A Linear Type System for Multicore Programming*

Rui Shi[1] **and Hongwei Xi**[2]

[1]Yahoo! Inc.

[2]Boston University

ruish@yahoo-inc.com, hwxi@cs.bu.edu

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 (Zhu and Xi 2005). 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 (Crary, Walker, and Morrisett 1999; Walker and Morrisett 2000). 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

* This work is supported in part by NSF grants no. CCR-0229480 and no. CCF-0702665.

[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.

to (Zhu and Xi 2005) for some short but realistic programs involving views and to (Chen and Xi 2005; Xi 2007) 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 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 $\mathbf{ptr}(L)$ denotes a singleton type for the only pointer that points to the address 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 **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 **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:

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

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

The type system we ultimately develop involves a long line of research on dependent types (Xi and Pfenning 1999; Xi 1998), linear types (Walker and Morrisett 2000; Mandelbaum, Walker, and Harper 2003; Zhu and Xi 2005), and programming with theorem proving (Chen and Xi 2005). 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 involves 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 (in a full-fledged programming language).

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 mention an extension of $\mathcal{L}_0^{||}$ to $\mathcal{L}_{\forall,\exists}^{||}$ with universally as well as existentially quantified types and then incorporate into $\mathcal{L}_{\forall,\exists}^{||}$ support for

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

Figure 1. Some syntax for $\mathcal{L}_0^{||}$

programming with theorem proving. In Section 3, 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.

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 elsewhere (Shi 2007).

Some syntax of $\mathcal{L}_0^{||}$ is given in Figure 1. 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.

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[2]. The notion of views is not present in $\mathcal{L}_0^{||}$, and it is to be introduced later 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

[2]The base type **int** here should not be confused with the dependent type constructor **int** in Section 1.

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

Figure 2. The definition of $\rho(\cdot)$

$(VT_1,\ldots,VT_n) \Rightarrow VT$ to each constant. We use α and $\hat{\alpha}$ for variables ranging over types and viewtype, respectively.

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). 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:

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

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 (Butenhof 1997), 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 $1 \leq k \leq n$. Given a mapping m, we write $\mathbf{dom}(m)$ for the domain of m. If $i \notin \mathbf{dom}(m)$, we use $m[i \mapsto o]$ for the mapping that extends m with a link from i to o. If $i \in \mathbf{dom}(m)$, 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 Figure 2 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 $\mathbf{if}(e_0, e_1, e_2)$ is constructed, and this justifies $\rho(\mathbf{if}(e_0, e_1, e_2))$ 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 $\mathbf{Res}$ of finite multisets of resources and assume the following:

- $\emptyset \in \mathbf{Res}$.
- For any R_1 and R_2, $R_2 \in \mathbf{Res}$ if $R_1 \in \mathbf{Res}$ 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 $R \in \mathbf{Res}$ 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 occur 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. 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 $1 \leq i \leq n$ 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 Figure 3.

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

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

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

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

$$\frac{}{\Gamma; \emptyset \vdash \langle\rangle : \textbf{1}}\ \textbf{(ty-unit)} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : T_1 \quad \Gamma; \Delta_2 \vdash e_2 : T_2}{\Gamma; \Delta_1, \Delta_2 \vdash \langle e_1, e_2\rangle : T_1 * T_2}\ \textbf{(ty-tup-i)}$$

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

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

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

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

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

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

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

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

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

Figure 3. 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:

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

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 2.1 *We define pure redexes and their reducts as follows.*

- $\textbf{if}(\textit{true}, e_1, e_2)$ *is a pure redex, and its reduct is* e_1.
- $\textbf{if}(\textit{false}, e_1, e_2)$ *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, $1 +$ *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 $randbit()$ 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, *alloc*$(n)$ reduces to a pointer that points to n consecutive memory units containing some unspecified values.

Definition 2.2 *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.*

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.

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:

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

$$\frac{\Pi_1(tid_0) = E[\mathit{thread\_create}(\mathbf{lam}\ x.\, e)] \quad \Pi_1 \backslash tid_0 \Rightarrow \Pi_2}{\Pi_1 \Rightarrow \Pi_2[tid_0 \mapsto E[\langle\rangle]][tid \mapsto \mathbf{app}(\mathbf{lam}\ x.\, e, \langle\rangle)]}\ \text{(PR2)}$$

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

To some extent, the definition of $\Rightarrow$ is analogous to the definition of parallel reduction in λ-calculus (Takahashi 1995). The rule PR1 essentially captures the idea that an indefinite number of threads may be evaluated simultaneous. As for thread creation and termination, the rules PR2 and PR3 should be applied, 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.

