Checking Type Safety of Foreign Function Calls *

Michael Furr
University of Maryland, College Park
furr@cs.umd.edu

Jeffrey S. Foster
University of Maryland, College Park
jfoster@cs.umd.edu

Abstract

We present a multi-lingual type inference system for checking type safety across a foreign function interface. The goal of our system is to prevent foreign function calls from introducing type and memory safety violations into an otherwise safe language. Our system targets OCaml's FFI to C, which is relatively lightweight and illustrates some interesting challenges in multi-lingual type inference. The type language in our system embeds OCaml types in C types and vice-versa, which allows us to track type information accurately even through the foreign language, where the original types are lost. Our system uses *representational* types that can model multiple OCaml types, because C programs can observe that many OCaml types have the same physical representation. Furthermore, because C has a low-level view of OCaml data, our inference system includes a dataflow analysis to track memory offsets and tag information. Finally, our type system includes garbage collection information to ensure that pointers from the FFI to the OCaml heap are tracked properly. We have implemented our inference system and applied it to a small set of benchmarks. Our results show that programmers do misuse these interfaces, and our implementation has found several bugs and questionable coding practices in our benchmarks.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Foreign Function Interfaces; D.2.4 [*Software Engineering*]: Software/Program Verification—Validation; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; D.2.12 [*Software Engineering*]: Interoperability

General Terms Languages, Verification

Keywords foreign function interface, FFI, foreign function calls, representational type, multi-lingual type system, multi-lingual type inference, flow-sensitive type system, dataflow analysis, OCaml

1. Introduction

Many programming languages contain a *foreign function interface* (FFI) that allows programs to invoke functions written in other languages. Such interfaces are important for accessing system-wide libraries and legacy components, but they are difficult to use correctly, especially when there are mismatches between native and foreign type systems, data representations, and run-time environments. In all of the FFIs we are aware of, there is little or no consistency checking between foreign and native code [4, 8, 15, 16, 17]. As a consequence, adding an FFI to a safe language potentially provides a rich source of operations that can violate safety in subtle and difficult-to-find ways.

This paper presents a multi-lingual type inference system to check type and garbage collection safety across foreign function calls. As far as we are aware, our paper is the first that attempts checking richer properties on the foreign language side between two general purpose programming languages. Our system targets the OCaml [16] foreign function interface to C [1], though we believe that our ideas are adaptable to other FFIs.

OCaml is a strongly-typed, mostly-functional language that includes garbage collection. C is a type-unsafe imperative language with explicit allocation and deallocation. Clearly these two languages have significant differences, and it is these differences that make multi-lingual type inference challenging.

In the OCaml FFI, most of the work is done in C "glue" code, which uses various macros and functions to deconstruct OCaml data and translate it to and from C representations. It is easy to make mistakes in this code, which is fairly low-level, because there is no checking that OCaml data is used at the right type. For example, one could treat unboxed (integer) types as pointers or vice-versa. Our type inference system prevents these kinds of errors using an extended, multi-lingual type language that embeds OCaml types in C types and vice-versa.

In the OCaml FFI, C programs can observe that many OCaml types have the same physical representation. For example, the value of OCaml type `unit` has the same representation as the OCaml integer 0, nullary OCaml data constructors are represented using integers, and OCaml records and tuples can be silently injected into sum types if they have the right dynamic tag. Thus to model OCaml data from the C perspective, we introduce *representational* types that can model any or all of these possibilities (Section 2). Conflating types in the foreign language is a common design. For example, the Java Native Interface [17] distinguishes a few primitive Java types in C, but the rest are represented with a single structure type.

One key feature of our type inference system is that it smoothly integrates flow-insensitive unification of type structure with precise flow-sensitive analysis of local information (Section 3). To understand why we need both, observe that OCaml data types are, by definition, flow-insensitive. For example, it would not make sense for a C FFI function to change the type of data in the OCaml heap, since that would likely lead to incorrect behavior. Thus our system uses unification to infer the structure of the OCaml types expected by C code. However, to access OCaml data, C programs perform dynamic tag tests and compute offsets into the middle of OCaml

* This research was supported in part by NSF CCF-0346982 and CCF-0430118.

PLDI'05, June 12–15, 2005, Chicago, Illinois, USA.

records and tuples. In order to model these operations, we use an iterative flow-sensitive dataflow analysis to track offset and tag information precisely within a function body. Our dataflow analysis is fairly simple, which turns out to be sufficient in practice because most programs use the FFI in a simple way, in part to avoid making mistakes. In our system, the results of dataflow analysis (e.g., where a pointer points in a structured block) inform unification (e.g., what the type of that element in the block is). We have proven that a restricted version of our type system is sound (Section 4), modulo certain features of C such as out-of-bounds array accesses or type casting.

Finally, recall that OCaml is a garbage-collected language. To avoid memory corruption problems, before a C program calls OCaml (which might invoke the garbage collector), it must notify the OCaml runtime system of any pointers it has to the OCaml heap. This is easy to forget to do, especially when the OCaml runtime is called indirectly. Our type system includes *effects* to track functions that may invoke the OCaml garbage collector and ensure that pointers to the OCaml heap are registered as necessary.

Most programs include "helper" C functions that take OCaml values, but are not called directly from OCaml. Thus we construct an inference system that can infer types for functions that have no declared OCaml types and can also check functions with annotated types from the FFI. To test our ideas, we have implemented our inference system and applied it to a small set of 11 benchmarks. Our implementation takes as input a program written in C and OCaml and performs type inference over both languages to compute multi-lingual types.

In our experiments we have found 24 outright bugs in FFI code, as well as 22 examples of questionable coding practice. Our results suggest that multi-lingual type inference is a beneficial addition to an FFI system.

In summary, the contributions of this work are as follows:

- We develop a multi-lingual type inference system for a foreign function interface. Our system mutually embeds the type system of each language within the other. Using this information, we are able to track type information across foreign function calls.
- Our type system uses representational types to model the multiple physical representations of the same type. In order to be precise enough in practice, our analysis tracks offset and tag information flow-sensitively, and it uses effects to ensure that garbage collector invariants are obeyed in the foreign language. We have proven that a restricted version of our system is sound.
- We describe an implementation of our system for the OCaml to C foreign function interface. In our experiments, we found a number of bugs and questionable practices in a small benchmark suite.

2. Multi-Lingual Types

We begin by describing OCaml's foreign function interface to C and developing a grammar for multi-lingual types.

In a typical use of the OCaml FFI, an OCaml program invokes a C routine, which in turn invokes a system or user library routine. The C routine contains "glue" code to manipulate structured OCaml types and translate between the different data representations of the two languages.

Figure 1 shows the source language types used in our system. OCaml (Figure 1a) includes unit and int types, product types (records or tuples), and sum types. Sums are composed of type constructors S, which may optionally take an argument. OCaml also includes types for updatable references and functions. Other OCaml types are not supported by our system; see Section 5.1 for a discussion. C (Figure 1b) includes types `void`, `int`, and the type

$$
\begin{array}{rcl}
mltype & ::= & \mathsf{unit} \mid \mathsf{int} \mid mltype \times mltype \\
& \mid & S + \cdots + S \mid mltype\ \mathsf{ref} \\
& \mid & mltype \rightarrow mltype \\
S & ::= & Constr \mid Constr\ \mathrm{of}\ mltype
\end{array}
$$

(a) OCaml Type Grammar

$$
\begin{array}{rcl}
ctype & :: & \texttt{void} \mid \texttt{int} \mid \texttt{value} \mid ctype\ \texttt{*} \\
& \mid & ctype \times \ldots \times ctype \rightarrow ctype
\end{array}
$$

(b) C Type Grammar

Figure 1. Source Type Languages

`value`, to which all OCaml data is assigned (see below). C also includes pointer types, constructed with postfix `*`, and functions.

To invoke a C function called *c_name*, an OCaml program must contain a declaration of the form

$$\mathsf{external}\ \ f : mltype = \text{"}\textit{c\_name}\text{"}$$

where $mltype$ is an OCaml function type. Calling f will invoke the C function declared as

```
value c_name(value arg1, ..., value argn);
```

As this example shows, all OCaml data is given the single type `value` in C. However, different OCaml types have various physical representations that must be treated differently, and there is no protection in C from mistakenly using OCaml data at the wrong type. As a motivating example, consider the following OCaml sum type declaration:

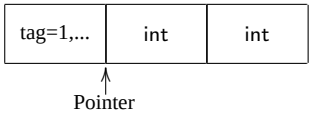
```
type t = W of int | X | Y of int * int | Z
```

This type has nullary (no-argument) constructors `X` and `Z` and non-nullary constructors `W` and `Y`.

