Checking Type Safety of Foreign Function Calls*

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

December 7, 2005

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 a *representational* type 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.

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 other 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 [3, 7, 13, 14, 15]. 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. Our system targets the OCaml [14] foreign function interface to C [1], though we believe that our ideas are adaptable to other FFIs.

In the OCaml FFI, most of the work is done in C "glue" code, which uses various macros and functions to pull apart and translate OCaml data 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. 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.

One interesting feature of the OCaml FFI is that C programs can observe that many OCaml types have the same physical representation. For example, the value of type `unit` has the same representation as the OCaml integer 0, nullary data constructors are represented using integers, and records and tuples can be 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).

Additionally, C programs can perform tag tests at runtime and compute offsets into the middle of OCaml records and tuples. Thus in addition to standard unification-style type inference, our type system includes a dataflow analysis to track offset and tag information precisely within a function body (Section 3). 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). 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

*This paper is submitted in partial satisfaction of the requirement for the degree of Master of Science in Computer Science at the University of Maryland, College Park. This research was supported in part by NSF CCF-0346982 and CCF-0430118

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

(a) OCaml Type Grammar

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

(b) C Type Grammar

Figure 1: Source Type Languages

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 GC and ensure that pointers to the OCaml heap are registered as necessary.

To test our ideas, we have implemented our inference system and applied it to a small set of benchmarks. In our experiments we have found a number of outright bugs in FFI code, as well as several 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 as are follows:

- We develop a multi-lingual type inference system for a foreign function interface that 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. 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. C (Figure 1b) includes types `void`, `int`, and the type `value`, to which all OCaml data is assigned (see below). C also includes pointer types, constructed with `*`, and functions.

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

$$\mathsf{external}\ f : \mathit{mltype} = \text{“}\mathit{c\_name}\text{”}$$

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 = A of int | B | C of int * int | D
```

```
1  if(Is_long(x)) {
2      switch(Int_val(x)) {
3          case 0:  /* B */ break;
4          case 1:  /* D */ break;
5  } } else {
6      switch(Tag_val(x)) {
7          case 0:  /* A */ break;
8          case 1:  /* C */ break;
9  } }
```

Figure 2: Code to Examine a Value of Type `t`

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

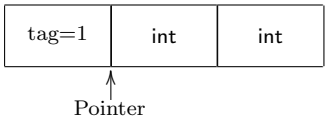
Figure 3: Multi-Lingual Type Language

This type has nullary (no-argument) constructors `B` and `D` and non-nullary constructors `A` and `C`.

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., `B` and `int` 0 are identical. Additionally, the value of type `unit` is also represented by the 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 dually apply `Val_int` to a `value`. In fact, we found several examples of this mistake 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, which contains, among other things, a *tag* with the constructor number. For example, the constructor `C` of 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 block.

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. 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.

Our goal is to accept this kind of code and infer the possible OCaml types for `x`. Figure 3 contains our combined, multi-lingual type language 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 as well as function types and custom types (see below). Note that α is a monomorphic type variable, and our system does not support polymorphic OCaml types, since they seem to be uncommon in foreign functions in practice (see Section 5.1).

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

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

Figure 4: Translation Rules for OCaml Types

All of the other OCaml types from Figure 1a—unit, integer, 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 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 the 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` has representational type $(2, (\top, \emptyset) + (\top, \emptyset) \times (\top, \emptyset)))$.

Notice in Figure 2 that our 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 simply is not checked for. Thus our grammars for Σ and Π include variables σ and π that range over sums and products [19], 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.

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. It is easy to forget to use these macros, especially when functions only indirectly call the OCaml runtime, as we have found in our experiments (Section 5.2). Thus in our type language, we annotate each function type with a garbage collection effect GC, either a variable γ, gc if the function may invoke the garbage collector, or nogc if it definitely will not. GC naturally forms the two-point lattice with order nogc $\sqsubseteq$ gc (note we reserve $\leq$ for the total ordering over the integers and use $\sqsubseteq$ for partial orders over any other lattice). 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 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 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. Restricting ourselves to the type information from OCaml is sufficient for checking that C code uses OCaml data correctly. We then combine the converted types into an initial type environment Γ_I, which feeds into the second stage.

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

Figure 5: Simplified C Grammar

We construct Γ_I using the type translation function Φ given in Figure 4. In this definition, we implicitly assume that $mltype_n$ is not constructed with $\rightarrow$, i.e., that the arity of the function whose type is being translated is n.

In Figure 4, ρ gives unit and int 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 no non-nullary constructors and exactly one nullary constructor. 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.

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 5, based on the intermediate representation of CIL [18], which we used to construct our implementation. In this language, expressions e are side-effect free and contain the usual constructs. We include pointer arithmetic e_1 `+`$_p$ e_2 for computing the address of offset e_2 from the structured block pointed to by e_1. 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 `(`ct`)` 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 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. L-values $lval$ are the restricted subset of expressions that can appear on the left-hand side of an assignment.

Statements s can be associated with a label L. We include as primitives three conditional tests for inspecting a `value` at run time. The conditional `if_sum_tag`(x) tests the runtime tag of a structured block pointed to by x. Similarly, the conditional `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. The conditional `if_unboxed`(x) checks to see whether x is not a pointer.

Statements also include the special form `CAMLreturn` for returning from a function, 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 declaring them with `CAMLprotect`, our formalism for `CAMLlocal` and `CAMLparam`.

Programs p consist of a sequence of function declarations and definitions f. We omit global variables, since our implementation forbids `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 or value of an integer. In our type system, B, I, and T are computed flow-sensitively, while ct is flow-insensitive. 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 $\bot \sqsubseteq \mathsf{unboxed} \sqsubseteq \top$. 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 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.

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. 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 would have type $ct[\mathsf{boxed}\{0\}]\{1\}$ since it represents constructor C.

