Safe Programming with Pointers through Stateful Views*

Dengping Zhu and Hongwei Xi

Computer Science Department
Boston University

{zhudp,hwxi}@cs.bu.edu

Abstract. The need for direct memory manipulation through pointers is essential in many applications. However, it is also commonly understood that the use (or probably misuse) of pointers is often a rich source of program errors. Therefore, approaches that can effectively enforce safe use of pointers in programming are highly sought after. ATS is a programming language with a type system rooted in a recently developed framework *Applied Type System*, and a novel and desirable feature in ATS lies in its support for safe programming with pointers through a novel notion of *stateful views*. In particular, even pointer arithmetic is allowed in ATS and guaranteed to be safe by the type system of ATS. In this paper, we give an overview of this feature in ATS, presenting some interesting examples based on a prototype implementation of ATS to demonstrate the practicality of safe programming with pointer through stateful views.

1 Introduction

The verification of program correctness with respect to specification is a highly significant problem that is ever present in programming. There have been many approaches developed to address this fundamental problem (e.g., Floyd-Hoare logic [Hoa69,AO91], model checking [EGP99]), but they are often too expensive to be put into general software practice. For instance, 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. Though larger programs can be handled with the help of automated theorem proving, it is still as challenging as it was to support Floyd-Hoare logic in a realistic programming languages. On the other hand, the verification of type correctness of programs, that is, type-checking, in languages such as ML and Java scales convincingly in practice. However, we must note that the types in ML and Java are of relatively limited expressive power when compared to Floyd-Hoare logic. Therefore, we are naturally led to form type systems in which more sophisticated properties can be captured and then verified through type-checking.

A heavy-weighted approach is to adopt a type system in which highly sophisticated properties on programs can be captured. For instance, the type system of NuPrl [C+86] based on Martin-Löf's constructive type theory is such a case. In such a type system,

* Partially supported by NSF grant no. CCR-0229480

types are exceedingly expressive but type-checking often involves a great deal of theorem proving and becomes intractable to automate. This is essentially an approach that strongly favors expressiveness over scalability.

We adopt a light-weighted approach, introducing a notion of restricted form of dependent types, where we clearly separate type index expressions from run-time expressions. In functional programming, we have enriched the type system of ML with such a form of dependent types, leading to the design of a functional programming language DML (Dependent ML) [Xi98,XP99]. In imperative programming, we have designed a programming language Xanadu with C-like syntax to support such a form of dependent types. Along a different but closely related line of research, a new notion of types called guarded recursive (g.r.) datatypes is recently introduced [XCC03]. Noting the close resemblance between the restricted form of dependent types (developed in DML) and g.r. datatypes, we immediately initiated an effort to design a unified framework for both forms of types, leading to the formalization of *Applied Type System* ($\mathcal{ATS}$) [Xi03,Xi04]. We are currently in the process of designing and implementing ATS, a programming language with its type system rooted in $\mathcal{ATS}$. A prototype of ATS (with minimal documentation and many examples) is available on-line [Xi03]. Note that we currently use the name $\mathcal{ATS}$*-style dependent types* for the dependent types in $\mathcal{ATS}$ so as to distinguish them from the dependent types in Martin-Löf's constructive type theory.

```
fun arrayAssign {a:type, n:nat} (A:array (a,n), B:array (a,n)): unit =
  let
     fun loop {i:nat | i <= n} (ind: int (i)): unit =
       if ind < length A then
         (set (B, ind, get (A, ind)); loop (ind + 1))
  in
     loop (0)
  end
```

Fig. 1. A simple example in ATS

ATS is a comprehensive programming language designed to support a variety of programming paradigms (e.g., functional programming, object-oriented programming, imperative programming, modular programming, meta-programming), and the core of ATS is a call-by-value functional programming language. In this paper, we are to focus on the issue of programming with pointers in ATS.

As programming with dependent types is currently not a common practice, we use a concrete example to give the reader some feel as to how dependent types can be used to capture program invariants. In Figure 1, we implement a function *arrayAssign* that assigns the content of one array to another array. The header in the definition of the function *arrayAssign* means that *arrayAssign* is assigned the following type:

$$\forall a : \mathit{type}.\forall n : \mathit{nat}.(\mathbf{array}(a,n), \mathbf{array}(a,n)) \to \mathbf{1}$$

We use $\mathbf{1}$ for the unit type, which roughly corresponds to the void type in C. Given a type T and an integer I, we use $\mathbf{array}(T, I)$ as the type for arrays of size I in which each element is assigned the type T. Therefore, the type given to *arrayAssign* indicates that *arrayAssign* can only be applied to two arrays of the same size. The quantifications

$\forall a : type$ and $\forall n : nat$ mean that a and n can be instantiated with any given type and natural number, respectively. The inner function *loop* is assigned the following type: $\forall i : nat.i \leq n \supset (\mathbf{int}(i) \rightarrow \mathbf{1})$. Given an integer I, we use $\mathbf{int}(I)$ as the singleton type for I, that is, the only value of type $\mathbf{int}(I)$ equals I. The type given to *loop* means that *loop* can only be applied to a natural number whose value is less than or equal to n, which is the size of the arguments of *arrayAssign*. In ATS, we call $i \leq n$ a guard and $i \leq n \supset (\mathbf{int}(i) \rightarrow \mathbf{1})$ a guarded type. Also we point out that the function *length* is given the following type:

$$length \quad : \quad \forall a : type.\forall n : nat.\mathbf{array}(a, n) \rightarrow \mathbf{int}(n)$$

and the array subscripting function *get* and the array updating function *set* are given the following types:

$$\begin{array}{lcl} get & : & \forall a : type.\forall n : nat.\forall i : nat.i < n \supset ((\mathbf{array}(a, n), \mathbf{int}(i)) \rightarrow a) \\ set & : & \forall a : type.\forall n : nat.\forall i : nat.i < n \supset ((\mathbf{array}(a, n), \mathbf{int}(i), a) \rightarrow \mathbf{1}) \end{array}$$

which indicate that the index used to access an array must be within the bounds of the array.

