Adding Dimension Analysis to Java as a Composable Language Extension[⋆]

Extended Abstract

Eric Van Wyk and Yogesh Mali

Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455, USA
`evw,yomali@cs.umn.edu`

Abstract. In this paper we describe a language extension that adds dimension analysis to Java. Dimension analysis can be used to check that values that represent physical measurements such as length and mass are not used inconsistently. What distinguishes this work from previous work that adds dimension analysis to programming languages is that here the extension is implemented as a *composable* language extension. This means that it can easily be combined with other extensions, possibly developed by other parties, to create an extended implementation of Java with new features that address concerns from several different domains.

1 Introduction

Dimension analysis can be used to check that a computer program does not incorrectly use values that represent physical measurements. For example, it can ensure that a value representing a length is not added to a value representing a mass. This analysis may be extended to also take into account the units of measurements of these values and thus check, for example, that a length measurement in feet is not added to a length measurement in meters.

Modern programming languages rarely provide support for this type of analysis and it is the source of a number of highly publicized software failures. For example, in September 1999, the unsuccessful landing of the Mars Climate Orbiter on the surface of Mars was traced back to a software failure in which a measurement in English units was interpreted as a measurement in metric units by the navigation software [13].

This paper is not the first to add dimension analysis to a programming language. To mention just a few, Wand and O'Keefe [20] and Kennedy [11] add dimension inference and analysis to ML and House [9] add it to Pascal. The work presented here differs from these in that it adds dimension analysis to Java as a *composable* language extension. The others add dimension analysis by creating a

[⋆] This work is partially funded by NSF CAREER Award #0347860, NSF CCF Award #0429640, and the McKnight Foundation.

new monolithic language that cannot be easily extended with some other features that may be desired. Our goal is to extend a language with dimension analysis in such a way that it is composable with other language extensions.

Our dimension analysis extension is implemented in the ableJ extensible language framework [17]. ableJ currently supports Java 1.4, but some aspects of Java 1.5 have been added. It is often the case in ableJ that the composition of the host language (Java) and several extensions can be done automatically. Thus, a programmer can direct the tools to build a customized language implementation for the host language extended with the unique set of extensions that he or she requires to handle the task at hand. This paper describes how dimension analysis can be implemented as a composable language extension. This extension follows the pattern described previous work on ableJ and composable langauge extensions [17].

To get a sense of the type of dimension analysis that is supported by this extension, consider the sample program in Figure 1. The key features include the new type expressions for specifying dimensions in types. For example, the fields `len` and `wid` are defined with dimension type "`Dim<int, L>`" whose dimension expression `L` indicates that this is a length measurement and whose *representation type* specifies that this measurement value is represented as an integer. These type expressions share the syntax of Java generics but are implemented as

```
public class Sample {
  Dim<int, L> len, width, perimeter ;
  Dim<int, L^2> area ;
  public Dim<int, a b> product (Dim<int, a> x, Dim<int, b> y)
    { return x * y ; }
  public Dim<int, a> sum (Dim<int, a> x, Dim<int, a> y)
    { return x + y ; }

  void demo (int l, int w) {
    len = (Dim<int, L>) l ;   // cast from primitive types
    wid = (Dim<int, L>) w ;
    perimeter = len + wid + len + wid ; // a valid sum
    Dim<int, T> t = (Dim<int, T>) 3 ;
    Dim<int, L/T^2> acc ;      // an acceleration variable
    area = len * wid ;         // a valid product
    len = sum(len, wid) ;      // valid call to method sum
    area = product(len, wid) ; // valid call to method product
    len = sum(len, area) ;     // invalid call to method sum
    len = len + area ;         // a dimension error
    acc = len / (t * t) ;      // an acceleration
  }
}
```

Fig. 1. A sample program using the dimension analysis extension.

new types; Dim is a new keyword, not the name of a parameterized class. Another feature to note is that arithmetic operators such as +, *, and other are overloaded for dimension types so that we can check that dimension values are added and multiplied correctly. Assignment is similarly overloaded. An overloaded method call operator ensures that for any method call, the dimension variables (a and b) in the dimension expressions in methods product and sum are substituted for types in a consistent manner. In the second assignment to area in demo we check that the dimension variable a is instantiated to the same dimension expression, in this case L, in both of the input types and in the output type and that these instantiated dimension expressions match those in the types of the actual arguments. In processing this program, the extended language implementation will check that the dimension types are used correctly and translate the program to a pure Java program in which the dimension types are translated to their representation types. Although this example program is a bit contrived it highlights the key features of the language extension.

