Lightweight static capabilities

Oleg Kiselyov

FNMOC

Chung-chieh Shan

Rutgers University

Abstract

We describe a modular programming style that harnesses modern type systems to verify safety conditions in practical systems. This style has three ingredients:

(i) A compact kernel of trust that is specific to the problem domain.
(ii) Unique names (*capabilities*) that confer rights and certify properties, so as to extend the trust from the kernel to the rest of the application.
(iii) Static (type) proxies for dynamic values.

We illustrate our approach using examples from the dependent-type literature, but our programs are written in Haskell and OCaml today, so our techniques are compatible with imperative code, native mutable arrays, and general recursion. The three ingredients of this programming style call for (1) an expressive core language, (2) higher-rank polymorphism, and (3) phantom types.

1 Introduction

This paper demonstrates a lightweight notion of *static capabilities* (Walker et al. 2000) that brings together increasingly expressive type systems and increasingly accessible program verification. Like many programmers, before verifying that our code is correct, we want to assure safety conditions: array indices remain within bounds; modular arithmetic operates on numbers with the same modulus; a file or database handle is used only while open; and so on. The safety conditions protect objects such as arrays, modular numbers, and files. Our overarching view is that a capability authorizes access to a protected object and simultaneously certifies that a safety condition holds. Rather than proposing a new language or system, our contribution is to substantiate the slogan that types are capabilities, today: we use concrete and straightforward code in Haskell and OCaml to illustrate that a programming language with an appropriately expressive type system is a static capability language. Because the capabilities are checked at compile time, we achieve the safety assurances with minimal impact to run-time performance.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

Section 2 presents a simplified introductory example: empty-list checking. Section 3 turns to a full-featured example: array-bound checking. In each case, we formalize our technique as a syntactic translation between two languages. Section 4 distills the three ingredients of our programming style and describes how expressive it needs the type system to be. Section 5 discusses related and future work.

Our technique scales up: First, shown in the Appendix is a more substantial example, Knuth-Morris-Pratt string search. Second, the Takusen database-access project uses our technique to verify the safety of session handles, cursors, prepared-statement handles, and result sets. For instance, any operation on a session is guaranteed to receive a valid session handle. We take our examples from Xi's pioneering work on practical dependent-type systems and Dependent ML, as well as from user suggestions.

2 Empty-list checking

We start with a simplified introductory example. Although it does not show all features of our approach, it sets the pattern we follow throughout the paper. The example is list reversal with an accumulator, which can be written in OCaml as

```
let rec rev l acc = if null l then acc
                    else rev (tail l) (cons (head l) acc)
```

The code is written for an arbitrary data structure satisfying the list API (`null`, `cons`, `head` and `tail`), so it does not use pattern matching.

The functions `head` and `tail` are partial because they do not make sense for the empty list. Therefore, these functions, to be safe, must check their argument for null before deconstructing it. This code for `rev` checks the same list `l` for null three times: once directly by calling `null`, and twice indirectly in `head` and `tail`.

We can remove excessive checks and gain confidence in the code by prohibiting attempts to deconstruct the empty list. We first define an abstract data type `'a fullList` with the interface and implementation below.

```
module FL : sig
   type 'a fullList
   val unfl : 'a fullList -> 'a list
   val indeed : 'a list -> 'a fullList option
   val head : 'a fullList -> 'a
   val tail : 'a fullList -> 'a list
end = struct
   type 'a fullList = 'a list
   let unfl l = l
   let indeed l = if null l then None else Some l
   let head l = Unsafe.head l
   let tail l = Unsafe.tail l
end
```

Here `Unsafe.head` blindly gives the head of its list argument, without checking if the argument is null. We claim that in well-typed programs the functions `FL.head` and `FL.tail` are total.

Our list reversal with accumulator is now safer and more efficient:

```
let rec rev' l acc = match FL.indeed l with
  | None -> acc
  | Some l -> rev' (FL.tail l) (cons (FL.head l) acc)
```

The code is basically the same as before, but it checks for null only once. The inferred type of `rev'` is the same as that of `rev`, that is, `'a list -> 'a list -> 'a list`.

The `indeed` function constructs an `option` value that is deconstructed by `rev'` right away. We can eliminate this tagging overhead by changing our code to continuation-passing style.

```
module FL : sig ...
val indeed : 'a list -> (unit -> 'w) -> ('a fullList -> 'w) -> 'w
let indeed l onn onf = if null l then onn () else onf l
...
let rec revc' l acc = FL.indeed l (fun () -> acc)
  (fun l -> revc' (FL.tail l) (cons (FL.head l) acc))
```

2.1 *Extending a kernel of trust*

This example illustrates the basic features of our approach. A *security kernel* `FL` implements an abstract data type `fullList`. A `fullList` at run time is the same as a regular `list` and need not impose any overhead (it helps to use a defunctorizing compiler such as MLton). The point of `fullList` is to *certify* a safety condition at compile time, in that a (non-bottom) `fullList` value is never null. The functions `FL.head` and `FL.tail` use this certificate in the type of their argument (rather than a dynamic check) to assure themselves that the access is safe.

Essentially, `fullList` (on which `FL.head` and `FL.tail` are defined) is a subtype of `list` (Appel and Leroy 2006). However, to avoid extending the underlying type system with this subtyping, we make projection explicit as `indeed`, and injection explicit as `unfl`. Experience with `toEnum`, `fromEnum`, `fromIntegral`, etc. in Haskell suggests that the resulting notational overhead is bearable, even familiar.

Another way to view the `fullList` certificate is as a *capability* (Miller et al. 2000) that authorizes access to the list components. This capability is *static* because it is expressed in a type rather than a value (Walker et al. 2000). This idea, to express the result of a dynamic value test as a static type certificate, is important in dependent-type programming (Altenkirch et al. 2005; Section 5). It is reminiscent of safe type-casting in type `dynamic` and of the type-equality assertions of Pašalić et al. (2002).

As above, the functions in the security kernel are generally simple and not recursive. In contrast, the *client* code whose safety we eventually wish to assure (`rev` in our example) is recursive. This pattern recurs throughout this paper: in the most complex example, Knuth-Morris-Pratt string search, the client code is imperative and nonprimitively recursive, yet the security kernel relies merely on addition, subtraction, and comparison.

Of course, safety depends on the fact that the capability is only issued for a nonempty list. Thus the security kernel has to be verified, perhaps formally. Because `FL.fullList` is opaque, we need only check that `indeed` issues the capability only when the list is nonempty. This claim is straightforward to prove formally:

Metavariables

Term variables	x, y, z
Terms	E
Type variables	s, t
Types	N, T, W
Natural numbers	m, n

Typing rules shared between Strict and Lax

$$\frac{T : \star \quad T' : \star}{T \to T' : \star} \qquad \frac{\begin{array}{c}[t : \star] \\ \vdots \\ T' : \star\end{array}}{\forall t.\, T' : \star} \qquad \frac{}{\mathrm{Int} : \star} \qquad \frac{T : \star}{\mathrm{List}\, T : \star} \qquad \frac{T : \star}{\mathrm{List}^{+}\, T : \star}$$

$$\frac{T : \star \quad \begin{array}{c}[x : T] \\ \vdots \\ E : T'\end{array}}{\lambda x.\, E : T \to T'} \qquad \frac{E_1 : T \to T' \quad E_2 : T}{E_1 E_2 : T'} \qquad \frac{\begin{array}{c}[t : \star] \\ \vdots \\ E : T'\end{array}}{\Lambda t.\, E : \forall t.\, T'} \qquad \frac{E : \forall t.\, T' \quad T : \star}{ET : T' \{t \mapsto T\}}$$

$$\frac{}{n : \mathrm{Int}} \qquad \frac{T : \star}{\mathrm{nil} : \mathrm{List}\, T} \qquad \frac{E_1 : T \quad E_2 : \mathrm{List}\, T}{E_1 :: E_2 : \mathrm{List}\, T}$$

$$\frac{E : \mathrm{List}\, T \quad E_1 : W \quad E_2 : \mathrm{List}^{+}\, T \to W}{\mathrm{indeed}\; E\; E_1\; E_2 : W} \qquad \frac{E : \mathrm{List}^{+}\, T}{\mathrm{head}\; E : T} \qquad \frac{E : \mathrm{List}^{+}\, T}{\mathrm{tail}\; E : \mathrm{List}\, T}$$

Typing rule in Strict

$$\frac{E_1 : T \quad E_2 : \mathrm{List}\, T}{\mathrm{nonempty}\; (E_1 :: E_2) : \mathrm{List}^{+}\, T}$$

Typing rule in Lax

$$\frac{E : \mathrm{List}\, T}{\mathrm{nonempty}\; E : \mathrm{List}^{+}\, T}$$

Fig. 1. Formalizing empty-list checking

- On one hand, we could prove along the operational lines of Moggi and Sabry (2001) and Walker et al. (2000) that no expression evaluates to an empty `fullList`.
- Or, we could show along the denotational lines of Launchbury and Jones (1995) that the functions in `FL` are parametric even when the logical relation for `fullList` excludes the empty `fullList` (Mitchell and Meyer 1985).

