Using Lightweight Theorem Proving in an Asynchronous Systems Context*

Matthew Danish and Hongwei Xi

Boston University Computer Science
111 Cummington Mall
Boston, MA 02215

Abstract. As part of the development of a new real-time operating system, an asynchronous communication mechanism, for use between applications, has been implemented in a programming language with an advanced static type system. This mechanism is designed to provide desired properties of asynchronicity, coherency and freshness. We used the features of the type system, including linear and dependent types, to represent and partially prove that the implementation safely upheld coherency and freshness. We believe that the resulting program code forms a good example of how easily linear and dependent types can be applied in practice to prove useful properties of low-level concurrent systems programming, while leaving no traces of runtime overhead.

1 Introduction

The Terrier [7] project focuses on doing low-level, OS-level systems programming while taking advantage of a dependently typed programming language named ATS [25]. The purpose of this project is to identify effective, practical means to create safer, more reliable systems through use of advanced type system features in programming languages. We are also interested in the implications of having powerful programming language tools available, and the effects on plausible system design. For example, Terrier moves much of the responsibility for program safety back out onto the programs themselves, rather than relying strictly on run-time checks or hardware protection mechanisms. For another, the Terrier program model is one in which asynchronous events play a central role in program design. These two shifts in thinking put more burden on the programmer—a burden that we expect to lighten through language-level assistance—but they also open up more flexibility in potential program design that we hope will enable higher performance and more naturally-written code in difficult problem domains.

ATS is a language with the goal of bringing together formal specification and practical programming. The core of ATS is an ML-like functional programming language which is compiled into C. The type system of ATS combines dependent and linear types to permit sophisticated reasoning about program behavior and

* This research is supported partly by NSF grant CCF-1018601.

J.M. Badger and K.Y. Rozier (Eds.): NFM 2014, LNCS 8430, pp. 158–172, 2014.

the safety of resource usage. The design of ATS provides close coupling of type-safe functional code and low-level C code, allowing the programmer to decide the balance between specification and speed. The ATS compiler can generate code which does not require garbage collection nor any other special run-time support, making it suitable for bare metal programming.

Using ATS, we have generated C code which links into our kernel to provide several critical components. We also encourage the use of ATS to help ensure the safety and correctness of programs that run under the OS. For example, programs that wish to communicate with one another are provided with libraries written in ATS which implement protocols that have been statically checked for safety and correctness.

One of those protocols that we implemented is Simpson's "four slot fully asynchronous communication mechanism" [20]. It is a shared memory communication protocol which was cleverly designed by its author to pack some desirable properties into just a few lines of code. It allows one-way, pool-based transmission of data without any synchronization delays between reader and writer. The communication medium is the normal shared memory that is found in most computer systems. The mechanism offers the properties of freshness and coherency, but not history. For example, you could imagine a bulletin board on which one person posts flyers while the other person reads, as shown in figure 1. This mechanism ensures that the reader only sees the latest, complete, coherent postings on the board.

The ATS implementation of the four slot mechanism is compact, efficient and shows how strongly specified types can provide useful assurances at a low-level without intruding into run-time performance of critical code or requiring voluminous quantities of proof-writing.

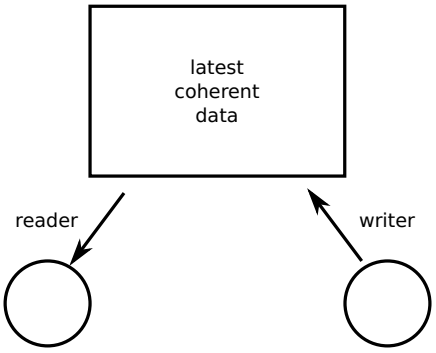

Fig. 1. Abstract depiction of four slot mechanism

1.1 The Four Slot Mechanism

The scenario for the four slot mechanism starts by assuming that there is a writer program which wishes to convey some information to a reader program. Furthermore, the information is able to be encoded into an arbitrary, fixed number of bytes, and there is a shared memory space large enough for at least four copies of the data to be stored, plus a few more bytes for state variables.

The four slot mechanism works by opening up space in memory for these four "slots" of data, usually in the form of an array. Coherency is ensured through program logic which keeps the reader and the writer apart: each has their own slot to operate upon, and further analysis will show that the mechanism prevents them from touching the same slot at the same time.

