CAL for Haskell Programmers

Bo Ilic

Last modified: January 16, 2007

CAL is a strongly typed lazy functional language. The number of such languages in current development is quite small, the main ones being Haskell (www.haskell.org) and Clean (http://clean.cs.ru.nl/).

This paper compares and contrasts CAL with Haskell and is intended for readers already familiar with Haskell.

1. Summary of CAL language features

CAL implements essentially all the non-syntactic sugar features of Haskell 98 (with its standard addendums) including:

- algebraic functions with parametric polymorphism and inferred types
  - type declarations can specialize or assert types
- data declarations for algebraic types
  - strictness flags for data constructor arguments
- a module system supporting separate compilation
- expression syntax supporting if-then-else, case, let (for both local variable and function definitions) and lambda expressions
  - support for most of Haskell's expression operators
- special syntax for tuples, strings, characters, numbers and lists
- an extensive collection of standard libraries
- single parameter type classes
  - superclasses
  - derived instances, such as the instance declaration for Eq List
  - deriving clauses for common classes
  - default class method definitions
  - higher-kinded type variables, such as with the Functor type class
- dynamics support via the Typeable type class and Dynamic type
- user documentation generated from source code (similar to Haddock)
- foreign function support
- hierarchical module names

CAL takes a different approach to syntax than Haskell:

- CAL uses an expression-oriented style rather than providing equal support for an equation-based pattern matching style and an expression style
  - there are syntactic features for improving the convenience of pattern matching using the expression oriented style e.g. CAL's use of data constructor field names
- CAL does not make use of layout. Semicolons are required to terminate function and other definitions and in case pattern clauses.
- CAL uses Java's syntax for comments
- CAL's version of Javadoc or Haddock is more similar to Javadoc (and is called CALDoc)

CAL has the following features and characteristics not found in Haskell 98:

- CAL is built on top of the Java Platform
  - CAL code compiles directly to JVM instructions (or optionally to Java source code). CAL code is portable to any JVM platform and can be distributed as JAR files.
  - CAL itself is implemented in Java

- CAL provides the ability to work with Java object and primitive types and call any Java method, field or constructor
- CAL can surface various Java operators for casting, array operations, instanceof checks, and null checks as CAL functions
- polymorphic extensible records as in the Trex extension to Hugs
    - tuples in CAL are extensible records with ordinal field names
- full Unicode support
- support for internationalization and localization of CAL modules
- strictness annotations (plinging) on function arguments as well as data constructor fields
- ability to interoperate CAL with Java in an event-driven style e.g. Java can call a CAL function that produces a (potentially not fully evaluated) CAL value that can then depending on the client Java code, be further explored via further calls to CAL functions exploring this value
- apis for programmatically building CAL code using Java at runtime
- deepSeq is supported for all types i.e. deepSeq :: a -> b -> b; (as well as the usual seq :: a -> b -> b)
- Rich support for Dynamics. All types are automatically instances of the Typeable type class (if possible)
- various debugging tools, such as the ability to inspect arbitrary CAL values without evaluating them
- extensible support for exceptions

2. Why did we create CAL instead of just using Haskell?

a) CAL is simple to use for interacting with real world environments

CAL takes a practical approach to interacting with the real world. This is one of the defining differences between CAL and Haskell, and was one of the main motivators for us in implementing a different language. For example, in the "History of Haskell" paper by Paul Hudak et al, in section 3.2 it says that "Once we were committed to a *lazy* language, a *pure* one was inescapable". Of course, Haskell supports functions like unsafePerformIO, so there are qualifications to this statement, but the general philosophy is an emphasis on purity, and this is usually stated right at the beginning as a requirement if one adopts the lazy approach.

In contrast, CAL takes a more pragmatic approach. CAL's foreign function and type API lets programmers import any Java method, field or constructor as a CAL function, and any Java type as a CAL type. This makes it easy and convenient to access the rich array of Java libraries available with minimal effort. The drawback is that these imported types may be mutable, and the imported functions may not be pure. However, the imported functions and types can be made private to a module and CAL wrapper functions written to make the public functions actually exposed by the module pure.