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 would have 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. Intuitively, a safe `value` can be used directly at its type, and for boxed types the header can be checked with our regular dynamic 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. Notice that in our system, data with a type where $I = 0$ is safe. Additionally, none of our type rules allow $I = \top$, and if that occurs during iteration the program will not type check.

Type environments Γ map variables to types $ct[B\{I\}]\{T\}$. Judgments also include a *protection set* P, which contains those variables that have been registered with the garbage collector by `CAMLprotect`. We split the type inference rules into expressions and statements, and discuss each in turn.

3.3.1 Expressions

Figure 6 gives our type rules for expressions. This system proves judgments of the form $\Gamma, P \vdash e : ct[B\{I\}]\{T\}$, meaning that in type environment Γ, 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.

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, the sum 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. Note that since this rule generates more restrictive constraints than (Val Deref Exp), fixpoint iteration still converges. Our implementation includes similar rules for using pointer arithmetic or reading unboxed data without a boxedness test, which we omit due to lack of space.

The rule (C Deref Exp) follows a C pointer. Notice that the resulting B and T are $\top$. (AOP Exp) performs the *aop* operation on T and T' in the types. (Add Val Exp) is similar to (Val Deref Exp). Notice that it must be possible to dereference the resulting pointer. 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.

$$\textsc{Int Exp}\ \frac{}{\Gamma, P \vdash n : \texttt{int}[\top\{0\}]\{n\}}$$

$$\textsc{Var Exp}\ \frac{x \in dom(\Gamma)}{\Gamma, P \vdash x : \Gamma(x)}$$

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

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

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

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

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

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

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

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

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

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

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

Figure 6: Type Inference for C Expressions

$$\textsc{Seq Stmt}\quad \frac{\Gamma, G, P \vdash s_1, \Gamma' \qquad \Gamma', G, P \vdash s_2, \Gamma''}{\Gamma, G, P \vdash s_1 \texttt{ ; } s_2, \Gamma''}$$

$$\textsc{Lbl Stmt}\quad \frac{G(L), G, P \vdash s, \Gamma' \qquad \Gamma \sqsubseteq G(L)}{\Gamma, G, P \vdash L\texttt{:}\ s, \Gamma'}$$

$$\textsc{Goto Stmt}\quad \frac{G := G[L \mapsto G(L) \sqcup \Gamma]}{\Gamma, G, P \vdash \texttt{goto}\ L, \mathit{reset}(\Gamma)}$$

$$\textsc{Ret Stmt}\quad \frac{\Gamma, P \vdash e : ct[B\{0\}]\{T\} \quad \Gamma \vdash \mathit{cur\_func} : \cdot \to .\ ct' \quad ct = ct' \qquad P = \emptyset}{\Gamma, G, P \vdash \texttt{return}\ e, \mathit{reset}(\Gamma)}$$

$$\textsc{CAMLReturn Stmt}\quad \frac{\Gamma, P \vdash e : ct[B\{0\}]\{T\} \quad \Gamma, P \vdash \mathit{cur\_func} : \cdot \to .\ ct' \quad ct = ct' \qquad P \neq \emptyset}{\Gamma, G, P \vdash \texttt{CAMLreturn}(e), \mathit{reset}(\Gamma)}$$

$$\textsc{If Stmt}\quad \frac{\Gamma, P \vdash e : \texttt{int}[\top\{0\}]\{T\} \quad G := G[L \mapsto G(L) \sqcup \Gamma]}{\Gamma, G, P \vdash \texttt{if}\ e\ \texttt{then}\ L, \Gamma}$$

$$\textsc{LSet Stmt}\quad \frac{\Gamma, P \vdash *(e_1 +_p n) : ct[\top\{0\}]\{\top\} \quad \Gamma, P \vdash e_2 : ct'[B\{0\}]\{T\} \quad ct = ct'}{\Gamma, G, P \vdash *(e_1 +_p n) := e_2, \Gamma}$$

$$\textsc{VSet Stmt}\quad \frac{\Gamma, P \vdash e : ct[B\{I\}]\{T\}}{\Gamma, G, P \vdash x := e, \Gamma[x \mapsto ct[B\{I\}]\{T\}]}$$

$$\textsc{CAMLProtect Var}\quad \frac{\Gamma, P \vdash x : ct[B\{I\}]\{T\} \quad P := P \cup \{x\}}{\Gamma, G, P \vdash \texttt{CAMLprotect}(x), \Gamma}$$

$$\textsc{If\_unboxed Stmt}\quad \frac{\Gamma, P \vdash x : mt\ \texttt{value}[B\{0\}]\{T\} \quad \Gamma' = \Gamma[x \mapsto mt\ \texttt{value}[\texttt{unboxed}\{0\}]\{T\}] \quad G := G[L \mapsto G(L) \sqcup \Gamma']}{\Gamma, G, P \vdash \texttt{if\_unboxed}(x)\ \texttt{then}\ L, \Gamma[x \mapsto mt\ \texttt{value}[\texttt{boxed}\{0\}]\{T\}]}$$

$$\textsc{Var Decl}\quad \frac{\Gamma, P \vdash e : ct[B\{I\}]\{T\} \qquad ct = \eta(\mathit{ctype})}{\Gamma, P \vdash \mathit{ctype}\ x = e, \Gamma[x \mapsto ct[B\{I\}]\{T\}]}$$

$$\textsc{If\_int\_tag Stmt}\quad \frac{\Gamma, P \vdash x : mt\ \texttt{value}[\texttt{unboxed}\{0\}]\{T\} \qquad mt = (\psi, \sigma) \quad n + 1 \leq \psi \qquad \Gamma' = \Gamma[x \mapsto mt\ \texttt{value}[\texttt{unboxed}\{0\}]\{n\}] \quad G := G[L \mapsto G(L) \sqcup \Gamma'] \qquad \psi, \sigma\ \text{fresh}}{\Gamma, G, P \vdash \texttt{if\_int\_tag}(x) == n\ \texttt{then}\ L, \Gamma}$$

$$\textsc{Fun Decl}\quad \frac{\text{let } ct = \eta(\mathit{ctype}_1) \times \ldots \times \eta(\mathit{ctype}_n) \to \eta(\mathit{ctype}) \quad f \in \mathit{dom}(\Gamma) \Rightarrow ct = \Gamma(f)}{\Gamma \vdash \texttt{function}\ \mathit{ctype}\ f(\mathit{ctype}_1\ x, \ldots, \mathit{ctype}_n\ x), \Gamma'[f \mapsto ct]}$$

$$\textsc{If\_sum\_tag Stmt}\quad \frac{\Gamma, P \vdash x : mt\ \texttt{value}[\texttt{boxed}\{0\}]\{T\} \quad mt = (\psi, \pi_0 + \cdots + \pi_n + \sigma) \quad \Gamma' = \Gamma[x \mapsto mt\ \texttt{value}[\texttt{boxed}\{0\}]\{n\}] \quad G := G[L \mapsto G(L) \sqcup \Gamma'] \qquad \psi, \pi_i, \sigma\ \text{fresh}}{\Gamma, G, P \vdash \texttt{if\_sum\_tag}(x) == n\ \texttt{then}\ L, \Gamma}$$

$$\textsc{Fun Defn}\quad \frac{\Gamma_0 = \Gamma[x_i \mapsto \eta(\mathit{ctype}_i)[\top\{0\}]\{\top\}, \mathit{cur\_func} \mapsto \Gamma(f)] \quad \Gamma_{i-1}, P \vdash d_i, \Gamma_i \qquad i \in 1..m \qquad P\ \text{fresh} \qquad P := \emptyset \quad \forall L \in \text{body of } f, G'(L) := \mathit{reset}(\Gamma_m) \qquad \Gamma_m, G', P \vdash s, \Gamma'}{\Gamma \vdash \texttt{function}\ \mathit{ctype}\ f(\mathit{ctype}_1\ x_1, \ldots, \mathit{ctype}_n\ x_n) d_1 \ldots d_m;\ s, \Gamma}$$

Figure 7: Type Inference for C Statements

(Custom Exp) casts 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 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 integers 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.

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. Additionally, we require that $GC' \sqsubseteq GC$, since if f might call the garbage collector, so might the current function.

The last hypothesis in this rule is a constraint that requires that if the function may call the garbage collector, every variable which points into the OCaml heap and is still live must have been registered with a call to `CAMLprotect`. Here $\mathit{ValPtrs}(\Gamma)$ 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 $\mathit{live}(\Gamma)$ is all variables live at the program point corresponding to Γ. We omit the computation of *live*, since it is standard.

3.3.2 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 until G has reached a fixpoint.

Since type environments are flow-sensitive, some of our type rules will need to constrain type environments to be compatible with each other. We define $\Gamma \sqsubseteq \Gamma'$ if $\Gamma(x) \sqsubseteq \Gamma'(x)$ for all $x \in dom(\Gamma) \cup dom(\Gamma')$, and we define $\Gamma \sqcup \Gamma'$ as $\Gamma(x) \sqcup \Gamma'(x)$ for all $x \in dom(\Gamma) \cup dom(\Gamma')$. 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 *ct*s:

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

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

Figure 7 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 be empty so that any variables registered with the garbage collector are released. (CAMLReturn Stmt) is identical to (Ret Stmt) except that we require P to be non-empty. In each of (Goto Stmt), (Ret Stmt), and (CAMLReturn Stmt), we use *reset* to compute a new, unconstrained type environment following these statements, since these are unconditional branches.

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 expressions. In particular, in the else branch we must know the offset of the `boxed` data is 0, which will enable us to do further tag tests.

Similarly, in (If_sum_tag) 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. (We could also omit this last requirement.) 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).

(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 6. 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 modeled flow-sensitively. Again, we abuse notation and allow the right-hand side to be a function application checked with (App). (CAMLProtect Var) 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.

Finally, rules (Var Decl), (Fun Decl), and (Fun Defn) bind names in the environment. All of these rules use our mapping 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.

3.3.3 Applying the Type Inference Rules

We apply the type rules in Figures 6 and 7 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_tag). 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