To achieve this property, the original four slot mechanism relies on several pieces of shared state, and it assumes that individual bits may be manipulated atomically. That is, all simultaneously accesses to a single bit will appear to have a definite ordering, either way [16]. In practice, atomic operations are offered at machine word sizes [2], not at the individual bit level, but the logic remains the same.

The shared state variables are used for coordination, by both reader and writer programs:

- The atomic bit variable named "reading" or R is intended to roughly indicate which side of the mechanism the reader program is currently using.
- The atomic bit variable named "latest" or L is intended to indicate which side of the mechanism was updated most recently.
- The bit array named "slot" further drills down on the specific slot of the array to be used. You could also choose to split this into two variables "slot0" and "slot1" for analysis purposes.
- Two bits are then used to index into the shared four-member array of data slots.

There are also private state variables which may only be accessed from within each program. Since they are symmetrical, we use a consistent naming scheme for them:

- A bit variable named "pair" which in the writer is named w_p and in the reader is named r_p.
- A bit variable named "index" which in the writer is named w_i and in the reader is named r_i.

Together, these private variables are used to index into the shared array, and that usage is denoted as $\texttt{write\_data}\,(w_p, w_i, item)$ or $item \leftarrow \texttt{read\_data}\,(r_p, r_i)$.

The diagram in figure 2 shows how the data flows in this protocol. A program obtains its "pair" from either the "reading" or "latest" atomic variable. It then uses the "pair" to pick an "index" from the "slot" array. It then uses the "pair" and "index" to select one of the four slots to work on.

In example diagram you can see that the reader has selected $0, 1$ and the writer has selected $1, 1$. Visually, you can see that they are able to independently read and write without conflict.

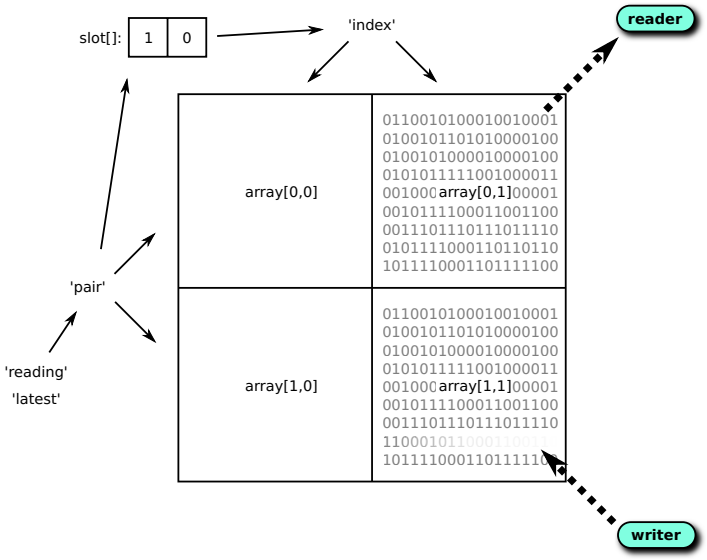

Fig. 2. Four Slot Mechanism

On the other hand, the diagram in figure 3 shows a case where the reader and writer are trampling over each other, likely causing data corruption. A properly working four slot mechanism will never allow this to occur.

The pseudocode for the four slot mechanism is shown in figure 4 and it is simple enough for a quick walk-through. For the first step of the writer, WS1, it reads the value of R, negates it, and stores it into the private variable w_p. For the second step, the writer reads a bit from the $slot[]$ array, indexed by w_p, negates it, and then uses that as the value of w_i. With both w_p and w_i, the writer is now ready to perform the actual write. Finally, the writer updates the shared state by writing its values of w_i, w_p into $slot[]$ and L respectively.

The reader also takes 5 steps which are aimed at obtaining values of r_p, r_i from shared state. However, the reader gets its value of r_p from the L variable, and it "stakes a claim" to that pair by writing it into the R variable. Then it finds out which slot is most up-to-date, reads the data, and returns it.