CAL supports various mechanisms to make it easy to control laziness and evaluation order in cases such as writing the wrapper functions mentioned above. These are also quite useful for dealing with certain classes of space leaks in a simple way. For example, CAL supports seq, deepSeq, plinging of function arguments and plinging of data constructor arguments. In addition, CAL has an eager function primitive that tells the code generator to not create a lazy thunk for an expression, but just compile strictly.

Any algebraic function created using algebraic CAL types and non-foreign functions is pure. Impurity in the language only arises via the foreign API when referring to non-immutable Java types. We have found that the use of wrappers often lets us localize the impurity in a simple way. One canonical example is with the function `outputListWith :: [a] -> (a -> JObject) -> JList`, where `JObject` and `JList` are CAL foreign types for the Java types `java.lang.Object` and `java.lang.List` respectively. The implementation involves calling a foreign function to create a new `java.util.ArrayList`, and then adding elements of the CAL list to it, using an element marshaling function. In particular, the created Java list is mutated in the body of the function. However, due to strictness annotations, this happens in a definite order, and the resulting function `outputListWith` is pure.

Another technique for localizing state is through the use of execution context properties. It is frequently the case that state, such as the current time zone, is locally constant over a computation. CAL has an execution context, into which clients can set such state for the duration of an execution. Constant applicative form (CAF) values are also tied to an execution context.

In addition, the execution context gives clients control over the execution of a CAL program. For example, they can request that the CAL program suspends execution.

Another example of the practical nature of CAL is that CAL's String type is a foreign type with implementation type java.lang.String and not [Char].

b) CAL is a low-risk choice for business applications

Since CAL programs can be distributed as a collection of Java JARS with no other run-time support required, CAL based applications have the platform independence and security of a Java application. This makes CAL based applications more acceptable in situations where conservative IT policies are in place.

In addition, standard tools and practices for working with Java based applications work for CAL. For example, people are very familiar with deploying JARs. Many people know how to configure and tune a JVM for heap size, garbage collection characteristics etc. Tools used to diagnose runtime problems, such as profilers and debuggers, work with CAL.

c) Programs do not need to be entirely written, or even primarily written in CAL

We wanted to create a language where it was possible to introduce strongly typed lazy functional programming in the places where it makes sense, but be able to coexist with teams that did not use or know how to use this technology. We wanted to lower the barrier of entry to obtaining the benefits of CAL by being able to localize its use to the situations where the advantages were most dramatic.

d) CAL has extensive runtime language creation capabilities

Whole CAL modules can be programmatically created, checked, and compiled on the fly. Essentially all of the services of the CAL compiler and runtime are accessible to the application programmer at runtime in an efficient way. For example, new modules, or functions can be created, checked for correctness, and executed without going through slow intermediate formats (such as a source language textual representation) and without creating any intermediate files on disk. This is done in a fully multi-threaded way, both for compilation and execution.

One illustration of parts of this functionality is with the GemCutter tool which is used for creating CAL functions (Gems) in a graphical environment. The GemCutter provides rapid feedback to the user as to whether a connection between functions is type correct, suggests valid connections, and allows for the execution of gems in such a way that unsatisfied arguments have their values prompted for at run-time.

e) CAL seeks to be as comfortable as possible for mainstream developers to use

One of our design goals was to make CAL as simple and comfortable as possible for mainstream developers to use. In particular, while we wanted to make extensive use of concepts and ideas from the strongly typed functional world, consistency with the notation, terminology and programming style of this world was not a main priority.

For example, we wanted to use longer, more descriptive, less mathematical names, as is common in Java. The Haskell functions null and nub are isEmpty and removeDuplicates in CAL. The Haskell type class Monoid is Appendable in CAL.

We wanted to use longer variable names in our code, including in the standard libraries, and explain, in comments, why things were implemented the way they were. Our comments follow the Java conventions in syntax and in style.

3. CAL performance

CAL has a mature compiler and runtime implementation that can produce very fast code. In the case of micro-benchmarks, it is possible to write CAL code that performs essentially as well as Java code, mainly because the resulting CAL code compiles down to essentially the same underlying optimal Java code.

