Improving your CASH flow: The *C*omputer *A*lgebra *SH*ell

Christopher Brown[1], Hans-Wolfgang Loidl[2], Jost Berthold[3], and Kevin Hammond[1]

[1] School of Computer Science, University of St. Andrews, UK.
`{chrisb,kh}@cs.st-andrews.ac.uk`
[2] School of Mathematical and Computer Sciences, Heriot-Watt University, UK.
`hwloidl@macs.hw.ac.uk`
[3] DIKU Department of Computer Science, University of Copenhagen, Denmark.
`berthold@diku.dk`

Abstract. This paper describes CASH (the Computer Algebra SHell), a new interface that allows Haskell programmers to access the complete functionality of a number of computer algebra systems directly and interactively. Using CASH, Haskell programmers can access previously-unavailable mathematical software. Additionally, users of computer algebra systems can exploit the rapidly growing Haskell code base and its rich set of libraries. In particular, CASH provides a simple and effective interface for users of computer algebra systems to parallelise their algorithms using domain-specific *skeletons* written in Haskell.

1 Introduction

Users of functional programming languages can often feel as if they are in a ghetto. While foreign-function interfaces (FFIs) often exist to e.g. C, they can, indeed, feel foreign to use, especially where complex data structures are involved. And while library developers work hard to provide new functionality to functional programmers, this hard-won capability often bit-rots faster than it can be used effectively. As part of an EU-funded project[4], we have been working to address some of these issues (and, incidentally, to provide a new user base for functional languages). In this way, we hope to open the gates to the ghetto of functional programming, and break some of the barriers preventing wider use.

This paper describes a new way to link Haskell [1] with computer algebra systems, avoiding the FFI route. CASH (the Computer Algebra SHell) provides a two-way interface to a number of widely-used computer algebra systems, including GAP [2], KANT [3], MuPAD[4], Maple [5] and Macaulay [6]. In order to achieve this, it exploits the generic SCSCP [7] protocol that has been implemented for all these systems. This, in turn, uses the *OpenMath* [8] standard format to describe the structure of mathematical objects.

[4] SCIEnce: Symbolic Computation Infrastructure in Europe, RII3-CT-2005-026133.

J.Hage and M.Morazan (Eds.), IFL 2010, LNCS 6647, 2011.

Interfacing between (an interactive shell for) Haskell and computer algebra systems has several advantages. Firstly, these systems are *domain-specific environments* that provide a large number of specialised data-types and that build on decades of expert experience. For instance, the open source system GAP [2] comes with a library of complex computer algebra algorithms that have been contributed by hundreds or thousands of domain experts. Exploiting existing optimised environments such as GAP from within Haskell increases programmer productivity since the Haskell programmer does not have to learn the mathematics behind the code. Providing an interface to Haskell from within a computer algebra system allows advanced features to be accessed, such as Haskell's parallelisation capabilities. The domain expert does not have to learn about parallel programming: they can use their favourite computer algebra system,

Contributions. The main technical contributions of this paper are as follows:

1. we show how CASH can be used to integrate Haskell and computer algebra code using SCSCP and OpenMath, giving examples that show how CASH can be used to call computer algebra systems from Haskell and vice-versa;
2. we show how to define *domain-specific* parallel skeletons for computer algebra in the Eden [9] dialect of Haskell, in particular, we describe a new *multiple homomorphic images skeleton*; and
3. we expose these parallel skeletons as SymGrid-Par [10] services, that can be called directly both from the CASH shell and from within the shell of any SCSCP-enabled computer algebra client.

The idea of interfacing functional languages and computer algebra systems is, of course, not new. For example, GHC-Maple [11] and Eden-Maple [12] provide dedicated sequential and parallel interfaces, respectively, to the commercial Maple system. However, CASH goes beyond these earlier approaches in terms of both usability and generality: CASH uses a Haskell implementation of the standardised OpenMath-based SCSCP [7] interface, which means that it can interact with *any* computer algebra system that implements the SCSCP standard; moreover, it also allows *any* SCSCP-compliant system to interact with Haskell.

2 Linking Haskell and Computer Algebra Systems

CASH uses the GHCi interpreter to provide a Haskell-side shell for accessing computer algebra systems via SCSCP. The underlying libraries provide data structures and conversion functions targeted at computer algebra, plus a "command line API" for an interactive front end, that is used to establish a connection to, and interact with, an SCSCP-enabled system. To show how powerful, expressive and useful the CASH system can be, we will consider a simple example involving matrix groups over *finite fields*. We first define two matrices, `m1` and `m2` over the finite field $\mathbb{Z}_3 = \{0, 1, 2\}$, interactively, using the CASH shell

```
*Cash> let m1 = [[Z(3)^0, Z(3)^0],[Z(3),0*Z(3)]]
*Cash> let m2 = [[Z(3), Z(3)],[Z(3),0*Z(3)]]
```