Each nullary constructor in a sum type is numbered from 0 and is represented in memory directly as that integer. Thus to C functions, nullary constructors look just like OCaml ints, e.g., `X` and 0:int are identical. Additionally, the value of type unit is also represented by the OCaml integer 0.

The low-order bit of such *unboxed* values is always set to 1 to distinguish them from pointers. C routines use the macro `Val_int` to convert to such tagged integers and `Int_val` to convert back. There are no checks, however, to ensure that these macros are used correctly or even at all. In particular, in the standard OCaml distribution the type `value` is a typedef (alias) of `long`. Thus one could mistakenly apply `Int_val` to a boxed `value` (see below), or apply `Val_int` to a `value`. In fact, we found several examples of these sorts of mistakes in our benchmarks (see Section 5.2).

Each non-nullary constructor in a sum type is also numbered separately from 0. These constructors are represented as *boxed* values or pointers to *structured blocks* on the heap. A structured block is an array of `value`s preceded by a header that contains, among other things, a *tag* with the constructor number. For example, the constructor `Y` of our example type `t` is represented as



Products that are not part of a sum are represented as structured blocks with tag 0.

Boxed values are manipulated using the macro `Field(`x`,`i`)`, which expands to `*((value*)`x`+`i`)`, i.e., it accesses the ith element in the structured block pointed to by x. There are no checks to prevent a programmer from applying `Field` to an unboxed `value` or from accessing past the end of a structured block.

```
1  if(Is_long(x)) {
2      switch(Int_val(x)) {
3          case 0: /* X */ break;
4          case 1: /* Z */ break;
5  } } else {
6      switch(Tag_val(x)) {
7          case 0: /* W */ break;
8          case 1: /* Y */ break;
9  } }
```

Figure 2. Code to Examine a `value` of Type `t`

$$
\begin{array}{rcl}
ct & ::= & \texttt{void} \mid \texttt{int} \mid mt\ \texttt{value} \mid ct\ \texttt{*} \\
 & \mid & ct \times \cdots \times ct \rightarrow_{GC} ct \\
GC & ::= & \gamma \mid \mathsf{gc} \mid \mathsf{nogc} \\
 & & \\
mt & ::= & \alpha \mid mt \rightarrow mt \mid ct\ \mathsf{custom} \mid (\Psi, \Sigma) \\
\Psi & ::= & \psi \mid n \mid \top \\
\Sigma & ::= & \sigma \mid \emptyset \mid \Pi + \Sigma \\
\Pi & ::= & \pi \mid \emptyset \mid mt \times \Pi
\end{array}
$$

Figure 3. Multi-Lingual Type Language

Clearly a value of type `t` may have many different representations, depending on its constructor. OCaml provides a series of macros for testing tags and for determining the boxedness of a `value`. For example, code to examine a value of type `t` is shown in Figure 2. Here, `Is_long()` on line 1 checks whether a value is a pointer (by examining the low-order bit). If it is unboxed, `Int_val()` on line 2 is used to extract the tag, otherwise `Tag_val()` is used on line 6 where `x` is known to be boxed.

In addition to using OCaml data at the correct type, C FFI functions that call the OCaml runtime must notify the garbage collector of any C pointers to the OCaml heap. To do so, C functions use macros `CAMLparam` and `CAMLlocal` to register parameters and locals, respectively. If a function registers any such pointers, it must call `CAMLreturn` upon exiting to release the pointers. We have found in our experiments that it is easy to forget to use these macros, especially when functions only indirectly call the OCaml runtime (Section 5.2).

All of the macros described above are left unchecked in part because the correct OCaml types are not available in the C code. Thus, our goal is to accept the kind of code presented in Figure 2 and infer the possible OCaml types for `x`. Since a single C value could represent several OCaml types, a more expressive type system is required than that of either C or OCaml. Furthermore, we wish to only accept C code that does not violate OCaml's garbage collector invariants. In order to achieve these goals, we have developed a combined, multi-lingual type language, shown in Figure 3, that integrates and generalizes the types in Figure 1.

Our grammar for C types ct embeds extended OCaml types mt in the type `value`, so that we can track OCaml type information through C. Additionally, we augment function types with an effect GC, discussed below. Our grammar for OCaml types mt includes type variables α[1] as well as function types and custom types (see below).

All of the other OCaml types from Figure 1a—unit, int, products, sums, and references—are modeled with a *representational type* (Ψ, Σ). In this type, Ψ bounds the unboxed values of the type. For a sum type, Ψ is an exact value n counting the number of nullary constructors of the type. Integers have the same physical representation as nullary constructors but could have any value, so for this case Ψ is $\top$. Ψ may also be a variable ψ. The Σ component of a representational type describes its possible boxed values, if any. Σ is a sequence of products Π, one for each non-nullary constructor of the type. The position of each Π in the sequence corresponds to the constructor tag number, and each Π itself contains the types of the elements of the structured block. For example, the OCaml type `t` presented above has representational type $(2, (\top, \emptyset) + (\top, \emptyset) \times (\top, \emptyset)))$. Here, $\Psi = 2$ since `t` has two nullary constructors (`X` and `Z`). Also, Σ contains two product types, the integer type $(\top, \emptyset)$ for `W`, and the integer pair type $(\top, \emptyset) \times (\top, \emptyset)$ for `Y`.

Notice in Figure 2 that our C code to examine a value of type `t` does not by itself fully specify the type of `x`. For example, the type could have another nullary constructor or non-nullary constructor that is not checked for. Thus our grammars for Σ and Π include variables σ and π that range over sums and products [21], which we use to allow sum and product types to grow during inference. Only when an inferred type is unified with an OCaml type can we know its size exactly.

Our type language also annotates each function type with a garbage collection effect GC, which can either be a variable γ, gc if the function may invoke the OCaml runtime (and thus the garbage collector), or nogc if it definitely will not. GC naturally forms the two-point lattice with order nogc $\sqsubseteq$ gc Note that we reserve $\leq$ for the total ordering over the integers and use $\sqsubseteq$ for other partial orders. Our type system ensures that all necessary variables are registered before calling a function with effect gc.

Finally, sometimes it is useful to pass C data and pointers to OCaml. For example, glue code for a windowing library might return pointers representing windows or buttons to OCaml. It is up to the programmer to assign such data appropriate (distinct) opaque OCaml types, but there is no guarantee that different C types will not be conflated and perhaps misused. Thus our grammar for OCaml types mt includes types ct custom that track the C type of the embedded data. Our inference system checks that OCaml code faithfully distinguishes the C types, so that it is not possible to perform a C type cast by passing a pointer through OCaml.

3. Type System

In this section, we present our multi-lingual type inference system. Our inference system takes as input a program written in both OCaml and C and proceeds in two stages. We begin by analyzing the OCaml source code and converting the source types of FFI functions into our multi-lingual types (Section 3.1). The second stage of inference begins with a type environment containing the converted types and applies our type inference algorithm to the C source code (Section 3.2) to detect any type errors (Section 3.3).

3.1 Type Inference for OCaml Source Code

The first stage of our algorithm is to translate each external function type declared in OCaml into our multi-lingual types. We only analyze the types in the OCaml source code and not the instructions since the OCaml type checker ensures that the OCaml source code does not contain any type errors. We then combine the converted types into an initial type environment Γ_I, which is used during the second stage.

We construct Γ_I using the type translation Φ given in Figure 4, which converts OCaml function types into representational types. In this definition, we implicitly assume that $mltype_n$ is not constructed with $\rightarrow$, i.e., the arity of the function is $n - 1$. Φ is defined in terms of helper function ρ. The translation ρ gives unit and int

[1] α is a monomorphic type variable. Our system does not support polymorphic OCaml types since they seem to be uncommon in foreign functions in practice.

$$\Phi(\mathsf{external}\ mltype_1 \to \cdots \to mltype_n) = \rho(mltype_1)\ \texttt{value} \times \cdots \times \rho(mltype_{n-1})\ \texttt{value} \to_\gamma \rho(mltype_n)\ \texttt{value}$$
γ fresh

$$\begin{array}{rcl} \rho(\mathsf{unit}) & = & (1, \emptyset) \\ \rho(\mathsf{int}) & = & (\top, \emptyset) \\ \rho(mltype\ \mathsf{ref}) & = & (0, \rho(mltype)) \\ \rho(mltype_1 \to mltype_2) & = & \rho(mltype_1) \to \rho(mltype_2) \\ \rho(L_1 \mid L_2\ \mathsf{of}\ mltype) & = & (1, \rho(mltype)) \\ \rho(mltype_1 \times mltype_2) & = & (0, \rho(mltype_1) \times \rho(mltype_2)) \end{array}$$

Figure 4. Translation Rules for OCaml Types

$$\mathsf{type\ s} = \mathsf{P\ of\ int\ ref} \mid R \mid Q$$
$$\mathsf{external\ f}_{ML} : \mathsf{int} \to \mathsf{s} \to \mathsf{unit} = \text{“}f_C\text{”}$$

(a) OCaml Program

$$\Gamma_I = \{f_C : (\top, \emptyset) \times (2, (0, (\top, \emptyset))) \to_\gamma (1, \emptyset)\}$$

(b) Initial Environment

Figure 5. First Stage Example

both pure unboxed types, with no Σ component. Since unit is a singleton type, we know its value is 0, and we assign it type $(1, \emptyset)$. This is the same as the representational type for a degenerate sum type with a single nullary constructor, e.g., `type t' = A`. This is correct because that one nullary constructor has the same representation as unit. In contrast, int may represent any integer, and so it is not compatible with any sum types.