In addition, the CAL runtime has the following optimizations:

- compiles tail recursive functions as loops
- makes use of unboxed values (for foreign types). For example, the unboxed form of the Prelude.Int type is the primitive unboxed Java int.
    - these are used (where possible) for function arguments, return values, and data constructor field values
- efficient run-time representation of record types, especially for tuples
- a global optimizer, written in CAL, that does global transformations such as inlining and fusion
- no use of slow features such as the Java reflection API at runtime

The CAL compiler and run-time are both fully multi-threaded. Adjunct modules can be created, compiled and functionality in them executed in a fully multi-threaded way.

The CAL compiler can be configured to produce Java source code as an intermediate format (rather than Java bytecode directly). The resulting Java source code is fairly readable and can be used to understand how CAL lazy constructs are compiled, and the special cases that we handle. In particular, many of our run-time code generation optimizations have been discovered by looking at the generated source for key functions of interest, and asking ourselves how we would speed up the Java implementation if it were hand written.

4. CAL syntax examples

Below are several code examples to illustrate some of the syntax of CAL for someone familiar with Haskell. They are taken from the Standard CAL libraries, although often with some condensation for purposes of clearer exposition.

a) Declaration of the List.map function

This example shows the use of case expressions, type declarations, as well as some of the syntactic sugar for the List type. The usage of semicolons is different from Haskell and whitespace is not significant.

```
map :: (a -> b) -> [a] -> [b];
public map mapFunction !list =
    case list of
    []     -> [];
    listHead : listTail ->
        mapFunction listHead : map mapFunction listTail;
    ;
```

b) Declaration of the List.length function

Shows the use of the accumulating parameters pattern in the declaration of the local function lengthHelper, as well as plinging on the acc argument to make the function use constant space.

```
length :: [a] -> Int;
```

```
public length !list =
    let
        lengthHelper :: Int -> [a] -> Int;
        lengthHelper !acc !list =
            case list of
            [] -> acc;
            listHead : listTail -> lengthHelper (acc + 1) listTail;
            ;
    in
        lengthHelper 0 list;
```

c) Declaration of the outputListWith function

This function converts a value of the CAL List type to a value of the CAL JList type. The JList type is a foreign type whose implementation type is the Java type java.lang.List. There are foreign type declarations for the JObject and JList types, as well as foreign function declarations for jArrayList_new and jList_add. Notice that the implementation of outputListWith mutates a Java list object, but the resulting function outputListWith is still a pure function.

This example also includes some CALDoc that provides end-user documentation of the outputListWith function.

```
/**
 * Converts a CAL list to a Java list using the element mapping
function {@code f@} of type {@code a -> {@link JObject@}@}
 * to convert elements of the CAL list.
 *
 * @arg list the CAL list.
 * @arg f the mapping function converting elements of the list to
{@link JObject@} values.
 * @return the corresponding {@code JList@}.
 */
outputListWith :: [a] -> (a -> JObject) -> JList;
public outputListWith !list f =
    let
        javaList :: JList;
        javaList = jArrayList_new;

        outputListWithHelper :: [a] -> (a -> JObject) -> JList ->
JList;
        outputListWithHelper !list elementMappingFunction !javaList =
            case list of
            [] -> javaList;
            x : xs ->
                if (jList_add javaList (elementMappingFunction x)) then
                    outputListWithHelper xs elementMappingFunction
javaList
                else
                    error "Adding an element to a Java list failed.";
            ;
    in
        outputListWithHelper list f javaList;

data foreign unsafe import jvm public "java.lang.Object"
    public JObject deriving Inputable, Outputable;
data foreign unsafe import jvm public "java.util.List"
   public JList deriving Inputable, Outputable;
```

```
foreign unsafe import jvm "constructor java.util.ArrayList"
    private jArrayList_new :: JList;
foreign unsafe import jvm "method add"
    private jList_add :: JList -> JObject -> Boolean;
```