Here we have used the GAP-like syntax `Z(3)` for the element which generates the multiplicative group in the finite field $\mathbb{Z}_3$, `0*Z(3)` for the 0-element and `Z(3)^0` for the 1-element. Using CASH, we can directly exploit any of the GAP functions on group algebra. Here we compute the multiplicative matrix groups generated by each of these two matrices and then determine their intersection.

```
*Cash> group(m1)
Group([ [ [ Z(3)^0, Z(3)^0 ], [ Z(3), 0*Z(3) ] ] ])
*Cash> group(m2)
Group([ [ [ Z(3), Z(3) ], [ Z(3), 0*Z(3) ] ] ])
*Cash> intersection(group(m1), group(m2))
Group([ [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ] ])
```

The generator of the intersection is the unit matrix multiplied by the scalar `Z(3)`. Note that the result of each group operation is simply a symbolic representation of the result in terms of one or more generators, rather than an enumeration of all possible elements of the group. This is important, because the number of elements in a group can be huge, and the user is usually only interested in some specific properties of the group. We can often determine these properties without needing to generate all the elements of the group. This dramatically reduces the amount of computation that is required.

2.1 The SCSCP Interface

The interface between a client and a server is defined by the SCSCP communication protocol [7], which itself builds on the existing OpenMath standard [8] for data representation of mathematical objects. The SCSCP communication protocol defines how messages are exchanged between client and server processes. Its main functionality is to support (remote) calls of SCSCP server-side services and to deliver results to the SCSCP client. In order to support automatic service discovery, the protocol also specifies how to obtain a list of all the available services that are supported by the SCSCP server, including their types.

Figure 1 shows a CASH session interacting with a GAP server using SCSCP. The CASH session first connects to GAP with a **Procedure Call** message, containing information such as the *Procedure name*, any *Arguments* and any *Options/Attributes*, encoded in OpenMath. Alternatively, the user may **interrupt** the interaction at any time between the computer algebra system and the server. In this case the computation terminates and no result is returned. The GAP server responds to the CASH client by sending a **Procedure Completed** message: an OpenMath encoding containing the information about the result of the procedure. This message contains the (OpenMath) *Result value*; any *Mandatory Additional Information*, such as the internal SCSCP call identifier; and any *Optional additional information*, such as the procedure runtime/memory usage. Alternatively, the GAP server may instead return a **Procedure Terminated** message containing an *Error* message. The CASH user-level interface is provided by the `callSCSCP` function, which takes a service name and a list of *OpenMath* objects as arguments and issues an SCSCP call to the server. To simplify this

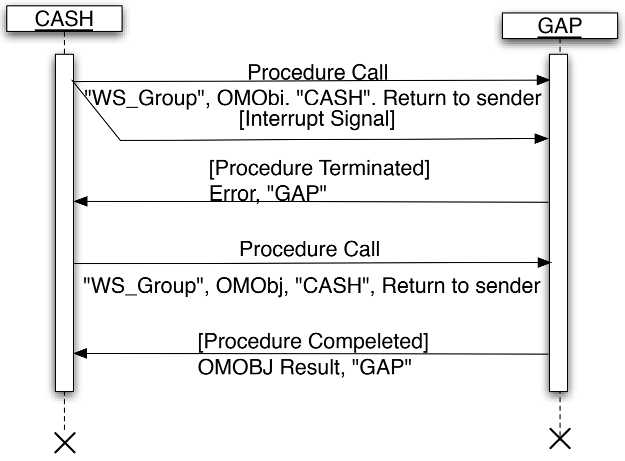

Fig. 1. An example interaction between CASH and GAP using SCSCP

interface, a family of functions `call1`, `call2` etc. are defined, which hide the (un)marshalling of the data-structures. Specifically, `call1` is used to connect with `SCSCP` functions that take one argument, `call2` is used to `SCSCP` functions taking two arguments, and so forth. These functions differ only in their arity, and could be automatically generated using tools such as Template Haskell.

2.2 The OpenMath Data Format

The OpenMath [8] data format is an XML-based representation of mathematical objects such as polynomials, finite field elements or permutations. Formally, an OpenMath object is a labelled tree whose leaves form typical computer algebra type representations. These include integers, unicode strings, variables or symbols, where *symbols* consist of a name and a *content dictionary*. For instance, a vector of finite field elements `[[Z(3)^0,Z(3)^0]]` can be encoded in the OpenMath representation shown in Figure 2. This uses the `linalg2` content dictionary to define `matrix` and `matrixrow`.

3 Calling GAP from Haskell and vice-versa

We now give two examples of using CASH that show the usefulness of being able to call computer algebra functionality from a Haskell environment without having to step out of the functional paradigm.

