Guilt Free Ivory

Trevor Elliott Lee Pike
Simon Winwood Pat Hickey
James Bielman Jamey Sharp
Galois, Inc.
{firstname.lastname}@galois.com

Eric Seidel
UC San Diego
eseidel@cs.ucsd.edu

John Launchbury
Willamette University
jlaunchb@willamette.edu

Abstract

Ivory is a language that enforces memory safety and avoids most undefined behaviors while providing low-level control of memory-manipulation. Ivory is embedded in a modern variant of Haskell, as implemented by the GHC compiler. The main contributions of the paper are two-fold. First, we demonstrate how to embed the type-system of a safe-C language into the type extensions of GHC. Second, Ivory is of interest in its own right, as a powerful language for writing high-assurance embedded programs. Beyond invariants enforced by its type-system, Ivory has direct support for model-checking, theorem-proving, and property-based testing. Ivory's semantics have been formalized and proved to guarantee memory safety.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages

Keywords Embedded Domain Specific Languages; Embedded Systems

1. Introduction

Recent reports of car-hacking via software flaws [11] and insecure low-level networking code [7] point toward the need for safe low-level programming languages. Languages like C or C++ are still the gold standard in embedded system development given the low-level control they provide in terms of memory usage and timing behavior. Unfortunately, these languages provide little support for creating high assurance software—they are unsafe and unanalyzable.

In this paper we present the language *Ivory*.[1] Ivory follows in the footsteps of other "safe C" programming languages, like Cyclone [20], BitC [33], and Rust [29]—languages that avoid many of the pitfalls of C, particularly related to memory safety and undefined behavior, while being suitable for writing low-level code (e.g., device drivers), and having minimal runtime systems.

Ivory is particularly designed for safety-critical embedded programming. Such a language should guarantee memory safety, prevent most undefined behaviors, and provide integrated testing and verification tools. Typical C coding conventions for safe embedded systems, such as those in use at NASA's Jet Propulsion Laboratory [21], are enforced by Ivory's type system. The major restrictions enforced by Ivory are restricted allocation for zero-overhead garbage collection, enforcing loops to have constant upper bounds, avoiding machine-dependent types (e.g., `int`), and safe (i.e., guaranteed non-null) dereferencing.

Ivory's implementation, however, is unique compared to previous safe C languages: Ivory is implemented as an embedded domain-specific language (EDSL) within Haskell. In addition to the benefits of rapid language development, this gives Ivory a powerful templating system—the language Haskell—allowing low-level programs to be written in a high-level style, despite some of the languages restrictions.

Ivory's type system is shallowly embedded within Haskell's type system, taking advantage of the extensions provided by GHC [26]. Thus, well-typed Ivory programs are guaranteed to produce memory safe executables, all without writing a stand-alone type-checker.

In contrast, the Ivory syntax is *deeply* embedded within Haskell. This novel combination of shallowly-embedded types and deeply-embedded syntax permits ease of development without sacrificing the ability to develop various back-ends and verification tools: in addition to the generation of embedded C for compilation, the Ivory language suite includes an integrated SMT-based symbolic simulator and a theorem-prover back-end. All these back-ends share the same AST: Ivory verifies what it compiles.

Ivory is not a toy language: we have used Ivory to write *SMACCMPilot* [19], a full-featured high-assurance autopilot for a small unpiloted air vehicle. Furthermore, Boeing has used Ivory to implement a level-of-interoperability for Stanag 4586 [17], a unpiloted air vehicle communications standard. We know of a few additional small projects by other developers in Ivory, as well. There are well over 100 KLoC of Ivory in existence.

Contributions The main research contribution of this paper is the design and implementation of Ivory: we show how to design a staged, type-safe, low-level language with a type system which guarantees the absence of the sorts of run-time errors common in low-level code, along with a powerful type-safe macro language. Previous safe-C languages have relied on specialized type-checkers, whereas we show it can be done (for the most part) within the type system of a general purpose functional programming language, GHC's implementation of Haskell (henceforth "GHC"). In GHC, a powerful subset of dependent typing features are available without sacrificing type-inference and decidable type-checking [26]; our work demonstrates a practical application of these extensions for a real-world language.

After providing a brief introduction to the Ivory language in Section 2, we describe Ivory's embedding in GHC's type system

[1] open-source (BSD3 license) and available at `ivory-lang.org`.

in Section 3. We highlight the aspects of the language particularly relevant to memory-safety (e.g., pointers, structures, and memory allocation). We also highlight shortcomings of the approach, describing aspects of the language that cannot be checked by the host language's type system (e.g., Ivory's module system).

Embedding a type system for a safe C language into GHC's type system is tricky business. To gain confidence that our embedding is correct, we formalized a model of Ivory in the Isabelle theorem prover [30], and used the model to formally prove progress and preservation properties for Ivory. In the process, we discovered minor bugs in Ivory's type embedding in GHC as well as generalizations to Ivory that still preserve safety. We describe the formalization, proofs, and extensions in Section 4.

Ivory goes beyond ensuring memory safety, the focus of most other safe C programming languages, and also provides automated support for preventing errors that result from other undefined behaviors in C (e.g., division by zero, left bit-shifts by a negative value, etc.) as well as support for checking user-provided assertions. Toward this end, Ivory supports writing user-supplied assertions and pre- and post-conditions on functions, and includes a built-in symbolic simulator targeting an SMT solver (CVC4 [10]), as well as an theorem-prover back-end targeting ACL2 [22]. For automated testing, a QuickCheck-like property-based test-case generator is integrated into Ivory. These tools are described in Section 5. In Section 6, we discuss some of the issues and our mitigations with using a large EDSL for embedded programming projects.

We describe related work in safe C language and EDSLs in Section 7 and provide concluding remarks in Section 8.

2. Ivory Overview

In this section, we give an illustrative overview of Ivory. An Ivory program is a Haskell program producing a collection of Ivory modules, each module containing type and procedure definitions.

Ivory is a staged language: the Haskell program compiles Ivory modules to produce an AST which is then passed to one or more back-ends. Thus, an executable is produced from an Ivory program by compiling and running the Haskell code to produce a C program, which is then compiled with a C compiler. Alternately, checking of pre- and post-conditions is performed by running the Haskell program in conjunction with the verification back-end.

In the following, we introduce both the types and values of Ivory programs but postpone most discussion of the types to Section 3. We focus on core aspects of the language in this introduction and throughout the paper. Ivory contains a large number of operators and standard libraries we elide. Examples include serialization, safe type casts, nullable pointers (for inter-operation with legacy C), function pointers, and bit operators.

2.1 Ivory Statements

Ivory statements are terms in the `Ivory` monad. This monad provides fresh variables, along with constructors for Ivory statements. Unlike C, Ivory expressions must be pure, so side-effecting operations take place at the statement level, in the context of the monad. This ensures a defined order for effects, eliminating large classes of unintuitive and undefined behaviors.

Memory in Ivory is manipulated via non-nullable references [13]. References are read and written using the `deref` and `store` statements, respectively. For example, the following Haskell function takes a reference to a signed 32-bit integer value and constructs a program fragment which increments the value of the reference.

```
incr_ref :: Ref s (Stored Sint32) -> Ivory eff ()
incr_ref r = do
    v <- deref r
    store r (v + 1)
```

A reference in Ivory may refer either to a global object, allocated at compile time, or a *local* object, allocated dynamically. Dynamic objects are created in ephemeral regions associated with the scope of the containing procedure; operationally, local objects are allocated on the stack, so regions are implicitly freed on procedure return. Ivory reference types are indexed by region variables, the parameter `s` seen in the type signatures above. Along with type variable scoping, these region annotations on references ensure that references do not escape the context in which they were allocated.

The definition `incr_ref` is not a complete Ivory procedure. Rather, it can be thought of as a template parameterised by a reference. Ivory procedures must be explicitly defined and exported through procedure definitions, such as

```
incr_def :: Def ('[Ref s (Stored Sint32)] :-> Sint32)
incr_def = proc "incr_def" $ \r -> body $ do
  incr_ref r
  v <- deref r
  ret v
```

The procedure `incr_def` calls the Haskell function `incr_ref` above. The `ret` statement returns a value from the Ivory procedure (we use `ret` rather than `return` to avoid conflicting with Haskell's `return` operator).