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

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

Figure 4. Some functions handling locks and tickets for uploading

Theorem 2.3 *(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* $\rho(\Pi_2) \in \mathbf{Res}$ *is valid.*

Theorem 2.4 *(Progress on Pools) Assume that* $\vdash\ \Pi_1\ :\ VT$ *is derivable and also* $\rho(\Pi_1)$ *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 (Shi 2007) for the detailed proofs[3] of Theorem 2.3 and Theorem 2.4.

Remark Theorem 2.3 and Theorem 2.4 may seem abstract but they can yield many desirable consequences. In particular, they guarantee that various race conditions can never occur in $\mathcal{L}_0^{||}$. 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.

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. We need to 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*. This extension process is given the name *predicatization*, and it is formally presented in (Chen and Xi 2005).

3. Multicore Programming in ATS

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

3.1. 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 $(viewtype) \rightarrow$

[3] In (Shi 2007), the language $\mathcal{L}_0^{||}$ is actually richer than what is presented in this paper: It also contains support for linear locks and tickets.

```
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 // this lock 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
```

Figure 5. Implementing joinable threads

$viewtype$. So each of these constructors can form a viewtype when applied to a viewtype. Some functions associated with these type constructors are given in Figure 4.

Let VT be certain viewtype. By calling $uplock\_create$, we can create an uninitialized lock of the viewtype $\mathbf{uplock0}(VT)$. Then by calling $upticket\_create$ on this uninitialized lock, we obtain an initialized lock of the viewtype $\mathbf{uplock1}(VT)$ and a ticket of the viewtype $\mathbf{upticket}(VT)$. 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 $uplock\_destroy$ is called on a lock into which no value has been uploaded, the call is blocked. In essence, we can use the functions in Figure 4 to build a linear (i.e., one-time) communication channel for a thread to send a value to another thread.

3.2. 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 $\mathbf{tid}(VT)$ 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:

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

Intuitively, $thread\_create\_join(f)$ spawns a thread and a value of viewtype $\mathbf{tid}(VT)$ is returned immediately so that the following function can use the value to join the thread after it terminates:

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

We give an implementation of the function $thread\_create\_join$ based on $thread\_create$ in Figure 5, where $\mathbf{tid}(VT)$ is defined to be $\mathbf{uplock1}(VT)$. 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.

3.3. 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:

$$spawn \quad : \quad \forall\hat{\alpha}.\,(\mathbf{1} \rightarrow_l \hat{\alpha}) \rightarrow_i \mathbf{spawn}(\hat{\alpha}) \qquad\qquad sync \quad : \quad \forall\hat{\alpha}.\,\mathbf{spawn}(\hat{\alpha}) \rightarrow_i \hat{\alpha}$$

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.

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 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 // end of [let]
```

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) and x2 = sync (tid2)
in
  // the scope of this let-binding
end // end of [let]
```

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

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 Eratosthenes's sieve algorithm (for finding prime numbers). Instead, we need make direct use of $spawn$ and $sync$.

	user+sys	real	PCPU		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%

Figure 6. Some Performance Measurements

4. Experimentation

We now report some performance measurements in Figure 6, which we gathered when running the following examples on an IBM xSeries 365 server with four 2.5 GHz Intel Xeon CPUs and 16GB RAM.

- The fibonacci example is a standard implementation that computes Fibonacci numbers.
- 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 on-line. 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 Figure 6, 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 (The Cilk Development Team).

5. 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 (Asperti 1987) 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. (Wadler 1990; Chirimar, Gunter, and Riecke 1996; Turner and Wadler 1999; Kobayashi 1999; Igarashi and Kobayashi 2000; Hofmann 2000; Walker and Watkins 2001)). However, convincing uses of linear types in practical programming are still rather rare.