2 Coherency

A cursory inspection of the pseudocode should reveal that it easily transmits a piece of data if the two programs run back-to-back with no overlap. But that is not very interesting. The real difficulty comes when you accept that the two programs may arbitrarily interleave with one another. Figure 5 is an annotated example of one possible interleaving, where writer and reader steps shown on the same line are happening in parallel:

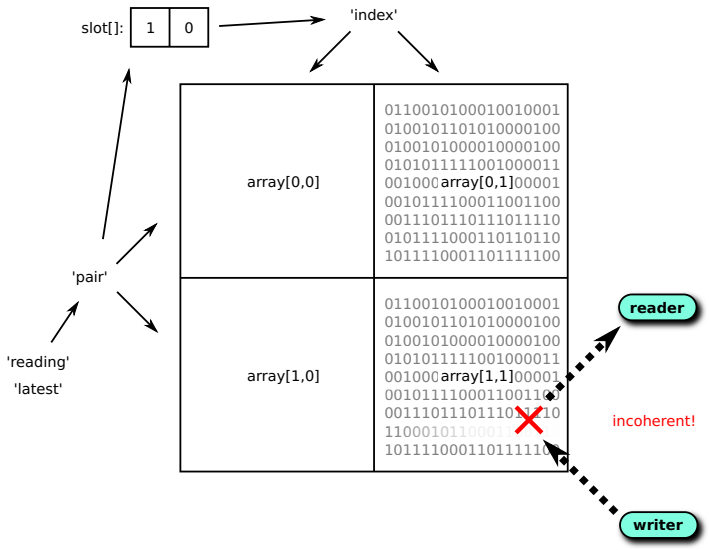

Fig. 3. Should not happen

```
WS1  w_p ← ¬R
WS2  w_i ← ¬slot[w_p]
WS3  write_data(w_p, w_i, item)
WS4  slot[w_p] ← w_i
WS5  L ← w_p
```

```
RS1  r_p ← L
RS2  R ← r_p
RS3  r_i ← slot[r_p]
RS4  item ← read_data(r_p, r_i)
RS5  return item
```

Fig. 4. Four slot mechanism pseudocode

– Suppose $L = 1$ and $R = 0$	
WS1 $w_p \leftarrow \neg R$	RS1 $r_p \leftarrow L$
– Now $w_p = r_p$	
WS2 $w_i \leftarrow \neg slot[w_p]$	RS2 $R \leftarrow r_p$
	RS3 $r_i \leftarrow slot[r_p]$
– And $w_i \neq r_i$	
WS3 write_data$(w_p, w_i, item)$	RS4 $item \leftarrow$ read_data(r_p, r_i)
– ...	

Fig. 5. Example of interleaving

You can see that when both programs reach their step 2, it is the case that $w_p = r_p$. But the design of the protocol, at this point, ensures that the writer picks the opposite index from the reader, so that $w_i \neq r_i$. Therefore the writer and the reader do not access the same data slot at the same time. This is just one

case, and we needed to show that the mechanism always respects a coherency property. More specifically, we looked at the "dangerous" steps WS3 and RS4, where the real data transfers take place, and needed to show that those two steps would never conflict.

Theorem 1 (Coherency). *The writer and the reader do not access the same data slot at the same time. More precisely, this assertion must be satisfied at potentially conflicting program points* WS3 *and* RS4*:*

$$w_p \neq r_p \vee w_i \neq r_i$$

or, the program points must be shown to be non-conflicting.

Problem is, w_p and r_p (as well as w_i and r_i) are private variables in separate programs. Therefore, dynamic or run-time checking was out of the question. So we turned to static checking encoded using dependent types. To find relevant properties, we looked at various "points of interaction" where the two programs might affect each other through atomic shared variables.

Recall that overlapping atomic operations will appear to occur in a definite ordering. Therefore, points of interaction involving atomic variables $R, L, slot\,[]$ can tell us facts about unseen state. Consider the orderings shown in figure 6.

$$\left.\begin{array}{ll} \mathsf{RS2} & R \leftarrow r_p \\ \mathsf{WS1} & w_p \leftarrow \neg R \end{array}\right\} w_p \neq r_p \text{ at } \mathsf{WS1}$$

$$\left.\begin{array}{ll} \mathsf{WS1} & w_p \leftarrow \neg R \\ \mathsf{RS2} & R \leftarrow r_p \end{array}\right\} w_p \stackrel{?}{=} r_p \text{ at } \mathsf{WS1}$$

Fig. 6. Two alternative orderings of WS1 and RS2

We do not know ahead of time whether WS1 or RS2 will occur first. But we do know the consequences of each ordering. In the first case, we can see that the writer will harmlessly pick the opposite value of "pair" from the reader. But in the second case, we have no idea what the values of w_p, r_p are. That tells us an important property:

Property 1. If $w_p = r_p$ at WS1 then WS1 precedes RS2.

