

# A Modality for Safe Resource Sharing and Code Reentrancy <sup>★</sup>

Rui Shi<sup>1</sup>, Dengping Zhu<sup>2</sup>, and Hongwei Xi<sup>3</sup>

<sup>1</sup> Yahoo! Inc.

<sup>2</sup> Bloomberg Inc.

<sup>3</sup> Boston University

**Abstract.** The potential of linear logic in facilitating reasoning on resource usage has long been recognized. However, convincing uses of linear types in practical programming are still rather rare. In this paper, we present a general design to effectively support practical programming with linear types. In particular, we introduce and then formalize a modality, which we refer to as the sharing modality, in support of sharing of linear resources (with no use of locks). We develop the underlying type theory for the sharing modality and establish its soundness based on a notion of *types with effects*. We also point out an intimate relation between this modality and the issue of code reentrancy. In addition, we present realistic examples to illustrate the use of sharing modality, which are verified in the programming language ATS and thus provide a solid proof of concept.

## 1 Introduction

Although linear logic arose historically from domain theory [6], its potential in facilitating reasoning on resource usage has been recognized since the very beginning of its invention. For instance, Asperti showed an interesting way to describe Petri nets [1] in terms of linear logic formulas. In type theory, we have so far seen a large body of research on using linear types to facilitate memory management (e.g. [16, 4, 15, 10, 8, 7, 17]).

When programming with linear resources, we often need to *thread these resources through functions*. Suppose that we have a linear array  $A$  of type  $\mathbf{larray}(T)$ , where  $T$  is the type for the elements stored in  $A$ . In order to use linear arrays, we need the following access functions:

$$\begin{aligned} lsub &: \mathbf{larray}(T) \otimes \mathbf{int} \rightarrow \mathbf{larray}(T) \otimes T \\ lupdate &: \mathbf{larray}(T) \otimes \mathbf{int} \otimes T \rightarrow \mathbf{larray}(T) \otimes \mathbf{1} \end{aligned}$$

where we use  $\otimes$  for linear conjunction and  $\mathbf{1}$  for the unit type. Intuitively, the subscripting function on a *linear* array  $A$  needs to take  $A$  and an index  $i$  and then return both  $A$  and the content stored in the cell indexed by  $i$  (so that  $A$  is still

---

<sup>★</sup> This work is partially supported by NSF grants No. CCR-0229480 and No. CCF-0702665.

available for subsequent uses). The same holds for the update function *lupdate* as well. With linearity, it can be guaranteed that there is exactly one access path for each linear resource so that the state of the resource can be soundly reasoned about (e.g. linear arrays can be safely freed). However, the need for threading linear resources can often be burdensome or even impractical (in cases where a large number of resources are involved).

The subscripting function on a *sharable* array<sup>4</sup> as is supported in languages such as ML takes the array and an index and then returns only the content stored in the cell indexed by *i*. Intuitively, a sharable array is just a linear array that can be shared (in some restricted manner if necessary). However, we find it highly challenging to properly explain in a type theory how a sharable array can be implemented on top of a linear array. In most safe languages, arrays are treated as an abstract data structure. For instance, there is an abstract type constructor **array** in ML that takes a type *T* to form the type **array**(*T*) for sharable arrays in which the stored elements are of type *T*, and the following functions are provided for creating (and initializing as well), subscripting and updating arrays, respectively:

$$\begin{array}{ll} \mathbf{array} & : \mathbf{int} * T \rightarrow \mathbf{array}(T) \\ \mathbf{sub} & : \mathbf{array}(T) * \mathbf{int} \rightarrow T \\ \mathbf{update} & : \mathbf{array}(T) * \mathbf{int} * T \rightarrow \mathbf{1} \end{array}$$

However, the type constructor **array** cannot be defined in ML and the functions **array**, **sub** and **update** have to be implemented in other (unsafe) languages such as C or assembly and then *assumed* to possess the types assigned to them. Though simple and workable, this approach to supporting arrays in safe languages is evidently uninspiring and unsatisfactory when type theory is of the concern. Also, this approach makes it difficult, if not entirely impossible, to directly manipulate memory at low level, which is often indispensable in systems programming.

We can also see the need for sharable data structures from a different angle. As a crude approximation, let us assume that the memory allocation/deallocation functions *malloc* and *free* are assigned the following types:

$$\mathit{malloc} : \mathbf{int} \rightarrow \mathbf{larray}(\mathbf{top}) \quad \mathit{free} : \mathbf{larray}(\mathbf{top}) \rightarrow \mathbf{1}$$

where we use **top** for the top type, that is, every type is a subtype of **top**. Clearly, this type assignment for *malloc* and *free* relies on the assumption that the free list<sup>5</sup> used in implementing these functions is shared. Otherwise, *malloc* and *free* need to be assigned the following types:

$$\begin{array}{l} \mathit{malloc} : \mathbf{freelist} \otimes \mathbf{int} \rightarrow \mathbf{freelist} \otimes \mathbf{larray}(\mathbf{top}) \\ \mathit{free} : \mathbf{freelist} \otimes \mathbf{larray}(\mathbf{top}) \rightarrow \mathbf{freelist} \otimes \mathbf{1} \end{array}$$

where we use **freelist** as a type for the (linear) free list. This simply means that the free list must be threaded through every client program that makes use

<sup>4</sup> Note that a sharable array is a mutable data structure and it should not be confused with a functional array (e.g. based on a Braun tree).

<sup>5</sup> A free list is a data structure commonly used in implementing *malloc* and *free* for the purpose of maintaining a list of available memory blocks.

of either *malloc* or *free*, which makes it rather impractical to construct critical system libraries such as memory allocator in this manner.

Ideally, resources should be manipulated at low-level where linear types are used to reason about resource usage. While, in order to be practical, it is desirable to make linear resources sharable at some point as shown by the above motivating examples. Apparently, a naive treatment which simply turns linear resources to nonlinear ones without any restrictions does not work. If unrestricted access to shared resources were allowed, the presence of alias could easily break type soundness.

In this paper, we develop a type system where linear types are available for safe programming with resources. In order to support safe resource sharing, we introduce a modality that intuitively means *a shared resource of some linear type can be borrowed only if it is guaranteed that another resource of the same linear type is to be returned*. The primary contribution of the paper lies in the identification and then the formalization of this modality through a notion of *types with effects* [9], where some interesting as well as challenging technical issues are addressed. As an application, we demonstrate that various features for safe memory manipulation at low level (including memory allocation/initialization and pointer arithmetic) can be effectively supported in the type system we develop. We show, for example, safe implementations of sharable arrays based on primitive memory operations, and we believe that such implementations are done (as far as we know) for the first time in a programming language. In addition, we also point out an intimate relation between this modality and the issue of code reentrancy, providing a formal account for code reentrancy as well as a means that can prevent non-reentrant functions like *malloc* and *free* from being called reentrantly (e.g., in threads).

