Sundials/ML: interfacing with numerical solvers

Timothy Bourke, Jun Inoue, Marc Pouzet

Sundials/ML: interfacing with numerical solvers*

Timothy Bourke
Inria Paris
École normale supérieure,
PSL Research University
Timothy.Bourke@inria.fr

Jun Inoue
National Institute of Advanced
Industrial Science and
Technology
Jun.Inoue@aist.go.jp

Marc Pouzet
Univ. Pierre et Marie Curie
École normale supérieure,
PSL Research University
Inria Paris
Marc.Pouzet@ens.fr

ABSTRACT

We describe a comprehensive OCaml interface to the Sundials suite of numerical solvers (version 2.6.2). Noteworthy features include the treatment of the central vector data structure and benchmarking results.

1. INTRODUCTION

Sundials [3] is a suite of six numerical solvers for ordinary differential equations, differential algebraic equations, and non-linear equations. We wrote an OCaml interface because we needed state-of-the-art solvers for the runtime system of Zélus [1], but our interface could benefit any project that involves both symbolic manipulation and numeric computation. It augments the C library with the usual features of ML—namely, static and dynamic safety checks, automatic memory management, error propagation via exceptions, callbacks with closures, and a programming model based on types and modules.

Although the basic techniques for interfacing OCaml and C are well understood [5, chap. 19], we required several iterations to find a design that minimizes copying, avoids memory leaks, and encompasses all features of the library without unnecessary duplication. The Sundials solvers share common datatypes for vectors and matrices. We present the solution we found for the former and benchmark results. Source-code and documentation are available online,[1] including the benchmark examples and scripts for analyzing them.

2. VECTORS

Vectors in Sundials are represented by an abstract data type that combines a pointer to the underlying array of floats and 25 function pointers to generic operations like *nvdotprod*, which computes the dot product, and *nvclone*, which creates a fresh copy of the given vector. There are five kinds of vectors: serial, OpenMP, Pthreads, parallel, and custom (with user-implemented operations). Vectors are passed to callbacks that perform problem-specific calculations in performance-critical solver loops, so an efficient way of accessing their payload from OCaml is important.

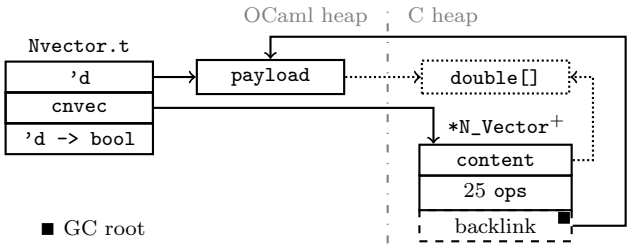

Figure 1: Interfacing (serial) vectors

Initially, we transparently converted Sundials vectors to and from bigarrays [5, chap. 30], which allow direct sharing of arrays of floats between OCaml and C. This approach is relatively simple and allows users to work directly with bigarrays, but it requires either frequent allocation of bigarray headers on the major heap or awkward caching. Moreover, different vector kinds require different interfaces requiring either code duplication or costly indirection.

The approach we eventually settled on exploits features of the vector abstract datatype and polymorphic typing to treat vectors uniformly without caching or code duplication. The opaque `Nvector.t` type used throughout our interface is represented internally as

```
type ('d, 'k) t = 'd * cnvec * ('d -> bool)
```

where `'d` is the type of the payload, a bigarray for serial vectors or a bigarray and an MPI communicator[2] for parallel vectors, and `'k` is a phantom type for distinguishing vector kinds. Figure 1 shows the memory layout; `Nvector.t` is pictured at left. The first component references the payload in the OCaml heap, the second points to a Sundials `N_Vector` in the C heap, and the third is a function for checking dynamic compatibility of vectors, like having equal lengths. The `cnvec` pointer passes via a custom block (not shown) in anticipation of planned changes to OCaml that forbid the use of naked C pointers as OCaml values.

Sundials requires a 'seed' vector to create a solver session and allocates all internal storage by replicating the seed with *nvclone*. We wrote a custom *nvclone* operation that allocates extra space after the `N_Vector` structure to store a

*This work was supported by the ITEA 3 project 11004 MODRIO (Model driven physical systems operation).

[1] http://inria-parkas.github.io/sundialsml/

[2] https://forge.ocamlcore.org/projects/ocamlmpi/.

ACM Workshop on ML 2016 September 22, 2016, Nara, Japan

ACM ISBN 123-4567-24-567/08/06…$15.00

DOI: 10.475/123_4

GC-registered root back to the associated OCaml payload. These roots are used to retrieve the values to pass to callback functions, which thus directly receive a payload rather than an `Nvector.t`. Extending `N_Vector` in this way means that the original code for operations like *nvdotprod* is called directly. The bigarray in the payload and the `N_Vector` both point to the same content array which is only freed when the bigarray is finalized by the garbage collector. We did not link back to an `Nvector.t` as this would have created an inter-heap cycle and necessitated special treatment to avoid memory leaks.