In Section 2 we describe the ableJ framework and the attribute grammar that defines Java. In Section 3 we describe the attribute grammar that defines the language extension that implements dimension analysis. In Section 4 we describe some related work and conclude in Section 5.

2 The ableJ 1.4 extensible language framework

In this section we briefly describe the ableJ extensible language framework. Java 1.4 and the dimension analysis language extension are implemented as attribute grammars written in the Silver [15] attribute grammar specification language. Many aspects of the ableJ grammar have been simplified for presentation reasons. Additional information about ableJ can be found in previous work [17].

Silver has many features beyond those originally introduced by Knuth [12]. These include higher-order attributes [19], collection attributes [3], forwarding [16], various general-purpose constructs such as pattern matching and type-safe polymorphic lists. Silver also has mechanisms for specifying the concrete syntax of languages and language extensions. It passes these specifications to Copper, our parser and scanner generator that implements context-aware scanning [18]. In this approach the scanner uses information from the LR-parser's state to disambiguate lexical syntax. The resulting scanner and parser are deterministic and also support the composition of language extensions. In this paper we will only present the abstract syntax of ableJ and the dimension extension.

Figure 2 presents a significantly simplified subset of the Silver specification of Java 1.4 which is used in our actual implementation. This grammar defines nonterminal symbols, terminal symbols, productions, and attributes. The nonterminals Stmt, Expr, and Type represent, respectively, Java statements, expressions, and type expressions that appear in the abstract syntax tree of a Java program. The nonterminal TypeRep is used in the symbol table (the attribute env) in bindings of names to type representations.

```
grammar edu:umn:cs:melt:ableJ14 ;
nonterminal Stmt, Expr, Type, TypeRep ;
terminal Id_t /[a-zA-Z][a-zA-Z0-9_]*/ ;

syntheszied attribute pp::String occurs on Expr, Stmt, Type;
synthesized attribute errors::[String] occurs on Expr, Stmt, ...
synthesized attribute typerep::TypeRep occurs on Expr, Type ;
synthesized attribute hostStmt::Stmt occurs on Stmt ;
synthesized attribute hostType::Type occurs on Type ;

abstract production if_then  s::Stmt ::= cond::Expr  body::Stmt
{ s.pp = "if (" ++ cond.pp ++ ") {\n" ++ body.pp ++ "}\n" ;
  s.hostStmt = if_then( cond.hostExpr, body.hostStmt ) ;
  cond.env = s.env ;   body.env = s.env ;
  s.errors = case cond.typerep of
               booleanTypeRep() => [ ]
             | _ => [ "Error: condition must be boolean." ]
             end ++ cond.errors ++ body.errors ;              }

abstract production add    e::Expr ::= l::Expr  r::Expr
{ e.pp = "( " ++ l.pp ++ " + " ++ r.pp ++ ")" ;
  attribute transforms :: [Expr] with ++ ;
  transforms := [ ] ;
  forwards to if length(transforms) == 1  then head(transforms)
         else if length(transforms) == 0  then exprWithErrors(
                   ["Type error on addition, types not supported."] )
         else exprWithErrors(["Internal compiler error."])  ;        }

abstract production localVarDcl   s::Stmt ::= t::Type id::Id_t
{ s.pp = t.pp ++ " " ++ id.lexeme ++ "\n" ;
  s.defs = [ varBinding(id.lexeme, t.typerep ) ] ;  }

abstract production boundId    e::Expr ::= id::Id_t  t::TypeRep
{ e.pp = id.lexeme ;
  attribute transforms :: [Expr] with ++ ;     transforms := [ ] ;
  forwards to if length(transforms) == 1   then head(transforms)
              else generic_boundId(id,t) ;    }

abstract production booleanTypeExpr  te::TypeExpr ::= b::'Boolean'
{ te.typerep = booleanTypeRep() ;  te.pp = "Boolean" ;
  te.hostType = booleanTypeExpr(b);  }
abstract production booleanTypeRep  tr::TypeRep ::=   { tr.pp="boolean"; }

aspect production add    e::Expr ::= l::Expr  r::Expr
{ transforms <- if match(intTypeRep(), l) && match(intTypeRep(), r)
                then add_int(l,r)   else [ ] ;   }

abstract production add_int   e::Expr ::= l::Expr  r::Expr
{ e.typerep = intTypeRep ( );  e.hostExpr = add(l.hostExpr,r.hostExpr); }
```

Fig. 2. Highlights of a simplified Java host language attribute grammar.

