A First Report on the $A^\sharp$ Compiler

Stephen M. Watt Peter A. Broadbery Samuel S. Dooley
Pietro Iglio Scott C. Morrison* Jonathan M. Steinbach Robert S. Sutor

IBM Thomas J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598 USA
{smwatt,peteb,dooley,iglio,jonms,sutor}@watson.ibm.com

1 INTRODUCTION

The $A^\sharp$ compiler allows users of computer algebra to develop programs in a context where multiple programming languages are employed. The compiler translates programs written in the $A^\sharp$ programming language [24][25] to a low-level intermediate language, Foam [26], from which it can generate stand-alone programs, native object libraries to be linked with other applications, or code to be read into closed environments. In addition, Foam code may be directly executed using an interpreter provided with the $A^\sharp$ compiler.

The $A^\sharp$ programming language provides support for object-oriented and functional programming styles. It is "higher-order" in the sense that both types and functions are first class, and may be manipulated in the same ways as any other values. The primary considerations in the formulation of the language have been generality, composibility, and efficiency. The language has been designed to admit a number of important optimizations, allowing compilation to machine code which is in many instances of efficiency comparable to that produced by a C or Fortran compiler.

The original motivation for $A^\sharp$ comes from the field of computer algebra: to provide an improved extension language for the Axiom computer algebra system [14].

Relation to work in computer algebra

From a computer algebra perspective, the most interesting aspects of the $A^\sharp$ compiler are that

- it allows programmers to develop code which can participate naturally in both C and Lisp based environments,
- the programmer can decide which primitives must be most efficient and obtain optimized code for them, and
- serious consideration has been given to the efficiency of both symbolic and floating point computation.

We put these developments in context to show why we have taken the present approach.

The first computer algebra software took the form of libraries, intended for use from Fortran or Lisp [5][10][11]. Shortly thereafter, special-purpose systems were introduced to allow computational input to be given in a more natural form. Some of these were programming languages backed by libraries [4][6], while others also provided an interactive programming environment [12][13]. These algebra systems were *closed*, in the sense that it was not readily possible to call code written in them from another programming language environment, such as Fortran, or *vice versa*.

The use of closed systems has predominated over the course of three decades, and the computer algebra software in use today is of this form [7][9][13][14][17][20][23][28]. It remains the case that it is not possible to call programs written in one of these systems from programs written in another. Nor is it possible to use general Fortran or C libraries without resorting to some relatively expensive communication scheme.

One consequence of this is that a great deal of effort is expended in reproducing the same programs in different computer algebra systems, or reproducing the function of other libraries within these systems. While there will always be cases where similar packages are developed independently to establish legal ownership for commercial development, it is regrettable that researchers must reproduce their programs in several different systems to make them available to their colleagues.

There have been a number of proposals to improve on this situation with structured data communication between processes [16][29][30]. This is only a partial solution, however, since IPC-based methods incur a significant communication overhead which can easily dominate the cost of $O(n)$ problems.

Another partial solution has been to build special-purpose interfaces to particular libraries. While this is useful for the particular cases handled, it does not accomodate new or third party libraries as they become available.

The $A^\sharp$ compiler presents an alternative approach to the problem: sharing code.

$A^\sharp$ libraries participate on an equal footing with other libraries in the host environment. Functions in $A^\sharp$ libraries may call and be called by functions in other libraries in the normal way and it is possible to share data in forms required by other programs. This makes it possible, for example, to use libraries for numeric problem solving or scientific visualization.

This approach is standard practice in the larger world, but has been abandoned for decades in mainstream computer algebra software. This discrepancy is perhaps the result of at least two pressures: users have come to appreciate the value of interactive environments for computer algebra,

*Present address: Autodesk, Multimedia Division, 2320 Marinship Way, Sausalito, CA 94965 USA.

pp. 25-31 *Proc. International Symposium on Symbolic and Algebraic Computation (ISSAC 94)*, 20-22 July 1994, Oxford UK.

and the languages incorporated in computer algebra systems have allowed programs to be written more conveniently than the alternatives. With $A^\sharp$, however, a user may prototype an application within an interactive environment, such as the Axiom system, or $A^\sharp$ with the Foam interpreter, and later generate code to be linked with other applications. Furthermore, it is not unreasonable to anticipate running the same $A^\sharp$ code in "closed" systems by taking the approach used with Lisp: from Foam generating source code to load into Maple, for example.