Our approach thus involves two classes of vector: those created by OCaml code with a lifetime linked to the associated `Nvector.t` and determined by the garbage collector, and those cloned within Sundials, for which there is no `Nvector.t` and whose lifetime ends when Sundials explicitly destroys them, though the payload may persist. Sundials only destroys vectors that it cloned itself: data linked from vectors created in OCaml is never prematurely deallocated.

Sundials poses two main constraints on vector use. Firstly, only a single kind may be used per solver session, since operations like *nvdotprod* access the underlying representation of all their arguments. Secondly, certain linear solvers may only be used with kinds that store their data in a single, local array; that is, with serial, OpenMP, or Pthreads vectors, but not with parallel vectors. These rules are enforced statically using the `'k` type variable and OCaml's polymorphic variants [2]. Three of the vector kinds are declared as closed sets of tags:

```
type Nvector_serial.kind   = [‘Serial]
type Nvector_openmp.kind   = [‘OpenMP | ‘Serial]
type Nvector_pthreads.kind = [‘Pthreads | ‘Serial]
```

Parallel and custom vectors have opaque kinds. Functions that accept any of the first three kinds but no others take a vector with `constraint 'k = [>Nvector_serial.kind]`, that is, any kind that includes the `‘Serial` constructor. The type of sessions includes a `'k` fixed by the initial seed vector.

3. BENCHMARKS

Sundials is distributed with 56 example programs that use serial, OpenMP, or Pthreads vectors and 21 that use parallel vectors (ignoring duplicates). We reimplemented them in OCaml and compared their performance to the C versions. This process uncovered many bugs in our code and several in Sundials itself. We repeatedly executed examples with manual garbage collector invocations to expose memory-handling errors.

After ensuring that the OCaml examples gave the same results as the C counterparts, we optimized running times. Most optimizations were in the example programs themselves where we followed the standard approach of adding type annotations to vector arguments to reduce polymorphism, avoided functions like `Bigarray.Array1.sub` that allocate fresh bigarrays on the major heap, and wrote numeric expressions and loops to avoid float boxing [4].

Figure 2 shows the ratios of the wall-clock times of the OCaml code against the C code. A value of 2.0 on the left axis means that the OCaml version takes twice as long as the C version. The running times of the parallel examples (not shown) are dominated by process set up and communication costs with ratios mostly close to 1.0 and rarely more than 1.1. The grey bars show the results for ‘customized’

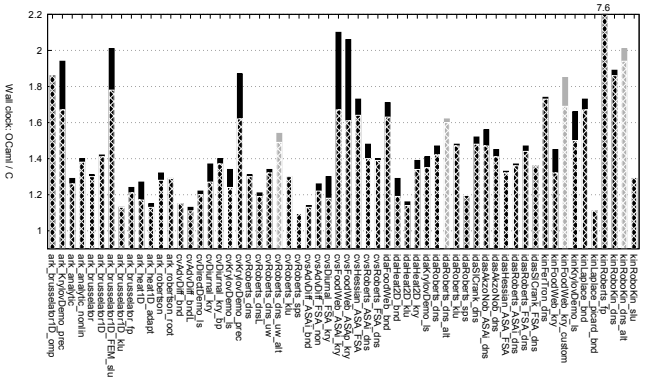

Figure 2: Serial examples: C (gcc 4.9.2 with -O2) versus OCaml native code (4.03.0). The hashed sub-bars show the results when compiled with unsafe. (Linux 3.16.0 on 2.60 GHz i7 with 1 MB L2/5 MB L1/8 GB RAM.)

examples: the kinFoodWeb_kry_custom example uses custom vectors with low-level operations implemented in OCaml on `float array`s; the ∗_alt examples use a linear solver reimplemented in OCaml. The hashed sub-bars give the ratios when the OCaml versions are compiled without checks on array access and vector compatibility.

Nearly all of the examples run in less than 1 ms; the two longest are on the order of 3 s. As we could not profile such short executions directly, we had the examples repeatedly execute their `main` functions. Since the C code `free`s its data at each iteration, we also manually triggered a full heap compaction in OCaml before terminating. The fastest examples require 1000s of iterations to become measurable, so we varied the number of repetitions per example to avoid the slower examples taking hours. The OCaml version with the worst ratio is iterated 100 000 times. When major collections are not invoked manually and the minor heap is enlarged (to 128 MB), the OCaml/C ratio of this example drops from 7.6 to 1.8. Otherwise, the graph shows that the OCaml versions are rarely more than 60% slower than the originals and they are often less than 30% slower.

4. REFERENCES

[1] T. Bourke and M. Pouzet. Zélus: A synchronous language with ODEs. In *HSCC*, pages 113–118. ACM Press, Apr. 2013.

[2] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*. ACM, Sept. 1998.

[3] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Mathematical Software*, 31(3):363–396, Sept. 2005.

[4] X. Leroy. Writing efficient numerical code in Objective Caml. http://caml.inria.fr/pub/old_caml_site/ocaml/numerical.html, July 2002.

[5] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system: Documentation and user's manual*. Inria, 4.03 edition, Apr. 2016.