Interfacing to foreign language data from the Nitro functional programming language

Allan Clark

Laboratory for Foundations of Computer Science, University of Edinburgh

Abstract

This paper introduces the Nitro foreign data interface which allows the interaction with foreign data from the Nitro programming language. This allows the programmer to describe the expected format in memory of values produced by a another programming language. Furthermore we can create values in that format to pass into routines provided by the foreign language. This means we can provide services for, or use libraries written in, other programming languages without the need to translate between two different value representations.

1 INTRODUCTION

Nitro is a functional language designed for use in low-level programming. This paper discusses the Nitro foreign data interface, which provides some facilities to interface with data from other languages. Both higher-level and lower-level languages are considered.

In general a high-level language does not allow the user to directly inspect the representation of data in the user's program. The user must use the primitives provided by the language to create, inspect and manipulate data. This allows an implementation of the language to ensure that all of the values in a program conform to some internal representation. This in turn allows the implementation to provide other runtime services such as garbage collection. The programmer cannot directly use data from another language, because it will not conform to the internal representation. The garbage collector and other runtime services, are usually written in another language, because it manipulates that internal representation which we are not allowed to inspect.

A marshalling interface is a common way for a high-level language to provide access to foreign data. Because the high-level language only permits operations on data that corresponds to its representation of values, the language must provide a means to marshal arbitrary values to and from that internal representation. Furthermore because the language does not provide the ability to inspect arbitrary values, the marshalling code must be written in a language which does. Usually such a high-level language will provide C macros and functions allowing the creation of values conforming to the internal representation. An interface is then built from one language to another by using C to inspect the values of the first language and create values of the second using the provided functions.

This arrangement has at least three drawbacks. Firstly our interfaces to foreign languages must be written in the C language, even when we are not interfacing with C. Though there are some attempts to generate the C stub code automatically[NJ97, NDT$^+$]. Secondly our marshalling barrier may be too expensive. An interface that requires repeated calls to and from the foreign language requires repeated marshalling to and from both language representations. In certain applications such an expense causes the higher-level language to be abandoned. Finally if several implementations of the language exist, their internal representations of values may differ, and hence interfaces must be written for each implementation - though this can be offset to a degree by specifying which C macros must be provided by an implementation.

A requirement of our foreign data interface is to avoid the use of such a marshalling interface.

An alternative arrangement is to provide our language with the facilities to inspect and manipulate foreign values. However, we require that we do this in a way that we can ensure is safe. To do this we provide the programmer with the ability to describe the representation of a foreign data type. A programmer may still write an unsafe program, if he makes a mistake when writing the interface. If however, the interface is correct, then our language can make sure that we do not perform illegal operations over the foreign data. Additionally the compiler can help detect an illogical interface, such as an ambiguous union type. An ambiguous union type is one in which a value could be considered to be of two incompatible types because the test cannot always determine between the two.

One drawback is that the values that we are now manipulating cannot be ensured to be of some common internal representation. This means that we may have to give up on such internal representations and runtime services which rely upon them. Alternatively we must find a way to separate out such values and consider them separately, these issues are not considered within the current paper.

In the first part of this paper the typing of arrays is introduced. We show that we can allow the access to foreign arrays which may not conform to any special internal representation. We must show that we can access such foreign arrays safely, that is we do not make an unchecked out of bounds access.

In the second part of this paper the mechanisms introduced to the Nitro programming language to allow the manipulation of other forms of external data are discussed. We will build up a printer for O'caml[LVD$^+$] values as we go. This is a informative example because it exercises much of the facilities provided by the foreign data interface including our foreign array typing.

All of the examples are in the Nitro programming language, however anyone familiar with the syntax of ML language and O'caml in particular will find the syntax familiar.

2 ARRAY TYPES

Arrays are a crucial part of the foreign data interface. If one is to access foreign values, one must be able to access an array of foreign values. A type safe language however must ensure that an access to an array is made within the bounds of the array.

In a high-level language this is commonly achieved by storing the length of the array together with the array, any access is first checked at runtime, if the index is out with the bounds of the array usually some exception is raised.

This however is not an option for external arrays, since while the length may be stored with the array, it may not be in the format expected, certainly there exist several language implementations that store the length of the array in different ways, hence we could not hope to access all of these array kinds with that technique.

