambda^{\forall,\exists}_{view}$ with persistent stateful views. For the sake of brevity, we refer the reader to [XZL05] for the detailed theoretical development of persistent stateful views. In the following, we briefly present some simple intuition behind our solution.

[4]We can certainly add into $\lambda^{\forall,\exists}_{view}$ some primitives to support references, but such a solution is inherently *ad hoc*.

```
// [getPtr0] is of the type: {a:type, l:addr} (a @ l | (*none*) | ptr l) -> a
fun getPtr0 {a:type, l:addr} (pf: a @ l | (*none*) | p: ptr l): a =
  getPtr (pf | p)

// [setPtr0] is of the type: {a:type, l:addr} (a @ l | (*none*) | ptr l, a) -> unit
fun setPtr0 {a:type, l:addr} (pf: a @ l | (*none*) | p: ptr l, x: a): unit =
  setPtr (pf | p, x)

fun getRef {a:type} (r: ref a): a =
  let val '(pf | p) = r in getPtr0 (pf | (*none*) | p) end

fun setRef {a:type} (r: ref a, x: a): unit =
  let val '(pf | p) = r in setPtr0 (pf | (*none*) | p, x) end
```

Figure 12: Implementing references

We start with an implementation of references in ATS. Essentially, references can be regarded as a special form of pointers such that we have no obligation to provide proofs (of views) when reading from or writing to them. In ATS, the type constructor *ref* for forming types for references can be defined as follows:

```
// [!] stands for a box
typedef ref (a: type) = [l:addr] '(!(a @ l) | ptr l)
```

In formal notation, given a type T, *ref*(T) is defined to be $\exists\lambda.(\Box(T@\lambda) \mid \mathbf{ptr}(\lambda))$, where $\Box(T@\lambda)$ is a *persistent* stateful view. When compared to the original stateful views, which we now refer to as *ephemeral* stateful views, persistent stateful views are intuitionistic and proofs of such views can be duplicated. Given an ephemeral view V, we can construct a persistent view $\Box V$, which we may refer to as the boxed V. Note that we use `!` for $\Box$ in the concrete syntax. At this point, we emphasize that it is incorrect to assume that a persistent view $\Box V$ implies the ephemeral view V. Essentially, $\Box$ acts as a form of modality, which restricts the use of a boxed view.

Given a view V, we say that a function of type $V \wedge T_1 \to V \wedge T_2$ treats the view V as an invariant since the function consumes a proof of V and then produces another proof of the same V. What we can formally show is that such a function can also be used as a function of type $\Box V \wedge T_1 \to T_2$. For instance, the function *getPtr*, which is given the type $\forall\tau.\forall\lambda.(\tau@\lambda) \wedge \mathbf{ptr}(\lambda) \to (\tau@\lambda) \wedge \tau$, can be used as a function (*getRef*) of type $\forall\tau.\forall\lambda.\Box(\tau@\lambda) \wedge \mathbf{ptr}(\lambda) \to \tau$ to read from a reference. Similarly, we can form a function (*setRef*) of type $\forall\tau.\forall\lambda.\Box(\tau@\lambda) \wedge (\mathbf{ptr}(\lambda) * \tau) \to \mathbf{1}$ for writing to a reference. The actual implementation of *getRef* and *setRef* are given in Figure 12. Note that in the concrete syntax, we write `(V1 | V2 | VT1) -> VT2` for

```
(V1, V2 | VT1) -> '(V1 | VT2)
```

so as to indicate that V_1 is an invariant. Therefore, *getPtr*$_0$ and *setPtr*$_0$ in Figure 12 are declared to be of the following types:

$$\begin{array}{lcl} \mathit{getPtr}_0 & : & \forall\tau.\forall\lambda.(\tau@\lambda) \wedge \mathbf{ptr}(\lambda) \to (\tau@\lambda) \wedge \tau \\ \mathit{setPtr}_0 & : & \forall\tau.\forall\lambda.(\tau@\lambda) \wedge (\mathbf{ptr}(\lambda) * \tau_2) \to (\tau@\lambda) \wedge \mathbf{1} \end{array}$$

Next we show that both product and sum types are implementable in ATS through the use of persistent stateful views.

4.1 Implementing Product and Sum

In ATS, both product and sum are implementable in terms of other primitive constructs. For instance, product and sum are implemented in Figure 13. In the implementation, the type *pair*(T_1, T_2) for a pair with the first and second components of types T_1 and T_2, respectively, is defined to be:

$$\exists\lambda.(\Box(T_1@\lambda) \land \Box(T_2@\lambda + 1)) \land \mathbf{ptr}(\lambda)$$

The function *makePair* is given the type $\forall\tau_1.\forall\tau_2.(\tau_1, \tau_2) \rightarrow \mathit{pair}(\tau_1, \tau_2)$, that is, it takes values of types T_1 and T_2 to form a value of type *pair*(T_1, T_2).[5]. Note that *viewbox* is a primitive that turns an ephemeral stateful view V into a persistent stateful view $\Box V$.