Further, by transitivity,

Property 2. If WS1 precedes RS2 then it also precedes RS3.

Recall that RS3 is the step where the reader obtains its value of r_i by reading $slot\,[r_p]$. And as the data dependency graph of figure 7 shows, it is the writer which is in control of the value of $slot\,[r_p]$. Therefore, intuitively, the writer has enough knowledge of the value of r_i to be able to pick a w_i of the opposite value.

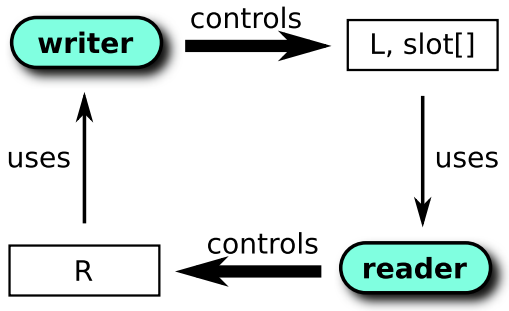

Fig. 7. Data dependencies

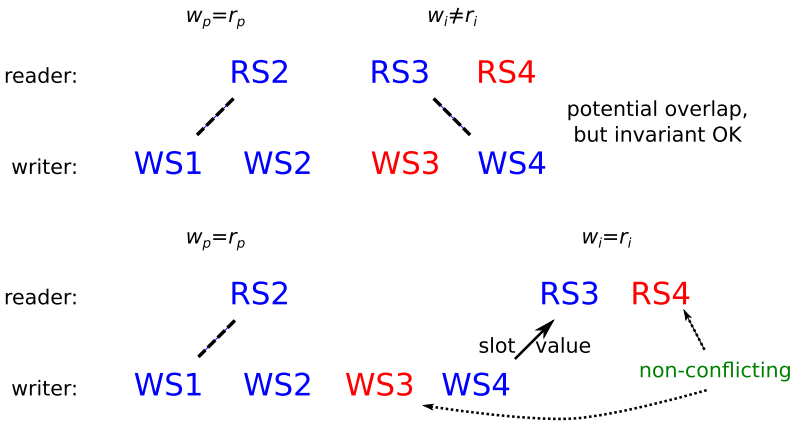

Fig. 8. Interaction between RS3 and WS4

This can be seen more precisely by examining the point of interaction between RS3 and WS4 via the *slot* [] array, as shown in figure 8, where the $w_p = r_p$ case is assumed. In this case, the only way that $w_i = r_i$ is if WS4 precedes RS3. By itself, it may seem to violate the coherency assertion but, in fact, the potentially conflicting program points WS3 and RS4 are non-conflicting because they are separated by an atomic operation with definite ordering. This is indicated on the diagram by the solid arrow.

3 Encoding the Proof

3.1 `Write`

Each step in the program is encoded into an ATS function prototype along with its types. The ATS code will be explained as we go.

WS1.

$$w_p \leftarrow \neg R$$

```
absview ws1_read_v (R: bit, rstep: int, rp: bit)

fun get_reading_state ():
  [rstep: nat]
  [R, rp: bit | R == rp || (R <> rp ==> rstep < 2)]
  (ws1_read_v (R, rstep, rp) | bit R)
```

For WS1 we have a kind of datatype definition as well as a function prototype. The datatype is abstract which means it does not have a real form in the output. That works because it is intended to be just a property which is erased by compilation. In ATS, properties which are linear are called "views" [26] and that just means that the property, after being introduced, must be consumed once and exactly once. They behave like a "resource" [24].

In this case, the `ws1_read_v` is an abstract view intended to represent the concept that we have observed the value of R at this particular moment, and this point of interaction tells us a certain fact about the reader, that is true until consumed.

