

# Applied Type System with Stateful Views\*

Hongwei Xi

Dengping Zhu

Yanka Li

Boston University

Boston University

Carnegie Mellon

hwxi@cs.bu.edu

zhudp@cs.bu.edu

yanka@cmu.edu

September 27, 2004

## Abstract

We present a type system that can effectively facilitate the use of types in capturing invariants in stateful programs that may involve (sophisticated) pointer manipulation. With its root in a recently developed framework *Applied Type System (ATS)*, the type system imposes a level of abstraction on program states by introducing a novel notion of recursive stateful views and then relies on a form of linear logic to reason about such views. We consider the design and then the formalization of the type system to constitute the primary contribution of the paper. In addition, we mention a prototype implementation of the type system and then give a variety of examples that attest to the practicality of programming with recursive stateful views.

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

---

\* Partially supported by NSF grants no. CCR-0224244 and 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 goes straight against the efforts to promote the use of safe languages such as ML and Java.

We have previously presented a framework *Applied Type System (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 (Xi 2004b), object-oriented programming (Xi, Chen, and Chen 2003; Chen, Shi, and Xi 2004), meta-programming (Chen and Xi 2003)) can be supported *within ATS* without resorting to *ad hoc* extensions. In this paper, we extend *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 splay trees (Sleator and Tarjan 1985).

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. We now present some examples to illustrate how several of such issues are dealt with in *ATS/SV*. Also, these examples can provide the reader with some concrete feel as to what can actually be accomplished in *ATS/SV*.

The first and foremost issue is the need for statically tracking changes made to states during program evaluation, where a state is basically a finite mapping from memory addresses to values. We are to address this issue by essentially following an idea in Typed Assembly Language (TAL) (Morisset, Walker, Crary, and Glew 1999). For example, a simple function written in ATS (Xi 2003), a programming language currently under development, is given in Figure 1 that swaps the contents stored at two (distinct) addresses. The syntax of ATS bears a great deal of resemblance to that of Standard ML (Milner, Tofte, Harper, and MacQueen 1997). The header of the function indicates that the following type is assigned to *swap*:

$$\forall \lambda. \forall \lambda'. \forall \tau. \forall \tau'. (\tau @ \lambda, \tau' @ \lambda' \mid \mathbf{ptr}(\lambda), \mathbf{ptr}(\lambda')) \rightarrow (\tau' @ \lambda, \tau @ \lambda' \mid \mathbf{1})$$

We use static variables  $\lambda$  and  $\tau$  to range over addresses and types, respectively, and  $\mathbf{ptr}(L)$  as a singleton type for the pointer pointing to the address  $L$ , and  $T @ L$  as a primitive stateful view to indicate that a value of type  $T$  is stored at  $L$ . The words *pointer* and *address* are used interchangeably

```

fun swap {l: addr, l': addr, t: type, t': type}
  (pf: t @ l, pf': t' @ l' | p: ptr(l), p': ptr(l'))
  // pf: a proof of t @ l and pf': a proof of t' @ l'
  : '(t' @ l, t @ l' | unit) =
  let tmp = !p in p := !p'; p' := tmp end

```

Figure 1: A simple swap function

```

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))

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

```

Figure 2: An example involving state views

in the following presentation, though we have a slight preference for the former if we are referring to a value at run-time. The type of *swap* means that (1) *swap* can be called on two pointers  $L$  and  $L'$  only if two values of some types  $T$  and  $T'$  are stored at addresses  $L$  and  $L'$ , respectively, and (2) two values of types  $T'$  and  $T$  are stored at addresses  $L$  and  $L'$ , respectively, when the call returns.

Clearly, we can only statically track the types of a fixed number of addresses in any given program. For instance, the types of exactly three addresses ( $p$ ,  $p'$  and  $tmp$ ) are tracked in the definition of *swap*. This is a severe limitation, making it difficult, if not completely impossible, to handle a mutable data structure such as linked lists that may involve an indefinite number of pointers. To address this issue, we are to introduce a notion of recursively defined stateful view (or view, for short) and then employ a form of linear logic to reason about this notion. As an example, we declare in Figure 2 a view constructor *arrayView*; when applied to a type  $T$ , an integer  $I$  and an address  $L$ , the constructor generates a view  $arrayView(T, I, L)$ , which essentially means that elements of type  $T$  are stored at addresses  $L + n$  for  $n = 0, \dots, I - 1$ , where  $I \geq 0$  is assumed. There are two view proof constructors *ArrayNone* and *ArraySome* associated with

*arrayView*, which are assigned the following two functional views, respectively:

$$\begin{aligned} \text{ArrayNone} & : \forall \lambda. \forall \tau. () \multimap \text{arrayView}(\tau, 0, \lambda) \\ \text{ArraySome} & : \forall \lambda. \forall \tau. \forall n : \text{nat}. (\tau @ \lambda, \text{arrayView}(\tau, n, \lambda + 1)) \multimap \text{arrayView}(\tau, n + 1, \lambda) \end{aligned}$$

For instance, the view assigned to *ArraySome* means that an array of size  $I + 1$  containing elements of type  $T$  is stored at  $L$  if an value of type  $T$  is stored at  $L$  and an array of size  $I$  containing values of type  $T$  is stored at  $L + 1$ . Note the involvement of pointer arithmetic:  $L + 1$  stands for the address immediately after  $L$ . In the following presentation, we are to follow the standard Curry-Howard isomorphism, treating views as a form of types for view proofs. Also, we point out that the type theory for views is based on intuitionistic linear logic.

The header of the function *getFirst* in Figure 2 indicates that the following type is assigned to it:

$$\forall \tau. \forall n : \text{int}. \forall \lambda. n > 0 \supset ((\text{arrayView}(\tau, n, \lambda) \mid \mathbf{ptr}(\lambda)) \rightarrow (\text{arrayView}(\tau, n, \lambda) \mid \tau))$$

where  $n > 0$  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. The (unfamiliar) syntax in the body of *getFirst* needs some explanation:  $pf$  is a proof of the view  $\text{arrayView}(a, n, l)$ , and it must be of the form  $\text{ArraySome}(pf_1, pf_2)$ , where  $pf_1$  and  $pf_2$  are proofs of views  $a @ l$  and  $\text{arrayView}(a, n - 1, l + 1)$ , respectively; the function *getVar* is assumed to be of the following type:

$$\forall \tau. \forall \lambda. (\tau @ \lambda \mid \mathbf{ptr}(\lambda)) \rightarrow (\tau @ \lambda \mid \tau)$$

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  $\tau @ \lambda$  and  $\text{ArraySome}(pf'_1, pf_2)$  is a proof of  $\text{arrayView}(a, n, l)$ . 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.

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

$$\forall \tau. \forall n : \text{int}. \forall i : \text{nat}. \forall \lambda. n > i \supset ((\text{arrayView}(\tau, n, \lambda) \mid \mathbf{ptr}(\lambda), \mathbf{int}(i)) \rightarrow (\text{arrayView}(\tau, n, \lambda) \mid \tau))$$

where  $\mathbf{int}(I)$  is a singleton type for the integer equal to  $I$ . This type simply means that *sub* 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  $T$ . The following pseudo code describes a naïve implementation of *sub*:

```
fun sub (p, offset) =
  if offset=0 then getFirst p else sub (p+1, offset-1)
```

```
prfun split {a:type, n:int, i:nat, l:addr | n >= i} <i>
  (pf: arrayView (a, n, l))
  : '(arrayView (a, i, l), arrayView (a, n - i, l + i)) =
  if i = 0 then '(ArrayNone, pf)
  else
    let
      prval ArraySome (pf1, pf2) = pf
      prval '(pf21, pf22) = split (pf2)
    in
      '(ArraySome (pf1, pf21), pf22)
    end

fun sub {a:type, n:int, i:nat, l:addr | n > i}
  (pf: arrayView (a,n,l) | p: ptr(l), offset: int(i))
  : '(arrayView (a, n, l) | a) =
  let
    prval '(pf1, pf2) = split (pf)
    // pf1: arrayView (a,i,l)
    // pf2: arrayView (a,n-i,l+i)
    val '(pf2 | x) = getFirst (pf2 | p + offset)
  in
    '(unsplit (pf1, pf2) | x)
  end
```

Figure 3: An implementation of array subscripting

where  $p$  is supposed to point to the array. While it can be assigned the above type for *sub* (after proper type annotation), this implementation yields an (unacceptably expensive)  $O(i)$ -time subscripting operation, where  $i$  is the offset value.

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 3, the *total* function *split* is assigned the following functional view:

$$\forall \tau. \forall n : \text{int}. \forall i : \text{nat}. \forall \lambda. i \leq n \supset \\ (\text{arrayView}(\tau, n, \lambda) \multimap (\text{arrayView}(\tau, i, \lambda), \text{arrayView}(\tau, n - i, \lambda + i)))$$

Note that *split* is recursively defined and the termination metric  $\langle i \rangle$  is used to verify that *split* is terminating. Please see (Xi 2002) for details on such a termination verification technique. To show that *split* is a total function, we also need to verify the following pattern matching in its body:

$$\text{prval ArraySome (pf1, pf2) = pf}$$

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

$$\forall \tau. \forall n : \text{int}. \forall i : \text{nat}. \forall \lambda. i \leq n \supset \\ ((\text{arrayView}(\tau, i, \lambda), \text{arrayView}(\tau, n - i, \lambda + i)) \multimap \text{arrayView}(\tau, n, \lambda))$$

With both *split* and *unsplit* to support view changes, an  $O(1)$ -time array subscripting function is implemented in Figure 3.

A major weakness in many typed programming languages lies in the treatment of the allocation and initialization of arrays. 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 *ATS/SV*, a function of the following type can be readily implemented that replaces elements of type  $T_1$  in an array with elements of type  $T_2$  when a function of type  $T_1 \rightarrow T_2$  is given:

$$\forall \tau_1. \forall \tau_2. \forall n : \text{nat}. \forall \lambda. (\text{arrayView}(\tau_1, n, \lambda) \mid \text{ptr}(\lambda), \tau_1 \rightarrow \tau_2) \rightarrow (\text{arrayView}(\tau_2, n, \lambda) \mid \text{ptr}(\lambda))$$

With such a function, the allocation and initialization of an array can clearly be separated. We can actually achieve much more than this in *ATS/SV*. For example, we have an example in which an array is first allocated and then turned into a linked list.

There is yet another issue that we must properly address in order to support practical programming with stateful views. This issue may at first seem rather technical and a bit subtle to understand, but it is crucial to the practicality of stateful views. The focus so far is on using views to track state changes made through pointers. Whenever reading from or writing to an address  $L$ , we are required to provide a proof of  $T@L$  for some type  $T$ . However, it is evident that we cannot practically track every single pointer even with recursive stateful views, and we are in need of a

means that can allow us to properly hide certain pointers when necessary. An illustrative example of this nature is the implementation of a reference as is supported in ML. Essentially, references can be regarded as a special form of pointers such that we have no obligation to provide proofs (of views) when reading or writing through them. In ATS, the type constructor *ref* for forming types for references can be defined as follows:

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

In formal notation, given a type  $T$ ,  $ref(T)$  is defined to be  $\exists \lambda. (! (T @ \lambda) | \mathbf{ptr}(\lambda))$ , where  $!(T @ \lambda)$  is a *persistent* stateful view. In general, a persistent view is always of the form  $!V$ , where  $V$  is a stateful view (which may also be referred to as an *ephemeral* stateful view from now on). It will become clear soon that reasoning on persistent views is based on a form of intuitionistic logic. However, we emphasize that it is in general wrong to assume that a persistent view  $!V$  implies the ephemeral view  $V$ . It is actually, more or less, the case that a ephemeral view  $V$  implies the persistent view  $!V$ . Given a type  $T$  and an address  $L$ , we say that a function of type  $(T @ L | \mathbf{ptr}(L)) \rightarrow (T @ L | T)$  treats the view  $T @ L$  as an invariant as the function consumes a proof of  $T @ L$  and then produces another proof of  $T @ L$ . We are to show later that such a function can also be used as a function of type  $!(T @ L) | \mathbf{ptr}(L) \rightarrow T$ . In informal terms, we may say that a persistent view may be changed but it is guaranteed to be changed back. The function *getVar*, which is given the type  $\forall \tau. \forall \lambda. (\tau @ \lambda | \mathbf{ptr}(\lambda)) \rightarrow (\tau @ \lambda | \tau)$ , can be used as a function of type  $\forall \tau. \forall \lambda. (! (\tau @ \lambda) | \mathbf{ptr}(\lambda)) \rightarrow \tau$  to read from a reference. Similarly, we can form a function of type  $\forall \tau. \forall \lambda. (! (\tau @ \lambda) | \mathbf{ptr}(\lambda), \tau) \rightarrow \mathbf{1}$  for writing to a reference. We regard the recognition and the formalization of the notion of persistent state views as a major contribution of the paper.

Of course, we may also introduce some primitives to directly support persistent references as is done in ML. Though simple, this approach to references is not only theoretically uninteresting but also practically inadequate as what we often encounter is a situation where we need to first track and then hide a pointer. In particular, this approach is inherently unable to properly address the issue of reference initialization.

The rest of the paper is organized as follows. In Section 2, we introduce *ATS/SV*, a type system rooted in *ATS* that supports a notion of stateful views. We formalize both the static and dynamic semantics of *ATS/SV* and establish its soundness. We then present in Section 3 some realistic examples that involve stateful views, demonstrating how programming with stateful views can be done in a practical manner. Lastly, we give a detailed account for some of related works and then conclude.

## 2 Formal Development

In this section, we formalize a type system *ATS/SV* that is essentially an applied type system (Xi 2004a; Xi 2003) extended with a notion of recursively defined stateful view. As in 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 in *ATS/SV*, respectively.