To support safe programming with pointers, a notion called *stateful view* is introduced in ATS to model memory layout. Given a type T and an address L, we use $T@L$ for the (stateful) view indicating that a value of type T is stored at address L. This is the only form of a primitive view and all other views are built on top of such primitive views. For instance, we can form a view $(T@L, T'@(L + 1))$ to mean that a value of type T and another value of type T' are stored at addresses L and $L + 1$, respectively, where we use $L + 1$ for the address immediately following L. A stateful view is similar to a type, and it can be assigned to certain terms, which we often refer to as proof terms (or simply proofs) of stateful views. We treat proofs of views as a form of resources, which can be consumed as well as generated. In particular, the type theory on views is based on a form of linear logic [Gir87].

Certain functions may require proofs of stateful views when applied and they may cause stateful views to change when executed. For instance, the functions *getVar* and *setVar* are given the following types:

$$\begin{array}{lcl} getVar & : & \forall a : type.\forall l : addr.(a@l \mid \mathbf{ptr}(l)) \rightarrow (a@l \mid a) \\ setVar & : & \forall a_1 : type.\forall a_2 : type.\forall l : addr.(a_1@l \mid a_2, \mathbf{ptr}(l)) \rightarrow (a_2@l \mid \mathbf{1}) \end{array}$$

where we use $\mathbf{ptr}(L)$ as the singleton type for the pointer pointing to a given address L.

The type assigned to *getVar* means that the function takes a proof of view $T@L$ for some type T and address L, and a value of type $\mathbf{ptr}(L)$, and then returns a proof of view $T@L$ and a value of type T. In this case, we say that a proof of view $T@L$ is consumed and another proof of view $T@L$ is generated. We emphasize that proofs are only used at compile-time for performing type-checking and they are neither needed nor available at run-time. We use *getVar* here as the function that reads from a given pointer. Note that the proof argument of *getVar* essentially assures that the pointer passed to *getVar* cannot be a dangling pointer as the proof argument indicates that a value of certain type is stored at the address to which the pointer points.

```
fun swap {t1:type, t2:type, l1:addr, l2:addr}
   (pf1: t1 @ l1, pf2: t2 @ l2 | p1: ptr (l1), p2: ptr (l2))
  : '(t1 @ l2, t2 @ l1 | unit) =
  let
     val '(pf1 | tmp1) = getVar (pf1 | p1)
     val '(pf2 | tmp2) = getVar (pf2 | p2)
     val '( pf1' | _ ) = setVar (pf1 | p1, tmp2)
     val '( pf2' | _ ) = setVar (pf2 | p2, tmp1)
  in
    '(pf2', pf1' | '())
  end
```

Fig. 2. A simple swap function

The type assigned to the function *setVar* can be understood in a similar manner: *setVar* takes a proof of view $T_1@L$ for some type T_1 and address L and a value of type T_2 for some type T_2 and another value of type $\mathbf{ptr}(L)$, and then returns a proof of view $T_2@L$ and the unit (of type **1**). In this case, we say that a proof of view $T_1@L$ is consumed and another proof of view $T_2@L$ is generated. Since we use *setVar* here as the function that writes to a given address, this change precisely reflects the situations before and after the function *setVar* is called: A value of type T_1 is stored at L before the call and a value of type T_2 is stored at L after the call.

The functions *allocVar* and *freeVar*, which allocates and deallocates a memory unit, respectively, are also of interest, and their types are given as follows:

$$\begin{array}{rcl} \mathit{allocVar} & : & () \to \exists l : \mathit{addr}.(\mathbf{top}@l \mid \mathbf{ptr}(l)) \\ \mathit{freeVar} & : & \forall a : \mathit{type}.\forall l : \mathit{addr}.(a@l \mid \mathbf{ptr}(l)) \to \mathbf{1} \end{array}$$

We use **top** for the top type, that is, every type is a subtype of **top**. So when called, *allocVar* returns a proof of view $\mathbf{top}@L$ for some address L and a pointer of type $\mathbf{ptr}(L)$. The proof is needed if a write operation through the pointer is to be done. On the other hand, a call to *freeVar* makes a pointer no longer accessible.

As an example, a function is implemented in Figure 2 that swaps the contents stored at two (distinct) addresses. We use $'(\ldots)$ to form tuples, 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 (,)).

Note that proofs are manipulated explicitly in the above implementation, and this could be burdensome to a programmer. In ATS we also allow certain proofs be consumed and generated implicitly. For instance, the function in Figure 2 may also be implemented as follows in ATS:

```
fun swap {t1:type, t2:type, l1:addr, l2:addr}
   (pf1: t1 @ l1, pf2: t2 @ l2 | p1: ptr (l1), p2: ptr (l2))
  : '(t1 @ l2, t2 @ l1 | unit) =
  let val tmp := !p1 in p1 := !p2; p2 := tmp end
```

where we use ! for *getVar* and := for *setVar* and deal with proofs in an implicit manner.