```
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           /* B */          // x : α value[unboxed{0}]{0}
6       if_int_tag(x) == 1   // 2 ≤ ψ
7           /* D */          // x : α value[unboxed{0}]{1}
8   } else {
9       // x : α value[boxed{0}]{⊤}
10      if_sum_tag(x) == 0   // σ = π0 + σ'
11          /* A */          // x : α value[boxed{0}]{0}
12      if_sum_tag(x) == 1   // σ' = π1 + σ''
13          /* C */          // x : α value[boxed{0}]{1}
14  } // x : α value[⊤{0}]{⊤}
```

Figure 8: Example with types

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 iterative 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, and those lattices have finite height, and because re-applying the type inference rules produces strictly more unification constraints.

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.

3.4 Example

In Figure 8, we present 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`. Upon seeing the `if_unboxed` call, α unifies with the representational type (ψ, σ). 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 $1 \leq \psi$. Thus on line 5, x is now fully known and can be safely used as the nullary type constructor `B`. Similarly, on line 7, x is known to be the constructor `D`.

On the false branch of the `if_unboxed` test, our rule gives x a boxed type with offset 0. 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 `A`, 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 `C`. At line 14, we join all of the branches together and lose information about the boxedness and tag of x, and we have $\alpha = (\psi, \pi_0 + \pi_1 + \sigma'')$ with $2 \leq \psi$, which correctly unifies with our original type `t`. When this unification takes place, we will also discover $\sigma'' = \emptyset$.

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 the appendix. 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

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

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 $\to^*$ as the reflexive, transitive closure of $\to$.

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*
   - $\Gamma \vdash \{l+n\} : (\Psi, \Sigma)$ $\texttt{value}[\textit{boxed}\{n\}]\{m\}$
   - $\Sigma = \Pi_0 + \cdots + \Pi_j$, $m \le j$
   - $\Pi_m = mt_0 \times \cdots \times mt_k$, $n \le k$
   - $\Gamma \vdash S_{ML}(\{l+n\}) : mt_n$ $\texttt{value}[\top\{0\}]\{\top\}$
   - $S_{ML}(\{l+-1\}) = m$
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 appendix.

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 $\langle S_C, S_{ML}, V, s\rangle \to^* \langle S'_C, S'_{ML}, V', ()\rangle$.*

5 Implementation and Experiments

5.1 Implementation

We have implemented the inference system described in Section 3. 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 types they hide (when available). 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 [18], 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 6 and 7. 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 `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

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 9: Experimental Results

for indexing into `value` types rarely have their address taken, so this usually does not affect our analysis. Similar, 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 an unknown 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 differences in the sign of a type.

In addition to the types we have described so far, OCaml also includes objects and polymorphic variants. 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 (Section 5.2).

Finally, recall that our analysis of C functions is monomorphic. Thus we cannot infer quantified types for C functions that are polymorphic in OCaml `value` parameters. Instead, we allow them to be hand-annotated as polymorphic. 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 9 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 by directly accessing the option block as if it were the expected type rather than an option sum 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. 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. Clearly this is very dangerous, because OCaml will allow *any* argument to be passed to this function. 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 has type $t*$ custom, then both $((t*)v + 1)$ and $(t*)(v + \text{sizeof}(t*))$ are equivalent. However, our system infers v to have a custom pointer type in the first case, and a custom integer type in the second case, creating a unification error.

Finally, in several of the benchmarks there are a number of places where our tool issues a warning because it does not have precise enough information to compute a type. For instance, this may occur when computing the type of $e_1 \texttt{+}_p e_2$ if e_2 has the type $\texttt{int}[\top\{0\}]\{\top\}$, 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 [3, 7, 13, 14, 15]. For example, Blume [3] proposes a system allowing arbitrary C data types to be accessed by OCaml. Fisher et al [8] 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. 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.

Recently, researchers have developed systems to check that dynamically-generated SQL queries are well-formed [5, 6, 9]. 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 [20] 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 [10] and SOM [11] 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 [4, 17], 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) [12, 16] 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.

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] 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`.

[3] M. Blume. No-Longer-Foreign: Teaching an ML compiler to speak C "natively". In Benton and Kennedy [2]. `http://www.elsevier.nl/locate/entcs/volume59.html`.

[4] 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.

[5] 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.

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

[7] 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.

[8] K. Fisher, R. Pucella, and J. Reppy. A framework for interoperability. In Benton and Kennedy [2]. `http://www.elsevier.nl/locate/entcs/volume59.html`.

[9] 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.

[10] 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.

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

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

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

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

$$
\begin{array}{rcl}
v & ::= & n \mid l \mid \{n\} \mid \{l+n\} \\
e & ::= & v \mid x \mid *e \mid e \; aop \; e \mid e +_p e \\
 & \mid & \texttt{Val\_int}\ e \mid \texttt{Int\_val}\ e \\
lval & ::= & x \mid *(e +_p n) \\
s & ::= & () \mid L\texttt{:}\ s \mid s\ \texttt{;}\ s \mid \texttt{goto}\ L \\
 & \mid & lval\ \texttt{:=}\ e \mid \texttt{if}\ e\ \texttt{then}\ L \\
 & \mid & \texttt{if\_sum\_tag}(x) == n\ \texttt{then}\ L \\
 & \mid & \texttt{if\_int\_tag}(x) == n\ \texttt{then}\ L \\
 & \mid & \texttt{if\_unboxed}(x)\ \texttt{then}\ L
\end{array}
$$

Figure 10: Simplified Grammar for Expressions and Statements

$$
\begin{array}{rcl}
R & ::= & [] \mid *R \mid R \; aop \; e \mid v \; aop \; R \mid R +_p e \\
 & \mid & v +_p R \mid \texttt{Val\_int}\ R \mid \texttt{Int\_val}\ R \\
 & \mid & R\ \texttt{;}\ s \mid \texttt{if}\ R\ \texttt{then}\ L \\
 & \mid & R\ \texttt{:=}\ e \mid v\ \texttt{:=}\ R
\end{array}
$$

Figure 11: Reduction Context

A Soundness Proof