The type system we ultimately develop involves a long line of research on dependent types [21, 18], linear types [22], programming with theorem proving [3], and type theory for resource sharing. To facilitate understanding, we give a detailed presentation of a simple but rather abstract type system that supports resource sharing, and then outline extensions of this simple type system with advanced types and programming features. The interesting and realistic examples we show all involve dependent types and possibly polymorphic types. In addition, they all rely on the feature of programming with theorem proving.

We organize the rest of the paper as follows. In Section 2, we present a language  $\mathcal{L}_0$  with a simple linear type system, setting up some machinery for further development. We extend  $\mathcal{L}_0$  to  $\mathcal{L}_\square$  with a modality in Section 3 to address the issue of resource sharing in programming. Furthermore, we extend  $\mathcal{L}_\square$  to  $\mathcal{L}_\square^{\forall, \exists}$  in Section 4 by incorporating universally as well as existentially quantified types, preparing to support direct memory manipulation at low level. In Section 5, we demonstrate through several interesting and realistic examples that the developed type theory can be effectively put into practice. In Section 6, we discuss how the code reentrancy problem is addressed in the presence of concurrency. Lastly, we mention some related work and conclude. In addition, we list the complete typing rules and more code examples in the appendix to facilitate assessment.

---

types	$T ::= \delta^i \mid T_1 * T_2 \mid VT_1 \rightarrow_i VT_2 \mid$
viewtypes	$VT ::= \delta^l \mid T \mid VT_1 \otimes VT_2 \mid VT_1 \rightarrow_l VT_2$
expressions	$e ::= c(e_1, \dots, e_n) \mid r \mid xf \mid \mathbf{if}(e_1, e_2, e_3) \mid \mathbf{fst}(e) \mid$ $\mathbf{snd}(e) \mid \langle e_1, e_2 \rangle \mid \mathbf{lam} x. e \mid e_1(e_2) \mid \mathbf{fix} f. v \mid$ $\mathbf{let} \langle x_1, x_2 \rangle = e_1 \mathbf{in} e_2 \mathbf{end}$
values	$v ::= cc(\bar{v}) \mid r \mid x \mid \langle v_1, v_2 \rangle \mid \mathbf{lam} x. e$
intuitionistic. exp. ctx.	$\Gamma ::= \emptyset \mid \Gamma, xf : T$
linear. exp. ctx.	$\Delta ::= \emptyset \mid \Delta, x : VT$

---

**Fig. 1.** The syntax of  $\mathcal{L}_0$

## 2 The Starting Point: $\mathcal{L}_0$

We first present a language  $\mathcal{L}_0$  with a simple linear type system, using it as a starting point to set up some machinery for further development. We do not address the issue of resource sharing in  $\mathcal{L}_0$ , which is to be done at the next stage. The syntax of the language  $\mathcal{L}_0$  is given in Figure 1.

We use  $c$  for constants, which include both constant functions  $cf$  and constant constructors  $cc$ , and  $r$  for resources. Note that we treat the resources in  $\mathcal{L}_0$  abstractly. For instance, when dealing with memory manipulation at low level, we introduce resources of the form  $v@L$ , where  $v$  and  $L$  range over values and memory addresses (represented as natural numbers), respectively. Intuitively,  $v@L$  means that the value  $v$  is stored at the address  $L$ . For a simple and clean presentation, we assume in this paper that values are properly boxed and can thus be stored in one memory unit. In practice, we can and do handle unboxed values without much complication.

We use  $T$  and  $VT$  for types and viewtypes, respectively, and  $\delta^i$  and  $\delta^l$  for base types and base viewtypes respectively, where the superscript  $i$  means intuitionistic while  $l$  means linear. For instance, **bool** and **int** are base types for booleans and integers while **int@L** is a base viewtype for the resource  $v@L$  given  $v$  is of type **int**.

Note that a type is always considered a viewtype. At this point, we emphasize that  $\rightarrow_l$  should not be confused with the linear implication  $\multimap$  in linear logic. Given  $VT_1 \rightarrow_l VT_2$ , the viewtype constructor  $\rightarrow_l$  simply indicates that  $VT_1 \rightarrow_l VT_2$  itself is a viewtype. The meaning of various forms of types and viewtypes is to be made clear and precise when the rules are presented for assigning viewtypes to expressions in  $\mathcal{L}_0$ .

We assume the existence of a signature SIG that assigns each resource  $r$  a base viewtype  $\delta^l$  and each constant  $c$  a constant type (c-type, for short) of the form  $(VT_1, \dots, VT_n) \Rightarrow VT$  ( $VT$  must be either  $\delta^i$  or  $\delta^l$  if  $c$  is a constructor), where  $n$  is the arity of  $c$ . For instance, the truth values *true* and *false* are assigned the c-type  $() \Rightarrow \mathbf{bool}$ .

The expressions  $e$  and values  $v$  in  $\mathcal{L}_0$  are mostly standard. We use  $x$  for lam-variables and  $f$  for fix-variables, where the former is a value but the latter is not,

and write  $xf$  for either  $x$  or  $f$ . We may write  $\bar{v}$  for a (possibly empty) sequence of values.

We use  $R$  for finite *multisets* of resources. Given  $R_1$  and  $R_2$ , we write  $R_1 \uplus R_2$  for the *multiset union* of  $R_1$  and  $R_2$ . Given an expression  $e$ , we use  $\rho(e)$  for the multiset of resources contained in  $e$ , which is defined as follows:

$$\begin{array}{ll} \rho(c(e_1, \dots, e_n)) &= \rho(e_1) \uplus \dots \uplus \rho(e_n) & \rho(r) &= \{r\} \\ \rho(x) &= \emptyset & \rho(\mathbf{if}(e_1, e_2, e_3)) &= \rho(e_1) \uplus \rho(e_2) \\ \rho(\mathbf{let} \langle x_1, x_2 \rangle = e_1 \mathbf{in} e_2 \mathbf{end}) &= \rho(e_1) \uplus \rho(e_2) & \rho(\mathbf{lam} x. e) &= \rho(e) \\ \rho(e_1(e_2)) &= \rho(e_1) \uplus \rho(e_2) & \dots &= \dots \end{array}$$

In the case where  $e = \mathbf{if}(e_1, e_2, e_3)$ , the type system of  $\mathcal{L}_0$  is to enforce that  $\rho(e_2) = \rho(e_3)$  if  $e$  can be assigned a viewtype, and this is the reason for defining  $\rho(\mathbf{if}(e_1, e_2, e_3))$  as  $\rho(e_1) \uplus \rho(e_2)$ .