The primary goal of the paper is to make ATS accessible to a wider audience who may or may not have adequate formal training in type theory. We are thus intentionally to avoid presenting the (intimidating) theoretical details on ATS as much as possible, striving for a clean and intuitive introduction to the use of stateful views in support of safe programming with pointers. For the reader who is interested in the technical development of ATS, please refer to [Xi03] for further details. Also, there is a prototype implementation of ATS as well as many interesting examples available on-line [Xi03].

We organize the rest of the paper as follows. In Section 2, we give brief explanation on some (uncommon) forms of types in ATS. We then present some examples in Section 3, showing how programming with pointers is made safe in ATS. We mention some related work in Section 4 and conclude in Section 5.

2 ATS/SV in a Nutshell

In this section, we present a brief overview of *ATS/SV*, the type system that supports imperative programming (with pointers) in ATS. As an applied type system, there are two components in *ATS/SV*: static component (statics) and dynamic component (dynamics). Intuitively, the statics and dynamics are each for handling types and programs, respectively, and we are to focus on the statics of *ATS/SV*.

sorts	$\sigma ::= \mathit{addr} \mid \mathit{bool} \mid \mathit{int} \mid \mathit{type}$
static contexts	$\Sigma ::= \emptyset \mid \Sigma, a : \sigma$
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.	$P ::= a \mid b \mid c_P(s_1, \ldots, s_n) \mid \neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \supset P_2$
types	$T ::= a \mid \delta(\vec{s}) \mid (\overline{V} \mid T) \rightarrow \mathit{CT} \mid P \supset T \mid \forall a : \sigma.T \mid P \wedge T \mid \exists a : \sigma.T$
computation types	$\mathit{CT} ::= \exists \Sigma, \overline{P}.(\overline{V} \mid T)$
stateful views	$V ::= \top \mid T@L \mid \overline{\delta}(\vec{s}) \mid V_1 \multimap V_2 \mid V_1 \otimes V_2$

Fig. 3. The syntax for the statics of *ATS/SV*

The syntax for the statics of *ATS/SV* is given in Figure 3. 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 *ATS/SV*: *addr*, *bool*, *int* and *type*; *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 (which are to be assigned to dynamic terms, i.e., programs). We use a for static variables, l for address constants $\mathbf{l}_0, \mathbf{l}_1, \ldots$, b for boolean values *tt* and *ff*, and i for integers $0, -1, 1, \ldots$. 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, P, I, T for static terms of sorts *addr*, *bool*, *int*, *type*, 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_P 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 $\vec{s}$ for a sequence of static terms, and $\overline{P}$, $\overline{T}$ and $\overline{V}$ for sequences of propositions, types and views, respectively, and $\emptyset$ for the empty sequence.

We use *ST* for a state, which is a finite mapping from addresses to values, and $\mathbf{dom}(\mathit{ST})$ for the domain of *ST*. We say that a value v is stored at l in *ST* if $\mathit{ST}(l) = v$. Note that we assume that every value takes one memory unit to store, and this, for instance, can be achieved through proper boxing. Given two states ST_1 and ST_2, we write $\mathit{ST}_1 \otimes \mathit{ST}_2$ for the union of ST_1 and ST_2 if $\mathbf{dom}(\mathit{ST}_1) \cap \mathbf{dom}(\mathit{ST}_2) = \emptyset$. We write $\mathit{ST} : V$ to mean that the state *ST* meets the view *V*. We now present some intuitive explanation on certain forms of views and types.

- We use $\top$ for the empty view, which is met by the empty state, that is, the state whose domain is empty.
- We use $\overline{\delta}$ for a view constructor and write $\vdash \overline{\delta}(\sigma_1, \ldots, \sigma_n)$ to mean that applying $\overline{\delta}$ to static terms $s_1, \ldots, s_n$ of sorts $\sigma_1, \ldots, \sigma_n$, respectively, generates a view $\overline{\delta}(s_1, \ldots, s_n)$. There are certain view proof constructors c associated with each $\overline{\delta}$, which are assigned views of the form $\forall \Sigma, \overline{P}.(\overline{V}) \multimap \overline{\delta}(\vec{s})$. For example, the (recursively defined) view constructor *arrayView* in Figure 6 (in Section 3) forms a view $\mathit{arrayView}(T, I, L)$ when applied to a type T, an integer I and an address L; the two proof constructors associated with *arrayView* are *ArrayNone* and *ArraySome*.
- Given L and T, we can form a primitive view $T@L$, which is met by the state that maps L to a value of type T.
- Given V_1 and V_2, a state *ST* meets $V_1 \multimap V_2$ if $\mathit{ST}_1 \otimes \mathit{ST}$ meets V_2 for any state $\mathit{ST}_1 : V_1$ such that $\mathbf{dom}(\mathit{ST}_1) \cap \mathbf{dom}(\mathit{ST}) = \emptyset$.
- Given V_1 and V_2, a state *ST* meets $V_1 \otimes V_2$ if $\mathit{ST} = \mathit{ST}_1 \otimes \mathit{ST}_2$ for some $\mathit{ST}_1 : V_1$ and $\mathit{ST}_2 : V_2$.
- In general, we use $\delta(\vec{s})$ for primitive types in *ATS/SV*. For instance, **top** is the top type, that is, every type is a subtype of **top**; **1** is the unit type; $\mathbf{ptr}(L)$ is a singleton type containing the only address equal to L, and we may also refer to a value of type $\mathbf{ptr}(L)$ as a pointer (pointing to L); $\mathbf{bool}(P)$ is a singleton type containing the only boolean value equal to P; $\mathbf{int}(I)$ is a singleton type containing the only integer equal to I.
- $(\overline{V} \mid T) \to \mathit{CT}$ is a type for (dynamic) functions that can be applied to values of type T only if the current state (when the application occurs) meets the views $\overline{V}$, and such an application yields a dynamic term that can be assigned the computation type *CT* of the form $\exists \Sigma', \overline{P}'.(\overline{V}' \mid T')$, which intuitively means that the dynamic term is expected to evaluate to value v at certain state *ST* such that for some static substitution Θ, each proposition in $\overline{P}'[\Theta]$ is true, v is of type $T'[\Theta]$ and *ST* meets $V'[\Theta]$. In the following presentation, we use $T_1 \to T_2$ as a shorthand for $(\emptyset \mid T_1) \to \exists \emptyset, \emptyset.(\emptyset \mid T_2)$ and call it a stateless function type.
- $P \supset T$ is called a guarded type and $P \wedge T$ is called an asserting type. As an example, the following type is for a function from natural numbers to negative integers:

