Science of Computer Programming 78 (2013) 1176–1192

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

A linear type system for multicore programming in ATS[☆]

Rui Shi[a], Hongwei Xi[b,*]

[a] Google, Inc., United States
[b] Boston University, United States

ARTICLE INFO

Article history:
Received 2 February 2010
Received in revised form 1 May 2011
Accepted 28 December 2011
Available online 2 October 2012

Keywords:
Type systems
Linear types
Multicore
ATS

ABSTRACT

In this day and age of multicore architectures, programming language support is in urgent need for constructing programs that can take great advantage of machines with multiple cores. We present in this paper an approach to safe multicore programming in ATS, a recently developed functional programming language that supports both linear and dependent types. In particular, we formalize a type system capable of guaranteeing safe manipulation of resources on multicore machines and establish its soundness. We also provide concrete examples as well as experimental results in support of the practicality of the presented approach to multicore programming.

1. Introduction

We use the phrase *multicore programming* in this paper to refer to the construction of concurrent multi-threaded programs that can run on machines with multiple cores. A fundamental challenging issue in concurrent programming is to protect (linear) resources (e.g., mutable arrays, sockets, files) from being accessed or modified in unintended manners (while still maintaining a satisfactory level of concurrency). In order to effectively reason about resource manipulation, we need a means that can describe resources in an informative manner. For instance, we may want to tell from the type assigned to a lock what kind of resource is protected by the lock. A means as such for describing resources can be found in earlier work on (stateful) views [37].

Linear types are based on linear logic [11], which, unlike classic logic or intuitionistic logic, is resource-sensitive. For instance, the following two functions (written in ML-like syntax) should be easily understood:

```
fun ignore (x: T): void = ()
fun duplicate (x: T): (T, T) = (x, x)
```

where we use (T, T) as a type for a pair of values that are of the type T. However, if T is a linear type, then neither *ignore* nor *duplicate* is well-typed. Intuitively, a linear type is for a value containing some resources (e.g., allocated memory), which cannot be arbitrarily discarded (as in the function *ignore*) or duplicated (as in the function *duplicate*). There is a common saying that one cannot have his or her cake and also eat it. Essentially, this saying amounts to the *ill-typedness* of the following function:

```
fun saying (x: cake): cake = let val () = eat (x) in x end
```

[☆] This work is supported in part by NSF grants no. CCR-0229480 and no. CCF-0702665.
[*] Corresponding author.
E-mail address: hwxi@cs.bu.edu (H. Xi).

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.09.005

```
datasort status = raw | bound | listen | connect

absview socket_v (int, status) // introducing an abstract view

dataview bind_opt_v (int, int) =
  | {i:nat} bind_none (i, -1) of socket_v (i, raw)
  | {i:nat} bind_some (i,  0) of socket_v (i, bound)
// end of [bind_opt_v]
```

Fig. 1. Some views for socket programming in ATS.

where *cake* is a linear type and *eat* is a function of the type $cake \rightarrow void$. After a value of the type *cake* is consumed (by a call to *eat*), it can no longer be returned.

We now give some explanation on using views to describe resources. Intuitively, a view can be thought of as a linear type for classifying capabilities [9,29]. For instance, given a type T and a (memory) address L, we can form a (primitive) view $T@L$ to mean that a value of type T is stored at the address L. We can also construct other forms of views in terms of primitive views. For instance, given types T_1 and T_2 and an address L, we can form a view $(T_1@L) \otimes (T_2@L+1)$ to mean that a value of type T_1 and another value of type T_2 are stored at addresses L and $L+1$, respectively, where $L+1$ stands for the address immediately after L.[1] Given an expression of some view V, we often say that the expression proves the view V and thus refer to the expression as a *proof* (of V). It is to be guaranteed (through proper typechecking) that every proof expression can be erased from a program without affecting the dynamic semantics of the program. We refer the reader to [37] for some short but realistic programs involving views and to [7,34] for some essential theoretical details on the issue of proof erasure.

We can combine a view V with a type T to form a *viewtype* $V \otimes T$ such that a value of the viewtype $V \otimes T$ is a pair $\langle pf, v\rangle$ in which pf is a proof of view V and v is a value of type T. For instance, the following type can be assigned to a function ptr_get_L that reads from the address L:

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

where we use $(\cdot, \cdot)$ to form a tuple type and $\mathbf{ptr}(L)$ for a singleton type such that the only pointer of this type is the one pointing to the address (or location) L. When applied to a proof pf_1 of view $T@L$ and a value v_1 of type $\mathbf{ptr}(L)$, the function ptr_get_L returns a pair $\langle pf_2, v_2\rangle$, where pf_2 is a proof of $T@L$ and v_2 is the value of type T that is supposed to be stored at L. We may think that the call to ptr_get_L first consumes pf_1 and then generates pf_2. Similarly, the following type can be assigned to a function ptr_set_L that writes a value of type T_2 to the address L where a value of type T_1 is stored at the time when a call to ptr_set_L is made:

$$(T_1@L, \mathbf{ptr}(L), T_2) \rightarrow (T_2@L) \otimes \mathbf{1}.$$

Note that $\mathbf{1}$ stands for the unit type. When applied to a proof pf_1 of view $T_1@L$, a value v_1 of type $\mathbf{ptr}(L)$ and another value v_2 of type T_2, ptr_set_L returns a pair $\langle pf_2, \langle\rangle\rangle$, where pf_2 is a proof of view $T_2@L$ and $\langle\rangle$ denotes the unit (of type $\mathbf{1}$). In this case, we may think that a call to ptr_set_L consumes a proof of view $T_1@L$ and then generates a proof of view $T_2@L$. In general, we can assign the following types to the read (*ptr_get*) and write (*ptr_set*) functions:

$$\begin{array}{rcl} ptr\_get & : & \forall\alpha.\forall\lambda.\ (\alpha@\lambda, \mathbf{ptr}(\lambda)) \rightarrow (\alpha@\lambda) \otimes \alpha \\ ptr\_set & : & \forall\alpha_1.\forall\alpha_2.\forall\lambda.\ (\alpha_1@\lambda, \mathbf{ptr}(\lambda), \alpha_2) \rightarrow (\alpha_2@\lambda) \otimes \mathbf{1} \end{array}$$

where we use α and λ as variables ranging over types and addresses, respectively.

Conceptually, a viewtype is for a value containing a proof part and a data part; the former is linear while the latter is non-linear. For instance, a C-style array can be assigned a viewtype in which the data part is just a pointer and the proof part describes the memory that is associated with the pointer; the proof part is erased after typechecking is done, leaving only a pointer. It is often said that a C-style array is just a pointer. However this description is grossly incomplete as it fails to mention the essential proof inside a C-style array.

In practice, we often introduce abstract views to describe (linear) resources. As an example, the code in Fig. 1 is written in the concrete syntax of ATS, a functional language with an expressive type system that supports both linear and dependent types. A datasort *status* is first declared in Fig. 1 and there are four constructors associated with it: *raw*, *bound*, *listen* and *connect*, which denote the possible states that a socket can be in. For instance, *bound* means that a socket is already bound to some port and *listen* indicates that a socket is currently listening (on certain port). A datasort declaration is like a datatype declaration in the functional language ML [19], but the declared sort is only for classifying terms to be used as indexes for forming dependent types. An abstract view constructor *socket_v* is introduced in Fig. 1. Given an integer i (representing the id or descriptor of a socket) and a status s, we can form a view $socket\_v(i, s)$ meaning that the socket whose id equals i is

[1] For a simple presentation, we assume in this paper that each value is properly boxed if necessary so that it can always be stored in one memory unit. In practice, we can and do handle unboxed values that may take multiple memory units to store.

(currently) of the status s. The dataview declaration in Fig. 1 introduces a view constructor *bind_opt_v* that takes two integers i and j to form a view $bind\_opt\_v(i, j)$. The two (proof) constructors associated with *bind_opt_v* are given the following types:

$$\begin{array}{lll} bind\_none & : & \forall i : nat.\, socket\_v(i, raw) \to bind\_opt\_v(i, -1) \\ bind\_some & : & \forall i : nat.\, socket\_v(i, bound) \to bind\_opt\_v(i, 0). \end{array}$$

If there exists a proof pf_1 of view $bind\_opt\_v(i, -1)$, then pf_1 must be of the form $bind\_none(pf'_1)$, where pf'_1 is a proof of view $socket\_v(i, raw)$. Similarly, if there exists a proof pf_2 of view $bind\_opt\_v(i, 0)$, then pf_2 must be of the form $bind\_some(pf'_2)$, where pf'_2 is a proof of view $socket\_v(i, bound)$. Therefore, the socket whose id equals i is of status *raw* if there is a proof of view $bind\_opt\_v(i, -1)$, and the socket is of status *bound* if there is a proof of view $bind\_opt\_v(i, 0)$. A function *bind*[2] for binding sockets can be given the following (dependent) type:

$$\begin{array}{lll} bind & : & \forall i : nat.\, (socket\_v(i, raw), \mathbf{int}(i)) \\ & & \quad \to \exists j : int.\, bind\_opt\_v(i, j) \otimes \mathbf{int}(j) \end{array}$$

where **int** is a type constructor that takes an integer I to form a singleton type $\mathbf{int}(I)$ in which the only value equals the integer I. The type assigned to *bind* indicates: When called on a proof showing that the socket whose id equals i is of status *raw* and a natural number equal to i (plus additional information such as a port number that is omitted here), *bind* returns an integer j paired with a proof of view $bind\_opt\_v(i, j)$; if j equals 0, then the call is successful and the status of the socket changes to *bound*; otherwise, the call results in an error and the status of the socket stays the same. Handling errors as such is of vital importance in practice.

The type system we ultimately develop involves a long line of research on dependent types [36,31], linear types [29,18,37], and programming with theorem-proving [7]. It is unrealistic to give a detailed presentation of the entire type system in this paper. Instead, we present a simple but rather abstract type system for a concurrent programming language that supports safe resource manipulation on a multicore machine, 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. The primary contribution of the paper lies in the design and formalization of a type system for supporting concurrent programming on multicore architectures. As far as we know, the approach to safe resource manipulation we take is novel and unique, and it has never before been put into practical use, especially, in a full-fledged programming language like ATS.

