Typed Assembly Language for Implementing OS Kernels in SMP/Multi-Core Environments with Interrupts

Toshiyuki Maeda
University of Tokyo

Akinori Yonezawa
University of Tokyo

Abstract

Although many people still think that it is difficult or even impossible to implement OS kernels in a strictly typed programming language, we dispelled the myth in our previous works by designing and implementing a typed assembly language which is flexible enough to implement essential functionalities of OS kernels (e.g., memory and multi-thread management facilities).

Taking a step further, in this paper, we show an extended typed assembly language which supports SMP/multi-core environments with CPU hardware interrupts and illustrate how to implement synchronization primitives, which are essential for implementing OS kernels in the environments.

1 Introduction

It has been considered difficult to implement system software (e.g., OS kernels) in strictly typed languages. This is because conventional strictly typed languages were not expressive enough to implement fundamental components of system software, such as memory management and multi-thread management facilities. Several previous works tried to implement OS kernels in strictly typed programming languages, but the fundamental components were out of their scope [2, 10] or rely on other verification approaches [20].

On the other hand, we have been working on implementation of an OS kernel in Typed Assembly Language (TAL) [16]. TAL is an ordinary assembly language except for being strictly typed and its memory safety (programs perform no illegal memory accesses) and control-flow safety (programs perform no illegal code execution) can be verified through its type checking. In fact, we designed and implemented a variant of TAL which is expressive enough to implement OS kernels and implemented a simple OS kernel (including the memory and multi-thread management facilities) in it [14, 13, 15].

However, there exists one problem; our previous works did not support SMP/multi-core environments with interrupts where programs run physically concurrently and asynchronously.

To solve the problem, this paper shows an extension of our previous TAL that supports the SMP/multi-core environments with interrupts. More precisely, this paper shows how to extend the type system for implementing synchronization primitives themselves that are essential for OS kernels in the SMP/multi-core environments.

There are two issues to be addressed for supporting the SMP/multi-core environments: safety of shared memory (that is, memory regions that may be accessed simultaneously by several programs) and its memory consistency model. This paper mainly focuses on the safety of shared memory, and discusses the consistency model briefly.

The rest of this paper is organized as follows. First, Sec. 2 shows problems for ensuring the safety of shared memory. Then, Sec. 3 describes our approach to the problems and Sec. 4 presents our extended TAL based on our approach. Next, Sec. 5 argues about the memory consistency model briefly. Finally, Sec. 6 discusses related work and Sec. 7 concludes the paper.

2 Problems in Handling Shared Memory

In the SMP/multi-core environments, one memory region can be accessed simultaneously by several programs. In order to achieve TAL which is able to ensure safety of shared memory and yet expressive enough to implement system software, we have to deal with memory updates that may alter the types of memory regions (we call them strong updates in this paper) and track pointer aliases between multiple threads.

2.1 Strong Updates

In typical and conventional strictly typed languages, the problem of strong updates does not apply simply because

they are not allowed (the types of memory regions are invariant). Therefore, the memory safety is not violated even if the memory regions are updated simultaneously (as long as the updates are atomic).

However, in order to implement memory and multi-thread management facilities, it is necessary to allow programs to perform strong updates. For example, typical implementation of memory management (e.g., malloc/free) requires to handle allocation and release of memory regions, that is, a memory region which is allocated as a certain type may be released and reused (re-allocated) as a different type from the original one.

In order to achieve type-safe strong updates in single-core environments, we designed and implemented a TAL [14] whose type system is integrated with alias types [18] and dependent types [19]. More precisely, illegal memory accesses are prevented by tracking pointer aliases with the type system. We also implemented the memory and multi-thread management facilities for the single-core environments in the extended TAL.

However, in order to adapt the language for the multi-core environments, it is necessary to address another problem described in the next section.

2.2 Pointer Aliases between Threads

In order to adapt the extended TAL mentioned in Sec. 1 for the multi-core environments, we need to track pointer aliases between multiple threads.

Conventional programming languages are equipped with synchronization mechanisms as language primitives. Therefore, pointer aliases are analyzed by utilizing the synchronization primitives as a hint. For example, let us consider the situation where a program accesses a shared memory region. We are able to know whether multiple threads access the shared memory region simultaneously by checking whether the synchronization lock associated to the memory region is always held.

However, our goal is to design a TAL which permits implementation of synchronization primitives themselves. That is, it cannot have such language primitives and only atomic instructions provided by CPU (e.g., atomic swap and/or compare-and-swap instructions) are available. Thus, we have to track pointer aliases between multiple threads only with the atomic instructions.

3 Our Approach for Ensuring Safety of Shared Memory

This section describes our approach to the problems shown in Sec. 2. The key idea is to allow strong updates only in atomic memory operations and ensure the types of memory regions do not change before and after executing each atomic memory operation. Here, one atomic memory operation consists of one atomic instruction of CPU whose runtime effects on one processor are all visible or all invisible to others, and pseudo instructions that only affect types and have no runtime effects.

```
 1: { p -> exists(i).
 2:       { q -> data if [i == 0]}.
 3:       (i, q) }
 4: ( r1 : p )
 5: lock:
 6:    mov r2 <- 1
 7:    unpack r1
 8:    xchg [r1], r2
 9:    pack r1
10:    bne r2, 0, lock
11:    jmp unlock
12:
13: { p -> exists(i).
14:       { q -> data if [i == 0]}.
15:       (i, q),
16:   q -> data }
17: ( r1 : p )
18: unlock:
19:    unpack r1
20:    mov [r1] <- 0
21:    pack r1
22:    ...
```

Figure 1: Example of acquiring and releasing a spin lock

In our approach, for example, spin locks can be implemented as in Fig. 1. In the figure, line 1 to 4 and line 13 to 17 represents label types, which are given to all the labels and represent the conditions that should be satisfied when a control reaches the labels. For example, line 1 to 3 represents the heap type which represents the memory state when a control reaches to the label `lock` and line 4 represents the state of registers. More concretely, line 1 to 4 indicates that the register `r1` holds a certain integer value `p` (line 4) and there exists data at the address `p` (line 1) (Please note that `p`, `q` and `i` are singleton integer types [19, 14]). Here, we assume that the data is shared between threads. The data has the existential type whose base is the tuple consists of two integer values (`i` and `q`) and there exists data of the type `data` at the address `q` only if `i` is 0 (line 2 and 3). The first element of the tuple (`i`) represents a lock and the second element (`q`) represents a pointer to the data guarded by the lock which is inaccessible unless unpacking the existential type [18, 14].

