A Quantitative Study of Reductions in Algebraic Libraries

Yue Li
Texas A&M University
College Station, TX, USA
yli@cse.tamu.edu

Gabriel Dos Reis
Texas A&M University
College Station, TX, USA
gdr@cse.tamu.edu

ABSTRACT

How much of existing computer algebra libraries is amenable to automatic parallelization? This is a difficult topic, yet of practical importance in the era of commodity multicore machines. This paper reports on a quantitative study of reductions in the AXIOM-family computer algebra systems. The experiment builds on the introduction of *assumptions* in OpenAxiom. It identifies a variety of reductions that are candidate for implicit concurrent execution. An assumption is an axiomatic statement of an algebraic property. We hope that this study will encourage wider adoption of axioms, not just for the purpose of expression simplification and provably correct libraries, but also to enable derivation of implicit concurrency in a scalable fashion.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; I.1.3 [**Symbolic and Algebraic Manipulation**]: Languages and Systems—*Special-purpose algebraic systems*

General Terms

Algorithms, Design, Languages

Keywords

Parallel reduction, assumptions, computer algebra, OpenAxiom

1. INTRODUCTION

Reduction is a standard operation in computational algebra. For instance, the content of an univariate integer polynomial $P \in \mathbb{Z}[x]$ of the form $P(x) = \sum_0^n a_n x^n$ is the greatest common divisor of all its coefficients [2]:

$$\text{cont}(P) = \text{greatest-common-divisor}(a_0, a_1, ..., a_n).$$

Equivalently, it is the reduction of the monoid operator gcd over the coefficients of P. Furthermore, the computation can be arranged in a divide-and-conquer fashion, say

$$\text{cont}(P) = \text{combine}(\text{gcd}(a_0, a_1), \dots, \text{gcd}(a_{n-1}, a_n)),$$

where operator "combine" further computes the greatest common divisor of the results produced by each pair. The above expression may be evaluated concurrently by distributing pairs of polynomial coefficients to different computation engines, and then combine the results, which is known as parallel reduction pattern.

Detecting reduction patterns in large and typed programs presents interesting challenges. A programmer can discover implicit concurrency by observation, and further rewrite them with parallelism. However, such manual investigation is not scalable. On the other hand, automating detection of reduction pattern is also not trivial. A static analysis tool needs semantic information to compute facts underlying reductions. The process is much harder for reductions of user-defined operators. The principal reason is that most users write algorithms in impoverished programming languages, thereby leaving out essential semantic information.

This paper suggests a solution via an extension to an existing computer algebra programming language. We take OpenAxiom [8] as our experimental platform. We extend it with a new construct called *assumption*. An *assumption* is an axiomatic statement of mathematical properties. For instance:

```
forall(T: EuclideanDomain)
  assume MonoidOperator(T, gcd) with
    neutralValue = 0$T
```

That *assumption* states that gcd is a monoid operator over Euclidean domains, admitting 0 as neutral value. `MonoidOperator` is the name of the Spad category defining that property. We note that in the current system, we do not attempt to prove assumptions. As the keyword `assume` indicates, we just take the assumptions on faith, as if they were axioms.
The contributions of this paper include:

1. A quantitative study of reductions in the OpenAxiom algebras. We analyze the complete set of OpenAxiom algebra library, and show rich parallelization opportunities brought by monoid operator reductions.

2. A static analysis tool for reduction detection. The detector utilizes the algebraic properties of operators specified via user assumptions to identify implicit concurrency in user code.

3. Extension of the Spad programming language with *assumptions*.

4. A library of category hierarchies for algebraic properties in OpenAxiom.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PASCO 2010, 21–23 July 2010, Grenoble, France.
Copyright 2010 ACM 978-1-4503-0067-4/10/0007 ...$10.00.

The rest of this paper is organized as follows: section 2 gives a short introduction to Spad programming language; section 3 discusses the design principle of a hierarchy of algebraic operator categories; section 4 studies the extension of Spad programming language with *assumption*; our reduction detector is discussed in section 5; and the implementation details of the reduction detector are discussed in section 6; we present and discuss experimental results and running examples in section 7; Related work is in section 8; We conclude and discuss future directions in section 9.