3.1 Greatest Common Divisor

Our first, simple, example shows how to implement a generic greatest common divisor (GCD) function by first calling existing factorisation functions in the

```
<OMOBJ>
    <OMA> <OMS cd="linalg2" name="matrix"/>
          <OMA> <OMS cd="linalg2" name="matrixrow"/>
                <OMA> <OMS cd="arith1" name="power"/>
                      <OMA> <OMS cd="finfield1" name="primitive_element"/>
                            <OMI>3</OMI>
                      </OMA>
                      <OMI>0</OMI>
                </OMA>
                <OMA> <OMS cd="arith1" name="power"/>
                      <OMA> <OMS cd="finfield1" name="primitive_element"/>
                            <OMI>3</OMI>
                      </OMA>
                      <OMI>0</OMI>
                </OMA>
          </OMA>
     </OMA>
</OMOBJ>
```

Fig. 2. An example of an OpenMath encoding for finite field elements

computer algebra system (here, GAP) and by then combining the results in Haskell. Although this operation is fairly trivial in itself, it highlights some important design features: first, we use Haskell's overloading mechanism to define a generic GCD implementation; and second, the compute-intensive parts (that is, the factorisation and polynomial operations) are all delegated to the computer algebra system using SCSCP services. The `myGcd` algorithm first factors the inputs `x` and `y`. It then perfoms a bag-intersection on the Haskell side, to identify all the common factors. Finally, the GCD is computed by folding the generic multiplication operation over the list of common factors.

```
-- intersection on multi-sets (bags)
bagInter :: (Eq a) => [a] -> [a] -> [a]
bagInter [] _ = []
bagInter (x:xs) ys | elem x ys = x:(bagInter xs (delete x ys))
                   | otherwise = bagInter xs ys

-- generic GCD computation
myGcd :: (Num a, Factorisable a) => a -> a -> a
myGcd x y = let  xs = factors x
                 ys = factors y
                 zs = xs `bagInter` ys
            in product zs
```

Marshalling and UnMarshalling Polynomials: In order to use the algorithm on polynomials in the computer algebra system, we need to define their OpenMath representation, as part of the `OMType` type. We currently implement `Polynomials` as Haskell `Strings`, since it is usually easy to convert strings to polynomials in the computer algebra system.

```
data OMType = List [OMType]  | Rational [OMType] | Matrix [OMType]
            | MatrixRow [OMType] | Mul FiniteField Int
            | Power FiniteField Int | Num Integer
            | Polynomial String
            | ...
data FiniteField = PrimEl Int
```

Basic operations on polynomials are implemented via SCSCP calls to the corresponding SCSCP services. These calls use the `call2` wrapper function, which implicitly applies `toOM`/`fromOM` to convert data to/from values of type `OMType`.

```
instance  Num (OMType) where
 (*) p1@(Polynomial _) p2@(Polynomial _) = call2 scscp_WS_ProdPoly p1 p2
 (*) ...
 (+) p1@(Polynomial _) p2@(Polynomial _) = call2 scscp_WS_SumPoly p1 p2
 (+) ...
```

Factorising Polynomials: We can now write Haskell code that calls computer algebra functions to factorise polynomials. The `Factorisable` class contains a single function `factors` that returns a list of all the factors of a given argument.

```
class Factorisable a where
  factors :: a -> [a]
```

We can easily define an instance of this class for polynomials that invokes the corresponding computer algebra service, as follows:

```
instance Factorisable (OMType) where
  factors = call1 scscp_WS_Factors
```

An Example CASH Session: The trace below shows how CASH can be used with our GCD implementation. We define two simple input polynomials `p1` and `p2` and compute their GCD in Haskell using GAP polynomial operations.

```
*Cash> let p1 = polyFromString "x_1^3-x_1"
*Cash> factors p1
[x_1-1,x_1,x_1+1]
*Cash> let p2 = polyFromString "x_1^2+x_1"
*Cash> factors p2
[x_1,x_1+1]
*Cash> myGcd p1 p2
x_1^2+x_1
```

3.2 A Linear System Solver

We now show how CASH can connect to GAP to run more serious computer algebra computations. Given a linear system of equations over arbitrary precision integers, represented as a matrix $A \in \mathbb{Z}^{n\times n}$ and a vector $b \in \mathbb{Z}^n, n \in \mathbb{N}$, we want to find a vector $x \in \mathbb{Z}^n$ s.t. $Ax = b$. As is common for symbolic computations, and in contrast to the more usually encountered numerical algorithms, we need to produce an *exact* solution for arbitrary precision integers.

Multiple Homomorphic Images: A potential problem in solving a system of linear equations is that the values in the matrix tend to become very large, meaning that even simple arithmetic operations can become expensive to compute. A common way to avoid this is to use a technique known as Multiple Homomorphic Images [13]. In order to control the size of the matrix elements, a solver based on homomorphic images will use a list of prime numbers, generate new matrices modulo each prime, find the solutions to these smaller problems, and finally combine these solutions using the Chinese Remainder algorithm (CRA). The general *multiple homomorphic images* approach [13] consists of the following three stages, where the names in brackets refer to the Haskell skeleton below:

1. map the input data into several homomorphic images (`fwd`);
2. compute the solution in each of these images (`sol`); and
3. combine the results of all images to a result in the original domain (`comb`).