We organize the rest of the paper as follows. In Section 2, we formalize a concurrent programming language $\mathcal{L}_0^{||}$ with a simple linear type system, setting up some machinery for further development. We extend $\mathcal{L}_0^{||}$ to $\mathcal{L}_{\forall,\exists}^{||}$ with universally as well as existentially quantified types and then incorporate into $\mathcal{L}_{\forall,\exists}^{||}$ support for programming with theorem-proving. In Section 4, we show how support for multicore programming can be built in ATS. In addition, we present some interesting and realistic examples as well as experimental measurements. Lastly, we mention some related work and conclude.

The theoretical development of the paper is considerably dense, and the reader may want to take a look at the examples in Section 4 so as to gain some intuition before studying the formal theory presented in Sections 2 and 3.

2. A type system for parallel reduction

We first present a language $\mathcal{L}_0^{||}$ with a simple linear type system, using it as a starting point to set up the basic machinery for further development. The dynamic semantics of $\mathcal{L}_0^{||}$ is based on a form of parallel reduction that simulates multi-threaded program evaluation on a multicore machine. The omitted proof details in this section can be found in the doctoral thesis of the first author [24].

Some syntax of $\mathcal{L}_0^{||}$ is given in Fig. 2. We use x for a lam-variable and f for a fix-variable, and xf for either a lam-variable or a fix-variable. Note that a lam-variable is considered a value but a fix-variable is not. This distinction means that an expression of the form $\mathbf{fix}\, f.f$ is not allowed in $\mathcal{L}_0^{||}$.

We use cr for constant resources and c for constants, which include both constant functions cf and constant constructors cc. Note that we treat resources abstractly in $\mathcal{L}_0^{||}$. We may for instance, deal with resources of the form $I@L$, where I and L range over integers and addresses, respectively. Intuitively, $I@L$ means that the integer I is (currently) stored at the address L. A more general form of resources is $v@L$, meaning that some value v is stored at the address L.

In this paper, we assume that all constant functions cf can be implemented atomically, and the actual implementation of a constant function cf may involve the use of some locking mechanism (e.g., mutexes) for protecting the state of cf.

We use T and VT for (nonlinear) types and (linear) viewtypes, respectively, and δ and $\hat{\delta}$ for base types and base viewtypes, respectively. For instance, **bool** is the base type for booleans and **int** for integers, and **int**$@L$ is the viewtype meaning that an integer is (currently) stored at the address L.[3] The notion of views is not present in $\mathcal{L}_0^{||}$, and it is to be introduced later

2 The function *bind* corresponds to a simplified version of the function with the same name that is declared in `system/socket.h`.
3 The base type **int** here should not be confused with the dependent type constructor **int** in Section 1.

$$
\begin{array}{llll}
\text{expr.} & e & ::= & x \mid f \mid cr \mid c(e_1, \ldots, e_n) \mid \\
 & & & \langle\rangle \mid \langle e_1, e_2\rangle \mid \textbf{fst}(e) \mid \textbf{snd}(e) \mid \\
 & & & \textbf{let}\ \langle x_1, x_2\rangle = e_1\ \textbf{in}\ e_2\ \textbf{end} \mid \\
 & & & \textbf{lam}\ x.\, e \mid \textbf{app}(e_1, e_2) \mid \textbf{fix}\ f.\, v \\
\text{values} & v & ::= & cc(\vec{v}) \mid cr \mid x \mid \langle v_1, v_2\rangle \mid \textbf{lam}\ x.\, e \\
\text{types} & T & ::= & \alpha \mid \delta \mid \mathbf{1} \mid T_1 * T_2 \mid VT_1 \rightarrow_i VT_2 \\
\text{viewtypes} & VT & ::= & \hat{\alpha} \mid \hat{\delta} \mid \\
 & & & T \mid VT_1 \otimes VT_2 \mid VT_1 \rightarrow_l VT_2 \\
\text{int. exp. ctx.} & \Gamma & ::= & \emptyset \mid \Gamma, xf : T \\
\text{lin. exp. ctx.} & \Delta & ::= & \emptyset \mid \Delta, x : VT
\end{array}
$$

Fig. 2. Some syntax for $\mathcal{L}_0^{||}$.

$$
\begin{array}{rcl}
\rho(c(e_1, \ldots, e_n)) & = & \rho(e_1) \uplus \cdots \uplus \rho(e_n) \\
\rho(cr) & = & \{cr\} \\
\rho(xf) & = & \emptyset \\
\rho(\textbf{if}(e_0, e_1, e_2)) & = & \rho(e_0) \uplus \rho(e_1) \\
\rho(\langle e_1, e_2\rangle) & = & \rho(e_1) \uplus \rho(e_2) \\
\rho(\textbf{let}\ \langle x_1, x_2\rangle = \langle e_1, e_2\rangle\ \textbf{in}\ e\ \textbf{end}) & = & \rho(e_1) \uplus \rho(e_2) \uplus \rho(e) \\
\rho(\textbf{fst}(e)) & = & \rho(e) \\
\rho(\textbf{snd}(e)) & = & \rho(e) \\
\rho(\textbf{lam}\ x.\, e) & = & \rho(e) \\
\rho(\textbf{app}(e_1, e_2)) & = & \rho(e_1) \uplus \rho(e_2) \\
\rho(\textbf{fix}\ f.\, v) & = & \rho(v)
\end{array}
$$

Fig. 3. The definition of $\rho(\cdot)$.

when $\mathcal{L}_0^{||}$ is extended. We assume a signature SIG for assigning a viewtype to each constant resource *cr* and a constant type (c-type) of the form $(VT_1, \ldots, VT_n) \Rightarrow VT$ to each constant. We use α and $\hat{\alpha}$ for variables ranging over types and viewtypes, respectively, but we do not support quantification over these variables until Section 3.

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 (and thus values of this type cannot be discarded or duplicated) while $VT_1 \rightarrow_i VT_2$ means that the type itself is a type (and thus values of this type can be discarded as well as duplicated). Note that the subscript i in $\rightarrow_i$ is often dropped, that is, $\rightarrow$ is assumed to be $\rightarrow_i$ by default. 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 $\mathcal{L}_0^{||}$.

There is a special constant function *thread_create* for thread creation, which is assigned the following rather interesting c-type:

$$\textit{thread\_create} \quad : \quad (\mathbf{1} \rightarrow_l \mathbf{1}) \Rightarrow \mathbf{1}.$$

A function of the type $\mathbf{1} \rightarrow_l \mathbf{1}$ is a procedure that takes no arguments and returns no result (when its evaluation terminates). Given that $\mathbf{1} \rightarrow_l \mathbf{1}$ is a viewtype, a procedure of this type may contain resources and thus must be called exactly once. The operational semantics of *thread_create* is to be formally defined later. The function *thread_create* is currently implemented on top of pthread creation [6], and each created thread is immediately detached. We will later show that *thread_create* can be used to implement a function *thread_create_join* for creating joinable threads.

A variety of mappings, finite or infinite, are to be introduced in the rest of the presentation. We use [] for the empty mapping and $[i_1, \ldots, i_n \mapsto o_1, \ldots, o_n]$ for the finite mapping that maps i_k to o_k for $1 \le k \le n$. Given a mapping m, we write $\textbf{dom}(m)$ for the domain of m. If $i \notin \textbf{dom}(m)$, we use $m[i \mapsto o]$ for the mapping that extends m with a link from i to o. If $i \in \textbf{dom}(m)$, we use $m \backslash i$ for the mapping obtained from removing the link from i to $m(i)$ in m, and $m[i := o]$ for $(m\backslash i)[i \mapsto o]$, that is, the mapping obtained from replacing the link from i to $m(i)$ in m with another link from i to o.

We define a function $\rho(\cdot)$ in Fig. 3 to compute the *multiset* of constant resources in a given expression. Note that $\uplus$ denotes the multiset union function. In the type system of $\mathcal{L}_0^{||}$, it is to be guaranteed that $\rho(e_1)$ equals $\rho(e_2)$ whenever an expression of the form $\textbf{if}(e_0, e_1, e_2)$ is constructed, and this justifies $\rho(\textbf{if}(e_0, e_1, e_2))$ being defined as $\rho(e_0) \uplus \rho(e_1)$.

We use R to range over finite multisets of resources. Therefore, R can also be regarded as a mapping from resources to natural numbers: $R(cr) = n$ means that there are n occurrences of cr in R. It is clear that we cannot combine resources arbitrarily. For instance, it is impossible to have resources $I_1@L$ and $I_2@L$ simultaneously (regardless whether I_1 equals I_2 or not). We fix a collection **Res** of finite multisets of resources and assume the following:

- $\emptyset \in$ **Res**.
- For any R_1 and R_2, $R_2 \in$ **Res** if $R_1 \in$ **Res** 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 $R \in$ **Res** holds.

In order to formalize threads, we introduce a notion of (program) pools. We use Π for pools, which are formally defined as finite mappings from thread ids (represented as natural numbers) to (closed) expressions in $\mathcal{L}_0^{||}$ such that 0 is always in the domain of such mappings. Given a pool Π and $tid \in \mathbf{dom}(\Pi)$, we refer to $\Pi(tid)$ as a thread in Π whose id equals tid. In particular, we refer to $\Pi(0)$ as the main thread in Π. We extend the definition of $\rho(\cdot)$ as follows to compute the multiset of resources in a given pool:

$$\rho(\Pi) = \uplus_{tid \in \mathbf{dom}(\Pi)} \rho(\Pi(tid)).$$

We are to define a parallel reduction relation on pools in Section 2.2, simulating multi-threaded program evaluation on a multicore machine.