We emphasize that resources are *not* necessarily preserved under evaluation. It is possible for an expression containing resources to be assigned a type or an expression containing no resources to be assigned a viewtype. For instance, suppose that *alloc* is a constant function that takes a natural number as its argument and returns some resources. Then the expression *alloc*(1) contains no resources but it cannot be assigned a type (as the evaluation of *alloc*(1) returns a value containing a resource  $v@L$  for some value  $v$  and address  $L$ ).

It is clear that we cannot combine resources arbitrarily. For instance, it is impossible to have resources  $v_1@L$  and  $v_2@L$ , simultaneously. We define **ST** as a collection of finite multisets of resources and assume that  $\emptyset \in \mathbf{ST}$  and **ST** is closed under subset relation, that is, for any  $R_1$  and  $R_2$ ,  $R_2 \in \mathbf{ST}$  if  $R_1 \in \mathbf{ST}$  and  $R_2 \subseteq R_1$ , where  $\subseteq$  is the subset relation on multisets. We say that  $R$  is a valid multiset of resources if  $R \in \mathbf{ST}$  holds. Note that the definition of **ST** is considered abstract, which is not specific to the language. For instance, if the type system is used for reasoning about memory manipulation, **ST** can be defined as the collection of all valid memory states.

**Dynamic Semantics** The definition of evaluation contexts  $E$  in  $\mathcal{L}_0$  is given as follows:

$$\begin{aligned} E ::= & \square \mid cc(v_1, \dots, v_{i-1}, E, e_{i+1}, \dots, e_n) \mid \mathbf{if}(E, e_1, e_2) \mid \langle E, e \rangle \mid \langle v, E \rangle \mid \\ & \mathbf{fst}(E) \mid \mathbf{snd}(E) \mid \mathbf{let} \langle x_1, x_2 \rangle = E \mathbf{in} e \mathbf{end} \mid E(e) \mid v(E) \end{aligned}$$

We are to use evaluation contexts to define the (call-by-value) dynamic semantics of  $\mathcal{L}_0$ . There are two forms of redexes in  $\mathcal{L}_0$ : pure redexes and *ad hoc* redexes. The pure redexes and their reducts are defined as follows:

- $\mathbf{if}(true, e_1, e_2)$  is a pure redex, and its reduct is  $e_1$ .
- $\mathbf{if}(false, e_1, e_2)$  is a pure redex, and its reduct is  $e_2$ .
- $\mathbf{let} \langle x_1, x_2 \rangle = \langle v_1, v_2 \rangle \mathbf{in} e \mathbf{end}$  is a pure redex, and its reduct is  $e[x_1, x_2 \mapsto v_1, v_2]$ .
- $\mathbf{fst}(\langle v_1, v_2 \rangle)$  is a pure redex, and its reduct is  $v_1$ .
- $\mathbf{snd}(\langle v_1, v_2 \rangle)$  is a pure redex, and its reduct is  $v_2$ .
- $(\mathbf{lam} x. e)(v)$  is a pure redex, and its reduct is  $e[x \mapsto v]$ .
- $\mathbf{fix} f. v$  is a pure redex, and its reduct is  $v[f \mapsto \mathbf{fix} f. v]$ .

Evaluating calls to constant functions is of particular importance in  $\mathcal{L}_0$  as it may involve resource generation and consumption. Assume that *cf* is a constant function of arity  $n$ . The expression  $cf(v_1, \dots, v_n)$  is an ad hoc redex if

$cf$  is defined at  $v_1, \dots, v_n$ , and any value  $v$  of  $cf(v_1, \dots, v_n)$  is a reduct of  $cf(v_1, \dots, v_n)$ . For instance,  $alloc$  is a function that takes a natural number  $n$  to return a pointer to some address  $L$  associated with a tuple of resources  $\langle v_0@L, v_1@L + 1, \dots, v_{n-1}@L + n - 1 \rangle$  for some values  $v_0, v_1, \dots, v_{n-1}$ , that is,  $alloc(n)$  reduces to a pointer that points to  $n$  consecutive memory units containing some unspecified values.

**Definition 1.** *Given expressions  $e_1$  and  $e_2$ , we write  $e_1 \hookrightarrow e_2$  if  $\rho(e_1) \in \mathbf{ST}$  holds,  $e_1 = E[e]$  and  $e_2 = E[e']$  for some evaluation context  $E$  and expressions  $e, e'$  such that  $e$  is a redex,  $e'$  is a reduct of  $e$  and  $\rho(E[e']) \in \mathbf{ST}$  holds.*

As usual, we use  $\hookrightarrow^*$  for the reflexive and transitive closure of  $\hookrightarrow$ . We say that  $e$  reduces to  $e'$  purely if the redex being reduced is pure. A type system is to be developed to guarantee that resources are preserved under pure reduction, that is,  $\rho(e) = \rho(e')$  whenever  $e$  reduces to  $e'$  purely. However, resources may be generated as well as consumed when ad hoc reduction occurs. Suppose that  $e_1 = E[alloc(1)]$  and  $v@L$  occurs in  $E$ . Though  $\langle v@L, L \rangle$  is a reduct of  $alloc$ , we cannot allow  $e_1 \hookrightarrow E[\langle v@L, L \rangle]$  as the resource  $v@L$  occurs repeatedly in  $E[\langle v@L, L \rangle]$ . This is precisely the reason that we require  $\rho(e_2) \in \mathbf{ST}$  whenever  $e_1 \hookrightarrow e_2$  holds.

**Static Semantics** An intuitionistic expression context  $\Gamma$  can be treated as a finite mapping that maps  $xf$  to  $T$  for each declaration  $xf : T$  in  $\Gamma$ , and we use  $\mathbf{dom}(\Gamma)$  for the domain of  $\Gamma$ . A linear expression context  $\Delta$  can be treated in the same manner. Given an intuitionistic expression context  $\Gamma$  and a linear expression context  $\Delta$  such that  $\mathbf{dom}(\Gamma) \cap \mathbf{dom}(\Delta) = \emptyset$ , we can form an expression context  $(\Gamma; \Delta)$ . Clearly, we can also treat expression contexts as finite mappings. Given  $\Gamma$  and  $\Delta$ , we use  $(\Gamma; \Delta), x : VT$  for either  $(\Gamma; \Delta, x : VT)$  or  $(\Gamma, x : VT; \Delta)$  (if  $VT$  is actually a type).