2. EXTENDING AXIOM LIBRARIES

Spad is the library extension language of the AXIOM-family systems. It is a strongly typed language, with a two-level type system—categories and domains. We briefly discuss the key ideas of the Spad language relevant to this experiment. An in-depth coverage of the essentials of Spad is available from the AXIOM book [4]. Abstract algebraic structures are defined using *categories*. Categories can be thought of as specifications of algebraic concepts and properties. For instance, the monoid algebraic structure can be expressed in Spad as:

```
Monoid(): Category == Type with
  *: (%, %) -> %
  1: %
```

The category Monoid above specifies that the monoid algebraic structure has a binary operator * whose neutral element is 1. Specifications declared by categories are implemented by *domains*. The following fragment shows a version of domain ListMonoid, which implements the specification of the category Monoid:

```
ListMonoid(T: Type): Monoid == add
  import List(T)
  Rep == List(T)
  (x:%) * (y:%) == per concat(rep x, rep y)
  1: % == per empty()
```

ListMonoid domain is internally represented by domain List(T). It asserts membership to the category Monoid, and implements the binary operator and the neutral element with list concatenation operator concat and an empty list empty(), respectively. In the next section, we show how categories can be used for specifying algebraic properties of operators.

3. PROPERTY CATEGORIES

Most algebraic operators enjoy non-trivial properties. And most interesting algebraic algorithms are the result of skillful exploitations of operator properties, and structures of data they manipulate. Our approach to implicit concurrency rests upon the idea that data structures from computational algebra usually have rich algebraic properties, and algorithms operating on those structures should somehow be "tainted" by those properties. In particular, we are interested in those properties that enable automatic exploitation of implicit concurrency. To that end, we need mechanisms to express algebraic properties directly in programs. Furthermore, we need a way to organize those properties as library components so that they can be reused in large scale development—after all, mathematics are about organization of facts.

An effective operational way to think about AXIOM categories is to consider them as specifications. However, in their existing form they are closer a sub-language for initial algebra specification of data types without laws or equations. Support for axioms in categories has been considered by several researchers over the year, but there is no concrete implementation to date. We extent that notion of data type specification to cover specification of operator properties. Let us consider a simple hierarchy of algebraic operator categories, sufficient for our discussion in this paper. The hierarchy starts with the notion of binary operation:

```
MagmaOperator(T: SetCategory, op: (T,T) -> T): Category
  == Type
```

This definition says a binary operation on a domain satisfies the *magma operator* property. That is just a statement of the obvious. Now, we move on a more computationally interesting property: *associative operator*

```
AssociativeOperator(T: SetCategory,op: (T,T)->T): Category
  == MagmaOperator(T,op) with
    associativity:
      rule forall(a:T, b: T, c: T)
             op(a,op(b,c)) == op(op(a,b),c)
```

This category definition is essentially a logical statement (as Spad expression) of the property that an operator is associative if it is a magma operator and follows of a certain rule of re-association.

Next we consider monoid structures. An operator is a *monoid operator* if, in addition to being associative, it admits a neutral element:

```
MonoidOperator(T: SetCategory, op: (T,T) -> T): Category
  == AssociativeOperator(T, op) with
    neutralValue: T
```

Note that the neutral element is not a parameter to MonoidOperator. Rather, it is specified as a constant depending on the parameter op. This reflects the mathematical fact that the neutral value of a monoid operator is unique, therefore completely determined by that operator.

4. ASSUMPTIONS

Users express operator properties as *assumptions*. Properties are stated as facts without any attempt at proof beyond conventional type checking. We distinguish two kinds of assumptions: *ground assumptions* and *parameterized assumptions*.

4.1 Ground assumptions

A ground assumption states properties for a specific operator, whose input and output types are concrete non-parametrized domains. For example, the assumption

```
assume AssociativeOperator(NonNegativeInteger, max)
```

says the function max is a monoid operator over the domain of non-negative integers, with 0 as neutral value.

4.2 Parameterized assumptions

A parameterized assumption states properties for a family of operators. For example,

```
forall(T: Type)
  assume MonoidOperator(List(T), concat) where
    neutralValue == empty()$List(T)
```

asserts that concat is a monoid operation over List T for any type T. Type variables may be constrained, reflecting constraints on domains or operations. For instance, the parameterized assumption

```
forall(T: GcdDomain)
  assume MonoidOperator(T, gcd) where
    neutralValue == 0$T
```

states that gcd is a monoid operation over all GCD domains. whose neutral value is 0 of type T.

As illustrated in both examples, type variables in parameterized assumptions are introduced by declarations announced by the keyword forall. They must occur in deducible positions in assumptions.