Type checking of lock acquisition (`lock`) is performed as follows. First, the `unpack` pseudo instruction (line 7) unpacks the existential type of the data in

the address p. The heap type becomes as follows:

```
{ p -> (i, q), q -> data if [i == 0]}
```

Here, the type of the memory region at the address p is modified, but it passes the type check because unpack is a pseudo instruction.

Next, xchg (line 8) atomically exchanges the content of the address which is specified by the register r1 and that of the register r2. Then, the register r2 has the singleton type i and the heap type becomes as follows:

```
{ p -> (1, q), q -> data if [i == 0]}
```

Here, because we assume that xchg is atomic, it is unnecessary to revert the heap type to the original one.

Then, the pack pseudo instruction modified the heap type as follows (please note that the heap type { q -> data if [i == 0] } is not encapsulated because we know that i is not equal to 0):

```
{ p -> exists(i).
       { q -> data if [i == 0]}.
       (i, q)
  q -> data if [i == 0] }
```

Here, the type of the memory region at the address p is reverted to the original one. These instructions of line 7 to 9 make up one atomic operation.

Next, the bne instruction loops back to the label lock if i is not equal to 0 or jumps to the label unlock if i is equal to 0. This means that it tries to acquire the lock again if the lock is already acquired, or releases the lock otherwise. In this example, the lock is immediately released after acquisition but in fact the memory region at the address q can be accessed before releasing the lock. More concretely, when jumping to the label lock, the condition specified in its label type is satisfied because $i \neq 0$ and the heap type is equal to as follows:

```
{ p -> exists(i).
       { q -> data if [i == 0]}.
       (i, q) }
```

In addition, when jumping to the label unlock, the condition specified in its label type is also satisfied because $i = 0$ and the heap type is equal to as follows:

```
{ p -> exists(i).
       { q -> data if [i == 0]}.
       (i, q)
  q -> data }
```

On the other hand, type checking of lock release (unlock) is performed as follows. First, the unpack pseudo instruction (line 19) unpacks the existential type and modifies the heap type as follows (please note that the heap type { q -> data if [i == 0] } is merged with { q -> data } because the former is a subset of the latter.):

```
{ p -> (i, q), q -> data}
```

Next, mov (line 20) modifies the heap type as follows:

```
 1: { p -> exists(i).
 2:        { q -> data if [i == 0]}.
 3:        (i, q) }
 4: ( r1 : p )
 5: lock:
 6:    mov r2 <- 1
 7:    unpack r1
 8:    mov r3 <- [r1]
 9:    mov [r1] <- r2
10:    mov r2 <- r3
11:    pack r1
12:    bne r2, 0, lock
13:    jmp unlock
14:    ...
```

Figure 2: Another example of lock acquisition (wrong)

```
{ p -> (0, q), q -> data}
```

Finally, the pack pseudo instruction (line 21) modifies the heap type as follows:

```
{ p -> exists(i).
       { q -> data if [i == 0]}.
       (i, q) }
```

Please note again that the type of the memory region at the address p is reverted to the original one and the memory region at the address q is no longer accessible because it is packed to the existential type, unless reacquiring the lock. These instructions of line 19 to 21 make up one atomic memory operation.

Thus, we can express the spin locks that are memory safe even if they are used in the environments where multiple threads run concurrently.

Here, let us consider the slightly modified example as shown in Fig. 2. More specifically, the xchg instruction in Fig. 1 is replaced with the three mov instructions, that is, one atomic memory operation is divided into three atomic memory operations (line 7 to 8, line 9, and line 10 to 11). In this case, after the mov instruction at line 9, the heap type becomes as follows:

```
{ p -> (i, q), q -> data if [i == 0]}
```

At this point, the type checking fails because the type of the memory region at the address p is not reverted to the original existential type after the atomic operation.

In fact, the program of Fig. 2 is wrong because race conditions may occur between line 8 and 9.

4 Our Typed Assembly Language

This section explains the details of our TAL based on the approach of Sec. 3. The syntax of the abstract machine is shown in Fig. 3, and that of the types is shown in Fig. 4.

$$
\begin{array}{lrcl}
(register) & r & ::= & r_1 \mid r_2 \mid \ldots \mid r_n \mid sp \\
(operand) & o & ::= & d \mid r \mid [r+d] \\
(inst.) & \iota & ::= & mov\ o \leftarrow o \mid bcc\ o, o, o \\
 & & & \mid jmp\ o \mid push\ o \mid pop\ o \mid ret \mid cli \mid sti \\
 & & & \mid pushf \mid popf \mid iret \mid block \mid unblock \\
 & & & \mid pack_{[\overline{c}\mid\Psi]}\ o\ as\ \tau \mid unpack\ o\ with\ \Delta \\
(insts) & I & ::= & \cdot \mid \iota\,;I \\
(tuple) & t & ::= & \langle d, \ldots, d\rangle \mid pack_{[\overline{c}\mid\Psi]}.t \\
(value) & v & ::= & t \mid \forall\Delta.C.\Phi.i.i.I \\
(heap) & H & ::= & \cdot \mid \{d \mapsto v\}H \\
(registers) & R & ::= & \{r_1 \mapsto d, \ldots, r_n \mapsto d\} \\
(stack) & s & ::= & \cdot \mid d :: s \\
(processor) & P & ::= & (R, s, d_{pc}, d_{ipc}, d_{if}) \\
(state) & M & ::= & (H, \overline{P}, d_g)
\end{array}
$$