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

 Figure 4: The syntax for the statics of *ATS/SV*

## 2.1 Statics

The syntax for the statics of *ATS/SV* is given in Figure 4. 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). We use  $a$  for static variables,  $l$  for address constants  $\mathbf{l}_0, \mathbf{l}_1, \dots$ ,  $b$  for boolean values  $tt$  and  $ff$ , and  $i$  for integers  $0, -1, 1, \dots$ . 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  $\sigma$  under  $\Sigma$ . 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  $\sigma$  we can form a proposition  $s_1 =_\sigma s_2$  if  $s_1$  and  $s_2$  are static terms of sort  $\sigma$ ; we may omit the subscript  $\sigma$  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 (to be defined later), and  $\mathbf{dom}(ST)$  for the domain of  $ST$ . We say that a value  $v$  is stored at  $l$  in  $ST$  if  $ST(l) = v$ . Note that we assume that every value takes one memory unit to store. Given two states  $ST_1$  and  $ST_2$ , we write  $ST_1 \otimes ST_2$  for the union of  $ST_1$  and  $ST_2$  if  $\mathbf{dom}(ST_1) \cap \mathbf{dom}(ST_2) = \emptyset$ . We write  $ST \models V$  to mean that the state  $ST$  meets the ephemeral stateful view  $V$ , which is to be formally defined later. 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  $\vdash \overline{\delta}(\sigma_1, \dots, \sigma_n)$  to mean that applying  $\overline{\delta}$  to static

terms  $s_1, \dots, s_n$  of sorts  $\sigma_1, \dots, \sigma_n$ , respectively, generates a view  $\bar{\delta}(s_1, \dots, s_n)$ . There are certain view proof constructors  $c$  associated with each  $\bar{\delta}$ , which are assigned views of the form  $\forall \Sigma, \bar{P}. (\bar{V}) \multimap \bar{\delta}(\bar{s})$ . For example, the (recursively defined) view constructor *arrayView* in Figure 2 forms a view  $\text{arrayView}(T, I, L)$  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  $ST_1 \otimes ST$  meets  $V_2$  for any state  $ST_1 : V_1$  such that  $\mathbf{dom}(ST_1) \cap \mathbf{dom}(ST) = \emptyset$ .
- Given  $V_1$  and  $V_2$ , a state  $ST$  meets  $V_1 \otimes V_2$  if  $ST = ST_1 \otimes ST_2$  for some  $ST_1 : V_1$  and  $ST_2 : V_2$ .
- In general, we use  $\delta(\bar{s})$  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; **ptr**( $L$ ) is a singleton type containing the only address equal to  $L$ , and we may also refer to a value of type **ptr**( $L$ ) as a pointer (pointing to  $L$ ); **bool**( $P$ ) is a singleton type containing the only boolean value equal to  $P$ ; **int**( $I$ ) is a singleton type containing the only integer equal to  $I$ .
- $(\bar{V} \mid T) \rightarrow CT$  is a type for (dynamic) functions that can be applied to values of type  $T$  only if the current state  $ST$  (when the application occurs) meets the views  $\bar{V}$ , and such an application yields a dynamic term that can be assigned the computation type  $CT$  of the form  $\exists \Sigma', \bar{P}'. (\bar{V}' \mid T')$ , which intuitively means that the dynamic term is expected to evaluate to value  $v$  at certain state  $ST'$  such that for some static substitution  $\Theta$ , each proposition in  $\bar{P}'[\Theta]$  is true,  $v$  is of type  $T'[\Theta]$  and  $ST$  meets  $\bar{V}'[\Theta]$ . In the following presentation, we use  $T_1 \rightarrow T_2$  as a shorthand for  $(\emptyset \mid T_1) \rightarrow \exists \emptyset, \emptyset. (\emptyset \mid T_2)$  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:

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

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. As another example, the following type:

$$\forall a : \text{bool}. \mathbf{bool}(a) \rightarrow a \wedge \mathbf{1}$$

indicates that a boolean value must be true if a function of this type called on the boolean value returns. Hence, we can assign this type to a function that verifies at run-time whether a given assertion (i.e., an boolean expression) holds.

There are so far two forms of constraints in *ATS/SV*:  $\Sigma; \bar{P} \models P$  (proposition) and  $\Sigma; \bar{P}; \bar{V} \models V$  (ephemeral view). We may write  $\Sigma; \bar{P} \vdash \bar{P}_0$  to mean that  $\Sigma; \bar{P} \models P$  holds for every  $P$  in  $\bar{P}_0$ . Also, we may write  $\Sigma; \bar{P}; \bar{V} \models \bar{V}'$  to mean  $\Sigma; \bar{P}; \bar{V} \models \otimes(\bar{V}')$ , where  $\otimes(\bar{V}')$  is defined to be  $\top$  if  $\bar{V}'$  is empty or  $V'_1 \otimes \dots \otimes V'_n$  if  $\bar{V}' = V'_1, \dots, V'_n$  for some  $n \geq 1$ . We also have 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  $T_1 \leq_{tp} T_2 \wedge T_2 \leq_{tp} T_1$ .

$$\begin{array}{c}
 \frac{}{\Sigma; \bar{P} \models tt} \text{ (reg-true)} \\
 \frac{}{\Sigma; \bar{P} \models P} \text{ (reg-false)} \\
 \frac{\Sigma; \bar{P} \models P_0}{\Sigma, a : \sigma; \bar{P} \models P_0} \text{ (reg-var-thin)} \\
 \frac{\Sigma \vdash P : bool \quad \Sigma; \bar{P} \models P_0}{\Sigma; \bar{P}, P \models P_0} \text{ (reg-prop-thin)} \\
 \frac{\Sigma, a : \sigma; \bar{P} \models P \quad \Sigma \vdash s : \sigma}{\Sigma; \bar{P}[a \mapsto s] \models P[a \mapsto s]} \text{ (reg-subst)} \\
 \frac{\Sigma; \bar{P} \models P_0 \quad \Sigma; \bar{P}, P_0 \models P}{\Sigma; \bar{P} \models P} \text{ (reg-cut)}
 \end{array}$$

Figure 5: The regularity rules for the proposition constraint relation

We are not to specify a set of rules for deriving proposition constraints. Instead, we require that the proposition constraint relation satisfy the regularity rules listed in Figure 5.

Some of the rules for proving ephemeral constraints are given in Figure 6, and the rest are associated with primitive view constructors. Given primitive view constructor  $\bar{\delta}$  with proof constructors  $c_1, \dots, c_n$ , we introduce the following rule for each  $c_i$ ,

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

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

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

$$\begin{array}{c}
 \frac{\Sigma; \bar{P} \models T \leq_{tp} T'}{\Sigma; \bar{P}; T@L \models T'@L} \\
 \frac{\Sigma; \bar{P}; \bar{V}_1 \models V_1 \quad \cdots \quad \Sigma; \bar{P}; \bar{V}_n \models V_n}{\Sigma; \bar{P}; \bar{V}_1, \dots, \bar{V}_n \models V_1 \otimes \dots \otimes V_n} \\
 \frac{\Sigma; \bar{P}; \bar{V}_1 \models \bar{V}'_1 \quad \Sigma; \bar{P}; \bar{V}'_1, \bar{V}_2 \models V}{\Sigma; \bar{P}; \bar{V}_1, \bar{V}_2 \models V} \\
 \frac{\Sigma; \bar{P}; \bar{V}, V_1 \models V_2}{\Sigma; \bar{P}; \bar{V} \models V_1 \multimap V_2} \\
 \frac{\Sigma; \bar{P}; \bar{V}_1 \models V_1 \multimap V_2 \quad \Sigma; \bar{P}; \bar{V}_2 \models V_1}{\Sigma; \bar{P}; \bar{V}_1, \bar{V}_2 \models V_2} \\
 \frac{\vdash \bar{\delta}(\sigma_1, \dots, \sigma_n) \quad \Sigma; \bar{P} \models s_i \equiv_{\sigma_i} s'_i \text{ for } 1 \leq i \leq n}{\Sigma; \bar{P}; \bar{\delta}(s_1, \dots, s_n) \models \bar{\delta}(s'_1, \dots, s'_n)} \\
 \frac{\Sigma; \bar{P}[a \mapsto i]; \bar{V}[a \mapsto i] \vdash V[a \mapsto i] \text{ for every integer } i}{\Sigma, a : int; \bar{P}, \bar{V} \vdash V}
 \end{array}$$

Figure 6: Some rules for ephemeral view constraints

The rules for deriving subtype judgments of the form  $\Sigma; \bar{P} \models T_1 \leq_{tp} T_2$  or  $\Sigma; \bar{P} \models CT_1 \leq_{tp} CT_2$  are given in Figure 7, where the obvious side conditions associated with certain rules are omitted.

The key point we stress here is that the proposition constraint relation, the ephemeral view constraint relation and the subtype relation can be formally defined. Also, the definition for a state meeting an ephemeral view is now formalized according to the rules in Figure 8.

## 2.2 Dynamics

The dynamics of *ATS/SV* is a typed language and static terms  $T$  of sort *type* serve as types for terms in dynamics. The syntax for the dynamics is given in Figure 10. We use  $x$  for a **lam**-variable and  $f$  for a **fix**-variable, and  $xf$  for either a **lam**-variable or a **fix**-variable; a **lam**-variable is a value but a **fix**-variable is not. The markers  $\supset^+(\cdot)$ ,  $\supset^-(\cdot)$ ,  $\wedge(\cdot)$ ,  $\vee^+(\cdot)$ ,  $\vee^-(\cdot)$ ,  $\exists(\cdot)$  are mainly needed to facilitate inductive proofs on typing derivations, and this point is already made clear in the development of *ATS* (Xi 2004a; Xi 2003). Note that we may often omit writing these markers in examples so as to deliver a simple and clean presentation.

There may be some predefined dynamic constants  $dc$ , which are either dynamic constant constructors  $dcc$  or dynamic constant functions  $dcf$ . We write  $dc(d_1, \dots, d_n)$  for applying  $dc$  to  $n$  arguments  $d_1, \dots, d_n$ , and may write  $dcc$  for  $dcc()$ . Each dynamic constant constructor  $dcc$  is assigned a dynamic constant constructor type (or  $dcc$ -type, for short) of the following form:

$$\begin{array}{c}
 \overline{\Sigma; \overline{P} \models T \leq_{tp} \mathbf{top}} \\
 \overline{\Sigma; \overline{P} \models T \leq_{tp} T} \\
 \frac{\vdash \delta(\sigma_1, \dots, \sigma_n) \quad \Sigma; \overline{P} \models s_i \equiv_{\sigma_i} s'_i \text{ for } 1 \leq i \leq n}{\Sigma; \overline{P} \models \delta(s_1, \dots, s_n) \leq_{tp} \delta(s'_1, \dots, s'_n)} \\
 \frac{\Sigma; \overline{P} \models T' \leq_{tp} T \quad \Sigma; \overline{P}; \overline{V}' \models \overline{V} \quad \Sigma; \overline{P} \models CT \leq_{ctp} CT'}{\Sigma; \overline{P} \models (\overline{V} \mid T) \rightarrow CT \leq_{tp} (\overline{V}' \mid T') \rightarrow CT'} \\
 \frac{\Sigma; \overline{P}, P' \models P \quad \Sigma; \overline{P}, P' \models T \leq_{tp} T'}{\Sigma; \overline{P} \models P \supset T \leq_{tp} P' \supset T'} \\
 \frac{\Sigma, a : \sigma; \overline{P} \models T \leq_{tp} T'}{\Sigma; \overline{P} \models \forall a : \sigma. T \leq_{tp} \forall a : \sigma. T'} \\
 \frac{\Sigma; \overline{P}, P \models P' \quad \Sigma; \overline{P}, P \models T \leq_{tp} T'}{\Sigma; \overline{P} \models P \wedge T \leq_{tp} P' \wedge T'} \\
 \frac{\Sigma, a : \sigma; \overline{P} \models T \leq_{tp} T'}{\Sigma; \overline{P} \models \exists a : \sigma. T \leq_{tp} \exists a : \sigma. T'} \\
 \frac{\Sigma, \Sigma_0; \overline{P}, \overline{P}_0 \models \overline{P}'_0 \quad \Sigma, \Sigma_0; \overline{P}, \overline{P}_0; \overline{V} \models \overline{V}' \quad \Sigma, \Sigma_0; \overline{P}, \overline{P}_0 \models T \leq_{tp} T'}{\Sigma; \overline{P} \models \exists \Sigma_0, \overline{P}_0. (\overline{V} \mid T) \leq_{ctp} \exists \Sigma_0, \overline{P}'_0. (\overline{V}' \mid T')}
 \end{array}$$

Figure 7: The subtype rules

$$\begin{array}{c}
 \overline{\emptyset; \emptyset; \emptyset \vdash v : T} \quad \overline{[l \mapsto v] \models T @ l} \quad \frac{ST_1 \models V_1 \quad ST_2 \models V_2}{ST_1 \otimes ST_2 \models V_1 \otimes V_2} \\
 \frac{ST_1 \models V_1 \quad ST_1 \otimes ST \models V_2 \text{ for all } ST_1 \text{ disjoint from } ST}{ST \models V_1 \multimap V_2} \\
 \frac{\vdash c : \forall \Sigma, \overline{P}. (\overline{V}) \multimap \delta(\sigma_1, \dots, \sigma_n) \quad \emptyset \vdash \Theta : \Sigma \quad \emptyset; \emptyset \models \overline{P}[\Theta] \quad ST \models \otimes(\overline{V}[\Theta])}{ST \models \overline{\delta}(\overline{s}[\Theta])}
 \end{array}$$

Figure 8: The rules for states meeting ephemeral views

$$\begin{aligned}
 \mathit{alloc} & : \forall a : \mathit{nat}. (\emptyset \mid \mathbf{int}(a)) \Rightarrow \exists \lambda. \lambda \neq \mathbf{0} \wedge (\mathit{arrayView}(\lambda, \mathbf{top}, a) \mid \mathbf{ptr}(\lambda)) \\
 \mathit{dealloc} & : \forall \lambda. \forall \tau. \forall a : \mathit{nat}. (\mathit{arrayView}(\tau, a, \lambda) \mid \mathbf{ptr}(\lambda), \mathbf{int}(a)) \Rightarrow (\emptyset \mid \mathbf{1}) \\
 \mathit{getVar} & : \forall \lambda. \forall \tau. (\tau @ \lambda \mid \mathbf{ptr}(\lambda)) \Rightarrow (\tau @ \lambda \mid \tau) \\
 \mathit{setVar} & : \forall \lambda. \forall \tau. (\mathbf{top} @ \lambda \mid \mathbf{ptr}(\lambda), \tau) \Rightarrow (\tau @ \lambda \mid \mathbf{1})
 \end{aligned}$$