In the experiments reported in this paper, an assumption is viewed like an annotation. It conveys a user's knowledge about an operator.

5. REDUCTION DETECTOR

The reduction detector takes a library file and assumptions from user, and attempts to extract reductions from the library file. The detection proceeds in two steps. First, it performs pattern matching on the fully typed abstract syntax trees obtained from the input program, after type checking and semantics elaboration. This phase yields a set of candidate reduction forms and scans forms. The detector then proceeds with only those binary operators for which associativity could be "certified" — either because they are built-ins or because they have matching assumptions.

5.1 Reduction forms

There are three ways to write reductions in AXIOM:

- explicit accumulation loop
- reduction operator form
- call to library function reduce

Explicit accumulation loop is probably the most widely known form. Consider the task of multiplying all integer values in a sequence seq. Writing that in AXIOM is just as simple as writing it in most programming languages:

```
result := 1
for v in seq repeat
  result := result * v
```

Here, the variable result is initialized with the neutral value of multiplication; then it is updated at every execution of the body of the loop. The final result is the accumulated value.

In the tradition of APL, AXIOM offers a short notation for reduction. The previous loop can be written as */seq. Here, / is the built-in reduction operator, not the ratio operator. It should be noted that in all flavours of AXIOM systems (including current releases of OpenAxiom), the built-in reduction operator is applicable only to a handful known monoid operators.

Finally, one can just call a library function: reduce(*,seq,1). Note that this form explicitly specifies the neutral value, just like in the accumulation loop case. That value was left implicit in the form using built-in reduction operator. Ideally, one should not have to supply the neutral value, since a monoid operation uniquely defines its neutral value.

5.2 Pattern matching reduction forms

Several semantic based pattern matching strategies are designed for extracting the various reduction forms.

Accumulation loop.

We introduce the concept *basic loop* for the purpose of describing our algorithms for detecting accumulation loops. A basic loop is a loop controlled by for-iterators, such that each statement in its body is either a variable definition, or an assignment to a previously defined variable. Example:

```
result : Integer := 0
for i in 1..10 repeat
  x : Integer := i+1         -- variable definition
  result := x * 2            -- variable assignment
```

We define an *accumulation loop* as basic atomic loop, where definitions or assignments involve certain expressions in *recognizable form*:

$$\iota_1 \cdots \iota_n \texttt{ repeat } \vec{\beta}$$

where $\iota_1, \ldots, \iota_n$ are for-iterators of the forms

- for v in e, with v a variable, and e a sequence
- for v in e_1..e_2, with v a variable, and e_1 and e_2 integer-valued expressions denoting the bounds of the loop-control variable v

An expression β is either a basic assignment of the form $v := f(\chi, e)$ or $v := f(e, \chi)$, or a conditional controlled by a side-effect free predicate and whose branches are sequences of basic assignments. The operand χ is either the same variable v being assigned to, or another expression of the form $f(\chi, e)$ or $f(e, \chi)$. We don't allow the control loop variables to be modified in the body of the loop. The accumulating variable v should have a linear occurrence in the right hand side of the assignment.

For simplicity, we take side-effect free predicate to mean a call to Boolean expression that does not use effectful functions. This is a semantics notion, therefore hard to check in practice. However, there is a notational convention used in AXIOM libraries, where a function name ending with symbol "!" indicates possibly effectful functions. Examples include concatenation function concat!, duplicate removal function removeDuplicate!, etc. The current implementation of the reduction detector does not allow effectful functions in accumulation—not just in the predicates.

The notion of recognizable accumulation loop is adapted from the aggregate array computation form studied by Liu and Stoller [6]. This adaptation is semantic-based since it draws heavily from type information. In each accumulation assignment, the same operator f has to be used consistently to accumulate values into the accumulation variable v. Because of operator overloading, we need to make sure that the same operator is applied consistently, and that requires overload resolution.

Built-in reduction operator.

The detection of built-in reduction operator is purely syntactic, unfortunately. For instance, parsing of +/[1,2,3,4] gives:

```
(REDUCE + 0
   (COLLECT (IN G784 (construct (One) 2 3 4)) G784))
```

The value 0 is automatically generated by compiler, and is inserted into the AST as the built-in neutral element of +. The reduction detector typechecks each parameter of the REDUCE operator except the neutral value which is automatically generated, and verifies that the operator parameter is a binary operator over some domain d, and the other parameter of the reduce form is a list whose element has type d.