Figure 3: Syntax of the abstract machine (d represents an integer value)

$$
\begin{array}{lrcl}
(type\ var) & \alpha, \gamma, \epsilon, \rho & & \\
(type\ vars) & \Delta & ::= & \cdot \mid \alpha, \Delta \mid \ldots \\
(int.\ type) & i & ::= & \alpha \mid d \mid \ldots \\
(word\ type) & w & ::= & \gamma \mid i \mid \forall\Delta.C.\Phi.i.i \\
(tuple\ type) & \tau & ::= & \langle w, \ldots, w\rangle \mid \exists\Delta.C.\Psi.\tau \\
(heap\ type) & \Psi & ::= & \cdot \mid \{i \mapsto \tau\ if\ C\}\ \Psi \\
 & & & \mid (\epsilon\ if\ C)\Psi \\
(stack\ type) & \sigma & ::= & \rho \mid \cdot \mid w :: \sigma \\
(regs.\ type) & \Gamma & ::= & \{r_1 \mapsto w; \ldots, r_n \mapsto w;\} \\
(store\ type) & \Phi & ::= & (\Psi, \Gamma, \sigma) \\
(cop) & cc & ::= & < \mid \leq \mid \ldots \\
(cstrts.) & C & ::= & \cdot \mid i\ cc\ i, C \\
(constr.) & c & ::= & i \mid w \mid \Psi \mid \sigma
\end{array}
$$

Figure 4: Syntax of the types

(In fact, our TAL is a bit more complicated but this paper explains the simplified one for the sake of brevity and space limitation.) The type system of our TAL ensures the memory safety and control-flow safety even in the SMP/multi-core environments with interrupts.

One difference between our TAL and conventional TALs is that the state of the abstract machine (M) of our TAL consists of the heap (H), the states of processors ($\overline{P}$), and the atomic flag (d_g). The state of the processor (P) consists of the register file (R), the stack (s), the program counter (d_{pc}), the address of an interrupt handler (d_{ipc}), and the interrupt flag (d_{if}).

The atomic flag is just an integer value which indicates which processor performs an atomic memory operation. If there exists a process which performs an atomic memory operation, the flag is set to the processor ID of the processor (the processor ID is a unique number which is assigned statically to each processor). On the other hand, if there is no such processor, the flag is set to 0.

The atomic flag is manipulated by the *block* and *unblock* instructions. They indicate the beginning and the end of an atomic memory operation, respectively.

The reason why we introduced *block* and *unblock*, which do not exist in typical CPUs, is to uniformly deal with the atomic instructions of the various CPUs.

For example, the `xchg` instruction of the Intel architecture [11] can be expressed with a combination of *block*, *unblock* and memory operations, as follows:

$block$; $mov\ r_{tmp} \leftarrow [r_d]$; $mov\ [r_d] \leftarrow r_s$;
$mov\ r_s \leftarrow r_{tmp}$; $unblock$

The following is another example: `cmpxchg`.

(* cmpxchg *)
$block$; $beq\ r_a, [r_d]$, cmpxchg_eq;
$mov\ r_a \leftarrow [r_d]$; jmp cmpxchg_end
(* cmpxchg_eq *)
$mov\ [r_d] \leftarrow r_s$; jmp cmpxchg_end
(* cmpxchg_end *)
$unblock$

One of the benefits of the above representations is that they can be used for implementing wait-free data structures, as pointed out in [9].

The interrupt handler is an instruction sequence which is executed for handling interrupts. When an interrupt occurs, the program counter and the interrupt flag are pushed onto the stack and the interrupt handler runs.

The interrupt flag controls interrupts. More concretely, interrupts are disabled when the flag is set to 0. On the other hand, interrupts are enabled when the flag is set to a non-zero value even if the interrupt handler is executed.

The *cli* and *sti* instructions set and clear the interrupt flag, respectively. The *pushf* instruction pushes the interrupt flag to the stack, while the *popf* instruction pops a value from the stack and sets the interrupt flag to the value. We can achieve nested interrupt disabling and enabling with these instructions. The *iret* instruction is the instruction for returning from the interrupt handler; it sets the interrupt flag to the value stored in the stack and jumps to the address also stored in the stack. The *pack* and *unpack* instructions are pseudo instructions for introducing and eliminating existential types, respectively.

The other instructions are the same as ordinary assembly languages. Please note that arithmetic instructions are omitted in this paper for brevity. They can be formalized without large modification to the type system.

The types of our TAL are shown in Fig. 4. In the same ways as our previous works [14, 13, 15], they consist of the heap type for typing the heaps, the registers type

$$(H, (\ldots, P_i, \ldots), d_g) \mapsto_M (H', (\ldots, P'_i, \ldots), d'_g)$$
where $(H, P_i, d_g) \mapsto_{P,i} (H', P'_i, d'_g)$
$i = d_g$ (if $d_g \neq 0$)
any processor ID (otherwise)

Figure 5: Operational semantics: the abstract machine

$$(H, (R, s, d_{pc}, d_{ipc}, d_{if}), d_g) \mapsto_{P,pid} L'$$
$L' = (H, P', d_g)$
where $P' = (R, s', d_{ipc}, d_{ipc}, 0)$
$s' = d_{pc} :: d_{if} :: s$ (if $d_g = 0 \wedge d_{if} \neq 0$)

Figure 6: Operational semantics: interrupts

for typing the register files, and the stack type for typing the stacks. Please note that the types for representing variable length arrays [14] are omitted in this paper.

They differ from our previous works in two points. First, the integer types i that represent the atomic flag and the interrupt flag are added to the label type which represents the address of instructions ($\forall\Delta.C.\Phi.i.i$). Second, the constraints C can be specified to each element of the heap type which represents the heap. For example, the heap type $\{i \mapsto \tau \; if \; i \neq 0\}$ indicates that there exists a memory region at the address i and its type is τ only if $i \neq 0$. If it is unnecessary to specify any constraint, we can omit the constraints as $\{i \mapsto \tau\}$.

4.1 Operational Semantics

The operational semantics of our TAL is defined in Fig. 5, 6, 7 and 8. Fig. 9 defines the auxiliary functions.