2.1. Static semantics

We present typing rules for $\mathcal{L}_0^{||}$ in this section. We require that each variable occurs at most once in an intuitionistic (linear) expression context Γ (Δ), and thus Γ (Δ) can be regarded as a finite mapping. Given Γ_1 and Γ_2 such that $\mathbf{dom}(\Gamma_1) \cap \mathbf{dom}(\Gamma_2) = \emptyset$, we write (Γ_1, Γ_2) for the union of Γ_1 and Γ_2. The same notation also applies to linear expression contexts (Δ). Given an intuitionistic expression context Γ and a linear expression context Δ, we can form an expression context $(\Gamma; \Delta)$ if $\mathbf{dom}(\Gamma) \cap \mathbf{dom}(\Delta) = \emptyset$. Given $(\Gamma; \Delta)$, we write $(\Gamma; \Delta), x : VT$ for either $(\Gamma; \Delta, x : VT)$ or $(\Gamma, x : VT; \Delta)$ (if VT is actually a type).

We use Θ for a substitution on type and viewtype variables:

$$\Theta \quad ::= \quad [] \mid \Theta[\alpha \mapsto T] \mid \Theta[\hat{\alpha} \mapsto VT].$$

Given a viewtype VT, we write $VT[\Theta]$ for the result of applying Θ to VT, which is defined in a standard manner. Given a constant resource cr, we write $\vdash cr : \hat{\delta}$ to mean that cr is assigned the viewtype $\hat{\delta}$ (in the signature SIG). Given a constant c, we use the following judgment:

$$\vdash c : (VT_1^0, \ldots, VT_n^0) \Rightarrow VT^0.$$

to mean that c is assigned a c-type of the form $(VT_1, \ldots, VT_n) \Rightarrow VT$ (in the signature SIG) and there exists Θ such that $VT_i^0 = VT_i[\Theta]$ for $1 \le i \le n$ and $VT^0 = VT[\Theta]$. In other words, $(VT_1^0, \ldots, VT_n^0) \Rightarrow VT^0$ is an instance of $(VT_1, \ldots, VT_n) \Rightarrow VT$.

A typing judgment in $\mathcal{L}_0^{||}$ is of the form $(\Gamma; \Delta) \vdash e : VT$, meaning that e can be assigned the viewtype VT under $(\Gamma; \Delta)$. The typing rules for $\mathcal{L}_0^{||}$ are listed in Fig. 4.

The following proposition, which plays a fundamental role in the design of $\mathcal{L}_0^{||}$, states that a closed value cannot contain any resources if it can be assigned a type.

Proposition 2.1. *Assume that $(\emptyset; \emptyset) \vdash v : T$ is derivable. Then $\rho(v) = \emptyset$.*

Proof. By an inspection of the rules in Fig. 4. □

The following lemma, which is often given the name of *Lemma of Canonical Forms*, relates the form of a value to its type:

Lemma 2.2. *Assume that $(\emptyset; \emptyset) \vdash v : VT$ is derivable.*

- *If $VT = \delta$, then v is of the form $cc(v_1, \ldots, v_n)$.*
- *If $VT = \hat{\delta}$, then v is of the form cr or $cc(v_1, \ldots, v_n)$.*
- *If $VT = \mathbf{1}$, then v is $\langle\rangle$.*
- *If $VT = T_1 * T_2$ or $VT = VT_1 \otimes VT_2$, then v is of the form $\langle v_1, v_2\rangle$.*
- *If $VT = VT_1 \to_i VT_2$ or $VT = VT_1 \to_l VT_2$, then v is of the form* **lam** *$x. e$.*

Proof. By an inspection of the rules in Fig. 4. □

We use θ for substitution on variables xf:

$$\theta \quad ::= \quad [] \mid \theta[x \mapsto v] \mid \theta[f \mapsto e]$$

For each θ, we define the multiset $\rho(\theta)$ of resources in θ as follows:

$$\rho(\theta) = \uplus_{xf \in \mathbf{dom}(\theta)} \rho(\theta(xf))$$

Given an expression e, we use $e[\theta]$ for the result of applying θ to e, which is defined in a standard manner. We write $(\Gamma_1; \Delta_1) \vdash \theta : (\Gamma_2; \Delta_2)$ to mean that

- $\mathbf{dom}(\theta) = \mathbf{dom}(\Gamma_2) \cup \mathbf{dom}(\Delta_2)$, and
- $(\Gamma_1; \emptyset) \vdash \theta(xf) : \Gamma_2(xf)$ is derivable for each $xf \in \Gamma_2$, and
- there exists a linear expression context $\Delta_{1,x}$ for each $x \in \mathbf{dom}(\Delta_2)$ such that $(\Gamma_1; \Delta_{1,x}) \vdash \theta(x) : \Delta_2(x)$ is derivable, and
- $\Delta_1 = \cup_{x \in \mathbf{dom}(\Delta_2)} \Delta_{1,x}$.

The following lemma, which is often given the name of *Substitution Lemma*, is needed to establish the soundness of the type system of $\mathcal{L}_0^{||}$:

Lemma 2.3. *(Substitution) Assume $(\Gamma_1; \Delta_1) \vdash \theta : (\Gamma_2; \Delta_2)$ and $(\Gamma_2; \Delta_2) \vdash e : VT$. Then $(\Gamma_1; \Delta_1) \vdash e[\theta] : VT$ is derivable and $\rho(e[\theta]) = \rho(e) \uplus \rho(\theta)$.*

Proof. By structural induction on the derivation of $(\Gamma_2; \Delta_2) \vdash e : VT$. □

$$\frac{\vdash cr : \hat{\delta}}{\Gamma; \emptyset \vdash cr : \hat{\delta}} \textbf{(ty-res)}$$

$$\frac{\begin{array}{c}\vdash c : (VT_1, \ldots, VT_n) \Rightarrow VT \\ \Gamma; \Delta_i \vdash e_i : VT_i \text{ for } 1 \le i \le n\end{array}}{\Gamma; \Delta_1, \ldots, \Delta_n \vdash c(e_1, \ldots, e_n) : VT} \textbf{(ty-cst)}$$

$$\frac{}{(\Gamma, xf : T; \emptyset) \vdash xf : T} \textbf{(ty-var-i)}$$

$$\frac{}{(\Gamma; \emptyset, x : VT) \vdash x : VT} \textbf{(ty-var-l)}$$

$$\frac{\begin{array}{c}\Gamma; \Delta_0 \vdash e_0 : \mathbf{bool} \\ \Gamma; \Delta \vdash e_1 : VT \quad \Gamma; \Delta \vdash e_2 : VT \quad \rho(e_1) = \rho(e_2)\end{array}}{\Gamma; \Delta_0, \Delta \vdash \mathbf{if}(e_0, e_1, e_2) : VT} \textbf{(ty-if)}$$

$$\frac{}{\Gamma; \emptyset \vdash \langle\rangle : \mathbf{1}} \textbf{(ty-unit)}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : T_1 \quad \Gamma; \Delta_2 \vdash e_2 : T_2}{\Gamma; \Delta_1, \Delta_2 \vdash \langle e_1, e_2\rangle : T_1 * T_2} \textbf{(ty-tup-i)}$$

$$\frac{\Gamma; \Delta \vdash e : T_1 * T_2}{\Gamma; \Delta \vdash \mathbf{fst}(e) : T_1} \textbf{(ty-fst)} \qquad \frac{\Gamma; \Delta \vdash e : T_1 * T_2}{\Gamma; \Delta \vdash \mathbf{snd}(e) : T_2} \textbf{(ty-snd)}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : VT_1 \quad \Gamma; \Delta_2 \vdash e_2 : VT_2}{\Gamma; \Delta_1, \Delta_2 \vdash \langle e_1, e_2\rangle : VT_1 \otimes VT_2} \textbf{(ty-tup-l)}$$

$$\frac{\begin{array}{c}\Gamma; \Delta_1 \vdash e_1 : VT_1 \otimes VT_2 \\ \Gamma; \Delta_2, x_1 : VT_1, x_2 : VT_2 \vdash e_2 : VT\end{array}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{let}\ \langle x_1, x_2\rangle = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : VT} \textbf{(ty-tup-l-elim)}$$

$$\frac{(\Gamma; \Delta), x : VT_1 \vdash e : VT_2}{\Gamma; \Delta \vdash \mathbf{lam}\, x.\, e : VT_1 \to_l VT_2} \textbf{(ty-lam-l)}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : VT_1 \to_l VT_2 \quad \Gamma; \Delta_2 \vdash e_2 : VT_1}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{app}(e_1, e_2) : VT_2} \textbf{(ty-app-l)}$$

$$\frac{(\Gamma; \emptyset), x : VT_1 \vdash e : VT_2 \quad \rho(e) = \emptyset}{\Gamma; \emptyset \vdash \mathbf{lam}\, x.\, e : VT_1 \to_i VT_2} \textbf{(ty-lam-i)}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : VT_1 \to_i VT_2 \quad \Gamma; \Delta_2 \vdash e_2 : VT_1}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{app}(e_1, e_2) : VT_2} \textbf{(ty-app-i)}$$

$$\frac{\Gamma, f : T; \emptyset \vdash v : T}{\Gamma; \emptyset \vdash \mathbf{fix}\, f.\, v : T} \textbf{(ty-fix)}$$

$$\frac{\begin{array}{c}(\emptyset; \emptyset) \vdash \Pi(0) : VT \\ (\emptyset; \emptyset) \vdash \Pi(tid) : \mathbf{1} \text{ for each } 0 < tid \in \mathbf{dom}(\Pi)\end{array}}{\vdash \Pi : VT} \textbf{(ty-pool)}$$

Fig. 4. The typing rules for $\mathcal{L}_0^{||}$.

2.2. Dynamic semantics

We present evaluation rules for $\mathcal{L}_0^{||}$ in this section. The evaluation contexts in $\mathcal{L}_0^{||}$ are defined below:

$$\begin{array}{l}\text{eval ctx.} \quad E \quad ::= \\ \quad [\,] \mid c(v_1, \ldots, v_{i-1}, E, e_{i+1}, \ldots, e_n) \mid \mathbf{if}(E, e_1, e_2) \mid \\ \quad \langle E, e\rangle \mid \langle v, E\rangle \mid \mathbf{let}\ \langle x_1, x_2\rangle = E\ \mathbf{in}\ e_1\ \mathbf{end} \mid \\ \quad \mathbf{fst}(E) \mid \mathbf{snd}(E) \mid \mathbf{app}(E, e) \mid \mathbf{app}(v, E)\end{array}$$

Given an evaluation context E and an expression e, we use $E[e]$ for the expression obtained from replacing the hole [] in E with e.

