Position: Lightweight static resources*

Sexy types for embedded and systems programming

Oleg Kiselyov[1] and Chung-chieh Shan[2]

[1] FNMOC oleg@pobox.com
[2] Rutgers University ccshan@rutgers.edu

Abstract

It is an established trend to develop low-level code—embedded software, device drivers, and operating systems—using high-level languages, especially functional languages with advanced facilities to abstract and generate code. To be reliable and secure, low-level code must correctly manage space, time, and other resources, so special type systems and verification tools arose to regulate resource access statically. However, a general-purpose functional language practical today can provide the same static assurances, also without run-time overhead. We substantiate this claim and promote the trend with two security kernels in the domain of device drivers:

1. one built around raw pointers, to track and arbitrate the size, alignment, write permission, and other properties of memory areas across indexing and casting;
2. the other built around a device register, to enforce protocol and timing requirements while reading from the register.

Our style is convenient in Haskell thanks to custom kinds and predicates (as type classes); type-level numbers, functions, and records (using functional dependencies); and mixed type- and term-level programming (enabling partial type signatures).

1 INTRODUCTION

It is increasingly popular to use typed functional languages to interface with hardware [5, 9, 25], to program special processors such as GPUs [12], and to design and code operating systems [10, 14]. The advanced abstraction facilities offered by these languages are useful even when the language implementations flout low-level storage or timing requirements at run time, because a high-level program is free to generate code to run later in a more constrained environment, and functional languages express code generators easily [12, 28].

Low-level programming is especially error-prone due to hardware constraints on control timing, data size, and pointer range, whose violation leads to immediate or imminent crash. Thus it is especially helpful to check at compile time for correctness, safety, dependability, and integrity [3, 11, 14, 16, 17, 23, 24]—such as to assure that memory accesses are in bounds and properly aligned. Such efforts to apply types and other high-level specifications to low-level constraints fall under the broad rubric of *resource-aware programming*: "using *static checking* to

*Thanks to Ehud Lamm, Martin Sulzmann, and Lambda the Ultimate for helpful discussions.

ensure that computations intended for execution on resource-bounded platforms are indeed resource-bounded" [28].

The constraints imposed by low-level applications go beyond what types typically check. For example, memory addresses must be aligned and sometimes fixed, array indices must be in range, pointers must remain valid, and data properties such as endianness and read-write permissions must be respected. These constraints must be tracked across the pointer arithmetic and casting that abound in low-level programming. Further, protocol and timing requirements arise in communication with external devices, such as that some device register must be read in some order or for some number of times through any execution path in a device driver.

The foreign-function interface usually found in functional languages lets one write low-level code with pointers and registers, but does not account statically for these complex constraints on data representation and control flow. The type level seems to need arithmetic to reason about ranges and alignment, and records to collect the often numerous properties of a resource such as permissions, placement, and layout. For example, tracking the alignment of a pointer as byte offsets are added to it requires computing the GCD of integers [11] and suggests the need for the expressive power of dependent types.

This criticism against the type systems of mainstream functional languages has motivated verification tools using code annotations [3, 14, 16, 17] and experimental languages [11, 13, 23, 24]. In contrast, we show it practical and beneficial to enforce these constraints statically in general-purpose functional languages today, without a special tool or language. This claim is what we mean by "lightweight". Hence, long live the trend of low-level assurances in high-level languages!

1.1 Organization

In the rest of this section, we describe why we need expressive types and why we already have them. Section 2 introduces our infrastructure for expressing numbers, records, and capabilities in types. Much of our infrastructure is general-purpose; we apply it in Section 3 to two applications that one might think require distinct, specialized type systems: enforcing size and alignment constraints of raw memory pointers, and enforcing protocol and timing constraints. We review related work in Section 4 and conclude in Section 5.

We take many examples deliberately from the literature, especially Diatchki and Jones's work on strongly typed memory areas [11], but we implement them, and more, in Haskell. Our complete code is online at `http://pobox.com/~oleg/ftp/Computation/resource-aware-prog/`

1.2 Sexy types on the down low

Memory access exemplifies how resource-aware programming calls for expressive types. Suppose that we have a pointer p to an array of n words in memory, each m bits wide. At a dynamically determined index i in this array, we want to write

a dynamically determined number j. To maintain the integrity of the system, we must have $0 \leq i < n$ and $0 \leq j < 2^m$. To assure these constraints statically, the type of p must mention n and m, and we need a type for `write_memory` like

```
Ptr n m -> IntBounded n -> IntBounded (2^m) -> IO ()
```

where `n` and `m` are types of kind `Nat`, and `IntBounded n` is the type of natural numbers less than n. Sheard [23, 24], Diatchki and Jones [11], and Taha [28] argue that mainstream functional languages like Haskell cannot express this type, so they develop their own experimental languages. Actually, we can write the Haskell type

```
Exp2 m k =>
Ptr n m -> IntBounded n -> IntBounded k -> IO ()
```

using the library described in this paper. The type-class constraint `Exp2 m k` is explicit here, but other constraints such as `Nat n` and `Nat m` are inferred. This example illustrates four points about our programming style that we explain below.

