Herzlich Willkommen zum 1. Haskell in Leipzig Treffen (HaL)

Hinweise

- Bitte Namensschild nehmen & beschriften.
- Getränke: Im Eingangsbereich (wie alles kostenlos)
- WLAN Zugang: Name iba-cps-wlan / Passwort haskell2006

Motivation von BO

Seit 1999 arbeitet die Firma Business Objects (Hersteller von Crystal Reports) an einer Technik um **"Business Logic" als wiederverwendbare und kombinierbare Komponenten** darzustellen.

Diese deklarativen Komponenten wurden "Gems" getauft, um sie freundlich wirken zulassen.

Ziel war eine **in Java einbettbare Lösung**, welche nahtlos mit anderen Java Objekten und der Applikationslogik verbunden werden kann.

Java als Applikationsplattform (GUI, Applikationserver, Datenbanken, ...) erweitert um einen funktionalen Kern für knackige Probleme.

Quelle: http://labs.businessobjects.com/cal/

Beworbene Eigenschaften

- lazily evaluated, strictly-typed language called CAL, with many similarities to Haskell
- Full, dynamic, control of function composition from pure Java (without using CAL)
- A compiler capable of generating Java bytecode with efficient schemes
- Many optimizations specific to targeting a dynamic, OO language platform, such as the JVM
- A graphical language and tooling for interactively developing and testing Gems
- Debugging features, with value inspection that doesn't force evaluation
- Eclipse integration (still in progress)
- Fully multithreaded compiler and runtime
- Control of evaluation from Java, if required (suspension, resumption, exploring different parts of the result with just-in-time evaluation)
- Dynamic compilation - use SDK to create Gems at runtime, or create ad hoc compositions that might represent specific data flows in an application
- Easy integration to use Java types in CAL and to call regular Java logic
- Exception support

Quelle: http://labs.businessobjects.com/cal/

```
/**
* An infinite list of ones. CAL is a lazy language so this sort of thing works.
*/
ones :: [Int];
ones = 1 : ones;

/**
* The Fibonacci numbers. Another infinite list.
*/
fibs :: [Integer];
fibs = 1 : 1 : zipWith add fibs (tail fibs);

lazyListExamples =
    //can evaluate the first 4 elements of ones without hanging due to laziness
    assert (take 4 ones == [1, 1, 1, 1])
    &&
    //can evaluate the first 20 elements of fibs without hanging due to laziness
    assert
    (
        take 20 fibs
         == [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377
            , 610, 987, 1597, 2584, 4181, 6765]
   )
   ;
```

```
//CAL supports records. Records are basically sets of name value pairs. They differ
//from lists in that
//each field value of a record may have a different type.
/**
* A record with 3 fields: field #1 has type String, field #2 has type Maybe Boolean
* and field #3 has type [Double].
* It is expressed using tuple notation.
*/
tupleRecord1 :: (String, Maybe Boolean, [Double]);
tupleRecord1 = ("apple", Just True, [2.0, 3.0, -5]);

/**
* This record actually has the same value as tupleRecord1, but it includes field names
* explicitly, and thus
* uses braces rather than parentheses. When using explicit field names, the ordering of
* the fields does not matter.
*/
tupleRecord2 :: {#1 :: String, #3 :: [Double], #2 :: Maybe Boolean};
tupleRecord2 = {#3 = [2.0, 3.0, -5], #1 = "apple", #2 = Just True};

/**
* Here is a record with both textual and ordinal fields.
*/
mixedRecord1 :: {#1 :: Maybe [Int], #2 :: Boolean, age :: Double, name :: String};
mixedRecord1 = {name = "Anton", age = 3.0, #1 = Just [10 :: Int, 20], #2 = False};
```

```
recordExamples =
    //the 2 tuples have the same value even though one was expressed in tuple notation
    //and one was expressed using record notation
    assert (tupleRecord1 == tupleRecord2)
    &&
    //the field1 function extracts the #1 field of a record. It works with all records
    //having a #1 field and not just pairs or even tuples. This is technically called
    //"structural polymorphism".
    assert ( field1 tupleRecord1 == "apple"    && field1 mixedRecord1 == Just [10, 20])
    &&
    //the . operator can also be used to extract fields from records.
    assert ( mixedRecord1.name == "Anton" && mixedRecord1.#2 == False
             && tupleRecord2.#3 == [2, 3, -5]
    ) &&
    //the fieldNames function gives the field names in a record in field-name order, namely
    // ordinal fields first ordinal order followed by textual fields in alphabetical order
    assert ( Prelude.fieldNames mixedRecord1 == ["#1", "#2", "age", "name"] )
    &&
    //records can be updated (:=) and extended (=). Note that in the original mixedRecord1
    //the age field has type Double, but in the updated record the age field has type
    //Maybe Double. Also, note that this is a
    //non-destructive update- the original mixedRecord1 is not modified.
    assert (
        {mixedRecord1 | age := Just 3.5, favoriteFood = "chocolate"}
        == {name = "Anton", age = Just 3.5, #1 = Just [10 :: Int, 20]
           , #2 = False, favoriteFood = "chocolate"}
    )
```