The ρ function encodes mutable references as a boxed type with a single non-nullary constructor of size 1. Regular function types are converted to mt function types. Finally, rather than give the general case for sums and products, we illustrate the translation with two sample cases. Sum types are handled by counting the nullary constructors and mapping each non-nullary constructor to a product type representing its arguments. In the definition of ρ in Figure 4, we show the translation of a sum type with one nullary constructor and one non-nullary constructor. Product types are handled by making an appropriate boxed type with no nullary constructors and a single non-nullary constructor of the appropriate size.

Figure 5a gives an example OCaml program that declares an FFI function. Figure 5b shows the initial environment Γ_I created by applying Φ to the function type in the OCaml program. The environment maps the function f_C to its representational type. The first argument of type int is represented as $(\top, \emptyset)$. The sum argument has one non-nullary constructor and two nullary constructors. The non-nullary constructor takes as its argument a reference to an integer which is converted to $(0, (\top, \emptyset))$. Therefore the type for s is $(2, (0, \emptyset))$. Finally, the return type of f_C is unit, which is represented as $(1, \emptyset)$ as in Figure 4.

3.2 C Source

After we have applied the rules in Figure 4 to the OCaml source code, we begin the second phase of our system, which infers types for C source code using the information gathered in the first phase. We present our algorithm for the C-like language shown in Figure 6, based on the intermediate representation of CIL [20], which we used to construct our implementation. In this language, expressions e are side-effect free. We include integers n, pointer dereferences $*e$, as well as the usual arithmetic operators. L-values $lval$ are the restricted subset of expressions that can appear on the left-hand side of an assignment, namely, variables x and pointer dereferences.

$$\begin{array}{rcl} e & ::= & n \mid \texttt{*}e \mid e\ aop\ e \mid lval \mid e\ \texttt{+}_p\ e \mid \texttt{(}ct\texttt{)}\ e \\ & \mid & \texttt{Val\_int}\ e \mid \texttt{Int\_val}\ e \\ lval & ::= & x \mid \texttt{*}(e +_p n) \\ aop & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{==} \mid \cdots \\ s & ::= & L\texttt{:}\ s \mid s\ \texttt{;}\ s \mid lval\ \texttt{:=}\ e \mid lval\ \texttt{:=}\ f(e, \ldots, e) \mid \texttt{goto}\ L \\ & \mid & \texttt{if}\ e\ \texttt{then}\ L \mid \texttt{if\_unboxed}(x)\ \texttt{then}\ L \\ & \mid & \texttt{if\_sum\_tag}(x) == n\ \texttt{then}\ L \\ & \mid & \texttt{if\_int\_tag}(x) == n\ \texttt{then}\ L \\ & \mid & \texttt{return}\ e \mid \texttt{CAMLreturn}(e) \\ d & ::= & ctype\ x = e \mid \texttt{CAMLprotect}(x) \\ f & ::= & \texttt{function}\ ctype\ f(ctype\ x, \ldots, ctype\ x)\ d^*\ s \\ & \mid & \texttt{function}\ ctype\ f(ctype\ x, \ldots, ctype\ x) \\ \mathcal{P} & ::= & f^* \end{array}$$

Figure 6. Simplified C Grammar

Expressions include pointer arithmetic $e_1\ \texttt{+}_p\ e_2$ for computing the address of offset e_2 from the start of the structured block pointed to by e_1. In C source code, pointer arithmetic can be distinguished from other forms using standard C type information. Our system allows `value`s to be treated directly as pointers, though in actual C source code they are first cast to `value *`. Our system includes type casts $\texttt{(}ct\texttt{)}\ e$, which casts e to type ct. Our formal system only allows certain casts to and from `value` types; other casts are modeled using heuristics in the implementation. We also include as primitives the `Val_int` and `Int_val` conversion functions. Note that we omit the address-of operation `&`. Variables whose address is taken are treated as globals by the implementation, and uses of `&` that interact with `*` are simplified away by CIL.

Statements s can be associated with a label L, and sequencing is written with `;`. We also have assignment statements $lval\ \texttt{:=}\ e$ and $lval\ \texttt{:=}\ f(e, \ldots, e)$, the latter of which stores in $lval$ the result of invoking function f with the specified arguments. A branch $\texttt{goto}\ L$ unconditionally jumps to the statement labeled L; we assume that labels are unique within a function, and jumping across function boundaries is not allowed. Conditional branches $\texttt{if}\ \ e\ \ \texttt{then}\ \ L$ jumps to the statement labeled L if the integer e evaluates to a non-zero number. Loop constructs and switch statements are omitted because CIL transforms these into if and goto statements.

We include as primitives three conditional tests for inspecting a `value` at run time. The conditional $\texttt{if\_unboxed}(x)$ checks to see whether x is not a pointer, i.e., its low-order bit is 1. The conditional $\texttt{if\_sum\_tag}(x)$ tests the runtime tag of a structured block pointed to by x. Similarly, the conditional $\texttt{if\_int\_tag}(x)$, used for nullary constructors, tests the runtime value of unboxed variable x. In actual C source code, these tests are made by applying `Tag_val` or `Int_val`, respectively, and then checking the result.

Statements also include $\texttt{return}\ \ e$, which exits the current function and returns the value of e. The special form `CAMLreturn` is used for returning from a function and releasing all variables registered with the garbage collector. This statement should be used in place of `return` if and only if local variables have been registered by `CAMLprotect`, our formalism for `CAMLlocal` and `CAMLparam`. We restrict occurrences of `CAMLprotect` to the top of a function so that the set of registered variables is constant throughout the body of a function.

Programs $\mathcal{P}$ consist of a sequence of function declarations and definitions f. We omit global variables, since our implementation forbids (via a warning message) `value`s from being stored in them

(see Section 5.1). We assume all local variables are defined at the top-level of the function.

3.3 Type Inference for C Source Code

The second phase of our type inference system takes as input C source code and the initial environment Γ_I from the first phase of the analysis (Section 3.1). Recall the example code in Figure 2 for testing the tags of a `value`. In order to analyze such a program, we need to track precise information about values of integers, offsets into structured blocks, and dynamic type tags for sum types. Thus our type system infers types of the form $ct\{B, I, T\}$, where B tracks boxedness (i.e., the result of `if_unboxed`), I tracks an offset into a structured block, and T tracks the type tag of a structured block or the value of an integer (Section 3.3.1). In our type system, B, I, and T are computed flow-sensitively, while ct is computed flow-insensitively.

Our inference algorithm computes B, I, and T using a standard fixpoint dataflow analysis, in which we iteratively apply the type rules in our formalism below. Our inference algorithm computes ct by solving flow-insensitive type constraints. Our algorithm generates four kinds of constraints: unification constraints $ct = ct'$, $mt = mt'$, $\pi = \pi'$, or $\sigma = \sigma'$; inequality constraints $T \leq \psi$ that give lower bounds on the number of primitive tags of a representational type; inequality constraints $GC \sqsubseteq GC'$ among garbage collection effects; and conditional constraints $GC \sqsubseteq GC' \Rightarrow P \subseteq P'$.

3.3.1 Flow Sensitive Types

In our system, the flow-sensitive type elements B, I, and T are given by the following grammar:

$$\begin{array}{rcl} B & ::= & \mathsf{boxed} \mid \mathsf{unboxed} \mid \top \mid \bot \\ I, T & ::= & n \mid \top \mid \bot \end{array}$$

I and T are lattices with order $\bot \sqsubseteq n \sqsubseteq \top$, and we extend arithmetic on integers to I as $\top \; aop \; I = \top$, $\bot \; aop \; I = \bot$, and similarly for T. B also forms a lattice with order $\bot \sqsubseteq \mathsf{boxed} \sqsubseteq \top$ and $\bot \sqsubseteq \mathsf{unboxed} \sqsubseteq \top$. Intuitively, $\top$ is used for an unknown type and $\bot$ is used for unreachable code, for example following an unconditional branch. We define $ct\{B, I, T\} \sqsubseteq ct'\{B', I', T'\}$ if $ct = ct'$, $B \sqsubseteq B'$, $I \sqsubseteq I'$, and $T \sqsubseteq T'$. We use $\sqcup$ to denote the least upper bound operator, and we extend $\sqcup$ to types $ct\{B, I, T\}$ similarly. Notice that B, I, and T do not appear in the grammar for ct in Figure 3, and thus our analysis does not try to track them for values stored in the heap. In our experience, this is sufficient in practice. In our type rules, we allow T to form constraints with Ψ from our representational types; the main difference between them is that Ψ may be a variable ψ that is solved for during unification, whereas T is computed flow-sensitively by iteratively applying our type rules.