The operational semantics of the whole abstract machine is define in Fig. 5. If the atomic flag is 0, one of the processors of the abstract machine takes a step. On the other hand, if the atomic flag is non-zero, the processor whose ID is equal to the atomic flag takes a step. That is, if one processor performs an atomic memory operation, the other processors do not execute instructions. From the viewpoint of memory consistency, this means that the abstract machine satisfies sequential consistency [1]. Sec. 5 discusses more relaxed memory consistency.

Although the operational semantics of the processor follows conventional typed assembly languages, it differs in handling the interrupt flag. More concretely, if the atomic flag (d_g) is 0 and the interrupt flag (d_{if}) is non-zero, then the interrupt handler (d_{ipc}) can be executed anytime (Fig. 6). Before executing the interrupt handler, the interrupt flag is set to 0, that is, the interrupts are disabled. In addition, the program counter (d_{pc}) and the interrupt flag (d_{if}) are pushed to the stack.

The operational semantics of the instructions is defined in Fig. 7 and 8. The *block* instruction sets the atomic flag to the self processor ID, while the *unblock* instruction clears the atomic flag to 0. The *cli* and *sti* instructions set the interrupt flag to 0 and 1, respectively. *pushf* pushes the interrupt flag to the stack, while *popf* pops the value from the stack and sets the interrupt flag to the value. The *iret* pops the stored program counter and the stored interrupt flag from the stack, sets the interrupt flag to the stored interrupt flag, and jumps to the stored program counter. The other instructions are the same as our previous TALs [13, 15].

$$(H, (R, s, d_{pc}, d_{ipc}, d_{if}), d_g) \mapsto_{P,pid} L'$$

if $H[d_{pc}] =$	then $L' =$
$mov\ o_1 \leftarrow o_2$	$(H', (R', s', d'_{pc}, d_{ipc}, d_{if}), d_g)$ where $S = (H, R, s)$ $d = get(S, o_1)$ $(H', R', s') = update(S, o_2, d)$
$jmp\ o$	$(H, (R, s, d, d_{ipc}, d_{if}), d_g)$ where $d = get((H, R, s), o)$
$bcc\ o_1, o_2, o_3$	$(H, (R, s, d, d_{ipc}, d_{if}), d_g)$ where $S = (H, R, s)$ if d_1 cc d_2 then $d = get(S, o_3)$ else $d = d'_{pc}$ $d_1 = get(S, o_1)$ $d_2 = get(S, o_2)$
$push\ o$	$(H, (R, s', d'_{pc}, d_{ipc}, d_{if}), d_g)$ where $S = (H, R, s)$ $s' = d :: s$ $d = get(S, o)$
$pop\ o$	$(H', (R', s'', d'_{pc}, d_{ipc}, d_{if}), d_g)$ where $s = d :: s'$ $S = (H, R, s')$ $(H', R', s'') = update(S, o, d)$
ret	$(H, (R, s', d, d_{ipc}, d_{if}), d_g)$ where $s = d :: s'$

where $d'_{pc} = d_{pc} + 1$

Figure 7: Operational semantics: instructions (1 of 2)

4.2 Typing Rules

The selected typing rules are shown in Fig. 10, 11, 13, 14 and 15. get_type and $update\_type$ are auxiliary functions for obtaining and updating the types of operands according to the specified heap and registers types, respectively (Fig. 12). The typing rules are decidable if the integer constraint solving $\Delta, C \models C$ is decidable.

There are two key points in the typing rules. First is that they statically keep track of the atomic and interrupt flags as the singleton integer types.

Second is that the shared memory regions, that is, the memory regions shared between the processors, and the interrupt handler and the interrupted program, are handled specially as the heap types Ψ_s and Ψ_b, respectively. The two heap types are introduced in order to achieve

$(H, (R, s, d_{pc}, d_{ipc}, d_{if}), d_g) \mapsto_{P,pid} L'$

if $H[d_{pc}] =$	then $L' =$
cli	$(H, (R, s, d'_{pc}, d_{ipc}, 0), d_g)$
sti	$(H, (R, s, d'_{pc}, d_{ipc}, 1), d_g)$
pushf	$(H, (R, s', d'_{pc}, d_{ipc}, d_{if}), d_g)$ where $s' = d_{if} :: s$
popf	$(H, (R, s', d'_{pc}, d_{ipc}, d'_{if}), d_g)$ where $s = d'_{if} :: s'$
iret	$(H, (R, s', d, d_{ipc}, d'_{if}), d_g)$ where $s = d :: d'_{if} :: s'$
$pack_{[\bar{c}\|\Psi]}$ *o as* τ	$(H', (R, s, d'_{pc}, d_{ipc}, d_{if}), d_g)$ where $H' = H\{d \mapsto t'\}$ $d = get((H, R, s), o)$ $t' = pack_{[\bar{c}\|\Psi]}.H(d)$
unpack o with Δ	$(H', (R, s, d'_{pc}, d_{ipc}, d_{if}), d_g)$ where $H' = H\{d \mapsto t'\}$ $d = get((H, R, s), o)$ $H(d) = pack_{[\bar{c}\|\Psi]}.t'$
block	$(H, (R, s, d'_{pc}, d_{ipc}, d_{if}), pid)$
unblock	$(H, (R, s, d'_{pc}, d_{ipc}, d_{if}), 0)$

where $d'_{pc} = d_{pc} + 1$

Figure 8: Operational semantics: instructions (2 of 2)

memory safe strong updates, which are essential for implementing the memory and multi-thread management facilities, as mentioned in Sec. 2.1.

The idea of tracking the heap types of the shared memory regions is to temporarily allow the strong updates only while an atomic memory operation is performed or the interrupts are disabled, as described in Sec. 3.

For example, the typing rule MOV indicates that the heap type of the shared memory between processors (Ψ_s) has to be included in that of the heaps updated with the mov instruction (denoted as $\Delta, C \vdash \Phi' \rightarrow \Psi_s$) if the atomic flag may have the value 0. In addition, the rule also indicates that the heap type of the shared memory between the interrupt handler and the interrupted program (Ψ_b) has to be included in that of the updated heaps ($\Delta, C \vdash \Phi' \rightarrow \Psi_b$) if the interrupt flag may be non-zero.