Figure 9: Some stateful functions and their types

$$\begin{aligned}
 \text{dynamic terms} \quad d & ::= x \mid f \mid \mathit{dc}(\vec{d}) \mid \mathbf{if}(d_1, d_2, d_3) \mid \\
 & \quad \mathbf{lam} \ x.d \mid \mathbf{app}(d_1, d_2) \mid \\
 & \quad \mathbf{fix} \ f.d \mid \mathbf{let}_c \ x = d_1 \ \mathbf{in} \ d_2 \mid \\
 & \quad \supset^+(v) \mid \supset^-(d) \mid \forall^+(v) \mid \forall^-(d) \mid \\
 & \quad \wedge(d) \mid \mathbf{let} \ \wedge(x) = d_1 \ \mathbf{in} \ d_2 \mid \\
 & \quad \exists(d) \mid \mathbf{let} \ \exists(x) = d_1 \ \mathbf{in} \ d_2 \\
 \text{values} \quad v & ::= x \mid \mathit{dcc}(\vec{v}) \mid \mathbf{lam} \ x.d \mid \supset^+(v) \mid \forall^+(v) \mid \wedge(v) \mid \exists(v) \\
 \text{dynamic var. ctx.} \quad \Delta & ::= \emptyset \mid \Delta, x : T
 \end{aligned}$$

 Figure 10: The syntax for the dynamics of *ATS/SV*

$$\forall a_1 : \sigma_1 \dots \forall a_k : \sigma_k. P_1 \supset (\dots (P_m \supset (T_1, \dots, T_n) \Rightarrow T) \dots)$$

where  $n$  indicates the arity of the dynamic constant  $\mathit{dcc}$ . We may write  $\forall \Sigma, \overline{P}. (\overline{T}) \Rightarrow T$  for such a type, where  $\Sigma = a_1 : \sigma_1, \dots, a_k : \sigma_k$ ,  $\overline{P} = P_1, \dots, P_m$  and  $\overline{T} = T_1, \dots, T_n$ . For instance, we assume that the unit constant  $\langle \rangle$  is assigned the dcc-type  $() \Rightarrow \mathbf{1}$ , each address constant  $l$  is assigned the dcc-type  $() \Rightarrow \mathbf{ptr}(l)$ , each boolean constant  $b$  is assigned the dcc-type  $() \Rightarrow \mathbf{bool}(b)$ , and each integer constant  $i$  is assigned the dcc-type  $() \Rightarrow \mathbf{int}(i)$ . Similarly, each dynamic constant function is assigned a dynamic constant function type (or dcf-type, for short) of the following form:

$$\forall a_1 : \sigma_1 \dots \forall a_k : \sigma_k. P_1 \supset (\dots (P_m \supset (\overline{V} \mid \overline{T}) \Rightarrow CT) \dots)$$

We may write  $\forall \Sigma, \overline{P}. (\overline{V} \mid \overline{T}) \Rightarrow CT$  for such a type, where  $\Sigma = a_1 : \sigma_1, \dots, a_k : \sigma_k$  and  $\overline{P} = P_1, \dots, P_m$ . For instance, the division function  $/$  on integers is assigned the following dcf-type:

$$\forall a_1 : \mathit{int}. \forall a_2 : \mathit{int}. a_2 \neq 0 \supset (\emptyset \mid \mathbf{int}(a_1), \mathbf{int}(a_2)) \Rightarrow (\emptyset \mid \mathbf{int}(a_1/a_2))$$

In Figure 9, we list some predefined stateful functions and their dcf-types. The functions  $\mathit{alloc}$  and  $\mathit{dealloc}$  allocate and deallocate  $n$  memory units for each natural number  $n$ , respectively, and the

functions *getVar* and *setVar* reads from and writes to an address, respectively. Note that we assume that the pointer returned by *alloc* is not 0 (the null pointer) and the memory unit allocated by a call to *alloc* are assumed to be uninitialized as they are assigned the type **top**. Also note that a proof of view  $T@L$  is required in order to read from or write to a pointer of the type  $\mathbf{ptr}(L)$ , preventing dangling pointers, which may be potentially generated, from being ever accessed. The following two functions can be readily implemented through the use of *alloc* and *dealloc*:

$$\begin{aligned} \mathit{genvar} & : \forall\tau.(\emptyset \mid \mathbf{1}) \rightarrow \exists\lambda.(\mathbf{top}@ \lambda \mid \mathbf{1}) \\ \mathit{delvar} & : \forall\lambda.\forall\tau.(\mathbf{top}@ \lambda \mid \mathbf{ptr}(\lambda)) \rightarrow (\emptyset \mid \mathbf{1}) \end{aligned}$$

In order to assign a call-by-value dynamic semantics to dynamic terms, we make use of evaluation contexts, which are defined as follows:

$$\begin{aligned} \text{evaluation contexts } E ::= & \ [] \mid dc(v_1, \dots, v_{i-1}, E, d_{i+1}, \dots, d_n) \mid \mathbf{if}(E, d_1, d_2) \mid \\ & \mathbf{app}(E, d) \mid \mathbf{app}(v, E) \mid \supset^-(E) \mid \forall^-(E) \mid \mathbf{let}_c x = E \mathbf{in} d \mid \\ & \wedge(E) \mid \mathbf{let} \wedge(x) = E \mathbf{in} d \mid \exists(E) \mid \mathbf{let} \exists(x) = E \mathbf{in} d \end{aligned}$$

We define redexes and their reductions as follows.

- $\mathbf{app}(\mathbf{lam} x.d, v)$  is a redex, and  $d[x \mapsto v]$  is its reduction.
- $\mathbf{fix} f.d$  is a redex, and  $d[f \mapsto \mathbf{fix} f.d]$  is its reduction.
- $\mathbf{if}(true, d_1, d_2)$  is a redex, and its reduction is  $d_1$ .
- $\mathbf{if}(false, d_1, d_2)$  is a redex, and its reduction is  $d_2$ .
- $\supset^-(\supset^+(v))$  is a redex, and  $v$  is its reduction.
- $\forall^-(\forall^+(v))$  is a redex, and  $v$  its reduction.
- $\mathbf{let} \wedge(x) = \wedge(v) \mathbf{in} d$  is a redex, and  $d[x \mapsto v]$  is its reduction.
- $\mathbf{let} \exists(x) = \exists(v) \mathbf{in} d$  is a redex, and  $d[x \mapsto v]$  is its reduction.
- $\mathbf{let}_c x = v \mathbf{in} d$  is a redex, and its reduction is  $d[x \mapsto v]$ .
- Given a stateless dynamic constant function  $dcf$ ,  $dcf(v_1, \dots, v_n)$  is a redex if it is defined to equal some value  $v$ , and  $v$  is its reduction.

Given two dynamic terms  $d_1$  and  $d_2$  such that  $d_1 = E[d]$  and  $d_2 = E[d']$  for some redex  $d$  and its reduction  $d'$ , we write  $d_1 \rightarrow_{ev} d_2$  and say that  $d_1$  reduces to  $d_2$  in one step. We use  $ST$  for states, which are finite mappings from addresses to values defined as follows:

$$\text{states } ST ::= \ [] \mid ST[l \mapsto v]$$

We use  $[]$  for the empty mapping, and  $ST[l \mapsto v]$  for the mapping that extends  $ST$  with a link from  $l$  to  $v$ , where we assume that  $l$  is not in the domain  $\mathbf{dom}(ST)$  of  $ST$ . We may also write  $[l_1 \mapsto v_1, \dots, l_n \mapsto v_n]$  for a state  $ST$  such that  $\mathbf{dom}(ST) = \{l_1, \dots, l_n\}$  and  $ST(l_i) = v_i$  for  $1 \leq i \leq n$ , where  $l_1, \dots, l_n$  are assumed to be distinct addresses.

We define the reduction relation  $(ST_1, d_1) \rightarrow_{ev/st} (ST_2, d_2)$  as follows:

- $d_1 \rightarrow_{ev} d_2$  and  $ST_2 = ST_1$ , or
- $d_1 = E[alloc(i)]$  for some  $i \geq 0$ , and  $ST_2 = ST_1[l + 0 \mapsto \langle \rangle] \dots [l + (i - 1) \mapsto \langle \rangle]$ , or
- $d_1 = E[dealloc(i)]$  for some  $i \geq 0$ , and  $ST_1 = ST_2[l + 0 \mapsto v_1] \dots [l + (i - 1) \mapsto v_i]$ , or
- $d_1 = E[getVar(l)]$  for some  $l \in \mathbf{dom}(ST_1)$  and  $ST_2 = ST_1$  and  $d_2 = E[ST_1(l)]$ , or
- $d_1 = E[setVar(l, v)]$  for some  $l \in \mathbf{dom}(ST_1)$  and  $ST_2 = ST_1[l := v]$  and  $d_2 = E[\langle \rangle]$ , where  $ST_1[l := v]$  is the mapping that maps  $l$  to  $v$  and coincides with  $ST_1$  everywhere else.

$$\begin{array}{c}
 \frac{\Sigma; \bar{P}; \Delta \vdash d : T \quad \Sigma; \bar{P} \models T \leq_{tp} T'}{\Sigma; \bar{P}; \Delta \vdash d : T'} \text{ (ty-sub)} \\
 \frac{\Sigma; \bar{P}; \bar{V}; \Delta \vdash d : CT \quad \Sigma; \bar{P} \models CT \leq_{ctp} CT'}{\Sigma; \bar{P}; \bar{V}; \Delta \vdash d : CT'} \text{ (ty-sub}_c\text{)} \\
 \frac{\Sigma; \bar{P}; \Delta \vdash d : T}{\Sigma; \bar{P}; \bar{V}; \Delta \vdash d : (\bar{V} \mid T)} \text{ (ty-state)} \\
 \frac{\Delta(xf) = T}{\Sigma; \bar{P}; \Delta \vdash xf : T} \text{ (ty-var)} \\
 \frac{\vdash dcc : \forall \Sigma_0, \bar{P}_0. (T_1, \dots, T_n) \Rightarrow T \quad \Sigma \vdash \Theta : \Sigma_0 \quad \Sigma; \bar{P} \models \bar{P}_0[\Theta] \quad \Sigma; \bar{P} \vdash d_i : T_i[\Theta] \text{ for } 1 \leq i \leq n}{\Sigma; \bar{P}; \Delta \vdash dcc(d_1, \dots, d_n) : T[\Theta]} \text{ (ty-dcc)} \\
 \frac{\vdash dcf : \forall \Sigma_0, \bar{P}_0. (\bar{V}_0 \mid T_1, \dots, T_n) \Rightarrow CT \quad \Sigma \vdash \Theta : \Sigma_0 \quad \Sigma; \bar{P} \models \bar{P}_0[\Theta] \quad \Sigma; \bar{P}; \bar{V} \models \bar{V}_0[\Theta] \quad \Sigma; \bar{P} \vdash d_i : T_i[\Theta] \text{ for } 1 \leq i \leq n}{\Sigma; \bar{P}; \bar{V}; \Delta \vdash dcf(d_1, \dots, d_n) : CT[\Theta]} \text{ (ty-dcf)} \\
 \frac{\Sigma; \bar{P}; \Delta \vdash d_1 : \mathbf{bool}(P) \quad \Sigma; \bar{P}, P; \bar{V}; \Delta \vdash d_2 : CT \quad \Sigma; \bar{P}, \neg P; \bar{V}; \Delta \vdash d_3 : CT}{\Sigma; \bar{P}; \bar{V}; \Delta \vdash \mathbf{if}(d_1, d_2, d_3) : CT} \text{ (ty-if)} \\
 \frac{\Sigma; \bar{P}; \bar{V}; \Delta, x : T \vdash d : CT}{\Sigma; \bar{P}; \Delta \vdash \mathbf{lam} x.d : (\bar{V} \mid T) \rightarrow CT} \text{ (ty-lam)} \\
 \frac{\Sigma; \bar{P}; \Delta, f : T \vdash d : T}{\Sigma; \bar{P}; \Delta \vdash \mathbf{fix} f.d : T} \text{ (ty-fix)} \\
 \frac{\Sigma; \bar{P}; \Delta \vdash d_1 : (\bar{V}' \mid T) \rightarrow CT \quad \Sigma; \bar{P}; \bar{V} \models \bar{V}' \quad \Sigma; \bar{P}; \Delta \vdash d_2 : T}{\Sigma; \bar{P}; \bar{V}; \Delta \vdash d_1(d_2) : CT} \text{ (ty-app)}
 \end{array}$$

Figure 11: The typing rules for dynamics (1)

$$\begin{array}{c}
 \frac{\Sigma; \bar{P}, P; \Delta \vdash v : T}{\Sigma; \bar{P}; \Delta \vdash \supset^+(v) : P \supset T} \text{ (ty-}\supset\text{-intro)} \\
 \frac{\Sigma; \bar{P}; \Delta \vdash d : P \supset T \quad \Sigma; \bar{P} \models P}{\Sigma; \bar{P}; \Delta \vdash \supset^-(d) : T} \text{ (ty-}\supset\text{-elim)} \\
 \frac{\Sigma, a : \sigma; \bar{P}; \Delta \vdash v : T}{\Sigma; \bar{P}; \Delta \vdash \forall^+(v) : \forall a : \sigma. T} \text{ (ty-}\forall\text{-intro)} \\
 \frac{\Sigma; \bar{P}; \Delta \vdash d : \forall a : \sigma. T \quad \Sigma \vdash s : \sigma}{\Sigma; \bar{P}; \Delta \vdash \forall^-(d) : T[a \mapsto s]} \text{ (ty-}\forall\text{-elim)} \\
 \frac{\Sigma; \bar{P} \models P \quad \Sigma; \bar{P}; \Delta \vdash d : T}{\Sigma; \bar{P}; \Delta \vdash \wedge(d) : P \wedge T} \text{ (ty-}\wedge\text{-intro)} \\
 \frac{\Sigma; \bar{P}; \Delta \vdash d_1 : P \wedge T \quad \Sigma; \bar{P}, P; \bar{V}; \Delta, x : T \vdash d_2 : CT}{\Sigma; \bar{P}; \bar{V}; \Delta \vdash \mathbf{let} \wedge(x) = d_1 \mathbf{in} d_2 : CT} \text{ (ty-}\wedge\text{-elim)} \\
 \frac{\Sigma \vdash s : \sigma \quad \Sigma; \bar{P}; \Delta \vdash d : T[a \mapsto s]}{\Sigma; \bar{P}; \Delta \vdash \exists(d) : \exists a : \sigma. T} \text{ (ty-}\exists\text{-intro)} \\
 \frac{\Sigma; \bar{P}; \Delta \vdash d_1 : \exists a : \sigma. T \quad \Sigma, a : \sigma; \bar{P}; \bar{V}; \Delta, x : T \vdash d_2 : CT}{\Sigma; \bar{P}; \bar{V}; \Delta \vdash \mathbf{let} \exists(x) = d_1 \mathbf{in} d_2 : CT} \text{ (ty-}\exists\text{-elim)} \\
 \frac{\Sigma \vdash \Theta : \Sigma' \quad \Sigma; \bar{P} \models \bar{P}'[\Theta] \quad \Sigma; \bar{P}; \bar{V}; \Delta \vdash d : (\bar{V}'[\Theta] \mid T[\Theta])}{\Sigma; \bar{P}; \bar{V}; \Delta \vdash d : \exists \Sigma', \bar{P}'.(\bar{V}' \mid T)} \text{ (ty-}\exists_c\text{-intro)} \\
 \frac{\Sigma; \bar{P}; \bar{V}_1; \Delta \vdash d_1 : \exists \Sigma', \bar{P}'.(\bar{V}'_1 \mid T) \quad \Sigma, \Sigma'; \bar{P}, \bar{P}'; \bar{V}'_1, \bar{V}_2; \Delta, x : T \vdash d_2 : CT}{\Sigma; \bar{P}; \bar{V}_1, \bar{V}_2; \Delta \vdash \mathbf{let}_c x = d_1 \mathbf{in} d_2 : CT} \text{ (ty-}\exists_c\text{-elim)}
 \end{array}$$

Figure 12: The typing rules for dynamics (2)

The typing rules for the dynamics are given in Figure 11 and Figure 12, where there are two forms of typing judgments:  $\Sigma; \overline{P}; \Delta \vdash d : T$  and  $\Sigma; \overline{P}; \overline{V}; \Delta \vdash d : CT$ . Generally speaking, if  $\emptyset; \emptyset; \emptyset \vdash d : T$  is derivable, then the evaluation of  $d$  can start at any state; if  $\emptyset; \emptyset; \overline{V}; \emptyset \vdash d : CT$  is derivable, then the evaluation of  $d$  can start at any state  $ST$  that meets  $\overline{V}$ , and if the evaluation terminates, then a value of type  $T[\Theta]$  is returned and a state  $ST'$  is reached that meets  $\overline{V}'[\Theta]$ , where  $CT = \exists \Sigma, \overline{P}. (\overline{V}' \mid T)$  and  $\Theta : \Sigma$  is a substitution that makes each proposition  $P$  in  $\overline{P}[\Theta]$  hold.