The meaning of $ct\{B, I, T\}$ depends on ct. If ct is `value`, then B represents whether the data is boxed or unboxed. If B is unboxed, then T represents the value of the data (which is either an integer or nullary constructor), and I is always 0. For example, on line 3 of Figure 2, x has type $ct\{\mathsf{unboxed}, 0, 0\}$. If B is boxed, then T represents the tag of the structured block and I represents the offset into the block. For example, on line 8 of Figure 2, x has type $ct\{\mathsf{boxed}, 0, 1\}$ since it represents constructor `Y`.

Otherwise, if ct is `int`, then B is $\top$, I is 0, and T tracks the value of the integer, either $\bot$ for unreachable code, a known integer n, or an unknown value $\top$. For example, the C integer 5 has type $\texttt{int}\{\top, 0, 5\}$. Finally, for all other ct types, $B = T = \top$ and $I = 0$.

We say that a `value` is *safe* if it is either unboxed or a pointer to the first element of a structured block, and we say that any other ct that is not `value` is also safe. In our type system, data with a type where $I = 0$ is safe. Intuitively, a safe `value` can be used directly at its type, and for boxed types the header can be checked with the regular dynamic tag tests. This is not true of a `value` that points into the middle of a structured block. Our type system only allows offsets into OCaml data to be calculated locally within a function, and so we require that any data passed to another function or stored in the heap is safe. Additionally, none of our type rules allow $I = \top$, and if that occurs during inference the program will not type check.

Finally, recall that the type translation Φ converts OCaml functions to representational types (Ψ, Σ). Since all data passed from OCaml is safe, these types are converted to our full multi-lingual types by adding the flow-sensitive tags $\{\top, 0, \top\}$ to each converted type.

```
1   // x : α value{⊤, 0, ⊤}
2   if_unboxed(x) {          // α = (ψ, σ) value
3       // x : α value{unboxed, 0, ⊤}
4       if_int_tag(x) == 0   // 1 ≤ ψ
5           /* X */          // x : α value{unboxed, 0, 0}
6       if_int_tag(x) == 1   // 2 ≤ ψ
7           /* Z */          // x : α value{unboxed, 0, 1}
8   } else {
9       // x : α value{boxed, 0, ⊤}
10      if_sum_tag(x) == 0   // σ = π0 + σ'
11          /* W */          // x : α value{boxed, 0, 0}
12      if_sum_tag(x) == 1   // σ' = π1 + σ''
13          /* Y */          // x : α value{boxed, 0, 1}
14  } // x : α value{⊤, 0, ⊤}
```

Figure 7. Example with types

3.3.2 Example

To motivate our discussion of the type inference rules, we present in Figure 7 the example from Section 2 written in our grammar. To enhance readability we omit labels and jumps, and instead show control-flow with indentation. We have annotated the example with the types assigned by our inference rules. The variable x begins on line 1 with an unknown type α `value`$\{\top, 0, \top\}$. B and T are $\top$ here because the boxedness and tag of x are unknown at this program point. I is set to zero since all data passed from OCaml is safe. Upon seeing the `if_unboxed` call, α unifies with the representational type (ψ, σ). Here ψ and σ are variables to be solved for based on the constraints generated in the remaining code. On the true branch, we give x an unboxed type but still an unknown tag. Line 4 checks the unboxed constructor for x and adds the constraint that $1 \leq \psi$, which models the fact that x can only be a constructor of a sum with at least 1 nullary constructor. Thus on line 5, x is now fully known, and can safely be used as the nullary type constructor `X`. Similarly, on line 7, x is known to be the constructor `Z` and we generated the constraint $2 \leq \psi$ from the tag test on line 6.

On the false branch of the `if_unboxed` test, our type rules give x a boxed type with offset 0 (since x is safe). After testing the tag of x against 0 on line 10, we know that x has at least one non-nullary constructor, which we enforce with the constraint $\sigma = \pi_0 + \sigma'$. On line 11, then, x can be safely treated as the constructor `W` (tag 0), and if we access fields of x in this branch they will be given types according to π_0. Similarly, on line 13 we know that x has constructor `Y` (tag 1). At line 14, we join all of the branches together and lose information about the boxedness and tag of x. When we solve the unification constraints on α and inequality constraints on ψ, we will discover $\alpha = (\psi, \pi_0 + \pi_1 + \sigma'')$ with $2 \leq \psi$, which correctly unifies with our original type `t`. When this occurs, we will also discover that $\sigma'' = \emptyset$.

3.3.3 Expressions

Figure 8 gives our type rules for expressions. These rules include type environments Γ, which map variables to types $ct\{B, I, T\}$, and a *protection set* P, which contains those variables that have been registered with the garbage collector by `CAMLprotect`. Our rules for expressions prove judgments of the form $\Gamma, P \vdash e : ct\{B, I, T\}$, meaning that in type environment Γ and protection set P, the C expression e has type ct, boxedness B, offset I, and value/tag T.

We discuss the rules briefly. In all of the rules, we assume that the program is correct with respect to the standard C types, and that full C type information is available. Thus some of the rules apply to the same source construct but are distinguished by the C types of the subexpressions. We also distinguish between rules based on the flow-sensitive type of a subexpression as explained below.

The rule (INT EXP) gives an integer the appropriate type, and (VAR EXP) is standard. (VAL DEREF EXP) extracts a field from a structured block. To assign a type to the result, e must have a known tag m and offset n, and we use unification to extract the field type. Notice that the resulting B and T information is $\top$, since they are unknown, but the offset is 0, since we will get back safe OCaml data. This rule, however, cannot handle the case when records or tuples that are not part of sums are passed to functions, because their boxedness is not checked before dereferencing. We use (VAL DEREF TUPLE EXP) in this case, where B is $\top$. This rule requires that the type have one, non-nullary constructor and no nullary constructors.

The rule (C DEREF EXP) follows a C pointer. Notice that the resulting B and T are $\top$. (AOP EXP) performs the operation *aop* on T and T' in the types. (ADD VAL EXP) computes an offset into a structured block. Notice that it must be possible to safely dereference the resulting pointer as the offset cannot be larger than the width of the block. While this is not strictly necessary (we could wait until the actual dereference to enforce the size requirement), it seems like good practice not to form invalid pointers. We use (ADD VAL TUPLE EXP) for computing offsets into tuples that are not part of sums. Similar to (VAL DEREF TUPLE EXP), we allow B to be $\top$, but add the constraint that the type have one, non-nullary constructor and no nullary constructors. (ADD C EXP) performs pointer arithmetic on C types other than `value`.

(CUSTOM EXP) casts a C pointer to a `value` type, and the result is given a ct `* custom value` type with unknown boxedness and tag. (VAL CAST EXP) allows a custom type to be extracted from a `value` of a known pointer type ct `*`. Notice that this is the only rule that allows casts from `value`, which are otherwise forbidden. We omit other type casts from our formal system; they are handled with heuristics in our implementation (Section 5.1).

(VAL INT EXP) and (INT VAL EXP) translate between C and OCaml integers. When a C integer is turned into an OCaml integer with `Val_int`, we do not yet know whether the result represents an actual `int` or whether it is a nullary constructor. Thus we assign it a fresh representational type (ψ, σ), where $T + 1 \leq \psi$. This constraint models the fact that e can only be a constructor of a sum with at least T nullary constructors. Similar to (VAL DEREF TUPLE EXP), (INT VAL UNBOXED EXP) handles the case where a value is used immediately as an integer without a boxedness test.