The Ivory monad tracks effects (the `eff` type parameter); see Section 3.1. One of these effects is the current allocation region: the allocation function `local` returns a reference that is tied to that region. For example, the following constructs a zero initialized reference to an integer; the `ival` constructs an initializer from a value:

```
make_zero :: (GetAlloc eff ~ Scope s)
          => Ivory eff (Ref s (Stored Sint32))
make_zero = local (ival 0)
```

2.2 Data structures

Ivory provides C-style arrays and data structures. Arrays types are parameterised by their size, the type system ensures that array accesses are within bounds. Data structures are defined using a quasi-quoter to specify the field names and their types. Fields in data structures are accessed via the ~> operator which takes a reference to a struct and a field name, and returns a reference at the field's type. For example, the following code declares a structure type named `position`, allocates an initialized instance, and then shows some basic operations on elements in the structure.

```
[ivory|
struct position
  { latitude  :: Stored IFloat
  ; longitude :: Stored IFloat
  ; altitude  :: Stored Sint32
  }
|]
struct_ex = do
  s <- local (istruct [ latitude  .= ival 45.52
                      , longitude .= ival (-122.68)
                      , altitude  .= ival 1524 ])
  lat <- deref (s ~> latitude)
  lon <- deref (s ~> longitude)
  incr_ref (s ~> altitude)
```

2.3 Control structures

Ivory supports the usual control structures: the `ifte_` statement constructor takes a boolean argument and two statements, one for each branch of the if-then-else, while the pure ternary operator, ?, selects from two alternatives at the expression level.

```
abs :: Def('[Sint16] :-> Sint16)
abs = proc "abs" $ \v -> body $ do
  ifte_ (v <? 0)
    (ret (-1*v))
    (ret v)

abs2 :: Def('[Sint16] :-> Sint16)
abs2 = proc "abs2" $ \v -> body $ do
  ret $ (v <? 0) ? ((-1*v), v)
```

Ivory has two classes of iteration constructs: `forever` for non-terminating loops such as OS tasks, and loops with a constant bounds. The prototypical bounded loop in Ivory is the `arrayMap`, which iterates over the elements of an array. For example, the following procedure adds `x` to each element of the array `arr`, noting that `arr ! ix` returns a *reference* to the `ix`-th element of `arr`.

```
mapProc = proc "mapProc"
        $ \arr x -> body
        $ arrayMap
        $ \ix -> do
            v <- deref (arr ! (ix :: Ix 4))
            store (arr ! ix) (v + x :: Uint8)
```

Note that we do not need to pass `arr` to `arrayMap` to determine the correct bounds on the loop; rather, as we explain in Section 3.4, GHC can *infer* the bounds from the loop body!

2.4 Assertions

Ivory supports pre- and post-conditions, along with assertions. The Ivory compiler can emit run-time assertions to enforce these conditions, or the model checker back-end can be used to statically verify these properties hold.

```
predicates_ex :: Def('[ IFloat ] :-> IFloat)
predicates_ex = proc "predicates_ex" $
    \i -> requires (i >? 0)
        $ ensures (\r -> r >? 0)
        $ body
        $ do (assert (i /=? 0))
             ret (i + 1))
```

An `ensures` clause takes a function, such that when applied to the return value at any return point in the procedure, the predicate should hold.

3. Ivory Embedding

In this section, we describe the implementation of Ivory, focusing on embedding the Ivory type system in the GHC type system.

3.1 The Ivory Monad

Ivory statements have the type

```
Ivory (eff :: Effects) a
```

This type wraps a writer monad transformer over a state monad. The writer monad writes statements into the Ivory abstract syntax tree, and the state monad is used to generate fresh variable names.

Effects The `eff` type parameter is a phantom type that tracks effects at the type level. (These effects have no relation to the recent work on effects systems for monad transformers [23].) Currently, we track three classes of effects for Ivory statement blocks:

- *Returns*: does the code block contain a `ret` statement, and is the type of the returned value correct?
- *Breaks*: does the code block contain a `break` statement?
- *Allocates*: does the code block contain local memory allocation?

Intuitively, these effects matter because their safety depends on the context in which the monad is used. For example, a `ret` statement is safe when used within a procedure, to implement a function return. However, an Ivory code block can also be used to implement an operating system task that should never return. Similarly in Ivory, `break` statements are used to terminate execution of an enclosing loop. (The other valid use of `break` in C99 is to terminate execution in a `switch` block, but Ivory does not contain `switch`.) By tracking break effects, we can ensure that an Ivory block containing a `break` statement is not used outside of a loop. Finally, allocation effects are used to guarantee that a reference to locally-allocated memory is not returned by a procedure, which would result in undefined behavior; see Section 3.3 for details. Moreover, we can prohibit a code block from allocating memory simply by removing allocation effects from its type.

The Ivory effects system is implemented by a type-level tuple where each of the three effects correspond to a field of the tuple. Type equality constraints enforce that a particular effect is (or is not) allowed in a given function signature.

We use GHC's data kinds extension [34] to lift the following type declaration to a *kind* declaration.

```
data Effects = Effects ReturnEff BreakEff AllocEff
```

The individual effect types are implemented similarly, using GHC to derive a kind from the type definition. For example, the `BreakEff` type/kind describes whether a break statement is allowed in a block of statements.

```
data BreakEff = Break | NoBreak
```

Type families [32] are used to access and modify the types at each field of the tuple. For example, the `GetBreaks` family extracts the `BreakEff` field of an `Effects` tuple. [2]

```
type family   GetBreaks (effs :: Effects) :: BreakEff
type instance GetBreaks ('Effects r b a) = b
```

The `AllowBreak` and `ClearBreak` families turn the effect "on" or "off", respectively.

```
type family   AllowBreak (effs :: Effects) :: Effects
type instance AllowBreak ('Effects r b a) =
    'Effects r 'Break a

type family   ClearBreak (effs :: Effects) :: Effects
type instance ClearBreak ('Effects r b a) =
    'Effects r 'NoBreak a
```

With this machinery, we can now use a type equality constraint to enforce the particular effects in a context. For example, Ivory's `break` statement has the type

```
break :: (E.GetBreaks eff ~ E.Break) => Ivory eff ()
```

3.2 Types

Ivory uses two type classes to define its domain: `IvoryType` and `IvoryArea`. `IvoryType` classifies all types that make up valid Ivory programs. As Ivory programs build up the AST of the program they represent when they are run, this class describes the set of types that contain fragments of the Ivory AST. The `IvoryArea` class serves to ensure that primitive types that are stored in references also have an instance of `IvoryType`. Types that have `IvoryType` instances include signed and unsigned integers, the void type `()`, and references, while types that have an `IvoryArea` instance are limited to those that have kind `Area`, defined in Section 3.4. All types used in Ivory programs will have an `IvoryType` or `IvoryArea` instance.

The `IvoryVar` and `IvoryExpr` class further stratify Ivory types that have values. The `IvoryVar` class, which is a superclass of `IvoryExpr`, describes all types that can have an Ivory expression extracted from them, as well as be created from a fresh name. This roughly corresponds to types that can be used as an L-value in

[2] The GHC syntax is to precede a data kind type constructor with a tick (') to disambiguate it from the corresponding data constructor.

```
class IvoryType t
class IvoryType t => IvoryVar t where
  unwrapExpr :: t -> Expr
  wrapVar    :: Var -> t
class IvoryVar t => IvoryExpr t
  wrapExpr   :: Expr -> t

class    IvoryArea (area :: Area *)
instance IvoryType t => IvoryArea (Stored t)
instance IvoryArea ...
```

Figure 1. Classes used to define Ivory's domain

assignments as well as formal parameters. The `IvoryExpr` class includes types that can be constructed from full expressions, and corresponds to the set of types whose values can be used in the position of an R-value. It might be tempting to say that the functionality of the `IvoryVar` belongs in the `IvoryType` class. However, Ivory has a void type (`()`) so we do require this distinction to prevent void values from being created. Most types used in Ivory provide instances for all three classes, `IvoryType`, `IvoryVar`, and `IvoryExpr`, with only a few exceptions like `()` defining a subset. See Figure 1 for the relationship between these classes.