For a technical reason, we are to replace the rule **(ty-var)** with the following rule

$$\frac{\Delta(xf) = T \quad \Sigma; \overline{P} \models T \leq_{tp} T'}{\Sigma; \overline{P}; \Delta \vdash xf : T'} \text{ (ty-var')}$$

and this replacement is needed when we prove Lemma 2.2.

**Lemma 2.1 (Substitution)** *We have the following.*

1. Assume that  $\Sigma \vdash s : \sigma$  is derivable.

- (a) If  $\Sigma, a : \sigma; \overline{P}; V \models V'$  is derivable, then  $\Sigma; \overline{P}[a \mapsto s]; V[a \mapsto s] \models V'[a \mapsto s]$  is also derivable.
- (b) If  $\Sigma, a : \sigma; \overline{P} \models T \leq_{tp} T'$  is derivable, then  $\Sigma; \overline{P}[a \mapsto s] \models T[a \mapsto s] \leq_{tp} T'[a \mapsto s]$  is also derivable.
- (c) If  $\Sigma, a : \sigma; \overline{P} \models CT \leq_{tp} CT'$  is derivable, then  $\Sigma; \overline{P}[a \mapsto s] \models CT[a \mapsto s] \leq_{ctp} CT'[a \mapsto s]$  is also derivable.
- (d) If  $\Sigma, a : \sigma; \overline{P}; \Delta \vdash d : T$  is derivable, then  $\Sigma; \overline{P}[a \mapsto s]; \Delta \vdash d : T[a \mapsto s]$  is also derivable.
- (e) If  $\Sigma, a : \sigma; \overline{P}; \overline{V}; \Delta \vdash d : CT$  is derivable, then  $\Sigma; \overline{P}[a \mapsto s]; \overline{V}[a \mapsto s]; \Delta \vdash d : CT[a \mapsto s]$  is also derivable.

2. Assume that  $\Sigma; \overline{P} \models P$  holds.

- (a) If  $\Sigma; \overline{P}; P; \Delta \vdash d : T$  is derivable, then  $\Sigma; \overline{P}; \Delta \vdash d : T$  is also derivable.
- (b) If  $\Sigma; \overline{P}; P; \overline{V}; \Delta \vdash d : CT$  is derivable, then  $\Sigma; \overline{P}; \overline{V}; \Delta \vdash d : CT$  is also derivable.

3. Assume that  $\Sigma; \overline{P}; \overline{V}_1; \Delta \models \overline{V}'_1$  holds.

- (a) If  $\Sigma; \overline{P}; \overline{V}'_1; \overline{V}_2; \Delta \vdash d : CT$  is derivable, then  $\Sigma; \overline{P}; \overline{V}_1; \overline{V}_2; \Delta \vdash d : CT$  is also derivable.

4. Assume that  $\Sigma; \overline{P}; \Delta \vdash v : T_1$  is derivable.

- (a) If  $\Sigma; \overline{P}; \Delta, x : T_1 \vdash d : T_2$  is derivable, then  $\Sigma; \overline{P}; \Delta \vdash d[x \mapsto v] : T_2$  is also derivable.
- (b) If  $\Sigma; \overline{P}; \overline{V}; \Delta, x : T_1 \vdash d : CT_2$  is derivable, then  $\Sigma; \overline{P}; \overline{V}; \Delta \vdash d[x \mapsto v] : CT_2$  is also derivable.

**Proof** Standard. The requirement for the regularity rules in Figure 5 is precisely for establishing this lemma. ■

**Lemma 2.2** *We have the following.*

1. Assume  $\mathcal{D} :: \Sigma; \overline{P}; \Delta, x : T_1 \vdash d : T$  and  $\Sigma; \overline{P} \models T'_1 \leq_{tp} T_1$ . Then there is a derivation  $\mathcal{D}' :: \Sigma; \overline{P}; \Delta, x : T'_1 \vdash d : T$  such that  $\mathbf{h}(\mathcal{D}') = \mathbf{h}(\mathcal{D})$ .
2. Assume  $\mathcal{D} :: \Sigma; \overline{P}; \overline{V}; \Delta, x : T \vdash d : CT$  and  $\Sigma; \overline{P} \models T'_1 \leq_{tp} T_1$ . Then there is a derivation  $\mathcal{D}' :: \Sigma; \overline{P}; \overline{V}; \Delta, x : T'_1 \vdash d : CT$  such that  $\mathbf{h}(\mathcal{D}') = \mathbf{h}(\mathcal{D})$ .

**Proof** By structural induction. ■

**Lemma 2.3 (Inversion)** *Assume  $\mathcal{D} :: \Sigma; \overline{P}; \Delta \vdash d : T$ . If  $d$  is of one of the forms  $\supset^+(d_1)$ ,  $\forall^+d_1$ ,  $\wedge(d_1)$ ,  $\exists(d_1)$  and **lam**  $x.d_1$ , then there is a derivation  $\mathcal{D}' :: \Sigma; \overline{P}; \Delta \vdash d : T$  such that  $\mathbf{h}(\mathcal{D}') \leq \mathbf{h}(\mathcal{D})$  and the last rule applied in  $\mathcal{D}'$  is not (**ty-sub**).*

**Proof** By an inspection of the typing rules in Figure 11 and Figure 12. ■

**Lemma 2.4 (Canonical Forms)** *Assume  $\mathcal{D} :: \emptyset; \emptyset; \emptyset \vdash v : T$ . Then we have the following.*

1. If  $T = \delta(\vec{s})$ , then  $v$  is of the form  $dcc(\vec{v})$  for some dynamic constructor associated with  $\delta$ .
2. If  $T = (\overline{V} \mid T_0) \rightarrow CT$ , then  $v$  is of the form **lam**  $x.d_0$ .
3. If  $T = P \supset T$ , then  $v$  is of the form  $\supset^+(d_0)$ .
4. If  $T = \forall a : \sigma.T$ , then  $v$  is of the form  $\forall^+d_0$ .
5. If  $T = P \wedge T$ , then  $v$  is of the form  $\wedge(d_0)$ .
6. If  $T = \exists a : \sigma.T$ , then  $v$  is of the form  $\exists(d_0)$ .

**Theorem 2.5 (Subject Reduction)** *We have the following.*

1. Assume that  $\emptyset; \emptyset; \emptyset \vdash d : T$  is derivable and  $d \rightarrow_{ev} d'$  holds. Then  $\emptyset; \emptyset; \emptyset \vdash d' : T$  is also derivable.
2. Assume that  $\emptyset; \emptyset; \overline{V}; \emptyset \vdash d : CT$  is derivable and  $(ST, d) \rightarrow_{ev/st} (ST', d')$  holds for some state  $ST$  such that  $ST$  meets  $\overline{V}$ . Then we have  $\overline{V}'$  such that  $ST'$  meets  $\overline{V}'$  and the judgment  $\emptyset; \emptyset; \overline{V}'; \emptyset \vdash d' : CT$  is derivable.

**Proof** For (1), the proof is by straightforward structural induction on the typing derivations of  $\emptyset; \emptyset; \emptyset \vdash d : T$ . For (2), the proof is by structural induction on the typing derivation of  $\emptyset; \emptyset; \overline{V}; \emptyset \vdash d : CT$ . ■

**Theorem 2.6 (Progress)** *We have the following.*

1. Assume that  $\emptyset; \emptyset; \emptyset \vdash d : T$  is derivable. Then we have that either (1)  $d$  is a value or  $d \rightarrow_{ev} d'$  for some  $d'$ .
2. Assume that  $\emptyset; \emptyset; \overline{V}; \emptyset \vdash d : CT$  is derivable and  $ST$  meets  $\overline{V}$ . Then we have that either (1)  $d$  is a value, or (2) for any  $ST_0$  disjoint from  $ST$ ,  $(ST_0 \otimes ST, d) \rightarrow_{ev/st} (ST_0 \otimes ST', d')$  for some  $ST'$  and  $d'$ , or (3)  $d = E[dcf(v_1, \dots, v_n)]$  such that  $dcf(v_1, \dots, v_n)$  is not a redex.

**Proof** By structural induction on the typing derivation of  $\emptyset; \emptyset; \overline{V}; \emptyset \vdash d : CT$ . ■

persistent stateful views	$W ::= !V \mid W_1 \wedge W_2 \mid P \supset W$
types	$T ::= \dots \mid W \supset_w T \mid W \wedge_w T \mid (\overline{V} \mid T) \rightarrow^* CT$
dynamic terms	$d ::= \dots \mid \supset_w^+(d) \mid \supset_w^-(d) \mid \wedge_w(d) \mid \mathbf{let} \wedge_w(x) = d_1 \mathbf{in} d_2$
values	$v ::= \dots \mid \supset_w^+(v) \mid \wedge_w(v)$

 Figure 13: The extended syntax for the dynamics of *ATS/SV*

### 2.3 Erasure

We present a function from dynamic terms to untyped  $\lambda$ -expressions that preserves semantics. We can then define a function  $|\cdot|$  as follows that translates dynamic terms into erasures.

$$\begin{aligned}
 |x| &= x \\
 |dcc(d_1, \dots, d_n)| &= dcc(|d_1|, \dots, |d_n|) \\
 |\mathbf{lam} x.d| &= \mathbf{lam} x.|d| \\
 |\mathbf{app}(d_1, d_2)| &= \mathbf{app}(|d_1|, |d_2|) \\
 |\supset^+(d)| &= |d| \\
 |\supset^-(d)| &= |d| \\
 |\wedge(d)| &= |d| \\
 |\mathbf{let} x = \wedge(d_1) \mathbf{in} d_2| &= \mathbf{let} x = |d_1| \mathbf{in} |d_2| \\
 |\forall^+ d| &= |d| \\
 |\forall^- d| &= |d| \\
 |\exists(d)| &= |d| \\
 |\mathbf{let} \exists(x) = d_1 \mathbf{in} d_2| &= \mathbf{let} x = |d_1| \mathbf{in} |d_2|
 \end{aligned}$$

**Theorem 2.7** *We have the following.*

1. Assume  $\mathcal{D} :: \emptyset; \emptyset; \emptyset \vdash d : T$ . Then  $d \rightarrow_{ev}^* v$  holds for some  $v$  such that  $|d| = |v|$ .

2. Assume  $\mathcal{D} :: \emptyset; \emptyset; \overline{V}; \emptyset \vdash d : CT$  and  $ST$  meets  $\overline{V}$ .

(a) If  $(ST, d) \rightarrow_{ev/st}^* (ST', v)$ , then  $(|ST|, |d|) \rightarrow_{ev/st}^* (|ST'|, |v|)$ .

(b) If  $(|ST|, |d|) \rightarrow_{ev/st}^* (ST_0, v_0)$ , then there is a state  $ST'$  and a value  $v$  such that  $(ST, d) \rightarrow_{ev/st}^* (ST', v)$  and  $|ST'| = ST_0$  and  $|v| = v_0$ .

**Proof** Straightforward. ■

With Theorem 2.7, we can evaluate a dynamic term  $d$  by simply evaluating the erasure of  $d$ .