A better way is to allow the programmer to describe to the compiler how the length is stored.

We describe the type of bare arrays and their index types. We give label variables to both bare array types and to integer types. Label variables act like existential type variables, they cannot match any other label variables but themselves. There is a special label **int** .(_) which is used for general integer types that we may want to unify together but are not equal. An integer type is written **int** .(v) where v is the label variable. An array type is written t **array**.(v) where v is the label variable and t is the type of the elements.

To make a bare array access we must provide the variable to be used as the array bounds check. The syntax for a bare array access is then e_1 .(len).[e_2], where e_1 is the expression that evaluates to the bare array, len is the variable used as a check for the array bounds access, and e_2 is the index expression. For such an expression to be valid, e_1 must have type 'a **array**.(v) and len must have type **int** .(v). Neither may have the special label .(_).

Given this typing of bare arrays, it is possible to define a type of native array that stores the length variable together with the array. Furthermore because the label variables are existential such an array representation can be seen as an abstract type in the style of [LO94]. We can provide an array module with the common array operations such as iter and copy. One can then use these arrays as conveniently as arrays are used in a traditional high level language, without the need to provide a check variable at each index operation.

For the purposes of this paper though we need only the ability to describe the representation of external arrays in order to access them.

Suppose we wish to access an array representation of some foreign language M. M arrays consists of a tuple of 3 elements, the first is the length, the second is some information used by the garbage collector and the third is the actual array itself. Here we assume all the elements are of type boolean. We define the type of M arrays and a function for indexing them. In this example we give the type constraint for the get_marray function, this is not really necessary, though it simplifies type inference. In [LO94] this problem is solved by requiring existential types to have

constructors, later we will see that we can also use this technique, but for now we cannot as a Constructor would invalidate the runtime representation.

```
type gc_info
type m_array = (int.(len), gc_info, bool array.(len))
let get_marray : int.(_) -> m_array -> bool
  i (l, _, a) = a.(l).[i]
```

Now suppose we would also like to access arrays from language N. These arrays are similar but contain a further tuple element before the actual array, which contains debugging information, we can define such arrays as:

```
type gc_info
type debug_info
type n_array = (int.(len), gc_info, debug_info, bool array.(len))
let get_narray : int.(_) -> n_array -> bool
  i (l, _, _, a) = a.(l).[i]
```

3 TAGGED TYPES

A common C idiom is a tagged union type. Union types are used as the type of a storage location into which values of two or more types can be stored. When we wish to use the value stored in a location of union type, we cast the storage location to the type that we expect the value stored there to be. Often we use a tag or kind field to record what type is stored within the union location. Here is an example:

```
typedef int circle_dimensions;
typedef int square_dimensions;
typedef struct rect_d
{ int width; int height; }
  rect_dimensions;

typedef enum { Circle = 1,
               Square = 2,
               Rectangle = 3
} shape_kind;

typedef struct shape_ {
  shape_kind tag;
  union { circle_dimensions circle;
          square_dimensions square;
          rect_dimensions rectangle; } dimensions;
} shape;
```

Often the programmer would then write macros for creating and accessing valid shape values in a safe and consistent manner. A high-level language will often provide a special syntax for creating such tagged types, creating such values and inspecting those values. Additionally, because it is not possible to examine the underlying representation of such a tagged value, it is not possible to subvert the type system and access a value without using the tag. Therefore all manipulations can be checked and ensured to be sensible.

Generally the lack of control over the underlying representation of tagged values is of no concern. It does not matter to us, whether the tagged value is represented with the tag first and then the value, or vice versa. Also it does not matter what numerical values are given to each of the tags. It might be defined by something like:

```
type shape = Circle of int
           | Square of int
           | Rectangle of int * int
```

However, where the type is that of a foreign language, such as the C shape type defined above, we cannot use such a type, since it is unlikely to be represented in the same way. If we could define where to place the tag and which numerical values each tag should take, then we would be able to define the C shape type, in our high-level language. We could access such a value directly from Nitro without marshalling. Also the type system of Nitro ensures that we cannot create an invalid shape value. It also ensures that we do not make a mistake when interpreting the contents of a shape value.