The typing rule UNBLOCK indicates that the heap type of the shared memory (Ψ_s) has to be included in that of the whole heaps because $unblock$ clears the atomic flag, that is, finishes an atomic memory operation and the other processors may access the shared memory. In other words, the strong updates are allowed after an atomic memory operation begins with $block$, but the heap types updated with the strong updates have to be reverted before the atomic memory operation finishes with $unblock$.

Please note that the number of addresses is fixed in the shared memory (Ψ_s), but an arbitrary number of memory regions can be handled by packing them to existentials.

$get((H, R, s), o) =$	
d	$(if\ o\ is\ d)$
$R(r)$	$(if\ o\ is\ r)$
$H(R(r))[c]$	$(if\ o\ is\ [r + c])$
$s[c]$	$(if\ o\ is\ [sp + c])$

$update((H, R, s), o, d) =$	
$H, R\{r \mapsto d\}, s$	$(if\ o\ is\ r)$
$H\{R(r) \mapsto v'\}, R, s$ where $v = H(R(r))$	$(if\ o\ is\ [r + c])$ $v' = v[c \mapsto d]$
$H, R, s[c \mapsto d]$	$(if\ o\ is\ [sp + c])$

Figure 9: Auxiliary functions for operational semantics

Brief descriptions of each typing rule are as follows. The rule STATE checks whether the heaps and the processors of the abstract machine state are well-typed. In addition, it also checks whether a part of the heaps satisfies the heap type of the shared memory between the processors (Ψ_s) if the atomic flag is 0. The rule PROCESSOR checks whether the registers, the stack, the instruction sequence ($H(d_{pc})$) pointed by the program counter d_{pc}, and the interrupt handler ($H(d_{ipc})$) are well-typed. In addition, it also checks whether a part of the heaps satisfies the heap type of the shared memory between the interrupt handler and the interrupted program (Ψ_b) if the interrupt flag d_{if} is non-zero. The rule HEAP checks whether each element of the heap can be typed as a tuple or an instruction sequence, the rule REGISTER checks whether each register is well-typed, and the rule STACK checks whether each element of the stack is well-typed.

The rule MOV updates the type of the operand o_1 with that of the operand o_2 and checks the following instructions I. In addition, as described above, it checks whether the heap type of the shared memory between the processors (Ψ_s) is included in the updated store type if it cannot be proved that the type of the atomic flag (i_g) is non-zero, and the heap type of the shared memory between the interrupt handler and the interrupt program (Ψ_b) is included in the updated store type if it cannot be proved that the type of the interrupt flag (i_i) is 0. Thus, the heap type of the shared memory is preserved even if other processors manipulate it or the interrupts occur.

The rule JMP checks whether the operand o has a label type and the store type Φ satisfies the store type Φ' specified in the label type (denoted as $\Delta, C \vdash \Phi \leq \Phi'$). In addition, it checks whether the constraints C' can be satisfied and the types of the atomic flag and the interrupt flag specified in the label type is consistent with the current state ($\Delta, C \models i_i = i'_i \wedge i_g = i'_g$).

The rule BCC first checks whether the operands o_1 and o_2 have integer types and the operand o_3 has a label type.

$$\frac{H \equiv H_1 \ldots H_n \qquad \Psi \equiv \Psi_1 \ldots \Psi_n \quad \vdash H_i : \Psi_i \qquad \Psi_i \vdash_H P_i : \Gamma_i, \sigma_i, d_g \quad d_g = 0 \Rightarrow \vdash H_s : \Psi_s \text{ where } H_s \subseteq H}{\vdash (H, \overline{P}, d_g) : (\Psi, \overline{(\Gamma, \sigma)})} \text{ (STATE)}$$

$$\frac{\begin{array}{c}\vdash H(d_{pc}) : \cdot.(\Psi, \Gamma, \sigma).d_{if}.d_g \quad \Psi \vdash s : \sigma \qquad \Delta \equiv \alpha, \bar{\gamma}, \epsilon, \rho \qquad \Psi' \equiv \Psi_b \epsilon \\ \Gamma' \equiv \{\overline{r_i \mapsto \gamma_i}\} \qquad \Phi' \equiv (\Psi', \Gamma', \rho) \\ \vdash H(d_{ipc}) : \forall \Delta. \cdot.(\Psi', \Gamma', \Phi'.\alpha.0 :: \alpha :: \rho).0.0 \\ d_{if} \neq 0 \Rightarrow \vdash H_b : \Psi_b \text{ where } H_b \subseteq H\end{array}}{\Psi \vdash_H (R, s, d_{pc}, d_{ipc}, d_{if}) : \Gamma, \sigma, d_g} \text{ (PROCESSOR)}$$

$$\frac{\begin{array}{c}\forall d \in Dom(H).\text{if } H(d) = t \text{ then } \vdash t : \Psi(d) \\ \text{else if } H(d) = \forall \Delta.C.\Phi.i_i.i_g.I \\ \text{then } \Psi(d) = \forall \Delta, C, \Phi, i_i, i_g \quad \Delta, C, \Phi, i_i, i_g \vdash I\end{array}}{\vdash H : \Psi} \text{ (HEAP)}$$

$$\frac{\vdash \Gamma \quad \forall r_i \in Dom(\Gamma).\Psi \vdash R(r_i) : \Gamma(r_i)}{\Psi \vdash R : \Gamma} \text{ (REGISTER)}$$

$$\frac{\Psi \vdash d_i : w_i}{\Psi \vdash d_1 :: \ldots :: d_n : w_1 :: \ldots :: w_n} \text{ (STACK)}$$

Figure 10: Typing rules for the abstract machine state (excerpt)

Next, it performs the same check as the rule JMP under the assumption that the branch is taken ($C \wedge i_1\ cc\ i_2$). Finally, it checks the following instructions under the assumption that the branch is not taken ($C \wedge \neg(i_1\ cc\ i_2)$).