Definition 2.4. We define pure redexes and their reducts as follows.

- $\mathbf{if}(\mathit{true}, e_1, e_2)$ is a pure redex, and its reduct is e_1.
- $\mathbf{if}(\mathit{false}, e_1, e_2)$ 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.
- $\mathbf{app}(\mathbf{lam}\ x.\,e, v)$ 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 $\mathcal{L}_0^{||}$. Assume that cf is a constant function of arity n. The expression $cf(v_1, \ldots, v_n)$ is an *ad hoc* redex if cf is defined at $v_1, \ldots, v_n$, and any value of $cf(v_1, \ldots, v_n)$ is a reduct of $cf(v_1, \ldots, v_n)$. For instance, $1 + 1$ is an ad hoc redex and 2 is its sole reduct. In contrast, $1 + \mathit{true}$ is not a redex as it is undefined. More interestingly, if we assume the availability of a nullary nondeterministic function *randbit* that can be called to generate bits randomly, then $\mathit{randbit}()$ is an ad hoc redex, and both 0 and 1 are its reducts. There are also constant functions for manipulating resources. For instance, we can assume a binary constant function *locadd* that consumes resources $I_1@L_1$ and $I_2@L_2$ for some distinct addresses L_1 and L_2 and then generates $\langle I_1@L_1, (I_1 + I_2)@L_2\rangle$. In other words, *locadd* adds the contents in two addresses L_1 and L_2 and then stores the result into L_2 (while preserving the content of L_1). As another example, *alloc* is a function that takes a natural number n to return a pointer to some address L associated with a tuple of resources $\langle v_0@L, v_1@L + 1, \ldots, v_{n-1}@L + n - 1\rangle$ for some values $v_0, v_1, \ldots, v_{n-1}$, that is, $\mathit{alloc}(n)$ reduces to a pointer that points to n consecutive memory units containing some unspecified values.

Definition 2.5. Given expressions e_1 and e_2, we write $e_1 \rightarrow e_2$ if $e_1 = E[e]$ and $e_2 = E[e']$ for some E, e and e' such that e' is a reduct of e and $\rho(e_2) \in \mathbf{Res}$, and we may say that e_1 reduces to e_2 purely if e is a pure redex.

The following proposition states that resources are preserved when pure redexes are reduced.

Proposition 2.6. *Assume that $(\emptyset; \emptyset) \vdash e_1 : VT$ is derivable and e_1 reduces to e_2 purely. Then $\rho(e_1) = \rho(e_2)$.*

Proof. With Proposition 2.1, the proof follows from structural induction on the derivation of $(\emptyset; \emptyset) \vdash e_1 : VT$. □

It is important to note that resources may be generated as well as consumed when ad hoc reduction occurs. Suppose that $e_1 = E[\mathit{alloc}(1)]$ and $v@L$ occurs in E. Though $\langle v@L, L\rangle$ is a reduct of *alloc*, we cannot allow $e_1 \rightarrow E[\langle v@L, L\rangle]$ as the resource $v@L$ occurs repeatedly in $E[\langle v@L, L\rangle]$. This is precisely the reason that we require $\rho(e_2) \in \mathbf{Res}$ whenever $e_1 \rightarrow e_2$ holds.

We state as follows the standard subject reduction theorem on expressions:

Theorem 2.7 (*Subject Reduction on Expressions*)**.** *Assume that $(\emptyset; \emptyset) \vdash e : VT$ is derivable and $e \rightarrow e'$ holds. Then $(\emptyset; \emptyset) \vdash e' : VT$ is derivable.*

Proof. By structural induction on the derivation of $(\emptyset; \emptyset) \vdash e : VT$. The Lemma 2.3 is needed in this proof. □

We state as follows the standard progress theorem on expressions:

Theorem 2.8 (*Progress on Expressions*)**.** *Assume that $(\emptyset; \emptyset) \vdash e : VT$ is derivable. Then we have the following possibilities:*

1. *e is a value, or*
2. *$e \rightarrow e'$ holds for some expression e'.*

Proof. By structural induction on the derivation of $(\emptyset; \emptyset) \vdash e : VT$. The Lemma 2.2 is needed in this proof. □

Parallel Reduction for Pools We now write $e_1 \Rightarrow e_2$ to mean either $e_1 = e_2$ or $e_1 \rightarrow e_2$. The single step parallel reduction relation on pools is given by the following rules:

$$\frac{\Pi_1(\mathit{tid}) \Rightarrow \Pi_2(\mathit{tid}) \text{ for each } \mathit{tid} \in \mathbf{dom}(\Pi_1) = \mathbf{dom}(\Pi_2)}{\Pi_1 \Rightarrow \Pi_2}\ \text{(PR1)}$$

$$\frac{\Pi_1(\mathit{tid}_0) = E[\mathit{thread\_create}(\mathbf{lam}\ x.\,e)] \quad \Pi_1 \backslash \mathit{tid}_0 \Rightarrow \Pi_2}{\Pi_1 \Rightarrow \Pi_2[\mathit{tid}_0 \mapsto E[\langle\rangle]][\mathit{tid} \mapsto \mathbf{app}(\mathbf{lam}\ x.\,e, \langle\rangle)]}\ \text{(PR2)}$$

$$\frac{\Pi_1 \Rightarrow \Pi_2}{\Pi_1[\mathit{tid} \mapsto \langle\rangle] \Rightarrow \Pi_2}\ \text{(PR3)}.$$

To some extent, the definition of $\Rightarrow$ is analogous to the definition of parallel reduction in λ-calculus [25]. The rule PR1 essentially captures the idea that an indefinite number of threads may be evaluated simultaneously. As for thread creation and termination, the rules PR2 and PR3 should apply, respectively. Note that a thread created by applying the rule PR2 is detached. We will show later how a joinable thread can be implemented on the top of a detached one.

Proposition 2.9. *If $\Pi_1 \Rightarrow \Pi_2$ holds and $\rho(\Pi_1) \in \mathbf{Res}$, then $\rho(\Pi_2) \in \mathbf{Res}$.*

Proof. The proposition follows from the definition of the evaluation relation $\Rightarrow$ on pools. □

The soundness of the type system of $\mathcal{L}_0^{||}$ rests upon the following two theorems:

Theorem 2.10 (*Subject Reduction on Pools*)**.** *Assume that* $\vdash \Pi_1 : VT$ *is derivable and* $\Pi_1 \Rightarrow \Pi_2$*. Then* $\vdash \Pi_2 : VT$ *is derivable and* $\rho(\Pi_2) \in$ **Res** *is valid.*

Theorem 2.11. *(Progress on Pools) Assume that* $\vdash \Pi_1 : VT$ *is derivable and* $\rho(\Pi_1)$ *is valid. Then we have the following possibilities:*

- Π_1 *is a singleton mapping* $[0 \mapsto v]$ *for some* v*, or*
- $\Pi_1 \Rightarrow \Pi_2$ *holds for some* Π_2*.*

Please see [24] for the detailed proofs[4] of Theorems 2.10 and 2.11.

Remark 1. Theorems 2.10 and 2.11 may seem abstract but they can yield many desirable consequences. In particular, it guarantees that various race conditions can never occur in $\mathcal{L}_0^{||}$, and this is achieved by the virtue of the designed type system. For instance, suppose that a thread (with its id = tid_1) makes the following function call $ptr\_set(r, v_{ptr}, v)$, where r is a resource of viewtype $T@L$ and v_{ptr} is the pointer to L and v is some value to be stored at L. At this point, it is impossible for any other thread to make a call of the following form $ptr\_set(r, v_{ptr}, v')$. Otherwise, there must be two occurrences of r in $R = \rho(\Pi)$, where Π is the current pool (of threads), thus making R invalid. In other words, a write/write race condition can never occur in $\mathcal{L}_0^{||}$. Also, it is clear by a similar argument that a read/write race condition can never occur in $\mathcal{L}_0^{||}$. Actually, it is impossible in $\mathcal{L}_0^{||}$ for two threads to read from the same address at any moment. To eliminate this restriction, we can introduce another form of resources for read-only access and allow each read-only resource to occur repeatedly in a valid multiset R of resources.

Remark 2. While the type system of $\mathcal{L}_0^{||}$ can prevent races conditions from ever happening, which is of great value in the construction of concurrent programs, we must also make sure that programming in such a type system can still be productive. After all, there is little practical value of any type system if one can hardly construct programs in it. Although it is difficult to address the issue of practicality formally, what we can claim is that the type system of $\mathcal{L}_0^{||}$ (or its design) forms the basis of the type system of ATS, and there are a variety of concurrent programs written in ATS (e.g., device driver, scheduler) that can be found on-line or mentioned in the literature [23,10].

3. Extensions

While the basic design of a type system for supporting safe concurrent programming with shared resources is already present in $\mathcal{L}_0^{||}$, the viewtypes in $\mathcal{L}_0^{||}$ are often not expressive enough to allow for accurate specification of resources that appear in practice. In this section, we extend $\mathcal{L}_0^{||}$ to $\mathcal{L}_{\forall,\exists}^{||}$ with universal and existential viewtypes and then incorporate into $\mathcal{L}_{\forall,\exists}^{||}$ support for *programming with theorem proving* [7]. These added features are all needed for the presentation of some interesting and realistic examples in Section 4.

3.1. Predicatization

We now outline an extension of $\mathcal{L}_0^{||}$ to $\mathcal{L}_{\forall,\exists}^{||}$ with universally as well as existentially quantified (dependent and polymorphic) viewtypes. This extension is mostly a standard process in the framework *Applied Type System* ($\mathcal{ATS}$) [33], and we refer to this process as *predicatization*. The machinery for the theoretical development of $\mathcal{L}_{\forall,\exists}^{||}$ can all be found in the prior work on Dependent ML (DML) [35], and we suggest that it be read together with the current section if needed.