In this appendix we develop a proof of soundness for a slightly simplified version of our multi-lingual type system. Our modified source language is shown in Figure 10. We have removed functions, casting operations, and `CAMLprotect` and `CAMLreturn`. We believe these features can be added without difficulty, though with more tedium. Thus we focus on checking the sequence of statements that forms the body of a function, with branches but no function calls. We have added to our grammar the empty statement () and a new non-terminal v for syntactic values, which are C integers n, C pointers l, OCaml integers $\{n\}$, and OCaml pointers $\{l+n\}$ (a pointer to base address l on the OCaml heap and offset n).

Our small-step semantics uses three stores S_C, S_{ML}, and V to model updatable references. Here, S_C maps C pointers l to values, S_{ML} maps OCaml pointers $\{l+n\}$ to values, and V maps variables x to values. Recall that a pointer into the OCaml heap points to a structured block just past the header. We therefore extend the definition of S_{ML} so that $S_{ML}(\{l+-1\})$ is defined to be the runtime tag of the block pointed to by l. We define a *configuration* to be a tuple $\langle S_C, S_{ML}, V, s\rangle$, meaning that statement s is being evaluated in the context of stores S_C, S_{ML}, and V. In order to enhance readability, we also allow configurations to contain expressions: $\langle S_C, S_{ML}, V, e\rangle$. To model branches, we define a mapping D from labels to the sequence of statements beginning at that label. Thus a branch to a label L results in the program evaluating statement $D(L)$ next.

As is standard, we define reduction contexts R in Figure 11 to specify the order of evaluation in our semantics. Here, each expression contains a hole [] which shows what must be evaluated next. We therefore use the notation $R[e]$ to mean the reduction context R where the hole is replaced by e.

Our small-step operational semantics are shown in Figure 12. These rules 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 paired with stores S_C,S_{ML}, and V, reduces to a new statement s' and yields new stores S'_C, S'_{ML}, and V'. Note that in Figure 12(a) reducing expressions does not yield any new stores since expressions in our language are side-effect free. We define $\rightarrow^*$ to be the reflexive, transitive closure of $\rightarrow$.

Most of the rules in Figure 12 are straightforward. In order to preserve soundness, we only allow trivial pointer arithmetic on C pointers in `o-c-add` and `o-c-assign`. In rules `o-ifsum` and `o-ifsum2`, recall that $S_{ML}(\{l+-1\})$

represents the run-time type tag of a structured block at location l.

We will prove soundness of the type checking versions of our rules shown in Figures 13 and 14. In Figure 13, we have added rules to type check C locations l, OCaml integers $\{n\}$, and OCaml locations $\{l+n\}$. Since these are checking rules, we assume that a label environment G has already been computed. In order to maintain soundness, we have also restricted e_2 to have value 0 in ADD C EXP, since S_C does not track the sizes of C memory blocks.

Our soundness proof will follow the usual pattern, showing subject reduction lemmas for statements and expressions. In our proof, as is standard, we will overload Γ so that in addition to containing types for variables, it will also contain types for C locations and ML locations. We define a *compatibility* relationship to define when a type environment Γ assigns correct types to the values stored in S_C, S_{ML}, and V:

Definition 4 (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 $x \in V$, $\Gamma \vdash V(x) : \Gamma(x)$*

We begin by showing that given any well typed expression that is not a value, one of the reduction rules from Figure 12(a) applies and the result of the reduction preserves the type of the expression.

Lemma 1 (Subject Reduction for Expressions) *If $\Gamma \vdash e : ct[B\{I\}]\{T\}$ and $\Gamma \sim \langle S_C, S_{ML}, V\rangle$, then either e is a value or there exists e' such that*

(1) $\langle S_C, S_{ML}, V, e\rangle \rightarrow \langle S_C, S_{ML}, V, e'\rangle$, *and*

(2) $\Gamma \vdash e' : ct[B\{I\}]\{T\}$

Proof: Proceed by induction on the structure of e:

case $n, l, \{n\}, \{l+n\}$: These are values, so there is nothing to prove.

case x: Since $\Gamma \vdash x : ct[B\{I\}]\{T\}$, x must be in the domain of Γ. Since $\Gamma \sim \langle S_C, S_{ML}, V\rangle$, x is also in the domain of V. Therefore we can apply the rule `o-var` to show (1), letting $e' = V(x)$. And then by compatibility, $\Gamma \vdash V(x) : ct[B\{I\}]\{T\}$, showing (2).

case $*e_1$: Note that either the type rule C DEREF EXP or VAL DEREF EXP may apply. First consider the former case. Since $\Gamma \vdash *e_1 : ct[B\{I\}]\{T\}$, C DEREF EXP states that $B = T = \top$, $I = 0$, and $\Gamma \vdash e_1 : ct\ *[\top\{0\}]\{\top\}$. If e_1 is not a value, then by induction there exists e_2 such that $\langle S_C, S_{ML}, V, e_1\rangle \rightarrow \langle S_C, S_{ML}, V, e_2\rangle$ and $\Gamma \vdash e_2 : ct\ *[\top\{0\}]\{\top\}$. So taking $e' = *e_2$, we have satisfied (1). Also, by applying the type rule C DEREF EXP to $*e_2$, we see that $\Gamma \vdash *e_2 : ct[\top\{0\}]\{\top\}$ and thus (2) is satisfied. For the remaining cases in this proof we can make a similar argument for the inductive case, and so rather than repeat this argument we will implicitly assume that all sub-expressions are values.

Now consider the case where C DEREF EXP applies and e_1 is a value. Recall that $\Gamma \vdash e_1 : ct\ *[\top\{0\}]\{\top\}$ and by examining the type rules, we see that the only rule which applies to a value with type $ct\ *$ is LOC EXP. Therefore e_1 must be a location l and the rule `o-c-deref` applies. Thus if we set $e' = S_C(l)$, then (1) is satisfied. Also, since $\Gamma \sim \langle S_C, S_{ML}, V\rangle$ and $\Gamma \vdash l : ct\ *[\top\{0\}]\{\top\}$ then $\Gamma \vdash S_C(l) : ct[\top\{0\}]\{\top\}$ by compatibility. Therefore $\Gamma' \vdash e' : ct[\top\{0\}]\{\top\}$ and (2) is satisfied.

Finally, consider the case where VAL DEREF EXP applies and e_1 is a value. Here, VAL DEREF EXP states

$$\frac{\Gamma \vdash e_1 : (\Psi, \Sigma)\ \texttt{value}[\texttt{boxed}\{n\}]\{m\} \qquad \Sigma = \Pi_0 + \cdots + \Pi_k \qquad m \leq k \qquad \Pi_m = mt_0 \times \cdots \times mt_j \qquad n \leq j}{\Gamma \vdash *e_1 : mt_n\ \texttt{value}[\top\{0\}]\{\top\}}$$

(o-var)	$\langle S_C, S_{ML}, V, R[x]\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, R[v]\rangle$	$V(x) = v$
(o-ml-add)	$\langle S_C, S_{ML}, V, R[\{l+n_1\} +_p n_2]\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, R[\{l+n\}]\rangle$	$n = n_1 + n_2$
(o-c-add)	$\langle S_C, S_{ML}, V, R[l +_p 0]\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, R[l]\rangle$	
(o-c-deref)	$\langle S_C, S_{ML}, V, R[*l]\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, R[v]\rangle$	$S_C(l) = v$
(o-ml-deref)	$\langle S_C, S_{ML}, V, R[*\{l+n\}]\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, R[v]\rangle$	$S_{ML}(\{l+n\}) = v$
(o-aop)	$\langle S_C, S_{ML}, V, R[n_1\ aop\ n_2]\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, R[n]\rangle$	$n = n_1\ aop\ n_2$
(o-valint)	$\langle S_C, S_{ML}, V, R[\texttt{Val\_int}\ n]\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, R[\{n\}]\rangle$	
(o-intval)	$\langle S_C, S_{ML}, V, R[\texttt{Int\_val}\ \{n\}]\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, R[n]\rangle$	

(a) Small-step Semantics for Expressions

(o-label)	$\langle S_C, S_{ML}, V, L : s; s'\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, s; s'\rangle$	
(o-goto)	$\langle S_C, S_{ML}, V, goto\ L; s\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, D(L)\rangle$	
(o-c-assign)	$\langle S_C, S_{ML}, V, *(l +_p 0) := v; s\rangle$	$\rightarrow$	$\langle S_C[l \mapsto v], S_{ML}, V, s\rangle$	
(o-ml-assign)	$\langle S_C, S_{ML}, V, *(\{l+n\}) := v; s\rangle$	$\rightarrow$	$\langle S_C, S_{ML}[\{l+n\} \mapsto v], V, s\rangle$	
(o-var-assign)	$\langle S_C, S_{ML}, V, x := v; s\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V[x \mapsto v], s\rangle$	