The rule PUSH concatenates the type of the operand o to the stack type σ and checks the following instructions with the updated stack type. Unlike the rule MOV, it is unnecessary to check the heap type of the shared memory because the stacks are not included in the shared memory and $push$ does not modify the heap type. The rule POP removes the word type w from the top of the stack type $w :: \sigma$. In addition, it modifies the store type with respect to the operand o and checks the following instructions with the modified store type (Φ'). Unlike the rule PUSH, because pop may modify the heap type, it performs the same check as the rule MOV on the heap type of the shared memory. The rule RET removes the word type from the top of the stack type and checks whether it is a label type and the conditions specified in the label type are satisfied by the store type, the constraints, and the interrupt flag of the current state. As the rule PUSH, it is unnecessary to check the type of the shared memory because ret does not modify the heap type.

The rule BLOCK checks the following instructions under the assumption that the atomic flag type is 1. The rule UNBLOCK checks the following instructions under the assumption that the atomic flag type is 0. It also

$$\frac{\begin{array}{c}w \equiv get\_type(\Delta, C, \Phi, o_2) \\ \Phi' \equiv update\_type(\Delta, C, \Phi, o_1, w) \\ \Delta, C \not\models i_i = 0 \Rightarrow \Delta, C \vdash \Phi' \rightarrow \Psi_b \\ \Delta, C \not\models i_g \neq 0 \Rightarrow \Delta, C \vdash \Phi' \rightarrow \Psi_s \\ \Delta, C, \Phi', i_i, i_g \vdash I\end{array}}{\Delta, C, \Phi, i_i, i_g \vdash mov\ o_1, o_2;\ I} \text{ (MOV)}$$

$$\frac{\begin{array}{c}C'.\Phi'.i_i'.i_g' \equiv get\_type(\Delta, C, \Phi, o) \\ \Delta, C \vdash \Phi \leq \Phi' \\ \Delta, C \models C' \qquad \Delta, C \models i_i = i_i' \wedge i_g = i_g'\end{array}}{\Delta, C, \Phi, i_i, i_g \vdash jmp\ o} \text{ (JMP)}$$

$$\frac{\begin{array}{c}i_1 \equiv get\_type(\Delta, C, \Phi, o_1) \\ i_2 \equiv get\_type(\Delta, C, \Phi, o_2) \\ C'.\Phi'.i_i'.i_g' \equiv get\_type(\Delta, C, \Phi, o_3) \\ \Delta, C \wedge (i_1\ cc\ i_2) \vdash \Phi \leq \Phi' \\ \Delta, C \wedge (i_1\ cc\ i_2) \models C' \\ \Delta, C \wedge (i_1\ cc\ i_2) \models i_i = i_i' \wedge i_g = i_t' \\ \Delta, C \wedge \neg(i_1\ cc\ i_2), \Phi, i_i, i_g \vdash I\end{array}}{\Delta, C, \Phi, i_i, i_g \vdash bcc\ o_1, o_2, o_3;\ I} \text{ (BCC)}$$

$$\frac{\begin{array}{c}w \equiv get\_type(\Delta, C, (\Psi, \Gamma, \sigma), o) \\ \Delta, C, (\Psi, \Gamma, w :: \sigma), i_i, i_g \vdash I\end{array}}{\Delta, C, (\Psi, \Gamma, \sigma), i_i, i_g \vdash push\ o;\ I} \text{ (PUSH)}$$

$$\frac{\begin{array}{c}\Phi' \equiv update\_type(\Delta, C, (\Psi, \Gamma, \sigma), o, w) \\ \Delta, C \not\models i_i = 0 \Rightarrow \Delta, C \vdash \Phi' \rightarrow \Psi_b \\ \Delta, C \not\models i_g \neq 0 \Rightarrow \Delta, C \vdash \Phi' \rightarrow \Psi_s \\ \Delta, C, \Phi', i_i, i_g \vdash I\end{array}}{\Delta, C, (\Psi, \Gamma, w :: \sigma), i_i, i_g \vdash pop\ o;\ I} \text{ (POP)}$$

$$\frac{\begin{array}{c}\Delta, C \vdash (\Psi, \Gamma, \sigma) \leq \Phi \\ \Delta, C \models C' \qquad \Delta, C \models i_i = i_i' \wedge i_g = i_g'\end{array}}{\Delta, C, (\Psi, \Gamma, C'.\Phi.i_i'.i_g' :: \sigma), i_i, i_g \vdash ret} \text{ (RET)}$$

Figure 11: Typing rules for the ordinary instructions

checks whether the heap type of the shared memory (Ψ_s) is included in the store type Φ because other processors may access the shared memory after $unblock$.

The rule CLI checks the following instructions under the assumption that the interrupt flag type is equal to 0. The rule STI checks the following instructions under the assumption that the interrupt flag type is equal to 1. In addition, it also checks whether the heap type of the shared memory between the interrupt handler and the interrupted program (Ψ_b) is included in the store type Φ because interrupts may occur after sti.

The rule PUSHF concatenates the interrupt flag type i_i to the stack type and checks the following instructions with the stack type. As the rule PUSH, it is unnecessary to check the heap type of the shared memory Ψ_b because $pushf$ does not modify the heap type. The rule POPF removes the word type from the top of the stack type and checks the following instructions under the assumption

$get\_type(\Delta, C, (\Psi, \Gamma, \sigma), o) =$	
d	$(if\ o\ is\ d)$
$\Gamma(r)$	$(if\ o\ is\ r)$
w_d	$where\ \Delta, C \models i = \Gamma(r) \quad \Delta, C \models C'$ $\Delta, C \vdash \Psi = \{i \mapsto \langle \ldots, w_d, \ldots \rangle\ if\ C'\}\Psi'$ $(if\ o\ is\ [r+d])$
w_d	$where\ \sigma = \ldots :: w_d :: \ldots \quad (if\ o\ is\ [sp+d])$