We use  $\Gamma; \Delta \vdash e : VT$  for a judgment stating that the viewtype  $VT$  can be assigned to  $e$  under  $(\Gamma; \Delta)$ . The rules for assigning viewtypes to expressions in  $\mathcal{L}_0$  are largely standard thus omitted. We provide some explanation for the following two:

$$\frac{\Gamma; \Delta, x : VT_1 \vdash e : VT_2}{\Gamma; \Delta \vdash \mathbf{lam} x. e : VT_1 \rightarrow_l VT_2} \text{ (ty-}\rightarrow_l\text{-intr)} \quad \frac{\Gamma; \emptyset, x : VT_1 \vdash e : VT_2 \quad \rho(e) = \emptyset}{\Gamma; \emptyset \vdash \mathbf{lam} x. e : VT_1 \rightarrow_i VT_2} \text{ (ty-}\rightarrow_i\text{-intr)}$$

Given two viewtypes  $VT_1$  and  $VT_2$ ,  $VT_1 \rightarrow_l VT_2$  is a viewtype and  $VT_1 \rightarrow_i VT_2$  is a type. The rules **(ty- $\rightarrow_l$ -intr)** and **(ty- $\rightarrow_i$ -intr)** assign a viewtype and type to a function, respectively; the function can use its argument many times if  $VT_1$  is a type or exactly once if  $VT_1$  is viewtype. Intuitively, when the rule **(ty- $\rightarrow_i$ -intr)** is applied, the body of the involved function must contain no resources as the function is a value to which a type (not just a viewtype) is assigned.

**Soundness** As usual, the soundness of the type system of  $\mathcal{L}_0$  rests on the following two theorems. The detailed proof can be found in [13].

**Theorem 1 (Subject Reduction).** *Assume that  $\emptyset; \emptyset \vdash e : VT$  is derivable,  $\rho(e) \in \mathbf{ST}$  and  $e \hookrightarrow e'$ . Then  $\emptyset; \emptyset \vdash e' : VT$  is also derivable and  $\rho(e') \in \mathbf{ST}$ .*

**Theorem 2 (Progress).** *Assume that  $\emptyset; \emptyset \vdash e : VT$  is derivable and  $\rho(e) \in \mathbf{ST}$ . Then either  $e$  is a value or  $e \mapsto e'$  holds for some expression  $e'$ .*

### 3 Supporting Resource Sharing: $\mathcal{L}_\square$

The need for resource sharing occurs immediately in practice. As mentioned in Section 1, we must employ some form of resource sharing when implementing sharable data structures. We introduce a form of modality  $\square$ , which we refer to as *sharing modality*, to support resource sharing. We first give some intuitive but rather informal explanation about the expected use of  $\square$ . Given a value  $v$  of viewtype  $VT$ , we can imagine that a box is created to store the value  $v$ . We use  $h$  for the handle of the box, which is assigned the type  $\square VT$  and can thus be duplicated. The unary type constructor  $\square$ , which takes a viewtype to form a type, imposes the following requirement on a program that attempts to access the value stored in the box through the handle  $h$  of a box: the program can take out the value in the box and manipulate it freely as long as it guarantees to return to the box a (possibly different) value of the same viewtype at the end of its evaluation.

With the sharing modality, there is an interesting but troubling problem of *double borrow* that must be properly addressed. Suppose that we are at a point where the value stored in a box has already been borrowed out but no value has been returned to the box yet. At this point, if there is another request to borrow from the box, then a scenario of double borrow occurs, which makes the following misbehaved program possible:

```

let  $\square r_1 = x$  in (* let  $\square$  is the syntax to borrow resource from a boxed value *)
  let  $\square r_2 = x$  in
    ... free( $r_1$ )... in ... access( $r_2$ ) end end end

```

where the resource boxed in  $x$  is double borrowed and bound to  $r_1$  and  $r_2$ , thus, both referring exactly the same resource. Furthermore, the program subsequently frees  $r_1$  before accessing  $r_2$ , which is clearly a safety violation.

In order to establish the soundness of a type system accommodating the sharing modality, we must prevent double borrow from ever happening. We achieve this by employing a notion of *types with effects* [9]. Specifically, we decorate the viewtype constructor  $\rightarrow_l$  with a bit  $b$  ranging over 0 and 1. Given a function of viewtype  $VT_1 \xrightarrow{b}_l VT_2$ , the evaluation of a call to this function is guaranteed to borrow no values from any boxes if  $b = 0$ . Otherwise, it may borrow values from some boxes. We decorate the type constructor  $\rightarrow_i$  with a bit in precisely the same manner.

The language  $\mathcal{L}_0$  is extended to  $\mathcal{L}_\square$  with some additional syntax in Figure 2. We use  $h$  for handles (of boxes) and assume that there exist infinitely many of them that can be generated freshly. Given an expression **let**  $\square x = e_1$  **in**  $e_2$  **end**, we expect that  $e_1$  evaluates to a handle  $h$ , and then  $x$  is bound to the value stored in the box with the handle  $h$  and  $e_2$  evaluates to a pair  $\langle v_1, v_2 \rangle$ , and then  $v_1$  is inserted into the box and  $v_2$  is the value of the expression **let**  $\square x = e_1$  **in**  $e_2$  **end**.

---

types	$T ::= \dots$		$\square VT$		$VT_1 \xrightarrow{\delta_l} VT_2$		$VT_1 \xrightarrow{\delta_i} VT_2$
expressions	$e ::= \dots$		$h$		$\square e$		<b>let</b> $\square x = e_1$ <b>in</b> $e_2$ <b>end</b>
values	$v ::= \dots$		$h$				
eval. ctx.	$E ::= \dots$		$\square E$		<b>let</b> $\square x = E$ <b>in</b> $e$ <b>end</b>		

---

**Fig. 2.** The additional syntax for  $\mathcal{L}_\square$

We are to use the type system of  $\mathcal{L}_\square$  to guarantee that the evaluation of  $e_2$  does not borrow values from any boxes.

We use  $\mathcal{M}$  for stores, which are finite mappings from handles to values or a special symbol  $\bullet$ . Given a store  $\mathcal{M}$  and a handle  $h$  in the domain  $\mathbf{dom}(\mathcal{M})$  of  $\mathcal{M}$ , we say that  $\mathcal{M}$  is *available* at  $h$  if and only if  $\mathcal{M}(h) = v$  for some value  $v$ . We say that  $\mathcal{M}$  is full if  $\mathcal{M}$  is available at each  $h \in \mathbf{dom}(\mathcal{M})$ . In the following presentation, we only deal with stores that are either full or not available at only one handle. We use  $\mathcal{M}[h \mapsto v]$  for the mapping that extends  $\mathcal{M}$  with an extra link from  $h$  to  $v$ , where  $h \notin \mathbf{dom}(\mathcal{M})$  is assumed. In addition, we use  $\mathcal{M}[h := v^*]$  for the mapping  $\mathcal{M}'$  such that  $\mathcal{M}'(h) = v^*$  and  $\mathcal{M}'(h') = \mathcal{M}(h')$  for each  $h' \in \mathbf{dom}(\mathcal{M}) = \mathbf{dom}(\mathcal{M}')$  that is not  $h$ , where  $v^*$  ranges over values and the special symbol  $\bullet$ . We say that  $\mathcal{M}'$  extends  $\mathcal{M}$  if  $\mathcal{M}'(h) = \mathcal{M}(h)$  for each  $h \in \mathbf{dom}(\mathcal{M}) \subseteq \mathbf{dom}(\mathcal{M}')$ .