1. We can express custom kinds (such as `Nat`) and predicates (such as exponentiation) as type classes. Kinds are types for compile-time data.
2. We can perform type-level arithmetic using functional dependencies (in fact, even in familiar term-level notation).
3. We can specify part of a type signature (such as `Exp2 m k` above) but leave the rest to be inferred (such as `Nat n` and `Nat m` above).
4. An abstract data type (such as `IntBounded n`) constitutes a *static capability* [29] certifying a *safety property* (that a number is in bounds), issued by a user-defined *security kernel* (the implementation of `IntBounded`).

Our techniques are not restricted to pointers. We also outline how to generate code for a GPU or other special processor while enforcing protocol and time constraints.

Experimental languages like Diatchki and Jones's [11] provide syntactic sugar for their domains, such as `4K` for 4KB, that general-purpose languages do not. The drawback of experimental languages is that they need new tools and libraries. Diatchki and Jones's language only looks like Haskell: it is strict and has type improvement rules more general than functional dependencies. In contrast, Haskell is in widespread use, and we find its general type system perfectly serviceable and more flexible for tracking a wide variety of resources: space, time, and beyond.

2 TYPE-LEVEL INFRASTRUCTURE

In this section, we show how to compute with numbers and records in Haskell types. Far from being unattainable or impractical in a general-purpose type system, these features can be packaged in a user-defined library that needs no extended language or extra tool. The accompanying source code contains the complete library. This library then enables resource-aware programming, as we show in Section 3. Code there also demonstrates term-level type programming and partial signatures.

Our organizing principle is to express custom kinds (such as those of type-level numbers and records) and predicates (such as numeric comparison and record lookup) as classes of *phantom types*. We also rely on functional dependencies to express type-level computations as multi-moded relations. We secure the constraints like any data-type invariant: keep types abstract and symbols unexported.

2.1 Numbers in types

We define binary digits at the type level and combine them to form binary numbers. For example, the type `U (U B1 B1) B0` represents 6. These types are *phantom* in the sense that, at run time, they have no representation and use no space or time—their only inhabitant is $\perp$, which we never bother storing at run time. We prohibit leading zeros in our representation of numbers.

```
data B0; data B1; data U x y
```

The type classes `Nat0` and `Nat` encode the kinds of non-negative and positive integers. The instances are trivial to define. The subclassing encodes subkinding.

```
class Nat0 a where toInt :: a -> Int
class Nat0 a => Nat a
```

We then define arithmetic operations: successor/predecessor, addition/subtraction, multiplication/division, and exponentiation/logarithm. Each pair is a *relation* that runs in all possible deterministic modes. For example, in the ternary relations `Add` and `Mul`, any two arguments determine the third. In other words, we solve arithmetic constraints such as $2^n = 256$ using the type-class system as a general constraint solver [26] rather than using Diatchki and Jones's specialized solver [11] or Sheard's *narrowing* [24]. (We omit below a few auxiliary class constraints.)

```
class (Nat0 x, Nat y) => Succ x y | x -> y, y -> x
class (Nat0 x, Nat0 y, Nat0 z)
    => Add x y z | x y -> z, z x -> y, z y -> x
class (Nat x, Nat y, Nat z)
    => Mul x y z | x y -> z, x z -> y, y z -> x
class (Nat0 x, Nat y) => Exp2 x y | x -> y, y -> x
```

We also implement comparisons between numbers. For convenience, we provide special type classes for $\leq$ and $<$, alongside a general comparison predicate.

```
data BLT; data BEQ; data BGT
class (Nat0 x, Nat0 y) => NCompare x y r | x y -> r
class (Nat0 x, Nat0 y) => NLessEq x y
class (Nat0 x, Nat y)  => NLess x y
```

Using the `NCompare` class above, we implement Euclid's algorithm for GCD.

```
class (Nat0 x, Nat0 y, Nat0 z) => GCD x y z | x y -> z
gcd :: GCD x y z => x -> y -> z; gcd = undefined
```

We define the term-level function `gcd` (analogously `add`, `sub`, and so on for the other classes above) even though it is `undefined` and used only for its type. Section 3 below illustrates the convenience of such term-level type programming.