(o-if)	$\langle S_C, S_{ML}, V, \texttt{if}\ n\ \texttt{then}\ L; s\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, D(L)\rangle$	if $n \neq 0$
(o-if2)	$\langle S_C, S_{ML}, V, \texttt{if}\ n\ \texttt{then}\ L; s\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, s\rangle$	if $n = 0$
(o-ifsum)	$\langle S_C, S_{ML}, V, \texttt{if\_sum\_tag}(x) == n\ \texttt{then}\ L; s\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, D(L)\rangle$	if $(S_{ML}(\{l+-1\})) = n$ $V(x) = \{l+0\}$
(o-ifsum2)	$\langle S_C, S_{ML}, V, \texttt{if\_sum\_tag}(x) == n\ \texttt{then}\ L; s\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, s\rangle$	if $(S_{ML}(\{l+-1\})) \neq n$ $V(x) = \{l+0\}$
(o-ifi)	$\langle S_C, S_{ML}, V, \texttt{if\_int\_tag}(x) == n_2\ \texttt{then}\ L; s\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, D(L)\rangle$	if $n_1 = n_2$ $V(x) = \{n_1\}$
(o-ifi2)	$\langle S_C, S_{ML}, V, \texttt{if\_int\_tag}(x) == n_2\ \texttt{then}\ L; s\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, s\rangle$	if $n_1 \neq n_2$ $V(x) = \{n_1\}$
(o-iflong)	$\langle S_C, S_{ML}, V, \texttt{if\_unboxed}(x)\ \texttt{then}\ L; s\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, D(L)\rangle$	$V(x) = \{n\}$
(o-iflong2)	$\langle S_C, S_{ML}, V, \texttt{if\_unboxed}(x)\ \texttt{then}\ L; s\rangle$	$\rightarrow$	$\langle S_C, S_{ML}, V, s\rangle$	$V(x) = \{l+0\}$

(b) Small-step Semantics for Statements

Figure 12: Small-step Semantics Rules

Since e_1 has a boxed value type, it must be an ML location $\{l+n\}$ by the ML Loc Exp type rule. By compatibility, $\{l+n\} \in dom(S_{ML})$, and hence the rule `o-ml-deref` applies. Setting $e' = S_{ML}(\{l+n\})$, (1) is satisfied. Also by compatibility, $S_{ML}(\{l+-1\}) = m$ and $\Gamma \vdash S_{ML}(\{l+n\}) : mt_n\ \texttt{value}[\top\{0\}]\{\top\}$, and thus (2) is satisfied.

case $e_1 +_p e_2$: Here, the either Add C Exp or Add Val Exp may apply. In the former case, we see that $\Gamma \vdash e_1 : ct\ *[\top\{0\}]\{\top\}$ and $\Gamma \vdash e_2 : \texttt{int}[\top\{0\}]\{0\}$. Therefore by inspecting the type rules, the only value with a pointer type is a location l and thus $e_1 = l$ and similarly, $e_2 = 0$. Therefore the rule `o-c-add` applies and $\langle S_C, S_{ML}, V, e_1 +_p e_2\rangle \rightarrow \langle S_C, S_{ML}, V, l\rangle$ and thus we have satisfied (1). Also, by Add C Exp, $\Gamma \vdash l : ct\ *[\top\{0\}]\{\top\}$ and thus we have satisfied (2).

Now assume that Add Val Exp applies. Therefore, $\Gamma \vdash e_1 : (\Psi, \Sigma)[\texttt{boxed}\{n\}]\{n_1\}$ and $\Gamma \vdash e_2 : \texttt{int}[\top\{0\}]\{m\}$. By examining the type rules, the only value with a `boxed` type is an ML location, so e_1 must be of the form $\{l+n\}$. Similarly, e_2 must be an integer m. Therefore the rule `o-ml-add` applies and $\langle S_C, S_{ML}, V, e_1 +_p e_2\rangle \rightarrow \langle S_C, S_{ML}, V, \{l+n'\}\rangle$ where $n' = n + m$ and we have satisfied (1). By again examining the type rules, we see that $\Gamma \vdash \{l+n'\} : (\Psi, \Sigma)[\texttt{boxed}\{n'\}]\{n_1\}$ and thus (2) is satisfied.

case $e_1\ aop\ e_2$: By examining the type rules, we see that the only rule that applies is AOP Exp and thus $\Gamma \vdash e_1\ aop\ e_2 : \texttt{int}[\top\{0\}]\{T\}$, $\Gamma \vdash e_1 : \texttt{int}[\top\{0\}]\{T_1\}$ and $\Gamma \vdash e_2 : \texttt{int}[\top\{0\}]\{T_2\}$ where $T = T_1\ aop\ T_2$. By again looking at the type rules, we see that the only values which have type `int` are integer values by Int Exp, therefore e_1 and e_2 must be some values n_1 and n_2, respectively, with $n_1 \sqsubseteq T_1$ and $n_2 \sqsubseteq T_2$. Thus the rule `o-aop` applies and we can let $e' = n$ where $n = n_1\ aop\ n_2$ and (1) is satisfied. Also, the rule Int Exp again applies since $n \sqsubseteq T$ and thus $\Gamma \vdash n : \texttt{int}[\top\{0\}]\{T\}$ and (2) is satisfied.

$$\textsc{Int Exp}\ \frac{0 \sqsubseteq I \qquad n \sqsubseteq T}{\Gamma \vdash n : \texttt{int}[\top\{I\}]\{T\}}$$

$$\textsc{Loc Exp}\ \frac{\Gamma(l) = ct\ *[\top\{I\}]\{\top\} \qquad 0 \sqsubseteq I}{\Gamma \vdash l : \Gamma(l)}$$

$$\textsc{ML Int Exp}\ \frac{n+1 \leq \Psi \qquad \mathsf{unboxed} \sqsubseteq B \qquad 0 \sqsubseteq I \qquad n \sqsubseteq T}{\Gamma \vdash \{n\} : (\Psi, \Sigma)\ \texttt{value}[B\{I\}]\{T\}}$$

$$\textsc{ML Loc Exp}\ \frac{\Gamma(\{l+n\}) = (\Psi, \Sigma)[B\{I\}]\{T\} \quad \mathsf{boxed} \sqsubseteq B \qquad n \sqsubseteq I \qquad m \sqsubseteq T \qquad \Sigma = \Pi_0 + \cdots + \Pi_j \qquad m \leq j \qquad \Pi_m = mt_0 \times \cdots \times mt_k \qquad n \leq k}{\Gamma \vdash \{l+n\} : \Gamma(\{l+n\})}$$

$$\textsc{Var Exp}\ \frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

$$\textsc{Add Val Exp}\ \frac{\Gamma \vdash e_1 : (\Psi, \Sigma)\ \texttt{value}[\mathsf{boxed}\{n\}]\{n'\} \quad \Gamma \vdash e_2 : \texttt{int}[\top\{0\}]\{m\} \qquad \Sigma = \Pi_0 + \cdots + \Pi_k \qquad n' \leq k \qquad \Pi_{n'} = mt_0 \times \cdots \times mt_j \qquad n + m \leq_t otj}{\Gamma \vdash e_1 +_p e_2 : (\Psi, \Sigma)\ \texttt{value}[\mathsf{boxed}\{n+m\}]\{n'\}}$$

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

$$\textsc{Val Deref Exp}\ \frac{\Gamma \vdash e_1 : (\Psi, \Sigma)\ \texttt{value}[\mathsf{boxed}\{n\}]\{m\} \qquad \Sigma = \Pi_0 + \cdots + \Pi_k \qquad m \leq k \qquad \Pi_m = mt_0 \times \cdots \times mt_j \qquad n \leq j}{\Gamma \vdash *e : mt_n\ \texttt{value}[\top\{0\}]\{\top\}}$$

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

$$\textsc{AOP Exp}\ \frac{\Gamma \vdash e_1 : \texttt{int}[\top\{0\}]\{T\} \qquad \Gamma \vdash e_2 : \texttt{int}[\top\{0\}]\{T'\}}{\Gamma \vdash e_1\ aop\ e_2 : \texttt{int}[\top\{0\}]\{T\ aop\ T'\}}$$

$$\textsc{Val Int Exp}\ \frac{\Gamma \vdash e : \texttt{int}[\top\{0\}]\{T\} \qquad T+1 \leq \Psi}{\Gamma \vdash \texttt{Val\_int}\ e : (\Psi, \Sigma)\ \texttt{value}[\mathsf{unboxed}\{0\}]\{T\}}$$

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

Figure 13: Type Checking Rules for C Expressions

$$\textsc{Empty Stmt}\ \frac{}{\Gamma, G \vdash (), \Gamma} \qquad \textsc{Seq Stmt}\ \frac{\Gamma, G \vdash s_1, \Gamma' \qquad \Gamma', G \vdash s_2, \Gamma''}{\Gamma, G \vdash s_1\ ;\ s_2, \Gamma''} \qquad \textsc{Lbl Stmt}\ \frac{G(L), G \vdash s, \Gamma' \qquad \Gamma \sqsubseteq G(L)}{\Gamma, G \vdash L\texttt{:}\ s, \Gamma'}$$

$$\textsc{Goto Stmt}\ \frac{\Gamma \sqsubseteq G(L)}{\Gamma, G \vdash \texttt{goto}\ L, reset(\Gamma)} \qquad \textsc{If Stmt}\ \frac{\Gamma \vdash e : \texttt{int}[\top\{0\}]\{T\} \qquad \Gamma \sqsubseteq G(L)}{\Gamma, G \vdash \texttt{if}\ e\ \texttt{then}\ L, \Gamma}$$

$$\textsc{If\_unboxed Stmt}\ \frac{\Gamma, G \vdash x : mt[B\{0\}]\{T\} \qquad \Gamma' = \Gamma[x \mapsto mt\ \texttt{value}[\texttt{unboxed}\{0\}]\{T\}] \qquad \Gamma' \sqsubseteq G(L)}{\Gamma, G \vdash \texttt{if\_unboxed}(x)\ \texttt{then}\ L, \Gamma[x \mapsto mt\ \texttt{value}[\texttt{boxed}\{0\}]\{T\}]}$$

$$\textsc{If\_sum\_tag Stmt}\ \frac{\Gamma \vdash x : mt\ \texttt{value}[\texttt{boxed}\{0\}]\{T\} \qquad mt = (C, \Pi_0 + \cdots + \Pi_n + \Sigma) \qquad \Gamma' = \Gamma[x \mapsto mt\ \texttt{value}[\texttt{boxed}\{0\}]\{n\}] \qquad \Gamma' \sqsubseteq G(L)}{\Gamma, G \vdash \texttt{if\_sum\_tag}(x) == n\ \texttt{then}\ L, \Gamma}$$

$$\textsc{If\_int\_tag Stmt}\ \frac{\Gamma \vdash x : mt\ \texttt{value}[\texttt{unboxed}\{0\}]\{T\} \qquad mt = (\Psi, \Sigma) \qquad n+1 \leq \Psi \qquad \Gamma' = \Gamma[x \mapsto mt\ \texttt{value}[\texttt{unboxed}\{0\}]\{n\}] \qquad \Gamma' \sqsubseteq G(L)}{\Gamma, G \vdash \texttt{if\_int\_tag}(x) == n\ \texttt{then}\ L, \Gamma}$$

$$\textsc{LSet Stmt}\ \frac{\Gamma \vdash *(e_1 +_p n) : ct[\top\{0\}]\{\top\} \qquad \Gamma \vdash e_2 : ct[B\{0\}]\{T\}}{\Gamma, G \vdash *(e_1 +_p n)\ \texttt{:=}\ e_2, \Gamma}$$