A second problem with closed systems is that certain computations will be much slower than they need to be. This arises because these systems are typically structured as a *kernel*, written in another programming language, and a *library* providing an assortment of additional functions. The operations coded directly in the kernel can be two or three orders of magnitude more efficient than the same operations written at the library level. It is not possible to know in advance what the efficiency-critical primitives will be for the complete spectrum of mathematics so key primitives for new application areas will likely be missing from the system kernel. These will have to be written at the library level, with the associated slow-down. A case in point is GAP [20], a system very similar to Maple, but with a focus on group theoretic computation. GAP was initially written because the necessary kernel-level efficiency for primitives in group computations could not be achieved by writing Maple library code.

The $A^\sharp$ compiler addresses this issue by giving the programmer the ability to develop optimized code for whatever primitives are required.

In practice, these problems are not as severe in Lisp-based systems [13][14][23]. Users of Lisp-based algebra systems may make their own kernel extensions by writing and compiling Lisp code. If the underlying Lisp system supports it, an implementation-specific foreign function interface can provide a mechanism to access C or Fortran libraries.

This direction, however, has drawbacks: In practice, to use a foreign library from Lisp it is necessary to cover the foreign functions with wrappers to convert between the Lisp and the foreign data representations. This may involve copying on each call and return and introduces semantic difficulties when considering sharing of substructure and pointer equality. Next, the Lisp foreign function interface will have a fixed vocabulary of understood data transformations and will not usefully pass function values, file pointers, or other interesting composite objects. Finally, it is necessary to ensure that the values passed to foreign functions do not relocate (e.g. by garbage collection) during the course of execution, and any pointers captured by the foreign functions migth be invalid in subsequent calls. This makes it very difficult to write code to embed in non-Lisp applications, since the controlling side will retain state between call-backs. Because of these difficulties, Lisp foreign function interfaces do not promise a great improvement over IPC methods as a means of accessing libraries.

At this point, one might ask why not move to stand-alone programs based on a mathematical library for C++ [22] or some other programming language. Our response is twofold: First, we felt that existing object systems were not capable of modelling the intricate relations among algebraic structures sufficiently naturally. Second, it is valuable to maintain a path of access to the existing computer algebra software base.

Relation to work in compilers

From a compiler implementation perspective, the most interesting aspects of the $A^\sharp$ compiler are:

- type inference in the context of overloaded identifiers and type constructors, with types being first class values,
- the combination of cross-file inlining and data structure elimination to reduce overhead
- the use of dataflow analysis to reduce flow graphs produced by constructs such as generators
- efficient implementation of types as run-time values,
- storage management performed through a run-time conservative garbage collector [2], and
- portability across many hardware platforms and operating systems.

As far as we are aware, the first three of these items are new. The remaining items have novel aspects.

2 COMPILER ORGANIZATION

External characteristics

The $A^\sharp$ compiler accepts several forms of source and can potentially produce multiple files, each containing a different representation of the program.

The most common use of the compiler is to take a source file and produce a platform-independent binary object "`aso`" file. The `aso` file contains types and documentation for exports, intermediate code, and other information. It is possible to produce C, Lisp or Foam files from an `aso` file. It is also possible to create a platform-specific object file or a symbol table file. The symbol table file contains human-readable type and documentation information in a form suitable for use by other programs. Any combination of these files can be produced in a single invocation of the compiler.

The generated C or Lisp code is designed to be used either by itself or as part of a larger software system. The generated code uses a judicious selection of macros for which sets of default definitions are provided.

The compiler has an additional interface to allow interactive investigation of compile-time errors. This has turned out to be very convenient for pinpointing errors with heavily overloaded operations. The information needed to fix the error can be located quickly since it is not lost in a flood of preemptive detail. The $A^\sharp$ compiler can also generate error messages in a special format so that it can interact with other programs. An X-Windows/Motif tool is supplied for inspecting compile-time errors in that graphical user interface environment.