The function simply reads the value of R and returns it, but it also gives us some properties about R which exist in the so-called "static" world [6] as opposed to the "dynamic" world. In ATS, there is a syntactic division between the static world and the dynamic world, represented by the vertical bar in concrete syntax. The static world can be more abstract, where you may encode relations that you do not want to appear in the final output, while the dynamic world is the practical side which will eventually be formed into the C output of the compiler.

The two are connected through use of indexed types. Here we see various type indices: `rstep`, `R`, `rp`. The index `R` represents the shared state variable R, `rp` represents the hidden private state variable r_p, and `rstep` represents the program counter of the reader program in an abstract form.

ATS allows some specifications about the values of these indices to be made in a convenient, set-like notation. In this case, we have made the following static assertion: that `R` is equal to `rp`, or, if `R` is not equal to `rp`, then `rstep` is less than 2.

Then we have returned the linear property indexed by these three, and on the dynamic side we have returned the actual bit value which is linked to the index `R`.

WS2.

$$w_i \leftarrow \neg slot\,[w_p]$$

```
absview ws2_slot_v (s: bit, rp: bit, ri: bit)

fun get_write_slot_index {R, wp, rp: bit} {rstep: nat} (
    pfr: !ws1_read_v (R, rstep, rp) | wp: bit wp
  ): [s, ri: bit | (rstep < 3 && wp == rp) ==> s == ri)]
    (ws2_slot_v (s, rp, ri) | bit s)
```

Again, we need a linear property—which here represents the value found in the slot array as well as some facts learned from that point of interaction. In this

case, `s` represents the slot value in memory s, `rp` is a stand-in for r_p, and `ri` for r_i.

In order to call this function for step 2, we need evidence that the reading state has been examined, and that evidence is provided by the `ws1_read_v` view. By default, a linear property like this is consumed and not usable again, but we want this particular property to be reproduced for later use, because we will want to make some statements about R in later steps. Now, one way to reproduce the property would be to explicitly return it again, but ATS provides the `!` operator as a bit of syntactic sugar to help make that common case convenient.

On the dynamic side, we need to pass the value of `wp` because that is used as an index into the slot array.

The return value is a view of the value of the slot that we received, as well as the actual value itself, combined with another fact about the relationship between the variables: if `rstep` was less than 3 and `wp` was equal to `rp`, then we know that `s` will be equal to `ri`.

The assertion `wp == rp ==> s == ri` is a simple statement about array access. The reader program performs $r_i \leftarrow slot\,[r_p]$, and if $w_p = r_p$ then the writer program will see the slot value s which is the same as r_i.

The reason why we have to check if `rstep` was less than 3 is because it might be possible for the reader to get to step 3 and then block for a long time. If the reader is blocking for a long time, it could have a stale value of r_i sitting around, a potentially conflicting value. By guarding against that, we know that the reader has yet to pick a value of r_i, and so when it does, it will be equal to the value of s.

WS3.

$$\texttt{write\_data}\,(w_p, w_i, item)$$

```
fun{a: t@ype} write_data
  {R, s, wp, wi, rp, ri: bit | wp <> rp || wi <> ri} {rstep: nat} (
    pfr: !ws1_read_v (R, rstep, rp),
    pfs: !ws2_slot_v (s, rp, ri) |
    wp: bit wp, wi: bit wi, item: a
  ): void
```

Armed with the facts that we have learned about w_p, w_i, r_p, r_i from the points of interaction, we are ready to use the function which does the actual work of writing data into memory. This function has theorem 1 (Coherency) encoded into its type: `wp <> rp || wi <> ri`, and therefore can only be called if this assertion can be statically proven, or else it results in a type error.

The ATS typechecker uses the Z3 SMT solver [8] on the facts that it has learned about these variables, and can discharge this assertion without further work by the programmer.

WS4.

$$slot\,[w_p] \leftarrow w_i$$

```
absview ws4_fresh_v (p: bit)

fun save_write_slot_index {s, wp, wi, rp, ri: bit | wi <> s} (
    pfs: ws2_slot_v (s, rp, ri) | wp: bit wp, wi: bit wi
  ): (ws4_fresh_v wp | void)
```

After the write is complete, we need to update the shared state variables so that subsequent reads will obtain the new value. This is a somewhat more minor concern than coherency, but still encoded enough to be sure that it is done properly. This step consumes the `ws2_slot_v` resource which is no longer valid or needed. It returns a new view which I have labeled `ws4_fresh_v`, which acts as an obligation to update the remaining shared state.