In this case the original domain is $\mathbb{Z}$, the set of (arbitrary precision) integer values, and the homomorphic images are $\mathbb{Z}$ modulo p, written $\mathbb{Z}_p$, with p being a prime number. If suitably large prime numbers are chosen, then cheap fixed-precision arithmetic can be used for each of the homomorphic images. It is only necessary to use expensive arbitrary precision arithmetic in the combination phase, when applying the Chinese Remainder Algorithm [14] in Step (3). Thus, this is already a very efficient sequential algorithm. It also has many uses: for example, it allows us to solve a wide class of problems over Euclidean domains and can be used to compute multivariate resultants.

A Multiple Homomorphic Images skeleton: It is straightforward to define a skeleton in Haskell to compute multiple homomorphic images:

```
multHomImg :: (Integer   -> OMType -> OMType) ->
              (Integer   -> OMType -> OMType) ->
              ([Integer] -> OMType -> OMType) ->
               [Integer] -> OMType -> OMType
multHomImg fwd sol comb ps x = res
 where xList   = zipWith fwd ps (repeat x)
       resList = zipWith sol ps xList
       res     = comb ps (toOMMatrix resList)
```

The local definitions are each the result of one of the three steps above. The lists of inputs, `xList`, and solutions in the homomorphic images, `resList`, are both (potentially infinite) lists. The combiner, `comb` is allowed to ignore the tail of the possibly infinite list, `ps`. This design uses Haskell's laziness to avoid hard-coding a bound on the list lengths into the skeleton itself. Since data marshalling in SCSCP is eager, however, these lists have to be finite when using SCSCP calls in a concrete instance of the skeleton. Only data structures that remain on the Haskell side can be infinite.

The Linear Solver: We can use this skeleton to define a linear system solver:

```
linearSolver :: [ Integer ] -> [ [Integer] ] -> [ Integer ] -> OMType
linearSolver ps ms vs
  = multHomImg (call2 scscp_WS_Mod) (call2 scscp_WS_Sol)
               (call2 scscp_WS_CRA) ps
               (toOMList ((toMatrix ms):[toOMList (map toNum vs)]))
```

The functions `scscp_WS_Mod`, `scscp_WS_Sol` and `scscp_WS_CRA` are *SCSCP* calls to the GAP functions `ModMat`, `SolMat` and `CRAMat`, respectively. `ModMat` takes a prime number, a matrix and vector and returns the matrix and vector modulo the prime number.

```
ModMat:=function(p, x)
   return([x[1] mod p, x[2] mod p]);
end;
```

Here, `x` is a GAP list containing both the matrix and vector, so `x[1]` accesses the matrix and `x[2]` accesses the vector. `SolMat` computes the solution for a matrix and vector modulo a prime number. The interesting thing here is that the GAP code computes the modulus by transforming the vector and matrix to an element in a finite field of p elements, where p is a prime number. The function checks that the determinant of the matrix is not 0 and returns the empty list if so, since there would otherwise be no solution. To compute the solution for a matrix and a vector in GAP we use the builtin library function `SolutionMat`.

```
SolMat:=function(p, x)
  local e, r;
  e:=One(GF(p));
  if Determinant(x[1]*e) = 0*Z(p) then
    return([]);
  fi;
  r:=SolutionMat(TransposedMat(x[1]*e),x[2]*e);
  return(List(r, Int));
end;
```

Finally, `CRAMat` combines the result vectors using GAP's built-in Chinese Remainder algorithm, which works over (finite) lists. We also need to eliminate any empty matrices that were generated by 0-determinants in the `SolMat` stage:

```
CRAMat:=function(ps, r)
  local i, pris, vecs, res;
  i:=0;
  vecs:=[];
  while ( i < Length(r) ) do
     if not r[i] =  [] then
        pris:=Concatenation(pris, ps[i]);
        vecs:=Concatenation(vecs, r[i]);
     fi;
     i:=i+1;
  od;
  res:=ChineseRem(pris, vecs);
  return res;
end;
```

Testing the solver in CASH: For the purposes of this example, we will generate a random 1000 x 1000 matrix, together with a 1000 element vector of solutions.

```
*Cash> let m = generateMatrix(1000)
*Cash> let v = generateVector(1000)
```

We can then compute $m \times v$ to form b, and test the solution in CASH:

```
*Cash> let b = m * v
*Cash> (linearSolver [1009, 10007, 100003, 1000003] m b) == v
True
```

We note that the operator `(*)` is *overloaded* here so that it actually calls the GAP `(*)` operator on matrices.