Synthesized attributes such as the pretty print attribute `pp` are defined. `pp` has type `String` and decorates (`occurs on`) tree nodes of type `Expr`, `Stmt`, and others. An errors attribute is used to collect type errors, and later dimension errors, found in a program. Both of these attributes are defined on the `if_then` production. The definition of the `errors` attribute uses pattern matching to check that the type (`typerep`) of the condition is Boolean.

The production `localVarDcl` creates a binding of the name of its identifier to its type representation and passes this up the tree in the `defs` attribute. At the statement-block level (and others) this information is collected to form the symbol table and passed back down the tree in the inherited attribute `env`.

Forwarding: We have previously introduced *forwarding*[16] as an extension to attribute grammars that is useful in specifying languages in a highly modular way. To use forwarding, a production specifies (using the `forwards to` clause) how to build a new AST that will be queried for any attributes that are not explicitly defined by an equation on the "forwarding" production. This new AST, called the "forwarded-to tree", can be seen as the *semantic equivalent* of the original forwarding AST. If the original tree does not explicitly define an attribute a, its value is automatically computed by copying it from the a attribute on the forwarded-to tree.

Forwarding is used by productions that define new language extensions to specify the semantically equivalent construct in the host language that they will "translate" to. The type expression "`Dim<int, L>`" used in the declaration of `len` and `wid` in the second line of Figure 1 will be represented in the program's AST by a production that forwards to the type expression tree for `int`. As we will see below (Figure 4), the extension production will define some attributes to facilitate dimension analysis but it will not define any of the `host`NT attributes that are used to translate the extended program to a semantically equivalent host language program. Each nonterminal NT in the host language is decorated by a synthesized attribute `host`NT of type NT that holds a node's translation to the host language. This attribute is defined only on host language productions and computes the translation using the host language production and the host language translations of its children (stored in their `host`NT attribute) to compute this. This can be seen on the `if_then` production. When the dimension type expression tree for "`Dim<int, L>`" is queried for the value of its `hostType` attribute it will forward that query to the type expression tree for "`int`" which will return a copy of itself. In this manner, we can extract the translation of the extended program to the host language.

Operator Overloading: Operator overloading also uses forwarding and this can be seen in the production `add`. This production is used by the parser in constructing the original AST. It is a place holder that will forward to a new `Expr` tree constructed by a production specific to the types of the operands. In the simplified example in the figure, this production specifies a *collection attribute* [3] named `transforms` that is given an initial value (by the `:=` operator) of an empty list. The *aspect production* for `add` near the bottom of the figure can remotely define values for attributes for the original `add` abstract production. In this case, it may add an `Expr` tree to the `transforms` list, using the `<-` operator, if the types of the operands `l` and `r` are both integers. The tree that it may add is constructed by the `add_int` production. This production defines the `errors` and `typerep` attributes for integer addition. When the original tree built by `add` is queried for its `typerep` attribute, for example, it forwards that query to the first tree in the `transforms` list. If `transforms` is empty, then there is a type error in the program; there is no implementation for `add` for the types of the child expressions. If there is more than one tree in `transforms` then an internal compiler error has occured since more than one aspect production has specifed an implementation for addition for the types of the child expressions. This can only occur when language extensions overload operators for host language types, which they rarely do. Extensions typically overload operators for the new types that they introduce; such overloading do not trigger this error and it is thus rare in practice. In Section 3 we will see that the dimension analysis extension overloads addition and other constructs in a similar fashion.