Either way, our proof is simpler than these authors' (for example, the logical relation may be unary rather than binary) because our safety condition is simpler (for example, we do not prove that the `fullList` does not escape some dynamic extent of execution).

2.2 Formalization

We now formally verify safety, by translating from a language called Strict to a language called Lax. We specify the security kernel by Strict and implement it in Lax. Figure 1 shows the type systems of both Strict and Lax, which extend System F and differ in only one rule. Strict's distinguished typing rule looks 'fancy' and has the flavor of dependent types. However, it simply ascribes a type to an expression of a particular syntactic structure, just like the other, more familiar rules.

The dynamic small-step semantics of these languages are the same and standard. The only interesting reduction rules are the following (where E_1, etc. are all values):

$$\begin{aligned} \text{head}\,(\text{nonempty}\ E_1 :: E_2) &\rightharpoonup E_1 \\ \text{tail}\,(\text{nonempty}\ E_1 :: E_2) &\rightharpoonup E_2 \\ \text{indeed nil}\ E_1\ E_2 &\rightharpoonup E_1 \\ \text{indeed}\,(E :: E')\ E_1\ E_2 &\rightharpoonup E_2(\text{nonempty}\ E :: E') \end{aligned} \tag{1}$$

For example, both Strict and Lax admit the following transition, which starts to compute the head of the list $5 :: 7 :: \text{nil}$.

$$\text{indeed}\,(5 :: 7 :: \text{nil})\ 0\ (\lambda x.\, \text{head}\ x) \rightharpoonup (\lambda x.\, \text{head}\ x)\big(\text{nonempty}\ (5 :: 7 :: \text{nil})\big) \tag{2}$$

We have formally proved, in Twelf,[1] that the type system of Strict is sound: it essentially performs abstract interpretation conservatively to ensure that a well-typed Strict program never tries to take the head or tail of an empty list. For example, the two terms in (2) have the following typing derivations.

$$\frac{\begin{array}{c}\vdots\\ 5 :: 7 :: \text{nil} : \text{List Int}\end{array} \quad \dfrac{}{0 : \text{Int}} \quad \begin{array}{c}\vdots\\ \lambda x.\, \text{head}\ x : \text{List}^{+}\ \text{Int} \to \text{Int}\end{array}}{\text{indeed}\,(5 :: 7 :: \text{nil})\ 0\ (\lambda x.\, \text{head}\ x) : \text{Int}} \tag{3}$$

$$\frac{\begin{array}{c}\vdots\\ \lambda x.\, \text{head}\ x : \text{List}^{+}\ \text{Int} \to \text{Int}\end{array} \quad \dfrac{\dfrac{}{5 : \text{Int}} \quad \begin{array}{c}\vdots\\ 7 :: \text{nil} : \text{List Int}\end{array}}{\text{nonempty}\ (5 :: 7 :: \text{nil}) : \text{List}^{+}\ \text{Int}}}{(\lambda x.\, \text{head}\ x)\big(\text{nonempty}\ (5 :: 7 :: \text{nil})\big) : \text{Int}} \tag{4}$$

The soundness of Strict relies on its distinguished typing rule: this is the only introduction rule for the type $\text{List}^{+}\ T$. Values of that type can only be constructed by attaching a special data constructor "nonempty" to a list. The typing rule of Strict stipulates that "nonempty" must be attached to a manifestly nonempty list.

In contrast, the typing system of Lax permits attaching "nonempty" to any list; therefore, the type system of Lax is not sound. For example, the term

$$\text{head}\,(\text{nonempty nil}) \tag{5}$$

is typable but stuck. However, Lax has the advantage that it is trivial to embed into a programming language like OCaml or Haskell, because it replaces Strict's

[1] http://pobox.com/~oleg/ftp/Computation/safety.elf

fancy typing rule for “nonempty” with a dull one. Clearly, “nonempty” is akin to a `newtype` in Haskell and needs no run-time representation.

To relate these two languages, we define a syntax-directed translation from Strict to Lax. In this section, this *relaxation* map is simply the identity function on terms and types. Relaxation preserves typing, valuehood, and (the transitive closure of) transitions.

We call a Lax program *sandboxed* if it is typable using only those typing rules shared between Strict and Lax (that is, it does not use “nonempty”). Clearly, every (well-typed) sandboxed Lax program is the relaxation of some (well-typed) Strict program. Because a well-typed Strict program does not get stuck, neither does a well-typed sandboxed Lax program, even though the latter may well transition to a non-sandboxed term such as (2), which uses “nonempty”.

When we embed Lax into Haskell or OCaml, the implementation of the rules (1) becomes the security kernel. Sandboxing stipulates that the data constructor “nonempty” may appear in the kernel only, not in the embedding of a sandboxed Lax program. We enforce this stipulation using Haskell or OCaml’s module system. The security kernel is correct if it implements the reduction rules of (1). We can check that the kernel is correct by inspecting it informally or verifying it formally.

3 Array-bound checking

We next illustrate our approach on the problem of array-bound checking in binary search (Xi and Pfenning 1998). This example involves array-index arithmetic and recursion. All indexing operations are statically guaranteed safe without run-time overhead. We show OCaml code below; the same idea works in Haskell 98 with higher-rank types. The type annotations we require are far simpler than those in Dependent ML. Also, only the small security kernel needs annotations, not the rest of the program.

Below is Xi and Pfenning’s original code for the example in Dependent ML (Xi and Pfenning 1998; Figure 3; see also http://www.cs.cmu.edu/~hwxi/DML/examples/).

```
datatype 'a answer = NONE | SOME of int * 'a

assert sub <| {n:nat, i:nat | i < n } 'a array(n) * int(i) -> 'a
assert length <| {n:nat} 'a array(n) -> int(n)

fun('a){size:nat}
bsearch cmp (key, arr) =
let
    fun look(lo, hi) =
        if hi >= lo then
            let
                val m = lo + (hi - lo) div 2
                val x = sub(arr, m)
            in
                case cmp(key, x) of LESS => look(lo, m-1)
                                  | EQUAL => (SOME(m, x))
                                  | GREATER => look(m+1, hi)
            end
```

```
        else NONE
    where look <| {l:nat, h:int | 0 <= l <= size /\ 0 <= h+1 <= size}
                  int(l) * int(h) -> 'a answer
in
    look (0, length arr - 1)
end
where bsearch <| ('a * 'a -> order) ->
                 'a * 'a array(size) -> 'a answer
```

The text after `<|` are dependent-type annotations that the programmer must specify.

3.1 An attempt: parameterized modules

This example differs from the one in Section 2 in an important way. There, we merely need to distinguish a nonempty list from a general list, so one abstract type `'a fullList` is enough. Here, to ensure that an array of size n is only accessed with non-negative indices less than n, we need two abstract types for each n: one for arrays of size n and one for non-negative indices less than n. That is, we need two infinite type families, parameterized by the array size n. Because the value n is only known at run-time, dependent types seem called for.