To illustrate these classes in practice, we write a Haskell function `max` that computes the maximum of two type-level numbers.

```
max :: (NCompare x y b, NMax x y b r) => x -> y -> r
max = undefined
class NMax x y b r | x y b -> r
instance NMax x y BLT y
instance NMax x y BEQ y
instance NMax x y BGT x
```

This module exports all these types but few of the classes, to keep other modules from compromising the classes' assurances by adding instances. For example, the module does not export `Nat`, so that another module cannot define an "exotic natural number". We do, however, want to let other modules specify *constraints* using classes like `Nat` and `Add`. Two strategies achieve this latter end. First, we can export a subclass `ExportNat` of `Nat`, with just one instance.

```
class Nat a => ExportNat a; instance Nat a => ExportNat a
```

Any other instance of `ExportNat` that another module might define would overlap with our instance above without relaxing the `Nat` constraint. Second, we can export not a class but a term whose type mentions the class. For example, we export `toInt` but not `Nat0` above. This strategy calls for partial signatures (Section 3).

We use no overlapping instances, and our algorithms always terminate when supplied with ground numerals as input. (If the needed inputs as specified by functional dependencies are not ground, as in the constraint `Add n B0 n`, then Haskell should just propagate the constraints without reducing them, thus avoiding the impossible task of deciding universally quantified arithmetic statements.) However, many of our proofs that constraint reduction terminates rely on a global ordering over type classes. For example, we define just one instance for the `Add` class, which invokes an auxiliary `Add'` class:

```
class (Nat0 x, Nat0 y, Nat0 z)
    => Add' x y z | x y -> z, z x -> y
class (Add' x y z, Add' y x z)
    => Add x y z | x y -> z, z x -> y, z y -> x
instance (Add' x y z, Add' y x z) => Add x y z
```

To deduce that the `Add` constraint always terminates, we note that no instance for `Add'` invokes `Add`. The `ExportNat` instance above terminates for the same reason. Such a lack of cycles cannot be established by considering each instance separately. Because GHC checks each instance separately for termination, we have to resort to enabling undecidable instances in GHC. If only GHC would let us specify a partial order among classes that prohibits all `Add'` instances from invoking `Add`, then we might convince it.

2.2 Records in types

Like Diatchki and Jones [11], we want to annotate resources like memory areas with compile-time properties to indicate where they are (for example, at a fixed address) and how to access them (for example, read only). An extensible facility for such annotations amounts to records at the type level. Diatchki and Jones [11] wish for type records as a desirable feature not supported by their language. This wish they leave for future work is however attainable in Haskell: we can express type-level records in Haskell as another user-defined library, following the footsteps of heterogeneous lists at the term level [18].

We define a type class `Property` to mean that a resource `label` has a named property `p` with some value `v`. Each area property (`AP`) is a phantom type.

```
class Property label p v | label p -> v
data APReadOnly; data APFixedAddr; data APInHeap
```

To express Boolean property values, we define the types `HTrue` and `HFalse`.

```
data HTrue; data HFalse
```

Any primitive memory operation that requires write permission can then impose a constraint of the form `Property label APReadOnly HFalse`. For example, to extend a record `l` with a read-only property, we define a new type constructor `ROTrait`. We need to "lift" other properties through `ROTrait`, which tempts us to indulge in an overlapping instance, but we do not.

```
data ROTrait l
instance Property (ROTrait l) APReadOnly HTrue
instance Property l APARef value
    => Property (ROTrait l) APARef value
instance Property l APInHeap value
    => Property (ROTrait l) APInHeap value -- etc.
```

Our type records subsume Diatchki and Jones's distinction between a byte array, which we can treat as an array of another type, and an array of higher-level data, which we must not. This distinction prevents overwriting an array of inbound indices (which we support) with arbitrary bytes. We introduce a Boolean property `APOverlayOK` to let the programmer authorize coercing a (part of a) memory area to a different type. Coercion imposes a constraint `Property area APOverlayOK HTrue`, meaning that only so authorized areas may be coerced. The subarea produced by coercion inherits any properties (such as being read-only) of the coerced area. (The coercion operation also statically checks the size of the subarea and computes its alignment using the type-level arithmetic described in Section 2.1.)

2.3 Lightweight static capabilities

The types introduced so far are all phantom. To regulate resources, we encapsulate the resources in abstract data types parameterized by the phantom types. We view

such an encapsulated resource as a *capability* [20, 29] that permits an operation and certifies a property [19]. These capabilities are *static* in that type checking takes place at compile time and does not affect the representation of resources or performance of operations at run time.