Language Extensions: In the following section we describe the attribute grammar specification of the dimension analysis language extension. This extension follows the pattern of other extensions to the ableJ framework. Extensions may introduce new language constructs, either by specifying new concrete syntax so that their abstract syntax is placed into the original AST, or by using operator overloading facilities so that a "place holder" production will forward to a production specified in the language extension. In either case, the productions defined in the extension will (and must) specify their transformation to a semantically equivalent host language construct using forwarding. In effect, the translation of the extended program to a host language program is carried out by the many local transformations that forwarding specifies. Different language extension constructs are not isolated from one another in the AST however since they can communicate when declarations add information to the symbol table that may be retrieved in other parts of the AST. The extension productions will do some semantic analysis (by explicitly defining some attributes). Forwarding is then used to specify their translation to the host language. The dimension analysis extension follows this pattern. The new productions perform the dimension analysis and then forward to pure Java constructs on which the dimension information has been translated away.

3 Dimension analysis extension

In this section we discuss some principles of dimension analysis and then show how dimension analysis can be added as a composable language extension to the ableJ specification of Java. This language extension defines new syntax for type expressions and overloads some existing host language syntax for arithmetic operators, among others, to perform dimension analysis and ensure that measurement values are not incorrectly used.

3.1 Principles of dimension analysis

Dimensions describe a specific type of measurement. These include length, mass, temperature, among others. These are not to be confused with units that specify the unit of measurement for a particular dimension. For example, the dimension of length can be measured in units of feet or meters. Dimensions can be classified as *base dimensions* or *derived dimensions*. Traditionally, base dimensions include length, mass, temperature, time, electric current, amount of material and luminosity. Derived dimensions are specified in terms of these. For example, area is derived from the base dimension of length as length squared; acceleration is derived from length and time as length divided by time squared. Dimension expressions are generated from base dimensions and dimension variables using the operations of product and inverse. From these operations we can define division and exponentiation. A *unit* dimension is represented as 1. We represent dimension expressions in our language extension using the nonterminal `DimExpr` and the productions shown in Figure 3. A nonterminal and productions for base dimensions are also shown. The acceleration dimension expression `L / T^2` is represented by the tree `divide(basedim(L()), power(basedim( time()),2))`.

We will define a few functions on `DimExpr` trees for use in type checking. A `unify` function is used to unify two dimension expressions and if successful, return the set of substitutions for dimension variables that unifies them. This function will be used in checking that two expressions that have dimension types can be added or copied in an assignment. If the dimension expressions that are components of their types can be unified then it is safe to add or copy them. Note that multiplication of values of different dimensions is always allowed. The dimension of the resulting product is the product of their respective dimension expressions. If unit checking is added, then multiplication can fail if the operands use units inconsistently.

```
nonterminal DimExpr ;
abstract production product de::DimExpr ::= l::DimExpr r::DimExpr {  }
abstract production divide  de::DimExpr ::= l::DimExpr r::DimExpr {  }
abstract production power   de::DimExpr ::= b::DimExpr e::Integer {  }
abstract production dimvar  de::DimExpr ::= v::String   { }
abstract production basedim de::DimExpr ::= bd::BaseDim { }
abstract production unit    de::DimExpr ::=    { }

nonterminal BaseDim ;
abstract production M  bd::BaseDim ::= { }   -- Mass
abstract production L  bd::BaseDim ::= { }   -- Length
abstract production T  bd::BaseDim ::= { }   -- Time
```

Fig. 3. Grammar for dimension expressions

3.2 Type expressions for dimension analysis

New type expressions: We need new type expressions so that programmers can specify dimension types. Thus, an abstract production `dimTypeExpr` with the left hand side as the host language nonterminal `Type` is introduced in the extension. This production is shown in Figure 4. This production specifies that dimension types consist of a representation type `rep` (int, float, Integer, etc) and a dimension expression `d` that specifies the dimensionality of the values to be represented. This production defines a pretty print attribute and errors attribute as expected. The `typerep` attribute specifies the *representation* of the type. This is used by productions such as the `localVarDcl` production from Figure 2 to add information to the symbol table that binds variable names to types. The end result is that there are entries in the symbol table `env` that bind variable names to information about their type. The `dimTypeExpr` production passes the symbol table `env` down the tree to its components and also forwards to the representation type `rep`.

Note that this extension does not check that the underlying representation types are type correct. Dimension analysis productions, such as `dimTypeExpr`, will forward to constructs on which the dimension types have been erased that do additional type checking on the translated version to ensure that the underlying representation types are used correctly. Errors detected here are reported to the programmer.