3.3 Memory Management

Ivory uses regions for memory management [13]. When data is allocated, a reference to the resulting data is returned, and tagged by the containing region via a type variable. Well-typed Ivory programs guarantee that references do not persist beyond the scope of their containing region. Regions in Ivory classify both global data and data allocated and freed on procedure entry/exit (the back-end relies on stack-based allocation in C). Corresponding to these two kinds of regions are the region tags that Ivory supports: `Global` which holds statically-allocated global data that is available for the lifetime of the program, and a local region unique to each procedure whose lifetime is tied to that of the procedure.

Data with `Global` scope is allocated through the use of the `area` top-level declaration, then converted to a reference through the use of the `addrOf` function. As the `area` function produces a top-level declaration, it also requires a symbol to use as the name of the allocated memory. Data allocated within a procedure is allocated through the use of the `local` function, and are tagged with the region of that function. Since procedure definition introduces a fresh region, with the constraint that no reference allocated in that region should show up in the return type of that procedure, Ivory avoids the introduction of dangling pointers. The embedding of this feature in Haskell will be described in more detail in Section 3.5.

Both forms of allocation take initializers, though `Global` allocation through `area` will default to zero-initialization if it is omitted. Initializers are functions that embed values into a structure that mirrors that of a memory area. As an example, the `example` value in Figure 2 defines an initializer for an array of three `Uint8` values. The types of the allocation functions, as well as a sample of the initializers available are given in Figure 2.

As noted in Section 3.1, allocation is tracked through an effect in the effect context of the Ivory monad. The result of this is that each call to the `local` allocation function produces references that are tied to that specific context. Conversely, if the current effect context has no allocation scope, there is no way to produce a new reference. As allocation that takes place at the top-level is implicitly in the `Global` region, there is no need to involve the Ivory monad.

Once a reference has been acquired, it may be stored to and read from in the context of the Ivory monad using the `store` and `deref` functions. The Ivory monad does not track effects for manipulating specific references, and instead allows reading and writing to any reference that is in scope, within the context of the Ivory monad.

```
data Area k = Array Nat (Area k)
            | CArray (Area k)
            | Struct Symbol
            | Stored k

store :: IvoryStore a
      => Ref s (Stored a) -> a -> Ivory eff ()
deref :: IvoryStore a
      => Ref s (Stored a) -> Ivory eff a

data Label (struct :: Symbol) (area :: Area *)
(~>) :: Ref s (Struct sym) -> Label sym a -> Ref s a
(!)  :: Ref s (Array n area) -> Ix n -> Ref s area

local  :: (GetAlloc eff ~ 'Scope s, IvoryArea area)
       => Init area -> Ivory eff (Ref s area)

data MemArea (area :: Area *)
area   :: (IvoryArea area)
       => Sym -> Maybe (Init area) -> MemArea area
addrOf :: (IvoryArea area)
       => MemArea area -> Ref Global area

data Init (area :: Area *)
izero  :: IvoryZero area => Init area
ival   :: IvoryType val  => val -> Init (Stored val)
iarray :: IvoryArea area
       => [Init area] -> Init (Array n area)

example :: Init (Array 3 (Stored Uint8))
example  = iarray (map ival [1,2,3])
```

Figure 2. Memory allocation, initialization, access functions, along with the Area kind.

3.4 Memory Areas

References are parameterized by both their containing region and an *area type* describing the layout of the referenced memory. We introduce area types through the `Area`[3] kind, and the four types that inhabit it (Figure 2). This typing of memory is heavily inspired by the work of Diatchki and Jones [13]. Ivory supports four kinds of areas that we explain below: arrays with statically known size, "C" arrays without statically known size (for communicating with external C functions), structs, and stored atomic values.

Stored values The simplest type of memory area is a single base type, lifted to the `Area` kind by the use of the `Stored` type constructor. For example, the area type of a `Sint32` would simply be `Stored Sint32`. The `store` and `deref` operators will only operate over references that point to `Stored` areas, mirroring the operations from [13], as this allows us to never deal directly with a value of type `Array`, or `Struct`; we only ever read and write references to values, never references to aggregate values.

As the `Stored` area-type allows the lifting of any star-kinded type to a memory area, we constrain the operations on references to restrict what is storable. This constraint is enforced via the `IvoryStore` class. While the `IvoryStore` constraint is used to rule out most types from being stored in a reference, it is worth noting that it is also used to prohibit the storing of other references. The reason for this restriction is twofold:

1. We allow the use of default initializers during allocation, but do not have a good way to say what parts of a structure are

[3] The type parameter on the `Area` kind is present so that when giving kind-signatures, we can fix the kind of stored-values as being star (`*`)-kinded. As Haskell currently lacks a construct for defining kinds without data, this parameterization is necessary, as kinds are specified with a syntax that is invalid where a type is expected. This technique was described by Magalhães [27].

```
arrayMap :: (Ix n -> Ivory (E.AllowBreak eff) a)
         -> Ivory eff a
arrayLen :: Num len => Ref s (Array n area) -> len
toCArray :: Ref s (Array n area)
         -> Ref s (CArray area)
```

Figure 3. Array support functions

required, thus potentially introducing a null reference when initializing structures that contain references.

2. As there is currently no connection between the region of a reference, and the region of any references it points to, it would be possible to persist a reference beyond its lifetime by storing it in a longer-lived reference.

As Ivory only supports static memory allocation, not being able to store references inside of other references has not been a restriction that seriously impacted program development.

Structs A reference that has an area-kind of type `Struct "x"` will point to memory whose layout corresponds to the definition of the struct with name “x”. Struct definitions are introduced through use of the ivory quasi-quoter [28]. For example, if a region of memory is typed using the following struct declaration, it would have type `Struct "a"`.

```
[ivory| struct a { field1 :: Stored Sint32
                 , field2 :: Struct "b"
                 } |]
```

Also introduced by the struct declaration are field labels. Field labels allow for indexing into a memory area, producing a reference to the value contained within the struct. For example, using the previous struct definition, the quasi-quoter introduces two labels, `field1` and `field2`, for accessing those fields given a reference to an “a” struct:

```
field1 :: Label "a" (Stored Sint32)
field2 :: Label "a" (Struct "b")
```

Using a struct label to select the field of a structure requires the use of the (~>) operator, which expects a reference to a structure as its first argument, and a compatible label as its second. The type of the (~>) operator is given in Figure 2. In the following example, the (~>) operator is used with a reference to an “a” struct, with the `field1` label, producing a new reference of type `Ref Global (Stored Sint32)`.

```
example :: Ref Global (Struct "a")
        -> Ref Global (Stored Sint32)
example ref = ref ~> field1
```

Operations for indexing are pure in Ivory, as they only manipulate a base pointer; the value of a reference is never dereferenced until an explicit use of the `deref` primitive, which is an effectful operation.

Arrays Arrays in Ivory take two type parameters: the length of the array as a type-level natural number, and the area type of its elements. For example, an array of 10 signed 32-bit integers would have the type `Array 10 (Stored Sint32)`. Indexing into arrays is accomplished through the use of the `(!)` operator, shown in Figure 2. Indexing an array does not dereference it, but returns a reference to the indexed cell.

An index into an array has the type `Ix`, which is parameterized by the size of the array that it is indexing into. The `Ix n` type will only hold values between zero and `n-1`, which allows us to avoid run-time array bounds checks [13]. One shortcoming of this approach is that the `(!)` operator will only accept indexes that are parameterized by the length of the array being indexed, while it would be useful to allow indexes that have a maximum value that is less than the length of the target array.

As array indexes are parameterized by the length of arrays they can index into, they become an interesting target for new combinators. In this vein, we introduce `arrayMap`, whose signature is shown in Figure 3. The intuition for the `arrayMap` function is that it invokes the function provided for all indexes that lie between 0 and $n - 1$. As the index argument given to the function is most often used with an array, type information propagates out from uses of the `(!)` operator, and it becomes unnecessary to give explicit bounds for the iteration. Additionally, as the size of the index is tied to the size of the array being indexed, it is unnecessary to provide an array as an argument to `arrayMap`: we rely on the use of the index to set the bounds of the loop. The implementation relies on type-level natural numbers being singleton types, with the ability to construct a value n inhabiting the type n.