WS5.

$$L \leftarrow w_p$$

```
fun save_latest_state
  {R, rp, wp: bit | wp <> R} {rstep: nat} (
    pfr: ws1_read_v (R, rstep, rp),
    pff: ws4_fresh_v wp |
    wp: bit wp
  ): void
```

The final step cleans up, consuming both remaining views, as they will both become invalid after this step. A final check is added to ensure that the new value of L will not be equal to the old value of R.

Putting it Together. The code for the `write` operation, alongside the pseudocode, is shown in figure 9. The real code is largely similar to the pseudocode, and each line is implemented through ATS's close integration with a small amount of C code which performs the low-level atomic operations and memory copying.

```
val (pfr | R) = get_reading_state ()                    WS1  w_p ← ¬R
val wp = not R

val (pfs | s) = get_write_slot_index (pfr | wp)          WS2  w_i ← ¬slot[w_p]
val wi = not s

val _ = write_data (pfr, pfs | wp, wi, item)            WS3  write_data(w_p, w_i, item)

val (pff | _) = save_write_slot_index (pfs | wp, wi)    WS4  slot[w_p] ← w_i

val _ = save_latest_state (pfr, pff | wp)               WS5  L ← w_p
```

Fig. 9. `write`

3.2 Read

Again, each step in the program is encoded along with a type.

RS1.

$$r_p \leftarrow L$$

```
absview rs1_latest_v (L: bit)

fun get_latest_state (): [L: bool] (rs1_latest_v L | bit L)
```

The value of L is returned along with a linear proposition stating that it was seen.

RS2.

$$R \leftarrow r_p$$

```
absview rs2_read_v (R: bit)

fun save_reading_state {L, rp: bit | L == rp} (
    pf: rs1_latest_v L | rp: bit rp
  ): [R: bit | R == rp] (rs2_read_v R | void)
```

With the proposition in hand, we show that we are saving the correct value in the R shared variable. In return, we learn the fact that $r_p = R$ now.

RS3.

$$r_i \leftarrow slot\,[r_p]$$

```
absview rs3_slot_v (s: bit, wp: bit, wi: bit)

fun get_read_slot_index {R, rp: bit} (
    pf: rs2_read_v R | rp: bit rp
  ): [s, wp, wi: bool | wp == rp ==> s == ~wi]
     (rs3_slot_v (s, wp, wi) | bit s)
```

The `rs2_read_v` is consumed to prove that we are using the pair corresponding to R. In return, we gain a property with a constraint based on simple facts about arrays as well as the behavior of the `write` program. It stipulates that if both reader and writer access $slot\,[r_p]$ when $r_p = w_p$, then the writer's w_i will be the opposite of whatever value is found in $slot\,[r_p]$.

RS4.

$$item \leftarrow \texttt{read\_data}\,(r_p, r_i)$$

```
fun{a: t@ype} read_data {rp, ri, wp, wi: bool | wp <> rp || ri <> wi} (
    pf: rs3_slot_v (ri, wp, wi) | p: bit rp, i: bit ri
  ): a
```

Finally, the last interesting step of `read` is the one that requires the coherency theorem to be satisfied, `wp <> rp || ri <> wi`.

Putting it Together. Figure 10 shows the code and much like before, it is close to the pseudocode, and yet compiles down into efficient C. The ATS typechecker is powerful enough to automatically solve the constraints.

```
val (pfl | rp) = get_latest_state ()
val (pfr | _) = save_reading_state (pfl | rp)
val (pfs | ri) = get_read_slot_index (pfr | rp)
val item = read_data (pfs | rp, ri)
```

RS1 $r_p \leftarrow L$

RS2 $R \leftarrow r_p$

RS3 $r_i \leftarrow slot\,[r_p]$

RS4 $item \leftarrow \texttt{read\_data}\,(r_p, r_i)$

Fig. 10. `read`

4 Related Work

4.1 The Four Slot Mechanism

Simpson developed a technique called "role model analysis" [21] and then applied it to his four slot mechanism [22] to verify properties of coherency and freshness. Henderson and Paynter [12] created a formal model of the four slot mechanism in PVS and used it to show that it was atomic under certain assumptions about interleaving. Rushby [19] used model checking to verify coherency and freshness in the four slot mechanism but also found the latter can only be shown if the control registers are assumed to be atomic. Our approach has been to encode pieces of the desired theorems into the type system, apply it to working code, and then allow the typechecker to verify consistency. If a mistake is made, it will be caught prior to compilation. Or, if the typechecker is satisfied, then the end result is efficient C code that may be compiled and linked and used directly by applications.