One example of a static capability is the `IntBounded` type from Section 1.2. The run-time representation of `IntBounded n` is just an ordinary machine integer, not a linked list of bits, a unary number, or a tuple of numbers. However, the public, 'smart' constructor of `IntBounded n` checks that the integer is non-negative and less than the natural number n represented by the phantom type `n`. Such an integer can then be used to index into an n-element array, stored in $\lceil \log_2 n \rceil$ bits of memory, or both, with neither the risk of overflow nor the run-time overhead of any further bounds check.

Although the public 'smart' constructor for `IntBounded n` checks its argument against both a lower bound and an upper bound, we can skip many bound checks when computing with `IntBounded` values already constructed. For example, averaging bounded values always yields a bounded value, so an algorithm such as binary search can use an exported averaging function of the type

```
IntBounded n -> IntBounded n -> IntBounded n
```

to avoid any bound check. To take another example, incrementing a bounded value always yields a lower-bounded value. The upper-bound check that we must perform after incrementing a bounded value to keep the bounds the same is often a loop termination test that the algorithm calls for anyway. Thus incrementing a bounded value is an operation of the type

```
IntBounded n -> Maybe (IntBounded n)
```

that skips the lower-bound check while doubling as a loop termination test. The implementation of `forEachIx` in Section 3 gives one example; we give more complex examples elsewhere [19]. We have expressed even generally recursive algorithms with no bound checks other than index comparisons in the algorithm.

Another example of a static capability is a reference to a memory area. At run time, the reference is simply represented by a raw pointer, but its type permits the reference holder to (say) store an array of 2000 16-bit words there. Just as static bounds on indices and pointers eliminate unnecessary comparisons, static alignment information eliminates unnecessary alignment checks.

We thus divide the program into two parts: a *security kernel* that we trust to produce and consume capabilities responsibly, and its client, the *sandbox*, which acquires and propagates capabilities with abandon. Our general approach of lightweight static capabilities is to separate these parts by type abstraction. We express the kernel explicitly as a user-defined library rather than a language extension with syntactic sugar [11, 23, 24, 28]. Like Diatchki and Jones, we expect the programmer to use the primitives provided by our kernel to build higher-level abstractions that encapsulate domain-specific constraints such as absolute memory addresses and hardware specifications. Of course, the programmer coding these abstractions

could misread the hardware specification and misuse an absolute address. Such a mistake would go undetected by our kernel and render the system unsafe. We strive to enable and ease the encapsulation process, not to supplant domain knowledge.

3 TWO APPLICATIONS OF STATIC RESOURCES

Using numbers and records at compile time, we show two applications of statically tracking resources and assuring their proper access by device drivers. The device driver in both applications may be a typed functional program or generated by one.

3.1 Space: Memory areas

Diatchki and Jones's seminal paper [11] introduces a language for manipulating data in raw memory. These memory areas include page tables, kernel data, and firmware and memory-based IO structures. Their location, size, alignment, representation, and byte layout are usually rigidly defined, and some of them are read-only or carry other access-pattern restrictions. To assure proper access, the language tracks some of these constraints in types across indexing, casting, and other operations.

Following their language's features, we implement a Haskell library of memory areas containing signed and unsigned, big- and little-endian integers of various sizes and alignments, as well as pairs and statically sized arrays of them. We first show how the library implements array indexing, then use it to access video RAM.

We implement our array-indexing operator `@@` as follows.

```
aref_sig :: INDEXABLE arr count base totalsize =>
            ARef al arr -> Ix count -> ARef al' base
aref_sig = undefined
r @@ i | False = aref_sig r i
r@(ARef p) @@ Ix i
  = cast_aref (p `plusPtr` (i * toInt base_size)) al base
  where al        = gcd (aref_al r) base_size
        base      = arr_base (aref_area r)
        base_size = size_of base
```

The type of `@@` combines constraints from two sources. First, the `undefined` function `aref_sig` contributes an `INDEXABLE` constraint, so the explicit type specified for `aref_sig` above is a *partial* signature for `@@`. Second, the term definition of `@@` invokes `undefined` term-level functions solely for their types. Besides `gcd` from Section 2.1, the accessor functions below are also `undefined`.

```
aref_area :: ARef al area -> area; aref_area = undefined
aref_al   :: ARef al area -> al;   aref_al   = undefined
size_of :: SizeOf area n => area -> n; size_of = undefined
```

Hence the terms defined in the `where` clause are all `undefined`. We care only for the resulting phantom types. They are passed—again as terms—to the unexported internal function `cast_aref`, which uses them to brand a raw pointer.

```
cast_aref :: Ptr Word8 -> align -> area -> ARef align area
cast_aref p _ _ = ARef p
```

GHC collects the constraints from the partial signature and the term-level type functions to infer a type for `@@` that is quite close to Diatchki and Jones's for `@`:

```
(INDEXABLE arr count base totalsize, GCD al n z,
 SizeOf base n) => ARef al arr -> Ix count -> ARef z base
```

Some constraints below and in our code online are much more involved and thus much messier to specify directly. Partial signatures and term-level type programming make these examples practical to type. Whereas type programming feels like logic programming when expressed at the type level, it feels like functional programming when expressed at the term level—even though both ways specify the same static, compile-time, type-level computation, erased at run time.