As an applied type system, $\mathcal{L}_{\forall,\exists}^{||}$ consists of a static component (statics) and a dynamic component (dynamics). Some extra syntax of $\mathcal{L}_{\forall,\exists}^{||}$ (over that of $\mathcal{L}_0^{||}$) is given in Fig. 5. The statics itself is a simply typed language and a type in it is referred to as a *sort*. We assume the existence of the following basic sorts: *addr*, *bool*, *int*, *type* and *viewtype* whose meanings are self-explanatory. In practice, we also allow new sorts to be introduced through datasort declarations as shown in Section 1.

In the following presentation, a static term is either an address L, or a boolean B, or an integer I, or a type T, or a viewtype VT. We assume some primitive static functions for constructing terms of sorts *addr*, *bool* and *int*, respectively. For instance, we can form terms such as $L + I$ (pointer arithmetic), $I_1 + I_2$, $I_1 - I_2$, $I_1 \leq I_2$, $\neg B$, $B_1 \wedge B_2$, etc.

The base types δ in $\mathcal{L}_0^{||}$ are now base type constructors that take static terms to form types. Formally, each δ in $\mathcal{L}_{\forall,\exists}^{||}$ is assigned a c-sort of the form $(\sigma_1, \ldots, \sigma_n) \rightarrow type$, which indicates that δ can be applied to static terms s_i of sorts σ_i for $1 \leq i \leq n$ to form a type $\delta(s_1, \ldots, s_n)$. For instance, the base type constructors **bool**, **int** and **ptr** are of c-sorts $(bool) \rightarrow type$, $(int) \rightarrow type$ and $(addr) \rightarrow type$ respectively; **bool**(B) is a singleton type in which the only value is the truth value of B; **int**(I) is a singleton type in which the only value is the integer I; **ptr**(L) is a singleton type in

4 In [24], the language $\mathcal{L}_0^{||}$ is actually richer than what is presented in this paper: It also contains support for linear locks and tickets.

$$
\begin{array}{llcl}
\text{sorts} & \sigma & ::= & \mathit{bool} \mid \mathit{int} \mid \mathit{addr} \mid \mathit{type} \mid \mathit{viewtype} \\
\text{types} & T & ::= & \ldots \mid a \mid B \supset T \mid \forall a : \sigma \,.\, T \mid \\
 & & & B \wedge T \mid \exists a : \sigma \,.\, T \\
\text{viewtypes} & \mathit{VT} & ::= & \ldots \mid a \mid B \supset \mathit{VT} \mid \forall a : \sigma \,.\, \mathit{VT} \\
 & & & \mid B \wedge \mathit{VT} \mid \exists a : \sigma \,.\, \mathit{VT} \\
\text{expr.} & e & ::= & \ldots \mid \supset^+(v) \mid \supset^-(e) \mid \forall^+(v) \mid \forall^-(e) \mid \\
 & & & \wedge(e) \mid \textbf{let } \wedge(x) = e_1 \textbf{ in } e_2 \textbf{ end} \mid \\
 & & & \exists(e) \mid \textbf{let } \exists(x) = e_1 \textbf{ in } e_2 \textbf{ end} \\
\text{values} & v & ::= & \ldots \mid \supset^+(v) \mid \forall^+(v) \mid \wedge(v) \mid \exists(v)
\end{array}
$$

Fig. 5. Some syntax for $\mathcal{L}^{||}_{\forall,\exists}$.

which the only value is the pointer to the address L. Similarly, we can have base viewtype constructors assigned c-sorts of the form $(\sigma_1, \ldots, \sigma_n) \to \mathit{viewtype}$ in $\mathcal{L}^{||}_{\forall,\exists}$. For instance, we may use **var** for a base viewtype constructor of the c-sort $(\mathit{type}, \mathit{addr}) \to \mathit{viewtype}$: $\textbf{var}(T, L)$ is a viewtype for a pointer to the address L where a value of type T is stored.

The general form of a c-type in $\mathcal{L}^{||}_{\forall,\exists}$ is

$$\forall \Sigma .\, \overline{B} \supset ((\mathit{VT}_1, \ldots, \mathit{VT}_n) \Rightarrow \mathit{VT})$$

which is shorthand for:

$$\forall a_1 : \sigma_1 \ldots \forall a_n : \sigma_n .\, B_1 \supset (\ldots (B_n \supset ((\mathit{VT}_1, \ldots, \mathit{VT}_n) \Rightarrow \mathit{VT})) \ldots)$$

where $\Sigma = a_1 : \sigma_1, \ldots, a_n : \sigma_n$ and $\overline{B} = B_1, \ldots, B_n$.

We call $B \supset T$ a guarded type and $B \wedge T$ an asserting type. As an example, the following type is for a function from natural numbers to negative integers:

$$\forall a_1 : \mathit{int}.a_1 \geq 0 \supset (\textbf{int}(a_1) \to_i \exists a_2 : \mathit{int}.(a_2 < 0) \wedge \textbf{int}(a_2)).$$

The guard $a_1 \geq 0$ indicates that the function can only be applied to an integer that is greater than or equal to 0; the assertion $a_2 < 0$ means that each integer returned by the function is negative. The guarded viewtypes $B \supset \mathit{VT}$ and asserting viewtypes $B \wedge \mathit{VT}$ are similar.

An expression e in $\mathcal{L}^{||}_{\forall,\exists}$ is referred to as a dynamic term. The markers $\supset^+(\cdot)$, $\supset^-(\cdot)$, $\wedge(\cdot)$, $\forall^+(\cdot)$, $\forall^-(\cdot)$, $\exists(\cdot)$ are primarily introduced to guarantee that the type of a value in $\mathcal{L}^{||}_{\forall,\exists}$ uniquely determines the form of the value. This is a property we need when establishing the so-called progress theorem for $\mathcal{L}^{||}_{\forall,\exists}$, that is, demonstrating that each closed well-typed expression in $\mathcal{L}^{||}_{\forall,\exists}$ is either a value or can be further evaluated. Please see [35] for proofs that make essential use of these markers.

Following the development of $\mathcal{L}^{||}_0$ and $\mathcal{ATS}$, it is a standard routine to establish the type soundness of $\mathcal{L}^{||}_{\forall,\exists}$.

3.2. Programming with theorem proving

In order to effectively deal with resource manipulation, we also need to support a paradigm that combines programming with theorem-proving. For brevity, we cannot give a detailed account of this paradigm in this paper. Instead, we refer the reader to a recent paper [7] for theoretical details and practical examples. We now sketch an extension of $\mathcal{L}^{||}_{\forall,\exists}$ to include support for programming with theorem-proving, briefly explaining the basic idea behind this extension.

We introduce two sorts *prop* and *view*, and use P and V for props and views, that is, static terms of the sorts *prop* and *view*, respectively. We assume that *prop* is a subsort of *view*, and thus every prop is considered a view. The relation between *prop* and *view* is parallel to that between *type* and *viewtype*. When assigning a prop or a view to a dynamic term, we need to verify that the dynamic term is total, that is, pure and terminating. This is done by employing a technique developed in Dependent ML [32] for program termination verification.

The primary difference between props (views) and types (viewtypes) is that the former can only be assigned to total dynamic terms, i.e., proofs, while the latter can be assigned to effectful and nonterminating dynamic terms, i.e., programs. The proofs are needed only for type-checking, and they are completely erased at compile-time and can thus incur no run-time overhead.

In the following presentation, we use @ for an infix base view constructor of the c-sort $(\mathit{type}, \mathit{addr}) \to \mathit{view}$. Given T and L, the view $T@L$ can be assigned to a resource of the form $v@L$ for some value v of type T. So, $T@L$ means that a value of type T is stored at address L. We assume the existence of primitive memory access functions *ptr_get* and *ptr_set* of the following c-types:

$$
\begin{array}{lcl}
\mathit{ptr\_get} & : & \forall \alpha . \forall \lambda .\, (\alpha @ \lambda, \textbf{ptr}(\lambda)) \Rightarrow (\alpha @ \lambda) \otimes \alpha \\
\mathit{ptr\_set} & : & \forall \alpha_1 . \forall \alpha_2 . \forall \lambda .\, (\alpha_1 @ \lambda, \textbf{ptr}(\lambda), \alpha_2) \Rightarrow (\alpha_2 @ \lambda) \otimes \mathbf{1}
\end{array}
.
$$

$$\begin{array}{rcl} \mathit{uplock\_create} & : & \forall\hat{\alpha}.\ \mathbf{1} \rightarrow_i \mathbf{uplock0}(\hat{\alpha}) \\ \mathit{uplock\_destroy} & : & \forall\hat{\alpha}.\ \mathbf{uplock1}(\hat{\alpha}) \rightarrow_i \hat{\alpha} \\ \mathit{upticket\_create} & : & \forall\hat{\alpha}.\ \mathbf{uplock0}(\hat{\alpha}) \rightarrow_i \mathbf{uplock1}(\hat{\alpha}) \otimes \mathbf{upticket}(\hat{\alpha}) \\ \mathit{upticket\_destroy} & : & \forall\hat{\alpha}.\ \mathbf{upticket}(\hat{\alpha}) \otimes \hat{\alpha} \rightarrow_i \mathbf{1} \end{array}$$

Fig. 6. Some functions handling locks and tickets for uploading.

where *ptr_get* reads through a pointer and *ptr_set* writes through a pointer. Note that we use ι and λ for variables ranging over static terms of the sorts *int* and *addr*, respectively.