  $$\forall a : \mathit{int}.a \geq 0 \supset (\mathbf{int}(a) \to \exists a' : \mathit{int}.(a' < 0) \wedge \mathbf{int}(a'))$$

  The guard $a \geq 0$ indicates that the function can only be applied to an integer that is greater than or equal to 0; the assertion $a' < 0$ means that each integer returned by the function is negative.

$$\frac{}{\Sigma; \overline{P} \models T \leq_{tp} \textbf{top}} \qquad \frac{}{\Sigma; \overline{P} \models T \leq_{tp} T} \qquad \frac{\Sigma; \overline{P} \models T_1 \leq_{tp} T_2 \quad \Sigma; \overline{P} \models T_2 \leq_{tp} T_3}{\Sigma; \overline{P} \models T_1 \leq_{tp} T_3}$$

$$\frac{\vdash \delta(\sigma_1, \ldots, \sigma_n) \quad \Sigma; \overline{P} \models s_i \equiv_{\sigma_i} s'_i \text{ for } 1 \leq i \leq n}{\Sigma; \overline{P} \models \delta(s_1, \ldots, s_n) \leq_{tp} \delta(s'_1, \ldots, s'_n)}$$

$$\frac{\Sigma; \overline{P}; \overline{V}' \models \otimes(\overline{V}) \quad \Sigma; \overline{P} \models T' \leq_{tp} T \quad \Sigma; \overline{P} \models CT \leq_{ct} CT'}{\Sigma; \overline{P} \models (\overline{V} \mid T) \to CT \leq_{tp} (\overline{V}' \mid T') \to CT'}$$

$$\frac{}{\Sigma; \overline{P} \vdash (\overline{V} \mid T) \to CT \leq_{tp} (\overline{V}, V \mid T) \to CT[V]} \textbf{(ext)}$$

$$\frac{\Sigma; \overline{P}, P' \models P \quad \Sigma; \overline{P}, P' \models T \leq_{tp} T'}{\Sigma; \overline{P} \models P \supset T \leq_{tp} P' \supset T'} \qquad \frac{\Sigma, a : \sigma; \overline{P} \models T \leq_{tp} T'}{\Sigma; \overline{P} \models \forall a : \sigma.T \leq_{tp} \forall a : \sigma.T'}$$

$$\frac{\Sigma; \overline{P}, P \models P' \quad \Sigma; \overline{P}, P \models T \leq_{tp} T'}{\Sigma; \overline{P} \models P \wedge T \leq_{tp} P' \wedge T'} \qquad \frac{\Sigma, a : \sigma; \overline{P} \models T \leq_{tp} T'}{\Sigma; \overline{P} \models \exists a : \sigma.T \leq_{tp} \exists a : \sigma.T'}$$

$$\frac{\Sigma, \Sigma_0; \overline{P}, \overline{P}_0 \models \overline{P}'_0 \quad \Sigma, \Sigma_0; \overline{P}, \overline{P}_0; \overline{V} \models \otimes(\overline{V}') \quad \Sigma, \Sigma_0; \overline{P}, \overline{P}_0 \models T \leq_{tp} T'}{\Sigma; \overline{P} \models \exists \Sigma_0, \overline{P}_0.(\overline{V} \mid T) \leq_{ct} \exists \Sigma_0, \overline{P}'_0.(\overline{V}' \mid T')}$$

Fig. 4. The subtype rules

There are two forms of constraints in *ATS/SV*: $\Sigma; \overline{P} \models P$ (persistent) and $\Sigma; \overline{P}; \overline{V} \models V$ (ephemeral), which are needed to define type equality. Generally speaking, we use intuitionistic logic and intuitionistic linear logic to reason about persistent and ephemeral constraints, respectively. We may write $\Sigma; \overline{P} \models \overline{P}_0$ to mean that $\Sigma; \overline{P} \models P$ holds for every P in $\overline{P}_0$. Most of the rules for proving persistent constraints are standard and thus omitted. For instance, the following rules are available:

$$\frac{}{\Sigma; \overline{P}, P \models P} \qquad \frac{\Sigma; \overline{P}, \neg P \models ff}{\Sigma; \overline{P} \models P} \qquad \frac{\Sigma; \overline{P}, P_1 \models P_2}{\Sigma; \overline{P} \models P_1 \supset P_2} \qquad \frac{\Sigma; \overline{P} \models P_1 \supset P_2 \quad \Sigma; \overline{P} \models P_1}{\Sigma; \overline{P} \models P_2}$$