Nitro provides both of the facilities that are required for this, we can define tagged types with specific numerical values for each tag. We can also specify where the argument is in relation to the tag. This is also the general tactic that the foreign data interface of Nitro employs. We use tagged types where possible to describe the structure of a type's representation, since the constructors of a tagged type provide a convenient and familiar syntax to create and inspect foreign values. To do this we extend tagged union types extensively. We call types that use such extensions, a *custom tagged type*.

We now discuss most of the facilities currently provided in the Nitro foreign data interface. As a platform for the discussion we build up an interface for accessing the internal representation of O'caml[LVD$^+$] values. In the process we create a printer for such O'caml values. As each new construct is added we discuss the extra obligations to the type checker imposed by the addition.

Before we begin a brief description of the internal representation of O'caml values.

Each value is either a long value which is represented by an integer, or a compound value represented by a pointer to a record allocated within the heap. In order that the two kinds can be distinguished from each other at runtime, the least significant bit is reserved as an indicator. It is set to 1 for long values and 0 for pointers. For long values we must make sure that we shift the actual value to the left by one and set the least significant bit to one. When we want to operate on the actual value we must make sure that we reverse the shifting operations. Integers are therefore restricted to 1 bit less than the length of a word for the machine.

Heap allocated values consist of a one word header followed by a number of ocaml values. The header contains three portions. The first is the ocaml tag, this is used to determine the kind of the value. There are some predefined tags that show the value represented is a string, a function closure, a double or an array. The other

tags are used to store the tag of an O'caml tagged value. The second portion of the header, is for use by the garbage collector and records the garbage collection status of the heap object, this is known as the colour of the value, and can be one of four values. Finally the length of the value is recorded, this tells us how many ocaml values follow the header.

We begin with a discussion of two very basic custom tagged type additions and then begin our representation of the ocaml internal value in the third subsection Masks for custom tags.

3.1 Custom tags

These are very simple to use. To use a custom tag to represent an external data type, one must only be aware of exactly how a value created with such a custom constructor is represented at runtime. It is represented as a pair, the first component of which is the tag value, the second of which is the argument. This means that custom tags on their own have limited use, however they form the basis for most of the other constructs.

Custom tags, are also very simple to type. We need only ensure that two tags do not have the same tag value. In fact we can be a little more liberal than this, we can say that two tags may in fact have the same tag value provided that the argument is the same. This can be useful, for example we may want a type of colour. This can contain the tags gray as well as grey, which we actually want to have equal values. Though this would perhaps be better done with a tag aliasing construct, we will see that in general allowing tags to overlap can be of use.

The shape example given above is a sufficient example for custom tags.

3.2 The immediate keyword

Where the tagged union type, is really an enumeration type, a C representation will not create a heap object for it. Instead it will be represented as a word value. The **immediate** keyword allows the user to define such a tagged value in Nitro. We can still give a custom value for the actual tag value.

Here we define the days of the week type, that could be used to represent the tm_wday field of a tm struct as defined in the time.h system header file. Notice that although the C code expects an integer our type would be more robust since we ensure that we cannot create an invalid day.

```
type immediate weekday =
  Sunday {0} | Monday {1} | Tuesday {2} | Wednesday {3}
| Thursday {4} | Friday {5} | Saturday {6}
```

Immediate tags alone require no extra typing.

3.3 Masks for custom tags

Often a tag does not take up the whole of a word value. In this case we want to test against only a portion of the word against which we are matching. A good

example of this is the O'caml representation of values. The garbage collector must be able to distinguish between long values used to represent integers, characters and other such values, and pointer values which reference the heap. The O'caml representation reserves the least significant bit of each value to record which kind a value is. Pointers are marked by a zero in the least significant bit. For long values, the least significant bit is set, which means that we have one less bit available for representing the underlying long value.

In Nitro we can specify this with a mask on a custom constructor, we also need to use the **immediate** keyword. We can now begin to build up a generic printer for O'caml runtime values. We begin by saying only what kind a value is.

```
type immediate ocaml_value =
    Ptr  { 0 with mask 1 }
  | Long { 1 with mask 1 }

let display_value p =
  match p with
    Ptr -> print_string "is pointer"
  | Long -> print_string "is long"
```

Custom tags have arguments, and below we will add arguments to immediate tags. This means that we must check that two tags do not overlap, or when they do that the types of the arguments are compatible. This is the same requirement as for custom tags without masks, only now with masks on our custom tags our tags are much more likely to overlap. This is because if two tags have different masks then they automatically overlap - that is a value could match both tags. In practice the masks for all constructors belonging to the same type are often the same as in the given example.