Library function call.

The function reduce is heavily overloaded in AXIOM algebra libraries for implementing different functionalities or semantics. Therefore, given an AST whose operator is reduce, the reduction detector needs to type check all arguments, given enough seed to proceed with overload resolution.

5.3 Parallel prefix forms

A *parallel prefix* form (or scan) is a generalization of reduction. A parallel prefix form takes a sequence and a binary operator, and returns a sequence. Each entry of the result is filled by a reduction, *i.e.*, the value of the i-th element is given by the reduction which applies the input binary operator to combine the values up to the i-th element of the input sequence. That can be expressed in at least two forms:

- explicit scan loop
- call to library function `scan`

We elaborate only on loops. The following example illustrates a prefix sum of the sequence `seq`:

```
sum := first seq
for i in 2.. #seq repeat
  sum := sum + seq.i
  res.i := sum
```

The implementation of OpenAxiom library functions `scan` are based on scan loops as well as self recursions.

Matching parallel prefixes.

Strategies for extracting scan operations are again based on pattern matching. We use two kinds of recognizable form.

$$\iota_1 \cdots \iota_n \texttt{ repeat } \vec{\xi}$$

where $\iota_1, \ldots, \iota_n$ are `for`-iterators. The form of the loop body $\vec{\xi}$ is partially determined by the specific form of `for`-iterators:

- If the iterators are of the form `for` v `in` e, with v a variable, and e a sequence, then $\vec{\xi}$ is a two statement sequence, where the first statement is an accumulation of the form $v := f(v, e)$ or $v := f(e, v)$ where v is a variable, and the second statement is for updating the vector for storing the prefix result, this can be a sequence concatenation expression such as $r := \text{concat}(r, v)$, where r is the resulting sequence of scan, and operator concat appends the value of v to the end of the resulting sequence v.
- When the iterators are of the form `for` i `in` e_1`..`e_2, with i a variable, and e_1 and e_2 integer-valued expressions denoting the bounds of the loop-control variable i. The loop can be either one statement of the form $r.i := f(r.(i-1), s.i)$ where the $(i-1)$-th element of the resulting sequence r and the i-th element of the input sequence s are combined together via binary operator f, the result is written into i-th element of r; Or $\vec{\xi}$ can be a sequence containing two assignments, where the first statement is of the form $v := f(v, s.i)$ or $v := f(s.i, v)$ where v is a variable, and the second statement updates value of v into the resulting vector r, *e.g.*, $r.i := v$, or $r := \text{concat}(r, v)$.

5.4 Assumption uses

Assumptions, as reported in this paper, are not used directly by users. Rather, they form an external knowledge database consulted by static analysis tools such as the reduction detector. For implicit concurrency, the most important property we focus on is associativity. To get there, the reduction detector needs to gather properties from the assumption database supplied by the user. Then, the computed information is passed to the associativity checking engine.

If the operator in a reduction form is associative, the form is deemed *fully parallelizable*. On the other hand, if associativity cannot be decided from the set of available assumptions, the accumulation loop is said to be *partially parallelizable*. For instance, the following loop

```
for i in 1..10 repeat
  x := x + i
  y := y * i
  z := z quo i
```

is only partially parallelizable. However, it can be rewritten as two loops: one that is fully parallelizable:

```
for i in 1..10 repeat
  x := x + i
```

and a second that is not readily so (according to our recognizable form definition):

```
for i in 1..10 repeat
  y := y * i
  z := z quo i
```

6. IMPLEMENTATION

The reduction detector is implemented as a Spad library of a branch of the OpenAxiom system [1]. The implementation of the reduction detector accounts for approximately 2000 lines of Spad code. It uses the AST library component of the standard OpenAxiom system. The overall workflow is illustrated in Figure 1. To maintain portability to other AXIOM systems not implementing assumptions, users are required to write their Spad code and assumptions in different files.

6.1 A Spad typechecker in Spad

The pattern matching and assumption verification steps of reduction detection requires typechecking. Instead of exporting the typechecking function from the Spad compiler which is written in a lower-level language named Boot, we built an independent typechecking library in Spad. The implementation of the typechecker consists of about 1000 lines of Spad code. It is used for typechecking, overload resolution, and assumption checking. The library is tested with the complete set of OpenAxiom algebra library, where some programs were edited before passing to the typechecker. It is still experimental.