Even though OCaml is usually not considered dependently typed, we can build such type families in OCaml, by encapsulating type declarations into a module parameterized over a value signature, and instantiating such a module inside a `let` expression (Frisch 2006). The interface and implementation of our trusted kernel would then look like the following. [2]

```
module TrustedKernel(A : sig val length : int end) : sig
   type 'a barray
   type bindex
   val brand : 'a array -> 'a barray
   ...
   val bget  : 'a barray -> bindex -> 'a
end = struct
   type 'a barray = 'a array
   type bindex = int
   let brand a = assert (Array.length a = A.length); a
   ...
   let bget = Array.unsafe_get
end
```

```
let bsearch cmp (key, arr) =
  let module BA = TrustedKernel
        (struct let length = Array.length arr end) in
  let arr = BA.brand arr in ...
```

A (non-bottom) value of type `'a BA.barray` is an array of size n, and a (non-bottom) value of type `BA.bindex` is a non-negative index less than n, where n is the size of the array `arr` in scope for constructing the instance of `module BA`. Consequently, if the expression `BA.get a i` is well typed and `a` and `i` are non-bottom values, then

[2] The complete code is available online at http://pobox.com/~oleg/ftp/ML/eliminating-array-bound-check-functor.ml

the index `i` is within the bounds of the array `a`.[3]

Because `TrustedKernel` is stateless, it can assure array-bound safety by relying merely on the fact that instances of `module BA` for different values of `length` are type-incompatible in OCaml. However, in the general case where the kernel has effects such as state, we need *generative* type abstraction: any two instantiations of `module BA` should be type-incompatible, even with the same `length` (Dreyer et al. 2003). This generativity also corresponds to the "fresh region index" of Moggi and Sabry (2001; Figure 4).

Alas, functors are not generative in OCaml. They are in SML, but most implementations (including SML/NJ) do not allow constructing a module inside `let`.

3.2 The solution: higher-rank types

Our solution is to emulate the generative module `BA` above using higher-rank types (Mitchell and Plotkin 1988; Russo 1998; Shao 1999a,b; Shields and Peyton Jones 2001, 2002). The OCaml code below corresponds as closely to the Dependent ML code above as possible, yet is more amenable to formalization. The emulation also works in Haskell, which does not have local module expressions.[4]

Our solution uses not only higher-rank but also higher-kind types. Rather than using types like `'a barray` and `bindex`, we parameterize them to form types like `('s,'a) barray` and `'s bindex`. We call the extra type parameter `'s` a *brand*.[5] Each possible size is represented by a type: perhaps `unit` represents 0, `unit list` represents 1, `unit list list` represents 2, and so on. Our kernel use these type-level proxies to brand arrays of that size and indices within that range (Pašalić et al. 2002). We can think of these types as a separate kind `Int`. We do not care which type represents which size; in fact, these types do not affect the run-time representation of arrays and indices at all, and we use higher-rank polymorphism to generate the types arbitrarily. Hence these types are called *phantom types* (Fluet and Pucella 2006).

Suppose that some brand `s` represents some size n. We ensure that a (non-bottom) value of type `(s,'a) barray` is an array of the length n, and a (non-bottom) value of type `s bindex` is a non-negative index less than n. This way, a branded index of the latter type is always in range for a branded array of the former type. We also define types `'s bindexL` and `'s bindexH`, so that a (non-bottom) value of type `s bindexL` is a non-negative index, and a (non-bottom) value of the type `s bindexH` is an index i less than n.

[3] The code above has a small inefficiency, in that the last three lines determine the length of the same array twice: we determine the length of an array so as to instantiate the `TrustedKernel`; the brand function will again obtain the run-time length of the array to make sure it matches the length associated with the particular instantiation of `TrustedKernel`. We may try to parameterize the trusted kernel by the array itself rather than by its length and so define the kernel as

```
module TrustedKernel(A : sig type e val a : e array end) ...
```

However, we must explicitly set the type of the array elements when instantiating `TrustedKernel`. That type cannot be polymorphic (Frisch 2006).

[4] Our Haskell code is available online at `http://pobox.com/~oleg/ftp/Haskell/eliminating-array-bound-check-literally.hs`. A slightly more general version at `http://pobox.com/~oleg/ftp/Haskell/eliminating-array-bound-check.lhs` accounts for Haskell arrays with arbitrary lower and upper bounds.

[5] "To burn a distinctive mark into or upon with a hot iron, to indicate quality, ownership, etc., or to mark as infamous (as a convict)." —*The Collaborative International Dictionary of English*

Our security kernel is a module with the following signature.

```
sig
   type ('s,'a) barray
   type 's bindex
   type 's bindexL
   type 's bindexH

   type ('w,'a) brand_k =
       {bk : 's . ('s,'a) barray * 's bindexL * 's bindexH -> 'w}
   val brand : 'a array -> ('w,'a) brand_k -> 'w

   val bmiddle : 's bindex -> 's bindex -> 's bindex
   val index_cmp : 's bindexL -> 's bindexH ->
                   (unit -> 'w) ->                    (* if >  *)
                   ('s bindex -> 's bindex -> 'w) -> (* if <= *)
                   'w
   val bsucc : 's bindex -> 's bindexL
   val bpred : 's bindex -> 's bindexH

   val bget  : ('s,'a) barray -> 's bindex -> 'a
   val unbi : 's bindex -> int
end
```

As in Section 2, we use continuation-passing style to avoid tagging overhead. The branding operation `brand` has an essentially higher-rank type: because higher-rank types in OCaml are limited to records, we define a record type `brand_k` with a universally quantified type variable `'s`. Besides branding arrays and indices, the kernel also performs range- (hence, brand-) preserving operations on indices: `bsucc` increments an index; `bpred` decrements an index; and `bmiddle` averages two indices. The operation `index_cmp l h k1 k2` compares an `indexL` with an `indexH`. If the former does not exceed the latter, we convert both values to `bindex` and pass them to the continuation `k2`. Otherwise, we evaluate the thunk `k1`.

Given such a kernel, we can write the binary search function as follows.

```
let bsearch' cmp (key,(arr,lo,hi)) =
   let rec look lo hi = index_cmp lo hi (fun () -> None)
     (fun lo' hi' ->
       let m = bmiddle lo' hi' in
       let x = bget arr m in
       let cmp_r  = cmp (key,x) in
       if cmp_r < 0 then look lo (bpred m)
       else if cmp_r = 0 then Some (unbi m, x)
       else look (bsucc m) hi)
   in
   look lo hi

let bsearch cmp (key, arr) =
   brand arr {bk = fun arrb -> bsearch' cmp (key, arrb)}
```

The code follows Xi and Pfenning's Dependent ML code as literally as possible, modulo syntactic differences between SML and OCaml. It is instructive to compare their code with ours. Our algorithm is just as efficient: each iteration involves one middle-index computation, one element comparison, one index comparison, and one index increment or decrement. No type annotation is needed. In contrast, the Dependent ML code requires complex dependent-type annotations, even for internal functions such as `look`. The inferred types for our functions are below.

```
val bsearch' :
  ('a * 'b -> int) ->
  'a * (('c, 'b) barray * 'c bindexL * 'c bindexH) ->
  (int * 'b) option = <fun>
val bsearch :
  ('a * 'b -> int) -> 'a * 'b array -> (int * 'b) option = <fun>
```

To complete the code, we need to implement the trusted kernel as a module. The full code is available online;[6] given below are a few notable excerpts. First, we need a way to create values of the type `('s,'a) barray` that ensure that a value of the type `(s,a) barray` is an array of elements of type `a` whose size is represented by the type proxy `s`. Thus we need to generate type proxies for array sizes encountered at run time. McBride (2002) and Kiselyov and Shan (2004) show one such approach in Haskell, which explicitly constructs a type to represent each value. Hayashi (1994), Xi and Pfenning (1998), and Stone (2000; Stone and Harper 2000) also represent values at the type level, using *singleton types*. These approaches better expose the connection between branding and dependent types, but they are more general than we need here. We simply generate a fresh type eigenvariable.

```
let brand a k = k.bk (a, 0, Array.length a - 1)
```