iba CONSULTING GESELLSCHAFT
for intelligent business architecture

```
/** Here is a sorting implementation that makes use of a Java destructive in-place sort to
actually do the sorting. The algorithm is simply to marshall the CAL list to a Java list,
sort the Java list in-place, and then marshall the Java list back to a CAL list. Make use
of logic implemented within Java, even in the case of destructively updating functions
such as java.util.Collections.sort.*/
sortExternal :: [Int] -> [Int];
sortExternal list =
    (input :: JObject -> [Int])          //input the java.util.List to get a CAL [Int]
    # (output :: JavaList -> JObject)    //upcast to a JObject
    # (\list -> javaSort list `seq` list) //sort the java.util.List in-place using the
                                         //java.util.Collections.sort
    # (input :: JObject -> JavaList)     //downcast the JObject to a JavaList
    # (output :: [Int] -> JObject)       //output the [Int] to a JObject, which is a
                                         //java.util.List
    $ list                               //the original [Int]
    ;

/** foreign data declaration that makes the Java type java.util.List useable in CAL as the
CAL type JavaList */
data foreign unsafe import jvm "java.util.List" JavaList
    //the deriving clause provides default definitions of Prelude.output and Prelude.input
    //In the case of Prelude.input :: JObject -> JavaList, this is a Java downcast
    //In the case of Prelude.output :: JavaList -> JObject, this is a Java upcast
    deriving Inputable, Outputable;

/** foreign function declaration that makes the java.util.Collections.sort method
accessible in CAL as the CAL function javaSort. */
foreign unsafe import jvm "static method java.util.Collections.sort" javaSort
     :: JavaList -> ();
```

CAL's; non-syntactic sugar features of Haskell 98

- algebraic functions with parametric polymorphism and inferred types
- data declarations for algebraic types
    - strictness flags for data constructor arguments
- a module system supporting separate compilation, hierarchical module names (in progress)
- 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
- 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

Quelle: http://labs.businessobjects.com/cal/

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)

Quelle: http://labs.businessobjects.com/cal/

Features and characteristics not found in Haskell 98

- 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
- ability to work with Java object, primitive types, call any Java method, field or constructor
- polymorphic extensible records as in the Trex extension to Hugs
    - tuples in CAL are extensible records with ordinal field names
- 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. Types are automatically instances of the Typeable (if possible).
- various debugging tools, such as the ability to inspect arbitrary CAL values without evaluating them
- extensible support for exceptions

Quelle: http://labs.businessobjects.com/cal/

Warum nicht gleich Haskell? (1)

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

History of Haskell, P. Hudak: "Once we were committed to a lazy language, a pure one was inescapable"

CAL nimmt hier einen pragmatischen Ansatz:

- foreign function und Type API lassen Programmierer Java Typen, Objekte und Methoden als CAL Typ und CAL Funktion nutzen
- Leichter Zugang zu reichen Auswahl an Java Bibliotheken (JDBC, Security, Reporting, GUI ...)
- Nachteil: importierte Objekte/Funktionen können veränderbar (mutable) sein
- Lösung: verstecken in privaten Modulen hinter puren und öffentlichen Wrapper Funktionen,
- CAL macht dieses einfach durch Kontrolle des Laziness und Evaluation Reihenfolge
  - seq, deepSeq, pringling of function arguments, eager keyword
  - geschützte Ausführungskontexte (Execution Context) ein bestimmter Zustand ist konstant über eine Berechnung hinweg z.B. aktuelle Zeitzone.

Warum nicht gleich Haskell? (2)

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

- Leichte Verteilung von CAL Anwendungen als JAR Archive, sonst nichts weiter notwendig.
- Erhöhung der Akzeptanz durch Platform Unabhängigkeit und Sicherheit von JAVA, selbst in Konservativen Umgebungen (Strenge IT Policies)

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

d) "CAL has extensive runtime language creation capabilities"

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

- Zugänglich machen von starkgetypen funktionalen Welt
- Dafür aber Kompormisse bei Notation, Style, Terminology zu machen
- Durch längere, mehr beschreibene und weniger mathematischen Bezeichnungen:
  - Monoid => Appendable Type Class, null => isEmpty, nub => removeDuplicates

Free Open Source CAL?

"As Business Objects is not a general development tools vendor, we are exploring possibilities of releasing the foundational technology openly to the wider community interested in its capabilities. In particular, we are eager to hear from organizations and individuals who may be interested in using the Quark Framework for Java, whether for research, teaching or commercial purposes."

Bemerkung: BO hat sehr viele Standard Bibliotheken von Haskell 1:1 nachgebaut und sehr gut kommentiert: Pretty Printer, Parsec, Prelude ...

Aufruf: Alle Interessenten sollte sich an Business Objects wenden und diese motivieren CAL unter eine Open Source Lizenz zugänglich zu bekommen (z.B. BSD). Nur so hat diese Sprache eine Chance wirklich wahrgenommen zu werden.