4 Using CASH to Enhance Parallelism

Using one of the several available parallel Haskell implementations, such as the one that is provided with GHC, it is easy to parallelise the multiple homomorphic images skeleton from Section 3.2. We can then make this available to computer algebra users through the CASH interface. Hitherto most computer algebra systems have limited, if any, support for parallelism. Therefore a Haskell-side parallel skeleton which is accessible to the computer algebra user in this way is a major advantage of our design. Here we use the Eden parallel dialect of Haskell, which targets parallel clusters. Using Eden, we can simply transform the original skeleton, which called the sequential `zipWith` function, into an equivalent parallel version that uses `parZipWith` instead.

```
multHomImg fwd sol comb ps x = res
   where xList = zipWith fwd ps (repeat x)
         resL  = parZipWith sol ps xList
         res   = comb ps (toOMMatrix resL)
```

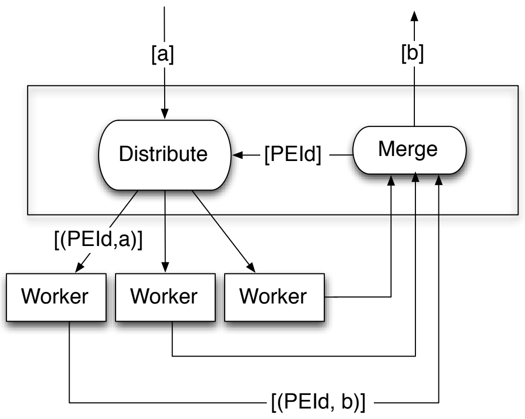

Fig. 3. The workpool skeleton

A parallel zipWith skeleton: The `parZipWith` skeleton uses a *workpool* [15] approach (see Figure 3). Workpools are commonly used where tasks have varying granularities, to dynamically balance the allocation of tasks to processors.

```
parZipWith f l1 l2 = newTasks
 where
  (newReqs, newTasks) = ...
  workerProcs = [process (zip [pe,pe..].(worker f))| pe <- [1..noPe]]
  worker f [] = []
  worker f ((v1, v2) : ts) = (f v1 v2) : worker f ts
```

The `parZipWith` skeleton creates one `worker` process for each available processor (PE), where each worker process applies `f` to a pair of two arguments in a Curried way. Each of these processes is executed in parallel using the Eden `process` construct. `workerProcs` defines this set of worker processes, pairing each result with the id of the PE that produced it, `pe`. This is used by the skeleton to determine which PEs have surplus capacity.

```
(newReqs, newTasks) = (unzip . merge) (zipWith ( # ) workerProcs
                          (distributeLists (l1, l2) requests))
requests = (concat (replicate 2 [1..noPe])) ++ newReqs
```

The skeleton pairs the input tasks (the pairs of arguments to the worker function `f`) with a list of *requests*. The `requests` value is a list of process ids that is used to map tasks to processes. Each task is paired with a process id using the `distributeLists` function and these ids are then used to assign the task to the corresponding Eden process. The results of executing the tasks on the processes are merged using Eden's non-deterministic `merge` operation, and then unzipped to give the lists of new requests, `newReq`, and new tasks, `newTasks`.

The `distributeLists` function simply accumulates the tasks for each PE in an ordered list so that these can be passed on to the correct worker process.

```
distributeLists tasks reqs = [taskList reqs tasks n | n <- [1..noPe]]
  where taskList (r:rs) (v1:vs1, v2:vs2) pe
          | pe == r   =  (v1, v2) : (taskList rs (vs1, vs2) pe)
          | otherwise =  taskList rs (vs1, vs2) pe
        taskList _ _ _ = []
```

Modifying the GAP functions: For the linear solver, the results need to be ordered so that they can be combined correctly. Since not all instances of the multiple homomorphic pattern will require this ordering, we offload it to the relevant GAP functions. We therefore also need to modify the GAP function `ModMat` so that it returns a pair of the solution vector and the prime. This pair is implicitly passed through the skeleton, so that in the combination phase, the `CRAMat` function first sorts the pairs by their prime numbers before computing the Chinese Remainder algorithm. The modified GAP functions are as follows:

```
SolMat:=function(p, x)
 ...
 return([List(r, Int), p]);
end;

CRAMat:=function(ps, r)
local x, pris, vecs, res;
pris:=[];
vecs:=[];
for x in r do
 if not x = [] then
    pris:=Concatenation(pris,[x[2]]);
    vecs:=Concatenation(vecs,[x[1]]);
 fi;
od;
res:=ChineseRem(pris, vecs);
return(res);
end;
```

Performance: In order to demonstrate the effectiveness of the parallel performance of the skeleton, we tested it on several sample matrices of 200 x 200 17-digit elements. Experimentation shows a 1.795 speedup on 2 cores over the original sequential version of the skeleton discussed in Section 3.2. Clearly, these early results show that the skeleton can be used to increase the performance of the linear solver; and, more importantly, the speedup shows that parallel skeletons can be an effective way of increasing the performance of computer algebra systems. Indeed, our domain-specific *orbit* skeleton has previously shown a maximum speedup of 8.295 on eight cores [16].