The (APP) rule models a function call. Technically, function calls are not expressions in our grammar, but we put this rule here to make the rules for statements a bit more compact. To invoke a function, the actual types and the formal types are unified; notice that the B_i and T_i are discarded, but we require that all actual arguments are safe ($I_i = 0$). Additionally, we require that $GC' \sqsubseteq GC$, since if f might call the garbage collector, so might the current function cur_func.

$$\textsc{Int Exp}\ \frac{}{\Gamma, P \vdash n : \texttt{int}\{\top, 0, n\}} \qquad \textsc{Var Exp}\ \frac{x \in dom(\Gamma)}{\Gamma, P \vdash x : \Gamma(x)}$$

$$\textsc{Val Deref Exp}\ \frac{\Gamma, P \vdash e : mt\ \texttt{value}\{\texttt{boxed}, n, m\} \quad mt = (\psi, \pi_0 + \cdots + \pi_m + \sigma) \quad \pi_m = \alpha_0 \times \ldots \times \alpha_n \times \pi \quad \psi, \pi_i, \sigma, \alpha_i, \pi \text{ fresh}}{\Gamma, P \vdash *e : \alpha_n\ \texttt{value}\{\top, 0, \top\}}$$

$$\textsc{Val Deref Tuple Exp}\ \frac{\Gamma, P \vdash e : mt\ \texttt{value}\{\top, n, T\} \quad mt = (0, \alpha_0 \times \cdots \times \alpha_n \times \pi) \quad \alpha_i, \pi \text{ fresh}}{\Gamma, P \vdash *e : \alpha_n\ \texttt{value}\{\top, 0, \top\}}$$

$$\textsc{C Deref Exp}\ \frac{\Gamma, P \vdash e : ct\ *\{\top, 0, \top\}}{\Gamma, P \vdash *e : ct\{\top, 0, \top\}}$$

$$\textsc{AOP Exp}\ \frac{\Gamma, P \vdash e_1 : \texttt{int}\{\top, 0, T\} \quad \Gamma, P \vdash e_2 : \texttt{int}\{\top, 0, T'\}}{\Gamma, P \vdash e_1\ aop\ e_2 : \texttt{int}\{\top, 0, T\ aop\ T'\}}$$

$$\textsc{Add Val Exp}\ \frac{\Gamma, P \vdash e_1 : mt\ \texttt{value}\{\texttt{boxed}, n, n'\} \quad \Gamma, P \vdash e_2 : \texttt{int}\{\top, 0, m\} \quad mt = (\psi, \pi_0 + \cdots + \pi_{n'} + \sigma) \quad \pi_{n'} = \alpha_0 \times \cdots \times \alpha_{n+m} \times \pi \quad \psi, \pi_i, \sigma, \alpha_i, \pi \text{ fresh}}{\Gamma, P \vdash e_1 +_p e_2 : mt\ \texttt{value}\{\texttt{boxed}, n + m, n'\}}$$

$$\textsc{Add Val Tuple Exp}\ \frac{\Gamma, P \vdash e_1 : mt\ \texttt{value}\{\top, n, n'\} \quad \Gamma, P \vdash e_2 : \texttt{int}\{\top, 0, m\} \quad mt = (0, \pi_0 + \cdots + \pi_{n'} + \sigma) \quad \pi_{n'} = \alpha_0 \times \cdots \times \alpha_{n+m} \times \pi \quad \pi_i, \sigma, \alpha_i, \pi \text{ fresh}}{\Gamma, P \vdash e_1 +_p e_2 : mt\ \texttt{value}\{\texttt{boxed}, n + m, n'\}}$$

$$\textsc{Add C Exp}\ \frac{\Gamma, P \vdash e_1 : ct\ *\{\top, 0, \top\} \quad \Gamma, P \vdash e_2 : \texttt{int}\{\top, 0, T\}}{\Gamma, P \vdash e_1 +_p e_2 : ct\ *\{\top, 0, \top\}}$$

$$\textsc{Custom Exp}\ \frac{\Gamma, P \vdash e : ct\ *\{\top, 0, \top\}}{\Gamma, P \vdash \texttt{(value)}e : ct\ *\ \texttt{custom value}\{\top, 0, \top\}}$$

$$\textsc{Val Cast Exp}\ \frac{\Gamma, P \vdash e : mt\ \texttt{value}\{B, I, T\} \quad mt = ct\ *\ \texttt{custom}}{\Gamma, P \vdash \texttt{(}ct\ *\texttt{)}\ e : ct\{\top, 0, \top\}}$$

$$\textsc{Val Int Exp}\ \frac{\Gamma, P \vdash e : \texttt{int}\{\top, 0, T\} \quad T + 1 \leq \psi \quad \psi, \sigma \text{ fresh}}{\Gamma, P \vdash \texttt{Val\_int}\ e : (\psi, \sigma)\ \texttt{value}\{\texttt{unboxed}, 0, T\}}$$

$$\textsc{Int Val Exp}\ \frac{\Gamma, P \vdash e : mt\ \texttt{value}\{\texttt{unboxed}, 0, T\}}{\Gamma, P \vdash \texttt{Int\_val}\ e : \texttt{int}\{\top, 0, T\}}$$

$$\textsc{Int Val Unboxed Exp}\ \frac{\Gamma, P \vdash e : mt\ \texttt{value}\{\top, 0, T\} \quad mt = (\psi, \emptyset) \quad \psi \text{ fresh}}{\Gamma, P \vdash \texttt{Int\_val}\ e : \texttt{int}\{\top, 0, T\}}$$

$$\textsc{App}\ \frac{\Gamma, P \vdash f : ct'_1 \times \cdots \times ct'_n \rightarrow_{GC'} ct \quad \Gamma, P \vdash e_i : ct_i\{B_i, 0, T_i\} \quad ct_i = ct'_i \quad i \in 1..n \quad \Gamma, P \vdash cur\_func : \cdot \rightarrow_{GC} \cdot \quad GC' \sqsubseteq GC \quad \texttt{gc} \sqsubseteq GC \Rightarrow (ValPtrs(\Gamma) \cap live(\Gamma)) \subseteq P}{\Gamma, P \vdash f(e_1, \ldots, e_n) : ct\{\top, 0, \top\}}$$

Figure 8. Type Inference for C Expressions

The last hypothesis in this rule is a constraint that requires that if this function may call the garbage collector, every variable that points into the OCaml heap and is still live must have been registered with a call to `CAMLprotect`. Here $ValPtrs(\Gamma)$ is the set of all variables in Γ with a type (Ψ, Σ) `value` where $|\Sigma| > 0$, i.e., the set of all variables that are pointers into the OCaml heap. (These sets are computed after unification is complete.) The set $live(\Gamma)$ is all variables live at the program point corresponding to Γ. We omit the computation of $live$, since it is standard. Solving these GC constraints is discussed in Section 3.3.5.

3.3.4 Statements

Judgments for statements are flow-sensitive, which we model by allowing the type environment to vary from one statement to another, even in the same scope. Intuitively, this allows us to track dataflow facts about local variables. In order to support branches, our rules will use a *label environment* G mapping labels to type environments. In particular, $G(L)$ is the type environment at the beginning of statement L. As inference proceeds, the type rules may update G, which we write with the $:=$ operator. Our analysis iteratively applies the type rules to a function body using a worklist algorithm until G has reached a fixpoint (Section 3.3.5).

Since type environments are flow-sensitive, some of our type rules need to constrain type environments to be compatible with each other. Let $dom(\Gamma) = dom(\Gamma')$. Then we define $\Gamma \sqsubseteq \Gamma'$ if $\Gamma(x) \sqsubseteq \Gamma'(x)$ for all $x \in dom(\Gamma)$, and we define $(\Gamma \sqcup \Gamma')(x) = \Gamma(x) \sqcup \Gamma'(x)$ for all $x \in dom(\Gamma)$. Also, for the fall-through case for an unconditional branch our rules need to reset all flow-sensitive information to $\bot$. We define $reset(\Gamma)(x) = ct\{\bot, \bot, \bot\}$, where $\Gamma(x) = ct\{B, I, T\}$.

Finally, recall that only plain $ctype$s are available in the source code. Hence, analogously to Φ in Figure 4, we define a function η to translate $ctype$s to cts:

$$\begin{array}{rcll} \eta(\texttt{void}) & = & \texttt{void} & \\ \eta(\texttt{int}) & = & \texttt{int} & \\ \eta(\texttt{value}) & = & \alpha\ \texttt{value} & \alpha \text{ fresh} \\ \eta(ctype\ \texttt{*}) & = & \eta(ctype)\ \texttt{*} & \end{array}$$

We do not translate C function types because they are not first class in our language.

Figure 9 gives our type rules for statements, which prove judgments of the form $\Gamma, G, P \vdash s, \Gamma'$, meaning that in type environment Γ, label environment G, and protection set P, statement s type checks, and after statement s the new environment is Γ'.

The (SEQ STMT) rule is straightforward, and the (LBL STMT) rule constrains the type environment $G(L)$ to be compatible with the current environment Γ. The (GOTO STMT) rule updates G if necessary. If G is updated at L, we add L to our standard fixpoint worklist so that we continue iterating. (RET STMT) unifies the type of e with the return type of the current function. We also require that e is safe and that P is empty so that any variables registered with the garbage collector are released. (CAMLRET STMT) is identical to (RET STMT) except that we require P to be non-empty since it must be paired with at least one `CAMLprotect` declaration. In each of (GOTO STMT), (RET STMT), and (CAMLRET STMT), we use $reset$ to compute a new, unconstrained type environment following these statements, since they are unconditional branches.