3.4 Immediate Arguments

The above example showed us that we could distinguish between pointers and longs. However this is of limited use if we cannot access the underlying value. In our O'caml library a pointer points to a heap_object. So far our heap objects representation is very simple, but one can think of a heap object as being some kind of record value.

Immediate arguments allow us to give the type of a value matching the given constructor. Here is a revised version of our O'caml printer, which depends upon display_hobj, our printer for heap_objects which so far can handle only doubles.

```
type immediate ocaml_value =
    Ptr  { 0 with mask 1 } of heap_object
  | Long { 1 with mask 1 } of int

let display_value p =
  match p with
    Ptr (hobj) -> display_hobj hobj
  | Long (i) -> print_int (i >> 1)
```

Notice that although the value is masked to test against a tag, the argument refers to the whole value. This is important, in the pattern Ptr (hobj), because Ptr is an immediate constructor, both the constructor and the argument pattern match against the same value. However the constructor matches against only the masked part, while the variable pattern hobj, matches against the whole value, hence the identifier is assigned to the whole value.

There is no extra typing required beyond that which we already have, for the masking of custom constructors. Recall that if two tags have incompatible argument types, then their tag values must be mutually exclusive.

Aside from typing considerations, adding arguments to immediate constructors does raise the possibility for a constructor application to fail. This is because the argument given to a constructor application will be the value used as both the argument and the tag, hence the argument itself must match the tag. The compiler inserts a check at each such constructor application, so for example Long 0 will fail at runtime. Hence a constructor application that type checks correctly may still fail at runtime.

3.5 Tag operations

In the previous sub section our representation of a long value had a slight deficiency. We had to remember to right shift the returned integer in order to get the actual integer represented. Before applying the Long constructor to an integer, we would have to remember to left shift and set the least significant bit of the integer argument or else the constructor application may fail and perhaps more to the point we would be representing the wrong value.

All of this was more or less an inconvenience, however this is because shifting and setting bits are all valid operations over integers. If the argument had been a pointer, then we could not perform such operations. For example if the O'caml bit tags were switched around, we would not be able to use the Ptr tag at all.

To overcome these problems, we provide tag operations. These are operations that are applied to the argument of a tag when the tag is matched against. The reverse of the operation is applied when the constructor is applied to an argument. Here is our new definition of the ocaml_value type and printer.

```
type immediate ocaml_value =
   Ptr  { 0 with mask 1 } of heap_object
 | Long { 1 with mask 1 } (>>. 1) of int

let display_value p =
  match p with
    Ptr (hobj) -> display_hobj hobj
  | Long (i) -> print_int i
```

The dot after the tag operation here, tells the compiler that the reverse shift operation should pad with 1s instead of zeros.

The typing of these tag operations is a little subtle. The promise of our foreign data interface so far has been that we must retain type safety in the absence of

a mistakenly written interface. What this means is that if we write our interface incorrectly, then we accept that we may access a foreign data value at the wrong type. However we should not be able to use the foreign data interface to subvert our own type system, that is we should not be able to incorrectly access a value that has been created by ourselves, even if it was created with a custom constructor.

Tag operations could exploit this possibility, because we may lose information by shifting bits out. This is why the tag operations and their reverse must be specified so precisely by the programmer. The compiler then must insert a check on each constructor application to ensure that the operation is reversible. In the above example the compiler inserts a check when the Long constructor is applied that the most significant bit is 0, otherwise we would return a different value when shifted in the other direction.

The checks inserted by the compiler for some constructors would often simply not be done by a low-level implementation, because the programmer has some reason to believe that the check would never fail. A smart compiler may be able to reduce some of the checks necessary. For example, as mentioned above, an application of the Long constructor above would first need to check that the most significant bit is 0 before performing the shift this ensures that we can retrieve the original value. After the shift we must check that the least significant bit is 1 which ensures that the argument matches the tag. It is very simple for a compiler to omit the second check because the reverse shift operation - used at constructor application - pads with 1s so after the shift the second test could never fail. In some cases a smart compiler may even be able to eliminate the first check, just by ensuring the value of the integer is low enough, certainly in the case of applying the constructor to a constant integer the check can be omitted.