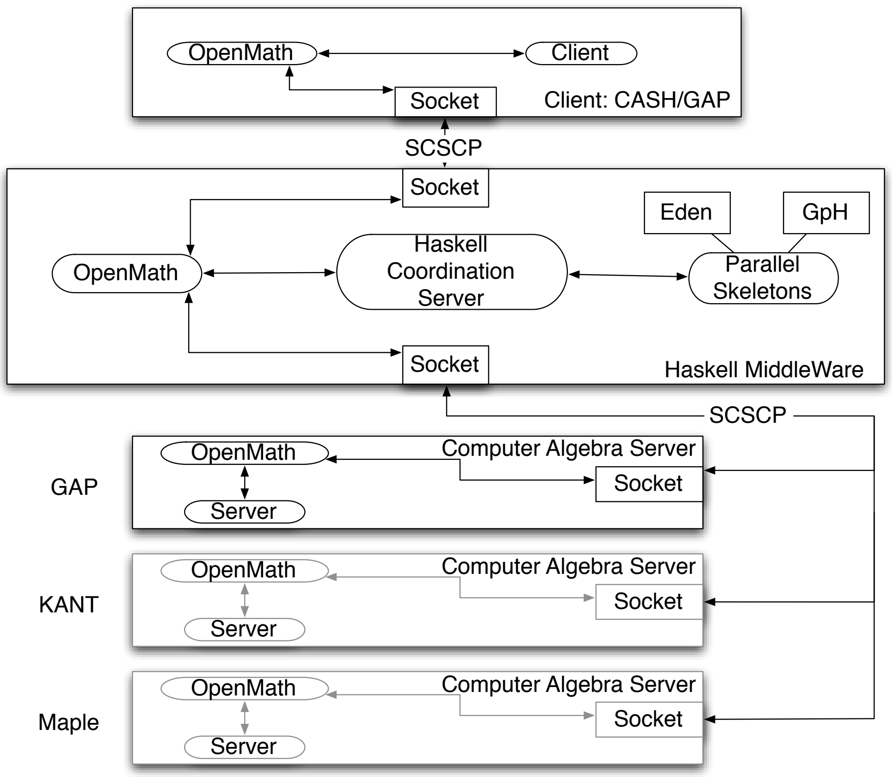

Fig. 4. SymGrid-Par System Architecture

4.1 SymGrid-Par

While this paper focuses on the interface between Haskell and computer algebra systems, the full power of Haskell as a coordination language only becomes apparent when we consider how large-scale symbolic computations can be coordinated. SymGrid-Par (Figure 4) has been designed to achieve a high degree of flexibility in constructing a platform for high-performance, distributed symbolic computation, including computer algebra systems. It has 3 main components:

– **The Client**. The end user works in his/her own familiar programming environment, so avoiding needing to learn a new computer algebra system, or a new language to exploit parallelism. We can imagine here the client being CASH, connecting through the Haskell middleware coordination layer to the computer algebra systems. The coordination layer is completely hidden from the CASH end users, and they work exactly as they would dealing directly with computer algebra systems. For example, the CASH client could send an SCSCP request to the coordination server to use the multiple homomorphic images skeleton.

- **The Coordination Server**. This middleware provides parallelised services and parallel skeletons to the client. The client may invoke these skeletons as standard higher-order functions. The Coordination Server then delegates work (usually calls to expensive computational algebra routines) to the Computation Server, which is another SCSCP-compliant computer algebra system. Currently the Coordination Server is implemented in Haskell, allowing the user to exploit polymorphism, purity and higher-order functions for effective implementation of high-performance parallelism. In this example, the Coordination Server executes the Eden multiple homomorphic images skeleton, creating GAP instances as separate worker processes.
- **The Computation Server**. This component is typically a dedicated computer algebra system, e.g. GAP. Each server handles the requests that are sent to it, sending the results back to the coordination layer for processing. Finally, the coordination layer returns the result to the client.

Currently, all communication between the components uses the SCSCP interface to maximise flexibility and interoperability. In future, we could adapt this to call computer algebra functions directly, so avoiding the marshalling and communication overhead. This could be useful on multi-core machines, whereas the generic interface is better suited for large-scale distribution. Such a design is discussed in [17].