A full description of the external characteristics of the compiler is given in the user's guide [25].

Internal structure

The compiler works in several passes, which may be broadly grouped as

- *syntactic analysis*: source inclusion, lexical analysis, parsing, macro expansion,
- *semantic analysis*: scope binding, type inference,
- *intermediate Foam code generation*,
- *intermediate code optimization* (several individual passes),
- *concrete code generation*: to C or Lisp.

The compiler can generate or accept files between any of these groups. For example, when machine-generating $A^\sharp$

programs it is possible to bypass the syntax analysis and pass generated abstract syntax trees into the semantic analysis phase.

The $A^\sharp$ compiler is about 120,000 lines of C code and 6000 lines of $A^\sharp$ code in its current implementation. About 1% of the compiler is code for Lisp generation and about 3% is code for C generation. Some code is shared between the general library and the default $A^\sharp$ run time system. Most of the $A^\sharp$ run-time system is written in $A^\sharp$, with the rest written in C or Lisp depending on the platform.

3 MAJOR COMPONENTS

Type inference

The largest single phase of the compiler is the type inference phase. As implemented in the $A^\sharp$ compiler, type inference determines a unique interpretation for every identifier in the source text, and assigns a unique type for every node in the abstract syntax tree. If this is not possible, the reasons are reported.

The algorithm which is responsible for making these assignments is complicated by the following features of the $A^\sharp$ type system:

- functions, domains, and categories as first-class values;
- parameterized domain and category constructors, parameterization of functions over domains, categories, other functions, and data values;
- dependent function types;
- categorical inheritance;
- categorical default implementations;
- conditional declarations and definitions;
- overloaded identifier names;
- nested function definitions;
- recursion in the types of domains and categories; and
- domain and category extension definitions.

A "domain" corresponds closely to an abstract type or class in other languages. For an explanation of these features the reader is referred to the language reference [24].

Briefly, the type inference algorithm proceeds in two major passes. When presented with an abstract syntax tree, the first pass collects the possible interpretations for each leaf node in the tree, and propagates a set of possible types, from the bottom up, for each node in the tree. The second pass traverses the tree to propagate type constraints which are inherent in the structure of the program, and uses these constraint types to select a unique interpretation for each leaf node. As the traversal unwinds back toward the root of the tree, a unique type is then computed for each node in the tree.

Interpretations for identifiers which appear in a program may be found from declarations in the current lexical scope in which the identifier appears, or may be found by *importing* definitions from (possibly parameterized) domains which are defined elsewhere. Definitions available via an `import` statement are visible throughout the lexical scope in which the `import` statement appears. Therefore the interpretation of each type which is imported in the current scope must be known before the set of possible meanings for an identifier can be collected. As a result, the bottom-up pass of the type inference phase, upon entry into a given lexical level, recursively invokes the type inference algorithm on the set of types which are used in that level.

The definitions of domains and categories in $A^\sharp$ may involve the inheritance of definitions from other domains and/or categories. As a result it is often the case that the interpretation of one domain D must be completed before the interpretation of another can be determined. These dependencies may be recursive, as in the following declaration:

```
Ladder(D: with (f: E->E), E: with (g: D->D)): with
        Ladderf: E -> E
        Ladderg: D -> D
== add
        import from D;
        import from E;
        ...
```

The parameters `D` and `E` of `Ladder` are domains, each of which exports an operation, whose type involves the other parameter. Within the definition of `Ladder` the definitions of `f` and `g` are available, and can be used to define `Ladderf` and `Ladderg`. The types of `D` and `E` are mutually-recursive. These must be determined by a process of simultaneous type inference which interleaves the usual bottom-up and top-down phases on each set of mutually-recursive types to determine unique interpretations.

The inference algorithm will be described in more detail in another paper.

Abstract Machine

A major part of the $A^\sharp$ compiler is concerned with producing optimized intermediate code, or Foam code. "Foam" is an acronym for "First Order Abstract Machine." The abstract machine is first order in the sense that it does not treat its types as values.