$$\textsc{VSet Stmt}\ \frac{\Gamma \vdash e : ct[B\{I\}]\{T\}}{\Gamma, G \vdash x\ \texttt{:=}\ e, \Gamma[x \mapsto ct[B\{I\}]\{T\}]}$$

Figure 14: Type Checking Rules for C Statements

case `Val_int` e_1: By inspecting the type rules, we see that the only rule that applies is Val Int Exp:

$$\frac{\Gamma \vdash e_1 : \texttt{int}[\top\{0\}]\{T\} \qquad T+1 \leq \Psi}{\Gamma \vdash \texttt{Val\_int}\ e_1 : (\Psi, \Sigma)\ \texttt{value}[\texttt{unboxed}\{0\}]\{T\}}$$

Therefore since e_1 is a value, it must be some number n with $n \sqsubseteq T$ by Int Exp. Thus the rule `o-valint` applies with $e' = \{n\}$ and (1) is satisfied. If T is $\top$, then Ψ must be ∞ and we can show $\Gamma \vdash \{n\} : (\infty, \Sigma)\ \texttt{value}[\texttt{unboxed}\{0\}]\{\top\}$ by ML Int Exp. If T is some number m then Int Exp states that $n = m$. Therefore we can show $\Gamma \vdash \{n\} : (\Psi, \Sigma)\ \texttt{value}[\texttt{unboxed}\{0\}]\{n\}$ by ML Int Exp. Note that T can not be $\bot$ since it is a value. Therefore we have exhausted all cases for T and (2) is satisfied.

case `Int_val` e_1: By inspecting the type rules, we see that the only rule that applies is Int Val Exp:

$$\frac{\Gamma \vdash e_1 : mt\ \texttt{value}[\texttt{unboxed}\{0\}]\{T\}}{\Gamma \vdash \texttt{Int\_val}\ e_1 : \texttt{int}[\top\{0\}]\{T\}}$$

Since e_1 is a value, it must have the form $\{n\}$ by ML Int Exp. Therefore the rule `o-intval` applies and setting $e' = n$ satisfies (1). If $T = \top$ then clearly $\Gamma \vdash n : \texttt{int}[\top\{0\}]\{\top\}$ by Int Exp. If $T = m$ then ML Int Exp states that $n = m$ and thus $\Gamma \vdash n : \texttt{int}[\top\{0\}]\{n\}$. Note that T can not be $\bot$ since it is a value. Therefore we have exhausted all cases for T and (2) is satisfied.

□

We next show subject reduction for statements. Recall that typing judgments for statements include label environments G. Thus we introduce a notion of compatibility of G with our statement store D, similar to the $\sim$ relation defined above:

Definition 5 (L-Compatibility) *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 $G(L), G \vdash D(L), \Gamma$.*

As we said above, whenever we branch to a label L, the next statement to be evaluated is $D(L)$. This is only valid if the statement to which D maps L is a labeled statement. Formally:

Definition 6 (Well Formedness of D) *D is said to be well formed if for all $L \in D$, $D(L)$ is a statement of the form $L : s$.*

Recall from Section 3.3.2 that we define $\Gamma \sqsubseteq \Gamma'$ if $\Gamma(x) \sqsubseteq \Gamma'(x)$ for all $x \in dom(\Gamma) \cup dom(\Gamma')$. Since compatibility is an important property to preserve in our subject reduction lemma for statements, we first present a result that shows that store compatibility follows this relation.

Lemma 2 *If* $\Gamma_1 \sqsubseteq \Gamma_2$ *and* $\Gamma_1 \sim \langle S_C, S_{ML}, V\rangle$ *then* $\Gamma_2 \sim \langle S_C, S_{ML}, V\rangle$

Proof: Let $l \in dom(S_C)$. Then by compatibility, there exists ct such that $\Gamma_1 \vdash S_C(l) : ct[\top\{0\}]\{\top\}$. Since $\Gamma_1 \sqsubseteq \Gamma_2$, then $\Gamma_2 \vdash S_C(l) : ct[\top\{I\}]\{\top\}$ where $0 \sqsubseteq I$. Since S_C maps locations to values, $S_C(l)$ could be one of n, l, $\{n\}$, or $\{l+n\}$. By examining the type rules, we see that Int Exp, Loc Exp, and ML Int Exp can assign $I = 0$ in the first three cases. Therefore consider when $S_C(l) = \{l' + n\}$. Since $\Gamma_1 \vdash \{l' + n\} : ct[\top\{0\}]\{\top\}$, then $n = 0$ by ML Loc Exp. Therefore $\Gamma_2 \vdash \{l' + 0\} : ct[\top\{0\}]\{\top\}$ by ML Loc Exp. Thus Γ_2 is compatible with S_C.

Let $\{l + n\} \in dom(S_{ML})$. Then by compatibility, there exists ct such that $\Gamma_1 \vdash S_{ML}(\{l + n\}) : ct[\top\{0\}]\{\top\}$. Since $\Gamma_1 \sqsubseteq \Gamma_2$, then $\Gamma_2 \vdash S_{ML}(\{l + n\}) : ct[\top\{I\}]\{\top\}$ where $0 \sqsubseteq I$. Since S_{ML} maps locations to values, $I = 0$ by a parallel argument to the above case and thus $\Gamma_2 \vdash S_{ML}(\{l + n\}) : ct[\top\{0\}]\{\top\}$. Since Γ_1 and Γ_2 only differ in the tags which they assign $\{l + n\}$ and not the ct type, we see that Γ_2 trivially satisfies the remaining compatibility requirements with S_{ML}.

Finally, let $x \in dom(V)$. Since Γ_1 is compatible with V, there exists ct, B, I, T such that $\Gamma_1 \vdash x : ct[B\{I\}]\{T\}$ and $\Gamma_1 \vdash V(x) : ct[B\{I\}]\{T\}$. Since $\Gamma_1 \sqsubseteq \Gamma_2$, $\Gamma_2 \vdash x : ct[B'\{I'\}]\{T'\}$ where $B \sqsubseteq B'$, $I \sqsubseteq I'$, and $T \sqsubseteq T'$. Since V maps variables to values, $V(x)$ must be one of n, $\{n\}$, l, or $\{l + n\}$. If $V(x) = n$, then Int Exp applies and $B = B' = \top$. Since $I \sqsubseteq I'$ and $0 \sqsubseteq I$, then $0 \sqsubseteq I'$. Similarly, since $T \sqsubseteq T'$ and $n \sqsubseteq T$ then $n \sqsubseteq T'$. Therefore Int Exp again applies and we see that Γ_2 will assign a compatible type. If $V(x) = l$, then Loc Exp applies and $B = B' = T = T' = \top$. Since $0 \sqsubseteq I$ and $I \sqsubseteq I'$, then $0 \sqsubseteq I'$ and thus Γ_2 will again assign a compatible type in this case. If $V(x) = \{n\}$ then ML Int Exp applies and since $\mathsf{unboxed} \sqsubseteq B$ and $B \sqsubseteq B'$, then $\mathsf{unboxed} \sqsubseteq B'$. Similarly $0 \sqsubseteq I'$ and $n \sqsubseteq T'$ and thus Γ_2 will assign a compatible type. Finally, if $V(x) = \{l + n\}$, then since $\mathsf{boxed} \sqsubseteq B$ and $B \sqsubseteq B'$, then $\mathsf{boxed} \sqsubseteq B'$. Similarly $0 \sqsubseteq I'$ and $n \sqsubseteq T'$. Thus in all cases Γ_2 will assign a compatible type to $V(x)$. □

Several of our statements given in Figure 10 contain a label L which the program may branch to. Therefore we first present a lemma for this common case:

Lemma 3 *If* $\Gamma_1 \sim \langle S_C, S_{ML}, V\rangle$, $D \sim_L G$, D *is well formed, and* $\Gamma_1 \sqsubseteq G(L)$, *then for any statement* s *such that* $\Gamma_1, G \vdash s, \Gamma_2$ *and*

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

there exist Γ_3, s' *such that*

(I) $\langle S_C, S_{ML}, V, s\rangle \rightarrow \langle S_C, S_{ML}, V, L : s'\rangle$,

(II) $G(L) \sim \langle S_C, S_{ML}, V\rangle$, *and*

(III) $G(L), G \vdash L : s', \Gamma_3$

Proof: Since D is well formed, there exists s' such that

$$\langle S_C, S_{ML}, V, s\rangle \rightarrow \langle S_C, S_{ML}, V, L : s'\rangle$$

thus showing (I). Also, since $D \sim_L G$, there exists Γ_3 such that $G(L), G \vdash L : s', \Gamma_3$ satisfying (II). Since $\Gamma_1 \sqsubseteq G(L)$ then by Lemma 2, $G(L) \sim \langle S_C, S_{ML}, V\rangle$. (3). □

Finally, we show subject reduction for statements. All of our statements will be reduced in one of three ways which correspond to the three possible conclusions below. Either the statement contains a sub-expression which can be reduced, the statement is part of a sequence $s_1; s_2$ and reduces to the second statement, or the statement makes a branch to a label. Note that certain statements may support more than one conclusion (if a sequence contains a label statement), which is allowed since we only require at least one conclusion to hold. Each conclusion is similar in that it ensures that at every step of the program: (a) it is possible to take a step, (b) the stores are still compatible with the type environments at that step, and (c) the new statement is still well typed.

Lemma 4 (Subject Reduction for Statements) *If* s *is a statement,* $\Gamma_1, G \vdash s, \Gamma_2$, $\Gamma_1 \sim \langle S_C, S_{ML}, V\rangle$, $D \sim_L G$, *and* D *is well formed then either* $s = ()$ *or* $s = s_1; s_2$ *and one of the following must hold:*

(1) There exist Γ'_1, s'_1 *such that*

(a) $\langle S_C, S_{ML}, V, s_1; s_2\rangle \rightarrow \langle S_C, S_{ML}, V, s'_1; s_2\rangle$