6.2 Preprocessing before pattern matching

A basic atomic loop needs to be preprocessed before pattern matching. The preprocessing step essentially consists of forward substitution of expression [7], eliminating assignments to the variables used as intermediate stores.

The purpose of this pass is to expose "hidden" accumulations. Indeed, some intermediate stores may introduce unnecessary dependencies, which may prevent an accumulation loop from being identified.

6.3 Inferring properties from assumptions

Assumptions are organized in hierarchies. So, it can happen that a property the reduction detector is looking for is not textually present in the assumption database, but must be derived using the semantics of hierarchy as entailment. Consider

```
forall(D: IntegralDomain)
  assume MonoidOperator(D, *) where
    neutralValue = 1$D
```

The assumption states that the operator * specified by `IntegralDomain` category is a monoid operator. This implies that all other properties entailed by the parents categories of `MonoidOperator` (e.g. associativity) are also inherited.

[1]The source code is at `http://open-axiom.svn.sf.net/svnroot/open-axiom/yli-sandbox/`

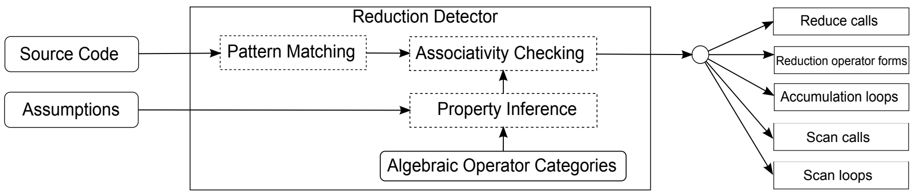

Figure 1: The workflow of our reduction detector.

7. EXPERIMENTS

The experiment has two parts. The first only uses the pattern matching component to identify all reductions used in OpenAxiom algebra libraries. The second identifies those reductions that are parallelizable.

7.1 OpenAxiom algebra library

The OpenAxiom library consists of type definitions. Each type is a package, a domain, or a category.

Item	Number
Type definitions	1129
Category definitions	222
Domain definitions	434
Package definitions	473
Function definitions	12748

Table 1: Statistical properties of OpenAxiom algebra library.

From Table. 1, we observe that categories account for the smallest portion of the library. This is expected since category are specifications, and it is reasonable to have fewer specifications than implementations. Packages are almost as prevalent as domains. On average, there are 11 function definitions per type definitions. Note that a category definition can also contain function definitions in its body, *e.g.*, the generic operator `gcd` of category `GcdDomain`.

7.2 Reduction extraction

The results of running the pattern matching engine over the algebra library files is summarized in Table. 2 and Table. 3. A significant portion (about 32%) of loops are atomic loops, and about 46% of atomic loops implement reductions. The number of reductions per 100 function definitions is smaller than our expectation, this is mainly because the current implementation does not compute the transitive closure of call chains to functions implementing reductions. Among the three kinds of reduction forms, built-in reduction operator is the most frequently used.

Table. 4 gives the distribution of accumulation loops according to the number of accumulation statements in their bodies. The data, with the limitations of the current implementation, suggest that most accumulation loops contain a single accumulation statement; those containing more than 2 accumulations are rare.

7.3 Distribution of reduction forms

A set of 13 operators were manually annotated as associative: + of category `AbelianMonoid`, * of category `Monoid`, `gcd` and `lcm` of category `GcdDomain`, `max` and `min` of category `OrderedSet`, two list concatenation operators `concat` and `append`, one list union operator `setUnion` of domain `List(T)` where `T` has category `Type`, two matrix concatenation operators `horizConcat` and `vertConcat` over domain `Matrix(T)` where `T` has `Type`, and two local functions

Item	Number
Loops	2181
Atomic loops	689
Reductions	820
Reductions/100 function defs	6

Table 2: Statistical properties of loops and reductions.

Item	Number	Percentage in reductions
Accumulation loops	333	41%
Function `reduce` calls	73	9%
Built-in reduce operator	414	50%

Table 3: Statistical properties of reduction forms.

No. of Acc. Stmt.	No. of Acc. Lps.	Percent. in Acc. Lps.
1	305	91.6%
2	22	6.6%
3	6	1.8%

Table 4: Distribution of the number of accumulation loops with different number of accumulation statements in their bodies.