We now turn to Diatchki and Jones's running example, a text-mode terminal driver for a generic PC. The hardware can be controlled simply by writing in a piece of memory `videoRAM` located at a dedicated address. We show functions to write a character and to clear the screen. We provide in Haskell the same static assurances as Diatchki and Jones do: only write in-range characters and attributes to in-range rows and columns, without confusing character bytes with attribute bytes, and without the overhead of unnecessary run-time checking or conversion.

The `videoRAM` memory area represents a PC screen by a 2D array of 25 rows of 80 columns of pairs of bytes. Each byte pair describes the attribute (background and foreground colors) and character at one screen location. We express this layout using the area constructors `Array`, `Pair`, and `AWord8`.

```
type ScreenT = Array N25 (Array N80 (Pair AWord8 AWord8))
```

This notation is almost Diatchki and Jones's, except `N25` and `N80` are synonyms for type-level numbers in Section 2.1. Using the type-level records in Section 2.2, we define `videoRAM` to refer to a new memory area `ScreenAbs`, an 8-byte-aligned read-write screen buffer at the fixed address `0xb8000` rather than the heap.

```
data ScreenAbs = ScreenAbs
instance Property ScreenAbs APARef (ARef N8 ScreenT)
instance Property ScreenAbs APReadOnly HFalse
instance Property ScreenAbs APFixedAddr HTrue
instance Property ScreenAbs APInHeap HFalse
videoRAM = area_at ScreenAbs (nullPtr `plusPtr` 0xb8000)
```

If we neglect to set `APFixedAddr` to `HTrue`, `area_at` will report a type error.

To index into `videoRAM` for a particular attribute or character, we define

```
charAt i j = asnd (videoRAM @@ i @@ j)
```

where `asnd` selects the second component of a pair area. Haskell infers the type

```
Ix N25 -> Ix N80 -> ARef B1 (AtArea ScreenAbs AWord8)
```

including the GCD alignment `B1` and the static index bounds `N25` and `N80`.

To clear the screen (fill `videoRAM` with blanks), we define a helper iterator

```
forEachIx proc = loop minBound where
  loop ix = do proc ix
               maybe (return ()) loop
                 (ixSucc ix <<= maxBound `asTypeOf` ix)
```

This iterator invokes `proc` on every index in range. The only run-time range check it incurs is the loop termination test `<<=`. The way we implement bounded indices [19] is quite different from Diatchki and Jones's. For example, we avoid introducing unintuitive "$n-k$ patterns" to specify index computations.

Our code online shows how to use `forEachIx` to iterate over the rows and columns of `videoRAM`, writing blank characters and attributes. It is more efficient though to coerce (*overlay*) `videoRAM` to a 1D array of 2-byte big-endian integers.

```
cls = forEachIx (\i -> write_area (vr @@ i) blank) where
  vr = as_area videoRAM
             (mk_array_t size (undefined::BEA_Int16)) nat0
  size = size_of (aref_area videoRAM) `div` nat2
blank :: Int16; blank = 0x7020 -- space on white background
```

The `size` of the overlay `vr` is expressed as a term but computed statically. If we forget the division by two or choose a wrong offset (`nat1` rather than `nat0`), the type system will complain that `vr` is too big or misaligned. These static computations are more involved than above and nice to be able to specify in terms.

Instead of checking the size of `vr`, the type system can also compute it for us. We conveniently specify by a dummy term that the overlay `vr` is as big as the base area `videoRAM`. The type system, in the spirit of logic programming, then solves the equation $2 \times \mathit{size} = 25 \times 80$ by division. Misalignment still triggers a type error.

```
cls = forEachIx (\i -> write_area (vr @@ i) blank) where
  vr = as_area videoRAM
         (mk_array_t undefined (undefined::BEA_Int16)) nat0
  _ = size_of (aref_area videoRAM) `asTypeOf`
      size_of (aref_area vr)
```

In the accompanying source code, `AreaTests.hs` shows how to store and retrieve array indices in memory as bounded integers, preserving their static upper bound and thus obviating run-time bound checks. We define a memory area whose first byte is an index into the array beginning at the second byte.

```
type ARIT = Pair (IntBounded N4 AWord8) (Array N4 AInt8)
```

The array has four signed byte (`AInt8`) elements, so we make the type of stored offsets be that of a bounded integer less than 4. The compiler checks that an `AWord8` can represent such an integer. Reading the first part of `ARIT` gives a value of the type `Ix N4`: the static bound propagates to the type of the index. When we use that value to index into the array in the second part of `ARIT`, the compiler checks that the index's static bound matches the static size of the array. Because the size of the array is `N4` as well, the operation type-checks.