Dimension Expressions: Concrete syntax productions are used to parse dimension expressions like those shown in the `Dim` type expressions in Figure 1 to construct abstract syntax trees (ASTs) using the productions in Figure 3. These trees are the representation of the dimensions used in dimension analysis. We do not describe the specification of the concrete syntax here as it is what one would expect. A normalization function (`normalize`) converts these expressions into a normalized form that simplifies the dimension analysis. For example, `normalize` would simplify the dimension expression `(L T) / (M T^2 M^(-1))` to `L / T`.

```
abstract production dimTypeExpr  te::Type ::= rep::Type d::DimExpr
{ te.pp = "Dim<" ++ rep.pp ++ "," ++ d.pp ++ ">" ;
  te.errors := rep.errors ++ d.errors;
  te.typerep = dimTypeRep_ST(rep.typerep, normalize(d)) ;
  rep.env = te.env ;     d.env = te.env ;
  forwards to rep ;           }

abstract production dimTypeRep_ST tr::TypeRep ::= rep::TypeRep de::DimExpr
abstract production dimTypeRep_Ex tr::TypeRep ::= de::DimExpr reptree::Expr
```

Fig. 4. Type expression and representation productions.

Type representations: The host language nonterminal `TypeRep` is used for internal representations of types that are used in type checking in ableJ. For our dimension extension to fit into the ableJ framework we define productions that define type representations for dimension types. There are two such representations. The first production, `dimTypeRep_ST`, is used in the symbol table and specifies the dimensionality and the representation type of variables declared as dimension types. The signature of this production is shown in Figure 4. This production is used in `dimTypeExpr` to define the type representation of the dimension type. The second, `dimTypeRep_Ex`, is used in type representations that decorate expressions. This one also specifies the dimensionality of the expression but instead of the type of the representation it specifies the tree that this expression will translate to. The type representation on this tree is the representation type. Thus, expressions (`Expr` trees) that have a dimension `typerep` will forward to this representation tree that is part of their type. This `TypeRep` production is used below in the `Expr` productions that overload arithmetic operators.

3.3 Overloading existing host language syntax

When introducing a new type, we often find it useful to overload the certain host language operations to provide type-specific behavior to these operations. For example, we will overload the host addition production `add` so that we can check that the dimensions of the operands are compatible on addition. This subsection describes several of the operators (productions) that are overloaded.

Overloading variable references: It is sometimes useful to overload the variable reference production so that variables that are bound to dimension types get their own type-specific production in the AST.

The ableJ infrastructure handles Java name disambiguation[1] and the looking up of names and binding them to their types. The abstract productions that perform this task are not relevant here. What matters is that they will forward to the production `boundId` shown in Figure 2. This production has a *collection* attribute called `transforms` that has the type `[Expr]`. Language extension will add new trees to this list if they want to overload a specific instance of a variable reference. This is similar to the way in which `add` is overloaded.

The aspect production at the top of Figure 5 from the dimension extension specifies that if the type bound to this identifier is a dimension type, then add the AST that is constructed with the bound identifier production specific to dimension types (`boundId_dims`) to the list of possible trees that the original `boundId` can transform to. If there is only one such tree, then the `boundId` production will forward to that. In the case where the type of the identifier is a dimension type, we then effectively overload variable references with the `boundId_dims` production. This production, also in Figure 5, defines the type (`typerep` attribute) to be the dimension type that contains the dimension expression of the type (`dimexpr(t)`) and the tree that this will eventually translate to (`reptree`). The

[1] This determines, for example, if "`a`" in "`a.b.c`" is a package, a class, or an object.

tree `reptree` is constructed with the `boundId` production but the type given to that production is the representation type of the dimension. As an example, consider the variable `len` in the example program in Figure 1. On the multiplication of `len` with `wid`, the `len` identifier in the original AST is overloaded using the production `boundId_dims` and sets its type to be a dimension type that contains a representation tree that is just the bound identifier "`len`" that has as its type the type `int`. Thus, we essentially erase the dimension information in the translation to Java once we have verified that the dimension values are used in a correct manner.

```
aspect production boundId   e::Expr ::= id::Id_t  t::TypeRep
{ transforms <- if match(dimTypeRep_ST(_,_), t)
                then [ boundId_dims(id,t) ]   else [ ] ;   }

abstract production boundId_dims  e::Expr ::= id::Id_t  t::TypeRep
{ e.pp = id.lexeme ;
  e.typerep = dimTypeRep_Ex(dimexpr(t), reptree) ;
  forwards to reptree ;
  local attribute reptree :: Expr ;
  reptree = boundId(id, reptyperep(t)) ;   }
```

Fig. 5. Overloading variable references.