`pairsum` of domain `List(T)` where `T` has category `Type`, and the operator `sum` over domain `Expression(DoubleFloat)`.

All algebraic properties in this experiment entail associativity. Table. 5 lists the distribution of the different parallel reductions regarding each reduction operator. Partially parallelizable loops (PPL) seem to be less frequent than other reduction loops. This indicates that these 13 operators rarely appear in an accumulation loop with more than one accumulation statements involving a different accumulation operator. Addition, multiplication, and list concatenation are dominant in parallelizable reduction forms.

7.4 Parallel prefix extraction

The statistics obtained for parallel prefix suggests that it is barely used in current versions of OpenAxiom library. The detector discovers 4 scan loops and 11 `scan` function calls. We found only two scan function calls which are parallelizable due to the use of associative operator + over `PolynomialCategory` and `Ring`, respectively.

7.5 Rejected cases

The analysis is conservative; that is it may reject some reduction forms on the ground that it cannot certify — based on available static information — that they are indeed bona fide parallel (prefix) reduction forms. For instance, the recognizable form requires that each accumulation assignment uses only one binary operator. In particular, it forbids cases where different accumulation operators appear in one accumulation. For example, the following accumulation loop from function `basisOfRightNucleus` of package

Operator	PPL	FPL	PRC	PRF
+ of AbelianMonoid	7	76	8	148
* of Monoid	6	21	1	76
gcd	1	0	4	4
lcm	0	0	3	7
concat	1	18	1	1
append	1	8	1	17
max	2	1	8	36
min	2	0	3	12
horizConcat	0	2	2	0
vertConcat	0	3	0	0
setUnion	0	2	1	35
pairsum	0	1	0	0
sum	0	0	3	0

Table 5: Amount of parallel reductions based on different user assumptions. PPL: Partially Parallelizable Loop, FPL: Fully Parallelizable Loop, PRC: Parallelizable `reduce` call, PRF: Parallelizable Reduce Form.

`AlgebraPackage`:

```
for l in 1..n repeat
  entry :=  entry + elt(gamma.l,k,i)*elt(gamma.s,j,l)_
               - elt(gamma.l,j,k)*elt(gamma.s,l,i)
```

was rejected. Indeed, parsing of the loop above gives the AST:

```
(REPEAT  (STEP l (One) 1 n)
 (%LET  entry
  (- (+ entry (* (elt (gamma l) k i) (elt (gamma s) j l)))
   (* (elt (gamma l) j k) (elt (gamma s) l i)))))
```

The appearance of the operator + does not match our definition of recognizable form. An algebraic term rewriting may help the detector. For instance, by applying the rewriting rule $-x \equiv +(-x)$ to the example above, we obtain:

```
for l in 1..n repeat
  entry := entry + elt(gamma.l,k,i) * elt(gamma.s,j,l)_
               + (-elt(gamma.l,j,k) * elt(gamma.s,l,i))
```

which becomes:

```
(REPEAT  (STEP l (One) 1 n)
 (%LET  entry
   (+ (+ entry (* (elt (gamma l) k i) (elt (gamma s) j l)))
    (- (* (elt (gamma l) j k) (elt (gamma s) l i))))))
```

The detector reported 14 cases of this kind. Other examples include 3 uses of side-effecting functions, and 2 of the unsupported recursive parallel prefix patterns.

8. RELATED WORK

8.1 Reduction detection

Reduction detection is a well developed program analysis technique in the compiler construction community. It is widely used in automatic parallelization of loops. An internal representation of loops needs to be specified, the definition of reduction patterns and the design of pattern matching algorithms are further based on that internal representation. Jouvelot and Dehbonei [5] represent loops symbolically, and reductions patterns are formalized as values of symbolic stores so that pattern matching is applied to the symbolic stores. Pinter and Pinter [9] use dependence graphs. Redon and Feautrier [10] use a preprocessor that taking loops and generates linear recurrence equations as internal representations, and reductions are discovered via reasoning on the generated linear equations. Liu and Stoller [6] describe how incrementalization aids optimizing aggregate array computation, which is a very typical generalization of reductions. In that paper, a recognizable form for reduction semantics is derived directly from the abstract syntax tree of an accumulation loop. The recognizable accumulation loop discussed in this paper builds on that work. This form is syntax directed, which simplifies implementation. Indeed, instead of building a tools for between several representations, the reduction detector presented in this paper shares the fully typed AST with the type checker. Recently, Gautam and Rajopadhye [3] studied polyhedra equations to represent reductions. The polyhedra model provides powerful mathematical foundations to simplify the algorithmic complexities of accumulation loops. Empirical data, obtained by augmenting our reduction detector extractor, show a total of 431 loop nests, out of which 57% are affine control loops, and an average of 2.4 per nest depth.

