petsc-hs: Haskell bindings for PETSc

Marco Zocca
Tanget software and TU Delft

PETSc User Meeting, June 28-30, 2016, Vienna

Abstract

Scientific programming is the craft of representing a physical phenomenon, or an algorithm that interacts with it, in form that can be executed on a computer. Correctness and performance arise as the primary concerns, however good scientific practice demands an experiment to be easily accessible to peer review. One approach to solving this conundrum is to embed high-performance numerical codes into a high-level language that lets users abstract away the necessary book-keeping. `petsc-hs` is an ongoing effort to embed PETSc in the purely-functional, statically-typed language Haskell; the type system assists the user in ensuring program correctness before its execution, and enables algebraic composition of functionality.

Overview of the language

In Haskell[1] every expression has a well-defined type that cannot change during execution, and if the user program "type-checks", the compiler infers the least-constrained type that represents it. Crucially, *functions* are syntactically distinct from *actions*: a function such as $\sqrt{\cdot}$ will always return the same result when called with the same arguments, whereas the same cannot be said of e.g. searching for a file in a directory or accessing a database. The latter two are examples of actions which require a notion of "state" that might change independently of the program accessing them, and in Haskell these are modeled via the Monad typeclass (which represents program concatenation). In particular, the `IO` monad represents interactions with the "outside world", e.g. memory, filesystem, keyboard/screen, network sockets.

Low-level bindings

In `petsc-hs`, the C language PETSc calls are wrapped via the `inline-c` package [3]. This provides bidirectional mapping of the elementary types (such as `Char`, `Int` and `Double`), a `Context` datatype for representing custom types (in the present case: `Vec`, `Mat`, `KSP`, `SNES`, etc.), and a quasiquoting syntax that lets the user splice in arbitrary C language expressions or code blocks with variable substitution:

```
vecCreate' :: Comm -> IO (Vec, CInt)
vecCreate' cc = withPtr $ \p -> [C.exp|int{VecCreate($(int c), $(Vec *p))} |]
   where c = unComm cc
```

Memory management and exception handling

Acquiring and releasing a resource (e.g. memory or a file/database handle) is a very common pattern. Omitting either leads to data being in an inconsistent state and memory leakage, respectively. In the current version of `petsc-hs`, resource handling is abstracted out using the `bracket` combinator from the `Control.Exception` library, which lets us write `with-` statements like the following :

```
withKsp :: Comm -> (KSP -> IO a) -> IO a
withKsp cc = bracket (chk1 $ kspCreate' cc) (chk0 kspDestroy')
```

The following specifies the setup sequence of the KSP object, while leaving the subsequent action(s) f unspecified:

```
withKspSetup :: Comm -> KspType_ -> Mat -> Mat -> Bool -> (KSP -> IO a) -> IO a
withKspSetup cc kt amat pmat ignz f = withKsp cc $ \ksp -> do
  kspSetOperators ksp amat pmat
  kspSetType ksp kt
  kspSetInitialGuessNonzero ksp ignz
  kspSetUp ksp
  f ksp
```

The next block specializes the one above by requiring solution of the linear system (`kspSolve`) before running its functional argument:

```
withKspSetupSolve ::
  Comm ->
  KspType_ ->
  Mat ->            -- linear operator
  Mat ->            -- preconditioner
  Bool ->           -- set initial solution guess to nonzero vector
  Vec ->            -- r.h.s
  Vec ->            -- solution (WILL BE OVERWRITTEN)
  (KSP -> IO a) ->  -- post-solve actions
  IO a
withKspSetupSolve cc kt amat pmat ignz rhsv solnv post =
  withKspSetup cc kt amat pmat ignz $ \ksp -> do
    kspSolve ksp rhsv solnv
    post ksp
```

The next function is a special case of the previous one, that also manages memory for a `Vec` that serves as solution of the linear system:

```
withKspSetupSolveAlloc ::
  Comm -> KspType_ -> Mat -> Mat -> Vec -> (KSP -> Vec -> IO a ) -> IO a
withKspSetupSolveAlloc cc kt amat pmat rhsv post =
  withVecDuplicate rhsv $ \soln ->
   withKspSetupSolve cc kt amat pmat True rhsv soln $ \ksp ->
    post ksp soln
```

The `chk0` and `chk1` helpers map PETSc return codes to Haskell exceptions, and these can either be handled specifically or "thrown" and displayed to the user; the surrounding `bracket` runs the "cleanup" action even if an exception has occurred during the execution of its body. In the above examples we also see how functional notation makes it straightforward to specify idiomatic use of PETSc as higher-level combinators, which are easier to remember and to convey to others.