(LSET STMT) typechecks writes to memory. We abuse notation slightly and allow e_2 on the right-hand side to be either an expression or a function call, which is checked with rule (APP) in Figure 8. Notice that since we do not model such heap writes flow-sensitively, we require that the type of e_2 is safe, and that the output type environment is the same as the input environment. In contrast, (VSET STMT) models writes to local variables, which are treated flow-sensitively. Again, we abuse notation and allow the right-hand side to be a function application checked with (APP). (VAR DECL) binds a local variable to the environment. This rule uses our mapping η to generate ct types from $ctype$s. (CAMLPROTECT DECL) takes a variable in the environment and adds it to the protection set P. Recall that this can only occur at the top-level of a function, and therefore P is constant throughout the body of a function.

The rule (IF STMT) models a branch on a C integer. (IF UNBOXED STMT) models one of our three dynamic tag tests. At label L, we know that local variable x is `unboxed`, and in the else branch (the fall-through case), we know x is `boxed`. We can only apply `if_unboxed` to expressions known to be safe. In particular, in the else branch we must know the offset of the `boxed` data is 0, to allow us to do further tag tests.

Similarly, in (IF SUM TAG STMT) we set x to have tag n at label L. Notice that this test is only valid if we already know (e.g., by calling `if_unboxed`) that x is boxed and at offset 0, since otherwise the header cannot be read. In the else branch, nothing more is known about x. In either case, we require that if this test is performed, then mt must have at least n possible tags. While omitting this last requirement would not create a runtime error, it may imply a coding error, since the program would be testing for more constructors than are defined by the type. Therefore our heuristic is to warn about this case by including that clause in our rules. In (IF INT TAG STMT), variable x is known to have value n at label L. Analogously with the previous rule, we require x to be `unboxed`, and with the constraint $n + 1 \leq \psi$ we require that x must have at least $n + 1$ nullary constructors (ψ is the count of the constructors, which are numbered from 0). Similarly to (VAL DEREF TUPLE EXP) and (INT VAL UNBOXED EXP), our implementation includes analogous variations on (IF SUM TAG STMT) and (IF INT TAG STMT) that allow $B = \top$ in exchange for stricter constraints on mt. These rules are not shown since they add no new issues.

Finally, rules (FUN DECL) and (FUN DEFN) bind function names in the environment. As with (VAR DECL), these rules use η to generate ct types from $ctype$s. Notice that in (FUN DEFN), the function type is not added to the environment; for simplicity, we assume all functions are declared before they are used. We also assume that all parameters are safe, which is enforced in (APP). The label environment G' is initialized to fresh copies of Γ_m for each label in the function body, and P is initialized to the empty set.

3.3.5 Applying the Type Inference Rules

We apply the type rules in Figures 8 and 9 to C source code beginning in type environment Γ_I from phase one. There are three components to applying the type rules. First, the rules generate equality constraints $ct = ct'$ and $mt = mt'$, which are solved with ordinary unification. When solving a constraint $(\Psi, \cdot) = (\Psi', \cdot)$, we require that Ψ and Ψ' are the same, i.e., n does not unify with $\top$. We are left with constraints of the form $T + 1 \leq \Psi$ from (VAL INT EXP) and (IF INT TAGSTMT). Recall that these ensure that nullary constructors can only be used with a sum type that is large enough. Thus in this constraint, if T is negative, we require $\Psi = \top$, since negative numbers are never constructors. After unification and fixpoint iteration (see below), we can simply walk through the list of these constraints and check whether they are satisfied.

Next, when computing $\Gamma \vdash f, \Gamma'$ for a function definition f, recall that label environment G may be updated. When this happens for $G(L)$, we add L to a worklist of statements. We iteratively re-apply the type inferences rules to statements on the worklist until we reach a fixpoint. This computation will clearly terminate because updates monotonically increase facts about B, I, and T, which are finite height lattices, and because re-applying the type inference rules produces strictly more unification constraints.

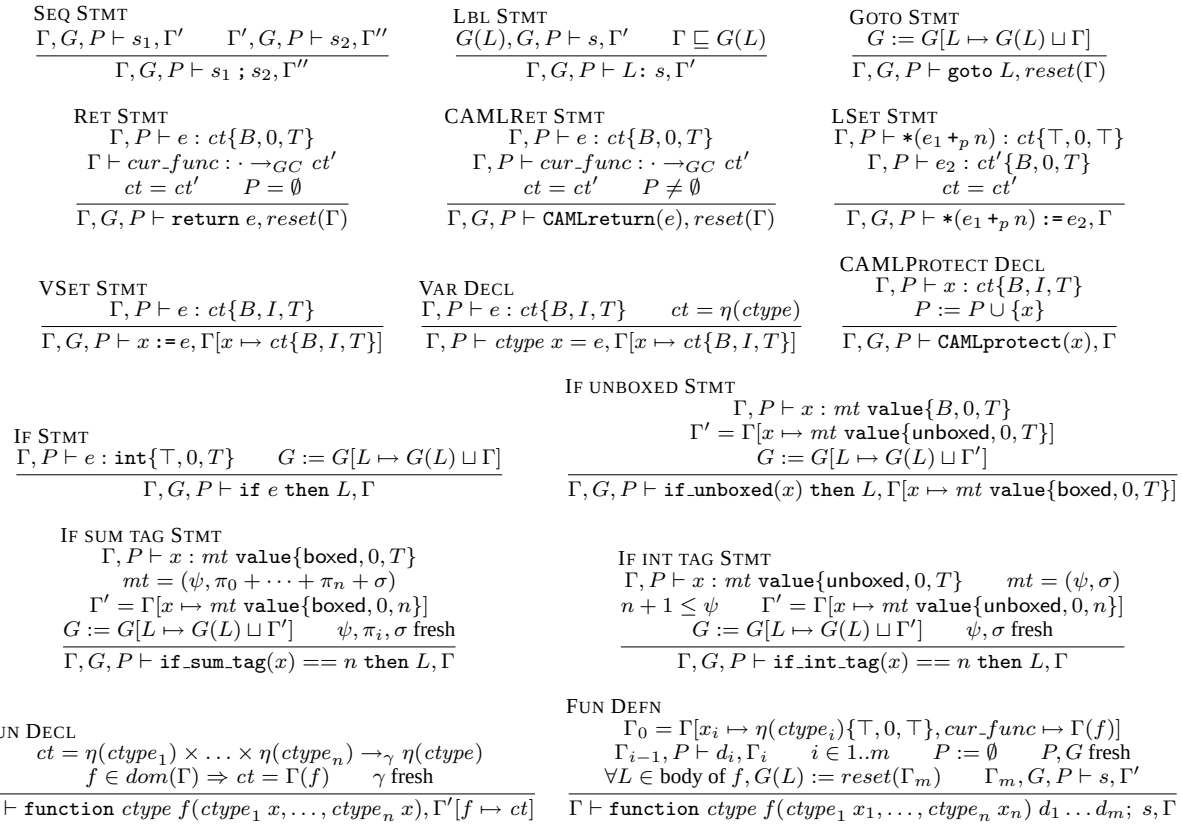


Figure 9. Type Inference for C Statements

Finally, we are left with constraints $GC \sqsubseteq GC'$. These atomic subtyping constraints can be solved via graph reachability. Intuitively, we can think of the constraint $GC \sqsubseteq GC'$ as an edge from GC to GC'. Such edges form a call graph, i.e., there is an edge from GC to GC' if the function with effect GC is called by the function with effect GC'. To determine whether a function with effect variable γ may call the garbage collector, we simply check whether there is a path from gc to γ in this graph, and using this information we ensure that any conditional constraints from (APP) are satisfied for gc functions.

4. Soundness

We now sketch a proof of soundness for a slightly simplified version of our multi-lingual type system that omits function calls, casting operations, and `CAMLprotect` and `CAMLreturn`. Full details are presented in a companion technical report [10]. We believe these features can be added without difficulty, though with more tedium. Thus our proof focuses on checking the sequence of statements that forms the body of a function, with branches but no function calls.

The first step is to extend our grammar for expressions to include C locations l, OCaml integers $\{n\}$, and OCaml locations $\{l + n\}$ (a pointer on the OCaml heap with base address l and offset n). We write $\{l + -1\}$ for the location of the type tag in the header block. We define the syntactic values v to be these three forms plus C integers n. As is standard, in our soundness proof we overload Γ so that in addition to containing types for variables, it contains types for C locations and OCaml locations. We also add the empty statement () to our grammar for statements.