We introduce a subtype relation $T_1 \leq_{tp} T_2$ on static terms of sort *type* and define the type equality $T_1 =_{type} T_2$ to be $T_1 \leq_{tp} T_2 \wedge T_2 \leq_{tp} T_1$. A subtype judgment is of the form $\Sigma; \overline{P} \models T_1 \leq_{tp} T_2$, and the rules for deriving such a judgment are given in Figure 4, where the obvious side conditions associated with certain rules are omitted. Note that $\otimes(\overline{V})$ is defined to be $\top$ if $\overline{V}$ is empty or $V_1 \otimes \ldots \otimes V_n$ if $\overline{V} = V_1, \ldots, V_n$ for some $n \geq 1$. In the rule **(ext)**, we write $CT[V]$ for $\exists \Sigma, \overline{P}.(\overline{V}, V \mid T)$, where CT is $\exists \Sigma, \overline{P}.(\overline{V} \mid T)$ and no free variables in V occur in Σ. For those who are familiar with *separation logic* [Rey02], we point out that this rule essentially corresponds to the frame rule there. The rule **(ext)** is essential: For instance, suppose the type of a function is $(\overline{V} \mid T) \to CT$ and the current state meets the view $\otimes(\overline{V}_0)$ such that $\overline{V}_0 = \overline{V}_1, V$ and $\emptyset; \emptyset; \overline{V}_1 \models \otimes(\overline{V})$ is derivable. In order to apply the function at the current state, we need to assign the type $(\overline{V}, V \mid T) \to CT[V]$ to the function so that the view V can be "carried over". This can be achieved by an application of the rule **(ext)**.

$$\frac{\Sigma; \overline{P} \models T \leq_{tp} T'}{\Sigma; \overline{P}; T@L \models T'@L} \quad \frac{}{\Sigma; \overline{P}; \emptyset \models \top} \quad \frac{\Sigma; \overline{P}; \overline{V} \models V}{\Sigma; \overline{P}; \overline{V}, \top \models V}$$

$$\frac{\Sigma; \overline{P}; \overline{V}_1 \models V_1 \quad \Sigma; \overline{P}; \overline{V}_2 \models V_2}{\Sigma; \overline{P}; \overline{V}_1, \overline{V}_2 \models V_1 \otimes V_2} \quad \frac{\Sigma; \overline{P}; \overline{V}, V_1, V_2 \models V}{\Sigma; \overline{P}; \overline{V}, V_1 \otimes V_2 \models V}$$

$$\frac{\Sigma; \overline{P}; \overline{V}, V_1 \models V_2}{\Sigma; \overline{P}; \overline{V} \models V_1 \multimap V_2} \quad \frac{\Sigma; \overline{P}; \overline{V}_1 \models V_1 \multimap V_2 \quad \Sigma; \overline{P}; \overline{V}_2 \models V_1}{\Sigma; \overline{P}; \overline{V}_1, \overline{V}_2 \models V_2}$$

$$\frac{\vdash \overline{\delta}(\sigma_1, \ldots, \sigma_n) \quad \Sigma; \overline{P} \models s_i \equiv_{\sigma_i} s'_i \text{ for } 1 \leq i \leq n}{\Sigma; \overline{P}; \overline{\delta}(s_1, \ldots, s_n) \models \overline{\delta}(s'_1, \ldots, s'_n)}$$

$$\frac{\Sigma; \overline{P}[a \mapsto i]; \overline{V}[a \mapsto i] \vdash V[a \mapsto i] \text{ for every integer } i}{\Sigma, a : \mathit{int}; \overline{P}; \overline{V} \vdash V}$$

Fig. 5. Some rules for ephemeral constraints

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

Fig. 6. An dataview for arrays

Some of the rules for proving ephemeral constraints are given in Figure 5, and the rest are associated with primitive view constructors. Given primitive view constructor $\overline{\delta}$ with proof constructors $c_1, \ldots, c_n$, we introduce the following rule for each c_i,

$$\frac{\Sigma \vdash \Theta : \Sigma_0 \quad \Sigma \models \overline{P}_0[\Theta] \quad \Sigma; \overline{P}; \overline{V} \models \otimes(\overline{V}_i[\Theta])}{\Sigma; \overline{P}; \overline{V} \models \overline{\delta}(\vec{s}_i[\Theta])}$$

where we assume that c_i is assigned the following view: $\forall \Sigma_i, \overline{P}_i.(\overline{V}_i) \multimap \overline{\delta}(\vec{s}_i)$; in addition, we introduce the following rule:

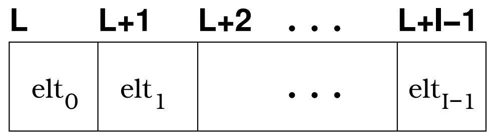
$$\frac{\Sigma, \Sigma_i; \overline{P}, \overline{P}_i, \vec{s} = \vec{s}_i; \overline{V}, \overline{V}_i \models V \text{ for } 1 \leq i \leq n}{\Sigma; \overline{P}; \overline{V}, \overline{\delta}(\vec{s}) \models V}$$

The key point we stress here is that both the persistent and ephemeral constraint relations can be formally defined.