d) Declaration of the Maybe, Either and TypeRep algebraic data types

Field names (such as "value") below are required in the data declaration. Unlike in Haskell, field names do not occupy the function namespace, and can be repeated within a module. Field names can be repeated within a single data declaration (such as "typeConsName" below) for a different data constructor, even when they have different types.

```
data public Maybe a =
    public Nothing |
    public Just
        value :: a
    deriving Eq, Ord;

data public Either a b =
    public Left
        value :: a |
    public Right
        value :: b
    deriving Eq, Ord;

data public TypeRep =
    protected TypeRep
        typeConsName :: !String
        argTypes :: ![TypeRep] |

    protected FunctionTypeRep
        domainType :: !TypeRep
        codomainType :: !TypeRep |

    protected ListTypeRep
        elementType :: !TypeRep |

    protected UnitTypeRep |

    protected ForeignTypeRep
        typeConsName :: !String

    deriving Eq;
    ;
```

e) Syntax for pattern matching

Here are some examples of pattern matching syntax in CAL. It shows the various syntactic shortcuts that are available for extracting the fields of a data constructor.

This example shows traditional position-based case extraction. Instead of pattern variable *args* one could use an underscore ("_") to indicate an unused field value.

```
case typeOf (Just 'a') of TypeRep name args -> name;
returns
"Prelude.Maybe"
```

An example of field name based case extraction. One can name a subset of the field names of the TypeRep data constructor. In particular, if the TypeRep data constructor added additional fields, this code would not need to be updated.

```
case typeOf (Just 'a') of TypeRep {typeConsName} -> typeConsName;
returns
"Prelude.Maybe"
```

Here are some examples of data constructor field selection using the "." operator.

```
(Left "pear" :: Either String Double).Left.value
returns
"pear"

(Just "apple").Just.value
returns
"apple"

(typeOf (undefined :: JList)).ForeignTypeRep.typeConsName
returns
"Prelude.JList"
```

f) The Appendable type class

The Appendable type class is similar to the Monoid type class in Haskell. Note that the concat class method has a default implementation.

```
public class Appendable a where
    public empty :: a;
    public isEmpty :: a -> Boolean;
    public append :: a -> a -> a;
    public concat :: [a] -> a
        default concatDefault;
    ;

instance Appendable [a] where
    empty = emptyList;
    isEmpty = isEmptyList;
    append = appendList;
    concat = concatList;
    ;

instance Appendable String where
    empty = emptyString;
    isEmpty = isEmptyString;
    append = appendString;
    concat = concatString;
    ;
```

Here are a few sample expressions. The operator form of append is ++. Note also the use of the append class method in infix form using back-quotes (as in Haskell).

```
"peat" ++ "bog"
returns
"peatbog"
```

```
"peat" `append` "bog"
returns
"peatbog"

[3 :: Int, 1, 4] ++ [1, 5]
returns
[3, 1, 4, 1, 5]
```

g) Monad and Sequence type classes

Here are some examples of some of the more interesting type class definitions possible in CAL. The Monad type class involves a declaration for a higher kinded type variable m. In addition, there is a superclass constraint in that Functor is a superclass of Monad.

The Sequence type class can be used to abstract functionality over, for example, the List and Array types. Notice the additional constraints on the sort and concat class methods.

```
public class Functor m => Monad m where
    public bind :: m a -> (a -> m b) -> m b;
    public anonymousBind :: m a -> m b -> m b;
    public return :: a -> m a;
    ;

class Sequence c where
    length :: c a -> Int;
    subscript :: c a -> Int -> a;
    sort :: Ord a => c a -> c a;
    concat :: Prelude.Appendable a => c a -> a;
    ;

instance Sequence Prelude.List where …
instance Sequence Array.Array where …
```

h) Tuples and records

Tuples in CAL are simply records with ordinal field names #1, #2, #3, …. For example, the following 3 tuples are all the same:

```
("Anton", 3.0, True)
{#1 = "Anton", #2 = 3.0, #3 = True}
{#2 = 3.0, #3 = True, #1 = "Anton"}
```

Textual field names are also permitted, so the following is also a valid record:

```
{name = "Anton", occupation = "toddler", #99 = [Just 2.0, Nothing]}
```

The record selection operator is the dot:

```
("Anton", 3.0, True).#1
returns
"Anton"

{name = "Anton", occupation = "toddler"}.occupation
returns
"toddler"
```