Our operational semantics uses three stores to model updatable references: S_C maps C locations to values, S_{ML} maps OCaml locations to values, and V maps local variables to values. In order to model branches, we also include a statement store D, which maps labels L, to statements s. Due to lack of space, we omit our small-step operational semantics, which define a reduction relation of the form

$$\langle S_C, S_{ML}, V, s\rangle \rightarrow \langle S'_C, S'_{ML}, V', s'\rangle$$

Here, a statement s in state S_C, S_{ML}, and V, reduces to a new statement s' and yields new stores S'_C, S'_{ML}, and V'. We define $\rightarrow^*$ as the reflexive, transitive closure of $\rightarrow$.

To show soundness, we require that upon entering a function, the stores are *compatible* with the current type environment:

DEFINITION 1 (Compatibility). *Γ is said to be compatible with S_C, S_{ML}, and V (written $\Gamma \sim \langle S_C, S_{ML}, V\rangle$) if*

1. *$dom(\Gamma) = dom(S_C) \cup dom(S_{ML}) \cup dom(V)$*
2. *For all $l \in S_C$ there exists ct such that $\Gamma \vdash l : ct\ *\{\top, 0, \top\}$ and $\Gamma \vdash S_C(l) : ct\{\top, 0, \top\}$.*
3. *For all $\{l + n\} \in S_{ML}$ there exist Ψ, Σ, j, k, m, $\Pi_0, \ldots, \Pi_j$, $mt_0, \ldots, mt_k$ such that*
   - *$\Gamma \vdash \{l + n\} : (\Psi, \Sigma)\ \texttt{value}\{\texttt{boxed}, n, m\}$*
   - *$\Sigma = \Pi_0 + \cdots + \Pi_j, m \leq j$*
   - *$\Pi_m = mt_0 \times \cdots \times mt_k, n \leq k$*
   - *$\Gamma \vdash S_{ML}(\{l + n\}) : mt_n\ \texttt{value}\{\top, 0, \top\}$*
   - *$S_{ML}(\{l + -1\}) = m$*
4. *For all $x \in V$, $\Gamma \vdash V(x) : \Gamma(x)$*

DEFINITION 2. *A statement store D is said to L-compatible with a label environment G, written $D \sim_L G$, if for all $L \in D$ there exists Γ such that $G(L), G \vdash D(L), \Gamma$.*

DEFINITION 3. *D is said to be well formed if for all $L \in D$, $D(L)$ is a statement of the form $L : s$.*

The standard approach to proving soundness is to show that reduction of a well-typed term does not become *stuck*. In our system, this corresponds to showing that every statement either diverges or eventually reduces to $()$, which we prove in the technical report [10].

THEOREM 1 (Soundness). *If $\Gamma \vdash s, \Gamma'$, $\Gamma \sim \langle S_C, S_{ML}, V\rangle$, $D \sim_L G$ and D is well formed, then either $\langle S_C, S_{ML}, V, s\rangle$ diverges, or $\langle S_C, S_{ML}, V, s\rangle \rightarrow^* \langle S'_C, S'_{ML}, V', ()\rangle$.*

5. Implementation and Experiments

We have implemented the inference system described in Section 3. We first discuss the details of our implementation that are not covered by our formal system, and then present the results of analyzing a small benchmark suite with our tool.

5.1 Implementation

Our implementation consists of two separate tools, one for each language. The first tool, based on the `camlp4` preprocessor, analyzes OCaml source programs and extracts the type signatures of any foreign functions. Because ultimately C foreign functions will see the physical representations of OCaml types, the tool resolves all types to a concrete form. In particular, type aliases are replaced by their base types, and opaque types are replaced by the concrete types they hide, when available. If the concrete type is not available, the opaque type is assigned a fresh type variable, and our tool simply checks to ensure it is used consistently. As each OCaml source file is analyzed, the tool incrementally updates a central type repository with the newly extracted type information, beginning with a pre-generated repository from the standard OCaml library. Once this first phase is complete, the central repository contains the equivalent of the initial environment Γ_I, which is fed into the second tool.

The second tool, built using CIL [20], performs the bulk of the analysis. This tool takes as input the central type repository and a set of C source programs to which it applies the rules in Figures 8 and 9. The tool uses syntactic pattern matching to identify tag and boxedness tests in the code.

One feature of C that we have not fully discussed is the address-of operator. Our implementation models address-of in different ways, depending on the usage. Any local variable with an integer type (or local structure with a integer field) that has its address computed is given the type $\texttt{int}\{\top, 0, \top\}$ everywhere. This conservatively models the fact that the variable may be updated arbitrarily through other aliases. It has been our experience that variables used for indexing into `value` types rarely have their address taken, so this usually does not affect our analysis. Similarly, we produce a warning for any variable of type `value` whose address is taken (or any variable containing a field of type `value`), as well as for any global variable of type `value`. When encountering a call through a C function pointer, our tool currently issues a warning and does not generate typing constraints on the parameters or return type.

We also treat unsafe type casts specially in our implementation. Our system tries to warn programmers about casts involving `value` types, but in order to reduce false positives we use heuristics rather than be fully sound. For instance, any cast through a `void *` type is ignored, as well as any signed-unsigned type differences.

In addition to the types we have described so far, OCaml also includes objects, polymorphic variants, and universally quantified types. Our implementation treats object types in the same way as opaque types, with no subtyping between different object types. We have not seen objects used in FFI C code. Our implementation does not handle polymorphic variants, which are used in FFI code, and this leads to some false positives in our experiments.

Finally, recall that our analysis of C functions is monomorphic. Therefore, if our Φ function from Figure 4 encounters a polymorphic type variable, Φ assigns it the representational type $(\psi, \pi + \sigma)$ where ψ, π and σ are fresh variables with the constraint $1 \leq \psi$. Since a polymorphic type could be either boxed or unboxed, this prevents a C function from using the polymorphic type directly as an integer or a boxed type without at least performing a boxedness test. We also cannot infer universally quantified types for C "helper" functions that are polymorphic in OCaml `value` parameters. Instead, we allow them to be hand-annotated as polymorphic, which prevents typing constraints between any of its actual and formal arguments. Such C functions appear to be rare in practice, as we only added these annotations 4 times in our benchmark suite.

5.2 Experiments

We ran our tool on several programs that utilize the OCaml foreign function interface. The programs we looked at are actually glue libraries that provide an OCaml API for system and third-party libraries. All of the programs we analyzed were from a tested, released version, though we believe our tool is also useful during development.

Figure 10 gives a summary of our benchmarks and results. For each program, we list the lines of C and OCaml code, and the running time (three run average) for our analysis on a 2GHz Pentium IV Xeon Processor with 2GB of memory. Recall from Section 3.1 that we do not directly analyze OCaml function bodies. Thus the bulk of the time is spent analyzing C code. Also, our analysis is done as the program is compiled, so these figures also include compilation time.

The next three columns list the number of errors found, the number of warnings for questionable programming practice, and the number of false positives, i.e., warnings for code that appears to be correct. The last column shows the number of places where the implementation warned that it did not have precise flow-sensitive information (see below). The total number of warnings is the sum of these four columns.

We found a total of 24 outright errors in the benchmarks. One source of errors was forgetting to register C references to the OCaml heap before invoking the OCaml runtime. This accounts for one error in each of ftplib, lablgl, and lablgtk. Similarly, the one error in each of ocaml-mad and ocaml-vorbis was registering a local parameter with the garbage collector but then forgetting to release it, thus possibly leaking memory or causing subtle memory corruption.

The 19 remaining errors are type mismatches between the C code and the OCaml code. For instance, 5 of the lablgtk errors and all ocaml-glpk and ocaml-ssl errors were due to using `Val_int` instead of `Int_val` or vice-versa. Another error was due to one FFI function mistreating an optional argument as a regular argument. Here, the function directly accessed the option block as if it were the expected type rather than an option type containing the expected type. Thus, the C code will most likely violate type safety. The other type errors are similar.

In addition to the 24 errors, our tool reported 22 warnings corresponding to questionable coding practices. A common mistake is declaring the last parameter in an OCaml signature as type unit even though the corresponding C function omits that parameter in its declaration:

```
OCaml : external f : int → unit → unit = "f"
    C : value f(value x);
```

Program	C loc	OCaml loc	Time (s)	Errors	Warnings	False Pos	Imprecision
apm-1.00	124	156	1.3	0	0	0	0
camlzip-1.01	139	820	1.7	0	0	0	1
ocaml-mad-0.1.0	139	38	4.2	1	0	0	0
ocaml-ssl-0.1.0	187	151	1.5	4	2	0	0
ocaml-glpk-0.1.1	305	147	1.3	4	1	0	1
gz-0.5.5	572	192	2.2	0	1	0	1
ocaml-vorbis-0.1.1	1183	443	2.8	1	0	0	2
ftplib-0.12	1401	21	1.7	1	2	0	1
lablgl-1.00	1586	1357	7.5	4	5	140	20
cryptokit-1.2	2173	2315	5.4	0	0	0	1
lablgtk-2.2.0	5998	14847	61.3	9	11	74	48
Total				24	22	214	75

Figure 10. Experimental Results

While this does not usually cause problems on most systems, it is not good practice, since the trailing `unit` parameter is placed on the stack. The warnings reported for ftplib, ocaml-glpk, ocaml-ssl, lablgl, and lablgtk were all due to this case.