The function `bmiddle` is a brand- (that is, range-) preserving operation on branded indices. Its type says that all indices involved have the same brand—that is, the same value range.

```
val bmiddle : 's bindex -> 's bindex -> 's bindex
let bmiddle i1 i2 = i1 + (i2 - i1)/2
```

The type of `bmiddle` corresponds to the proposition

$$0 \le i_1 < n \quad \wedge \quad 0 \le i_2 < n \quad \to \quad 0 \le \texttt{bmiddle}\ i_1\ i_2 < n,$$

where n is the integer represented by the type proxy `s`. The implementation for `bmiddle` delivers a certificate for the proposition.

```
let index_cmp i j ong onle = if i <= j then onle i j else ong ()
let bsucc = succ and bpred = pred
```

3.3 Formalization

As in Section 2.2, we can verify safety by a syntactic translation from a sound, fancy language called Strict to an unsound, dull language called Lax. Figure 2 shows how we extend Strict and Lax from Figure 1 with constructs for array-bound checking. We model an n-element array by an n-element list, whose first element has the index 1. Crucially, we add types $\bar{n}$ to Strict, which represent natural numbers (array sizes) n. To maintain compatibility with Lax, these types $\bar{n}$ are of kind $\star$ rather than a separate kind of type-level naturals.

[6] http://pobox.com/~oleg/ftp/ML/eliminating-array-bound-check-literally.ml

Additional typing rules shared between Strict and Lax

$$\frac{N : \star \quad T : \star}{\mathrm{List}^N\, T : \star} \qquad \frac{N : \star}{\mathrm{Int}^N : \star} \qquad \frac{N : \star}{\mathrm{Int}^N_{\mathrm{L}} : \star} \qquad \frac{N : \star}{\mathrm{Int}^N_{\mathrm{H}} : \star}$$

$$\frac{E : \mathrm{List}\, T \quad E' : \forall s.\, \mathrm{List}^s\, T \to \mathrm{Int}^s_{\mathrm{L}} \to \mathrm{Int}^s_{\mathrm{H}} \to W}{\mathrm{brand}\; E\; E' : W} \qquad \frac{E_1 : \mathrm{List}^N\, T \quad E_2 : \mathrm{Int}^N}{\mathrm{get}\; E_1\; E_2 : T}$$

$$\frac{E_{\mathrm{L}} : \mathrm{Int}^N_{\mathrm{L}} \quad E_{\mathrm{H}} : \mathrm{Int}^N_{\mathrm{H}} \quad E_1 : W \quad E_2 : \mathrm{Int}^N \to \mathrm{Int}^N \to W}{\mathrm{compare}\; E_{\mathrm{L}}\; E_{\mathrm{H}}\; E_1\; E_2 : W} \qquad \frac{E_1 : \mathrm{Int}^N \quad E_2 : \mathrm{Int}^N}{\mathrm{middle}\; E_1\; E_2 : \mathrm{Int}^N}$$

$$\frac{E : \mathrm{Int}^N}{\mathrm{succ}\; E : \mathrm{Int}^N_{\mathrm{L}}} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{pred}\; E : \mathrm{Int}^N_{\mathrm{H}}} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{unbi}\; E : \mathrm{Int}}$$

Additional typing rules in Strict

$$\frac{}{\bar{n} : \star} \qquad \frac{E_1 : T \quad \ldots \quad E_n : T}{\mathrm{array}\; E_1 :: \ldots E_n :: \mathrm{nil} : \mathrm{List}^{\bar{n}}\, T} \qquad \frac{1 \le m \le n}{m_{\mathrm{I}} : \mathrm{Int}^{\bar{n}}} \qquad \frac{1 \le m}{m_{\mathrm{L}} : \mathrm{Int}^{\bar{n}}_{\mathrm{L}}} \qquad \frac{m \le n}{m_{\mathrm{H}} : \mathrm{Int}^{\bar{n}}_{\mathrm{H}}}$$

Additional typing rules in Lax

$$\frac{E : \mathrm{List}\, T \quad N : \star}{\mathrm{array}\; E : \mathrm{List}^N\, T} \qquad \frac{N : \star}{m_{\mathrm{I}} : \mathrm{Int}^N} \qquad \frac{N : \star}{m_{\mathrm{L}} : \mathrm{Int}^N_{\mathrm{L}}} \qquad \frac{N : \star}{m_{\mathrm{H}} : \mathrm{Int}^N_{\mathrm{H}}}$$

Fig. 2. Formalizing array-bound checking

The dynamic semantics of Strict [7] follows the type system and is standard. For example, it contains the following small-step transitions, which start to compute the middle element of the list $5 :: 7 :: \mathrm{nil}$.

$$\begin{aligned} &\mathrm{brand}\; (5 :: 7 :: \mathrm{nil})\; \big(\Lambda s.\, \lambda xyz.\, \mathrm{compare}\; y\; z\; 0\; \lambda yz.\, \mathrm{get}\; x\; (\mathrm{middle}\; y\; z)\big) \\ &\quad \rightharpoonup \big(\Lambda s.\, \lambda xyz.\, \mathrm{compare}\; y\; z\; 0\; \lambda yz.\, \mathrm{get}\; x\; (\mathrm{middle}\; y\; z)\big) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \bar{2}\; (\mathrm{array}\; 5 :: 7 :: \mathrm{nil})\; 1_{\mathrm{L}}\; 2_{\mathrm{H}} \\ &\quad \rightharpoonup^* \mathrm{get}\; (\mathrm{array}\; 5 :: 7 :: \mathrm{nil})\; 1_{\mathrm{I}} \end{aligned} \tag{6}$$

The type system of Strict is sound as before; in particular, a well-typed Strict program never tries to access an array beyond its bounds. For example, the first and last terms above have the following typing derivations.

$$\frac{\dfrac{\vdots}{5 :: 7 :: \mathrm{nil} : \mathrm{List\ Int}} \qquad \dfrac{\vdots}{\begin{array}{l}\Lambda s.\, \lambda xyz.\, \mathrm{compare}\; y\; z\; 0\; \lambda yz.\, \mathrm{get}\; x\; (\mathrm{middle}\; y\; z) \\ \quad : \forall s.\, \mathrm{List}^s\, \mathrm{Int} \to \mathrm{Int}^s_{\mathrm{L}} \to \mathrm{Int}^s_{\mathrm{H}} \to \mathrm{Int}\end{array}}}{\mathrm{brand}\; (5 :: 7 :: \mathrm{nil})\; \big(\Lambda s.\, \lambda xyz.\, \mathrm{compare}\; y\; z\; 0\; \lambda yz.\, \mathrm{get}\; x\; (\mathrm{middle}\; y\; z)\big) : \mathrm{Int}} \tag{7}$$

$$\frac{\dfrac{\vdots}{\mathrm{array}\; 5 :: 7 :: \mathrm{nil} : \mathrm{List}^{\bar{2}}\, \mathrm{Int}} \qquad \dfrac{\overline{1 \le 1 \le 2}}{1_{\mathrm{I}} : \mathrm{Int}^{\bar{2}}}}{\mathrm{get}\; (\mathrm{array}\; 5 :: 7 :: \mathrm{nil})\; 1_{\mathrm{I}} : \mathrm{Int}} \tag{8}$$

[7] The Twelf formalization is available at http://pobox.com/~oleg/ftp/Computation/safety-array.elf

As in Section 2.2, the soundness of Strict depends on its special typing rules for the distinguished data constructors such as "array". In contrast, the corresponding typing rules in Lax remove the side conditions on array lengths and indices, and so permit constructing values of the type List^N Int for any Lax type N whatsoever. For example, the transition

$$\begin{aligned} &\text{brand}\,(5 :: 7 :: \text{nil})\,\big(\Lambda s.\,\lambda xyz.\,\text{compare}\;y\;z\;0\;\lambda yz.\,\text{get}\;x\;(\text{middle}\;y\;z)\big) \\ &\quad\rightharpoonup \big(\Lambda s.\,\lambda xyz.\,\text{compare}\;y\;z\;0\;\lambda yz.\,\text{get}\;x\;(\text{middle}\;y\;z)\big) \\ &\qquad\qquad\qquad\qquad N\;(\text{array}\;5 :: 7 :: \text{nil})\;1_{\text{L}}\;2_{\text{H}}. \end{aligned} \tag{9}$$

is type-preserving in Lax for any Lax type N (say Int, but not $\bar{2}$ because $\bar{2}$ is only a Strict type). Nothing in Lax prevents constructing well-typed values such as array nil : List^NInt and $5_{\text{I}} : \text{Int}^N$, which, when passed to "get", cause the computation to become stuck. Without restricting the use of "array" and index constructors, the type system of Lax is unsound.