Record extension is supported with special syntax or just using a record case expression:

```
{("Anton", 3.0) | toys = True}
returns
{#1 = "Anton", #2 = 3.0, toys = True}

case ("Anton", 3.0) of (f1, f2) -> {#1 = f1, #2 = f2, toys = True};
returns
{#1 = "Anton", #2 = 3.0, toys = True}
```

Record update is also supported with special syntax or just using a record case expression:

```
{("Anton", 10.0) | #2 := 11.0}
returns
("Anton", 11.0)

case ("Anton", 10.0) of (f1, _) -> (f1, 11.0);
returns
("Anton", 11.0)
```

Records in CAL support polymorphic record extension. For example, using the field1 function to extract the #1 field of a record works on any record with a #1 field:

```
field1 :: r\#1 => {r | #1 :: a} -> a;
public field1 !r = r.#1;

field1 ("pear", 2.0, Just 'a')
returns
"pear"

field1 {#1 = "fig", size = 100.0}
returns
"fig"
```

Finally, note that records can be instances of type classes. For example, all records are instances of the Eq, Ord, Show and Typeable type classes provided that their field types are also instances of these classes.

So for example:

```
sort [([1.0, 3.0], Just 'b'), ([1.0, 3.0], Just 'a'), ([1.0, 2.0], Just
'b'), ([1.0, 2.0], Just 'a')]

returns
[([1.0, 2.0], Just 'a'), ([1.0, 2.0], Just 'b'), ([1.0, 3.0], Just
'a'), ([1.0, 3.0], Just 'b')]
```

This example shows how records as well as the List, Char, Double and Maybe types are instances of the Eq and Ord type classes. Note that in the case of List and Maybe these are derived instance declarations. In the case of the record type we call the instance declarations *universal record instance declarations*. For example, there is an instance:

```
instance Eq r => Eq {r} where …
```

where the constraint implies that a record is an instance of Eq provided that any field also has a type that is an instance of Eq.

i) Exception support

Here is an example showing a function throwing a CAL value of a record-type as an exception, catching it, and then doing some simple manipulations:

```
//tests using various Cal types as Exception types, including the
interesting case of records
calThrownException5 =
    throw ("abc", 1 :: Int, 2 :: Integer, ["abc", "def"], Just (20 ::
Int))
    `catch`
    (
        let
            handler :: (String, Int, Integer, [String], Maybe Int) ->
String;
            handler r = show (r.#5, r.#4, r.#3, r.#2, r.#1);
        in
            handler
    );

//evaluates to True
testCalThrownException5 =
    calThrownException5 == "(Prelude.Just 20, [\"abc\", \"def\"], 2, 1,
\"abc\")";

instance Exception String where;
instance Exception Int where;
instance Exception Integer where;
instance Exception a => Exception (Maybe a) where;
instance Exception a => Exception [a] where;
instance Exception r => Exception {r} where;
```

j) Debugging support

CAL has a variety of features supporting debugging. This is a short overview of one such feature, the showInternal function.

The showInternal function is useful for seeing to what extent a lazy value has been evaluated, without in the process affecting the evaluation. Here is an example showing that the showInternal function does not force evaluation of its argument:

```
showInternal (Just 2.0).Just.value
returns
"(Prelude.Just 2.0).Prelude.Just.value"
```

Sharing is shown for nodes with children:

```
showInternal (let x = Just 'a'; in (x, x))
returns
"(<@1 = (Prelude.Just a)>, <@1>)"
```

The following expression shows an interesting example of sharing in one of the standard educational definitions of the Fibonacci numbers. Notice the @1 and @2 tags in the returned value. These show the sharing that is occurring in the graph.

```
let fibs = 1 : 1 : List.zipWith Prelude.add fibs (List.tail fibs) ::
[Integer]; in (deepSeq (take 7 fibs) (showInternal fibs))
```

returns

```
"(Prelude.Cons 1 (Prelude.Cons 1 (Prelude.Cons 2 (Prelude.Cons 3
(Prelude.Cons 5 <@1 = (Prelude.Cons 8 <@2 = (Prelude.Cons 13
(List.zipWith Prelude.addInteger <@1> <@2>))>)>)))))"
```