The implementation of sum is more interesting. We define $T_1 + T_2$ to be $\exists\iota.\exists\lambda.(\iota = 0 \lor \iota = 1) \supset V \land \mathbf{ptr}(\lambda)$, where V is given as follows:

$$(\Box(\mathbf{int}(\iota)@\lambda) \land (\iota = 0 \supset \Box(T_1@\lambda + 1)) \land (\iota = 1 \supset \Box(T_2@\lambda + 1)))$$

Note the use of guarded persistent stateful views here. Essentially, a value of type $T_1 + T_2$ is represented as a tag (which is an integer of value 0 or 1) followed by a value of type T_1 or T_2 determined by the value of the tag. Both the left and right injections can be implemented straightforwardly. Given that recursive types are available in ATS, datatypes as supported in ML can all be readily implemented in a manner similar to the implementation of sum.

5 Current Status of ATS

We have finished a running implementation of ATS, which is currently available on-line [Xi08], and the type system presented in this paper is a large part of the implementation. At this moment, well-typed programs in ATS are interpreted. We have so far gather some empirical evidence in support of the practicality of programming with stateful views. For instance, the library of ATS alone already contains more than 20,000 lines of code written in ATS itself, involving a variety of data structures such as cyclic linked lists and doubly-linked binary trees that make (sophisticated) use of pointers. In particular, the library code includes a portion modeled after the Standard Template Library (STL) of C++ [PSL00], and the use of stateful views (both ephemeral and persistent) is ubiquitous in this portion of code.

[5] In ATS, we support functions of multiple arguments, which should be distinguished from functions that takes a tuple as a single argument.

```
typedef pair (a1: type, a2: type) = [l: addr] '(!(a1 @ l), !(a2 @ l+1) | ptr l)

fun makePair {a1:type, a2:type} (x1: a1, x2: a2): pair (a1, a2) =
  let
     val '(pf | p) = alloc (2)
     prval ArraySome (pf1, ArraySome (pf2, ArrayNone ())) = pf
     val '(pf1 | _) = setPtr (pf1 | p, x1)
     val '(pf2 | _) = setPtr (pf2 | p + 1, x2)
  in
     '(viewbox pf1, viewbox pf2 | p)
  end

fun getFst {a1:type, a2:type} (p: pair (a1, a2)): a1 =
  let val '(pf1, _ | p0) = p in getPtr0 (pf1 | (*none*) | p0) end

fun getSnd {a1:type, a2:type} (p: pair (a1, a2)): a2 =
  let val '(_, pf2 | p0) = p in getPtr0 (pf2 | (*none*) | p0 + 1) end

typedef sum (a1: type, a2: type) =
  [l: addr, i: int | i == 0 || i == 1]
    '(!(int (i) @ l), {i == 0} !(a1 @ l+1), {i == 1} !(a2 @ l+1) | ptr l)

// left injection
fun inl {a1: type, a2: type} (x: a1): sum (a1, a2) =
  let
     val '(pf | p) = alloc (2)
     prval ArraySome (pf1, ArraySome (pf2, ArrayNone ())) = pf
     val '(pf1 | _) = setPtr (pf1 | p, 0)
     val '(pf2 | _) = setPtr (pf2 | p + 1, x)
  in
     '(viewbox pf1, viewbox pf2, '() | p)
  end

// right injection
fun inr {a1: type, a2: type} (x: a2): sum (a1, a2) =
  let
     val '(pf | p) = alloc (2)
     prval ArraySome (pf1, ArraySome (pf2, ArrayNone ())) = pf
     val '(pf1 | _) = setPtr (pf1 | p, 1)
     val '(pf2 | _) = setPtr (pf2 | p + 1, x)
  in
     '(viewbox pf1, '(), viewbox pf2 | p)
  end
```

Figure 13: implementations of product and sum

6 Related Work and Conclusion

A fundamental issue in programming is on program verification, that is, verifying (in an effective manner) whether a program meets its specification. In general, existing approaches to program verification can be classified into two categories. In one category, the underlying theme is to develop a proof theory based Floyd-Hoare logic (or its variants) for reasoning about imperative stateful programs. In the other category, the focus is on developing a type theory that allows the use of types in capturing program properties.