3.2 Time: Processing ticks

Effect types let us extend our approach from data resources such as space to control resources such as time. When generating device-driver code in Haskell, even loops, we can check that it adheres to a protocol or stops within a worst-case time limit. We show just two examples: first, assuring that the driver reads a register the same number of times in any execution path; second, tracking the maximum number of times the register is read. More examples are online in `RealTime.hs`.

It is useful to track the number of times a hardware register is read, when each read yields the next byte in a protocol frame. To save space and time, some drivers read a frame byte by byte, not all at once. Each byte read affects how the driver deals with the rest of the frame. To obey the protocol, the driver must read exactly all of one frame before moving to the next.

To guarantee statically that a part of the driver always reads the same number of bytes, we define a *parameterized monad* [2] `VST m si so`, which transforms a base monad `m` to track a counter whose initial value is `si` and final value is `so`. We write >== and +>> for the analogues of >>= (bind) and >> in `VST`. Haskell thus infers the type `VST IO N0 (U (U B1 B0) B1) Int` for the code below, meaning that it reads the register 5 times no matter which control path it takes.

```
read_frame = tread_byte >== \b1 ->
             if b1 > 10 then branch1 b1
             else branch2 b1 +>> tread_byte +>> tread_byte
 where
  branch1 b = tread_byte >== \b2 ->
              if b2 > 10 then tread_byte +>> branch2 b2
              else branch3 b2
  branch2 b = tread_byte +>> tread_byte
  branch3 b = tread_byte +>> glift (putStrLn "branch3")
                         +>> tread_byte +>> tread_byte
```

This code receives a frame of data from a device. The first byte identifies the frame, so we read a byte `b1` and branch upon its value. In `branch1`, we read another byte and branch again. Some branches perform dummy reads to match the reads in other branches. If we miss those, the compiler complains. Unlike in the parameterized monad of type-changing state, the types `si` and `so` are phantom, so the static counting does not affect run-time performance.

Suppose we replace each conditional `if test then thb else elb` in `read_frame` by `gif test thb elb`, where `gif` has the (inferred) type

```
gif :: (Monad m, NCompare sthen selse b,
        BinaryNumber.NMax sthen selse b so) =>
       Bool -> VST m si sthen v ->
               VST m si selse v -> VST m si so v
```

Then each control path can increase the counter differently, but the type checker still accumulates the maximum increase among paths so as to statically determine

the worst 'time complexity' of the code. When using this technique in a code generator (such as Elliott's [12]), we associate a counter with a generated instruction and the generating expression. This lets us statically bound the time complexity of our computation and determine if it will finish quickly enough.

To handle loops, we use the fact that the number of steps in each processing cycle of the embedded system must be statically bounded. We ensure this fact by using the bounded type `Ix n` for the loop counter, which also lets us bound the latency of the loop. For example, the following program first reads a byte from the register. If the read value is less than 10, we read twice that many more bytes.

```
read_frame' = tread_byte >== \b1 ->
              gmaybe tread_byte (loop b1) (toIxN nat10 b1)
 where
  loop b ix = foldTM branch1 b ix
  branch1 b = tread_byte +>> tread_byte
```

The type checker infers the final state `U (U (U (U B1 B0) B0) B1) B1`, that is, the register will be read at most 19 times. Here `foldTM` is a tick-counting `foldM` combinator; its type is

```
(Monad m, ExportNat n, Succ n' n, Add si diff so',
 Mul diff n' grand_diff, Add si grand_diff so) =>
(v -> VST m si so' v) -> v -> Ix n -> VST m si so v
```

Statically tracking tick counters in the `VST` monad assures that the monadic computation terminates: within one run of the monadic computation (one cycle through the device driver), the device register shall be accessed finitely many times.

4 RELATED WORK

Several authors [7, 8, 11] have reviewed the long history of space- and time-aware functional programming. Unfortunately, most approaches (such as Hughes and Pareto's [15]) severely restrict the language, such as limiting it to first-order and requiring the programmer to specify memory usage in detailed annotations.