4.2 Calling Haskell from GAP

Using the SymGrid-Par architecture, which provides an SCSCP server with a collection of parallel skeletons, we can now use our interface to call parallel Haskell code from a GAP client. In order to call the Multiple Homomorphic Images skeleton from GAP, we first install it as an SCSCP service called `CS_MHI`. We then create a GAP wrapper that calls this parallel SCSCP service:

```
ParMHI:=function( mod_fct, sol_fct, cra_fct, primes, matvec )
local res;
res:=EvaluateBySCSCP("CS_MHI",
                     [mod_fct, sol_fct, cra_fct, primes, matvec],
                     SCSCPclientHost, SCSCPclientPort);
return res.object;
end;
```

In the GAP shell, we can then simply call this SCSCP wrapper as follows:

```
gap> ParMHI("WS_Mod", "WS_Sol", WS_CRA", Primes, [m,v])
```

Here, `Primes` is the list of prime numbers passed to the algorithm and `m` and `v` are GAP representations of a matrix and a vector that need to be solved. This is only an example of the power of the approach, of course: in addition to *domain-specific skeletons* such as this, we also provide a number of general-purpose skeletons such as parallel maps and folds that can be freely used by the computer algebra user.

5 Related Work

We are aware of two previous attempts to link computer algebra systems with a general-purpose functional language. The GHC-Maple [11] system provides a bespoke interface between sequential Haskell and the Maple heap, using an internal, string-based representation of mathematical objects in Maple. The Eden-Maple system [12] builds on this interface, using Haskell's foreign function interface (FFI) to link Maple with the parallel Eden system. The main difference to CASH is that both these systems are restricted to one computer algebra system, using an interface that is specific to that system. They are therefore fragile to change. In the wider arena, one early system that used a declarative language to coordinate parallel Maple programs was ||MAPLE|| [18]. This used the guarded horn clause language Strand to coordinate parallel Maple computations, executing on networks for workstations. Another system enabling parallelism for Maple is Distributed Maple [19], which provides primitives for explicit thread creation and synchronisation, as well as non-determinism and speculation. Compared to the above approaches, it provides a lower level of coordination with explicit threads. In contrast to the approach we have taken here, of *offloading* computer algebra to dedicated systems, DoCON, the Algebraic Domain Constructor [20] aims to provide a full computer algebra system for students, using a Haskell implementation. At present, DoCON provides only a small subset of the functionality of computer algebra systems such as GAP, mainly providing support for linear algebra, polynomial GCD and Gröbner bases.

GAP itself has been parallelised with the ParGAP [21] package, using the MPI message passing library. To ease parallel programming, it builds higher-level abstractions over the MPI layer, for example a `ParList` skeleton that computes a function in parallel over a list. Work is still underway. The latest development in the open source GAP system is to make the GAP kernel itself parallel [2], by providing both high and low levels of parallel abstraction for the end GAP user. However, in contrast to CASH, all these approaches are specific to GAP, and cannot exploit the existing large body of work on parallelism in Haskell.

Finally, the recently revised Numeric Prelude [22], is a Haskell library that contains many complex mathematical representations together with those most commonly found in computer algebra systems. For example, it contains support for groups, rings, domains, fields, lattices, monoid, polynomials and basic matrix manipulation. In contrast to our design, the numeric prelude does not offer a full system for computer algebra and tends to focus on numerical rather than symbolic computation. In the longer term we intend to support the evolving class hierarchy that is being developed as part of the Numeric Prelude to simplify our Haskell-side usage of symbolic computations, while maintaining the SCSCP link to computer algebra systems.

6 Conclusions

By connecting SymGrid-Par to the CASH front-end client, using the generic SC-SCP interface, we have allowed computer algebra systems to be easily combined with functional programming. This makes the existing rich repertoire of complex and efficient library functions over mathematical data structures, with all the mathematical knowledge that has been captured in them, easily available to the Haskell programmer. It also makes Haskell available to computer algebra users. One particular advantage of this is that parallelism can be much more easily expressed in Haskell, using any of the available parallel Haskell implementations. We have used this in the Coordination Server component of SymGrid-Par, showing here how the Eden distributed memory implementation can be exploited to parallelise the new, domain-specific multiple homomorphic images skeleton.

There are several possible avenues for future work, both for SymGrid-Par and for CASH itself. Firstly, in order to simplify the interface for the CASH user, we intend to build on the evolving Haskell class hierarchies that are being developed as part of the Numeric Prelude [22]. We are also in the process of extending the library of parallel skeletons that are available to the CASH user by implementing more domain-specific skeletons for computer algebra, and conversely are consolidating client-side functionality by defining and revising content dictionaries for particular computer algebra sub-domains [23]. Finally, we are promoting the SC-SCP protocol in order to increase the number of computer algebra (and other) systems that can be exploited in this way.

This paper represents a first, but important, step in removing the walls of the ghetto between functional programming and computer algebra systems. We are pleased by our initial results and believe that there are significant benefits to both communities both from this development, and from the future cooperations that this will enable.