$update\_type(\Delta, C, (\Psi, \Gamma, \sigma), o, w) =$	
(Ψ, Γ', σ)	$where\ \Gamma' = \Gamma\{r \mapsto w\}\ (if\ o\ is\ r)$
(Ψ'', Γ, σ)	$where\ \Delta, C \models i = \Gamma(r) \quad \Delta, C \models C'$ $\Delta, C \vdash \Psi = \{i \mapsto \langle \ldots, w_d, \ldots \rangle\ if\ C'\}\Psi'$ $\Psi'' \equiv \{i \mapsto \langle \ldots, w, \ldots \rangle\ if\ C'\}\Psi'$ $(if\ o\ is\ [r+d])$
(Ψ, Γ, σ')	$where\ \sigma = \ldots :: w_d :: \ldots$ $\sigma' = \ldots :: w :: \ldots \quad (if\ o\ is\ [sp+d])$

Figure 12: Auxiliary functions for typing rules

$$\frac{\Delta, C, \Phi, i_i, 1 \vdash I}{\Delta, C, \Phi, i_i, i_g \vdash block;\ I}\ (\text{BLOCK})$$

$$\frac{\Delta, C \vdash \Phi \to \Psi_s \qquad \Delta, C, \Phi, i_i, 0 \vdash I}{\Delta, C, \Phi, i_i, i_g \vdash unblock;\ I}\ (\text{UNBLOCK})$$

Figure 13: Typing rules for atomic memory operations

that the interrupt flag set is set to the removed word type. Unlike *pop*, *popf* does not modify the heap type. However, it is necessary to check whether the heap type of the shared memory between the interrupt handler and the interrupted program Ψ_b is included in the heap type Ψ if it cannot be proved that the interrupt flag type is equal to 0 because *popf* updates the interrupt flag. The rule IRET is basically a combination of the rule RET and POPF.

The rule PACK first checks whether the type of the operand o is equal to the type which is instantiated from the existential type specified in *pack* by substituting the type variables (Δ') with the types $\bar{c}$. Next, it checks whether the heap type Ψ_1 specified in the instruction is included in the heap type Ψ. Finally, it checks the following instructions with the store type which excludes Ψ_1 under the assumption that the operand o has the specified existential type. The rule UNPACK first checks whether the operand o has an existential type. Then, it unpacks the constraints, the heap type and the tuple type, and checks the following instructions with them.

5 Memory Consistency

Although the formalization of Sec. 4 contains the notion of atomic memory operations, it does not consider memory consistency (except for the sequential consistency). Roughly speaking, in the relaxed consistency models, effects of memory operations performed by one processor can be observed in a different order by the other processors. Thus, memory barrier operations have to be used explicitly for ensuring consistency of shared memory. For example, the release consistency model, which is adopted by the many recent CPUs, is equipped with two barriers: *acquire*, which ensures that all the processors do not observe effects of all its succeeding operations, and *release*, which ensures that all the processors observe effects of all its preceding operations [1].

From the viewpoint of the release consistency, all we have to do is to figure out when the two barriers are necessary. More specifically, *acquire* has to be performed when the *unpack* operation extracts memory regions from existential types in shared memory, and *release* has

$$\frac{\Delta, C, \Phi, 0, i_g \vdash I}{\Delta, C, \Phi, i_i, i_g \vdash cli; I}\ (\text{CLI})$$

$$\frac{\Delta, C \vdash \Phi \to \Psi_b \qquad \Delta, C, \Phi, 1, i_g \vdash I}{\Delta, C, \Phi, i_i, i_g \vdash sti; I}\ (\text{STI})$$

$$\frac{\Delta, C, (\Psi, \Gamma, i_i :: \sigma), i_i, i_g \vdash I}{\Delta, C, (\Psi, \Gamma, \sigma), i_i, i_g \vdash pushf;\ I}\ (\text{PUSHF})$$

$$\frac{\Delta, C, (\Psi, \Gamma, \sigma), i', i_g \vdash I \quad \Delta, C \not\models i' = 0 \Rightarrow \Delta, C \vdash \Psi \supseteq \Psi_b}{\Delta, C, (\Psi, \Gamma, i' :: \sigma), i_i, i_g \vdash_{\Psi_s} popf;\ I}\ (\text{POPF})$$

$$\frac{\begin{array}{c}\Delta, C \vdash (\Psi, \Gamma, \sigma) \leq \Phi \quad \Delta, C \not\models i_1 = 0 \Rightarrow \Delta, C \vdash \Psi \supseteq \Psi_b \\ \Delta, C \models C' \qquad \Delta, C \models i_1 = i_2 \wedge i_g = i'_g\end{array}}{\begin{array}{c}\Delta, C, (\Psi, \Gamma, (C', \Phi, i_2, i'_g) :: i_1 :: \sigma), i_i, i_g \\ \vdash iret\end{array}}\ (\text{IRET})$$

Figure 14: Typing rules related to the interrupts

$$\frac{\begin{array}{c}\Phi \equiv (\Psi, \Gamma, \sigma) \qquad \tau \equiv \exists \Delta'.C'.\Psi'.\tau' \\ \Delta, C \vdash \Psi = \Psi_1\Psi_2 \qquad \Delta, C \vdash \Psi_1 = \Psi'[\bar{c}/\Delta'] \\ i \equiv get\_type(\Delta, C, (\Psi, \Gamma, \sigma), o) \\ \Delta, C \vdash \Psi_2 = \{i \mapsto \tau'[\bar{c}/\Delta']\}\Psi'_2 \\ \Phi' \equiv (\{i \mapsto \tau\}\Psi'_2, \Gamma, \sigma) \\ \Delta, C \models C'[\bar{c}/\Delta'] \qquad \Delta, C, \Phi', i_i, i_g \vdash I\end{array}}{\Delta, C, \Phi, i_i, i_g \vdash pack_{[\bar{c}|\Psi_1]}\ o\ as\ \tau; I}\ (\text{PACK})$$

$$\frac{\begin{array}{c}\Phi \equiv (\Psi, \Gamma, \sigma) \qquad i \equiv get\_type(\Delta, C, \Phi, o) \\ \Delta, C \vdash \Psi = \{i \mapsto \exists \Delta'.C'.\Psi'.\tau'\}\Psi'' \\ \Psi_1 \equiv \{i \mapsto \tau''\}\Psi'' \qquad C'' \equiv C'[\Delta''/\Delta'] \\ \Psi_2 \equiv \Psi'[\Delta''/\Delta'] \qquad \tau'' \equiv \tau'[\Delta''/\Delta'] \\ \Delta\Delta'', C \wedge C'', (\Psi_1\Psi_2, \Gamma, \sigma) \vdash I\end{array}}{\Delta, C, \Phi, i_i, i_g \vdash unpack\ o\ with\ \Delta''; I}\ (\text{UNPACK})$$

Figure 15: Typing rules related to the existential types

to be performed when the $pack$ operation encapsulates them to shared memory. For example, in Fig. 1, $acquire$ is necessary between line 7 and 8, and $release$ is necessary between line 20 and 21. Please note that the barriers are unnecessary at line 9 and 19 because no memory regions are encapsulated or extracted. In addition, please also note that the $acquire$ and `xchg` operations, and the $release$ and `mov` operations can be performed atomically in the recent CPUs (e.g., the Intel Architecture [11]).