Overloading arithmetic operations: In Section 2 we showed how addition can be overloaded with a type specific production. In Figure 6 are the aspect and dimension-type-specific productions that accomplish this for dimension types.

The production `add_dims` will unify the dimension expressions from the type representations of the two operands `l` and `r`. If unification succeeds, this process returns a list of bindings (`bnds` of type `[UnifyBnd]`) that map dimension variables to dimension expressions that will unify the two dimension expressions and an empty list of errors. Otherwise an empty list of bindings and a list with one error message specifying that unification failed is returned. The type of `unify` is given in Figure 7. If the case of the addition in Figure 1 of `x` and `y` in method `sum` which both have the dimension expression `a`, the unification succeeds and returns no bindings since they are the same expression. In the case of the additions that compute `perimeter` the dimension expressions are always `L` which also unify. If we were to unify dimension expressions `L` and `a` unification would succeed with the binding `a` $\mapsto$ `L`. In these cases the bindings are applied (using the `apply` function) to the dimension expression of `l` to get the new dimension expression used in the `typerep` for the type of the sum. For the erroneous addition of `len` and `area`, the dimension expressions `L` and `L L` will not unify. In this case the dimension expression in the `typerep` is an erroneous dimension expression. Our extension uses the unification algorithm given by Kennedy in his work on extending ML with dimension analysis [11].

```
aspect production add    e::Expr ::= l::Expr  r::Expr
{ transforms <- if match(dimTypeRep_Ex(_,_), l) &&
                   match(dimTypeRep_Ex(_,_), r)
                then [ add_dims(l,r) ]   else [ ] ;   }

abstract production add_dims   e::Expr ::= l::Expr  r::Expr
{ e.pp = l.pp ++ " + " ++ r.pp ;
  local attribute bnds :: [UnifyBnd] ;
  local attribute errs :: [String];
  (bnds,errs) = unify ( get_dimexpr(l.typerep), get_dimexpr(r.typerep) ) ;
  e.typerep = dimTypeRep_Ex(apply(bnds, get_dimexpr(l.typerep)), rep_tree);
  forwards to if   null(l.errors ++ r.errors)  then rep_tree
              else exprWithErrors(l.errors ++ r.errors)  ;
  local attribute rep_tree :: Expr ;
  rep_tree = if   null(errs)
             then add (get_rep_tree(l.typerep), get_rep_tree(r.typerep))
             else exprWithErrors(["Dimensions incompatible on addition."]);}
```

Fig. 6. Overloading addition.

The tree that the `add_dims` production will forward to (`rep_tree`) is either the sum of the representation trees of `l` and `r` (constructed by the production `add`) or an erroneous tree indicating that the dimensions were incompatible. It is this tree that this production will forward to. When unification succeeds, `rep_tree` is simply the same expression in which the dimension information has been removed and the variables and expressions have the type of the underlying representation type instead of the dimension type.

Overloading multiplication is done in a similar fashion except that we need only generate the product of the dimension expressions of the operands since multiplication of any dimensions is valid.

```
function unify   ([UnifyBnd],[Error]) ::= d1::DimExpr d2::DimExpr { ... }
function apply   DimExpr ::=  b::[UnifyBnd] de::DimExpr  { ... }
function compose [UnifyBnd] ::= b1::[UnifyBnd] b2::[UnifyBnd]  { ... }
```

Fig. 7. Function headers for unify, compose, and apply.

Overloading assignment and parameter passing: The productions for assignment and method call can also be overloaded in a similar fashion. For many language extensions that introduce new types this is often not necessary however since, if these productions are not overloaded, they forward to trees that use the `copy` production whose signature is shown below:

```
abstract production copy e::Expr ::= s::Expr t::TypeRep { ... }
```

By overloading this production, an extension in essence overloads assignment, the copying of parameters into a method, and the copying of the return value back out. In the dimension extension we overload this production with a `copy_dims` production that unifies the dimension expressions on `s` and `t` to check that the dimensions are compatible. It is quite similar to `add_dims` and we thus do not show it here.