We introduce these restrictions by sandboxing Lax programs, as in Section 2.2. Sandboxed programs must be typable in Lax using only the typing rules shared with Strict. As before, we define relaxation, a syntax-directed translation from Strict to Lax. This time relaxation is not just identity, but maps $\bar{n}$ to the N in (9). Still, relaxation preserves typing, valuehood, and (the transitive closure of) transitions. Because again every (well-typed) sandboxed Lax program is the relaxation of some (well-typed) Strict program, a well-typed sandboxed Lax program does not get stuck, even though it may well transition to a non-sandboxed term such as (9), which uses "array".

We have mechanized these type soundness arguments in Twelf, slightly less trivially than in Section 2.2. One crucial lemma is that, if a Strict value has a type of the form Int^T (where T is any type), then T must be of the form $\bar{n}$ (where n is a natural number). Intuitively, this lemma means that the type system does not lose any precision due to our not introducing a separate kind for type-level naturals.

3.4 Multiple arrays of various sizes

A more complex example[8] (suggested by a user and a reviewer) is folding over multiple arrays of various sizes. Our goal is a Haskell function

```
marray_fold :: (Ix i, Integral i) =>
               (seed -> [e] -> seed) -> seed -> [Array i e] -> seed
```

which folds over an arbitrary number of arrays, whose lower and upper bounds may differ. The index ranges of some arrays do not even have to overlap and may be empty. Neither the number of arrays to process nor their bounds are statically known, yet we guarantee that all array accesses are within bounds. The key function in this example brands multiple arrays with a type proxy that represents the intersection of their index ranges:

```
brands :: (Ix i, Integral i) => [Array i e] ->
          (forall s. ([BArray s i e], BLow s i, BHi s i) -> w) ->
          w -> w
```

[8] http://pobox.com/~oleg/ftp/Haskell/eliminating-mult-array-bound-check.lhs

```
brands [arr] consumer onempty =
    brand arr (\ (barr,bl,bh) -> consumer ([barr],bl,bh)) onempty
brands (a:arrs) consumer onempty =
    brands arrs (\bbars -> brand_merge bbars a consumer onempty)
                onempty

brand_merge :: (Ix i, Integral i) =>
         ([BArray s i e], BLow s i, BHi s i)
         -> Array i e
         -> (forall s'. ([BArray s' i e], BLow s' i, BHi s' i) -> w)
         -> w -> w
brand_merge (bas,(BLow bl),(BHi bh)) (a :: Array i e) k kempty =
    let (l,h) = bounds a
        l' = max l bl
        h' = min h bh
    in if l' <= h' then
             k (((BArray a)::BArray () i e) :
                 (map (\ (BArray a) -> BArray a) bas),
                BLow l', BHi h')
       else kempty
```

Typing this example in a genuinely dependent type system appears quite challenging.

4 Types as static capabilities

In the style just exemplified, the programmer begins verification by building a domain-specific *kernel* module that represents and defends the desired safety condition. This kernel provides *capabilities* to other modules so that they can work safely. Many safety conditions can be expressed using types as proxies for values.

We now describe each step and the language support they need in turn.

4.1 A domain-specific kernel of trust

Program verification typically begins by fixing an assertion language. Given a program, its safety condition is then extracted automatically or specified manually before being proven. The soundness of the proof checker guarantees that a verified program will behave safely.

While this approach lets the designer of the verification framework prove soundness once and for all, the desired safety condition may not reside at the same level of abstraction as the assertion language. Such a mismatch makes the safety assertion burdensome to construct formally and brittle to prove automatically. For example, if the assertions speak of bytes and registers, then it is hard to verify that modular numbers of different moduli are never mixed together. It takes a lot of work today to translate among layers of representation and verify their correspondence, so this approach works best at a fixed (often low) level of abstraction, as in proof-carrying code (Necula 1997) and typed assembly language (Morrisett et al. 1999).

We let the programmer design more of the assertion language. For example, it is uncontroversial to let the programmer specify a set of events that need to be checked using temporal logic, rather than fixing a set of events (such as operating-system calls) to track. This way, even given that the framework is sound, whoever uses the framework must ensure that the assertions soundly express the safety condition desired. In exchange, the programmer can mold the assertion language, for example to express the safety condition for an array index not as a conjunction of inequalities but as an atomic assertion whose meaning is not known to the verifier.

Now that the verification framework no longer knows what the assertions mean, it can no longer build in axioms to justify atomic assertions: because the programmer never defines events in terms of system calls, the framework needs to be told when events occur; because the programmer never defines array bounds in terms of inequalities, the framework needs to be told how to judge an array index in bounds. We call this knowledge a *kernel* of trust, which the programmer creates to represent domain-specific safety conditions.

By extending the kernel of trust, the programmer can verify new safety conditions as needed. Each extension must be scrutinized closely to preserve soundness. In exchange, we gain a "continuum of correctness" in which the programmer can verify more safety conditions as needed.

An expressive programming language allows the user to define and combine a domain-specific library of components. In this regard, the kernel of trust is like any other domain-specific language: its construction relies on succinct facilities for higher-order abstraction.

4.2 Capabilities for extending trust

Our "verifier", the type system, does not track system calls or solve inequalities, but propagates certificates of assertions from the user-defined kernel of trust. Safety then extends from the kernel to the rest of the program. It turns out that type systems are good at this propagation: we trust types.

More precisely, we represent trust by *type eigenvariables*. A type system that supports either higher-rank polymorphism or existential types generates a type eigenvariable fresh in the universal introduction or existential elimination rule (Pierce and Sumii 2000; Reynolds 1983; Rossberg 2003). An opaque type from another module is another instance of a type eigenvariable (Mitchell and Plotkin 1988). Type eigenvariables are good for representing trust to be propagated, because they are

- unforgeable (so only the kernel of trust can manufacture them),
- opaque (so their identity is the only information they convey), and
- propagated by type inference (so they extend trust from the kernel to the rest of the application).

In other words, type eigenvariables turn a static language of types into a *capability language* (Miller et al. 2000).

The notion of a capability (Miller et al. 2000; Section 3) originated in OS design. A capability is a "protected ability to invoke arbitrary services provided by other processes" (Wulf et al. 1974). For a language system to support capabilities (Miller et al. 2000), access to a particular functionality (for example, access to a collection) must only be via an unforgeable, opaque, and propagated *handle*. For a computation to use a handle, it must have created the handle, received it from another computation, or looked it up in the initial environment. To use a handle, a computation can only propagate it or perform a set of predetermined actions (for example, read an array).

We represent capabilities as types, so we express safety conditions in types, as in dependent-type programming. If a program type-checks, then the type system and the kernel of trust together verify that the safety conditions hold in any run of the program. In most cases, this static assurance costs us no run-time overhead. In the remaining cases, an optimizing compiler can discover and eliminate statically apparent identity functions at compile time. By guaranteeing safety statically, we can avoid (often excessive) run-time safety checks such as array bound checks.

A capability is commonly viewed as "a pairing of a designated process with a set of services that the process provides" (Miller et al. 2000; Section 3). Hence a special case of a capability, illustrated in Section 2, is an abstract data type. An abstract data type certifies the invariants internal to its implementation: if the implementation preserves the invariants, then the invariants are preserved throughout the application because only the implementation can manipulate values of the abstract type. In general, a capability to access an object certifies the safety condition of that object.