### 2.4 Persistent Stateful Views

We now introduce a notion called *persistent stateful view*, which is needed in the implementation of a reference (like one in ML). The extended syntax for *ATS/SV* is given in Figure 13. We use

$$\begin{array}{c}
\frac{ST_0 \models V \text{ for some } ST_0 \subseteq ST}{ST \models !V} \qquad \frac{ST \models W_1 \quad ST \models W_2}{ST \models W_1 \wedge W_2} \\
\frac{\emptyset; \emptyset \models P \quad ST \models W}{ST \models P \supset W} \qquad \frac{\emptyset; \emptyset \models \neg P}{ST \models P \supset W}
\end{array}$$

Figure 14: The rules for states meeting persistent views

$W$  for persistent stateful views and  $\overline{W}$  of a (possibly empty) sequence of persistent views. We use a judgment of the form  $ST \models W$  to mean that  $ST$  meets  $W$  and present the rules for deriving such a judgment in Figure 14, and we write  $ST \models \overline{W}$  to mean that  $ST \models W$  holds for each  $W$  in  $\overline{W}$ . In addition, we introduce a judgment of the form  $ST \models_V W$  to mean that  $ST_0 \otimes ST \models W$  holds whenever  $ST_0 \models V$  does for a state  $ST_0$  disjoint from  $ST$ . We write  $ST \models_V \overline{W}$  to mean that  $ST \models_V W$  holds for each  $W$  in  $\overline{W}$ .

$$\begin{array}{c}
\frac{W \in \overline{W}}{\Sigma; \overline{P}; \overline{W} \models W} \qquad \frac{\Sigma; \overline{P} \models \text{ff}}{\Sigma; \overline{P}; \overline{W} \models W} \\
\frac{\Sigma; \overline{P}; \overline{W} \models W_1 \quad \Sigma; \overline{P}; \overline{W} \models W_2}{\Sigma; \overline{P}; \overline{W} \models W_1 \wedge W_2} \\
\frac{\Sigma; \overline{P}; \overline{W} \models W_1 \wedge W_2}{\Sigma; \overline{P}; \overline{W} \models W_1} \qquad \frac{\Sigma; \overline{P}; \overline{W} \models W_1 \wedge W_2}{\Sigma; \overline{P}; \overline{W} \models W_2} \\
\frac{\Sigma; \overline{P}, P; \overline{W} \models W}{\Sigma; \overline{P}; \overline{W} \models P \supset W} \qquad \frac{\Sigma; \overline{P}; \overline{W} \models P \supset W \quad \Sigma; \overline{P} \models P}{\Sigma; \overline{P}; \overline{W} \models W} \\
\frac{\Sigma; \overline{P}; \emptyset \models V_1 \multimap V_2 \quad \Sigma; \overline{P}; \emptyset \models V_2 \multimap V_1 \quad \Sigma; \overline{P}; \overline{W} \models !V_1}{\Sigma; \overline{P}; \overline{W} \models !V_2} \\
\frac{\Sigma; \overline{P}; \overline{W} \models !(V_1 \otimes V_2)}{\Sigma; \overline{P}; \overline{W} \models !V_1} \qquad \frac{\Sigma; \overline{P}; \overline{W} \models !(V_1 \otimes V_2)}{\Sigma; \overline{P}; \overline{W} \models !V_2}
\end{array}$$

Figure 15: Some rules for persistent view constraints

We introduce a new form of constraint:  $\Sigma; \overline{P}; \overline{W} \models W$  (persistent view) and present some of the rules for deriving such a constraint in Figure 15. We can now form two new forms of types:  $W \supset_w T$  and  $W \wedge_w T$ . Intuitively,  $W \supset_w T$  is like a guarded type: A value of type  $W \supset_w T$  can be used only if the persistent view  $W$  is met. Similarly,  $W \wedge_w T$  is like an asserting type: A value of type  $W \wedge_w T$  indicates that the persistent view  $W$  is met. As for a type of the form  $(\overline{V} \mid T) \rightarrow^* CT$ , it is almost identical to the type  $(\overline{V} \mid T) \rightarrow CT$ ; the essential difference is that a call to a function of the former type may involve persistent views while a call to a function of the

latter type may not; in particular, the latter is considered a subtype of the former.

Given a pair of disjoint states  $ST_1$  and  $ST_2$ , we call  $(ST_1; ST_2)$  a combined state, which meets  $(\overline{W}; \overline{V})$  if both  $ST_1 \models \overline{W}$  and  $ST_2 \models \overline{V}$  hold. We call  $ST_1$  and  $ST_2$  the persistent component and the ephemeral component in the combined state  $(ST_1; ST_2)$ . Also, we say  $(ST_1; ST_2) \models_V (\overline{W}; \overline{V})$  holds if both  $ST_1 \models_V \overline{W}$  and  $ST_2 \models \overline{V}$  hold. We may also mean  $(ST_1; ST_2) \models_V (\overline{W}; \overline{V})$  by saying that  $(ST_1; ST_2)$  meets  $(\overline{W}/V; \overline{V})$ .

Given a state  $ST$ , we say that  $ST$  meets  $(\overline{W}; \overline{V})$  if  $ST = ST_1 \otimes ST_2$  and  $(ST_1; ST_2)$  meets  $(\overline{W}; \overline{V})$ . Similarly, we say that  $ST$  meets  $(\overline{W}/V; \overline{V})$  if  $ST = ST_1 \otimes ST_2$  and  $(ST_1; ST_2)$  meets  $(\overline{W}/V; \overline{V})$ .

The previous two forms of typing judgment  $\Sigma; \overline{P}; \Delta \vdash d : T$  and  $\Sigma; \overline{P}; \overline{W}; \Delta \vdash$  are modified to the following two forms:  $\Sigma; \overline{P}; \overline{W}; \Delta \vdash d : T$  and  $\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash d : CT$ , respectively. General speaking, if  $\emptyset; \emptyset; \overline{W}; \emptyset \vdash d : T$ , then  $d$  can be evaluated at any combined state  $(ST_1; ST_2)$  such that  $ST_1 \models \overline{W}$  holds, and a value of type  $T$  is returned if the evaluation terminates; if  $\emptyset; \emptyset; \overline{W}; \overline{V}; \emptyset \vdash d : T$  is derivable, then the evaluation of  $d$  can start at any combined state  $(ST_1; ST_2)$  that meets  $(\overline{W}; \overline{V})$ , and a value of type  $T[\Theta]$  is returned (if the evaluation terminates) and a combined state  $(ST'_1; ST'_2)$  is reached that meets  $(\overline{W}; \overline{V}'[\Theta])$ , where  $CT = \exists \Sigma, \overline{P}. (\overline{V}' \mid T)$  and  $\Theta : \Sigma$  is a substitution that makes each proposition  $P$  in  $\overline{P}[\Theta]$  hold.

Furthermore, a subtype judgment is now of the form  $\Sigma; \overline{P}; \overline{W} \vdash T_1 \leq_{tp} T_2$  or  $\Sigma; \overline{P}; \overline{W} \vdash CT_1 \leq_{ctp} CT_2$ , and the rules for deriving subtype judgments are given in Figure 16.

The rules for deriving such typing judgments are given in Figure 17, Figure 18 and Figure 19.

The rule **(ty-inv-intro)** indicates that a combined state  $(ST_1 \otimes ST_{22}; ST_{21})$  meets  $(\overline{W}; !V; \overline{V})$  if  $(ST_1; ST_{21} \otimes ST_{22})$  meets  $(\overline{W}; \overline{V}, V)$ . Intuitively, this means that a state in the ephemeral component of a combined state can be shifted to the persistent component of the combined state. The rule **(ty-inv-elim)** is somewhat the opposite of the rule **(ty-inv-intro)**, indicating that a state in the persistent component of a combined state may be shifted to the ephemeral component of the combined state. Note that 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  $\Sigma$ . The shifting of a state from the persistent component of a combined state to the ephemeral component can only be done in a rather restricted manner:  $(ST_{11} \otimes ST_{12}; ST_2)$  meeting  $(\overline{W}; !V; \overline{V})$  can be used as a combined state  $(ST_{11}; ST_2 \otimes ST_{12})$  meeting  $(\overline{W}; \overline{V}, V)$  only when computation involving no persistent views is to be performed and a state  $ST'_{12}$  meeting  $V$  is guaranteed to be produced in the ephemeral component of the combined state reached at the end of the computation. In other words, the ephemeral component can only temporarily borrow  $ST_{12}$  from the persistent component.

**Lemma 2.8 (Substitution)** *We have the following.*

1. Assume that  $\Sigma \vdash s : \sigma$  is derivable.

- (a) If  $\Sigma, a : \sigma; \overline{P}; V \models V'$  is derivable, then  $\Sigma; \overline{P}[a \mapsto s]; V[a \mapsto s] \models V'[a \mapsto s]$  is also derivable.
- (b) If  $\Sigma, a : \sigma; \overline{P}; \overline{W} \models T \leq_{tp} T'$  is derivable, then  $\Sigma; \overline{P}[a \mapsto s]; \overline{W}[a \mapsto s] \models T[a \mapsto s] \leq_{tp} T'[a \mapsto s]$  is also derivable.
- (c) If  $\Sigma, a : \sigma; \overline{P}; \overline{W} \models CT \leq_{ctp} CT'$  is derivable, then  $\Sigma; \overline{P}[a \mapsto s]; \overline{W}[a \mapsto s] \models CT[a \mapsto s] \leq_{ctp} CT'[a \mapsto s]$  is also derivable.

$$\begin{array}{c}
 \hline
 \Sigma; \overline{P}; \overline{W} \models T \leq_{tp} \mathbf{top} \\
 \hline
 \Sigma; \overline{P}; \overline{W} \models T \leq_{tp} T \\
 \hline
 \frac{\vdash \delta(\sigma_1, \dots, \sigma_n) \quad \Sigma; \overline{P}; \overline{W} \models s_i \equiv_{\sigma_i} s'_i \text{ for } 1 \leq i \leq n}{\Sigma; \overline{P}; \overline{W} \models \delta(s_1, \dots, s_n) \leq_{tp} \delta(s'_1, \dots, s'_n)} \\
 \hline
 \frac{\Sigma; \overline{P}; \overline{W} \models T' \leq_{tp} T \quad \Sigma; \overline{P}; \overline{W}; \overline{V}' \models \overline{V} \quad \Sigma; \overline{P}; \overline{W} \models CT \leq_{ctp} CT'}{\Sigma; \overline{P}; \overline{W} \models (\overline{V} | T) \rightarrow^* CT \leq_{tp} (\overline{V}' | T') \rightarrow^* CT'} \\
 \hline
 \frac{\Sigma; \overline{P}, P' \models P \quad \Sigma; \overline{P}, P'; \overline{W} \models T \leq_{tp} T'}{\Sigma; \overline{P}; \overline{W} \models P \supset T \leq_{tp} P' \supset T'} \\
 \hline
 \frac{\Sigma; \overline{P}; \overline{W}, W' \models W \quad \Sigma; \overline{P}; \overline{W}, W' \models T \leq_{tp} T'}{\Sigma; \overline{P}; \overline{W} \models W \supset_w T \leq_{tp} W' \supset_w T'} \\
 \hline
 \frac{\Sigma, a : \sigma; \overline{P}; \overline{W} \models T \leq_{tp} T'}{\Sigma; \overline{P}; \overline{W} \models \forall a : \sigma. T \leq_{tp} \forall a : \sigma. T'} \\
 \hline
 \frac{\Sigma; \overline{P}, P; \overline{W} \models P' \quad \Sigma; \overline{P}, P; \overline{W} \models T \leq_{tp} T'}{\Sigma; \overline{P}; \overline{W} \models P \wedge T \leq_{tp} P' \wedge T'} \\
 \hline
 \frac{\Sigma; \overline{P}; \overline{W}, W \models W' \quad \Sigma; \overline{P}; \overline{W}, W \models T \leq_{tp} T'}{\Sigma; \overline{P}; \overline{W} \models W \wedge_w T \leq_{tp} W' \wedge_w T'} \\
 \hline
 \frac{\Sigma, a : \sigma; \overline{P}; \overline{W} \models T \leq_{tp} T'}{\Sigma; \overline{P}; \overline{W} \models \exists a : \sigma. T \leq_{tp} \exists a : \sigma. T'} \\
 \hline
 \frac{\Sigma, \Sigma_0; \overline{P}, \overline{P}_0 \models \overline{P}'_0 \quad \Sigma, \Sigma_0; \overline{P}, \overline{P}_0; \overline{W}; \overline{V}' \models \overline{V}' \quad \Sigma, \Sigma_0; \overline{P}, \overline{P}_0; \overline{W} \models T \leq_{tp} T'}{\Sigma; \overline{P}; \overline{W} \models \exists \Sigma_0, \overline{P}_0. (\overline{V} | T) \leq_{ctp} \exists \Sigma_0, \overline{P}'_0. (\overline{V}' | T')} \\
 \hline
 \end{array}$$

Figure 16: The subtype rules

$$\begin{array}{c}
 \frac{\Sigma; \overline{P}; \overline{W}; \Delta \vdash d : T \quad \Sigma; \overline{P}; \overline{W} \models T \leq_{tp} T'}{\Sigma; \overline{P}; \overline{W}; \Delta \vdash d : T'} \text{ (ty-sub)} \\
 \frac{\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash d : CT \quad \Sigma; \overline{P}; \overline{W} \models CT \leq_{ctp} CT'}{\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash d : CT'} \text{ (ty-sub}_c\text{)} \\
 \frac{\Sigma; \overline{P}; \overline{W}; \Delta \vdash d : T}{\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash d : (\overline{V} \mid T)} \text{ (ty-state)} \\
 \frac{\Delta(xf) = T}{\Sigma; \overline{P}; \overline{W}; \Delta \vdash xf : T} \text{ (ty-var)} \\
 \frac{\begin{array}{c} \vdash dcc : \forall \Sigma_0, \overline{P}_0, \overline{W}_0. (T_1, \dots, T_n) \Rightarrow T \\ \Sigma \vdash \Theta : \Sigma_0 \quad \Sigma; \overline{P} \models \overline{P}_0[\Theta] \quad \Sigma; \overline{P}; \overline{W} \models \overline{W}_0 \\ \Sigma; \overline{P}; \overline{W} \vdash d_i : T_i[\Theta] \text{ for } 1 \leq i \leq n \end{array}}{\Sigma; \overline{P}; \overline{W}; \Delta \vdash dcc(d_1, \dots, d_n) : T[\Theta]} \text{ (ty-dcc)} \\
 \frac{\begin{array}{c} \vdash dcf : \forall \Sigma_0, \overline{P}_0, \overline{W}_0. (\overline{V}_0 \mid T_1, \dots, T_n) \Rightarrow CT \\ \Sigma \vdash \Theta : \Sigma_0 \quad \Sigma; \overline{P}; \overline{W} \models \overline{P}_0[\Theta] \quad \Sigma; \overline{P}; \overline{W} \models \overline{W}_0 \quad \Sigma; \overline{P}; \overline{V} \models \overline{V}_0[\Theta] \\ \Sigma; \overline{P}; \overline{W} \vdash d_i : T_i[\Theta] \text{ for } 1 \leq i \leq n \end{array}}{\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash dcf(d_1, \dots, d_n) : CT[\Theta]} \text{ (ty-dcf)} \\
 \frac{\begin{array}{c} \Sigma; \overline{P}; \overline{W}; \Delta \vdash d_1 : \mathbf{bool}(P) \\ \Sigma; \overline{P}, P; \overline{W}; \overline{V}; \Delta \vdash d_2 : CT \quad \Sigma; \overline{P}, \neg P; \overline{W}; \overline{V}; \Delta \vdash d_3 : CT \end{array}}{\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash \mathbf{if}(d_1, d_2, d_3) : CT} \text{ (ty-if)} \\
 \frac{\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta, x : T \vdash d : CT}{\Sigma; \overline{P}; \overline{W}; \Delta \vdash \mathbf{lam } x.d : (\overline{V} \mid T) \rightarrow CT} \text{ (ty-lam)} \\
 \frac{\Sigma; \overline{P}; \overline{W}; \Delta, f : T \vdash d : T}{\Sigma; \overline{P}; \overline{W}; \Delta \vdash \mathbf{fix } f.d : T} \text{ (ty-fix)} \\
 \frac{\Sigma; \overline{P}; \overline{W}; \Delta \vdash d_1 : (\overline{V}' \mid T) \rightarrow^* CT \quad \Sigma; \overline{P}; \overline{V} \models \overline{V}' \quad \Sigma; \overline{P}; \overline{W}; \Delta \vdash d_2 : T}{\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash d_1(d_2) : CT} \text{ (ty-app)}
 \end{array}$$