We extend the definition of  $\rho$  to deal with the new syntax:  $\rho(h) = \emptyset$ ,  $\rho(\square e) = \rho(e)$ ,  $\rho(\mathbf{let} \ \square x = e_1 \ \mathbf{in} \ e_2 \ \mathbf{end}) = \rho(e_1) \uplus \rho(e_2)$ ,  $\rho(\bullet) = \emptyset$  and  $\rho(\mathcal{M}) = \uplus_{h \in \mathbf{dom}(\mathcal{M})} \rho(\mathcal{M}(h))$ .

We use  $\hat{e}$  for *intermediate expressions*, which are either closed expressions  $e$  or triples of the form  $\langle E, h, e \rangle$ , where  $E, h, e$  range over evaluation contexts, handles and closed expressions, respectively. We define  $\rho(\hat{e})$  to be  $\rho(e)$  if  $\hat{e} = e$  or  $\rho(E[e])$  if  $\hat{e} = \langle E, h, e \rangle$ . In  $\mathcal{L}_\square$ , the evaluation relation  $\hookrightarrow$  is defined on pairs of the form  $\langle \mathcal{M}, \hat{e} \rangle$ .

We say that  $\mathcal{M}$  matches  $\hat{e}$  if either  $\hat{e} = e$  for some  $e$  and  $\mathcal{M}$  is full or  $\hat{e} = \langle E, h, e \rangle$  for some  $E, h, e$  and  $\mathcal{M}$  is not available only at  $h$ .

**Definition 2.** (*Reduction in  $\mathcal{L}_\square$* ) We say that  $(\mathcal{M}, \hat{e})$  reduces to  $(\mathcal{M}', \hat{e}')$  if  $(\mathcal{M}, \hat{e}) \hookrightarrow (\mathcal{M}', \hat{e}')$ , which is defined as follows:

- If  $e$  reduces to  $e'$  and  $\rho(\mathcal{M}) \uplus \rho(e') \in \mathbf{ST}$ , then  $(\mathcal{M}, e) \hookrightarrow (\mathcal{M}, e')$ .
- If  $e$  reduces to  $e'$  and  $\rho(\mathcal{M}) \uplus \rho(E[e']) \in \mathbf{ST}$ , then  $(\mathcal{M}, \langle E, h, e \rangle) \hookrightarrow (\mathcal{M}, \langle E, h, e' \rangle)$ .
- If  $h \notin \mathbf{dom}(\mathcal{M})$ , then  $(\mathcal{M}, E[\square v]) \hookrightarrow (\mathcal{M}[h \mapsto v], E[h])$ .
- If  $\mathcal{M}(h) = v$ , then  $(\mathcal{M}, E[\mathbf{let} \ \square x = h \ \mathbf{in} \ e \ \mathbf{end}]) \hookrightarrow (\mathcal{M}[h := \bullet], \langle E, h, e[x \mapsto v] \rangle)$ .
- If  $\mathcal{M}(h) = \bullet$ , then  $(\mathcal{M}, \langle E, h, \langle v_1, v_2 \rangle \rangle) \hookrightarrow (\mathcal{M}[h := v_1], E[v_2])$ .

It is clear that an intermediate expression of the form  $\langle E, h, e[x \mapsto v] \rangle$  is generated when we evaluate  $(\mathcal{M}, E[\mathbf{let} \ \square x = h \ \mathbf{in} \ e \ \mathbf{end}])$ , where  $v$  is the value stored in the box with the handle  $h$ ; we are disallowed to borrow values from any boxes when evaluating  $e[x \mapsto v]$ ; the expression  $e[x \mapsto v]$  is expected to evaluate to a pair  $\langle v_1, v_2 \rangle$ , allowing  $v_1$  to be inserted into the box with the handle  $h$  and the evaluation of  $(\mathcal{M}[h := v_1], E[v_2])$  to start.

It may be argued that the restriction is too severe that disallows borrowing values from any boxes during the evaluation of  $\langle E, h, e[x \mapsto v] \rangle$  as it clearly suffices to only disallow borrowing values from the box with the handle  $h$ . However, it is highly nontrivial to use the type of an expression to indicate from which boxes values can or cannot be borrowed during the evaluation of the expression. Also, it is unclear whether there are strong practical reasons to do so. For instance, we can always extract values from two shared resources in sequence (and manipulate the values thereafter) instead of accessing them at the same time. In essence, the shared resources constructed through the modality can be used exactly the same way as ones available in ML such as references and arrays.

**Static Semantics** We use  $\mu$  for store types, which are finite mappings from handles to viewtypes. Also, we write  $\mu[h \mapsto VT]$  for the mapping that extends  $\mu$  with an extra link from  $h$  to  $VT$ , where  $h \notin \mathbf{dom}(\mu)$  is assumed. We say that  $\mu'$  extends  $\mu$  if  $\mu'(h) = \mu(h)$  for each  $h \in \mathbf{dom}(\mu) \subseteq \mathbf{dom}(\mu')$ .

We use  $\Gamma; \Delta \vdash_{\mu}^b e : VT$  for judgments assigning viewtypes to expressions, where the bit  $b$  ranges over 0 and 1. Given two bits  $b_1$  and  $b_2$ , the bit  $b_1 \oplus b_2$  is 0 if and only if  $b_1 = b_2 = 0$ . In other words,  $\oplus$  is the OR function on bits. The rule of most interest is

$$\frac{\Gamma; \Delta_1 \vdash_{\mu}^b e_1 : \square VT_1 \quad \Gamma; \Delta_2, x : VT_1 \vdash_{\mu}^0 e_2 : VT_1 \otimes VT_2}{\Gamma; \Delta_1 \uplus \Delta_2 \vdash_{\mu}^1 \mathbf{let} \square x = e_1 \mathbf{in} e_2 \mathbf{end} : VT_2} \quad (\mathbf{ty}\text{-}\square\text{-elim})$$

where the second premise states that  $e_2$  must be *borrow-free* (with the bit 0) while the final judgement is with bit 1, indicating some resource is borrowed. The complete rules for assigning viewtypes to expressions in  $\mathcal{L}_{\square}$  are omitted for brevity. We write  $\vdash \mathcal{M} : \mu$  to mean that for each  $h \in \mathbf{dom}(\mathcal{M}) = \mathbf{dom}(\mu)$ , either  $\mathcal{M}(h) = \bullet$  or  $\emptyset; \emptyset \vdash_{\mu}^0 \mathcal{M}(h) : \mu(h)$  is derivable.

**Soundness** As usual, the soundness of  $\mathcal{L}_{\square}$  rests on the following theorems. The detailed proof can be found in [13].