As with all type systems, the language must be somewhat restricted because the ability to impose constraints competes with the flexibility to express computations. However, even in a device driver, we can separate the resource-sensitive part of the program (the part that interfaces with hardware, where space and time constraints are paramount) from the rest (relatively unconstrained "simplifications" [21]). We can then write embedded software in a powerful, functional language with higher-order functions, user-defined data structures, and polymorphism. This separation is explicit if our program is a code generator (such as Elliott's [12] for a GPU).

Such a separation, or *staging*, is the focus of resource-aware programming (see Taha's survey [28]). This separation is also present in Hume [7], which defines a nested hierarchy of languages with a series of tradeoffs between expressiveness and decidability. Yet another example of separating resource-sensitive computations is

Timber [22], where only monadic actions have timing constraints. Timber is based on reactive objects and designed for real-time embedded systems, not as a general-purpose functional language.

To express hardware constraints statically, we need an advanced type system. In fact, Taha [28] defines the latter as an attribute of resource-aware programming. Often the advance is dependent types, in which the same terms manipulated at run time encode constraints checked at compile time. For example, Brady and Hammond [8] use dependent types to represent and infer size metrics of a program; they have implemented several examples in Coq. In contrast, Diatchki and Jones [11], Sheard [23], and us encode constraints using types rather than terms. This phase distinction between types and terms nicely matches the stage separation discussed above, but at the cost of proliferating representations. Nevertheless, our techniques of term-level type programming and partial signatures make the notation of these three approaches more uniform and closer to that of dependently typed systems such as Coq [4] and Epigram [1].

Diatchki and Jones's [11] and Sheard's [23] experimental languages are like Haskell but strict. They add built-in kinds such as `Nat` and can evaluate type functions forwards and backwards [24]. Inspired by these languages, we show here that Haskell can already represent such numeric kinds and type-level functions, albeit perhaps less prettily. Unlike the experimental languages, our "lightweight" approach relies on the type system only to propagate assurances (such as indices being in range) soundly from the security kernel to the rest of the code [19], not to check that the kernel itself is sound (such as arithmetic or hardware constraints being encoded correctly). After all, our security kernel is just a library of types with no intrinsic interpretation in Haskell.

All related approaches above are based on experimental language systems, whose development is limited and user base is small. We aim for resource-aware functional programming in a mature language as it is. Like `ml-nlffigen` [6], we use types to refine a foreign-function interface, but our approach is more expressive because Haskell can compute with types. The Cyclone language, although not functional, permits low-level programming with raw pointers and static assurances such as memory references being valid. Our assurances are programmed in Haskell rather than built into the language, and so can be extended by the programmer.

5 CONCLUSIONS

Our examples show Haskell practical for high-assurance low-level programming. Our type-system library statically assures hardware-imposed constraints on control and data when accessing raw memory and external devices. It can also be used for arithmetic without overflow.

Type classes help, but we still wish for: a `do`-like notation for parameterized monads; more informative type-error messages [27]; and defining a type synonym for the inferred type of an expression.

REFERENCES

[1] Altenkirch, Thorsten, Conor McBride, and James McKinna. 2005. Why dependent types matter. `http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf`.

[2] Atkey, Robert. 2006. Parameterised notions of computation. In *MSFP 2006: Workshop on mathematically structured functional programming*, ed. Conor McBride and Tarmo Uustalu. Electronic Workshops in Computing, British Computer Society.

[3] Barnes, John. 2003. *High integrity software: The SPARK approach to safety and security*. Boston: Addison-Wesley.

[4] Bertot, Yves, and Pierre Castéran. 2004. *Interactive theorem proving and program development: Coq'Art: The calculus of inductive constructions*. EATCS Texts in Theoretical Computer Science, Berlin: Springer-Verlag.

[5] Biagioni, Edoardo, Robert Harper, Peter Lee, and Brian G. Milnes. 1994. Signatures for a network protocol stack: A systems application of Standard ML. In *Proceedings of the 1994 ACM conference on Lisp and functional programming*, 55–64. New York: ACM Press.

[6] Blume, Matthias. 2001. No-longer-foreign: Teaching an ML compiler to speak C "natively". In *BABEL'01: 1st international workshop on multi-language infrastructure and interoperability*, ed. P. Nick Benton and Andrew Kennedy. Electronic Notes in Theoretical Computer Science 59(1), Amsterdam: Elsevier Science.

[7] Bonenfant, Armelle, Zezhi Chen, Kevin Hammond, Greg Michaelson, Andy Wallace, and Iain Wallace. 2007. Towards resource-certified software: A formal cost model for time and its application to an image-processing example. In *ACM Symposium on Applied Computing (SAC 07), Seoul, Korea, March 11–15*. To appear.