3 Examples

3.1 Arrays

Array is probably the most commonly used data structure in programming. We declare in Figure 6 a dataview for representing arrays. Given a type T, an integer I and an address L, *arrayView*(T, I, L) is a view for an array pictured as follows,



such that (1) each element of the array is of type T, (2) the length of the array is I and (3) the array starts at address L and ends at address $L+I-1$.There are two view proof constructors *ArrayNone* and *ArraySome* associated with the view *arrayView*, which are assigned the following functional views:

$$\begin{array}{lcl} \mathit{ArrayNone} & : & \forall a:\mathit{type}.\forall l:\mathit{addr}.()\multimap \mathit{arrayView}(a,0,l) \\ \mathit{ArraySome} & : & \forall a:\mathit{type}.\forall l:\mathit{addr}.\forall n:nat.(a@l,\mathit{arrayView}(a,n,l+1))\multimap \mathit{arrayView}(a,n+1,l) \end{array}$$

For instance, the view assigned to *ArraySome* means that an array of size $I+1$ containing elements of type T is stored at address L if an value of type T is stored at L and an array of size I containing values of type T is stored at $L+1$.

```
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) = getVar (pf1 | p)
     // pf1': a@l
  in
     '(ArraySome (pf1', pf2) | x)
  end
```

Fig. 7. A simple function on arrays

We now implement a simple function *getFirst* in Figure 7 that takes the first element in a nonempty array. The header of the function *getFirst* indicates that the following type is assigned to it:

$$\forall a:\mathit{type}.\forall n:\mathit{int}.\forall l:\mathit{addr}.n>0\supset((\mathit{arrayView}(a,n,l)\mid \mathbf{ptr}(l))\rightarrow(\mathit{arrayView}(a,n,l)\mid a))$$

The (unfamiliar) syntax in the body of *getFirst* needs some explanation: *pf* is a proof of the view $\mathit{arrayView}(a,n,l)$, and it must be of the form $\mathit{ArraySome}(pf_1,pf_2)$, where pf_1 and pf_2 are proofs of views $a@l$ and $\mathit{arrayView}(a,n-1,l+1)$, respectively; recall that the function *getVar* is assumed to be of the following type:

$$\forall a:\mathit{type}.\forall l:\mathit{addr}.(a@l\mid \mathbf{ptr}(l))\rightarrow(a@l\mid a)$$

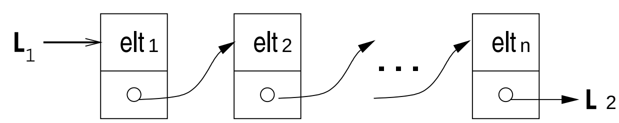
which simply means that applying *getVar* to a pointer of type $\mathbf{ptr}(L)$ requires a proof of $T@L$ for some type T and the application returns a value of type T as well as a proof of $T@L$; thus pf'_1 is also a proof of $a@l$ and $\mathit{ArraySome}(pf'_1,pf_2)$ is a proof of $\mathit{arrayView}(a,n,l)$. 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.

```
dataview slseg (type, int, addr, addr) =
  | {a:type, l:addr} SlsegNone (a, 0, l, l)
  | {a:type, n:nat, first, next, last | first <> null}
    SlsegSome (a, n+1, first, last) of
      ((a, ptr (next)) @ first, slseg (a, n, next, last))

viewdef sllist (a, n, l) = slseg (a, n, l, null)
```

Fig. 8. A dataview for singly-linked list segments

3.2 Singly-Linked Lists

We can declare a dataview for representing singly-linked list segments in Figure 8. Note that we write $(T_0, \cdots, T_n)@L$ for a sequence of views: $T_0@(L+0), \cdots, T_n@(L+n)$. Given a type T, an integer I and two addresses L_1 and L_2, $slseg(T, I, L_1, L_2)$ is a view for a singly-linked list segment pictured as follows:



where (1) each element in the segment is of type T, (2) the length of the segment is n and (3) the segment starts at L_1 and ends at L_2. A singly-linked list is simply a special kind of singly-linked list segment that ends with a null pointer, and this is clearly reflected in the definition of the view constructor *sllist* presented in Figure 8.

We now present an interesting example in Figure 9. The function *array2sllist* in the upper part of the figure turns an array into a singly-linked list. To facilitate understanding, we also present in the lower part of the figure a corresponding function implemented in C. If we erase the types and proofs in the implementation of *array2sllist* in ATS, then the implementation is tail recursive and tightly corresponds to the loop in the implementation in C. What is remarkable here is that the type system of ATS can guarantee the memory safety of *array2sllist* (even in the presence of pointer arithmetic).