3.6 Wild cards

At this point we take a brief detour from our goal of representing O'caml internal values to explain wild card constructors. Wild card constructors are used to represent the following situation. We have one or more special values of a certain type, while all other values that are not one of the special values are valid for that type, it must be first checked that they are not one of the special values.

The most familiar situation in which this arises is a *possibly null pointer*. We want to indicate that a pointer value is either the special null value, or it can be considered to be a pointer to the argument type. We write wild card tags using an underscore. Assuming we have a type for cstrings then, we could represent a possibly null cstring with the following type.

```
type immediate pn_cstring = Null {0} | Cstring {_} of cstring
```

In order to extract the best possible use out of such wild card tags we require to alter the typing of match expressions. However the typing is still straightforward. The basic requirement is that a wild card constructor may only be matched against once all other constructors from the same type have been matched against. Where

the type is also immediate, we must also check at constructor application time that the argument is not one of the other tags.

This extra check at constructor application time is often unnecessary, so to avoid this we say that the check is omitted if the argument type of the wild card constructor used, is safe to use in place of all the arguments of non-wild-card constructors. In practice where wild card constructors are used, the non-wild card constructors have no argument, as is the case above. This means that all arguments are safe to use in place of a non-argument and hence all such constructor application checks can be omitted.

3.7 Preceeds

So far we have concentrated a great deal on the custom description of a tag. The data surrounding a tag though has been left as standard. Either the argument has been assumed to follow the tag directly or, where the **immediate** keyword is used, actually be the tag.

We now consider the cases where the tag is followed by more than one argument in memory. In this case we use the **preceeds** keyword which accepts one argument. Normally the argument given will be a record or tuple type, so effectively we have described several arguments that will follow the main tag. When we match against the tag we can assign to the argument a name which can then be used to de-construct the record or tuple type and gain access to the multiple arguments.

An alternative solution to this is to allow the definition of tagged types with multiple arguments, however the **preceeds** method has the advantage that we can give to the argument type, an array type, for which we do not know, at compile time, how many elements will follow the tag. Generally contained within the tag, or surrounding the tag, is an element stating how long the array is otherwise we could not safely access the array.

We can now begin to access ocaml heap objects. Our ocaml value printer above alluded to a printer for heap objects; we can now partially define such heap objects - only those that are floating point values - and begin a definition for display_hobj.

```
type heap_object =
     Double { 253 with mask 0xFF } preceeds (int * int)

let display_hobj h =
  match h with
    Double f -> print_float f
```

We forbid a tag within an **immediate** type to contain a **preceeds** argument, because in general there is no address associated with an **immediate** value. We must also disallow the creation of a tagged type, using a constructor with a **preceeds** argument which is its own type. This is due to the difficulty of creating such a value.

This means that the following definition can be used to inspect null terminated integer arrays, but we cannot create any such values ourselves.

```
type null_term_int =
```

```
  End {0}
| One_value {_ of int} preceeds null_term_int
```

3.8 External Arrays

We now have enough machinery to revisit external arrays. Recall that bare arrays, allow us to type array operations without depending upon representing arrays with a private internal representation, we can use this to access external arrays. We will create two external array types, the first is a simple pair representation of arrays, the second is analogous to what we require for our O'caml value representation.

Our first example represents arrays as a pair, to ensure that we don't allow simply any pair consisting of an integer and a bare array but one in which the integer represents the length of the bare array, we use a custom tagged type. However there is only one constructor for the type and that has a wild card tag value. This is essentially the same as the pair representation for external arrays we had before, but now the type system can infer the external array type because we use a constructor to match against such values.

```
type pair_array =
  Parray { _ of int.(len) } of bool array.(len)
```

Because the custom type is not immediate, values of this type are automatically represented as a pair consisting of first the tag followed by the boolean array. Because both the tag value and the array length have the same integer label variable we know that the two are equal.

Our next example is a popular array representation. The array and the length are stored together in one contiguous portion of memory. The first word of which is a header value containing the length, while the remainder is the array itself. We can use the **preceeds** keyword to indicate this format.

```
type value
type safe_array =
   Sarray { _ int.(len) } preceeds of value array.(len)

let print_safearray p =
  match p with
    Sarray {length}(a) ->
      for i = 0 to length - 1 do
        print_value (a.(length).[i]
      done
```

When we match against the Sarray constructor we can set the second argument to a variable which we can then use to access the array argument because they both have the same integer label type.