Foam is designed to contain only those concepts which can have an efficient realization in both Lisp and C. For example it is not possible to take an address of a variable because that would be inefficient in Lisp (a closure would be created). Nor are dynamic type tests allowed, as that would be inefficient in C. We have been asked how the lack of address arithmetic limits the potential performance of compiled $A^\sharp$ *vs* hand-coded C which uses pointers to traverse arrays in inner loops. It is our experience that this is a minor concern on current architectures with optimizing compilers.

Foam is not restricted to the precise intersection of C and Lisp. Some aspects are handled by support libraries. Big integer arithmetic is assumed as part of Foam, and this is provided as a library for C. Also the memory model differs from both C and Lisp in some details: garbage collection is assumed (this is a run time support library in C) and it is possible to make an explicit request to free storage (in Lisp this is ignored).

A Foam program is comprised of a flat sequence of commands. Foam types have various sizes and uses. or example, "`Char`" is a text character whereas "`Byte`" is a character sized integer, "`DFlo`" is a double precision floating point, "`Ptr`" can point to an array, record, arbitrary sized integer, etc. Reference instructions contain the kind of reference and the position, e.g., "`Loc 3`" refers to the third local variable of the current function and "`RElt 7` x `2`" indicates the 2nd field of the record x, using the 7th layout format. Foam operations consist of instructions, such as "`If` b n," which indicates that if b is true then proceed to label n, and builtin operations, e.g., "`HIntLT` a b" is a half-word-integer less-than comparison. The builtin operations are type specific and conversion operations are generally provided. A detailed description of Foam is given elsewhere [26].

The abstract machine does not support asharp types directly and relies on the code generator to produce appropriate calls to create and maintain types. This has the advantage that one can use these calls to add new representations of types to the system. These representations may be written in $A^\sharp$itself, or some other language. This is used in order to interface with the Axiom system, and may be extended to other object/type systems, such as CLOS [21] and C++.

Optimization

The $A^\sharp$ compiler performs a number of optimizations on the intermediate Foam code. Further optimizations may be performed downstream by a C or Lisp compiler, depending on the environment.

The criterion for deciding which Foam optimizations to include in the $A^\sharp$ compiler has been to select those optimizations which have maximum payoff but cannot be expected from a downstream compiler. These optimizations are:

Program specialization is used to exploit specific instances of generic type constructors. For example, if the generic constructor `Complex` is used as `Complex(DoubleFloat)`, then specialized versions of the complex arithmetic operations are produced at compile-time. On the other hand, if the constructor is used as `Complex(R)`, where `R` is a parameter, then code is generated which binds the operations from `R` and `Complex(R)` at run-time.

Procedural integration (or inlining) is used to eliminate function calls. This is particularly effective where many exported operations are simple combinations of other functions. The compiler performs *cross-file procedural integration* by extracting code bursts from other compiled files. This optimization is only performed when a *client* of a package declares that *it* is willing to depend on the package's implementation.

Data structure elimination converts heap-allocated structures to collections of temporaries, when possible. This optimization often eliminates the need to heap-allocate temporary results. For example, arithmetic operations on rational or complex number structures are converted into sequences of component operations. This optimization allows efficient code without complicated explicit reference accounting on the part of the programmer.

Dataflow analysis of condition variables eliminates computed branches which arise from inlining generators. The analysis identifies generator control nodes and splits them to form reducible flow graphs. This allows programs to use the high-level control abstractions without a performance penalty.

Some other optimizations that would be expected of any downstream concrete-code generator are performed on Foam as prerequisites of other optimizations. These include: *copy propagation*, *dead variable elimination*, *dead code elimination*, *constant folding* (including big integer arithmetic), and various peep-hole optimizations.

Together these optimizations seem quite effective. In particular, the combination of cross-file inlining and data structure elimination produces significant gains for the style of programming encouraged by the language.

While we have not performed a careful comparison, certain high-level examples generate code which is comparable in performance to optimized hand-written C.