3.3 A Buffer Implementation

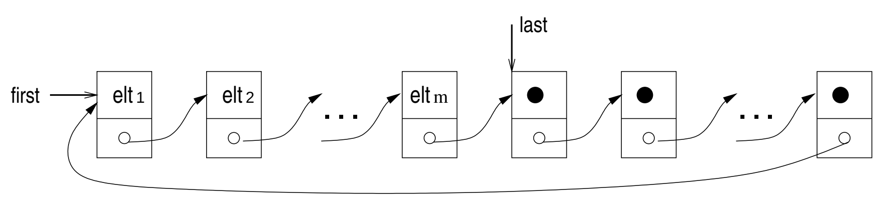
We present an implementation of buffers based on linked lists in this section. We first define a view constructor *bufferView* as follows:

```
viewdef bufferView (a:type, m:int, n:int, first: addr, last: addr) =
  '(slseg (a, m, first, last), slseg (top, n-m, last, first))
```

where m and n represent the number of elements stored in a buffer and the maximal buffer size, respectively. For instance, such a buffer can be pictured as follows:

```
fun array2sllist {l:addr, n:nat | n >= 1, l <> null}
  (pf: arrayView (top, n+n, l) | p: ptr(l), s: int(n))
  : '(sllist (top, n, l) | unit) =
  if s ieq 1 then
    let
       prval ArraySome (pf0, ArraySome (pf1, ArrayNone)) = pf
       val '(pf1 | _) = setVar (pf1 | p + 1, null)
    in
       '(SlsegSome ('(pf0, pf1), SlsegNone) | '())
    end
  else
    let
       prval ArraySome (pf0, ArraySome (pf1, pf)) = pf
       val '(pf1 | _) = setVar (pf1 | p + 1, p + 2)
       val '(rest | _) = array2sllist (pf | p + 2, s - 1)
    in
       '(SlsegSome ('(pf0, pf1), rest) | '())
    end

/////////////////////////////////////////////////////////////////

/* The following program in C corresponds the above one in ATS */

typedef struct slseg { int val; struct slseg * next; } slseg;

void array2sllist (int* p, int size) {
  int s;

  for (s = size; s > 1; s = s - 1) { *(p+1) = p+2; p = p+2; }

  *(p+1) = 0; /* assign the null pointer */
}
```

Fig. 9. Converting an array into a singly-linked list



where we use • for uninitialized or discarded content. In the above picture, we see that a buffer of maximal size n consists of two list segments: one with length m, which contains the values that are currently placed in the buffer, starts at address *first* and ends at *last*, and we call it the *occupied segment*; the other with length $(n - m)$, which contains all free cells in this buffer, starts at *last* and ends at *first*, and we call it *free segment*. The address *first* is often viewed as the head of a buffer.

In Figure 10, we present a function *addIn* that inserts an element into a buffer and another function *takeOut* that removes an element from a buffer. The header of the function *addIn* indicates that the following type is assigned to it,

$$\forall a : \mathit{type}.\forall m : \mathit{nat}.\forall n : \mathit{nat}.\forall l_1 : \mathit{addr}.\forall l_2 : \mathit{addr}.m < n \supset \\ (\mathit{bufferView}(a, m, n, l_1, l_2) \mid a, \mathbf{ptr}(l_2)) \to \exists l_3 : \mathit{addr}.(\mathit{bufferView}(a, m+1, n, l_1, l_3) \mid \mathbf{ptr}(l_3))$$

```
fun addIn {a:type, m: nat, n:nat, first:addr, last:addr | m < n}
    (pf: bufferView (a, m, n, first, last) | x: a, t: ptr(last))
  : [last':addr]
      '(bufferView (a, m+1, n, first, last') | ptr (last')) =
   let
      prval '(pf0, pf1) = pf
      prval SlsegSome ('(pf100, pf101), pf11) = pf1
      val '(pf100 | _) = setVar (pf100 | t, x)
      val '(pf101 | p) = getVar (pf101 | t + 1)
      prval pf0 =
        slsegAppend (pf0, SlsegSome ('(pf100, pf101), SlsegNone))
   in
     '( '(pf0, pf11) | p )
   end

fun takeOut {a:type, m:nat, n:nat, first:addr, last:addr | m>0, n>=m}
    (pf: bufferView (a, m,  n, first, last) | h: ptr(first))
  : [first':addr]
      '(bufferView (a, m-1, n, first', last) | '(a, ptr(first'))) =
   let
      prval '(pf0, pf1) = pf
      prval SlsegSome ('(pf000, pf001), pf01) = pf0
      val '(pf000 | x) = getVar (pf000 | h)
      val '(pf001 | p) = getVar (pf001 | h + 1)
      prval pf1 =
        slsegAppend (pf1, SlsegSome ('(pf000, pf001), SlsegNone))
   in
     '( '(pf01, pf1) | '(x, p) )
   end
```

Fig. 10. Two functions on cyclic buffers

which simply means that inserting into a buffer requires that the buffer is not full and, if it succeeds, the application increases the length of *occupied segment* by one and returns a new *ending* address for *occupied segment* (a.k.a. the new *starting* address for *free segment*). Similarly, the following type is assigned to the function *takeOut*,

$$\begin{aligned}&\forall a : \mathit{type}.\forall m : nat.\forall n : nat.\forall l_1 : \mathit{addr}.\forall l_2 : \mathit{addr}.m \leq n \wedge m > 0 \supset \\ &\quad (\mathit{bufferView}(a, m, n, l_1, l_2) \mid \mathbf{ptr}(l_1)) \to \exists l_3 : \mathit{addr}.(\mathit{bufferView}(a, m-1, n, l_3, l_2) \mid a, \mathbf{ptr}(l_3))\end{aligned}$$

which means that removing an element out of a buffer requires that the buffer is not empty and, if it succeeds, the application decreases the length of *occupied segment* by one and returns the element and a new *starting* address for *occupied segment* (a.k.a the new *ending* address for *free segment*). In addition, from the type of function *takeOut*, we can see that there is no need to fix the position of the buffer head and, in fact, the head of a buffer moves along the circular list if we keep taking elements out of that buffer.