The type system developed in this paper rests upon a great deal of research on dependent types (Xi and Pfenning 1999; Xi 1998), linear types (Walker and Morrisett 2000; Mandelbaum, Walker, and Harper 2003; Zhu and Xi 2005), and programming with theorem proving (Chen and Xi 2005). In Dependent ML (Xi and Pfenning 1999; Xi 1998), 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 (Walker and Morrisett 2000) 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 (Chen and Xi 2005), allowing the programmer to handle hard type constraints by supplying explicit proofs.

Hoare Type Theory (HTT) (Nanevski, Morrisett, and Birkedal 2006) 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 facilitating the construction of safe concurrent programs is through race detection. So far there have been a number of studies on using types (Flanagan and Freund 2000; Abadi, Flanagan, and Freund 2006) or ownership types (Boyapati and Rinard 2001) 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 (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 detecting race conditions.

Another promising approach to multicore programming is based on transactional memory (TM). For instance, Haskell is taking such an approach (Harris, Marlow, Peyton-Jones, and Herlihy 2005). The approach we present is mostly orthogonal to transactional memory. As a matter of fact, we are currently considering adding (software) support for TM in ATS as well.

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.

References

Abadi, M., C. Flanagan, and S. N. Freund (2006). Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst. 28*(2), 207–255.

Asperti, A. (1987). A logic for concurrency. Technical report, Dipartimento di Informatica, University of Pisa.

Boyapati, C. and M. Rinard (2001, October). 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.

Butenhof, D. R. (1997). *Programming with POSIX Threads*. Addison Wesley Professional.

Chen, C. and H. Xi (2005, September). Combining Programming with Theorem Proving. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, Tallinn, Estonia, pp. 66–77.

Chirimar, J., C. A. Gunter, and G. Riecke (1996). Reference Counting as a Computational Interpretation of Linear Logic. *Journal of Functional Programming 6*(2), 195–244.

Crary, K., D. Walker, and G. Morrisett (1999, January). 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.

Flanagan, C. and S. Freund (2000, June). Type-based race detection for Java. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada.

Harris, T., S. Marlow, S. Peyton-Jones, and M. Herlihy (2005, June). Composable Memory Transactions. In *Proceedings of the 2005 ACM SIGPLAN symposium on principles and practice of parallel programming*, Chicago, IL, pp. 48–60.

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

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

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

Mandelbaum, Y., D. Walker, and R. Harper (2003, September). An effective theory of type refinements. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, Uppsala, Sweden, pp. 213–226.

Nanevski, A., G. Morrisett, and L. Birkedal (2006, September). Polymorphism and separation in hoare type theory. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, New York, NY, USA, pp. 62–73. ACM Press.

Shi, R. (2007). *Types for Safe Resource Sharing in Sequential and Concurrent Programming*. Ph. D. dissertation, Boston University. Available at: `http://cs-people.bu.edu/shearer/thesis/`.

Takahashi, M. (1995). Parallel Reduction. *Information and Computation 118*, 120–127.

The Cilk Development Team. The Cilk Project. Available at: `http://supertech.csail.mit.edu/cilk`.

Turner, D. N. and P. Wadler (1999, October). Operational interpretations of linear logic. *Theoretical Computer Science 227*(1–2), 231–248.

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

Walker, D. and G. Morrisett (2000, September). Alias Types for Recursive Data Structures. In *Proceedings of International Workshop on Types in Compilation*, pp. 177–206. Springer-Verlag LNCS vol. 2071.

Walker, D. and K. Watkins (2001, September). On Regions and Linear Types. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*, Florence, Italy, pp. 181–192.

Xi, H. The ATS Programming Language. Available at: `http://www.ats-lang.org/`.

Xi, H. (1998). *Dependent Types in Practical Programming*. Ph. D. thesis, Carnegie Mellon University. pp. viii+189. Available at: `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.

Xi, H. (2007). Attributive Types for Proof Erasure. In *post-workshop Proceedings of TYPES 2007*. Springer-Verlag LNCS.

Xi, H. and F. Pfenning (1999, January). Dependent Types in Practical Programming. In *Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, San Antonio, Texas, pp. 214–227. ACM press.

Zhu, D. and H. Xi (2005, January). Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, Long Beach, CA. Springer-Verlag LNCS, 3350.

A. 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/EXAMPLE/MULTICORE/`

The implementation of $spawn$ and $sync$ is avaliable at

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