4 TARGETS

Presently, only C and Lisp code are generated directly from Foam. By compiling generated C into object code, $A^\sharp$ programs may participate in open software architectures. The $A^\sharp$ compiler treats Lisp systems as if they were closed, in the sense of Section 1. We generate Lisp code from Foam, and use the Lisp system's compiler, if it has one, to generate Lisp-loadable modules.

C generation

C code can be generated in platform-independent K&R [15] or ANSI [1] form. This code uses macros for operations important in efficient arithmetic: double word multiply, division with remainder, multiply-add, etc. If the $A^\sharp$ compiler is used in conjunction with a C compiler which supports `asm` statements, these macros can be redefined to use single machine instructions. With appropriate declarations in the $A^\sharp$ program, the generated C code uses appropriate naming and linkage conventions to allow access to other libraries, for example, the X Window System.

A default run-time library is supplied for storage management, big integer arithmetic, and so on. The storage manager assumes that other storage managers may be operating simultaneously. It allows deallocation of store as well as garbage collection. The garbage collector assumes that pointers may be captured by other programs so it will not collect storage which is referenced by global data, foreign heaps, the stack or registers.

By replacing just one header file, the C code may use other packages for this support.

Lisp generation

The generated Lisp code is heavily abstracted by macros which have access to information that is potentially useful to a Lisp compiler. It would be straight-forward in principle to provide macro implementations for several dialects of Lisp. We currently provide only Common Lisp [21] versions of the macros, Standard Lisp [18] and Scheme [19] were also considered in their design.

Macros are used to provide low-level type information is given for all variables. All operations are implemented in terms of monomorphic macros (for example, fixnum "+" uses a different macro than double precision "+".) The Common Lisp implementation uses all this information to produce fully proclaimed code which a good Lisp compiler can use to avoid all run-time type tests.

As discussed earlier, some Lisp systems do provide mechanisms for linking foreign functions. It would therefore be possible, in principle, to instead generate C code and use the foreign function interfact to link the resulting objects into the Lisp system. This, however, would be highly unportable accross different Lisp systems and would suffer all the drawbacks described in Section 1.

Portability

The compiler development work is regularly checkpointed, and as a foremost consideration, portability is tested against a wide variety of platforms. The $A^\sharp$ source compiles without warnings on many compilers, including: Borland, DEC, Free Software Foundation, IBM, Metaware, and Mips.

Test platforms include several operating systems: DOS, CMS, VMS, OS/2 and many UNIX derivatives, as well as hardware architectures with different word sizes, byte orders, and character and floating point representations. These include: Intel 8086, 80X86, Motorola 680X0, IBM 370, RT PC, and RS/6000, DEC Alpha AXP, Sun SPARC, and MIPS processors. Run-time support is 16/32/64 bit clean.

Platform-independent object files store integers, characters and floating point data in a standard format, which preserves all representable floating point numbers in the native architectures. (Some formats, such as XDR, lose significant bits or exponent range.)

5 EXAMPLE

The following $A^\sharp$ program computes the Mandelbrot set by determining the number of iterations of the function $f(z) = z^2 + c$ required to send each point in a given region of the complex plane to a point outside the circle of radius 2:

```
-- Computation engine for the X mandelbrot program.
#include "aslib.as"

-- Macros
I  ==> SingleInteger
F  ==> DoubleFloat
CF ==> Complex DoubleFloat

maxIters: I == 100

import from CF -- Make operations from CF visible.
inline from CF -- Give optimizer permission to depend on CF.

default ar, br, ai, bi: F
default nr, ni: I

++ Draw the Mandelbrot set in the given region.
++ The pixel drawing function is passed as a parameter.

drawMand(ar,br,nr, ai,bi,ni, draw: (I,I,I,I)->I): () ==

   mandel(c: CF): I ==
      z: CF := 0
      n: I  := 0
      for it in 1..maxIters while norm z < 4.0 repeat
         z := z*z + c
         n := it
      n

   for i in step(ni)(ai,bi) for ic in 0..ni-1 repeat
      for r in step(nr)(ar,br) for rc in 0..nr-1 repeat
         nit := mandel complex(r, i)
         draw(rc, ic, maxIters, nit)
```

The function `drawMand` illustrates several interesting features of the $A^\sharp$ language: domains (the `DoubleFloat` argument to `Complex`) and functional closures (the `draw` parameter to `drawMand`) as first-class objects, domains parameterized over other domains (`Complex(DoubleFloat)`), cross-file inlining, inlining of functions from parameterized domains, and iteration using generator functions (`step(ni)(ai,bi)`) These features are described in more detail in [24].