While Floyd-Hoare logic has been studied for at least three decades [Hoa69, Hoa71], its actual use in general software practice is rare. In the literature, Floyd-Hoare logic is mostly employed to prove the correctness of some (usually) short but often intricate programs, or to identify some subtle problems in such programs. In general, it is still as challenging as it was to support Floyd-Hoare logic in a realistic programming language. On the other hand, the use of types in capturing program invariants is wide spread. For instance, types play a significant rôle in many modern programming languages such as ML and Java. However, we must note that the types in these programming languages are of relatively limited expressive power when compared to Floyd-Hoare logic. In Martin-Löf's constructive type theory [Mar84, NPS90], dependent types offer a precise means to capture program properties, and complex specifications can be expressed in terms of dependent types. If programs can be assigned such dependent types, they are *guaranteed* to meet the specifications. However, because there exists no separation between programs and types, that is, programs may be used to construct types, a language based on Martin-Löf's type theory is often too pure and limited to be useful for practical purpose.

In Dependent ML (DML), a restricted form of dependent types is proposed that completely separates programs from types, this design makes it rather straightforward to support realistic programming features such as general recursion and effects in the presence of dependent types. Subsequently, this restricted form of dependent types is used in designing Xanadu [Xi00] and DTAL [XH01] so as to reap similar benefits from dependent types in imperative programming. In hindsight, the type system of Xanadu can be viewed as an attempt to combine type theory with Floyd-Hoare logic.

In Xanadu, we follow a strategy in Typed Assembly Language (TAL) [MWCG99] to statically track the changes made to states during program evaluation. A fundamental limitation we encountered is that this strategy only allows the types of the values stored at a fixed number of addresses to be tracked in any given program, making it difficult, if not entirely impossible, to handle data structures such as linked lists in which there are an indefinite number of pointers involved. We have seen several attempts made to address this limitation. In [SRW98], finite shape graphs are employed to approximate the possible shapes that mutable data structures (e.g., linked lists) in a program can take on. A related work [WM00] introduces the notion of alias types to model mutable data structures such as linked lists. However, the notion of view changes in *ATS/SV* is not present in these works. For instance, an alias type can be readily defined for circular lists, but it is rather unclear how to program with such an alias type. As a comparison, a view can be defined as follows in *ATS/SV* for circular lists of length n:

```
viewdef circlistView (a:type,n:int,l:addr) = slsegView (a,n,l,l)
```

With properly defined functions for performing view changes, we can easily program with circular lists. For instance, we have finished a queue implementation based on circular lists [Xi08].

Along a related but different line of research, separation logic [Rey02] has recently been introduced as an extension to Hoare logic in support of reasoning on mutable data structures. The effectiveness of separation logic in establishing program correctness is convincingly demonstrated in various nontrivial examples (e.g., singly-linked lists and doubly-linked lists). It can be readily noticed that proofs formulated in separation logic in general correspond to the functions in *ATS/SV* for performing view changes, though a detailed analysis is yet to be conducted. In a broad sense, *ATS/SV* can be viewed as a novel attempt to combine type theory with (a form of) separation logic. In particular, the treatment of functions as first-class values is a significant part of *ATS/SV*, which is not addressed in separation logic. Also, we are yet to see programming languages (or systems) that can effectively support the use of separation logic in practical programming.

There is a large body of research on applying linear type theory based on linear logic [Gir87] to memory management (e.g. [Wad90, CGR96, TW99, Kob99, IK00, Hof00]), and the work [PHCP03] that attempts to give an account for data layout based on ordered linear logic [PP99] is closely related to *ATS/SV* in the aspect that memory allocation and data initialization are completely separated. However, due to the rather limited expressiveness of ordered linear logic, it is currently unclear how recursive data structures such as arrays and linked lists can be properly handled.

There have been a large number of studies on verifying program safety properties by tracking state changes. For instance, Cyclone [JMG$^+$01] allows the programmer to specify safe stack and region memory allocation; both CQual [FTA02] and Vault [FD02] support some form of resource usage protocol verification; ESC [Det96] enables the programmer to state various sorts of program invariants and then employs theorem proving to prove them; CCured [NMW02] uses program analysis to show the safety of mostly unannotated C programs. In particular, the type system of Vault also rests on (a form of) linear logic, where two constructs *adoption* and *focus* are introduced to reduce certain conflicts between linearity and sharing. Essentially, *focus* temporarily provides a linear view on an object of nonlinear type while *adoption* does the opposite, and our treatment of persistent stateful views bears some resemblance to this technique. However, the underlying approaches taken in these mentioned studies are in general rather different from ours and a direct comparison seems difficult.