Another example is restricting the IO monad to a few actions. In Haskell, many tasks require the IO monad: file I/O, invoking foreign functions, asking the OS for the time of day or a random number, and so on. The IO monad contains many actions, so a piece of code that can use the IO monad to generate a random number can also use IO to overwrite files on disk and otherwise wreck any guarantee on the code. Instead of providing the code with the IO monad directly, we can provide an monad `m`, where `m` is a type eigenvariable, along with an action of type `m Int` that generates a random integer. Although the program eventually instantiates `m` with the IO monad, the opacity of the type eigenvariable `m` guarantees that the code can only generate random numbers. This basic idea appears in the encapsulation of mutable state by Moggi and Sabry (2001). It is also used realistically in the Zipper file-system project, to statically enforce process separation.

4.3 Static proxies for dynamic values

To express assertions involving run-time values, we associate each value with a type, such that type equality entails value equality. We call these types *proxies* for the values (Pašalić et al. 2002).

The same proxy appearing in the types of multiple values may make additional operations available from the kernel. For example, the branding described in Section 3.2 lets us access an array at an index that is within the bounds of the *same* array. This availability is known as *rights amplification* in the capabilities literature. Miller et al. (2000) writes:

> With rights amplification, the authority accessible from bringing two references together can exceed the sum of authorities provided by each individually. The classic example is the can and the can-opener—only by bringing the two together do we obtain the food in the can.

5 Discussion

We have argued that the Hindley-Milner type system with higher-rank types is a static capability language with rights amplification. Our take on program verification is not to prove the safety conditions from a fixed foundation but to rely on the programmer's trust in a domain-specific kernel. Our technique works in existing languages like Haskell and OCaml, and is compatible with their facilities like mutable cells, native arrays, and general recursion. It requires a modicum of type annotations in the kernel only.

We use types to certify properties of values. For example, the type `s bindex` in Section 3.2 certifies that the index is a non-negative integer less than the array size represented by `s`. The use of an abstract data type whose values can only be produced by a trusted kernel, and the use of a type system to guarantee this last property, is due to Robin Milner in the design of Edinburgh LCF back in the early 1970s (Gordon 2000). (Incidentally, the language ML—whose early offspring OCaml we use in this paper—was originally designed as a scripting language for the LCF prover.) Our branding technique builds on this fundamental idea using an infinite family of abstract data types, indexed by a type proxy for a run-time value. Our approach still has the serious limitation that we do not produce independently statically checkable certificates.

5.1 On trusting trust

Our lightweight approach depends on a trusted kernel. Because we expect this kernel to vary across applications and change over time, it is harder to trust the kernel, compared to a genuine dependent type system. We have only optimistic speculations to offer at this point.

On one hand, a small kernel may be more amenable to formal treatment than the entire application at once. Even in our most complex examples, verifying imperative and nonprimitively recursive code, our trusted kernel had no recursive functions (and at most relied on simple arithmetic). Seen this way, delineating a kernel of trust is simply a modular strategy towards complete verification. This strategy straddles the line between proof assistants and programming environments, calling for their further integration.

On the other hand, programmers may be more productive, and verification failures more informative, if the framework does not force verifying the part of correctness that is closest to the foundations first. After all, successive refinement of (sketches of) proofs is a time-tested technique. Moving along this "continuum of correctness" may also give a better idea where the code tends to have bugs, and hence where to concentrate verification.

5.2 Dependent type systems

Altenkirch et al. (2005; Section 2) survey dependent type systems and their emulations (Dybjer 1991; Martin-Löf 1984; Nordström et al. 1990). Our use of type proxies and run-time verifiable certificates puts us near the dependently-typed system MetaD (Pašalić et al. 2002). Our work may be thought of as yet another poor man's emulation of a dependent type system (Fridlender and Indrika 2000; McBride 2002): instead of putting values in types, we put type proxies for values in types; instead of trusting strongly normalizing terms in type theory, we trust a kernel of uninterpreted capabilities; instead of embracing a new programming practice, we embed in existing programming systems. As Altenkirch et al. write, "many programmers are already finding practical uses for the approximants to dependent types which mainstream functional languages (especially Haskell) admit, by hook or by crook." We are not just exploring a toy however: we can express complex reasoning (such as multiple-array bound checking) on real-world applications (such as interfacing to a database) in existing, well-supported language implementations.

Our lightweight approach reasons about the same topics that dependent type systems and optimizing compilers tend to reason about: control flow, aliasing, and ranges. For example, optimizing compilers often perform range analysis to eliminate run-time array-bound checking. However, our reasoning kernel is exposed as a module, not tucked away in a compiler and hidden from the view of a regular programmer.

5.3 Mixing static and dynamic checking

Static program analyses are rarely exact because they approximate program behavior without knowing dynamic data. The approximation must be conservative, and so the range analysis, for example, may worry that an index is out of bounds of an array although in reality it is not. To reduce the approximation error, an analysis may insert dynamic checks. Exactly the same is the case for lightweight static capabilities, except the programmer rather than the compiler controls where to insert dynamic checks. We would expect the programmer to understand the program better than the compiler does, and hence to know better where dynamic checks are appropriate and where they are excessive.

A good concrete example is using one index to access two arrays of the same size. Suppose that we want to feed a branded array `ba1` to an array-to-array function `compute_array`, which we expect to return another array `a2` of equal size. We then want to access both arrays using one index. Because `ba1` is branded before `a2` is created, we cannot brand the two arrays at the same time as in Section 3.4. Instead, we can forget the branding of `ba1`, compute the array `a2`, and assign `a2` the brand of `ba1` after a run-time test:

```
let a2 = compute_array (unbrand ba1)
in brand_as a2 ba1 on_mismatched_size (fun ba2 -> ...)
```

The arrays `ba1` and `ba2` now have the same brand. We assume the kernel has generic, application-*independent* functions `unbrand` and `brand_as`.

If we can *prove* that `compute_array` yields an array of size equal to that of its argument, then we can make the function return a branded array, and thus eliminate the run-time size test. Because branding can only be done in the kernel, we must put the function into the kernel, after appropriate rigorous (perhaps formal) verification. The programmer decides whether to expand the trusted kernel for a new application, balancing the cost of the run-time check against the cost of verifying the kernel extension.

The `brand_as` approach is similar to the `assert/cast` dynamic test in MetaD (Pašalić et al. 2002). Such a "cop-out" to deciding type equality is necessary anyway in a dependent type system with general recursion, where type equality is not decidable in general (Altenkirch et al. 2005; Section 3).

5.4 Syntactic sugar

Writing conditionals in continuation-passing-style, as we do here, makes for ungainly code. We also miss pattern matching and deconstructors. These syntactic issues arise because neither OCaml nor Haskell was designed for this kind of programs. The ugliness is far from a show stopper, but an incentive to develop front ends to improve the appearance of lightweight static capabilities in today's programming languages.

Acknowledgement

We thank Mark S. Miller, Alain Frisch, and the reviewers and participants of the PLPV workshop.

References

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

Appel, Andrew, and Xavier Leroy. 2006. A list-machine benchmark for mechanized metatheory. In *Proceedings of LFMTP'06: Logical frameworks and meta-languages: Theory and practice (LICS'06 and IJCAR'06 workshop)*, ed. Alberto Momigliano and Brigitte Pientka, 85–97.

Dreyer, Derek R., Karl Crary, and Robert Harper. 2003. A type system for higher-order modules. In *POPL '03: Conference record of the annual ACM symposium on principles of programming languages*, 236–249. New York: ACM Press.

Dybjer, Peter. 1991. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Logical frameworks*, ed. Gérard Huet and Gordon Plotkin, 280–306. Cambridge: Cambridge University Press.

Fluet, Matthew, and Riccardo Pucella. 2006. Phantom types and subtyping. *Journal of Functional Programming*.

Fridlender, Daniel, and Mia Indrika. 2000. Do we need dependent types? *Journal of Functional Programming* 10(4):409–415.

Frisch, Alain. 2006. Re: Eliminating array bounds check. Messages to the Caml-list. http://caml.inria.fr/pub/ml-archives/caml-list/2006/09/bfb2d1de69a1cb092bc147e876ede237.en.html http://caml.inria.fr/pub/ml-archives/caml-list/2006/09/dc4a1068c718487a0ca7637dcf6dc190.en.html.