The non-optimized intermediate Foam code generated from the above program illustrates the character of the abstract machine. The code fragment shown below is produced from the inner loop which steps the variables `r` and `rc` in parallel.

```
(Label 2)
(Set (Loc 10 r) (CCall Word (Loc 13)))
(If (EElt 4 (Loc 12) 0 0 done) 3)
(Set (Loc 9 rc) (CCall Word (Loc 16)))
(If (EElt 4 (Loc 15) 0 0 done) 3)
(Set (Loc 0 nit)
     (OCall Word (Const 2 mandel) (Env 0)
            (CCall Word (Lex 1 18 complex)
                   (Loc 10 r) (Loc 2 i))))
(CCall Word (Par 6 draw) (Loc 9 rc) (Loc 1 ic)
       (Lex 1 1 maxIters) (Loc 0 nit))
(Goto 2)
```

A few words might make this code less cryptic. The instructions `Par` and `Loc` refer to the current function's parameters and temporary variables. They are annotated with the name from the user-level program, when there is one. The `Lex` instruction refers to variables in the lexical environment of the current function. `EElt` refers to variables in other lexical environments. The `OCall` and `CCall` instructions call programs as function-environment pairs or closures respectively. The first parameter in `OCall` and `CCall` special form is the Foam type of the return value. The second argument to the `If` instruction is a branch label.

The optimized Foam code demonstrates the level of efficiency which can be realized by programs written in $A^\sharp$:

```
(If (Loc 33 done) 39)
(Set (Loc 35 f0) (DFlo 4.000000e+00))
(Set (Loc 38 f0) (BCall DFloTimes (Loc 57 f0) (Loc 57 f0)))
(Set (Loc 39 f0) (BCall DFloTimes (Loc 58 f0) (Loc 58 f0)))
(Set (Loc 40 f0) (BCall DFloPlus (Loc 38 f0) (Loc 39 f0)))
(If (Cast Word (BCall DFloLE (Loc 35 f0) (Loc 40 f0))) 39)
(Set (Loc 41 f0) (BCall DFloTimes (Loc 57 f0) (Loc 57 f0)))
(Set (Loc 42 f0) (BCall DFloTimes (Loc 58 f0) (Loc 58 f0)))
(Set (Loc 55 f0) (BCall DFloMinus (Loc 41 f0) (Loc 42 f0)))
(Set (Loc 43 f0) (BCall DFloTimes (Loc 57 f0) (Loc 58 f0)))
(Set (Loc 44 f0) (BCall DFloTimes (Loc 58 f0) (Loc 57 f0)))
(Set (Loc 56 f0) (BCall DFloPlus (Loc 43 f0) (Loc 44 f0)))
(Set (Loc 57 f0) (BCall DFloPlus (Loc 55 f0) (Loc 52 f0)))
(Set (Loc 58 f0) (BCall DFloPlus (Loc 56 f0) (Loc 48 f0)))
(Set (Loc 9) (Loc 11))
(Goto 38)
(Label 39)
(Set (Loc 0 nit) (Loc 9))
(CCall Word (Par 6 draw) (Loc 2 rc) (Loc 1 ic)
       (Lex 1 1 maxIters) (Loc 0 nit))
(Goto 2)
```

The above code fragment is the body of the same loop, after functions (e.g. `mandel`) have been inlined and non-escaping record values (e.g. complex numbers) have been replaced by collections of temporaries.