4.2 Operating System Verification

The seL4 project is based on a family of microkernels known as L4 [15]. In that work, a refinement proof was completed that demonstrates the adherence of a high-performance C implementation to a generated executable specification, created from a prototype written in Haskell, and checked in the Isabelle [17] theorem proving system. The prototype itself is checked against a high-level design. One difference with our work is that we seek to eliminate the phase of manual translation from high to low level language. Another difference is that, while the seL4 approach can certainly bring many benefits, we feel that the cost associated with it is too high for ordinary use. For example, it may turn out to be intractably difficult to apply this technique to a multiprocessor kernel. That is currently an open problem [9].

Singularity [14] is a microkernel OS written in a high-level and type-safe language that employs language properties and software isolation to guarantee memory safety and eliminate the need for hardware protection domains in many cases. In particular, it makes use of a form of linear types in optimizing communication channels. Singularity was an inspiration for Terrier, although several goals are different. For instance, Terrier seeks to avoid, as much as possible, the overhead associated with high-level languages. Terrier's design is more explicitly geared towards embedded devices responding to real-time events. And inter-program communication in Terrier is left open enough to accommodate multiple approaches, tailored to the particular application domain.

House [11] is an operating system project written primarily in the Haskell functional programming language. It takes advantage of a rewrite of the GHC [18] run-time environment that eliminates the need for OS support, and instead operates directly on top of PS/2-compatible hardware. Then a foreign function interface is used to create a kernel written in Haskell. There is glue code written in C that glosses over some of the trickiness. For example, interrupts are handled by C code which sets flags that the Haskell code can poll at safe points. This avoids potentially corrupting the Haskell heap due to interruptions of the Haskell garbage collector while it is an inconsistent state. The Hello Operating System [10] is an earlier than and similar project to House which features a kernel written and compiled using Standard ML of New Jersey [4], bootstrapped off of Linux [23]. SPIN [3] is a pioneering effort along these lines which used the Modula-3 language [5] to provide a protection model and extensibility. In general, these types of systems do not tackle the problem of high-level language overhead, generally do not handle multiprocessing well if at all, and only offer guarantees as good as their type system can handle.

Both VFiasco [13] and Verisoft [1] take a different approach to system verification. Verisoft relies upon a custom hardware architecture that has itself been formally verified, and a verified compiler to that instruction set. VFiasco claims that it is better to write the kernel in an unsafe language such as C++ and then mechanically generate theorems from that source code, to be discharged by an external proof engine.

5 Conclusion

Our challenge was to take the four slot mechanism and encode at least some of reasoning behind it into dependent types that would compose into a safety theorem. We found this to be feasible, as well as an illuminating example of using a lightweight approach with a dependently typed language to prove useful properties in a non-traditional, concurrent systems programming environment. The code shown in this paper is not a toy example. It is adapted from the actual implementation which is used for inter-program communication in Terrier. The only difference between this and the actual ATS code is the omission of a "handle" parameter which threads state through the functions, and would only complicate the explanation of the proof without adding any strength to it.

To be utterly clear, we are not claiming a full verification of the safety or freshness of the four slot mechanism here. Instead, this approach is a hybrid, based on an advancement in type system power, allowing the programmer to decide what constitutes a sufficient level of assurance. In this case, the types are strong enough that they are able to catch most slight variations. Errors that common type systems would not catch are caught by the ATS typechecker; for example, failing to negate a bit value appropriately, or swapping the order of two seemingly interchangeable statements. These are changes that would break the four slot mechanism but cannot be protected against by a type system without the help of dependent and linear types.