Gordon, Michael J. C. 2000. From LCF to HOL: a short history. In *Proof, language, and interaction*, ed. G. Plotkin, Colin P. Stirling, and Mads Tofte. MIT Press.

Hayashi, Susumu. 1994. Singleton, union, and intersection types for program extraction. *Information and Computation* 109(1–2):174–210.

Kiselyov, Oleg, and Chung-chieh Shan. 2004. Functional pearl: Implicit configurations—or, type classes reflect the value of types. In *Proceedings of the 2004 Haskell workshop*. New York: ACM Press.

Launchbury, John, and Simon L. Peyton Jones. 1995. State in Haskell. *Lisp and Symbolic Computation* 8(4):293–341.

Martin-Löf, Per. 1984. *Intuitionistic type theory*. Napoli: Bibliopolis.

McBride, Conor. 2002. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming* 12(4–5):375–392.

Miller, Mark S., Chip Morningstar, and Bill Frantz. 2000. Capability-based financial instruments. In *Financial cryptography*, ed. Yair Frankel, vol. 1962 of *Lecture Notes in Computer Science*, 349–378. Springer.

Mitchell, John C., and Albert R. Meyer. 1985. Second-order logical relations (extended abstract). In *Logics of programs*, ed. Rohit Parikh, 225–236. Lecture Notes in Computer Science 193, Berlin: Springer-Verlag.

Mitchell, John C., and Gordon D. Plotkin. 1988. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems* 10(3):470–502.

Moggi, Eugenio, and Amr Sabry. 2001. Monadic encapsulation of effects: A revised approach (extended version). *Journal of Functional Programming* 11(6):591–627.

Morrisett, Greg, David Walker, Karl Crary, and Neal Glew. 1999. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems* 21(3):527–568.

Necula, George C. 1997. Proof-carrying code. In *POPL '97: Conference record of the annual ACM symposium on principles of programming languages*, 106–119. New York: ACM Press.

Nordström, Bengt, Kent Petersson, and Jan M. Smith. 1990. *Programming in Martin-Löf's type theory: An introduction*. http://www.cs.chalmers.se/Cs/Research/Logic/book/.

Pašalić, Emir, Walid Taha, and Tim Sheard. 2002. Tagless staged interpreters for typed languages. In *ICFP '02: Proceedings of the ACM international conference on functional programming*, 218–229. New York: ACM Press.

Pierce, Benjamin, and Eijiro Sumii. 2000. Relating cryptography and polymorphism. Available at http://www.yl.is.s.u-tokyo.ac.jp/~sumii/pub/.

Reynolds, John C. 1983. Types, abstraction and parametric polymorphism. In *Proceedings of 9th IFIP world computer congress, Information Processing '83*, ed. R. E. A. Mason, 513–523. Amsterdam: North-Holland.

Rossberg, Andreas. 2003. Generativity and dynamic opacity for abstract types (extended version). Tech. Rep., Universität Saarbrücken. Also PPDP2003.

Russo, Claudio V. 1998. Types for modules. Ph.D. thesis, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh. Also as Tech. Rep. ECS-LFCS-98-389.

Shao, Zhong. 1999a. Transparent modules with fully syntactic signatures. In *ICFP '99: Proceedings of the ACM international conference on functional programming*, vol. 34(9) of *ACM SIGPLAN Notices*, 220–232. New York: ACM Press.

———. 1999b. Transparent modules with fully syntactic signatures. Tech. Rep. YALEU/DCS/TR-1181, Department of Computer Science, Yale University, New Haven.

Shields, Mark B., and Simon L. Peyton Jones. 2001. First-class modules for Haskell. Tech. Rep., Microsoft Research. http://www.cse.ogi.edu/~mbs/pub/first_class_modules/.

———. 2002. First-class modules for Haskell. In *9th international workshop on foundations of object-oriented languages*, 28–40. New York: ACM Press.

Stone, Christopher A. 2000. Singleton kinds and singleton types. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Also as Tech. Rep. CMU-CS-00-153.

Stone, Christopher A., and Robert Harper. 2000. Deciding type equivalence in a language with singleton kinds. In *POPL '00: Conference record of the annual ACM symposium on principles of programming languages*, 214–227. New York: ACM Press.

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.

Wulf, William A., Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. 1974. Hydra: The kernel of a multiprocessor operating system. *Commun. ACM* 17(6):337–345.

Xi, Hongwei, and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *PLDI '98: Proceedings of the ACM conference on programming language design and implementation*, vol. 33(5) of *ACM SIGPLAN Notices*, 249–257. New York: ACM Press.

Appendix: Knuth-Morris-Pratt string matching in Dependent ML and Haskell

We borrow another involved example from Xi and Pfenning (1998), Knuth-Morris-Pratt string matching (KMP). This algorithm uses mutable arrays whose elements' values determine indices in turn. It also uses the deliberately one-off index -1 as a special flag. Our Haskell code[9] using higher-rank types has the same run-time costs and static guarantees as the Xi and Pfenning's Dependent ML code: all array and string operations are verified to be safe.

The Dependent ML code (http://www.cs.cmu.edu/~hwxi/DML/examples/kmpHard.mini), is quoted below for the sake of reference. The code contains (after `<|`) a fair amount of DML annotations: declarations of dependent types. The function `sub` is a DML array access operation with no bound check. The functions `arrayShift`, `subShift` and `updateShift` are the creator, the accessor and the setter for `shiftArray`, whose element type is dependent on the run-time value, the length of the pattern string.

```
structure KMP =
  struct
    assert sub <| {size:nat, i:int | 0 <= i < size}
                 'a array(size) * int(i) -> 'a
    and length <| {n:nat} 'a array(n) -> int(n)
```

[9] http://pobox.com/~oleg/ftp/Haskell/KMP-deptype.hs

```
  fun{slen:nat, plen:nat}
    kmpMatch(str, pat) =
    let
      type intShift = [i:int| 0 <= i+1 < plen ] int(i)

      assert arrayShift <| {size:nat}
             int(size) * intShift -> intShift array(size)
      and subShift <| {size:nat, i:int | 0 <= i < size}
             intShift array(size) * int(i) -> intShift
      and updateShift <| {size:nat, i:int | 0 <= i < size}
             intShift array(size) * int(i) * intShift -> unit

      val slen = length(str)
      and plen = length(pat)

      val shiftArray = arrayShift(plen, ~1)

      fun loopShift(i, j) = (* calculate the shift array *)
        if (j = plen) then ()
        else
          if sub(pat, j) <> sub(pat, i+1) then
            if (i >= 0) then
               loopShift(subShift(shiftArray, i), j)
            else loopShift(~1, j+1)
          else ((if (i+1 < j)
                then updateShift(shiftArray, j, i+1)
                else ()) <| unit;
               loopShift(subShift(shiftArray, j), j+1))
      where loopShift <| {j:int | 0 < j <= plen}
            intShift * int(j) -> unit

      val _ = loopShift(~1, 1)

      fun loop(s, p) = (* this the main search function *)
        if p < plen then
          if s < slen then
            if sub(str, s) = sub(pat, p) then loop(s+1, p+1)
            else
              if (p = 0) then loop(s+1, p)
              else loop(s, subShift(shiftArray, p-1)+1)
          else ~1
        else s - plen
      where loop <| {s:nat, p:nat | s <= slen /\ p <= plen}
            int(s) * int(p) -> int
    in
      loop(0, 0)
    end
  where kmpMatch <| int array(slen) * int array(plen) -> int
end
```

Our Haskell code extensively uses lightweight static capabilities with rights amplification. To properly model the DML code, we will be using array-like so-called 'packed' Haskell strings. We perform imperative computations in the `ST` monad and use the mutable array data type `STArray` to realize `shiftArray`.