The function *slsegAppend* is involved in the implementation of *addIn* and *takeOut*. This is a proof function that combines two list segment views into one list segment view, and it is assigned the following functional view:

$$\begin{aligned}&\forall a : \mathit{type}.\forall n_1 : nat.\forall n_2 : nat.\forall l_1 : \mathit{addr}.\forall l_2 : \mathit{addr}.\forall l_3 : \mathit{addr}. \\ &\quad (slseg(a, n_1, l_1, l_2), slseg(a, n_2, l_2, l_3)) \multimap slseg(a, n_1 + n_2, l_1, l_3)\end{aligned}$$

Note that this function is only used for type-checking at compile-time and is neither needed nor available at run-time.

3.4 Other Examples

In addition to arrays and singly-linked lists, we have also handled a variety of other data structures such as doubly-linked lists and doubly-linked binary trees that make (sophisticated) use of pointers. Some of the examples involving such data structures (e.g., a splay tree implementation based on doubly-linked binary trees) can be found on-line [Xi03].

4 Related Work

A fundamental problem in programming is to find approaches that can effectively facilitate the construction of safe and reliable software. In an attempt to address this problem, studies on program verification, that is, verifying whether a given program meets its specification, have been conducted extensively.

Some well-known existing approaches to program verification include model checking (which is the algorithmic exploration of the state spaces of finite state models of systems), program logics (e.g., Floyd-Hoare logic), type theory, etc. However, both model checking and Floyd-Hoare logic are often too expensive to be put into software practice. For instance, although model checking has been used with great success in hardware verification for more than twenty years, its application in software is much less common and the focus is often on verifying programs such as device drivers that are closely related to hardware control. In particular, model checking suffers from problems such as state space explosion and highly non-trivial abstraction and is thus difficult to scale in practice. There are also many cases reported in the literature that make successful use of program logics in program verification. As (a large amount of) theorem proving is often involved, such program verification is often too demanding for general practice.

On the other hand, the use of types in program error detection is ubiquitous. However, the types in programming languages such as ML and Java are often too limited for capturing interesting program invariants. Our work falls naturally in between full program verification, either in type theory or systems such as PVS, and traditional type systems for programming languages. When compared to verification, our system is less expressive but much more automatic. Our work can be viewed as providing a systematic and uniform language interface for a verifier intended to be used as a type system during the program development cycle. Our primary motivation is to all the programmer to express more program properties through types and thus catch more program errors at compile-time.

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 employed in designing Xanadu [Xi00] and DTAL [XH01] in attempts to reap similar benefits from dependent types in imperative programming. In hindsight, it can be readily noticed that the type systems of Xanadu and DTAL bear a close relation to Floyd-Hoare logic.

Along another line of research, a new notion of types called guarded recursive (g.r.) datatypes is recently introduced [XCC03]. Noting the close resemblance between the restricted form of dependent types (developed in DML) and g.r. datatypes, we immediately initiate an effort to design a unified framework for both forms of types, leading to the design and formalization of the framework *Applied Type System*. To support safe programming with pointers, the framework is further extended with stateful views [Xi03].

Also, the work in [OSSY02] is casually related to this paper as it shares the same goal of ruling out unsafe memory accesses. However, the underlying methodology adopted there is fundamentally different. In contrast to the static approach we take, it essentially relies on run-time checks to prevent dangling pointers from being accessed as well as to detect stray array subscripting.

There have been a great number of research activities 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 [MWH03], we also see an attempt to develop a general theory of type refinements for reasoning about program states.

5 Conclusion

Despite a great deal of research, it is still largely an elusive goal to verify the correctness of programs. Therefore, it is important to identify the properties that can be practically verified for realistic programs. We have shown with concrete examples the use of a restricted form of dependent types combined with stateful views in facilitating program verification in the presence of pointer arithmetic. A large number of automated program verification approaches often focus on verifying sophisticated properties of some particularly chosen programs. We feel that it is at least equally important to study scalable approaches to verifying elementary properties of programs in general programming as we have advocated in this paper.

In general, we are interested in promoting the use of light-weighted formal methods in practical programming, facilitating the construction of safe and reliable software. We have presented some examples in this paper in support of such a promotion, demonstrating a novel approach to safe programming with pointers.

References

[AO91] Krzysztof R. Apt and Olderog, E.-R. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, New York, 1991. ISBN 0-387-97532-2 (New York) 3-540-97532-2 (Berlin). xvi+441 pp.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986. ISBN 0-13-451832-2. x+299 pp.

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

[EGP99] E.M.Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[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.

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

[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.

[OSSY02] Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-safe ansi-c compiler: An approach to making c programs secure (progress report). In *International Symposium on Software Security*, volume 2609 of *Lecture Notes in Computer Science*. Springer-Verlag, November 2002.

[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)*, 2002. URL `citeseer.nj.nec.com/reynolds02separation.html`.

[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.

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

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. viii+181 pp. pp. viii+189. Available at `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.

[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.

[Xi03] Hongwei Xi. Applied Type System, July 2003. Available at: `http://www.cs.bu.edu/~hwxi/ATS`.

[Xi04] Hongwei Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

[XP99] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227. San Antonio, Texas, January 1999.