For compatibility with C, we also introduce a type for arrays that are not parameterized by their length, `CArray`. There are no operations to work with references to `CArray`s in Ivory, as the assumption is that they will only ever be used when interacting with external C functions. As many C functions that consume arrays require both a pointer and a length, we also provide the `arrayLen` function, which allows the length of an Ivory array to be demoted to a value. When used in conjunction with `toCArray`, this function allows for fairly seamless integration with external C code.

3.5 Procedures

Ivory procedures differ from Haskell functions in that they behave as compiled procedures, not macros; Haskell functions that produce `Ivory` values will be expanded at compile time, while Ivory procedures will be translated into procedures in the target language. Procedures in Ivory inhabit the `Def` type which is parameterized by the signature of the function it names. Procedure signatures inhabit the `Proc` kind, which provides one type constructor: `:->`. The `:->` type constructor takes two arguments: the types of the argument list, and the return type of the whole procedure. The intent behind the use of the `:->` type is to suggest that all of the arguments to the left of the arrow must be provided before a result may be produced.

Definition Procedures are defined through the use of the `proc` function, which requires two arguments: a symbolic name for the generated procedure and its implementation. The implementation takes the form of a Haskell function that accepts Ivory value arguments, and produces a result in the Ivory monad. Again, viewing Haskell functions that produce values in the Ivory monad as macros, the `proc` function can be seen as operating at the meta-level, accepting a symbol name and a macro as its arguments, and producing a procedure with the given name, and the fully-applied macro as its body. Correct procedure definition is guarded by the `IvoryProcDef` class, shown in Figure 5, which constrains uses of the `proc` function.

The `IvoryProcDef` type class has two parameters: signature and implementation. This class relates the `Proc` type of the resulting Ivory procedure and the Haskell function given as the its implementation. There are only two instances for `IvoryProcDef`: the case where the argument list is empty, and the case where the argument list is extended by one argument, corresponding to the cases for the `'[]` and `(':)` type constructors. The latter case also requires that the argument added be an Ivory type that is inhabited by a use of the `IvoryVar` constraint. This constraint both ensures that the argument type is acceptable as an argument to an Ivory function, and allows the use of pseudo-higher order abstract syntax by calling the body at fresh variables.

Examining the functional dependencies for the `IvoryProcDef` class from Figure 5, we see that the implementation function (`impl`) determines the signature of the resulting procedure (`sig`).

```
f = proc "f" $ body $ do
  ref <- local (izero :: Init Sint32)
  ret ref
```

Figure 4. Attempted creation of a dangling pointer

The effect of this dependency in the context of the `proc` function is that the user will rarely need to write an accompanying `Def` signature for Ivory procedures they define; uses of the arguments to a procedure will often yield a monomorphic implementation function, which through the functional dependency will produce a monomorphic `Def` type.

The implementation function is required to produce a value of type `Body r`, which is simply an Ivory monadic action with its allocation context hidden, and return type exposed as the type variable `r`. The `Body` type serves two purposes: it removes the need to write an instance of `IvoryProcDef` that involves a rank-2 function, and it defines an extension point for modifying the body of the procedure. Pre- and post-conditions can be added to a procedure body by the use of the `requires` and `ensures` functions, respectively. Both functions allow arbitrary Ivory statements to be added, but disallow all effects. The result of this restriction is that memory can be read and validated, but control flow and allocation effects are prohibited.

The procedure body can be defined through the use of the `body` function, whose signature is shown in Figure 5, which lifts an Ivory computation that returns a result `r` and allocates data in a region `s` into a value of type `Body r`. As the allocation scope expected by the given Ivory computation is quantified over in a rank-2 context by the `body` function, it *cannot* appear in the type of the result, `r`. This prevents anything allocated within the implementation function from being returned, a source of dangling pointer bugs. The same technique was used by Launchbury and Peyton Jones [24] to prevent mutable state from leaking out of the context of the run function for the `ST` monad.

For example, the procedure `f` defined in Figure 4 will produce a type error, as it attempts to return a locally-allocated reference; references are parameterized by the scope they were allocated in, and as that scope variable is quantified over in the rank-2 context of the argument to the `body` function, that reference is prevented from showing up in the return type of the procedure, `r`.

Invocation Procedures are called through the use of the `call` function, which takes a `Def` as its first argument, using its signature to determine the arguments needed. The arguments needed are determined by the `IvoryCall` class, which uses the signature information to produce a continuation that requires parameters that match the type of the argument list from the signature of the `Def`. The `IvoryCall` class mirrors the structure of the `IvoryProcDef` instances structure, though it adds one additional parameter: `eff`. This additional parameter is required so that the containing effect context of the call can be connected to the result of the continuation generated by the instances of `IvoryCall`. For example, calling a procedure with type `Def ('[Sint32] :-> Sint32)` will produce a continuation of the type, `Sint32 -> Ivory eff Sint32`, where the `eff` parameter is inherited from the current environment.

3.6 Bit-Data

Introduction Low-level systems programming often requires extensive manipulation of binary data packed into multi-field integer values. For example, a hardware register may contain several single-bit flags along with multi-bit fields that may not be aligned to byte boundaries.

When programming in C, these bit values are typically accessed by defining a set of integer constants and using bit operations to shift and mask the correct bits into place with little to no type safety.

```
data Proc k = [k] :-> k

class IvoryProcDef (sig :: Proc *) impl | impl -> sig
instance IvoryProcDef ('[] :-> r) (Body r)
instance (IvoryProcDef (as :-> r) impl, IvoryVar a)
  => IvoryProcDef ((a ': as) :-> r) (a -> impl)

class IvoryCall eff (sig :: Proc *) impl
  | sig eff -> impl, impl -> eff
instance IvoryCall eff ([] :-> r) (Ivory eff r)
instance (IvoryExpr a, IvoryCall eff (as :-> r) impl)
  => IvoryCall eff ((a ': as) :-> r) (a -> impl)

body :: (forall s. Ivory (ProcEffects s r) ())
     -> Body r

data Def (sig :: Proc *)
proc :: IvoryProcDef sig impl
     => Sym -> impl -> Def sig

call :: IvoryCall sig eff impl => Def sig -> impl
```

Figure 5. Function definition support.

In support of high assurance low-level programming, Ivory's standard library contains a data definition language for these "bit data" types. Our system is a subset of the bit data implementation described in [14], which allows the programmer to define bit data as algebraic data types that can be nested and accessed in a type-safe manner.

Implementation Ivory's type system supports a set of unsigned integer types with specific bit sizes (8, 16, 32, and 64 bits), as in the C language. In order to support bit data of arbitrary width (up to the maximum supported length of 64 bits), we use a type family `BitRep n` to map an integer size in bits to the smallest concrete Ivory type that can hold an integer of that size:

```
type family BitRep (n :: Nat) :: *
type instance BitRep 1 = Uint8
type instance BitRep 2 = Uint8
{- ... -}
type instance BitRep 64 = Uint64
```

Ivory adds additional type safety to arbitrary width integers by wrapping these values in an opaque type `Bits n`. Haskell's module system is used to hide the raw constructor for these values, only permitting valid values to be created via the use of smart constructors:

```
newtype Bits (n :: Nat) = Bits (BitRep n)

zeroBits :: Bits n
repToBits :: BitRep n -> Bits n
bitsToRep :: Bits n -> BitRep n
```

Smart constructors that are partial due to narrowing, such as `repToBits`, automatically mask out any bits that are out of range. It is also possible to define runtime-checked versions of these functions that treat such "junk values" as an error.

To support combining multiple bit fields into a single value, we generalize the "bit data" concept with a type class `BitData` that captures the interface of a value that may be converted to and from its representation as raw bits:

```
class BitData a where
  type BitType a :: *
  toBits :: a -> BitType a
  fromBits :: BitType a -> a
```

The `Bits n` type is a trivial instance of this type class:

```
instance BitData (Bits n) where
  type BitType (Bits n) = Bits n
  toBits   = id
  fromBits = id
```

Type Definition The language for defining bit data types mirrors Haskell's syntax for defining data types. Each bit data type contains one or more constructors, each of which may have zero or more data fields.