**Theorem 3 (Subject Reduction).** *Assume that  $\vdash \mathcal{M} : \mu$  holds and  $\vdash_{\mu}^b \hat{e} : VT$  is derivable. If  $(\mathcal{M}, \hat{e}) \hookrightarrow (\mathcal{M}', \hat{e}')$ , then there exists  $\mu'$  extending  $\mu$  such that  $\vdash \mathcal{M}' : \mu'$ ,  $\rho(\mathcal{M}') \uplus \rho(\hat{e}') \in \mathbf{ST}$  and  $\vdash_{\mu'}^{b'} \hat{e}' : VT$  is derivable for some  $b' \leq b$ .*

**Theorem 4 (Progress).** *Assume that  $\vdash \mathcal{M} : \mu$  holds,  $\mathcal{M}$  matches  $\hat{e}$ ,  $\rho(\mathcal{M}) \uplus \rho(\hat{e}) \in \mathbf{ST}$  and  $\vdash_{\mu}^b \hat{e} : VT$  is derivable. Then  $(\mathcal{M}, \hat{e}) \hookrightarrow (\mathcal{M}', \hat{e}')$  for some  $\mathcal{M}'$  and  $\hat{e}'$ .*

## 4 Extensions

While the basic design for supporting safe resource sharing in programming is already demonstrated in  $\mathcal{L}_{\square}$ , it is nonetheless difficult to truly reap the benefits of this design given that the type system of  $\mathcal{L}_{\square}$  is simply too limited. We need two extensions when presenting some interesting and realistic examples in Section 5.

---

sorts	$\sigma ::= \text{bool} \mid \text{int} \mid \text{addr} \mid \text{type} \mid \text{viewtype}$
types	$T ::= \dots \mid a \mid B \supset T \mid \forall a : \sigma. T \mid B \wedge T \mid \exists a : \sigma. T$
viewtypes	$VT ::= \dots \mid a \mid B \supset VT \mid \forall a : \sigma. VT \mid B \wedge VT \mid \exists a : \sigma. VT$
expressions	$e ::= \dots \mid \supset^+(v) \mid \supset^-(e) \mid \forall^+(v) \mid \forall^-(e) \mid \wedge(e) \mid \mathbf{let} \ \wedge(x) = e_1 \ \mathbf{in} \ e_2 \ \mathbf{end} \mid \exists(e) \mid \mathbf{let} \ \exists(x) = e_1 \ \mathbf{in} \ e_2 \ \mathbf{end}$
values	$v ::= \dots \mid \supset^+(v) \mid \forall^+(v) \mid \wedge(v) \mid \exists(v)$

---

**Fig. 3.** The additional syntax of  $\mathcal{L}_{\square}^{\forall, \exists}$

We first extend  $\mathcal{L}_{\square}$  to  $\mathcal{L}_{\square}^{\forall, \exists}$  by incorporating universally as well as existentially quantified viewtypes, which include both polymorphic viewtypes and dependent viewtypes. The extra syntax of  $\mathcal{L}_{\square}^{\forall, \exists}$  (over that of  $\mathcal{L}_{\square}$ ) is given in Figure 3. Following the work on the framework Applied Type System (ATS) [19, 20], this extension is largely standard.

In order to effectively deal with memory manipulation at low level, we also need to support a paradigm that combines programming with theorem proving. We introduce a language  $\mathcal{L}_{\square, pf}^{\forall, \exists}$  (detailed syntax are omitted for brevity) by extending  $\mathcal{L}_{\square}^{\forall, \exists}$  with a component in which only pure and total programs can be constructed, and this component is referred to as the theorem-proving component of  $\mathcal{L}_{\square, pf}^{\forall, \exists}$ . Proofs (pure and total dynamic terms) can be constructed in the theorem-proving component to attest various properties of programs (effectful and nonterminating dynamic terms). Due to the space constraint, we cannot give a detailed account of this paradigm in this paper. Instead, we refer the interested reader to [3] for theoretical details and practical examples. Note that the proofs in  $\mathcal{L}_{\square, pf}^{\forall, \exists}$  are only needed for type-checking purpose and they are completely erased before run-time execution.

## 5 Examples

In this section, we present several examples taken from the programming language ATS [20] to give the reader some concrete feel as to how the developed type theory for resource sharing can be put into practice. All examples are presented in the concrete syntax of ATS, which is inspired by the syntax of Standard ML [11]. A (partial) type inference process [2] is used to elaborate programs written in the concrete syntax into the formal syntax of  $\mathcal{L}_{\square, pf}^{\forall, \exists}$ .

We assume the existence of primitive memory access functions `ptr_get` and `ptr_set` of the following types:

$$\begin{aligned} \text{ptr\_get} & : \forall \tau. \forall \lambda. (\tau @ \lambda, \text{ptr}(\lambda)) \Rightarrow (\tau @ \lambda) \otimes \tau \\ \text{ptr\_set} & : \forall \tau. \forall \lambda. (\mathbf{top} @ \lambda, \text{ptr}(\lambda), \tau) \Rightarrow (\tau @ \lambda) \otimes \mathbf{1} \end{aligned}$$

where we use  $\tau$  and  $\lambda$  for bounded static variables of sorts *type* and *addr*, respectively, and **top** for the type such that every type is a subtype of **top**. Basically, `ptr_get` reads through a pointer and `ptr_set` writes through a pointer. Given

---

```

fun ref_make {a:type} (x: a): ref a = let
  val (pf | p) = alloc (1) // pf: array_v (top, 1, 1) and p: ptr 1
  // The pattern matching is verified to be exhaustive
  prval ArraySome (pf1, ArrayNone ()) = pf // pf1: top @ 1
  val (pf1 | ()) = ptr_set (pf1 | p, x) // pf1: a @ 1
in (vbox pf1 | p) end

fun ref_get {a:type} (r: ref a): a = let
  val (h | p) = r // h: vbox(a @ 1) for some 1
  prval vbox pf = h // (pf : a @1) is borrowed
in ptr_get (pf | p) end

fun ref_set {a:type} (r: ref a, x: a): void = let
  val (h | p) = r // h: vbox(a @ 1) for some 1
  prval vbox pf = h // (pf: a @ 1) is borrowed
in ptr_set {a} (pf | p, x) end

```

---

**Fig. 4.** An implementation of references

$T$  and  $L$ , `ptr_get` takes a proof of  $T@L$  and a pointer to  $L$  and it returns a proof of  $T@L$  and a value of type  $T$ . So a linear proof of  $T@L$  threads through `ptr_get`. On the other hand, given  $T$  and  $L$ , `ptr_set` takes a proof of  $\mathbf{top}@L$ , a value of type  $T$  and a pointer to  $L$  and it returns a proof of  $T@L$  and the unit. So a proof of  $\mathbf{top}@L$  is consumed and then a proof of  $T@L$  is generated by `ptr_set`.