In order to specify more sophisticated memory layouts, we need to form recursive views. For instance, we can declare a (dependent) view constructor *array_v*: Given a type T, an integer I and an address L, $\mathit{array\_v}(T, I, L)$ forms a view stating that there are I values of type T stored at addresses $L, L+1, \ldots, L+I-1$. There are two proof constructors *array_none* and *array_some* associated with *array_v*, which are assigned the following c-types:

$$\begin{array}{rcl} \mathit{array\_none} & : & \forall\lambda.\forall\alpha.() \Rightarrow \mathit{array\_v}(\alpha, 0, \lambda) \\ \mathit{array\_some} & : & \forall\lambda.\forall\alpha.\forall\iota.\iota \geq 0 \supset \\ & & (\alpha@\lambda, \mathit{array\_v}(\alpha, \iota, \lambda+1)) \Rightarrow \mathit{array\_v}(\alpha, \iota+1, \lambda) \end{array}$$

Note that we use () in the type of *array_none* to mean that the constructor takes no arguments. Intuitively, *array_none*() is a proof of $\mathit{array\_v}(T, 0, L)$ for any type T and address L, and $\mathit{array\_some}(pf_1, pf_2)$ is a proof of $\mathit{array\_v}(T, I+1, L)$ for any type T, integer I and address L if pf_1 and pf_2 are proofs of views $T@L$ and $\mathit{array\_v}(T, I, L+1)$, respectively. Moreover, we can implement proofs to manipulate views. For instance, two proof functions *array_split* and *array_unsplit* of the following props can be constructed to split an array into two consecutive ones and unsplit two consecutive arrays into one, respectively:

$$\begin{array}{rcl} \mathit{array\_split} & : & \forall\alpha.\forall\iota_1.\forall\iota_2.\forall\lambda.\ (0 \leq \iota_2 \wedge \iota_2 \leq \iota_1) \supset \\ & & (\mathit{array\_v}(\alpha, \iota_1, \lambda), \mathbf{int}(\iota_2)) \rightarrow_i \\ & & \mathit{array\_v}(\alpha, \iota_2, \lambda) \otimes \mathit{array\_v}(\alpha, \iota_1 - \iota_2, \lambda + \iota_2) \\ \mathit{array\_unsplit} & : & \forall\alpha.\forall\iota_1.\forall\iota_2.\forall\lambda. \\ & & \mathit{array\_v}(\alpha, \iota_1, \lambda) \otimes \mathit{array\_v}(\alpha, \iota_2, \lambda + \iota_1) \rightarrow_i \\ & & \mathit{array\_v}(\alpha, \iota_1 + \iota_2, \lambda). \end{array}$$

These two proof functions are used in a multi-threaded implementation of quicksort to be presented next.

4. Multicore programming in ATS

We are now ready to outline the approach to multicore programming in ATS.

4.1. Linear locks and tickets for uploading

We need a means for a child thread to pass values to its parent thread. For this, we introduce three type constructors **uplock0**, **uplock1** and **upticket** of c-sort (*viewtype*) $\rightarrow$ *viewtype*. So each of these constructors can form a viewtype when applied to a viewtype. Some functions associated with these type constructors are given in Fig. 6.

Let *VT* be certain viewtype. By calling the function *uplock_create*, we can create an uninitialized lock of the viewtype **uplock0**(*VT*). Then by calling the function *upticket_create* on this uninitialized lock, we obtain an initialized lock of the viewtype **uplock1**(*VT*) and a ticket of the viewtype **upticket**(*VT*). We can call *upticket_destroy* on the ticket paired with a value of the viewtype *VT* to upload the value into the lock with which the ticket is associated, and this call also destroys the ticket. In order to retrieve the uploaded value, we can call *uplock_destroy* on the initialized lock, and this call destroys the lock when it returns. In the case where *upticket_destroy* is called on a lock into which no value has been uploaded, the call is blocked. In essence, we can use the functions in Fig. 6 to build a linear (i.e., one-time) communication channel for a thread to send a value to another thread.

4.2. Joinable threads

As is stated earlier, threads created by *thread_create* are detached. In practice, joinable threads are also widely used. Let **tid** be a viewtype constructor of c-sort (*viewtype*) $\rightarrow$ *viewtype*. Given a viewtype *VT*, a value of the viewtype **tid**(*VT*) is assumed to contain certain necessary information (e.g., thread id) about a joinable thread with a return value of viewtype *VT*. The constant function for creating joinable threads is given as follows:

$$\mathit{thread\_create\_join} \quad : \quad \forall\hat{\alpha}.(\mathbf{1} \rightarrow_l \hat{\alpha}) \Rightarrow \mathbf{tid}(\hat{\alpha}).$$

Intuitively, *thread_create_join*(f) spawns a thread and a value of viewtype **tid**(*VT*) is returned immediately so that the following function can use the value to join the thread after it terminates:

$$\mathit{thread\_join} \quad : \quad \forall\hat{\alpha}.(\mathbf{tid}(\hat{\alpha})) \Rightarrow \hat{\alpha}.$$

```
viewtypedef tid (a: viewtype) = uplock1 (a)

// we use [-<lin>] to for a linear function type
fun thread_create_join
  {a:viewtype} (f: () -<lin> a): tid (a) = let

  // creating a lock for uploading
  val lock0 = uplock_create {a} ()

  // creating a ticket for uploading
  val (lock1, tick) = upticket_create (lock0)

  // [llam] indicates that a linear function is constructed
  val () = begin // spawning a thread
    thread_create (llam () => upticket_upload (tick, f ()))
  end
in
  lock1 // it is needed for retrieving the uploaded value
end // end of [thread_create_join]

fun thread_join {a:viewtype} (lock: tid a): a =
  uplock_destroy (lock) // retrieving the uploaded value
```

Fig. 7. Implementing joinable threads.

In Fig. 7, we give an implementation of the function *thread_create_join* based on *thread_create*, where **tid**(*VT*) is defined to be **uplock1**(*VT*). In order to create a joinable thread using a function f of the viewtype $\mathbf{1} \rightarrow_l VT$ for some viewtype *VT*, we can first create an uninitialized lock for uploading by calling the function *uplock_create*. We next call *upticket_create* on the uninitialized lock to obtain an initialized lock and a ticket for uploading. We then construct a function f' that calls f and then uses the ticket to upload the value returned by f into the lock with which the ticket is associated. Lastly, we create a detached thread using f' and then return the initialized lock immediately. The function *thread_join* simply waits on the lock returned by a call to *thread_create_join* until it is available, and then *thread_join* destroys the lock and returns the value already uploaded into the lock.

As an example, we present an implementation of the Fibonacci function in Fig. 8 that makes use of joinable threads.

4.3. Scheduled spawning and synchronizing

The functions *thread_create* and *thread_create_join* do not take into account the availability of CPU resource. So a function like *fib_mt1* is most likely to run *slower* rather than faster when compared a sequential implementation as the cost for thread creation can easily surpass any gain obtained from concurrent execution.

We have implemented in ATS two functions *spawn* and *sync* of the following types:

$$\begin{array}{rcl} \textit{spawn} & : & \forall\hat{\alpha}.\, (\mathbf{1} \rightarrow_l \hat{\alpha}) \rightarrow_i \mathbf{spawn}(\hat{\alpha}) \\ \textit{sync} & : & \forall\hat{\alpha}.\, \mathbf{spawn}(\hat{\alpha}) \rightarrow_i \hat{\alpha} \end{array}$$

where **spawn** is an abstract type constructor that forms a viewtype when applied to a viewtype. When executing a program on a multicore machine, we first create $N - 1$ threads (not counting the main thread), where N is often the number of cores in the machine. When *spawn* is called on a function, it determines at run-time whether the work to evaluate the function should be done by the current thread or put into a queue so that another thread can pick it up. This strategy, which is standard, eliminates the cost of thread creation at run-time. As an example, the second implementation of the Fibonacci function in Fig. 8 makes use of *spawn* and *sync*.

Parallel let-binding In order to better support multicore programming, we have introduced into ATS some syntax to support parallel let-binding. Intuitively, the keyword *par* in the following pseudo-code

```
let
  // [x1] and [x2] represent some variables
  // [e1] and [e2] represent some expressions
  val par x1 = e1 and x2 = e2 // parallel let-binding
in
  // the scope of this parallel let-binding
end
```

```
// [CUTOFF] is some fixed integer constant
// [fib] refers to a sequential implementation of the Fibonacci
// function

// An implementation using joinable threads
fun fib_mt1 (n: int): int = begin
  if n <= CUTOFF then fib (n) else let
    // [llam] indicates the construction of a linear function
    val tid1 = thread_create_join (llam () => fib_mt1 (n-1))
    val tid2 = thread_create_join (llam () => fib_mt1 (n-2))
    val ans1 = thread_join (tid1)
    val ans2 = thread_join (tid2)
  in
    ans1 + ans2
  end // end of [if]
end // end of [fib_mt1]

// An implementation using [spawn] and [sync]
fun fib_mt2 (n: int): int = begin
  if n <= CUTOFF then fib (n) else let
    val tid1 = spawn (llam () => fib_mt2 (n-1))
    val tid2 = spawn (llam () => fib_mt2 (n-2))
    val ans1 = sync (tid1)
    val ans2 = sync (tid2)
  in
    ans1 + ans2
  end // end of [if]
end // end of [fib_mt2]

// An implementation using parallel let-binding
fun fib_mt3 (n: int): int = begin
  if n <= CUTOFF then fib (n) else let
    // The keyword [par] indicates parallel let-binding
    val par ans1 = fib_mt3 (n-1) and ans2 = fib_mt3 (n-2)
  in
    ans1 + ans2
  end // end of [if]
end // end of [fib_mt3]
```

Fig. 8. Implementations of the Fibonacci function.

indicates that the let-expression should be transformed into the following one:

```
let
  // [tid1] and [tid2] are some fresh variable names
  val tid1 = spawn (llam () => e1)
  val tid2 = spawn (llam () => e2)
  val x1 = sync (tid1)
  val x2 = sync (tid2)
in
  // the scope of this parallel let-binding
end
```

Evidently, the 2 bindings here generalize to n bindings for every $n \geq 2$.

As an example, the third implementation of the Fibonacci function in Fig. 8 makes use of this let-binding syntax. This implementation is essentially a sugared version of the second implementation of the Fibonacci function.