Much of this work [5, 9, 10, 6] supports reduction extraction for Fortran program, Liu and Stoller's work [3] implements reduction detection and simplification using the ALPHA language and the MMALPHA framework for transforming ALPHA program [1]. However, for programming languages such as Spad, Aldor, C++, and Java, those tools cannot correctly extract reductions of user defined functions—compiler does not have knowledge about the algebraic properties of some reduction operator over user defined types due to overloading. The assumption mechanism presented in this paper helps user convey operator properties to compiler tools.

8.2 Attributes

The original AXIOM system supports algebraic properties via *axioms* and *attributes* in type descriptions [4, Chapter 12]. However, axioms are just stated as comments, they do not affect the compiler in any way. *Attributes* are uninterpreted identifiers or tags. However, the attribute support is very limited and it is not possible to express that a function is a monoid structure with a specific neutral value.

8.3 Maple `assume` facility

The Maple computer algebra system has enjoyed an assume facility since the work of Weibel and Gonnet [13, 14]. It enables a powerful conditional rewriting tool for expressions simplification. That functionality is not directly supported by the work described in this paper. However, it would be interesting to explore how Maple's `assume` facility—which is mostly dynamic—could be combined with the type-directed approach of this paper to support correct and fast algebraic computations on modern computers.

9. CONCLUSIONS AND FUTURE WORK

This paper presented an empirical and quantitative study of reductions in the AXIOM algebra libraries. The experiments suggest rich parallelization opportunities exposed by the uses of a language extension called *assumptions*. Experimental data show that specifying operator properties directly in code, checkable by the compiler, is beneficial for parallelizable reductions. The core idea of this approach is not restricted to Spad or Aldor programs. It can be applied to programs written in higher-level languages such as C++ or Java.

There are several directions for future work that we would like to explore. Currently, a parallel library is being built in OpenAxiom aiming at providing support for higher-level parallel programming. It would be interesting to see how a compiler that utilizes the information provided by our reduction detector, could perform an effective implicit parallization. Another direction would be to develop more use cases of *assumption* to increase its benefit to symbolic computation. As inspired by recent work on axiom based verifications for Java program [12, 11], it would be also interesting to see how user assumption can help verify program transformations for computer algebra code.

10. REFERENCES

[1] Alpha. `http://www.irisa.fr/cosi/ALPHA`, IRISA, France, 2010.

[2] J. V. Z. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2003.

[3] Gautam and S. Rajopadhye. Simplifying reductions. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 30–41, New York, NY, USA, 2006. ACM.

[4] R. D. Jenks and R. S. Sutor. *AXIOM: The Scientific Computation System*. Springer-Verlag, 1992.

[5] P. Jouvelot and B. Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing*, pages 186–194, New York, NY, USA, 1989. ACM.

[6] Y. A. Liu, S. D. Stoller, N. Li, and T. Rothamel. Optimizing aggregate array computations in loops. *ACM Trans. Program. Lang. Syst.*, 27(1):91–125, 2005.

[7] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[8] OpenAxiom. `open-axiom.org`, 2010.

[9] S. S. Pinter and R. Y. Pinter. Program optimization and parallelization using idioms. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 79–92, New York, NY, USA, 1991. ACM.

[10] X. Redon and P. Feautrier. Detection of recurrences in sequential programs with loops. In *PARLE '93: Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 132–145, London, UK, 1993. Springer-Verlag.

[11] R. Tate, M. Stepp, and S. Lerner. Generating compiler optimizations from proofs. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 389–402, New York, NY, USA, 2010. ACM.

[12] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 264–276, New York, NY, USA, 2009. ACM.

[13] T. Weibel and G. H. Gonnet. An algebra of properties. In *ISSAC '91: Proceedings of the 1991 international symposium on Symbolic and algebraic computation*, pages 352–359, New York, NY, USA, 1991. ACM.

[14] T. Weibel and G. H. Gonnet. An assume facility for cas, with a sample implementation for maple. In *DISCO '92: Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems*, pages 95–103, London, UK, 1993. Springer-Verlag.