The warning in gz is an interesting abuse of the OCaml type system. The gz program contains an FFI function to *seek* (set the file position) on file streams, which have either type `input_channel` or `output_channel`. However, instead of taking a sum type as a parameter (to allow both kinds of arguments), the function is declared with the polymorphic type $'a$ as its parameter.

```
OCaml : external seek : int → 'a → unit = "seek"
    C : value seek(value pos, value chan){
          FILE *strm = Field(chan,0);
          fseek(strm,...);
```

Clearly using `chan` in this way is very dangerous, because OCaml will allow *any* argument to be passed to this function, including unboxed integers. In this case, however, only the right types are passed to the function, and it is encapsulated so no other code can access the function, and so we classify this as questionable programming practice rather than an error.

Our tool also reported a number of false positives, i.e., warnings for code that seems correct. One source of false positives is due to polymorphic variants, which we do not handle. The other main source of false positives is due to pointer arithmetic disguised as integer arithmetic. Recall that the type `value` is actually a typedef for long. Therefore if v is an OCaml value with type $t*$ `custom`, then both $((t*)v + 1)$ and $(t*)(v + \text{sizeof}(t*))$ are equivalent. However, our system will not type check the second case because direct arithmetic is performed on a `value` type.

Finally, in several of the benchmarks there are a number of places where our tool issued a warning because it does not have precise enough information to compute a type. For instance, this may occur when computing the type of $e_1 +_p e_2$ if e_2 has the type `int`$\{\top, 0, \top\}$, since the analysis cannot determine the new offset. We also classify warnings about global `value` types and the use of function pointers as imprecision warnings. However, these did not occur very often, only 10 and 8 times respectively. One interesting direction for future work would be eliminating these warnings and instead adding run-time checks to the C code for these cases.

6. Related Work

Most languages include a foreign function interface, typically to C, since it runs on many platforms. For languages with semantics and runtime systems that are close to C, "foreign function" calls to C can typically be made using simple interfaces. For languages that are further from C, FFIs are more complicated, and there are many interesting design points with different tradeoffs [4, 8, 15, 16, 17]. For example, Blume [4] proposes a system allowing arbitrary C data types to be accessed by ML. Fisher et al [9] have developed a framework that supports exploration of many different foreign interface policies. While various interfaces allow more or less code to be written natively (and there is a trend towards more native code rather than glue code), the problem of validating usage of the interface on the foreign language side still remains.

Recently, researchers have developed systems to check that dynamically-generated SQL queries are well-formed [6, 7, 11]. In a sense, these systems are checking a foreign-function interface between SQL and the source language. In order to model SQL queries, the systems focus on string manipulations rather than standard type structure, and so they are considerably different than our type system.

Trifonov and Shao [22] use effects to reason about the safety of interfacing multiple safe languages with different runtime resource requirements in the same address space. Their focus is on ensuring that code fragments in the various languages have access to necessary resources while preserving the languages' semantics, which differs from our goal of checking types and GC properties in FFIs.

Systems like COM [12] and SOM [13] provide interoperability between object-oriented frameworks. Essentially, they are foreign function interfaces that incorporate an object model. Typically these systems include dynamic type information that is checked at runtime and used to find methods and fields. We leave the problem of statically checking such object FFIs to future work.

Our type system bears some resemblance to systems that use physical type checking for C [5, 19], in that both need to be concerned with memory representations and offsets. However, our system is considerably simpler than full-fledged physical type checking systems simply because OCaml data given type `value` is typically only used in restricted ways.

One way to avoid foreign function interfaces completely is to compile all programs down to a common intermediate representation. For example, the Microsoft common-language runtime (CLR) [14, 18] includes a strong type system and is designed as the target of compilers for multiple different languages. While this solution avoids the kinds of programming difficulties that can arise with FFIs, it does not solve the issue of interfacing with programs in non-CLR languages or with unmanaged (unsafe) CLR code. Another approach used by SWIG [2], is to automatically generate glue code for the low level language based on an interface specification file. This has the advantage of eliminating the need for custom glue code (and thus eliminate safety violations), but it exposes all of the low level types to the high level language, creating a possibly awkward interface. All of the programs in our benchmark suite contained custom glue code written without the use of an interface

generator, suggesting that hand writing FFI code is still a popular approach.

7. Conclusion

We have presented a multi-lingual type inference system for checking type and GC safety across the OCaml-to-C foreign function interface. Our system embeds the types of each language into the other, using representational types to model the overlapping physical representations in C of different OCaml types. Our type inference algorithm uses a combination of unification to infer OCaml types and dataflow analysis to track offset and tag information. We use effects to track garbage collection information and to ensure that C pointers to the OCaml heap registered with the garbage collector. Using an implementation of our algorithm, we found several errors and questionable coding practices in a small benchmark suite. We think our results suggest that multi-lingual type inference can be an important part of foreign function interfaces, and we believe these same techniques can be extended and applied to other FFIs.

References

[1] ANSI. *Programming languages – C*, 1999. ISO/IEC 9899:1999.

[2] D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++,.

[3] N. Benton and A. Kennedy, editors. *BABEL'01: First International Workshop on Multi-Language Infrastructure and Interoperability*, volume 59 of *Electronic Notes in Theoretical Computer Science*, Firenze, Italy, Sept. 2001. `http://www.elsevier.nl/locate/entcs/volume59.html`.

[4] M. Blume. No-Longer-Foreign: Teaching an ML compiler to speak C "natively". In Benton and Kennedy [3]. `http://www.elsevier.nl/locate/entcs/volume59.html`.

[5] S. Chandra and T. W. Reps. Physical Type Checking for C. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 66–75, Toulouse, France, Sept. 1999.

[6] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In R. Cousot, editor, *Static Analysis, 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 1–18, San Diego, CA, USA, June 2003. Springer-Verlag.

[7] R. DeLine and M. Fähndrich. The Fugue Protocol Checker: Is your software Baroque? Technical Report MSR-TR-2004-07, Microsoft Research, Jan. 2004.

[8] S. Finne, D. Leijen, E. Meijer, and S. P. Jones. Calling hell from heaven and heaven from hell. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming*, pages 114–125, Paris, France, Sept. 1999.

[9] K. Fisher, R. Pucella, and J. Reppy. A framework for interoperability. In Benton and Kennedy [3]. `http://www.elsevier.nl/locate/entcs/volume59.html`.

[10] M. Furr and J. S. Foster. Checking Type Safety of Foreign Function Calls. Technical Report CS-TR-4627, University of Maryland, Computer Science Department, Nov. 2004.

[11] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 645–654, Edinburgh, Scotland, UK, May 2004.

[12] D. N. Gray, J. Hotchkiss, S. LaForge, A. Shalit, and T. Weinberg. Modern Languages and Microsoft's Component Object Model. *Communications of the ACM*, 41(5):55–65, May 1998.

[13] J. Hamilton. Interlanguage Object Sharing with SOM. In *Proceedings of the Usenix 1996 Annual Technical Conference*, San Diego, California, Jan. 1996.

[14] J. Hamilton. Language Integration in the Common Language Runtime. *ACM SIGPLAN Notices*, 38(2):19–28, Feb. 2003.

[15] L. Huelsbergen. A Portable C Interface for Standard ML of New Jersey. `http://www.smlnj.org//doc/SMLNJ-C/smlnj-c.ps`, 1996.

[16] X. Leroy. The Objective Caml system, Aug. 2004. Release 3.08, `http://caml.inria.fr/distrib/ocaml-3.08/ocaml-3.08-refman.pdf`.

[17] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.

[18] E. Meijer, N. Perry, and A. van Yzendoorn. Scripting .NET using Mondrian. In J. L. Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming, 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 150–164, Budapest, Hungary, June 2001. Springer-Verlag.

[19] G. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, Portland, Oregon, Jan. 2002.

[20] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228, Grenoble, France, Apr. 2002. Springer-Verlag.

[21] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of the 16th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 77–88, Austin, Texas, Jan. 1989.

[22] V. Trifonov and Z. Shao. Safe and Principled Language Interoperation. In D. Swierstra, editor, *8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 128–146, Amsterdam, The Netherlands, Mar. 1999. Springer-Verlag.