[8] Brady, Edwin, and Kevin Hammond. 2006. A dependently typed framework for static analysis of program execution costs. In *Revised selected papers from IFL 2005: 17th international workshop on implementation and application of functional languages*, ed. Andrew Butterfield, Clemens Grelck, and Frank Huch, 74–90. Lecture Notes in Computer Science 4015, Berlin: Springer-Verlag.

[9] Claessen, Koen. 2000. An embedded language approach to hardware description and verification. Master's thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden.

[10] Derrin, Philip, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. 2006. Running the manual: An approach to high-assurance microkernel development. In *Proceedings of the 2006 Haskell workshop*, 60–71. New York: ACM Press.

[11] Diatchki, Iavor S., and Mark P. Jones. 2006. Strongly typed memory areas: Programming systems-level data structures in a functional language. In *Proceedings of the 2006 Haskell workshop*. New York: ACM Press.

[12] Elliott, Conal. 2004. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell workshop*, 45–56. New York: ACM Press.

[13] Grossman, Dan, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based memory management in Cyclone. In *PLDI '02: Proceedings of the ACM conference on programming language design and implementation*, 282–293. New York: ACM Press.

[14] Hallgren, Thomas, Mark P. Jones, Rebekah Leslie, and Andrew P. Tolmach. 2005. A principled approach to operating system construction in Haskell. In *ICFP '05: Proceedings of the ACM international conference on functional programming*, 116–128. New York: ACM Press.

[15] Hughes, John, and Lars Pareto. 1999. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *ICFP '99: Proceedings of the ACM international conference on functional programming*, 70–81. New York: ACM Press.

[16] Hunt, Galen, James R. Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian D. Zill. 2005. An overview of the Singularity project. Tech. Rep. MSR-TR-2005-135, Microsoft Research.

[17] Kieburtz, Richard B. 2002. P-logic: Property verification for Haskell programs. `ftp://ftp.cse.ogi.edu/pub/pacsoft/papers/Plogic.pdf`.

[18] Kiselyov, Oleg, Ralf Lämmel, and Keean Schupke. 2004. Strongly typed heterogeneous collections. In *Proceedings of the 2004 Haskell workshop*. New York: ACM Press.

[19] Kiselyov, Oleg, and Chung-chieh Shan. 2006. Lightweight static capabilities. In *PLPV 2006: Programming languages meet program verification*, ed. Aaron Stump and Hongwei Xi. Electronic Notes in Theoretical Computer Science, Amsterdam: Elsevier Science.

[20] Miller, Mark Samuel, Chip Morningstar, and Bill Frantz. 2000. Capability-based financial instruments. In *Proceedings of FC 2000: Financial cryptography, 4th international conference*, ed. Yair Frankel, 349–378. Lecture Notes in Computer Science 1962, Berlin: Springer-Verlag.

[21] Moggi, Eugenio, and Sonia Fagorzi. 2003. A monadic multi-stage metalanguage. In *Proceedings of FoSSaCS 2003: Foundations of software science and computational structures, 6th international conference*, ed. Andrew D. Gordon, 358–374. Lecture Notes in Computer Science 2620, Berlin: Springer-Verlag.

[22] Nordlander, Johan, Magnus Carlsson, and Mark P. Jones. 2004. Programming with time-constrained reactions. `http://www.cse.ogi.edu/pacsoft/projects/Timber/publications.htm`.

[23] Sheard, Tim. 2005. Another look at hardware design languages: Freeing ourselves from the tyranny of the host language. `http://web.cecs.pdx.edu/~sheard/papers/secondLook.ps`.

[24] ———. 2006. Type-level computation using narrowing in Ωmega. In *PLPV 2006: Programming languages meet program verification*, ed. Aaron Stump and Hongwei Xi. Electronic Notes in Theoretical Computer Science, Amsterdam: Elsevier Science.

[25] Sheeran, Mary. 2005. Hardware design and functional programming: A perfect match. *Journal of Universal Computer Science* 11(7).

[26] Stuckey, Peter J., and Martin Sulzmann. 2005. A theory of overloading. *ACM Transactions on Programming Languages and Systems* 27(6):1216–1269.

[27] Stuckey, Peter J., Martin Sulzmann, and J. Wazny. 2004. Improving type error diagnosis. In *Proceedings of the 2004 Haskell workshop*, 80–91. New York: ACM Press.

[28] Taha, Walid. 2005. Resource-aware programming. In *Embedded software and systems, 1st international conference, ICESS 2004, revised selected papers*, ed. Zhaohui Wu, Chun Chen, Minyi Guo, and Jiajun Bu. Lecture Notes in Computer Science 3605, Berlin: Springer-Verlag.

[29] Walker, David, Karl Crary, and Greg Morrisett. 2000. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems* 22(4):701–771.