Property testing

The Hspec [4] package lets the user specify expected outcomes via a DSL that provides documentation and expectation (e.g. `shouldBe`) combinators. The following example shows the setup and solution of a small linear system (using a KSP bracket from the previous section), and tests convergence of the norm of the residual with respect to the exact solution:

```
t_linSys_r3_1 = describe "t_linSys_r3_1" $
  it "solves a 3x3 linear system" $
   withPetscMatrix com m n MatAij ixd nz InsertValues $ \mat ->
    withVecNew com vrhs $ \rhs -> do
     let (_, _, _, mu) = fromPetscMatrix mat
     withKspSetupSolveAlloc com KspGmres mu mu rhs $ \ksp soln ->
       withVecNew com vsolnExact $ \solnE ->
        withVecVecSubtract soln solnE $ \solnDiff -> do
          nd <- vecNorm solnDiff VecNorm2
          nd < diffNormTol `shouldBe` True      -- test criterion
      where
        (m, n) = (3, 3)                         -- matrix size
        vrhs = V.fromList [3, 7, 18]            -- r.h.s.
        vsolnExact = V.fromList [1, 1, 1]       -- exact solution
        ixd = listToCSR m n [1,2,0,0,3,4,5,6,7] -- matrix elements, by rows
        diffNormTol = 1e-16                     -- linear solve convergence tolerance
        nz = VarNZPR (dnnz, onnz) where         -- matrix sparsity pattern
           dnnz = V.convert $ V.fromList [1,1,1]
           onnz = V.convert $ V.fromList [1,1,2]
```

The above, when run, results in :

```
t_linSys_r3_1
  solves a 3x3 linear system

Finished in 0.0186 seconds
1 example, 0 failures
```

The next example shows the use of the linear eigenproblem solver `EPS` from SLEPc, and tests whether the computed eigenvalues are all purely real :

```
t_eigen_r3_1 = describe "t_eigen_r3_1" $
  it "solves a 3x3 real linear eigenproblem: eigenvalues are real numbers" $
   withPetscMatrix com m n  MatAij ixd nz InsertValues $ \mat -> do
    let (_, _, _, mu) = fromPetscMatrix mat
    withEpsCreateSetupSolve com mu Nothing EpsHep $ \eps _ _  _ -> do
      ve <- epsGetEigenvalues eps
      let (_, ei) = V.unzip ve
      V.all (<= imzTol) ei `shouldBe` True
      where
        (m, n) = (3, 3)
        imzTol = 1e-16
        ixd = ixd3x3      -- same matrix and sparsity pattern as the previous example
        nz = nz3x3
```

Usage and deployment

In single-node user settings, `petsc-hs` can either be linked to the interactive Haskell compiler GHCi and used as a command-line application, or compiled as a native standalone binary. The multi-node version is currently being developed, on top of a library for "cloud" cluster computing[5].

Outlook and future work

The project is Open Source under the GPL3 license, development is public on GitHub [2] and has already attracted contributions from both the functional programming and numerical computing communities. There are numerous extensions under development, for example representing the `SNES`, `TAO`, `TS` parts of the library for nonlinear optimization and time integration, augmenting the PETSc objects with algebraic properties via typeclasses, and a domain-specific language for finite element programs.

Acknowledgements

This work was initially funded by STW grant 11363, "Robust Design Optimization for Integrated Photonic Systems".

References

- haskell.org
- github.com/ocramz/petsc-hs
- hackage.haskell.org/package/inline-c
- hspec.github.io
- github.com/tanget-sw/compute-cluster-sandbox