Compared to *spawn* and *sync*, parallel let-binding makes code cleaner and probably easier to understand. However, parallel let-binding is also less flexible. For instance, we find it difficult to employ parallel let-binding when handling Erastothenes's sieve algorithm (for finding prime numbers). Instead, we need make direct use of *spawn* and *sync*.

```
//
// multi-threaded quicksort: CUTOFF is some fixed integer
//
fun qsort_mt {n:nat} {l:addr}
  (pf: array_v (int, n, l) | p: ptr l, n: int n)
  : (array_v (int, n, l) | unit) = begin
  if n <= CUTOFF then qsort (pf | p, n) else let
    val (pf | i) = partition (pf | p, n)

    // pf1 : array_v (int,i,l) and pf2 : array_v (int,n-i,l+i)
    prval (pf1, pf2) = array_split (pf | i)

    // pf21 : int @ (l+i) and pf22 : array_v (int,n-i-1,l+i+1)
    prval array_some (pf21, pf22) = pf2

    val par // parallel let-binding
      (pf1 | ()) = qsort_mt (pf1 | p, i)
    and // notice the pointer arithmetic
      (pf22 | ()) = qsort_mt (pf22 | p + (i + 1), n - (i + 1))

    // reassemble the view of the entire array
    // pf : array_v (a, n, l)
    prval pf = array_unsplit (pf1, array_some (pf21, pf22))
  in
    (pf | ())
  end
end // end of [qsort_mt]
```

Fig. 9. A multi-threaded implementation of quicksort in ATS.

$$
\begin{array}{rcl}
\mathit{partition} & : & \forall\iota.\forall\lambda.\,(0 < \iota) \supset \\
 & & \quad \mathit{array\_v}(\mathbf{int},\iota,\lambda) \otimes (\mathbf{ptr}(\lambda) * \mathbf{int}(\iota)) \rightarrow_i \\
 & & \qquad \exists\iota'.\,(0 \leq \iota' \wedge \iota' < \iota) \wedge (\mathit{array\_v}(\mathbf{int},\iota,\lambda) \otimes \mathbf{int}(\iota')) \\
\mathit{qsort} & : & \forall\iota.\forall\lambda.\,(0 \leq \iota) \supset \\
 & & \quad \mathit{array\_v}(\mathbf{int},\iota,\lambda) \otimes (\mathbf{ptr}(\lambda) * \mathbf{int}(\iota)) \rightarrow_i \mathit{array\_v}(\mathbf{int},\iota,\lambda) \otimes \mathbf{1} \\
\mathit{qsort\_mt} & : & \forall\iota.\forall\lambda.\,(0 \leq \iota) \supset \\
 & & \quad \mathit{array\_v}(\mathbf{int},\iota,\lambda) \otimes (\mathbf{ptr}(\lambda) * \mathbf{int}(\iota)) \rightarrow_i \mathit{array\_v}(\mathbf{int},\iota,\lambda) \otimes \mathbf{1}
\end{array}
$$

Fig. 10. The types of some functions in the multi-threaded quicksort implementation.

4.4. Examples

We have already put into practice the developed type theory and support for concurrent programming on multicore architectures. Larger and more realistic examples that we have completed include a scull (simple character utility for loading localities) Linux device driver [23] and a web server. Also, some examples of concurrent programming in ATS are available online. We have a process to elaborate programs written in the concrete syntax of ATS, which is largely based on the syntax of Standard ML [19], into the (properly extended) formal syntax of $\mathcal{L}^{||}_{\forall,\exists}$, but it is beyond the scope of the paper to discuss details about this elaboration process. Instead, we rely on (informal) comments to help the reader understand the meaning of the presented code.

We now present an implementation of quicksort in Fig. 9 that makes use of parallel let-binding. Given an array of elements, we use a pivot to partition the array into two subarrays such that the first subarray contains all the elements that are less than or equal to the pivot and the second subarray contains the rest of the elements. A thread may be spawned to sort a subarray, depending on the run-time scheduling. The types of some functions involved in the implementation are formally written in Fig. 10.

- The type of *partition* means that the function takes a proof of view $\mathit{array\_v}(T, I, L)$ for some T, I and L, a pointer pointing to L and an integer equal to I, and it returns a proof of view $\mathit{array\_v}(T, I, L)$ and an integer I' satisfying $0 \leq I' < I$. The effect of *partition* should be obvious, and the actual implementation of *partition* is omitted.
- The type of *qsort* means that the function takes a comparison function, a proof of view $\mathit{array\_v}(T, I, L)$ for some T, I and L, a pointer pointing to L and an integer equal to I, and it returns a proof of view $\mathit{array\_v}(T, I, L)$ (and the unit). The implementation of *qsort* is omitted.

- The type of *qsort_mt* is the same as that of *qsort*. The implementation of *qsort_mt* in Fig. 9 largely corresponds to a similar one in Cilk [26].

Note that the concurrent implementation of quicksort in Fig. 9 becomes a sequential one if the keyword *par* is erased.

Of course, such an implementation of quicksort can be readily done in many other languages like Cilk. However, the type system of ATS can guarantee that the implementation in Fig. 9 is memory-safe and free of race conditions even though it involves explicit use of pointers and pointer arithmetic. At present, we are not aware of other languages with support for multicore programming that can provide a guarantee as such.

5. Some issues in concurrent programming

Concurrent programming is complex and difficult. There are many issues in concurrent programming that we have not even touched upon so far. We now mention two important issues that we have dealt with in ATS.

5.1. Deadlocking

Deadlocking is a challenging issue in concurrent programming. Just like we can construct nonterminating programs in ATS, we can readily construct deadlocking programs as well. However, it is possible to employ linear types in ATS to statically enforce locking policies that can prevent certain forms of deadlocking. For instance, suppose that there are two kinds of locks: A and B, and we want to enforce a policy stating that any thread holding a B lock is not allowed to acquire an A lock. This policy can prevent a form of so-called "ABBA" deadlocking (this is the kind of deadlocking depicted in the famous example of five philosophers). To do this, we can introduce two kinds of (linear) tickets: A and B. In order to acquire an A lock, an A ticket and a B ticket must be present but only an A ticket is consumed. In order to acquire a B lock, only a B ticket needs to be present and it is consumed. If an A(B) lock is released, then an A(B) ticket is returned. As a thread holding a B lock can no longer show a B ticket, it becomes impossible for the thread to acquire an A lock (since doing so requires the presence of a B ticket). The actual implementation of *spawn* in ATS, which is available online, does precisely what is described here to prevent this kind of "ABBA" deadlocking.

5.2. Reentrancy

Reentrancy is another challenging issue in concurrent programming. As not all the shared resources are protected by locks, we need to prevent a scenario in which such a resource is manipulated concurrently by two or more threads. A well-known example is the free-list data structure used by functions like *malloc* and *free*. If two or more threads call such functions concurrently, then the free-list data structure is most likely to become corrupted. Functions like *malloc* and *free* are nonreentrant as they manipulate shared resources not protected by locks. In ATS, we make use of a notion of *types with effects* [16] to track nonreentrant functions (among other things), preventing them from being called within any thread except the main one. Please see [24] for more details on handling the issue of reentrancy in ATS.

6. Experimentation

We now report some performance measurements in Fig. 11, which we gathered when running the following examples on an IBM xSeries 365 server with four 2.5 GHz Intel Xeon CPUs and 16 GB RAM.

- The fibonacci example is given in Fig. 8, where the value of CUTOFF is set to be 20.
- The partial-sum example sums up the power series $\Sigma_{i=1}^{n}(2/3)^i$ for $n = 10^7$.
- The quicksort example implements the standard quicksort and it sorts an array of size 50,000,000 in which each element is a randomly generated float point number of double precision.
- The mergesort example implements the standard mergesort and it sorts an array of size 50,000,000 in which each element is a randomly generated float point number of double precision.
- The sieve example implements Eratosthenes's sieve algorithm and finds all the prime numbers up to 10^8. The test we use determines that a natural number n is a prime if it cannot be divided by any natural number satisfying $1 < i \leq \sqrt{n}$.

The source code for all these examples is available online. All of them except the last one can be readily parallelized by employing the syntax for parallel let-binding. The sieve example is significantly more involved and makes explicit use of *spawn* and *sync*. When compared to the other examples, this one is less commonly used for demonstrating parallelism due to some intricacy involved in parallelizing it.

In Fig. 11, the numbers N in parentheses indicate how many cores are used in the experiments. The numbers in the column *user+sys* report the sum of user time and system time (spent on all cores) in each experiment while the numbers in the column *real* show the wall clock time. The time unit is second. The numbers in the column *PCPU* indicate the percentage of CPU usage, which is bounded by $N \cdot 100\%$. One major goal of multicore programming is to minimize the real time by maximizing the use of all available cores.

While the scheduling algorithm (for spawning work) in ATS is still in its infancy, these numbers do bring a sense of realism. We are currently improving the scheduling algorithm by following the strategy in Cilk [26].

	user+sys	real	PCPU
fibonacci(1)	2.62	2.62	100%
fibonacci(2)	2.79	1.40	200%
fibonacci(3)	2.94	1.00	292%
fibonacci(4)	3.08	0.82	375%
partial-sum(1)	8.14	8.14	100%
partial-sum(2)	8.17	4.08	200%
partial-sum(3)	8.26	2.78	296%
partial-sum(4)	8.28	2.10	394%
quicksort(1)	49.90	49.91	100%
quicksort(2)	51.50	30.38	169%
quicksort(3)	53.61	26.09	205%
quicksort(4)	58.92	25.38	232%
mergesort(1)	84.00	84.00	100%
mergesort(2)	85.03	47.49	179%
mergesort(3)	88.60	39.22	226%
mergesort(4)	89.50	30.89	290%
sieve(1)	11.52	11.52	100%
sieve(2)	13.13	8.89	147%
sieve(3)	13.94	6.82	204%
sieve(4)	15.24	6.29	242%

Fig. 11. Some performance measurements.

7. Related work and conclusion

The potential of linear logic in facilitating reasoning on resource usage has long been recognized. For instance, Asperti showed an interesting way to describe Petri nets [2] 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. [28,8,27,17,15,14,30]). However, convincing uses of linear types in practical programming are still rather rare.

The type system developed in this paper involves a long line of research on dependent types [36,31], linear types [29,18,37], and programming with theorem-proving [7]. In Dependent ML [36,31], a restricted form of dependent types is introduced that completely separates programs from types, and this design makes it straightforward to support realistic programming features such as general recursion and effects in the presence of dependent types. Subsequently, (recursive) alias types [29] are studied in an attempt to specify or model mutable data structures. However, programming with alias types is greatly limited by the lack of a proof system for reasoning about such types. This situation is remedied in a recent study that combines programming with theorem-proving [7], allowing the programmer to handle hard type constraints by supplying explicit proofs.