For example, consider a control register for a communication device with a 2-bit field used to specify the baud rate. The user can provide a quasi-quoter-defined bit data type `BaudRate` that enumerates the legal 2-bit values:

```
[ivory|
  bitdata BaudRate :: Bits 2
    = baud_9600   as 0b00
    | baud_19200  as 0b01
    | baud_38400  as 0b10
    -- bit pattern 0b11 is invalid
|]
```

Using Template Haskell, the definition generates an opaque Haskell type `BaudRate`, implements an instance of the `BitData` type class, and defines zero-argument constructors for each value:

```
newtype BaudRate = {- ... -}
instance BitData BaudRate where {- ... -}
baud_9600, baud_19200, baud_38400 :: BaudRate
```

Bit data types can be arbitrarily nested to define more complex types. To continue the example, we define the entire control register consisting of enable bits for a transmitter and receiver, along with the baud rate:

```
[ivory|
  bitdata CtrlReg :: Bits 8 = ctrl_reg
    { ctrl_tx_enable  :: Bit
    , ctrl_rx_enable  :: Bit
    , ctrl_baud_rate  :: BaudRate
    } as 0b0000 # ctrl_tx_enable # ctrl_rx_enable
                # ctrl_baud_rate |]
```

This definition of `CtrlReg` defines a single constructor for building a `CtrlReg` value out of its constituent fields:

```
ctrl_reg :: Bit -> Bit -> BaudRate -> CtrlReg
```

The field definitions define accessors for the fields of a `CtrlReg`. Because the type of these accessors contains both the type of the containing bit data and the field being accessed, Haskell's type system prevents errors such as accessing a bit in the wrong register:

```
ctrl_rx_enable :: BitDataField CtrlReg Bit
```

Usage In a typical low-level application, these fields are accessed with a read-modify-write cycle which is supported efficiently by the `withBits` function and Haskell's `do` notation:

```
init_ctrl_reg = proc "init_ctrl_reg" $ body $ do
  reg <- call read_ctrl_reg
  call_ write_ctrl_reg $ withBits reg $ do
    setBit   ctrl_tx_enable
    setBit   ctrl_rx_enable
    setField ctrl_baud_rate baud_9600
```

3.7 Module System

The Ivory module system packages up the collection of procedures, data declarations, and dependencies to be passed to a back-end, such as the C code generator. The module system is implemented as a writer monad that produces a list of abstract syntax values that are processed by the various back-ends.

Because our primary backend is C, Ivory modules respect some of the conventions of C modules in which header files are used to specify shared declarations. For example, declarations can be declared as either public or private, and modules can depend on other modules.

At best, forgetting to include an Ivory dependency is an inconvenience. This inconvenience can be substantial in the case that an inter-module dependency is omitted, which still permits the Ivory program to type-check. If the dependency missing is a C function implementation, for example, C code is generated and compiles, but fails during link time. The error does not result in a safety violation, but in large projects, such as the SMACCMPilot autopilot written in Ivory [19], the error can take several minutes to detect.

```
foo :: Def ('[Sint32] :-> Sint32)
foo = proc "foo" $ \_ -> body $ ret 0

fooInternal :: Def ('[Ref s (Stored Sint32)] :-> Sint32)
fooInternal = proc "foo" $ \ref -> body $ do
 x <- deref ref
 ret x

main :: Def ('[] :-> Sint32)
main = proc "main" $ body $ do
 x <- call foo 0
 ret x

cmodule :: Module
cmodule = package "Evil" $ do
 incl fooInternal
 incl main
```

Figure 6. Unsafe module usage

Worse, a naive implementation of the module system can lead to safety violations. For example, consider the program in Figure 6. Two procedures, `foo` and `fooInternal` are defined but given the same string used as the procedure name, used in the generated C. The Ivory program is type-correct and safe, but by passing `fooInternal` into the module, it is compiled rather than `foo`. And given the C99 specification, the program compiles without warnings or errors, since `0` can be implicitly cast to a pointer to a signed 32-bit integer.[4] The result is a null-pointer dereference.

To ensure this does not happen, a simple type-check pass over the Ivory AST is performed before compilation. The type-check pass ensures that the prototype of a function matches the types of the arguments.

4. Ivory Semantics

In the previous section, we described our embedding of Ivory into the GHC type system and made the claim that this guarantees memory safety. We modeled a simplified version of the Ivory language inside Isabelle/HOL[30], henceforth *Core Ivory*, to support this claim. In this section we present a semantics based upon the Isabelle/HOL development, and outline the proof of type safety.

Developing this model provides a number of benefits for a modest investment—we developed the model in under a person month, albeit one of the authors has significant experience with Isabelle. In addition to the basic benefits formalisation provides, we can experiment with extensions to Ivory.

In one such experiment, we extended the model to allow references in the heap, a feature we avoided in the development of Ivory due to soundness concerns. While a simple extension to the syntax and semantics of Ivory, the effort involved in extending the soundness proofs was almost as much as developing the initial model.

Due to space constraints we discuss only those particulars of Core Ivory which differ significantly from a standard imperative language; see the supplemental material for more details.

4.1 Syntax

The syntax for Core Ivory is given in Figure 7. Core Ivory is based upon a typical typed imperative language with function calls,

[4] In practice, the C we generate does contain a warning, since it contains additional type annotations.

pure expressions

$$
\begin{array}{lll}
e & ::= & 0 \mid 1 \mid \ldots \mid \texttt{true} \mid \texttt{false} \mid () \mid x \mid e_1 \; op \; e_2 \\
\multicolumn{3}{l}{\textit{impure expressions}} \\
i & ::= & \texttt{pure}(e) \mid \texttt{alloc}(e) \mid \texttt{read}(e) \mid \texttt{write}(e_1, e_2) \\
\multicolumn{3}{l}{\textit{statements}} \\
s & ::= & \texttt{skip} \mid \texttt{return}(e) \mid s_1; s_2 \\
 & \mid & \texttt{if}(e) \; \texttt{then} \; s_1 \; \texttt{else} \; s_2 \\
 & \mid & \texttt{for}(x = e_1; e_2; e_3)\{s\} \\
 & \mid & \texttt{let} \; x = i \; \texttt{in} \; s \\
 & \mid & \texttt{let} \; x = f(e_1, \ldots, e_n) \; \texttt{in} \; s \\
\multicolumn{3}{l}{\textit{values}} \\
v & ::= & 0 \mid 1 \mid \ldots \mid \texttt{true} \mid \texttt{false} \mid () \\
w & ::= & \texttt{stored}(v) \mid \texttt{ref}(r, n) \\
\multicolumn{3}{l}{\textit{Stores}} \\
E & \in & x \to w \\
\multicolumn{3}{l}{\textit{regions and heaps}} \\
R & \in & \mathbb{N} \to w \\
H & ::= & H, R \mid \emptyset \\
\multicolumn{3}{l}{\textit{stacks}} \\
F & ::= & \texttt{rframe}(x, E, s) \mid \texttt{sframe}(E, s) \\
S & ::= & F, S \mid \emptyset \\
\multicolumn{3}{l}{\textit{configurations}} \\
C & \in & H \times S \times E \times s \mid \texttt{finished}(v) \\
\multicolumn{3}{l}{\textit{types}} \\
\rho & \in & \textit{region variables} \\
\alpha & ::= & \texttt{nat} \mid \texttt{bool} \mid \texttt{unit} \\
\tau & ::= & \texttt{storedt}(\alpha) \mid \texttt{reft}(\rho, \alpha) \\
\multicolumn{3}{l}{\textit{procedure definitions}} \\
P & ::= & \texttt{proc} \; f(\tau_1 \; x_1, \ldots, \tau_n \; x_n) : \tau \; \{s\}
\end{array}
$$

Figure 7. Concrete syntax of Core Ivory

references, and memory allocation (but not memory deallocation). Core Ivory attempts to stay faithful to Ivory wherever possible, and so variables are let-bound with forms for binding the result of expression evaluation and function calls. Furthermore, Core Ivory expressions are stratified into *pure* and *impure*, the latter allowing operations on the heap: allocation, reading, and writing references.

Ivory uses regions to manage memory. Thus, the heap is modeled as a list of *regions*, each region a finite map from *offsets*, modeled as natural numbers, to *stored values*; Ivory does not allow references in heap allocated values, and so a stored value is any value which is not a reference. A *reference* contains both a *region index* into the list of regions, and an offset with the region. To simplify the presentation, we will use $H(r, n)$ to denote the value at offset n in the rth region of H, and similarly with updates.