Figure 17: The typing rules for dynamics (1)

$$\begin{array}{c}
 \frac{\Sigma; \overline{P}, P; \overline{W}; \Delta \vdash v : T}{\Sigma; \overline{P}; \overline{W}; \Delta \vdash \supset^+(v) : P \supset T} \text{ (ty-}\supset\text{-intro)} \\
 \frac{\Sigma; \overline{P}; \overline{W}; \Delta \vdash d : P \supset T \quad \Sigma; \overline{P} \models P}{\Sigma; \overline{P}; \overline{W}; \Delta \vdash \supset^-(d) : T} \text{ (ty-}\supset\text{-elim)} \\
 \frac{\Sigma; \overline{P}; \overline{W}, W; \Delta \vdash d : T}{\Sigma; \overline{P}; \overline{W}; \Delta \vdash \supset_w^+(d) : W \supset_w T} \text{ (ty-}\supset_w\text{-intro)} \\
 \frac{\Sigma; \overline{P}; \overline{W}; \Delta \vdash d : W \supset_w T \quad \Sigma; \overline{P}; \overline{W} \models W}{\Sigma; \overline{P}; \overline{W}; \Delta \vdash \supset_w^-(d) : T} \text{ (ty-}\supset_w\text{-elim)} \\
 \frac{\Sigma, a : \sigma; \overline{P}; \overline{W}; \Delta \vdash v : T}{\Sigma; \overline{P}; \overline{W}; \Delta \vdash \forall^+(v) : \forall a : \sigma. T} \text{ (ty-}\forall\text{-intro)} \\
 \frac{\Sigma; \overline{P}; \overline{W}; \Delta \vdash d : \forall a : \sigma. T \quad \Sigma \vdash s : \sigma}{\Sigma; \overline{P}; \overline{W}; \Delta \vdash \forall^-(d) : T[a \mapsto s]} \text{ (ty-}\forall\text{-elim)} \\
 \frac{\Sigma; \overline{P} \models P \quad \Sigma; \overline{P}; \overline{W}; \Delta \vdash d : T}{\Sigma; \overline{P}; \overline{W}; \Delta \vdash \wedge(d) : P \wedge T} \text{ (ty-}\wedge\text{-intro)} \\
 \frac{\Sigma; \overline{P}; \overline{W}; \Delta \vdash d_1 : P \wedge T \quad \Sigma; \overline{P}, P; \overline{W}; \overline{V}; \Delta, x : T \vdash d_2 : CT}{\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash \mathbf{let} \wedge(x) = d_1 \mathbf{in} d_2 : CT} \text{ (ty-}\wedge\text{-elim)} \\
 \frac{\Sigma; \overline{P}; \overline{W} \models W \quad \Sigma; \overline{P}; \overline{W}; \Delta \vdash d : T}{\Sigma; \overline{P}; \overline{W}; \Delta \vdash \wedge_w(d) : W \wedge_w T} \text{ (ty-}\wedge_w\text{-intro)} \\
 \frac{\Sigma; \overline{P}; \overline{W}; \Delta \vdash d_1 : W \wedge_w T \quad \Sigma; \overline{P}; \overline{W}, W; \overline{V}; \Delta, x : T \vdash d_2 : CT}{\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash \mathbf{let} \wedge_w(x) = d_1 \mathbf{in} d_2 : CT} \text{ (ty-}\wedge_w\text{-elim)} \\
 \frac{\Sigma \vdash s : \sigma \quad \Sigma; \overline{P}; \overline{W}; \Delta \vdash d : T[a \mapsto s]}{\Sigma; \overline{P}; \overline{W}; \Delta \vdash \exists(d) : \exists a : \sigma. T} \text{ (ty-}\exists\text{-intro)} \\
 \frac{\Sigma; \overline{P}; \overline{W}; \Delta \vdash d_1 : \exists a : \sigma. T \quad \Sigma, a : \sigma; \overline{P}; \overline{W}; \overline{V}; \Delta, x : T \vdash d_2 : CT}{\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash \mathbf{let} \exists(x) = d_1 \mathbf{in} d_2 : CT} \text{ (ty-}\exists\text{-elim)} \\
 \frac{\Sigma \vdash \Theta : \Sigma' \quad \Sigma; \overline{P} \models \overline{P}'[\Theta] \quad \Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash d : (\overline{V}'[\Theta] \mid T[\Theta])}{\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash d : \exists \Sigma', \overline{P}'.(\overline{V}' \mid T)} \text{ (ty-}\exists_c\text{-intro)} \\
 \frac{\Sigma; \overline{P}; \overline{W}; \overline{V}_1; \Delta \vdash d_1 : \exists \Sigma', \overline{P}'.(\overline{V}'_1 \mid T) \quad \Sigma, \Sigma'; \overline{P}, \overline{P}'; \overline{W}; \overline{V}'_1, \overline{V}_2; \Delta, x : T \vdash d_2 : CT}{\Sigma; \overline{P}; \overline{W}; \overline{V}_1, \overline{V}_2; \Delta \vdash \mathbf{let}_c x = d_1 \mathbf{in} d_2 : CT} \text{ (ty-}\exists_c\text{-elim)}
 \end{array}$$

Figure 18: The typing rules for dynamics (2)

$$\begin{array}{c}
 \frac{\Sigma; \overline{P}; \overline{W}; !V; \overline{V}; \Delta \vdash d : CT}{\Sigma; \overline{P}; \overline{W}; \overline{V}; V; \Delta \vdash d : CT} \text{ (ty-inv-intro)} \\
 \frac{\Sigma; \overline{P}; \overline{W} \models !V \quad \Sigma; \overline{P}; \overline{V}; V; \Delta \vdash d : CT[V]}{\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash d : CT} \text{ (ty-inv-elim)} \\
 \frac{\Sigma; \overline{P}; \overline{V}; \Delta \vdash d : CT[V]}{\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash_V d : CT} \text{ (ty-thin)} \\
 \frac{\Sigma; \overline{P}; \overline{W}; \overline{V}_1; \Delta \vdash_V d_1 : \exists \Sigma', \overline{P}'. (\overline{V}'_1 \mid T) \quad \Sigma, \Sigma'; \overline{P}, \overline{P}'; \overline{W}; \overline{V}'_1, \overline{V}_2; \Delta, x : T \vdash d_2 : CT}{\Sigma; \overline{P}; \overline{W}; \overline{V}_1, \overline{V}_2; \Delta \vdash_V \mathbf{let}_c x = d_1 \mathbf{in} d_2 : CT} \text{ (ty-}\exists_c\text{-elim')}
 \end{array}$$

Figure 19: The typing rules for dynamics (3)

- (d) If  $\Sigma, a : \sigma; \overline{P}; \overline{W} \vdash d : T$  is derivable, then  $\Sigma; \overline{P}[a \mapsto s]; \overline{W}[a \mapsto s] \vdash d : T[a \mapsto s]$  is also derivable.
- (e) If  $\Sigma, a : \sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash d : CT$  is derivable, then  $\Sigma; \overline{P}[a \mapsto s]; \overline{W}[a \mapsto s]; \overline{V}[a \mapsto s] \vdash d : CT[a \mapsto s]$  is also derivable.
- (f) If  $\Sigma, a : \sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash_V d : CT$  is derivable, then  $\Sigma; \overline{P}[a \mapsto s]; \overline{W}[a \mapsto s]; \overline{V}[a \mapsto s] \vdash_{V[a \mapsto s]} d : CT[a \mapsto s]$  is also derivable.
2. Assume that  $\Sigma; \overline{P} \models P$  holds.
- (a) If  $\Sigma; \overline{P}; P; \overline{W}; \Delta \vdash d : T$  is derivable, then  $\Sigma; \overline{P}; \overline{W} \vdash d : T$  is also derivable.
- (b) If  $\Sigma; \overline{P}; P; \overline{W}; \overline{V}; \Delta \vdash d : CT$  is derivable, then  $\Sigma; \overline{P}; \overline{W}; \overline{V} \vdash d : CT$  is also derivable.
- (c) If  $\Sigma; \overline{P}; P; \overline{W}; \overline{V} \vdash_V \Delta d : CT$  is derivable, then  $\Sigma; \overline{P}; \overline{W}; \overline{V} \vdash_V d : CT$  is also derivable.
3. Assume that  $\Sigma; \overline{P}; \overline{W} \models W$  holds.
- (a) If  $\Sigma; \overline{P}; \overline{W}; W \vdash d : T$  is derivable, then  $\Sigma; \overline{P}; \overline{W} \vdash d : T$  is also derivable.
- (b) If  $\Sigma; \overline{P}; \overline{W}; W; \overline{V} \vdash d : CT$  is derivable, then  $\Sigma; \overline{P}; \overline{W}; \overline{V} \vdash d : CT$  is also derivable.
- (c) If  $\Sigma; \overline{P}; \overline{W}; W; \overline{V} \vdash_V d : CT$  is derivable, then  $\Sigma; \overline{P}; \overline{W}; \overline{V} \vdash_V d : CT$  is also derivable.
4. Assume that  $\Sigma; \overline{P}; \overline{V}_1 \models \overline{V}'_1$  holds.
- (a) If  $\Sigma; \overline{P}; \overline{W}; \overline{V}'_1, \overline{V}_2; \Delta \vdash d : CT$  is derivable, then  $\Sigma; \overline{P}; \overline{W}; \overline{V}_1, \overline{V}_2 \vdash d : CT$  is also derivable.
- (b) If  $\Sigma; \overline{P}; \overline{W}; \overline{V}'_1, \overline{V}_2 \vdash_V \Delta d : CT$  is derivable, then  $\Sigma; \overline{P}; \overline{W}; \overline{V}_1, \overline{V}_2 \vdash_V d : CT$  is also derivable.
5. Assume that  $\Sigma; \overline{P}; \overline{W}; \Delta \vdash v : T_1$  is derivable.

- (a) If  $\Sigma; \overline{P}; \overline{W}; \Delta, x : T_1 \vdash d : T_2$  is derivable, then  $\Sigma; \overline{P}; \overline{W}; \Delta \vdash d[x \mapsto v] : T_2$  is also derivable.
- (b) If  $\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta, x : T_1 \vdash d : CT_2$  is derivable, then  $\Sigma; \overline{P}; \overline{W}; \overline{V}; \Delta \vdash d[x \mapsto v] : CT_2$  is also derivable.

**Proof** Standard. ■

**Theorem 2.9 (Subject Reduction)** *We have the following:*

1. Assume that  $\emptyset; \emptyset; \overline{W}; \emptyset \vdash d : T$  is derivable and  $d \rightarrow_{ev} d'$  holds. Then  $\emptyset; \emptyset; \overline{W}; \emptyset \vdash d : T'$  is also derivable.
2. Assume that  $\emptyset; \emptyset; !V; \overline{V}; \emptyset \vdash d : CT$  is derivable and  $(ST, d) \rightarrow_{ev/st} (ST', d')$  holds for some state  $ST$  such that  $ST$  meets  $(!V; \overline{V})$ . Then
  - we have  $V'$  and  $\overline{V}'$  such that both  $\emptyset; \emptyset; !V'; \overline{V}'; \emptyset \vdash d' : CT$  and  $\emptyset; \emptyset; !V' \models !V$  are derivable and  $ST'$  meets  $(!V'; \overline{V}')$ , or
  - we have  $\overline{V}'$  and  $V_0$  such that both  $\emptyset; \emptyset; !V \models !V_0$  and  $\emptyset; \emptyset; !V; \overline{V}'; \emptyset \vdash_{V_0} d' : CT$  are derivable and  $ST'$  meets  $(!V/V_0; \overline{V}')$ .
3. Assume that  $\emptyset; \emptyset; !V; \overline{V}; \emptyset \vdash_{V_0} d : CT$  is derivable and  $(ST, d) \rightarrow_{ev/st} (ST', d')$  holds for some state  $ST$  such that  $ST$  meets  $(!V/V_0; \overline{V})$ . Then
  - we have  $\overline{V}'$  such that the judgment  $\emptyset; \emptyset; !V; \overline{V}'; \emptyset \vdash_{V_0} d' : CT$  is derivable and  $ST'$  meets  $(!V/V_0; \overline{V}')$ , or
  - we have  $\overline{V}'$  such that the judgment  $\emptyset; \emptyset; !V; \overline{V}'; \emptyset \vdash d' : CT$  is derivable and  $ST'$  meets  $(!V; \overline{V}')$ .

**Proof** The proof proceeds by induction on the height of the typing derivations  $\mathcal{D}$  of  $\emptyset; \emptyset; \overline{W}; \emptyset \vdash d : T$ ,  $\emptyset; \emptyset; !V; \overline{V}; \emptyset \vdash d : CT$  and  $\emptyset; \emptyset; !V; \overline{V}; \emptyset \vdash_{V_0} d : CT$ . We present some interesting cases as follows.

- $\mathcal{D}$  is of the following form:

$$\frac{\mathcal{D}_1 :: \emptyset; \emptyset; !V, !V_0; \overline{V}; \Delta \vdash d : CT}{\emptyset; \emptyset; !V; \overline{V}, V_0; \Delta \vdash d : CT} \text{ (ty-inv-intro)}$$

We may assume the existence of  $\mathcal{D}'_1 :: \emptyset; \emptyset; !(V \otimes V_0); \overline{V}; \emptyset \vdash d : CT$  such that  $\mathbf{h}(\mathcal{D}'_1) = \mathbf{h}(\mathcal{D}_1)$ . By induction hypothesis on  $\mathcal{D}'_1$ , the case follows immediately.