In this paper, we are primarily interested in providing a framework based on type theory to reason about program states. This aspect is also shared in the research on an effective theory of type refinements [MWH03], where the aim is to develop a general theory of type refinements for reasoning about program states. Also, the formalization of *ATS/SV* bears considerable resemblance to the formalization of the type system in [MWH03]. This can also be said about the work in [AW03], where the notion of primitive stateful view like $T@L$ is already present and there are also various logic connectives for combining primitive stateful views. However, the notions such as recursive stateful views (e.g., *arrayView*) and view changes, which constitute the key contributions of this paper, have no counterparts in either [MWH03] or [AW03]. Recently, a linear language with locations (L^3) is presented by Morrisett et al [MAF05], which attempts to explore foundational typing support for strong updates. In L^3, stateful views of the form $T@L$ are present, but recursive stateful views are yet to be developed. The notion of *freeze* and *thaw* in L^3 seems to be closely related to our handling of persistent stateful views, but we have also noticed some fundamental differences. For instance, the function *viewbox* that turns an ephemeral stateful view into a persistent stateful view seems to have no counterpart in L^3.

Another line of related studies are in the area of shape analysis [SRW98, LAS00, SRW02]. While we partly share the goal of shape analysis, the approach we take is radically different from the one underlying shape analysis. Generally speaking, TVLA performs fixed-point iterations over abstract descriptions of memory layout. While it is automatic (after an operational semantics is specified in 3-valued logic for a collection of primitive operations on the data structure in question), it may lose precision when performing fixed-point iteration and thus falsely reject programs. Also, many properties that can be captured by types in *ATS/SV* seem to be beyond the reach of TVLA. For instance, the type of the list reversal function in Figure 11 states that it is length-preserving, but this is difficult to do in TVLA. Overall, it probably should be said that *ATS/SV* (type theory) and shape analysis (static analysis) are complementary.

In [MS01], a framework is presented for verifying partial program specifications in order to capture type and memory errors as well as to check data structure invariants. In general, a data structure can be handled if it can be described in terms of graph types [KS93]. Programs are annotated with partial specifications expressed in Pointer Assertion Logic. In particular, loop and function call invariants are required in order to guarantee the decidability of verification. This design is closely comparable to ours given that invariants correspond to types and verification corresponds to type-checking. However, arithmetic invariants are yet to be supported in the framework.

In summary, we have presented the design and formalization of a type system *ATS/SV* that makes use of stateful views in capturing invariants in stateful programs that may involve (sophisticated) pointer manipulation. We have not only established the type soundness of *ATS/SV* but also given a variety of running examples in support of programming with stateful views. We are currently keen to continue the effort in building the programming language ATS, making it suitable for both high-level and low level programming. With *ATS/SV*, we believe that a solid step toward reaching this goal is made.

References

[AW03] Amal Ahmed and David Walker. The Logical Approach to Stack Typing. In *Proceedings of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, pages 74–85. ACM press, January 2003.

[CGR96] Jawahar Chirimar, Carl A. Gunter, and G. Riecke. Reference Counting as a Computational Interpretation of Linear Logic. *Journal of Functional Programming*, 6(2):195–244, 1996.

[CSX04] Chiyan Chen, Rui Shi, and Hongwei Xi. A Typeful Approach to Object-Oriented Programming with Multiple Inheritance. In *Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages*, Dallas, TX, June 2004. Springer-Verlag LNCS vol. 3057.

[CX03] Chiyan Chen and Hongwei Xi. Meta-Programming through Typeful Code Representation. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 169–180, Uppsala, Sweden, August 2003.

[CX05] Chiyan Chen and Hongwei Xi. Combining Programming with Theorem Proving. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 66–77, Tallinn, Estonia, September 2005.

[Det96] David Detlefs. An overview of the extended static checking system. In *Workshop on Formal Methods in Software Practice*, 1996.

[FD02] M. Fahndrich and R. Deline. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 13–24, Berlin, June 2002.

[FTA02] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive Type Qualifiers. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, June 2002.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[Hoa71] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In Erwin Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer-Verlag, Berlin, 1971.

[Hof00] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, Winter 2000.

[IK00] Atsushi Igarashi and Naoki Kobayashi. Garbage Collection Based on a Linear Type System. In *Preliminary Proceedings of the 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC '00)*, September 2000.

[JMG$^+$01] Trevor Jim, Greg Morrisett, Dan Grossman, Mike Hicks, Mathieu Baudet, Matthew Harris, and Yanling Wang. Cyclone, a Safe Dialect of C, 2001. Available at: `http://www.cs.cornell.edu/Projects/cyclone/`.