6 Related Work

Although there exist several works [6, 12, 8, 7] that deal with synchronization mechanisms at the level of high-level programming languages, their main goal is to prevent race conditions, deadlocks and so on, and they assume the synchronization mechanisms as language primitives. Therefore, they cannot be used for implementing the synchronization mechanisms themselves, while our TAL can be used to implement the mechanisms only relying on atomic instructions of CPU. Cyclone introduced a swap operation for pointers as a language primitive in order to keep track of aliases by ensuring uniqueness of pointers [9]. Our TAL is more expressive in the sense that the swap operation can be implemented as a combination of $block$, $unblock$ and memory operations.

Vasconcelos et al. presented a variant of TAL for supporting synchronization locks in the SMP/multi-core environments [17]. However, their TAL treats the locks and the lock operations as the language primitives, so it cannot be used for implementing the locks themselves and ensuring their memory safety. On the other hand, our TAL is expressive enough to directly implement the locks and their lock operations as shown in Sec. 3. Moreover, in their TAL, thread cloning is also provided as the language primitives, while we are able to implement the multi-thread management facility in our TAL (by incorporating our previous works [14, 15]).

Feng et al. [4] showed an approach of manually verifying properties of synchronization primitives under the existence of the interrupts by using a proof assistant. While our approach verifies simple type safety (the memory safety and the control-flow safety) mechanically through type checking, their approach can be applied to verification of a more wide range of properties. However, SMP is not considered explicitly in their approach.

7 Conclusion

This paper presented a typed assembly language which is expressive enough to implement synchronization primitives in the SMP/multi-core environments with the CPU hardware interrupts. This paper also illustrated how to implement the spin locks, which are essential for implementing OS kernels. Our future work is to formalize memory consistency models (by utilizing, e.g., [3, 5]) and implement more complex synchronization primitives (e.g., readers-writer locks and read-copy-update).

References

[1] ADVE, S. V., AND GHARACHORLOO, K. Shared memory consistency models: A tutorial. *IEEE Comp. 29*, 12 (1996), 66–76.

[2] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M., BECKER, D., EGGERS, S., AND CHAMBERS, C. Extensibility, safety and performance in theh SPIN operating system. In *Proc. of SOSP'95* (1995), pp. 267–284.

[3] BOUDOL, G., AND PETRI, G. Relaxed memory models: an operational approach. In *Prof. of POPL'09* (2009), pp. 392–403.

[4] FENG, X., SHAO, Z., DONG, Y., AND GUO, Y. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proc. of PLDI'08* (2008), pp. 170–182.

[5] FERREIRA, R., FENG, X., AND SHAO, Z. Parameterized Memory Models and Concurrent Separation Logic. *Programming Languages and Systems, LNCS 6012* (2010), 267–286.

[6] FLANAGAN, C., AND ABADI, M. Object types against races. In *Proc. of CONCUR'99* (1999), pp. 288–303.

[7] FLANAGAN, C., AND FREUND, S. N. Type inference against races. *Sci. Comput. Program. 64*, 1 (2007), 140–165.

[8] GROSSMAN, D. Type-safe multithreading in cyclone. In *Proc. of TLDI'03* (2003), pp. 13–25.

[9] HICKS, M., MORRISETT, G., GROSSMAN, D., AND JIM, T. Experience with safe manual memory-management in cyclone. In *Prof. of ISMM'04* (2004), pp. 73–84.

[10] HUNT, G. C., LARUS, J. R., ABADI, M., AIKEN, M., BARHAM, P., FÄHNDRICH, M., HAWBLITZEL, C., HODSON, O., LEVI, S., MURPHY, N., STEENSGAARD, B., TARDITI, D., AND ZILL, T. W. B. An overview of the Singularity project. Tech. Rep. MSR-TR-2005-135, Microsoft Corporation, 2005.

[11] INTEL CORPORATION. Intel® 64 and IA-32 Architectures Software Developer's Manuals. http://www.intel.com.

[12] IWAMA, F., AND KOBAYASHI, N. A new type system for jvm lock primitives. In *Proc. of ASIA-PEPM'02* (2002), pp. 71–82.

[13] MAEDA, T. *Writing an Operating System with a Strictly Typed Assembly Language*. PhD thesis, University of Tokyo, 2006.

[14] MAEDA, T., AND YONEZAWA, A. Writing practical memory management code with a strictly typed assembly language. In *Proc. of SPACE'06* (2006).

[15] MAEDA, T., AND YONEZAWA, A. Writing an OS Kernel in a Strictly and Statically Typed Language. *Formal to Practical Security, LNCS 5458* (2009), 181–197.

[16] MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems 21*, 3 (1999), 528–569.

[17] VASCONCELOS, V. T., AND MARTINS, F. A multithreaded typed assembly language. In *Proc. of TV'06* (2006), pp. 133–141.

[18] WALKER, D., AND MORRISETT, G. Alias types for recursive data structures. In *Proc. of TIC'00* (2000).

[19] XI, H., AND PFENNING, F. Dependent types in practical programming. In *Proc. of POPL'99* (January 1999), pp. 214–227.

[20] YANG, J., AND HAWBLITZEL, C. Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System. In *Proc. of PLDI'10* (2010).