- $\mathcal{D}$  is of the following form:

$$\frac{\emptyset; \emptyset; !V \models !V_0 \quad \mathcal{D}_1 :: \emptyset; \emptyset; \overline{V}, V_0; \emptyset \vdash d : CT[V_0]}{\emptyset; \emptyset; !V; \overline{V}; \emptyset \vdash d : CT} \text{ (ty-inv-elim)}$$

So we have  $\mathcal{D}'_1 :: \emptyset; \emptyset; \overline{V}' ; \emptyset \vdash d' : CT[V_0]$  for some  $\overline{V}'$  and then the following derivation  $\mathcal{D}'$ :

$$\frac{\mathcal{D}'_1 :: \emptyset; \emptyset; \overline{V}' ; \emptyset \vdash d' : CT[V_0]}{\emptyset; \emptyset; !V; \overline{V}' ; \emptyset \vdash_{V_0} d : CT} \text{ (ty-thin)}$$

Other cases can be handled similarly. ■

**Theorem 2.10 (Progress)** *We have the following:*

1. Assume that  $\emptyset; \emptyset; !V; \emptyset \vdash d : T$  is derivable. Then we have that (1) either  $d$  is a value, or (2)  $d \rightarrow_{ev} d'$  for some  $d'$ , or (3)  $d = E[dcf(v_1, \dots, v_n)]$  such that  $dcf(v_1, \dots, v_n)$  is not a redex.
2. Assume that  $\emptyset; \emptyset; !V; \overline{V}; \emptyset \vdash d : CT$  is derivable and  $ST$  meets  $(!V; \overline{V})$ . Then we have that either (1)  $d$  is a value, or (2)  $(ST, d) \rightarrow_{ev/st} (ST', d')$  for some  $ST'$  and  $d'$ , or (3)  $d = E[dcf(v_1, \dots, v_n)]$  such that  $dcf(v_1, \dots, v_n)$  is not a redex.
3. Assume that  $\emptyset; \emptyset; !V; \overline{V}; \emptyset \vdash_{V_0} d : CT$  is derivable and  $ST$  meets  $(!V \setminus V_0; \overline{V})$ . Then we have that either (1)  $d$  is a value, or (2)  $(ST, d) \rightarrow_{ev/st} (ST', d')$  for some  $ST'$  and  $d'$ , or (3)  $d = E[dcf(v_1, \dots, v_n)]$  such that  $dcf(v_1, \dots, v_n)$  is not a redex.

**Proof** By structural induction. ■

### 3 Programming with Stateful Views

We are currently in the process of designing and implementing ATS, a programming language with a type system rooted in the framework *ATS*. In particular, we have already implemented a type-checker for ATS, which is largely based on the elaboration algorithm developed in Dependent ML (Xi 1998). It is beyond the scope of the paper to present a detailed formal account for the implementation of the type-checker. Instead, we use some concrete examples chosen from our current implementation to give the reader a feel as to how programming with stateful views can actually be done. The actual code for the presented examples and many more (e.g., a cyclic buffer implementation, a splay tree implementation) can be found on-line (Xi 2003).

There are three forms of constraints in *ATS/SV*:  $\Sigma; \overline{P} \models P$  (proposition) and  $\Sigma; \overline{P}; \overline{V} \models V$  (ephemeral view) and  $\Sigma; \overline{P}; \overline{W} \models W$  (persistent view). While we may assume the existence of an oracle for deciding the validity of constraints in the formalization of *ATS/SV*, we need to find a means that can effectively solve constraints encountered in practice.

By imposing some syntactic restriction (e.g., requiring that only linear static integer terms be allowed), we can turn the problem of solving proposition constraints into the well-known problem of linear integer programming, for which there exist a variety of effective methods.<sup>1</sup>

As for (ephemeral and persistent) view constraints, we employ an entirely different approach. We essentially require that the programmer be responsible for constructing proofs for solving such

---

<sup>1</sup>Though linear integer programming is NP-complete, the constraints we encounter in practice are often solved with great efficiency.

constraints. By Curry-Howard isomorphism, this amounts to constructing proofs for given views, and the validity of such proofs can be verified through type-checking. A concrete example of this style of programming can be found in (Chen, Zhu, and Xi 2004), where a case study on simulating dependent types in Haskell is conducted.

It can certainly be burdensome to construct proofs for views during programming. To alleviate the problem, we have implemented a strategy to handle proofs for views of the form  $T@L$  in an implicit manner. Given a pointer  $p$  of type  $\mathbf{ptr}(L)$ , the programmer may simply write  $!p$  to read from  $p$ ; the type-checker implicitly tries to find a proof  $pf$  of view  $T@L$  for some  $T$ ; if it succeeds,  $!p$  is elaborated into  $getVar(pf \mid p)$ ; otherwise, an error message is raised. This strategy also applies to writing to an address. For instance, the example in Figure 1 makes extensive use of this strategy.

### 3.1 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 20. 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:

$$\exists \lambda. (! (T_1 @ \lambda) \wedge ! (T_2 @ \lambda + 1)) \wedge_w \mathbf{ptr}(\lambda)$$

The function *makePair* is given the type  $\forall \tau_1. \forall \tau_2. (\tau_1, \tau_2) \rightarrow pair(\tau_1, \tau_2)$ , that is, it takes values of types  $T_1$  and  $T_2$  to form a value of type  $pair(T_1, T_2)$ .<sup>2</sup> Note that a use of *invar* in the body of *makePair* corresponds an application of the rule (**ty-inv-intro**).

The implementation of sum is more interesting. We define  $T_1 + T_2$  to be  $\exists a : two. \exists \lambda. W \wedge_w \mathbf{ptr}(\lambda)$ , where *two* is the subset sort  $\{a : int \mid 0 \leq a \wedge a \leq 1\}$  and  $W$  is

$$(! (int(a) @ \lambda) \wedge (a = 0 \supset ! (T_1 @ \lambda + 1)) \wedge (a = 1 \supset ! (T_2 @ \lambda + 1)))$$

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. Note that an use of *prunit* corresponds to an application of the following rule:

$$\frac{\Sigma; \overline{P} \models ff}{\Sigma; \overline{P}; \overline{W} \models W}$$

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.

---

<sup>2</sup>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
    // alloc2 allocates two memory units
    val '(pf1, pf2 | p) = alloc2 ()
    val '(pf1 | _) = setVar (pf1 | p, x1)
    val '(pf2 | _) = setVar (pf2 | p + 1, x2)
  in
    '(invar pf1, invar pf2 | p)
  end

typedef sum (a1: type, a2: type) =
  [l: addr, i: two]
  (!(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 '(pf1, pf2 | p) = alloc2 ()
    val '(pf1 | _) = setVar (pf1 | p, 0)
    val '(pf2 | _) = setVar (pf2 | p + 1, x)
  in
    '(invar pf1, invar pf2, prunit | p)
  end

// right injection
fun inr {a1: type, a2: type} (x: a2): sum (a1, a2) =
  let
    val '(pf1, pf2 | p) = alloc2 ()
    val '(pf1 | _) = setVar (pf1 | p, 1)
    val '(pf2 | _) = setVar (pf2 | p + 1, x)
  in
    '(invar pf1, prunit, invar pf2 | p)
  end

```

Figure 20: Implementations of product and sum

```

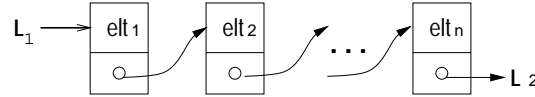
dataview slseg (type, int, addr, addr) =
  | {a:type, l:addr} SlsegNone (a, 0, l, l)
  | {a:type, n:nat, first, next, last | first <> null}
    // 'first <> null' is added so that nullity test can
    // be used to check whether a list segment is empty.
    SlsegSome (a, n+1, first, last) of
      ((a,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

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

Figure 21: A dataview for singly linked list segments

### 3.2 Singly-linked Lists

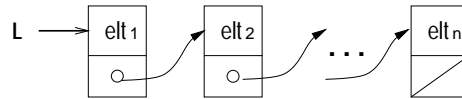
We declare in Figure 21 a dataview in ATS for representing singly-linked lists. Note that we write  $(T_0, \dots, T_n)@L$  for a sequence of views:  $T_0@(L + 0), \dots, T_n@(L + n)$ . Given a type  $T$ , an integer  $I$ , and two addresses  $L_1$  and  $L_2$ ,  $slseg(T, I, L_1, L_2)$  is a view for a singly-linked list segment pictured as follows:



such that

- each element of the segment is of type  $T$ , and
- the length of the segment is  $n$ , and
- the segment starts at  $L_1$  and ends at  $L_2$ .

There are two view proof constructors  $SlsegNone$  and  $SlsegSome$  associated with  $slseg$ . A singly-linked list is just a special case of singly-linked list segment that ends at the null address. Therefore,  $sllist(T, I, L)$  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  $I$ . In Figure 22, we present the implementation of a function that reverses a singly-linked list by resetting the pointers in it; the type of  $reverse$ , which is formally written as follows,

$$\forall \tau. \forall n : \text{nat}. \forall \lambda. (sllist(\tau, n, \lambda) \mid \mathbf{ptr}(\lambda)) \rightarrow \exists \lambda. (sllist(\tau, n, \lambda) \mid \mathbf{ptr}(\lambda))$$

means that it returns a pointer to a singly-linked list of length  $n$  when applied to a singly-linked list of length  $n$ ; the type of the inner function  $rev$  means that it returns a pointer to a singly-linked

```

fun reverse {a:type, n:nat, l:addr}
  (pf: sllist (a, n, l) | p: ptr (l))
  : [l: addr] '(sllist (a, n, l) | ptr (l)) =
  let
    fun rev {n1:nat,n2:nat,l1:addr,l2:addr}
      (pf1: sllist (a,n1,l1), pf2: sllist(a,n2,l2) |
        p1: ptr(l1), p2: ptr (l2))
      : [l:addr] '(sllist (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) = !(p2 + 1)
          val '(pf211 | _) = (p2 + 1 := p1)
          prval pf1 = SlsegSome ('(pf210, pf211), pf1)
        in
          rev (pf1, pf22 | p2, next)
        end
      end
  in
    rev (SlsegNone, pf | null, p)
  end

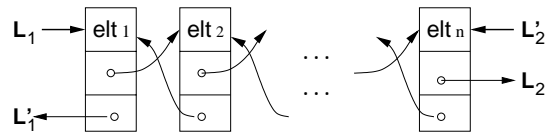
```

Figure 22: A function that reverses singly-linked lists

list of length  $n_1 + n_2$  when applied to two pointers to singly-linked lists of length  $n_1$  and  $n_2$ . We suggest that the reader compare this implementation with a corresponding one in (Reynolds 2002) to see how the proofs (of views) here correspond to the proofs given there.

### 3.3 Doubly-linked Lists

Doubly-linked lists are a form of commonly used data structure in practice. We declare two dataviews in Figure 23 for doubly-linked list segments. Given a type  $T$ , an integer  $I$  and four addresses  $L_1, L'_1, L_2, L'_2$ ,  $dlseg(T, I, L_1, L'_1, L_2, L'_2)$  and  $dlseg'(T, I, L_1, L'_1, L_2, L'_2)$  forms a (front) view and a (back) view, respectively, for a doubly-linked list segment pictured as follows:



where each element in the segment is of type  $T$ . For a doubly-linked list pictured as follows:

```

dataview dlseg (type, int, addr, addr, addr, addr) =
  | {a:type, l, l'} DlsegNone (a, 0, l, l', l, l')
  | {a:type, n:nat, first, prev, next, last, first' |
    first <> null, last <> null}
    DlsegSome (a, n+1, first, prev, next, last) of
      ((a, ptr first', ptr prev) @ first,
       dlseg(a, n, first', first, next, last))

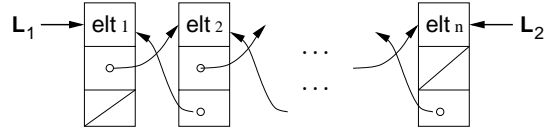
dataview dlseg' (type, int, addr, addr, addr, addr) =
  | {a:type, l, l'} DlsegNone' (a, 0, l, l', l, l')
  | {a:type, n:nat, first, prev, next, last, last' |
    first <> null, last <> null}
    DlsegSome' (a, n+1, first, prev, next, last) of
      ((a, ptr next, ptr last') @ last,
       dlseg' (a, n, first, prev, last, last'))

viewdef dllist (a:type, n:int, l1: addr, l2: addr) =
  dlseg (a, n, l1, null, null, l2)

viewdef dllist' (a:type, n:int, l1: addr, l2: addr) =
  dlseg' (a, n, l1, null, null, l2)

```

Figure 23: Dataviews for doubly-linked list segments



its front and back views are  $dlseg(T, I, L_1, \mathbf{0}, \mathbf{0}, L_2)$  and  $dlseg'(T, I, L_1, \mathbf{0}, \mathbf{0}, L_2)$ , respectively.

When programming with doubly-linked lists, we need various functions for performing view changes. For instance, a function of the following functional view turns a front view of doubly-linked list segment into a back view:

$$\forall \tau. \forall n : nat. \forall \lambda_1. \forall \lambda'_1. \forall \lambda_2. \forall \lambda'_2. dlseg(\tau, n, \lambda_1, \lambda'_1, \lambda_2, \lambda'_2) \multimap dlseg'(\tau, n, \lambda_1, \lambda'_1, \lambda_2, \lambda'_2)$$

As another example, a function of the following functional view combines the front views of two doubly-linked list segments into the front view of one doubly-linked list segment:

$$\forall \tau. \forall n_1 : nat. \forall n_2 : nat. \forall \lambda_1. \forall \lambda'_1. \forall \lambda_2. \forall \lambda'_2. \forall \lambda_3. \forall \lambda'_3. \\ (dlseg(\tau, n_1, \lambda_1, \lambda'_1, \lambda_2, \lambda'_2), dlseg(\tau, n_2, \lambda_2, \lambda'_2, \lambda_3, \lambda'_3)) \multimap dlseg(\tau, n_1 + n_2, \lambda_1, \lambda'_1, \lambda_3, \lambda'_3))$$

As this example is largely taken from (Reynolds 2002), it may be helpful for the reader to understand this example by contrasting it with the corresponding one in (Reynolds 2002).

### 3.4 Doubly-linked Binary Trees

Another form of commonly used data structure is doubly-linked binary trees, which, for instance, can be employed to implement red-black trees and splay trees. In Figure 24, we present two recursively defined view constructors  $binTreeRoot$  and  $binTreeInside$ .