As might be expected, the concrete C code generated by the back-end of the compiler closely parallels the optimized Foam code:

```
        if (T33_done) goto L39;
        T35_f0 = 4.000000e+00;
        T38_f0 = T57_f0*T57_f0;
        T39_f0 = T58_f0*T58_f0;
        T40_f0 = T38_f0 + T39_f0;
        if ((FiWord) (T35_f0 <= T40_f0)) goto L39;
        T41_f0 = T57_f0*T57_f0;
        T42_f0 = T58_f0*T58_f0;
        T55_f0 = T41_f0 - T42_f0;
        T43_f0 = T57_f0*T58_f0;
        T44_f0 = T58_f0*T57_f0;
        T56_f0 = T43_f0 + T44_f0;
        T57_f0 = T55_f0 + T52_f0;
        T58_f0 = T56_f0 + T48_f0;
        T9 = T11;
        goto L38;
L39:    T0_nit = T9;
        fiCCall4(FiWord, P6_draw, T2_rc, T1_ic,
                 l1->X1_maxIters, T0_nit);
        goto L2;
```

Each identifier is starts with a letter to indicate its kind (e.g., T: local, P: parameter, X: lexical, etc.) and is enumerated to handle name overloading. The statement `fiCCall4` is a macro to call a functional closure which takes four arguments. It takes the function return type, then the closure value, followed by the four function parameters.

The corresponding Lisp code is similar and is omitted for brevity. Macros are used for the various primitive operations to allow the Lisp compiler to generate better code. For example, the multiplications in this example would be generated as calls to "`DFloTimes`," which is defined for Common Lisp as

```
(defmacro |DFloTimes| (x y)
   `(the |DFlo| (* (the |DFlo| ,x) (the |DFlo| ,y))))
```

and `DFlo` is defined using `deftype`.

6 BENCHMARKS

The table below shows a few preliminary benchmarks obtained from $A^\sharp$, and compares them with equivalent programs in Axiom and C. The $A^\sharp$ figures were obtained for compilation to both C and Lisp target environments.

Program S1 is a symbolic computation benchmark, performing polynomial arithmetic to compute a Hilbert function of a monomial ideal [3]. This test compares $A^\sharp$ on Lisp and C bases against the same computation in Axiom. This test is not applicable to C so no figures appear in those positions in the table.

Program N1 is a numeric benchmark computing a 600 by 600 region of the Mandelbrot set using a naive method. The $A^\sharp$ version of the code is that given in Section 5. The Axiom version is a simple transcription of this program. The C version is carefully written, with the complex arithmetic expanded by hand to operations on the real and imaginary parts.

Platform	S1-unopt	S1-opt	N1-unopt	N1-opt
Axiom	—	8.5	—	2230.0
$A^\sharp$ (Lisp)	21.0	4.0	>4000.0	48.0
$A^\sharp$ (C)	7.2	1.4	847.0	9.7
C	—	—	11.7	7.9

For $A^\sharp$ and C, the tests have been run both with and without compiler optimizations in effect. In Axiom the optimizer is always on so no entries appear in the "unopt" columns for that row.

All times are in seconds, measured on an IBM RS/6000 model 530E. The same XLC C compiler was used for both the C and $A^\sharp$(C) timings and the same AKCL Lisp environment was used for both Axiom and $A^\sharp$(Lisp). The Axiom time includes garbage collection, and excludes time needed to initialize the domains and packages used by the program. The $A^\sharp$ time includes garbage collection, as well as initializations. An extended version of this paper provides the precise benchmark examples [27].

As can be seen from the above, both Axiom and $A^\sharp$ generate suboptimal code for the test N1. The $A^\sharp$ compiler is then able to transform this code into equivalent code that runs at nearly the same speed as carefully written C.

Acknowledgements

The authors would like to thank the many other people who have contributed one way or another to the current and previous compiler implementations. Earlier, experimental implementations had significant contributions from William Burge, Marc Gaetano, Michael Monagan, Simon Robinson, Knut Wolf and others. Gerald Baumgartner contributed to the current implementation in the summer of 1993.

The authors would also like to those who have taken the time to β-test the compiler, and Barry Trager for providing the S1 benchmark example.