Clay [13] is a recently developed programming language which uses linear as well as dependent types to reason about low-level memory manipulation. There is certain resemblance between ATS and Clay. For instance, both views and viewtypes have their counterparts in Clay. Also, the sort *view* and proof functions in our system roughly corresponds to the kind $\hat{0}$ and the coercion functions in Clay, respectively. At this point, Clay is mostly a language rather close to the level of assembly for supporting systems programming.

Along a closely related but different line of research, separation logic [22] is introduced as an extension to Hoare logic in support of reasoning about shared mutable data structures. Recently, it is shown in [21] how separation logic can be extended with rules to reason about concurrent programs, which is referred to as *Concurrent Separation Logic (CSL)*. Also, a denotational semantics based on action traces is constructed for CSL [5]. The key insight of CSL lies in the new interpretation of assertions as permissions, which are transferred among multiple concurrent threads. In contrast, Separation Logic, as a programming logic, does not directly interact with the type system of the underlying language, which may make it unwieldy to integrate programming features such as abstracting over specifications, strong updates, polymorphism, etc.

Hoare Type Theory (HTT) [20] is a recently developed framework for reasoning about memory manipulation in imperative programming. The key merit of HTT lies in a novel combination of Hoare-style specification and types. Notably, the introduction of Hoare types of the form $\{P\}x : \tau\{Q\}$ makes it possible to specify computations with precondition P and postcondition Q and return a value of type τ. Ideas from Separation Logic are adopted by HTT as well to enable local reasoning. In contrast, we syntactically restrict the effects from appearing in types whereas HTT depends on monadically encapsulated computations. The pros and cons of these two different designs are yet to be observed in practice.

A common approach to facilitate the construction of safe concurrent programs is through race detection. So far there have been a number of studies on using types [1] or ownership types [4] to prevent races on shared data among multiple threads. In these systems, types are formed to record sets of locks associated with objects and typing rules are designed to reflect a form of control-flow analysis on lock uses in programs that are often written in Java. In contrast, our system starts from a different perspective, that is, focusing on ensuring consistency of resources. In addition, the use of views allows us to support fine-grained resource sharing (e.g., in the multi-threaded quicksort implementation, locks are employed to protect different portions of an array), which seems difficult to achieve in these systems designed for race detection.

Programs that may cause deadlocking at run-time can be constructed in $\mathcal{L}^{||}_{\forall,\exists}$. A simple strategy in detecting potential deadlocks at compile-time is to associate with locks a partial order and then verify that threads must acquire locks in an increasing order [1,3]. We may incorporate this idea into the type system of $\mathcal{L}^{||}_{\forall,\exists}$ in future. However, we emphasize that there are various causes for deadlocks that cannot be prevented by this simple strategy. In practice, many deadlocks result from program errors. For instance, a familiar scenario is that a thread holding a lock may exit without releasing the lock. The use of linear locks can already detect this kind of error at compile-time.

Another promising approach to multicore programming is based on transactional memory (TM). For instance, Haskell is taking such an approach [12]. The approach we present is mostly orthogonal to transactional memory.

Acknowledgments

We thank Rick Lavoie for his efforts on implementing a thread-safe garbage collector for ATS. The second author also wants to acknowledge some helpful discussions with Likai Liu on the subject of concurrent programming.

Appendix. More information on ATS

ATS is freely available to the public, and it can be downloaded from the following site:

http://www.ats-lang.org.

The source code for programs used in benchmarking is available at

http://www.ats-lang.org/htdocs-old/EXAMPLE/MULTICORE/.

The implementation of *spawn* and *sync* is available at

http://www.ats-lang.org/htdocs-old/IMPLEMENTATION/Geizella/ATS/libats/DATS/parallel.dats.

References

[1] M. Abadi, C. Flanagan, S. N. Freund, Types for safe locking: static race detection for java, ACM Trans. Program. Lang. Syst. 28 (2) (2006) 207–255.
[2] A. Asperti, A logic for concurrency, Tech. rep., Dipartimento di Informatica, University of Pisa, 1987.
[3] C. Boyapati, R. Lee, M. Rinard, Ownership types for safe programming: preventing data races and deadlocks, in: 17th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA. pp. 211–230, November 2002. URL: citeseer.ist.psu.edu/boyapati02ownership.html.
[4] C. Boyapati, M. Rinard, A parameterized type system for race-free Java programs, in: 16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA. Tampa Bay, FL, pp. 56–69, October 2001.
[5] S. D. Brookes, A semantics for concurrent separation logic, in: CONCUR, pp. 16–34, 2004.
[6] D. R. Butenhof, Programming with POSIX Threads, Addison Wesley Professional, 1997.
[7] C. Chen, H. Xi, Combining programming with theorem proving, in: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming, Tallinn, Estonia, pp. 66–77, September 2005.
[8] J. Chirimar, C. A. Gunter, G. Riecke, Reference counting as a computational interpretation of linear logic, Journal of Functional Programming 6 (2) (1996) 195–244.
[9] K. Crary, D. Walker, G. Morrisett, Typed Memory Management in a Calculus of Capabilities, in: Proceedings of the 26th ACM Symposium on Principles of Programming Languages, San Antonio, TX, pp. 262–275, January 1999.
[10] M. Danish, H. Xi, Operating System Development with ATS, in: Proceedings of the International Workshop on Programming Languages Meets Program Verification, PLPV'10, Madrid, Spain, pp. 9–14, January 2010.
[11] J.-Y. Girard, Linear logic, Theoretical Computer Science 50 (1) (1987) 1–101.
[12] T. Harris, S. Marlow, S. Peyton-Jones, M. Herlihy, Composable memory transactions, in: Proceedings of the 2005 ACM SIGPLAN symposium on principles and practice of parallel programming. Chicago, IL, pp. 48–60, June 2005.

[13] C. Hawblitzel, E. Wei, H. Huang, E. Krupski, L. Wittie, Low-level linear memory management, in: Proceedings of SPACE 2004, 2004.
[14] M. Hofmann, A type system for bounded space and functional in-place update, Nordic Journal of Computing 7 (4) (2000) 258–289.
[15] A. Igarashi, 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.
[16] P. Jouvelot, D. K. Gifford, Algebraic reconstruction of types and effects, in: Proceedings of 18th ACM SIGPLAN Symposium on Principles of Programming Languages, pp. 303–310, January 1991.
[17] N. Kobayashi, 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, 1999.
[18] Y. Mandelbaum, D. Walker, R. Harper, An effective theory of type refinements, in: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, Uppsala, Sweden, pp. 213–226, September 2003.
[19] R. Milner, M. Tofte, R. W. Harper, D. MacQueen, The Definition of Standard ML (Revised), MIT Press, Cambridge, Massachusetts, 1997.
[20] A. Nanevski, G. Morrisett, L. Birkedal, Polymorphism and separation in Hoare type theory, in: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP'06, ACM Press, New York, NY, USA, 2006, pp. 62–73.
[21] P. W. O'Hearn, Resources, concurrency and local reasoning, in: CONCUR, in: LNCS, vol. 3170, Springer-Verlag, 2004, pp. 49–67.
[22] J. Reynolds, Separation logic: a logic for shared mutable data structures, in: Proceedings of 17th IEEE Symposium on Logic in Computer Science, LICS'02, IEEE Computer Society, 2002, pp. 55–74.
[23] R. Shi, Implementing reliable linux device drivers in ATS, in: Proceedings of the International Workshop on Programming Languages Meets Program Verification, PLPV'07, Freiberg, Germany, October 2007, pp. 41–45.
[24] R. Shi, Types for safe resource sharing in sequential and concurrent programming, Ph.D. Dissertation, Boston University, 2007, Available at http://cs-people.bu.edu/shearer/thesis/.
[25] M. Takahashi, Parallel reduction, Information and Computation 118 (1995) 120–127.
[26] The Cilk Development Team, 2009, The Cilk Project. Available at: http://supertech.csail.mit.edu/cilk.
[27] D. N. Turner, P. Wadler, Operational interpretations of linear logic, Theoretical Computer Science 227 (1–2) (1999) 231–248.
[28] P. Wadler, Linear types can change the world, in: M. Broy, C. Jones (Eds.), Programming Concepts and Methods, 1990, pp. 561–581.
[29] D. Walker, G. Morrisett, Alias types for recursive data structures, in: Proceedings of International Workshop on Types in Compilation, in: LNCS, vol. 2071, Springer-Verlag, 2000, pp. 177–206.
[30] D. Walker, K. Watkins, On regions and linear types, in: Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming, ICFP'01, Florence, Italy, pp. 181–192, 2001.
[31] H. Xi, Dependent types in practical programming. Ph.D. Thesis, Carnegie Mellon University, 1998, pp. viii+189. Available at: http://www.cs.cmu.edu/~hwxi/DML/thesis.ps.
[32] H. Xi, Dependent types for program termination verification, Journal of Higher-Order and Symbolic Computation 15 (1) (2002) 91–132.
[33] H. Xi, Applied type system (extended abstract), in: Post-Workshop Proceedings of TYPES 2003, in: LNCS, vol. 3085, Springer-Verlag, 2004, pp. 394–408.
[34] H. Xi, Attributive types for proof erasure, in: Post-Workshop Proceedings of TYPES 2007, in: LNCS, Springer-Verlag, 2007, pp. 188–202.
[35] H. Xi, Dependent ML: an approach to practical programming with dependent types, Journal of Functional Programming 17 (2) (2007) 215–286.
[36] H. Xi, F. Pfenning, Dependent types in practical programming, in: Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages, ACM press, San Antonio, Texas, 1999, pp. 214–227.
[37] D. Zhu, H. Xi, Safe programming with pointers through stateful views, in: Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages, in: LNCS, vol. 3350, Springer-Verlag, Long Beach, CA, 2005, pp. 83–97.