As with values, types classifying values in Core Ivory are stratified into storable and reference types; a reference type $\texttt{reft}(\rho, \alpha)$ is a reference to an object of type α in region ρ, where α is not a reference.

4.2 Operational Semantics

Core Ivory's semantics are modeled as an abstract machine over configurations. The judgement

$$\models C \longmapsto C'$$

states that configuration C transitions to configuration C'. A configuration consists of a heap, a stack, a store, and the current statement. The stack contains continuations for both function calls and statement sequences while the store maps variables to values. The semantics of sequencing is slightly non-standard as variables are let-bound rather than assigned, and so statement sequencing preserves the store across execution of the first statement.

For example, the semantics of dereferencing is given by the rule

$$\frac{[\![e]\!]E = \texttt{ref}(r, n) \qquad (r, n) \in \texttt{dom}(H)}{\models (H; S; E); \texttt{let} \; x = \texttt{read}(e) \; \texttt{in} \; s \longmapsto (H; S; E[x \mapsto w]); s}$$

where $[\![e]\!]E$ evaluates the pure expression e under the store E. The premise $(r, n) \in \texttt{dom}(H)$ requires the existence of the region and offset pointed to by the reference resulting from evaluating e.

Operationally, the heap is extended on a function call, an empty region being added to the end of the list, and shrunk on function return, removing the last region. Allocating an object extends the current (last) region.

A configuration is stuck if there is no available transition. For instance, an attempted heap access or update where the region index does not exist or at an offset which has not been allocated will result in a stuck configuration. In particular, accessing a region after it has been removed will result in a stuck state.

4.3 Typing Ivory

Core Ivory's typing judgements extend standard statement typing judgements with a current region variable. The typing judgement

$$\Gamma; \rho \vdash_s s : \tau$$

holds when the statement s is well-formed under the store environment Γ, current region ρ, with any return statements returning values of type τ.

The region variable ρ represents the current region and is used when checking memory allocation. The typing rule for allocation is then

$$\frac{\Gamma \vdash_e e : \alpha \qquad \Gamma[x \mapsto \texttt{reft}(\rho, \alpha)]; \rho \vdash_s s : \tau}{\Gamma; \rho \vdash_s \texttt{let} \; x = \texttt{alloc}(e) \; \texttt{in} \; s : \tau}$$

where the body of the let statement is checked under the additional assumption that the variable x has reference type $\texttt{reft}(\rho, \alpha)$, noting the region variable on the reference type comes from the current region variable. The judgement $\Gamma \vdash_e e : \alpha$ holds when pure expression e has type α.

We fix the set of procedures as $\texttt{Procs}$. The typing rule for procedure bodies

$$\frac{\begin{array}{c} \forall \; \texttt{proc} \; f(\tau_1 \; x_1, \ldots, \tau_n \; x_n) : \tau \; \{\textit{body}\} \in \texttt{Procs} \\ \rho \text{ fresh} \qquad \texttt{frees}(\tau) \subseteq \texttt{frees}(\tau_1) \cup \ldots \cup \texttt{frees}(\tau_n) \\ [x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n]; \rho \vdash_s s : \tau \end{array}}{\vdash \texttt{Procs}}$$

ensures that this region variable is fresh; this constraint, together with the constraint that region variables in the procedure's return type must occur in an argument type, ensures that references cannot escape the scope in which they were allocated. These constraints are then fundamental to the type safety of Core Ivory programs.

4.4 Type Safety

We prove type safety, that is, well-typed programs do not get stuck, by proving the usual progress and preservation lemmas. As is common with type safety for imperative languages, we define auxiliary well-formedness invariants on configurations. The progress lemma then states that well-formed configurations are not stuck, and preservation states that well-formed configurations transition to well-formed configurations.

Theorem 1 (Type Safety) *Given* $\vdash \texttt{Procs}$ *and*

$$\texttt{proc main}() : \texttt{nat} \; \{s\} \in \texttt{Procs}$$

there either exists some number of steps n and value $v \in \mathbb{N}$ such that

$$\models (\emptyset; \emptyset; \emptyset); \texttt{let} \; x = \texttt{main}() \; \texttt{in return}(x) \longmapsto^n \texttt{finished}(v)$$

or, for all n there is a well-formed configuration C such that

$$\models (\emptyset; \emptyset; \emptyset);\ \mathit{let}\ x = \mathit{main}()\ \mathit{in}\ \mathit{return}(x) \longmapsto^n C$$

Informally, a well-formed program, when called via the `main` procedure, will either terminate in a finite number of steps, or will diverge through well-formed configurations.

The well-formedness invariants are typical, and are based upon a well-formed value judgement. For our purposes, the interesting rule here is for references

$$\frac{\Delta(\rho) = r \qquad \Theta(r, n) = \alpha}{\Theta; \Delta \vdash \mathtt{ref}(r, n) : \mathtt{reft}(\rho, \alpha)}$$

which links reference types to reference values through the region environment Δ, mapping region variables to region indices, and heap type Θ, mapping region indices and offsets to types. Well-formed heaps and stores then follow point-wise from this judgement. Well-formedness of stacks follows, ensuring that each continuation on the stack is well-formed.

A well-formed configuration, in addition to the well-formedness of the heap, stack, store, and current statement, constrains the region environment (Δ). In particular, the variable representing the current region must be mapped to the length of the current heap, every region index in the range of Δ must be below this length, and the type variables occurring in the store environment Γ must be mapped by Δ.

The progress lemma follows from well-formedness. The preservation proof, as usual, is the trickier of the two proofs. In particular, the case for return involves showing that the various configuration members are well-formed under a heap where the last region has been removed from the heap. This involves showing that references are well-formed under this smaller heap which follows from the stack and region environment well-formedness invariants. The case for function calls is also involved, although this is primarily due to the instantiation of the type variables in the type of the called function.

4.5 Discussion

The type safety proofs are greatly simplified by the restriction of heap values to non-references: it is trivially true, for example, that the heap is well-formed after a return as the well-formedness of non-reference values does not depend on the type of other heap elements.

We extended the Isabelle model to remove the stratification of values, allowing references to appear in the heap. Syntactically, this simplifies the language, at the expense of more complicated proofs. For example, we extend the well-formedness of heaps to require that heaps are downward closed with respect to region references: that is, references in the heap refer only to regions up to and including the containing region. This work extended the proof development from approximately 2500 lines of proof to 3300 lines of proof, and took approximately 2 person weeks to finish.

Developing this model uncovered a bug in the Ivory embedding into Haskell, namely that the Haskell erroneously allowed functions to end without encountering a return statement. Thus, a program could claim to return, say, a valid index but in reality return an out-of-bounds index, breaking memory safety. In the model this check is part of the type checking rules (although elided for clarity in this paper) but is implemented as an explicit type-check over the Ivory AST in the Haskell implementation.

The existence of the formal model is not a guarantee that the Ivory implementation is sound, however: there is no formal link between the implementation and the model. For example, we discovered a number of bugs due to the way in which references are initialized: these bugs are impossible in the model, but allowed in the implementation.

In addition, Core Ivory covers a subset of the Ivory language, currently missing data structures and arrays. We are working to reduce this gap; furthermore, in future, we plan to investigate making the core of the implementation more closely resemble Core Ivory. Doing so would have the additional benefit of allowing us to verify the correctness of operations performed over the core AST.

We plan on investigating first-class regions as a further extension to the model. This feature would allow programmers to name and pass around allocation contexts, allowing helper functions to, for example, allocate and return objects in a parent functions region. While we are confident that this extension is sound, *proving* it sound allows a much greater confidence in the implementation: memory deallocation is notoriously easy to get wrong, so having a formal proof of soundness would be greatly comforting.

5. Ivory Testing and Verification

Ivory contains built-in tools to support high-assurance software development. These tools include a correctness condition generator, a SMT-based symbolic simulator, a theorem-prover translator, and a QuickCheck engine for randomized testing [12]. We describe each of them briefly below.