Acknowledgements

This work has been generously supported by the EU Framework VI SCIEnce project (Symbolic Computation Infrastructure in Europe, RII3-CT-2005-026133), and by the UK's Engineering and Physical Sciences Research Council (HPC-GAP: High Performance Computational Algebra and Discrete Mathematics, EP/G 055181).

Downloading CASH

CASH is available to download from Hackage at `http://hackage.haskell.org`, or with the Cabal installer [24] using `cabal install cash`.

References

1. Peyton Jones, S., Hammond, K.: Haskell 98 Language and Libraries, the Revised Report. Cambridge University Press (December 2003)
2. The GAP Group: GAP – Groups, Algorithms, and Programming, Version 4.4.12. (2008) `http://www.gap-system.org`.
3. Daberkow, M., Fieker, C., Klüners, J., Pohst, M., Roegner, K., Schörnig, M., Wildanger, K.: KANT V4. J. Symb. Comput. **24**(3/4) (1997) 267–283

4. Morisse, K., Kemper, A.: The Computer Algebra System MuPAD. Euromath Bulletin **1**(2) (1994) 95–102
5. Char, B.W.: Maple V Lang. Ref. Manual. Maple Publ., Waterloo Canada. (1991)
6. Grayson, D.R., Stillman, M.E.: Macaulay 2, a Software System for Research in Algebraic Geometry. Available at http://www.math.uiuc.edu/Macaulay2/
7. Freundt, S., Horn, P., Konovalov, A., Linton, S., Roozemond, D.: Symbolic Computation Software Composability. In: AISC/MKM/Calculemus 2008. LNCS, vol.5144, Springer, Berlin/Heidelberg (2008) pp.285–295
8. Abbott, J., Díaz, A., Sutor, R.S.: A Report on OpenMath: a Protocol for the Exchange of Mathematical Information. SIGSAM Bull. **30**(1) (1996) 21–24
9. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel Functional Programming in Eden. J. Func. Prog. **15**(3) (2005) 431–475
10. A. Al Zain and K. Hammond and P. Trinder and S. Linton and H-W. Loidl and M. Costantini: SymGrid-Par: Designing a Framework for Executing Computational Algebra Systems on Computational Grids. In: ICCS'07. LNCS, vol.4488, Springer, Berlin/Heidelberg (2007) pp.617–624
11. Schreiner, W., Loidl, H.W.: The GHC-Maple Interface. On-line Documentation (2000) `http://www.risc.jku.at/software/ghc-maple/`.
12. Martínez, R., Pena, R.: Building an Interface Between Eden and Maple. In: IFL'03. LNCS, vol.3145, Springer, Berlin/Heidelberg (2004) pp.135–151
13. Lauer, M.: Computing by Homomorphic Images. In: Computer Algebra — Symbolic and Algebraic Computation. Springer-Verlag (1982) pp.139–168
14. Lipson, J.D.: Chinese Remainder and Interpolation Algorithms. In: Symp. on Symbolic and Algebraic Manipulation, Academic Press (1971) pp.372–391
15. Klusik, U., Loogen, R., Priebe, S., Rubio, F.: Implementation Skeletons in Eden: Low-Effort Parallel Programming. In: IFL '00. LNCS, vol.2011, Springer Berlin/Heidelberg (2001) pp.71–88
16. Brown, C., Hammond, K.: Ever-Decreasing Circles: a Skeleton for Parallel Orbit Calculations in Eden. In: TFP'10 – Draft Proceedings, Oklahoma, US (May 2010)
17. SCIEnce Team: SymGrid-Par: Parallel Orchestration of Symbolic Computation Systems. In: ISSAC10, Munich (July 2010) Software Demonstration.
18. Siegl, K.: Parallelizing Algorithms for Symbolic Computation Using ||MAPLE||. SIGPLAN Not. **28**(7) (1993) pp.179–186
19. Schreiner, W., Mittermaier, C., Bosa, K.: Distributed Maple: Parallel Computer Algebra in Networked Environments. J. of Symb. Comp. **35**(3) (2003) pp.305–347
20. Mechveliani, S.D.: The Haskell Functional Language and Computer Algebra. In: Proc. of Program Systems. (2003) pp.56–64
21. Cooperman, G.: Parallel GAP: Mature Interactive Parallel groups and Computation. Technical report, College of Comp. Sci., Northeastern Univ. (2001)
22. Thurston, D., Thielemann, H., Johansson, M.: Numeric prelude (0.2). http://www.haskell.org/haskellwiki/Numeric_Prelude (2010)
23. Linton, S., Hammond, K., Konovalov, A., Al Zain, A., Trinder, P., Horn, P., Roozemond, D.: Easy Composition of Symbolic Computation Software: A New Lingua Franca for Symbolic Computation. In: ISSAC10, ACM (2010) pp.339–346
24. Jones, I.: The Haskell Cabal: A Common Architecture for Building Applications and Libraries. In van Eekelen, M., ed.: TFP'05, Intellect (2006) pp.340–354