Given a type  $T$ , an integer  $I$  and two addresses  $L_1$  and  $L_2$ ,  $binTreeRoot(T, I, L_1, L_2)$  is a view for a doubly-linked binary tree such that (1) each node in the tree contains an element of type  $T$ , (2) the number of nodes in the tree is  $I$ , (3) the root of the tree is at  $L_1$  and (4) the parent of the tree is at  $L_2$ . Note that the functional view assigned to the proof constructor  $BTRsome$  is formally written as follows:

$$\forall \tau. \forall n_1 : nat. \forall n_2 : nat. \forall \lambda_s. \forall \lambda_p. \forall \lambda_l. \forall \lambda_r. \\ ((a, \mathbf{ptr}(\lambda_p), \mathbf{ptr}(\lambda_l), \mathbf{ptr}(\lambda_r)) @ \lambda_s, binTreeRoot(\tau, n_1, \lambda_l, \lambda_s), binTreeRoot(\tau, n_2, \lambda_r, \lambda_s)) \multimap \\ binTreeRoot(\tau, n_1 + n_2 + 1, \lambda_s, \lambda_p)$$

which precisely captures the invariant that the parent of the children of every node in a doubly-linked binary tree is the node itself. In Figure 25, we use a picture to show that  $BTRsome$  turns the following three views:

$$(T, \mathbf{ptr}(L_1), \mathbf{ptr}(L_3), \mathbf{ptr}(L_4)) @ L_2, binTreeRoot(T, n_l, L_3, L_2), binTreeRoot(T, n_r, L_4, L_2)$$

into one view:  $binTreeRoot(T, n_l + n_r + 1, L_2, L_1)$ .

```

// root view for doubly-linked binary trees

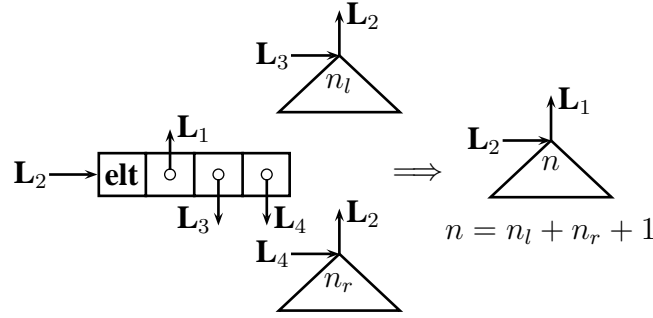
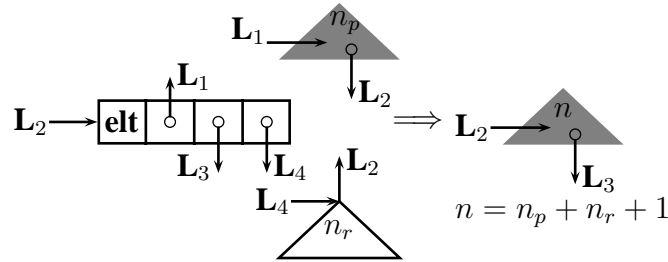
dataview binaryTreeRoot (type, int, addr, addr) =
  | {a:type, parent} BTRnone (a, 0, null, parent)
  | {a:type, n1:nat, n2:nat, self, parent, left, right |
    self <> null, left <> right}
    BTRsome (a, n1+n2+1, self, parent) of
      ((a, ptr parent, ptr left, ptr right) @ self,
        binaryTreeRoot (a, n1, left, self),
        binaryTreeRoot (a, n2, right, self))

// inside view for doubly-linked binary trees with
// one missing subtree

dataview binaryTreeInside (type, int, addr, addr) =
  | {a:type, child:addr} BTIempty (a, 0, null, child)
  | {a:type, n1:nat, n2:nat, self, parent, left, right |
    self <> null, left <> right}
    BTIleft (a, n1+n2+1, self, left) of
      ((a, ptr parent, ptr left, ptr right) @ self,
        binaryTreeInside (a, n1, parent, self),
        binaryTreeRoot (a, n2, right, self))
  | {a:type, n1:nat, n2:nat, self, parent, left, right |
    self <> null, left <> right}
    BTIright (a, n1+n2+1, self, right) of
      ((a, ptr parent, ptr left, ptr right) @ self,
        binaryTreeInside (a, n1, parent, self),
        binaryTreeRoot (a, n2, left, self))

```

Figure 24: Dataviews for doubly-linked binary trees


 Figure 25: A picture for *BTRsome*

 Figure 26: A picture for *BTleft*

Given a type  $T$ , an integer  $I$  and two addresses  $L_1$  and  $L_2$ ,  $\text{binTreeInside}(T, I, L_1, L_2)$  is a view for a doubly-linked binary tree with one missing subtree such that (1) each node in the tree contains an element of type  $T$ , (2) the number of nodes in the tree is  $I$ , (3) the parent of the root of the missing subtree is at  $L_1$ , and (4) the missing subtree itself is at  $L_2$ . We encounter such a view when traversing from the root of a doubly-linked binary tree to an inner node, and the proof constructor *BTleft* and *BTright* are used to indicate whether the left or right child of a node is visited. In Figure 26, we use a picture to show that *BTleft* turns the following three views:

$$(T, \text{ptr}(L_1), \text{ptr}(L_3), \text{ptr}(L_4)) @_{L_2}, \text{binTreeInside}(T, n_p, L_1, L_2), \text{binTreeRoot}(T, n_r, L_4, L_2)$$

into one view:  $\text{binTreeInside}(T, n_l + n_r + 1, L_2, L_3)$ . Given two views  $\text{binTreeInside}(T, I_1, L_1, L_2)$  and  $\text{binTreeRoot}(T, I_2, L_2, L_3)$ , we can combine them to form a view  $\text{binTreeRoot}(T, I_1 + I_2, L, L')$  for some addresses  $L, L'$ .

An actual implementation of splay trees using *binTreeInside* and *binTreeRoot* can be found on-line (Xi 2003), where splaying is done in a bottom-up fashion.

## 4 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 (Hoare 1969; Hoare 1971), 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 (Martin-Löf 1984; Nordström, Petersson, and Smith 1990), 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 (Xi 2000) and DTAL (Xi and Harper 2001) 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) (Morrisett, Walker, Crary, and Glew 1999) 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 (Sagiv, Reps, and Wilhelm 1998), 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 (Walker and Morrisett 2000) 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 circulist (a:type, n:int, l: addr) = slseg (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 buffer implementation based on circular lists (Xi 2003).

Along a related but different line of research, separation logic (Reynolds 2002) 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.

There is a large body of research on applying linear type theory based on linear logic (Girard 1987) to memory management (e.g. (Wadler 1990; Chirimar, Gunter, and Riecke 1996; Turner and Wadler 1999; Kobayashi 1999; Igarashi and Kobayashi 2000; Hofmann 2000)), and the work (Petersen, Harper, Crary, and Pfenning 2003) that attempts to give an account for data layout based on ordered linear logic (Polakow and Pfenning 1999) 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.

A new notion of types called guarded recursive (g.r.) datatypes has recently been introduced (Xi, Chen, and Chen 2003). 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 development of *ATS*. The formalization of *ATS/SV* largely follows the guidelines set in *ATS* for formalizing applied type systems.

There have been a large number of research activities on verifying program safety properties by tracking state changes. For instance, Cyclone (Jim, Morrisett, Grossman, Hicks, Baudet, Harris, and Wang 2001) allows the programmer to specify safe stack and region memory allocation; both CQual (Foster, Terauchi, and Aiken 2002) and Vault (Fahndrich and Deline 2002) support some form of resource usage protocol verification; ESC (Detlefs 1996) enables the programmer to state various sorts of program invariants and then employs theorem proving to prove them; CCured (Necula, McPeak, and Weimer 2002) uses program analysis to show the safety of mostly unannotated C programs. 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 (Mandelbaum, Walker, and Harper 2003), where the aim is to develop a general theory of type refinements for reasoning about program states. However, the notions such as recursive stateful views and view changes, which constitute the key contributions of this paper, have no counterparts there.

In summary, we have presented the design and formalization of a type system *ATS/SV* that make 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 the practicality of programming with stateful views. As for future research, our immediate plan is to incorporate exceptions into *ATS/SV*. We are currently keen to build the programming language *ATS* suitable for both high-level and low level

programming. With *ATS/SV*, we believe that a solid step towards reaching this grand goal is made.

**Acknowledgment** We acknowledge some discussions with Chiyang Chen on the subject of stateful views and thank him for providing us with valuable comments on a previous draft of the paper.

## References

- Chen, C., R. Shi, and H. Xi (2004, June). 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.
- Chen, C. and H. Xi (2003, August). Meta-Programming through Typeful Code Representation. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, Uppsala, Sweden, pp. 169–180.
- Chen, C., D. Zhu, and H. Xi (2004, June). implementing Cut Elimination: A Case Study of Simulating Dependent Types in Haskell. In *Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages*, Dallas, TX.
- Chirimar, J., C. A. Gunter, and G. Riecke (1996). Reference Counting as a Computational Interpretation of Linear Logic. *Journal of Functional Programming* 6(2), 195–244.
- Detlefs, D. (1996). An overview of the extended static checking system. In *Workshop on Formal Methods in Software Practice*.
- Fahndrich, M. and R. Deline (2002, June). Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Berlin, pp. 13–24.
- Foster, J., T. Terauchi, and A. Aiken (2002, June). Flow-sensitive Type Qualifiers. In *ACM Conference on Programming Language Design and Implementation*, Berlin, pp. 1–12.
- Girard, J.-Y. (1987). Linear logic. *Theoretical Computer Science* 50(1), 1–101.
- Hoare, C. A. R. (1969, October). An axiomatic basis for computer programming. *Communications of the ACM* 12(10), 576–580 and 583.
- Hoare, C. A. R. (1971). Procedures and parameters: An axiomatic approach. In E. Engeler (Ed.), *Symposium on Semantics of Algorithmic Languages*, Volume 188 of *Lecture Notes in Mathematics*, pp. 102–116. Berlin: Springer-Verlag.
- Hofmann, M. (2000, Winter). A type system for bounded space and functional in-place update. *Nordic Journal of Computing* 7(4), 258–289.
- Igarashi, A. and N. Kobayashi (2000, September). Garbage Collection Based on a Linear Type System. In *Preliminary Proceedings of the 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC '00)*.
- Jim, T., G. Morrisett, D. Grossman, M. Hicks, M. Baudet, M. Harris, and Y. Wang (2001). Cyclone, a Safe Dialect of C. Available at <http://www.cs.cornell.edu/Projects/cyclone/>.

- Kobayashi, N. (1999). Quasi-linear types. In *Proceedings of the 26th ACM Sigplan Symposium on Principles of Programming Languages (POPL '99)*, San Antonio, Texas, USA, pp. 29–42.
- Mandelbaum, Y., D. Walker, and R. Harper (2003, September). An effective theory of type refinements. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, Uppsala, Sweden, pp. 213–226.
- Martin-Löf, P. (1984). *Intuitionistic Type Theory*. Naples, Italy: Bibliopolis.
- Milner, R., M. Tofte, R. W. Harper, and D. MacQueen (1997). *The Definition of Standard ML (Revised)*. Cambridge, Massachusetts: MIT Press.
- Morrisett, G., D. Walker, K. Crary, and N. Glew (1999, May). From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems* 21(3), 527–568.
- Necula, G. C., S. McPeak, and W. Weimer (2002, January). CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, London, pp. 128–139.
- Nordström, B., K. Petersson, and J. M. Smith (1990). *Programming in Martin-Löf's Type Theory*, Volume 7 of *International Series of Monographs on Computer Science*. Oxford: Clarendon Press.
- Petersen, L., R. Harper, K. Crary, and F. Pfenning (2003, January). A Type Theory for Memory Allocation and Data Layout. In *Proceedings of the 30th ACM Sigplan Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, Louisiana, USA, pp. 172–184.
- Polakow, J. and F. Pfenning (1999, April). Relating natural deduction and sequent calculus for intuitionistic non-commutative linear logic. In A. Scedrov and A. Jung (Eds.), *Proceedings of the 15th Conference on Mathematical Foundations of Programming Semantics*, New Orleans, Louisiana. Electronic Notes in Theoretical Computer Science, Volume 20.
- Reynolds, J. (2002). Separation Logic: a logic for shared mutable data structures. In *Proceedings of 17th IEEE Symposium on Logic in Computer Science (LICS '02)*.
- Sagiv, M., T. Reps, and R. Wilhelm (1998, January). Solving shape-analysis problems in languages with destructive updating. *ACM Transactions of Programming Languages and Systems (TOPLAS)* 20(1), 1–50.
- Sleator, D. D. and R. E. Tarjan (1985). Self-adjusting Binary Search Trees. *Journal of the ACM* 32(3), 652–686.
- Turner, D. N. and P. Wadler (1999, October). Operational interpretations of linear logic. *Theoretical Computer Science* 227(1–2), 231–248.
- Wadler, P. (1990). Linear types can change the world. In *TC 2 Working Conference on Programming Concepts and Methods (Preprint)*, Sea of Galilee, pp. 546–566.
- Walker, D. and G. Morrisett (2000, September). Alias Types for Recursive Data Structures. In *Proceedings of International Workshop on Types in Compilation*, pp. 177–206. Springer-Verlag LNCS vol. 2071.

- Xi, H. (1998). *Dependent Types in Practical Programming*. Ph. D. thesis, Carnegie Mellon University. pp. viii+189. Available at <http://www.cs.cmu.edu/~hwxi/DML/thesis.ps>.
- Xi, H. (2000, June). Imperative Programming with Dependent Types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, Santo Barbara, CA, pp. 375–387.
- Xi, H. (2002, March). Dependent Types for Program Termination Verification. *Journal of Higher-Order and Symbolic Computation* 15(1), 91–132.
- Xi, H. (2003, July). Applied Type System. Available at: <http://www.cs.bu.edu/~hwxi/ATS>.
- Xi, H. (2004a). Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pp. 394–408. Springer-Verlag LNCS 3085.
- Xi, H. (2004b, March). Expressing Modular Structure through Conditional Type Equality. Available at <http://www.cs.bu.edu/~hwxi/academic/drafts/ModTyp.ps>.
- Xi, H., C. Chen, and G. Chen (2003, January). Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, New Orleans, LA, pp. 224–235.
- Xi, H. and R. Harper (2001, September). A Dependently Typed Assembly Language. In *Proceedings of International Conference on Functional Programming*, pp. 169–180.