This style of development, intertwining program and proof, with an incremental approach, is the basis of the Terrier project. The four slot mechanism is one example of a component which applies those principles to achieve reliability and efficiency. More complex mechanisms are layered on top of this library, with the confidence that the type system enforces the correct usage of the interface, while the ATS compiler strips away the overhead in the end.

Acknowledgment. We thank Richard West for his guidance on the topics related to operating systems.

References

1. Alkassar, E., Hillebrand, M.A., Leinenbach, D., Schirmer, N.W., Starostin, A.: The Verisoft Approach to Systems Verification. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 209–224. Springer, Heidelberg (2008)
2. ARM Limited. ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition (2011)
3. Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C.: Extensibility, Safety and Performance in the SPIN Operating System. In: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, pp. 267–284 (1995)
4. Blume, M., et al.: Standard ML of New Jersey (2009), `http://www.smlnj.org/`
5. Cardelli, L., et al.: Modula-3 report (revised). Technical report, Digital Equipment Corp. (now HP Inc.) (November 1989), `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-52.html`
6. Chen, C., Xi, H.: Combining Programming with Theorem Proving. In: ICFP 2005: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming, pp. 66–77. ACM Press (2005)
7. Danish, M.: Terrier OS, `http://www.github.com/mrd/terrier`
8. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
9. Elphinstone, K., Heiser, G.: From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels? In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP 2013, pp. 133–150. ACM, New York (2013)
10. Fu, G.: Design and Implementation of an Operating System in Standard ML. Master's thesis, University of Hawaii (August 1999), `http://www2.hawaii.edu/~esb/prof/proj/hello/`

11. Hallgren, T., Jones, M.P., Leslie, R., Tolmach, A.: A principled approach to operating system construction in Haskell. SIGPLAN Not. 40(9), 116–128 (2005)
12. Henderson, N., Paynter, S.E.: The formal classification and verification of Simpson's 4-slot asynchronous communication mechanism. Springer, Heidelberg (2002)
13. Hohmuth, M., Tews, H.: The VFiasco approach for a verified operating system. In: Proceedings of the 2nd ECOOP Workshop on Programming Languages and Operating Systems (2005), http://www.cs.ru.nl/H.Tews/Plos-2005/ecoop-plos-05-letter.pdf
14. Hunt, G.C., Laru, J.R.: Singularity: Rethinking the Software Stack. In: ACM SIGOPS Operating System Review, vol. 41, pp. 37–49. Association for Computing Machinery (April 2007)
15. Klein, G., Elphinstone, K., Heiser, G., et al.: seL4: Formal verification of an OS kernel. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles, Big Sky, MT, USA (October 2009)
16. Lamport, L.: On interprocess communication. Distributed Computing 1-2, 77–101 (1986)
17. Paulson, L.C.: Isabelle. LNCS, vol. 828. Springer, Heidelberg (1994)
18. Peyton-Jones, S., Marlow, S., et al.: The Glasgow Haskell Compiler, http://www.haskell.org/ghc/
19. Rushby, J.: Model checking Simpson's four-slot fully asynchronous communication mechanism. Computer Science Laboratory–SRI International, Tech. Rep. Issued (2002)
20. Simpson, H.R.: Four-slot fully asynchronous communication mechanism. In: IEE Proceedings, vol. 137, Pt. E, No. 1. IEE (January 1990)
21. Simpson, H.R.: Correctness analysis for class of asynchronous communication mechanisms. IEE Proceedings E (Computers and Digital Techniques) 139, 35–49 (1992)
22. Simpson, H.R.: Role model analysis of an asynchronous communication mechanism. In: Computers and Digital Techniques, IEE Proceedings, vol. 144, pp. 232–240. IET (1997)
23. Torvalds, L., et al.: Linux, http://www.linuxfoundation.org/
24. Wadler, P.: A taste of linear logic. In: Borzyszkowski, A.M., Sokolowski, S. (eds.) MFCS 1993. LNCS, vol. 711, pp. 185–210. Springer, Heidelberg (1993), http://dx.doi.org/10.1007/3-540-57182-5_12
25. Xi, H., et al.: The ATS language, http://www.ats-lang.org/
26. Zhu, D., Xi, H.: Safe Programming with Pointers through Stateful Views. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2004. LNCS, vol. 3350, pp. 83–97. Springer, Heidelberg (2005)