Our main function `kmpMatch` has no correspondence in the DML code, because the latter does not seem to define the handling of empty strings or patterns. It is not clear what happens if the DML function `kmpMatch` is invoked with the empty pattern; probably a compiler error is reported because the (dependent) type `intShift` becomes unpopulated. Our approach however forces us to confront the issue; in particular, to resolve what happens when both the string and the pattern are empty.

```
kmpMatch str pat =
  brandPS str
    (\bstrlen ->
      brandPS pat (\bpatlen -> runST (kmpMatch' bstrlen bpatlen))
        0 -- empty pattern, matches the beginning of (nonempty) string
    )
    (-1)   -- empty string, doesn't match any pattern
```

The KMP algorithm itself, for nonempty `str` and `pat`, is as follows. It rather closely resembles the DML code:

```
kmpMatch' (bstr,slen) (bpat,plen) =
    do
    shiftArray <- arrayShift plen index_m1

    let -- loopShift :: IntShift r -> BIndexP1 r -> ST s ()
        loopShift i j = -- calculate the shift array
          index_p_lt j plen (Else $ return ())
            (\j' -> let i1 = intshift_succ i
                    in if bpat !. j' /= bpat !. i1
                          then index_m_gt i index_m1
                               (Else $ loopShift index_m1
                                         (index_succ j'))
                               (\i' -> do
                                       i'' <- subShift shiftArray i'
                                       loopShift i'' j)
                          else do
                               index_lt i1 j'
                                 (Else $ return ())
                                 (updateShift shiftArray j')
                               i'' <- subShift shiftArray j'
                               loopShift i'' (index_succ j')
            )
    loopShift index_m1 index_p1

    let -- loop :: Nat -> Nat -> ST s Int
        loop s p = -- this the main search function
           nat_p_lt p plen (Else $ return $ (unNat s) - (unP1 plen))
            (\p' ->
              nat_p_lt s slen (Else $ return (-1))
               (\s' ->
                 if bstr !. s' == bpat !. p'
```

```
                    then loop (nat_succ s) (nat_succ p)
                    else index_pred p' (Else $ loop (nat_succ s) p)
                          (\p1 -> do
                                  i <- subShift shiftArray p1
                                  loop s $ i2n (intshift_succ i))
                 )
              )
    loop nat_0 nat_0
```

It is instructive to compare the *inferred* type of `loopShift` or `loop` with the annotations in the corresponding DML code. The annotations cannot be inferred and must be specified by the programmer. The appearance of the Haskell code can be improved if we replace various comparison functions such as `index_p_lt`, `nat_p_lt`, etc. with one (type-class) overloaded infix operator, e.g., `<.`

We now describe the trusted kernel for our Haskell code; the kernel also implements functions that correspond to DML's dependently-typed built-ins `sub`, `length`, `arrayShift`, etc. The kernel uses a number of wrapper types such as `BIndex`, which represent various capabilities. These wrappers are `newtype`s and so have no run-time cost. The data constructors of the wrappers must not be exported from the trusted kernel; only the kernel should be allowed to create the capabilities.

The capabilities such as `BIndex r` or `BPackedString r` are tagged by a phantom type `r`, which is a type proxy for a positive natural number `plen` (the length of a nonempty string). The wrapper types assert particular propositions about the wrapped values and `plen` (neither of which are known at compile time). The functions creating wrapped values must be verified to make sure the propositions hold.

```
newtype BIndex r  = BIndex Int
newtype BPackedString r   = BPackedString PackedString
```

The type `BIndex r` asserts that the wrapped integer i satisfies $0 <= i < plen$ where `plen` is the integer represented by the proxy `r`. This newtype declaration corresponds to DML's `{j:int | 0 <= j < plen}`. Likewise, `BPackedString r` is a type proxy for a nonempty packed string of the size represented by `r`. Since the type `BIndex r` assures that the index is definitely within the bounds of the string `BPackedString r`, we could *safely* use `unsafeIndexPS` to access the element of the packed string:

```
infixl 5 !.
(!.):: BPackedString r -> BIndex r -> Char
(BPackedString s) !. (BIndex i) = indexPS s i
```

We introduce two other type proxies, for offset indices: `BIndexP1 r` asserts that the wrapped integer j satisfies $0 < j <= plen$; `IntShift r` is a type proxy for the integer i such that $0 <= (i+1) < plen$. It is instructive to compare the latter with the DML declaration of the dependent type `intShift`.

```
newtype BIndexP1 r  = BIndexP1 Int
newtype IntShift r  = IntShift Int
```

The type proxy `r` is actually an eigenvariable, introduced by the following function after a check that the packed string (whose length is `plen`) is indeed nonempty:

```
brandPS:: PackedString
```

```
          -> (forall r. (BPackedString r, BIndexP1 r) -> w) -> w -> w
brandPS str k kempty =
    let l = lengthPS str
    in if l > 0 then k (BPackedString str, BIndexP1 l)
    else kempty
```

We also introduce the mutable shiftArray and its getters and setters. The type `BShiftArray` makes it clear that the range of values of all the elements is bounded by the positive integer represented by `r`. Therefore, we could have *safely* used `unsafeReadArray` and `unsafeWriteArray` operations.

```
newtype BShiftArray r s = BShiftArray (STArray s Int (IntShift r))

arrayShift :: BIndexP1 r -> IntShift r -> ST s (BShiftArray r s)
arrayShift (BIndexP1 r) e = newArray (0,r) e >>= return . BShiftArray

subShift :: BShiftArray r s -> BIndex r -> ST s (IntShift r)
subShift (BShiftArray arr) (BIndex i) = readArray arr i

updateShift :: BShiftArray r s -> BIndex r -> IntShift r -> ST s ()
updateShift (BShiftArray arr) (BIndex i) v = writeArray arr i v
```

The rest of the kernel is a (quite general purpose) index operation library. To save space, we elide repetitive fragments; the full code is available at http://pobox.com/~oleg/ftp/Haskell/KMP-deptype.hs.

```
newtype Else w = Else w   -- Just a syntactic sugar
unP1 (BIndexP1 i) = i     -- forget the branding

index_m1 :: IntShift r
index_m1 = IntShift (-1)

index_p1 :: BIndexP1 r
index_p1 = BIndexP1 1

intshift_succ :: IntShift r -> BIndex r
intshift_succ (IntShift i) = BIndex (succ i)

index_succ :: BIndex r -> BIndexP1 r
index_succ (BIndex i) = BIndexP1 (succ i)
```

It is straightforward to verify the safety propositions associated with the above terms. For example, `-1` indeed satisfies $i : \mathit{int} \| 0 <= i + 1 < \mathit{plen}$ for any positive `plen` represented by `r`, and so `IntShift (-1)` is justified.

One of the interesting operations is the comparison of indices, e.g., the comparison of two `BIndexP1`. If the first is less than the second, we invoke the `onless` continuation, passing the first `BIndexP1` converted to `BIndex`. The safety proposition takes the form

$$0 < i \le \mathit{plen} \quad \wedge \quad 0 < j \le \mathit{plen} \quad \wedge \quad i < j \quad \rightarrow \quad 0 \le i < \mathit{plen}$$

whose conclusion justifies the use of `BIndex` in the result:

```
index_p_lt :: BIndexP1 r -> BIndexP1 r ->
```

```
              Else w -> (BIndex r -> w) -> w
index_p_lt (BIndexP1 i) (BIndexP1 j) (Else onother) onless =
    if i < j then onless (BIndex i) else onother
```

Another interesting operation is decrementing the index. If the index is already zero, we invoke the onzero continuation. The safety proposition is

$$0 \leq i < plen \quad \wedge \quad i \neq 0 \quad \rightarrow \quad 0 \leq i - 1 < plen$$

whose conclusion again justifies the use of BIndex in the result:

```
index_pred :: BIndex r -> Else w -> (BIndex r -> w) -> w
index_pred (BIndex i) (Else onzero) onfurther =
    if i == 0 then onzero else onfurther (BIndex (pred i))
```

The Haskell KMP code also uses the type proxy Nat for a non-negative integer. We elide the corresponding operations.

We should stress again the opportunity of making the syntax better by using overloaded functions and operators. The fact all these branded values have distinct types facilitates such overloading.