[Kob99] Naoki Kobayashi. Quasi-linear types. In *Proceedings of the 26th ACM Sigplan Symposium on Principles of Programming Languages (POPL '99)*, pages 29–42, San Antonio, Texas, USA, 1999.

[KS93] Nils Klarlund and Michael Schwartzbach. Graph types. In *Proceedings of 20th Annual ACM Symposium on Principles of Programming Languages (POPL '93)*, pages 49–54, January 1993.

[LAS00] Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In *Proceedings of the 7th Static Analysis Symposium (SAS '00)*, 2000.

[MAF05] Greg Morrisett, Amal Ahmed, and Matthew Fluet. L^3: A Linear language with locations. In *Proceedings of Typed Lambda Calculi and Applications (TLCA'05)*, Nara, Japan, April 2005. To appear.

[Mar84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, Italy, 1984.

[MS01] Anders Møller and Michael Schwartzbach. The pointer assertion logic. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, 2001.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[MWH03] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 213–226, Uppsala, Sweden, September 2003.

[NMW02] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 128–139, London, January 2002.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*, volume 7 of *International Series of Monographs on Computer Science*. Clarendon Press, Oxford, 1990.

[PHCP03] Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A Type Theory for Memory Allocation and Data Layout. In *Proceedings of the 30th ACM Sigplan Symposium on Principles of Programming Languages (POPL '03)*, pages 172–184, New Orleans, Louisiana, USA, January 2003.

[PP99] Jeff Polakow and Frank Pfenning. Relating natural deduction and sequent calculus for intuitionistic non-commutative linear logic. In Andre Scedrov and Achim Jung, editors, *Proceedings of the 15th Conference on Mathematical Foundations of Programming Semantics*, New Orleans, Louisiana, April 1999. Electronic Notes in Theorectical Computer Science, Volume 20.

[PSL00] P. J. Plauger, Alexander A. Stepanov, and Meng Lee. *The C++ Standard Template Library*. Prentice Hall PTR, December 2000.

[Rey02] John Reynolds. Separation Logic: a logic for shared mutable data structures. In *Proceedings of 17th IEEE Symposium on Logic in Computer Science (LICS '02)*, pages 55–74. IEEE Computer Society, 2002.

[SRW98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating . *ACM Transactions of Programming Languages and Systems (TOPLAS)*, 20(1):1–50, January 1998.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *ACM Transactions of Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[TW99] David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1–2):231–248, October 1999.

[Wad90] Philip Wadler. Linear types can change the world. In *TC 2 Working Conference on Programming Concepts and Methods (Preprint)*, pages 546–566, Sea of Galilee, 1990.

[WM00] David Walker and Greg Morrisett. Alias Types for Recursive Data Structures. In *Proceedings of International Workshop on Types in Compilation*, pages 177–206. Springer-Verlag LNCS vol. 2071, September 2000.

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, LA, January 2003. ACM press.

[XH01] Hongwei Xi and Robert Harper. A Dependently Typed Assembly Language. In *Proceedings of International Conference on Functional Programming*, pages 169–180, September 2001.

[Xi00] Hongwei Xi. Imperative Programming with Dependent Types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, pages 375–387, Santo Barbara, CA, June 2000.

[Xi02] Hongwei Xi. Dependent Types for Program Termination Verification. *Journal of Higher-Order and Symbolic Computation*, 15(1):91–132, March 2002.

[Xi04a] Hongwei Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

[Xi04b] Hongwei Xi. Dependent Types for Practical Programming via Applied Type System. Available at:
`http://www.cs.bu.edu/~hwxi/academic/drafts/ATSdml.ps`,
September 2004.

[Xi08] Hongwei Xi. The ATS Programming Language System, 2008. Available at:
`http://www.ats-lang.org/`.

[Xi09] Hongwei Xi. A Simple and General Theoretical Account for Abstract Types. In *Proceedings of Brazilian Symposium on Formal Methods*, pages 336–349, Gramado, Brazil, August 2009. Springer-Verlag LNCS 5902.

[XZ04] Hongwei Xi and Dengping Zhu. Views, Types and Viewtypes, October 2004. Available at:
`http://www.cs.bu.edu/~hwxi/ATS/PAPER/VsTsVTs.ps`.

[XZL05] Hongwei Xi, Dengping Zhu, and Yanka Li. Applied Type System with Stateful Views. Technical Report BUCS-2005-03, Boston University, 2005. Available at:
`http://www.cs.bu.edu/~hwxi/ATS/PAPER/ATSwSV.ps`.

[ZX05] Dengping Zhu and Hongwei Xi. Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, pages 83–97, Long Beach, CA, January 2005. Springer-Verlag LNCS, 3350.