(b) $\Gamma_1' \sim \langle S_C, S_{ML}, V \rangle$

(c) $\Gamma_1', G \vdash s_1'; s_2, \Gamma_2$

(2) There exist $\Gamma_1', S_C', S_{ML}', V'$ *such that*

(a) $\langle S_C, S_{ML}, V, s_1; s_2 \rangle \rightarrow \langle S_C', S_{ML}', V', s_2 \rangle$

(b) $\Gamma_1' \sim \langle S_C', S_{ML}', V' \rangle$

(c) $\Gamma_1', G \vdash s_2, \Gamma_2$

(3) There exist Γ_4, s_3 *such that*

(a) $\langle S_C, S_{ML}, V, s_1; s_2 \rangle \rightarrow \langle S_C, S_{ML}, V, L : s_3 \rangle$

(b) $G(L) \sim \langle S_C, S_{ML}, V \rangle$

(c) $G(L), G \vdash L : s_3, \Gamma_4$

Proof: By induction on the structure of s_1:

case L`:` s' In this case, the rule `o-label` applies and thus

$$\langle S_C, S_{ML}, V, L : s'; s_2 \rangle \rightarrow \langle S_C, S_{ML}, V, s'; s_2 \rangle$$

and so we are in case (1) and (a) has been satisfied. Further, the type rule Lbl Stmt applies and thus $\Gamma_1 \sqsubseteq G(L)$ and

$$G(L), G \vdash s', \Gamma_2$$

Thus selecting $\Gamma_1' = G(L)$ satisfies (c). Note also that since $\Gamma_1 \sqsubseteq \Gamma_1'$, $\Gamma_1' \sim \langle S_C, S_{ML}, V \rangle$ by Lemma 2 and (b) is satisfied.

case `goto` L In this case, the rule `o-goto` applies and thus

$$\langle S_C, S_{ML}, V, \texttt{goto}\ L; s_2 \rangle \rightarrow \langle S_C', S_{ML}', V', D(L) \rangle$$

Note also that the type rule Goto Stmt applies and thus $\Gamma_1 \sqsubseteq G(L)$. Therefore by Lemma 3 we have shown conclusion (3).

case e_1 `:=` e_2 In this case, we have several sub-cases depending on whether e_1 or e_2 is a value or not and which type rule applies. First consider the case where e_2 is not a value and thus we will show (1). Note that either VSet Stmt or LSet Stmt can apply. First consider the case where the rule VSet Stmt applies. Here, there must exist ct, B, I, T such that $\Gamma_1 \vdash e_2 : ct[B\{I\}]\{T\}$. Since e_2 is not a value, Lemma 1 states there exists an e_2' such that $\langle S_C, S_{ML}, V, e_2 \rangle \rightarrow \langle S_C, S_{ML}, V, e_2' \rangle$ with $\Gamma_1 \vdash e_2' : ct[B\{I\}]\{T\}$. Therefore

$$\langle S_C, S_{ML}, V, e_1 := e_2; s' \rangle \rightarrow \langle S_C, S_{ML}, V, e_1 := e_2'; s' \rangle$$

and we have satisfied (a). Since $\Gamma_1 \vdash e_2' : ct[B\{I\}]\{T\}$ then $\Gamma_1, G \vdash e_1 := e_2'; s', \Gamma_2$ and thus we have shown (b). (1) can also be shown when LSet Stmt applies (and e_2 not a value) by a parallel argument.

Now we will consider the cases when e_2 is a value. Note that according to our grammar e_1 can either be of the form x or $*(e_3 +_p n)$.

If $e_1 = x$ then we will show conclusion (2). Here, the rule `o-var-assign` applies and thus

$$\langle S_C, S_{ML}, V, x := v; s \rangle \rightarrow \langle S_C, S_{ML}, V', s \rangle$$

where $V' = V[x \mapsto v]$ and so conclusion (a) is satisfied.

Furthermore, the type rule VSet Stmt applies and thus there exist ct, B, I, T such that $\Gamma_1 \vdash e_2 : ct[B\{I\}]\{T\}$. The rule also states that $\Gamma_1, G \vdash x := e_2, \Gamma_1'$ where $\Gamma_1' = \Gamma_1[x \mapsto ct[B\{I\}]\{T\}$. Therefore $\Gamma_1' \vdash V'(x) : \Gamma_1'(x)$. Since S_C and S_{ML} are unchanged, then $\Gamma_1' \sim \langle S_C, S_{ML}, V' \rangle$ and we have shown (b).

Recall from our hypothesis that $\Gamma_1, G \vdash s_1; s_2, \Gamma_2$ and thus $\Gamma_1, G \vdash x := e_2; s_2, \Gamma_2$. Here, the type rule Seq Stmt applies and since $\Gamma_1, G \vdash x := e_2, \Gamma_1'$, then $\Gamma_1', G \vdash s_2, \Gamma_2$ and thus we have shown (c).

Now consider the case where e_1 has the form of $*(e_3 +_p n)$. If e_3 is not a value, then we can show that (1) holds by a parallel argument to the case where e_2 was not a value. If e_3 is a value, then the rule LSet Stmt applies and we will show (2). Recall that LSet Stmt states that there exists ct such that $\Gamma_1 \vdash *(e_3 +_p n) : ct[\top\{0\}]\{\top\}$. By examining the type rules, we see that either C Deref Exp or Val Deref Exp may apply.

First consider the case when C Deref Exp applies and therefore $\Gamma_1 \vdash (e_3 +_p n) : ct\ *[\top\{0\}]\{\top\}$. The only type rule which applies to $(e_3 +_p n)$ is Add C Exp and therefore $\Gamma_1 \vdash e_3 : ct\ *[\top\{0\}]\{\top\}$ and $\Gamma_1 \vdash n : \texttt{int}[\top\{0\}]\{0\}$. By again examining the type rules, the only value with a C pointer type is a location l and therefore since e_3 is a value, $e_3 = l$. Similarly, we see that n must be 0. Therefore the rule `o-c-assign` applies and thus

$$\langle S_C, S_{ML}, V, *(l +_p 0) := e_2; s' \rangle \rightarrow \langle S'_C, S_{ML}, V, s' \rangle$$

where $S'_C = S_C[l \mapsto e_2]$ and thus we have shown (a). Since $\Gamma_1 \vdash e_2 : ct[B\{0\}]\{T\}$ then $\Gamma_1 \vdash e_2 : ct[\top\{0\}]\{\top\}$. Since S_{ML} and V are unchanged then $\Gamma_1 \sim \langle S_c, S_{ML}, V \rangle$ and thus we have shown conclusion (b).

Recall from our hypothesis that $\Gamma_1, G \vdash s_1; s_2, \Gamma_2$ and thus $\Gamma_1, G \vdash *(e_3 +_p n) := e_2; s_2, \Gamma_2$. Here, the type rule Seq Stmt applies and since $\Gamma_1, G \vdash *(e_3 +_p n) := e_2, \Gamma_1$, then $\Gamma_1, G \vdash s_2, \Gamma_2$ and thus we have shown (c), concluding the case where C Deref Exp applies.

Finally, consider the case where e_1 has the form $*(e_3 +_p n)$, e_3 is a value, and Val Deref Exp applies to $*(e_3 +_p n)$. Here, we will show conclusion (2). Recall type rule Val Deref Exp:

$$\frac{\begin{array}{c}\Gamma_1 \vdash (e_3 +_p n) : (\Psi, \Sigma)\ \texttt{value}[\texttt{boxed}\{n_1\}]\{m\} \\ \Sigma = \Pi_0 + \cdots + \Pi_k \qquad m \leq k \qquad \Pi_m = mt_0 \times \cdots \times mt_j \qquad n_1 \leq j\end{array}}{\Gamma_1 \vdash *(e_3 +_p n) : mt_{n_1}\ \texttt{value}[\top\{0\}]\{\top\}}$$

(and thus $ct = mt_{n_1}$). Therefore, by LSet Stmt, $\Gamma_1 \vdash e_2 : mt_{n_1}\ \texttt{value}[B\{0\}]\{T\}$. Note that the only rule which applies to $(e_3 +_p n)$ in this situation is Add Val Exp:

$$\frac{\begin{array}{c}\Gamma_1 \vdash e_3 : (\Psi, \Sigma)\ \texttt{value}[\texttt{boxed}\{n_2\}]\{m\} \\ \Gamma_1 \vdash n : \texttt{int}[\top\{0\}]\{n\} \qquad \Sigma = \Pi_0 + \cdots + \Pi_k \qquad m \leq k \qquad \Pi_m = mt_0 \times \cdots \times mt_j \qquad n + n_2 \leq j\end{array}}{\Gamma_1 \vdash e_3 +_p n : (\Psi, \Sigma)\ \texttt{value}[\texttt{boxed}\{n_2 + n\}]\{m\}}$$

(and note $n_1 = n + n_2$). By again examining the type rules, we see that the only value with an ML pointer type is an ML location, and thus $e_3 = \{l + n_2\}$. Therefore the rule `o-ml-assign` applies and thus

$$\langle S_C, S_{ML}, V, *(e_3 +_p n) := e_2; s' \rangle \rightarrow \langle S_C, S'_{ML}, V, s' \rangle$$

where $S'_{ML} = S_{ML}[\{l + n_1\} \mapsto e_2]$ and thus we have shown (a).

Since $\Gamma_1 \vdash e_2 : mt_{n_1}[B\{0\}]\{T\}$, then all of the bullets in requirement (3.) of compatibility are still satisfied except for the fourth. However, Since $\Gamma_1 \vdash e_2 : ct[B\{0\}]\{T\}$ then clearly $\Gamma_1 \vdash e_2 : ct[\top\{0\}]\{\top\}$ and thus Γ_1 is compatible with S'_{ML}. Therefore, since S_C and V have not changed, $\Gamma_1 \sim \langle S_C, S'_{ML}, V \rangle$ and thus (b) holds.

Recall from our hypothesis that $\Gamma_1, G \vdash s_1; s_2, \Gamma_2$ and thus $\Gamma_1, G \vdash *(e_3 +_p n) := e_2; s_2, \Gamma_2$. Here, the type rule Seq Stmt applies and since $\Gamma_1, G \vdash *(e_3 +_p n) := e_2, \Gamma_1$, then $\Gamma_1, G \vdash s_2, \Gamma_2$ and thus we have shown (c).

case `if` e `then` L Note that the type rule If Stmt applies and thus $\Gamma_1 \sqsubseteq G(L)$ and there exists T such that $\Gamma_1 \vdash e : \texttt{int}[\top\{0\}]\{T\}$. If e is not a value, then we are in case (1). By Lemma 1 there exists e' such that $\Gamma_1 \vdash e' : \texttt{int}[\top\{0\}]\{T\}$ and