The header word commonly contains more than just the length. The ocaml representation for which we are building a generic printer, is one such case, in the next subsection we see how we extract multiple data from single tags and complete our ocaml value printer.

3.9 Multiple arguments packed in a tag

Often the number of different tags is rather small compared with the amount possible from the word size of the machine. In addition the arguments may not require a whole word. When this situation arises it is tempting to store the tag of the value and its argument in the same word. We actually had something similar to this when we used the least significant bit of an ocaml_value to flag whether the value was a pointer or not. To store multiple tags is always possible by using intermediate types and shifting. Sometimes though this is inconvenient when we want to access only one of the values stored within a value without going through several layers of constructors.

This is the situation for the heap objects in the ocaml value representation. Recall that the header contains three pieces of information; the tag describing the kind of heap object, the garbage collector colour, and the length of the heap object.

To represent a heap object in Nitro, we use multiple arguments within a tag, since this allows us to access these 3 pieces of information in any order. Here is the definition:

```
type gc_colour =
   Caml_white {0x000}
 | Caml_grey  {0x100}
 | Caml_blue  {0x200}
 | Caml_black {0x300}

type heap_object =
    Closure {247 with mask 0xFF;
             _   with mask 0x300 of gc_colour;
             _   (>> 10) of int;}

  | Double { 253 with mask 0xFF;
             _   with mask 0x300 of gc_colour;
           } preceeds (int * int)

  | Tag { _ with mask 0xFF of int;
          _ with mask 0x300 of gc_colour;
          _ (>>10) of int.(len) } preceeds ocaml_value array.(len)

and immediate ocaml_value =
    Ptr { 0 with mask 1} of heap_object
  | Long { 1 with mask 1} (>>. 1) of int


let rec display_hobj obj =
  match obj with
    Closure {_; gc; len} -> print_string "<fun>"
  | Double {_; gc;}(f) -> print_double f
  | Tag {_; gc; len}(a) ->
     for i = 0 to len - 1 do
       display_value a.(len).[i]
     done
and display_value p = (* as before *)
```

A valid type definition containing multiple arguments within a tag requires careful typing but the extra checks are straightforward. We must ensure that the ”extra” arguments within a tag can not be seen at an incorrect type. There are several ways in which to express this constraint, for clarity we insist that the first argument of each constructor must be able to distinguish between values created with each constructor. As before we can use wild card patterns, but the same restrictions apply as before, so that match expressions must first check for all other constructors. In the example above we must check for the Closure and Double tags before the generic Tag constructor.

This completes our simple printer for O'caml values. Some of the details have been left out such as indenting purely for space reasons. The important part is that we can safely access values of a foreign language directly.

4 RELATED WORK

4.1 No Longer Foreign Function Interface

Nlffi [Blu01] is an encoding of (nearly) the whole of the C type system into the type system of ML. There is a program generator which will generate some stub code required, but in general, as in this work, C values are accessed directly.

The main difference between this work and the Nlffi is that with the Nlffi one attempts to represent the C representation of a foreign data format. The Nitro foreign data interface attempts to allow one to describe the underlying format of the data values, regardless of how one might have implemented this in C. Even if the foreign language is indeed C.

The most important advantage of the Nitro foreign data interface is that we retain type safety. There is nothing in the C type system that prevents one from picking any field from a union type, regardless of what the actual value stored there is. Another example is possibly null pointers, which can be represented as described above. Once this is done we cannot incorrectly attempt to dereference a null pointer. In contrast with the Nlffi, one checks for a null pointer with **if** C.Ptr.isNull l **then** ... **else** ... . However there is nothing to prevent you dereferencing such a pointer in either branch of the conditional, or even ensuring you have performed the check. To retain type safety in this case the stub code implementing the dereferencing could apply a check, but then we would have performed the check twice.

There are several advantages to using an interface in the style of Nlffi. These include; it can be used with an existing language there is no need to add constructs to your language to deal with foreign data. There are some cases where the Nlffi can be more efficient, these correspond exactly to those cases where C can be more efficient and in general correspond to the places where there is a possible safety hole. There is some scope for the nlffi to be more general, again this comes from those cases where one can use runtime information to avoid some checks that a static checker must include.