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

$$\begin{aligned}
\text{ArrayNone} &: \forall \lambda. \forall \tau. () \Rightarrow \text{array\_v}(\tau, 0, \lambda) \\
\text{ArraySome} &: \forall \lambda. \forall \tau. \forall \iota. \iota \geq 0 \supset (\tau @ \lambda, \text{array\_v}(\tau, \iota, \lambda + 1)) \Rightarrow \text{array\_v}(\tau, \iota + 1, \lambda)
\end{aligned}$$

Intuitively, `ArrayNone` is a proof of `array_v( $T, 0, L$ )` for any type  $T$  and address  $L$ , and `ArraySome( $v_1, v_2$ )` is a proof of `array_v( $T, I + 1, L$ )` for any type  $T$ , integer  $I$  and address  $L$  if  $v_1$  and  $v_2$  are proofs of views  $T@L$  and `array_v( $T, I, L + 1$ )`, respectively.

**References** In Figure 4, we give an implementation of references (as commonly supported in ML) in ATS. The symbol `|` in the code is a separator (just like a comma), which separates proofs from values (so as to make the code easier to read).

Given a type  $T$ , we use  $\mathbf{ref}(T)$  for the type of references to values of type  $T$ , which is formally defined as  $\exists \lambda. \square.(T @ \lambda) * \mathbf{ptr}(\lambda)$ . Therefore, a reference is just a pointer to some address  $L$  paired with the handle of a box containing a proof stating that a value of type  $T$  is stored at  $L$ . The three implemented functions

in Figure 4 are given the following types, respectively:

$$\mathit{ref\_make} : \forall \tau. \tau \xrightarrow{i} \mathbf{ref}(\tau) \quad \mathit{ref\_get} : \forall \tau. \mathbf{ref}(\tau) \xrightarrow{i} \tau \quad \mathit{ref\_set} : \forall \tau. \mathbf{ref}(\tau) * \tau \xrightarrow{i} \mathbf{1}$$

We use the keyword `prval` to introduce bindings on proof variables. As such bindings are to be erased before run-time, ATS automatically verifies that the pattern matching involved is exhaustive.

Clearly, `ref_get` and `ref_set` are operationally equivalent (after types and proofs are erased) to `ptr_get` and `ptr_set`, respectively, which is expected. What we find a bit surprising is that a feature as simple and common as references can involve so much type theory (on universal and existential types, linear types, sharing modality, programming with theorem proving, etc.).

**Sharable Arrays** We give an implementation of sharable arrays in ATS. The following two functions:

$$\begin{aligned} \mathbf{array\_init} &: \forall \tau. \forall \iota. \forall \lambda. \iota \geq 0 \supset (\mathit{array\_v}(\mathbf{top}, \iota, \lambda) * \mathbf{ptr}(\lambda) * \tau \xrightarrow{i} \mathit{array\_v}(\tau, \iota, \lambda) * \mathbf{1}) \\ \mathbf{array\_get} &: \forall \tau. \forall \iota_1. \forall \iota_2. \forall \lambda. 0 \leq \iota_2 \wedge \iota_2 < \iota_1 \supset \\ &\quad (\mathit{array\_v}(\tau, \iota_1, \lambda) * \mathbf{ptr}(\lambda) * \mathbf{int}(\iota_2) \xrightarrow{i} \mathit{array\_v}(\tau, \iota_1, \lambda) * \tau) \end{aligned}$$

are needed in the implementation. However, the code for implementing them is omitted here for brevity.

The sharable array can be implemented similarly. Given a type  $T$ , we define  $\mathbf{SArray}(T)$  as

$$\exists \lambda. \exists \iota. \iota \geq 0 \wedge (\square(\mathbf{int}(\iota) @ (\lambda - 1) \otimes \mathit{array\_v}(T, \iota, \lambda)) * \mathbf{ptr}(\lambda))$$

That is, we represent a sharable array as a pointer to some address  $L$  associated with a box of two proofs showing that a natural number  $I$  is stored at  $L - 1$  and an array of size  $I$  starts at  $L$ , where  $v_1, \dots, v_I$  are values stored in the array. The following functions are implemented for array creation and subscription:

$$\mathbf{SArray\_make} : \forall \tau. \mathbf{Nat} * \tau \xrightarrow{i} \mathbf{SArray}(\tau) \quad \mathbf{SArray\_get} : \forall \tau. \mathbf{SArray}(\tau) * \mathbf{Int} \xrightarrow{i} \tau$$

where  $\mathbf{Nat}$  and  $\mathbf{Int}$  are defined as  $\exists \iota. \iota \geq 0 \wedge \mathbf{int}(\iota)$  and  $\exists \iota. \mathbf{int}(\iota)$ , respectively. For brevity, the actual implementation is omitted.

Similarly, back to our motivating examples in Section 1, the `malloc` and `free` functions can now be given the ideal types (no free list threaded through) with the help of the sharing modality. The free list, as a linear resource, will be internal to `malloc` and `free` in which complex memory manipulations can be reasoned and ensured to be safe by linear types. While, the free list is completely hidden from the client point of view so that both `malloc` and `free` can be readily used in practical programming.

## 6 Code Reentrancy

While we have so far only studied the sharing modality  $\square$  in sequential programming, a major intended use of the modality is actually in multi-threaded

programming. We attempt to give an intuitive but rather informal explanation on this issue as a formalized account for multi-threaded programming is simply beyond the scope of the paper. We refer the interested readers to [14] for a theoretical development and practical examples of multi-threaded programming in ATS.

As mentioned earlier, the sharing modality  $\square$  is unable to support safe resource sharing in concurrent programming. The main reason is that a thread evaluating an expression of the form **let**  $\square x = h$  **in**  $e$  **end** may be suspended at a time when the box with the handle  $h$  is empty, and meanwhile another thread may attempt to borrow from the box and thus result in a case of double borrow. In other words, code of the form **let**  $\square x = h$  **in**  $e$  **end** is in general unsafe to be executed in a thread. However, if  $e$  can be evaluated atomically, that is, without the possibility of suspension during the evaluation of  $e$ , then it is safe to execute code of the form **let**  $\square x = h$  **in**  $e$  **end** as the problem of double borrow can no longer occur. We give some common cases where this can happen.

- If  $e$  can be compiled into a single atomic instruction, then the evaluation of  $e$  is guaranteed to be atomic. For instance, in the example on implementing references, the body of *ref\_get* (*ref\_set*) may be compiled into a single atomic read (write) instruction on memory. If this is true, then *ref\_get* (*ref\_set*) can be safely used in threads.
- Some hardware support (e.g., disabling interrupts) can be employed to ensure that the evaluation of  $e$  is atomic. For instance, the function *kmalloc* in Linux (for allocating memory in kernel space) is often implemented to be reentrant by disabling interrupts (on a single core machine) during its execution.
- A thread may use signals to put all other threads into sleep before executing non-reentrant code and then wakes them up after the execution. For instance, a thread that does garbage collection often makes use of this strategy.