5.1 Correctness Conditions

Some correctness properties such as arithmetic underflow and overflow cannot be embedded in the Haskell type system. For this class of properties the translation into the Ivory AST adds instrumentation containing appropriate assertions. For example, from the function

```
add_ex :: Def ('[Sint32, Sint32] :-> Sint32)
add_ex = proc "add_ex" $ \x y -> body $
  ret (x + y)
```

the following function is generated

```
int32_t add_ex(int32_t var0, int32_t var1)
{
    bool i_ovf0 = add_ovf_i32(var0, var1);
    COMPILER_ASSERTS(i_ovf0);
    return (int32_t) (var0 + var1);
}
```

Note the addition of an overflow check, performed by the function `add_ovf_i32`, before the expression is evaluated. The result of this check is handled by the macro `COMPILER_ASSERTS`; the response when a property is violated is platform-dependent, and may include logging, do-nothing, running a recovery procedure, and so forth.

Ivory inserts correctness condition checks for the following conditions, as requested by the user:

- no arithmetic underflow and overflow,
- no division-by-zero,
- no bit-shifts are greater than or equal to the value's width, (bit-shifts on signed integers are prevented statically by the type system)
- no floating-point operations result in `inf` or `NaN` values.

These correctness conditions are added as assertions by the Ivory front-end. While the implementation is straightforward, we note that care must be taken to not over-constrain the emitted program: Ivory contains the short-cutting expressions of conjunctions, disjunctions, and conditionals, analogs of C's `&&`, `||`, and `_ ? _:_`, respectively. For these expressions, the generated assertions must contain as a precondition that short-cutting has not occurred so as not to be overly pessimistic. For example, for the expression

```
x != 0 ? 3/x : 0
```

the (tautological) correctness condition

```
(x == 0) || (x != 0)
```

is generated.

In addition, Ivory's EDSL implementation encourages the use of macros which can result in large expressions. In this case, repeated sub-expressions will result in repeated assertions. Standard common subexpression elimination greatly reduces the number of instrumented assertions.

5.2 Discharging Correctness Conditions

Generated correctness conditions as well as general user assertions can be discharged via testing or using model-checking or theorem-proving. We describe Ivory's tooling for these approaches below.

Symbolic Simulation Ivory contains a prototype symbolic simulator built over CVC4 [10] for verifying programs. Given a partially annotated program, the simulator attempts to verify any inline assertions and postconditions. We have used the symbolic simulator to analyze various Ivory programs, having found an off-by-one bug in a ring buffer and verified the correctness of a safety state-machine in SMACCMPilot.

The symbolic simulator abstracts various domains. Floating point types are abstracted as reals. However, fixed-width values are modeled precisely as are arrays.

During analysis, inter-procedural calls are inlined or abstracted as directed by the user. If they are abstracted, then the callee's precondition is added as a verification condition at the call site. The callee's postcondition is added to the set of invariants following the call. Ivory allows procedures imported from C to be summarized with pre- and postconditions.

Ivory programs are amenable to formal analysis as they guarantee the absence of memory-safety errors, and contain no pointer arithmetic or concurrency. In addition, Ivory programs typically use unbounded iteration only at the top-level, thus all loops of interest have statically known bounds and can be unrolled.

Theorem Proving Eakman et al. [15] implemented a theorem-prover back-end for Ivory that targets ACL2 [6]. The theorem-prover performs inter-procedural analysis, abstracting procedure calls by their contracts, as with the symbolic simulator. The ACL2 backend, being interactive, has the potential to support more complex assertions than the symbolic simulator, at the expense of user-directed proofs. Eakman et al. provide examples of the use of the theorem-proving back-end.

Property-Based Testing Finally, Ivory contains a QuickCheck [12] like property-based testing framework. The framework tests procedures by randomly generating values both for their formal arguments as well as for global values. These values are constrained to satisfy any pre-conditions.

While Ivory has an interpreter, the testing framework uses the C backend to compile the tests. This ensures that the tested code is that which will be run in the final compiled artifact, eliminating any potential bugs introduced by the semantic gap between interpreter and the compiled binary.

6. A Safe-C EDSL

In this section, we discuss the following issues we encountered when implementing an EDSL targeted toward constructing realistic embedded systems, notably integrating a C-like concrete syntax into the EDSL, error-reporting, and uses of macro programming specifically tailored to embedded system programming.

Concrete Syntax A benefit of the EDSL approach is that it relieves the language developer from having to define and implement front-end syntax. However, that also generally means that only users of the host language will be attracted to using the EDSL. We want C/C++ developers to use Ivory!

```
void mapProc(*uint8_t[4] arr, uint8_t x) {
  map ix {
    let v = arr@ix;
    *v = *v + x;
  }
}
```

Figure 8. Concrete Syntax for Ivory

We developed a concrete C-like syntax for Ivory as a quasi-quoter, which facilitated use by Boeing as mentioned in Section 1. In the above we presented specific uses of quasi-quotation in Ivory to define a concrete syntax for structs (Section 3.4) and bit-data (Section 3.6). In Figure 8 we show an example quasi-quoted Ivory procedure, equivalent to the that with the same name in Section 2.3. The syntax is similar to a imperative languages like C or Python, with some variations. For example, array types are not treated specially like in C; the type (e.g., `uint_8[4]`) merely describe a memory area, and a reference (`*`) must be provided to use it. Instead of for-loops, we have a `map` operator that calculates the number of iterations from the array being looped over. Let bindings are introduced. Finally, the `@` operator indexes into an array without dereferencing.

A quasi-quoted Ivory program is guaranteed to be type-safe, since the generated Haskell program is type-checked, an important feature of the quasi-quoted language is that it automatically generates the appropriate type signatures for Ivory programs, relieving the programmer from doing so. The quasi-quoter also generates Ivory modules automatically and guarantees that procedure names match their Haskell identifiers, obviating the problems discussed in Section 3.7. The quasi-quoter supports anti-quotation, so that Haskell can still be used as a macro language. All of Boeing's development in Ivory is via the quasi-quoter.

Implementing a quasi-quoter means that we have defined a lexer and parser for the concrete syntax. Because Ivory is deeply embedded in Haskell, there is a concrete data type (i.e., the abstract syntax tree (AST)) over which optimizations and back-ends are implemented. The distance from Ivory as an EDSL and a stand-alone compiler is surprisingly small, essentially requiring a type-checker and a front-end targeting the AST directly. In this manner, we are able to "grow" a compiler, from an EDSL to a stand-alone system.

Haskell as Macro Language C programmers are accustomed to using the C Pre-Processor (CPP) as a limited macro and metaprogramming system. Embedding Ivory within Haskell allows users the full power of Haskell as a type-safe metaprogramming environment for Ivory. We have used this feature in a variety of ways, specifically beneficial to embedded development.

- In embedded systems, a peripheral device's clock may be derived from a platform-wide clock by a programmable clock divider. However, the platform clock may be configurable. Configuring, say, a serial port to operate at 115200bps, may require a complex search over the space of clock divider and other configuration settings based on the current platform clock. These settings will be the same on every run for a given application, so we can precompute them in Haskell.
- We frequently use Haskell type-classes to provide function overloading for Ivory. For example, the ivory-serialize library provides a "Packable" typeclass capturing the code needed to serialize or deserialize user-specified types. This style of overloading is resolved during Haskell run-time, so the generated C is monomorphized.

- Mathematical metaprogramming has been a particularly powerful tool when combined with Ivory. Edward Kmett's automatic differentiation [1] and linear algebra [5] packages, which depend primarily on the Haskell Prelude's numeric typeclasses, operate transparently over Ivory expressions—despite having been designed independently. This allowed us to build a standalone package, "estimator" [2], which automatically derives an Extended Kalman Filter from a concise model written in pure Haskell. Like linear and ad, the estimator package is independent of Ivory, but when evaluated over an Ivory floating-point type, the result is a C implementation of the derived Kalman filter.

Error Reporting Dynamically reporting errors in an EDSL is made difficult by the lack of source locations. Consider the Ivory expression `x / y`, which induces a runtime assertion `y != 0`, inserted into the generated back-end (e.g., C code). If the assertion fails, we would like to include a *Haskell* source location in the error message to direct the programmer to the source of the error. The question is then: how are we to obtain the source location in the first place? Ivory programs are comprised of Haskell expressions and Haskell is pure–functions cannot depend on their call-site–so we must look outside the language proper to obtain source locations.

A common approach for writing location-aware functions in Haskell, epitomized by the `file-location` package [3], is to use a Template Haskell splice to query the compiler for the current source location and insert it into the AST. This approach did not appeal to us as Template Haskell incurs considerable syntactic overhead and is incompatible with the concrete syntax described above.