4.2 Checking type safety of foreign function calls

Another approach, taken in [FF05], is to do multiple language type checking of the foreign function calls. The authors' goal is to check the safety of uses of an existing foreign function interface, in this case that of O'caml. This means that we still require stub code and there is still a marshalling barrier. However there is now greater confidence that the marshalling interface is indeed written correctly. The foreign data interface described in this work cannot achieve this, essentially the programmer is on their own when actually writing the interface and mistakes can still be made. A combination of the two approaches seems like a promising direction for future study.

5 CONCLUSIONS

This paper has described an alternative way for a type safe language to provide access to foreign data. We have discussed how to access foreign arrays which may store the length of the array in various ways. For general foreign value representations, tagged types are extended to allow more general description of the format of a value of such a tagged type. This gives us the ability to describe the format of runtime type information that allows the inspection of values at the correct type.

The three major advantages of this approach are:

- Foreign data is accessed directly with no marshalling routines required.
- We retain type safety; meaning both that provided we have written our interface correctly, we cannot inspect foreign data values at an incorrect type and also that we cannot use the foreign data interface to subvert our own type system.
- The programmer can write the interface in the host language.

Because we retain type safety there is the opportunity for one to write data structures that are entirely for use within the host program, but take advantage of the control over data format to optimise the representation. While this is possible in a foreign data interface that does not retain type safety it diminishes the advantages of working within a type safe language.

There are some disadvantages to this style of foreign data interface. Firstly because of the requirement to retain type safety the compiler must insert the occasional check that is not strictly necessary. This was mentioned before in the context of our Long constructor. There are other examples, the Ptr constructor must make sure that the argument's least significant bit is zero, however this should always be the case for a valid word aligned pointer and so the check is not necessary. As was mentioned above some of these checks can be removed by a smart compiler. For checks that cannot be removed one might argue that C code without the corresponding check is more difficult to maintain as one must be sure that changes do not effect the assumed guarantees. Also because our interfaces rarely

correspond directly with that of, say the C representation, automatic interface generation would probably produce at best mixed results. Once again one can argue that this is because the foreign data interface provides more expressive power than that of an unsafe language. Perhaps the most prevalent disadvantage is the fact that our host language, must give up an internal representation for its own values, or find someway to separate internal values from foreign data.

The ideas put forward in this paper have been implemented in a compiler for the Nitro programming language. This has then been used to program several examples. The most important examples include the library for interfacing with the O'caml internal representation of values, some of which was described above, and a library for interfacing with the ncurses[Dic] C library for building command line user interfaces in which a small text editor has been built. So far the expressive power of the foreign data interface has been sufficient and no extra C glue code has been required.

6 FUTURE WORK

The array type contains a representation of the length of the array. It is assumed that every value within the array is initialised. This means that we require an array initialisation primitive. The crucial failing is that the length of the initialised portion of the array is assumed to be the same size as the space allocated for the array. In attempting to re-write some C code in Nitro I have found occasion where this is a limitation. In everyday coding it can lead to an inefficient representation because all values of an array are initialised only to be then re-assigned to some other value. Current work then is the removal of this limitation, the array type containing both a length and an initialised part.

Once this has been done the author is a little dissatisfied with the current syntax of array types. Adding an initialisation part gives us three pieces of information, two of which are commonly equal. It is a wish then to investigate a better syntax.

REFERENCES

[Blu01] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[Dic] TE Dickey. ncurses text user interface library available at http://dickey.his.com/ncurses/ncurses.html.

[FF05] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 62–72, New York, NY, USA, 2005. ACM Press.

[LO94] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types, 1994. Transactions on Programming Languages and Systems.

[LVD+] X. Leroy, J. Vouillon, D. Doligez, et al. `http://caml.inria.fr`. The Objective Caml system. Software and documentation available on the Web.

[NDT+] Thien-Thi Nguyen, Loic Dachary, Oleg Tolmatcev, et al. `http://www.swig.org`. The Swig interface compiler, Software and documentation available on the Web.

[NJ97] T Nordin and SL Peyton Jones. Green card: a foreign-language interface for haskell. In *Proceedings of the Haskell Workshop*, June 1997.