Let us now introduce a half bit  $.5$  for decorating typing judgments. Note that  $b_1 \oplus b_2$  is defined as  $\max(b_1, b_2)$ . In addition to the previously presented typing rules, we also add the following one

$$\frac{\begin{array}{c} \Sigma; \bar{B}; (\Gamma; \Delta_1) \vdash_{\mu}^b e_1 : \square VT_1 \\ \Sigma; \bar{B}; (\Gamma_2; \Delta_2, x : VT_1) \vdash_{\mu}^0 e_2 : VT_1 \otimes VT_2 \end{array}}{\Sigma; \bar{B}; (\Gamma; \Delta_1 \uplus \Delta_2) \vdash_{\mu}^{.5 \oplus b} \mathbf{let} \square x = e_1 \mathbf{in} \mathbf{atm}(e_2) \mathbf{end} : VT_2} \quad (\mathbf{ty}\text{-}\square\text{-elim}\text{-}\mathbf{atom})$$

where  $\mathbf{atm}(e)$  indicates that  $e$  is an expression that must be evaluated atomically (possibly with some unspecified hardware/software support). We name this rule (**ty- $\square$ -elim-atom**). Suppose that  $\emptyset; \emptyset; (\emptyset; \emptyset) \vdash_{\mu}^b e$  is derivable. Then we expect  $e$  to be reentrant if  $b < 1$  and non-reentrant if  $b = 1$ . So for a function of type  $VT_1 \xrightarrow{b}_1 VT_2$  or  $VT_1 \xrightarrow{b}_i VT_2$ , the function is reentrant if  $b < 1$  and non-reentrant if  $b = 1$ . If thread creation only accepts functions of the viewtype  $\mathbf{1} \xrightarrow{b}_1 \mathbf{1}$  for some  $b < 1$ , then non-reentrant functions are prevented from being called within threads.

## 7 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 on Hoare logic (or its invariants) 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.

In [22], we outlined a type system  $ATS/SV$ , which is rather similar to  $\mathcal{L}_{\square, pf}^{\forall, \exists}$  minus the sharing modality.  $ATS/SV$  is effective in supporting memory manipulation through pointers, and a variety of mutable data structures (e.g., linked lists, splay trees, etc.) are implemented in  $ATS/SV$ . However, resource sharing cannot be properly dealt with in  $ATS/SV$ , causing serious difficulties in practice. The sharing modality  $\square$  is introduced precisely for the purpose of addressing this limitation in  $ATS/SV$ .

The sharing modality in our approach bears certain resemblance to the notions of focus/adoption [5] and freeze/thaw [12] respectively in the literature. However, there is some fundamental difference lying in between. In Vault [5], the sharing (of an *adoptee*) enabled by an adoption is temporary (within the scope of the *adopter*). As a consequence, this design choice makes it difficult to support general sharing such as ML references, which can be arbitrarily aliased and passed around without any constraints. We can readily give some intuitive account of adoption/focus in our framework. Given two linear resources  $r_1$  and  $r_2$ , adopting  $r_1$  by  $r_2$  corresponds to forming a combined resource  $r'_2 = r_1 \otimes r_2$  and focussing can be encoded as applying the prop  $r'_2 \rightarrow_i (r_1 \otimes (r_1 \rightarrow_l r'_2))$  to obtain/restore  $r_1$  back and forth from/to  $r'_2$ . In  $L^3$  [12], to prevent forming *re-thawing* a sharable object (similar to *double borrow* in our context), a notion named *thaw token* (which is linear) is adopted to keep track all the thawed objects. Although more flexible in theory, each thaw token must be threaded through every function that uses shared objects and the practicality of such an approach is yet to be shown.

In summary, we give a design in this paper to effectively promote the use of linear types in reasoning about resource usage in practical programming. We formalize this design in a type system where a modality is introduced to support safe resource sharing. In particular, we make use of a notion of *types with effects* [9] in overcoming the problem of double borrow. We also show some interesting and realistic programming examples including implementations of references and sharable arrays, which are all verified in ATS [20].

## References

1. A. Asperti. A logic for concurrency, 1987. Technical report, Dipartimento di Informatica, University of Pisa.
2. C. Chen. Type Inference in Applied Type System, 2005. PhD thesis, Boston University, September 2005.

3. C. Chen and H. Xi. Combining Programming with Theorem Proving. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 66–77, Tallinn, Estonia, September 2005.
4. J. Chirimar, C. A. Gunter, and G. Riecke. Reference Counting as a Computational Interpretation of Linear Logic. *Journal of Functional Programming*, 6(2):195–244, 1996.
5. M. Fahndrich and R. Deline. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 13–24, Berlin, June 2002.
6. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.
7. M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, Winter 2000.
8. A. Igarashi and N. Kobayashi. Garbage Collection Based on a Linear Type System. In *Preliminary Proceedings of the 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC '00)*, September 2000.
9. P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of 18th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 303–310, January 1991.
10. N. Kobayashi. Quasi-linear types. In *Proceedings of the 26th ACM Sigplan Symposium on Principles of Programming Languages (POPL '99)*, pages 29–42, San Antonio, Texas, USA, 1999.
11. R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.
12. G. Morrisett, A. Ahmed, and M. Fluet. L<sup>3</sup>: A Linear language with locations. In *Proceedings of Typed Lambda Calculi and Applications (TLCA '05)*, Nara, Japan, April 2005.
13. R. Shi. Types for Safe Resource Sharing in Sequential and Concurrent Programming. PhD thesis, Boston University, 2007.
14. R. Shi and H. Xi. A Linear Type System for Multicore Programming. In *Proceedings of Simposio Brasileiro de Linguagens de Programacao*, Gramado, Brazil, August 2009.
15. D. N. Turner and P. Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1–2):231–248, October 1999.
16. P. Wadler. Linear types can change the world. In *TC 2 Working Conference on Programming Concepts and Methods (Preprint)*, pages 546–566, Sea of Galilee, 1990.
17. D. Walker and K. Watkins. On regions and linear types. In *International Conference on Functional Programming*, pages 181–192, 2001.
18. H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. pp. viii+189. Available at <http://www.cs.cmu.edu/~hwxi/DML/thesis.ps>.
19. H. Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.
20. H. Xi. Applied Type System, 2005. Available at: <http://www.cs.bu.edu/~hwxi/ATS>.
21. H. Xi and F. Pfenning. Dependent Types in Practical Programming. In *Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas, January 1999. ACM press.
22. D. Zhu and H. Xi. Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, Long Beach, CA, January 2005. Springer-Verlag LNCS, 3350.