References

[1] *American National Standard Programming Language C,* ANSI X3.159-1989, American National Standards Institute, 1989.

[2] H.J. Boehm and M. Weiser, *Garbage collection in an Uncooperative Environment*, Software Practice and Experience, September 1988.

[3] Bigatti, Caboara, Robbiano, *On the Computation of Hilbert-Poincare Series,* AAECC vol 2, Jan 1991, pp 21–33.

[4] S.R. Bourne and J.R. Horton, *The CAMAL System Manual,*, Computer Laboratory, Cambridge, 1971.

[5] W.S. Brown, *The ALPAK System for Nonnumerical Algebra on a Digital Computer,* Bell Systems Tech. Journal, Murray Hill, 1963.

[6] W.S. Brown, *ALTRAN User's Manual,* Bell Laboratories, Murray Hill, 1973.

[7] J. Cannon and C. Playoust, *An Introduction to MAGMA* University of Sydney, 1993.

[8] B.W. Char, K.O. Geddes, W.M. Gentleman and G.H. Gonnet, *The design of Maple: A compact, portable, and powerful computer algebra system,* in Proc. EUROCAL '83, Springer Verlang LNCS 162, 1983.

[9] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan and S.M. Watt, *Maple V Language Reference Manual,* Springer Verlag, 1991.

[10] G.E. Collins, *PM, A System for Polynomial Manipulation,* Communications of the ACM 9, 1966.

[11] G.E. Collins, *The SAC-1 System: An Introduction and Survey,* ACM, Proceedings of the 2nd Symposium on Symbolic and Algebraic Manipulation, 1971.

[12] C. Engelman, *MATHLAB: A Program for On-line Assistance in Symbolic Computations,* Proceedings of FJCC, 1965.

[13] A.C. Hearn, *Reduce 3 User's Manual, version 3.3* Rand Corporation, 1987.

[14] R.D. Jenks and R.S. Sutor, *Axiom — The Scientific Computation System,* Springer Verlag, 1992.

[15] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language,* First edition, Prentice Hall, 1978.

[16] J. Purtilo, *A Software Interconnection Technology,* UMCP, Computer Science Department TR-2139, 1988.

[17] A. Rich and D. Stoutemyer, *DERIVE Reference Manual,* SoftWarehouse, 1992.

[18] J. Marti, A.C. Hearn, M.L. Griss, and C. Griss, *Standard LISP Report,* SIGPLAN Notices 14, 1979.

[19] Guy L. Steele Jr.and G.J. Sussman, *The Revised Report on SCHEME: A Dialect of LISP,* MIT Artificial Intelligence Lab Memo 452, 1978.

[20] M. Schönert, *GAP - Groups, Algorithms, and Programs,* Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, Third edition, 1993.

[21] Guy L. Steele Jr. *Common Lisp: The Language,* Second edition, Digital Press, 1990.

[22] Bjarne Stroustrup, *The C++ Programming Language,* Second edition, Addison-Wesley, 1991.

[23] *MACSYMA Reference Guide,* Symbolics Inc, 1985.

[24] S.M. Watt, *A♯ Language Reference, V 0.35*, IBM Research Report 19530, 1994.

[25] S.M. Watt, P.A. Broadbery, S.S. Dooley, P. Iglio, S.C. Morrison, J.M. Steinbach and R.S. Sutor, *A♯ User's Guide,* NAG Ltd, 1994.

[26] S.M. Watt, P.A. Broadbery, P. Iglio, S.C. Morrison and J.M. Steinbach, *Foam: A First Order Abstract Machine, V 0.35*, IBM Research Report RC 19528, 1994.

[27] S.M. Watt, P.A. Broadbery, S.S. Dooley, P. Iglio, S.C. Morrison, J.M. Steinbach and R.S. Sutor, *A First Report on the A♯ Compiler (including benchmarks),* IBM Research Report RC 19529, 1994.

[28] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer,* Second edition, Addison-Wesley, 1991.

[29] *Mathlink*, Wolfram Research Inc.

[30] *OpenMath*, Workshop notes, ETH Zürich, 1993.