Instead, we opted to write a compiler plugin that rewrites GHC's intermediate representation to add the source locations. To do so, we extended the `Ivory` statement type with a new `Location` constructor that contains a location in the Haskell source, essentially a special type of comment. The plugin then extracts the source locations from GHC and wraps all actions in the `Ivory` monad with a `withLocation` function that emits a `Location` statement before executing the wrapped action.

The approach limits the location granularity to lines rather than columns, but it requires only modest changes to the Ivory compiler and–importantly–no changes to Ivory code. As the plugin itself requires little knowledge of Ivory, we have abstracted it out into a separate package [4] that can be reused by other projects. Furthermore, we have since submitted a lightweight extension to GHC that allows functions to request their call-site by taking a special implicit parameter [25] as an argument, which should become available in the 7.12 release.

7. Related Work

The general idea of safe C languages is not new; our main contribution is embedding a type system into GHC as well as our support of verification tools.

Pioneering work in the area is the Cyclone language and compiler [20]. Cyclone is a dialect of C. Cyclone is less restrictive than Ivory, relying on both static analysis and runtime checks to enforce memory safety. Cyclone provides regions for dynamic memory allocation; garbage collection is optional. Cyclone programs are typically slightly larger than their C equivalents and mostly syntactically the same. In contrast to Ivory, Cyclone does not provide macro-programming facilities (beyond the C preprocessor), nor does it interface to verification and testing tools. Unfortunately, Cyclone is not actively maintained.

Bit-data and memory areas in Ivory borrow heavily from Diatchki et al.'s previous work [13, 14]. Indeed, one can consider the present work as demonstrating the feasibility of embedding this language into Haskell and GHC types. BitC is another deprecated research language that explored a similar design space [33].

Spark/Ada is a mature language for high-assurance embedded programming, with a contract language and verification tools to prove invariants [9]. To support verification, the language is very restrictive: in particular, there are no references in the language.

Rust is an actively-developed safe C language, originating from Mozilla [29]. Rust has a powerful affine type system[5] that enforces the safe use of heap-based data structures, reference counting garbage collection as a library, and hygenic macros. The language has a property-based testing framework, but no mature support for verification, or even static checks for undefined behavior (e.g., division by a constant zero expression).

Other safe C EDSLs exist such as Atom, a language for lock-free embedded programs [18]; Copilot, a stream-oriented synchronous language [31]; SBV, a Haskell-based SMT symbolic simulator with a C code generator [16]; and Feldspar, a language specialized for high-level and efficient specifications of digital signal processing [8]. Compared to these languages, Ivory is more focused on the kinds of C in low-level code such as device drivers, with bit-data and memory area manipulation.

8. Conclusions and Future Work

In this paper, we have described a full-featured EDSL for high-assurance embedded systems programming. Ivory's type system ensures safe C development, and being an EDSL, it allows programmers the flexibility to create high-level constructs in a type-safe fashion. This feature is used extensively in the development of the SMACCMPilot project, allowing common idioms to be abstracted. We have demonstrated the feasibility of developing large embedded systems in Ivory ourselves, and there is a growing user community. Some of our experiences with programming large embedded systems in Ivory are reported in [19].

As Ivory's type system is embedded in GHC's, the properties that Ivory's type system can encode is limited to what can be expressed in GHC: for instance, while procedures are guaranteed to be consistent in their return type, that they use the return statement must be checked during a separate phase. In practice, this limitation results in the discovery of errors later in the compilation pipeline than would be the case in a standalone compiler. Conversely, Ivory can take advantage of new developments in GHC's type system. For instance, there are plans to integrate SMT solving into GHC's constraint solver, which would enable more expressive array operations in the Ivory core language, as well as enabling a richer set of derived operations.

Use of Ivory has exposed a number of avenues for future work. As mentioned in Section 4, we are investigating the addition of nested references. We also plan to investigate decoupling regions from function bodies, thus giving finer-grained control over memory lifetimes. Finally, we are considering making regions first-class, allowing allocation to take place in a parent region. On the verification side, we are considering developing a weakest-precondition style verification tool for Ivory programs, and extending the assertion language with separation-logic predicates.

Acknowledgments

We thank Kathleen Fisher, Iavor Diatchki, Mike Whalen, Gerwin Klein, Ben Lippmeier, and the members of the SMACCM team from Rockwell Collins, Boeing, NICTA for comments on this paper as well as stimulating discussions in the development of Ivory. This work is supported by DARPA under contract no. FA8750-12-9-0169. Opinions expressed herein are our own.

[5] An affine type system prevents pointer aliasing errors.

References

[1] ad. Website `http://hackage.haskell.org/package/ad`. Retrieved Feb. 2015.

[2] estimator. Website `http://hackage.haskell.org/package/estimator`. Retrieved Feb. 2015.

[3] file-location. Website `http://hackage.haskell.org/package/file-location`. Retrieved Feb. 2015.

[4] ghc-srcspan-plugin. Website `http://hackage.haskell.org/package/ghc-srcspan-plugin`. Retrieved Feb. 2015.

[5] linear. Website `http://hackage.haskell.org/package/linear`. Retrieved Feb. 2015.

[6] *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[7] Heartbleed. `http://heartbleed.com/`, February 2015.

[8] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The design and implementation of Feldspar - an embedded language for digital signal processing. In *Implementation and Application of Functional Languages*, volume 6647 of *LNCS*, pages 121–136. Springer, 2011.

[9] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[10] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 171–177, 2011.

[11] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security*, 2011.

[12] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM, 2000.

[13] I. S. Diatchki and M. P. Jones. Strongly typed memory areas programming systems-level data structures in a functional language. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 72–83. ACM, 2006.

[14] I. S. Diatchki, M. P. Jones, and R. Leslie. High-level views on low-level representations. In *Intl. Conference on Functional Programming*, pages 168–179. ACM, 2005.

[15] G. Eakman, H. Reubenstein, T. Hawkins, M. Jain, and P. Manolios. Practical formal verification of domain-specific language applications. In *NASA Formal Methods Symposium*. Springer, 2015.

[16] L. Erkok. SBV: SMT based verification in Haskell. Website, 2014. `http://leventerkok.github.io/sbv/`.

[17] S. Frazzetta and M. Pacino. A STANAG 4586 oriented approach to UAS navigation - the case of Italian Sky-Y flight trials. *Journal of Intelligent and Robotic Systems*, 69:21–31, 2013.

[18] T. Hawkins. Controlling hybrid vehicles with Haskell. Presentation. *Commercial Users of Functional Programming* (CUFP), 2008. Available at `http://cufp.galois.com/2008/schedule.html`.

[19] P. C. Hickey, L. Pike, T. Elliott, J. Bielman, and J. Launchbury. Building embedded systems with embedded DSLs (experience report). In *Intl. Conference on Functional Programming (ICFP)*. ACM, 2014.

[20] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Conference*, Berkeley, CA, USA, 2002. USENIX.

[21] JPL. JPL institutional coding standard for the C programming language. Technical Report JPL DOCID D-60411, Jet Propulsion Laboratory, 2009. Available at `http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf`.

[22] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer, 2000.

[23] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: An alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 59–70. ACM, 2013.

[24] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. pages 24–35, June 1994.

[25] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 108–118. ACM, 2000.

[26] S. Lindley and C. McBride. Hasochism: The pleasure and pain of dependently typed haskell programming. In *Symposium on Haskell*, pages 81–92. ACM, 2013.

[27] J. P. Magalhães. The right kind of generic programming. In *Proceedings of the 8th ACM SIGPLAN Workshop on Generic Programming*, WGP '12, pages 13–24, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1576-0. .

[28] G. Mainland. Why it's nice to be quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, pages 73–82. ACM, 2007.

[29] N. D. Matsakis and F. S. Klock, II. The Rust language. *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, 34(3):103–104, Oct. 2014.

[30] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3-540-43376-7.

[31] L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: A hard real-time runtime monitor. In *Runtime Verification (RV)*, volume 6418, pages 345–359. Springer, 2010.

[32] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. *Intl. Conference on Functional Programming*, pages 51–62, Sept. 2008. ISSN 0362-1340.

[33] S. Sridhar. *BitC: A Safe Systems Programming Language*. PhD thesis, 2009.

[34] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66. ACM, 2012.