$$\langle S_C, S_{ML}, V, e \rangle \rightarrow \langle S_C, S_{ML}, V, e' \rangle$$

Therefore $\Gamma_1, G \vdash \texttt{if}\ e'\ \texttt{then}\ L, \Gamma_2$ and

$$\langle S_C, S_{ML}, V, \texttt{if}\ e\ \texttt{then}\ L; s' \rangle \rightarrow \langle S_C, S_{ML}, V, \texttt{if}\ e'\ \texttt{then}\ L; s' \rangle$$

and thus we have shown conclusion (a). Since $\Gamma_1 \vdash e' : \texttt{int}[\top\{0\}]\{T\}$, then $\Gamma_1, G \vdash \texttt{if}\ e'\ \texttt{then}\ L; s', \Gamma_1$ which satisfies (c). Since our output environment is unchanged, then clearly (b) is satisfied.

If e is a value, then by inspecting the type rules, the only values with type `int` are numbers, so e must be some number n. If $n \neq 0$ then the rule `o-if` applies and thus

$$\langle S_C, S_{ML}, V, \texttt{if}\ e\ \texttt{then}\ L; s' \rangle \rightarrow \langle S_C, S_{ML}, V, D(L) \rangle$$

Since we have $\Gamma_1 \sqsubseteq G(L)$ from above, Lemma 3 can be applied and conclusion (3) is satisfied.

If $n = 0$ then the rule `o-if2` applies and we will show (2). Since `o-if2` applies, then

$$\langle S_C, S_{ML}, V, \texttt{if } e \texttt{ then } L; s' \rangle \rightarrow \langle S_C, S_{ML}, V, s' \rangle$$

which satisfies (a). Since s_1 has not updated the environment, then (b) holds trivially. Finally, since $\Gamma_1, G \vdash s_1; s_2, \Gamma_2$, and $\Gamma_1, G \vdash \texttt{if } e \texttt{ then } L, \Gamma_1$, then $\Gamma_1, G \vdash s_2, \Gamma_2$ by SEQ STMT and thus (2) is satisfied.

case $\texttt{if\_sum\_tag}(x) == n \texttt{ then } L$ Note that the type rule IF_SUM_TAG STMT applies and thus $\Gamma_1' \sqsubseteq G(L)$. Also by IF_SUM_TAG STMT, there exist mt, T such that $\Gamma_1 \vdash x : mt \texttt{ value}[\texttt{boxed}\{0\}]\{T\}$. Since Γ_1 is compatible with V, then $\Gamma_1 \vdash V(x) : mt \texttt{ value}[\texttt{boxed}\{0\}]\{T\}$. Since $V(x)$ must be value, then by inspecting the type rules, the only values with type $mt \texttt{ value}[\texttt{boxed}\{0\}]\{T\}$ are ML locations, so $V(x)$ must be some location $\{l + 0\}$. Also, note that $\{l + 0\} \in S_{ML}$ and thus $S_{ML}(\{l + -1\}) = m$ by compatibility.

If $m = n$ then since $\Gamma_1' \vdash x : mt \texttt{ value}[\texttt{boxed}\{0\}]\{n\}$, then $\Gamma_1' \sim \langle S_C, S_{ML}, V \rangle$. Also the rule `o-ifsum` applies and thus

$$\langle S_C, S_{ML}, V, \texttt{if\_sum\_tag}(x) == n \texttt{ then } L; s' \rangle \rightarrow \langle S_C, S_{ML}, V, D(L) \rangle$$

Since we showed $\Gamma_1' \sqsubseteq G(L)$ above and $\Gamma_1' \sim \langle S_C, S_{ML}, V \rangle$, we can apply Lemma 3 and thus conclusion (3) is satisfied.

If $m \neq n$ then the rule `o-ifsum2` applies and thus

$$\langle S_C, S_{ML}, V, \texttt{if\_sum\_tag}(x) == n \texttt{ then } L; s' \rangle \rightarrow \langle S_C, S_{ML}, V, s' \rangle$$

and (2a) is satisfied. Since s_1 has not updated the environment, compatibility holds trivially and thus (2b) is satisfied. Finally, recall from our hypothesis that $\Gamma_1, G \vdash s_1; s_2, \Gamma_2$ and thus $\Gamma_1, G \vdash \texttt{if\_sum\_tag}(x) == n \texttt{ then } L; s_2, \Gamma_2$. Here, the type rule SEQ STMT applies and since $\Gamma_1, G \vdash \texttt{if\_sum\_tag}(x) == n \texttt{ then } L, \Gamma_1$, then $\Gamma_1, G \vdash s_2, \Gamma_2$ and thus we have shown (2c).

case $\texttt{if\_int\_tag}(x) == n \texttt{ then } L$ Note that the type rule IF_INT_TAG STMT applies and thus $\Gamma_1' \sqsubseteq G(L)$ and there exist mt, T such that $\Gamma_1 \vdash x : mt \texttt{ value}[\texttt{unboxed}\{0\}]\{T\}$. Also $\Gamma_1 \vdash V(x) : mt \texttt{ value}[\texttt{unboxed}\{0\}]\{T\}$ by compatibility. Since $V(x)$ must be a value, then by inspecting the type rules, the only values with type $mt \texttt{ value}[\texttt{unboxed}\{0\}]\{T\}$ are ML numbers, so $V(x)$ must be some number $\{m\}$. If $n = m$ then since $\Gamma_1' \vdash x : mt \texttt{ value}[\texttt{unboxed}\{0\}]\{n\}$, then $\Gamma_1' \sim \langle S_C, S_{ML}, V \rangle$. Also, the rule `o-ifi` applies and thus

$$\langle S_C, S_{ML}, V, \texttt{if\_int\_tag}(x) == n \texttt{ then } L; s' \rangle \rightarrow \langle S_C, S_{ML}, V, D(L) \rangle$$

Since $\Gamma_1' \sqsubseteq G(L)$ and $\Gamma_1' \sim \langle S_C, S_{Ml}, V \rangle$, then we can apply Lemma 3 and thus conclusion (3) is satisfied.

If $n \neq m$ then the rule `o-ifi2` applies and thus

$$\langle S_C, S_{ML}, V, \texttt{if\_int\_tag}(x) == n \texttt{ then } L; s' \rangle \rightarrow \langle S_C, S_{ML}, V, s' \rangle$$

and thus conclusion (2a) is satisfied. Since s_1 has not updated the environment, then conclusion (2b) hold trivially. Finally, recall from our hypothesis that $\Gamma_1, G \vdash s_1; s_2, \Gamma_2$ and thus $\Gamma_1, G \vdash \texttt{if\_int\_tag}(x) == n \texttt{ then } L; s_2, \Gamma_2$. Here, the type rule SEQ STMT applies and since $\Gamma_1, G \vdash \texttt{if\_int\_tag}(x) == n \texttt{ then } L, \Gamma_1$, then $\Gamma_1, G \vdash s_2, \Gamma_2$ and thus we have shown (2c).

case $\texttt{if\_unboxed}(x) \texttt{ then } L$ Note that the type rule IF_UNBOXED STMT applies and thus there exist mt, B, T such that $\Gamma_1 \vdash x : mt \texttt{ value}[B\{0\}]\{T\}$. Also, $\Gamma_1 \vdash V(x) : mt \texttt{ value}[B\{0\}]\{T\}$ by compatibility. Since $V(x)$ must be a value, then by inspecting the type rules, the only values with type $mt \texttt{ value}[B\{0\}]\{T\}$ are ML numbers and ML locations. Therefore $V(x)$ must be either a number $\{n\}$ or a location $\{l + 0\}$. If $V(x) = \{n\}$ then since $\Gamma_1' \vdash x : mt \texttt{ value}[\texttt{unboxed}\{0\}]\{T\}$ then Γ_1' is compatible with V by the type rule ML INT EXP and thus $\Gamma_1' \sim \langle S_C, S_{ML}, V \rangle$. Also, the rule `o-iflong` applies and thus

$$\langle S_C, S_{ML}, V, \texttt{if\_unboxed}(x) \texttt{ then } L; s' \rangle \rightarrow \langle S_C, S_{ML}, V, D(L) \rangle$$

Since we showed $\Gamma_1 \sqsubseteq G(L)$ by IF_UNBOXED STMT, we can apply Lemma 3 and thus conclusion (3) is satisfied.

If $V(x) = \{l + 0\}$ then let

$$\Gamma_1'' = \Gamma_1[x \mapsto mt\ \texttt{value}[\texttt{boxed}\{0\}]\{T\}]$$

By examining the type rules, we see that $\Gamma_1'' \vdash V(x) : \Gamma_1''(x)$ by ML LOC EXP and thus $\Gamma_1'' \sim \langle S_C, S_{ML}, V\rangle$ which satisfies conclusion (2b). Also, the rule `o-iflong2` applies and thus

$$\langle S_C, S_{ML}, V, \texttt{if\_unboxed}(e)\ \texttt{then}\ L; s'\rangle \rightarrow \langle S_C, S_{ML}, V, s'\rangle$$

which satisfies (2a). Finally, recall from our hypothesis that $\Gamma_1, G \vdash s_1; s_2, \Gamma_2$ and thus $\Gamma_1, G \vdash \texttt{if\_unboxed}(x)\ \texttt{then}\ L; s_2, \Gamma_2$. Here, the type rule SEQ STMT applies and since $\Gamma_1, G \vdash \texttt{if\_unboxed}(x)\ \texttt{then}\ L, \Gamma_1''$, then $\Gamma_1'', G \vdash s_2, \Gamma_2$ and thus we have shown (2c).

□

The standard approach to proving soundness is to show that if $e \rightarrow^* v$, then v is not *stuck*. Since statements in our language do not reduce to values, the only statement which is not stuck is the empty statement, (). Therefore it is sufficient to show that every statement either diverges or eventually reduces to ().

Theorem 2 (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 $\langle S_C, S_{ML}, V, s\rangle \rightarrow^* \langle S_C', S_{ML}', V', ()\rangle$.*

Proof: By Lemma 4 we can continually reduce the statement and reestablish our compatibly assumptions. Therefore either this process will continue forever, or there exists s' such that $\langle S_C, S_{ML}, V, s\rangle \rightarrow^* \langle S_C', S_{ML}', V', s'\rangle$ and for all s'', $\langle S_C, S_{ML}, V, s'\rangle \not\rightarrow \langle S_C', S_{ML}', V', s''\rangle$. Since s' is well typed by Lemma 4, it must be () or else we could apply Lemma 4 again and produce s''. □