Overloading method calls: Although the `copy` production above would ensure that on method calls the dimension expressions of the formal and actual parameters unify individually, we must check that they unify in a consistent manner and that we provide consistent substitutions for dimension variables in all places. For example, the call to `sum` in Figure 1 with arguments `len` and `area` is incorrect because we must unify the dimension variable `a` to the same dimension expression. Although `copy` would unify `a` to `L` and `a` to `L L` for each parameter individually this is not enough. Thus, we will overload the method call production with a type specific production for dimension types if any of the arguments have a dimension type. (In the case where only the return type is a dimension type the overloading of `copy` is sufficient.)

This new method call production calls the function `check_call` shown in Figure 8 and passes it the dimension expressions of the formal parameters and actual parameters that have dimension types. It also passes in an empty set of bindings. It will first check that for each parameter either both the formal and the actual parameter or neither have a dimension type. The `check_call` function calls `unify` on the first dimension expression in the `formals` and `actuals` lists after applying any previously discovered substitutions (`psubs`) to them. If unification succeeds, then it calls itself with the tails of the lists and the new substitutions (`new_subs`) and the application of them to the dimension expressions in the previous bindings `psubs`.

In the case of the incorrect call to `sum` mentioned above, `check_call` will first unify `a` to `L` for the first parameter `len`. It will then pass this binding in `psubs` in the recursive call. On the second call, it first applies this substitution to `a`, the dimension expression of the formal parameter `y` to get `L`, and then applies it to `L L`, the dimension expression the actual parameter `area`, to get `L L`. It then attempts to unify `L` and `L L` and fails—thus indicating that the arguments to `sum` have incompatible dimensions.

If the call to `check_call` succeeds, the returned set of substitutions are applied to the dimension expression of the return type to get the dimension expression that is used in the type representation of the method call. In the case of the valid call to `sum` with parameters `len` and `wid`, the substitution mapping `a` to `L` is applied to the return type dimension expression `a` to yield `L`—the length dimension that is then correctly used in the assignment to `len`.

3.4 Composing ableJ and the dimension analysis language extension

The attribute grammar fragment presented above in this section defines the dimension analysis constructs and analysis needed to extend the ableJ attribute

```
function check_call ([UnifyBnd], [String])
    ::= formals::[DimExpr] actuals::[DimExpr] psubs::[UnifyBnd]
{ return
   if      null(formals) && null(actuals)
   then    (psubs, [ ])
   else if null(formals) || null(actuals)
   then    (psubs, [ "incorrect number of arguments" ] )
   else if ! null(unify_errors)
   then    (psubs, [ "incompatible dimensions:" ++ head(formals).pp ++
                                      " and " ++ head(actuals).pp ] )
   else    check_call( tail(formals), tail(actuals),
                       new_subs ++ compose(new_subs,psubs)) ;
  local attribute new_subs :: [ UnifyBnd ] ;
  local attribute unify_errs :: [ String ] ;
  ( new_subs, unify_errs ) = unify ( apply(psubs,head(formals)) ,
                                     apply(psubs,head(actuals)) ) ;    }
```

Fig. 8. The check_call function

grammar specification of Java with dimension analysis. The process of composing these two attribute grammars is performed by the Silver tools. A component-wise union of the sets of nonterminals, terminals, productions, and attribute definitions on productions is straightforward and implements the composition of the host language and the language extension. This is easily extended to handle more than one language extension. Since this composition is performed by the Silver tools, non-experts can specify a host language and a desired set of language extensions that can be automatically composed to create a specification of a unique language that has features tailored to a particular task at hand. For more information on the composition process readers are directed to our previously published paper on this topic [17].

4 Related work

Dimension analysis: Previous research in this area has illustrated different techniques for adding dimension and unit analysis to programming languages. House [9] implemented the extension of dimensions to Pascal. He implemented a polymorphic dimension type system in the monomorphic type system of Pascal. Wand and O'Keefe [20] designed dimensional inference in an ML-like type system. In their extension, they extended single numeric types parameterized on dimension. They assumed a fixed number of base dimensions. Delft [5] extended Java with dimension analysis in a monolithic way. Kennedy [11] has implemented dimension extension to Standard ML programming using the ML's capabilities of polymorphism and type inference. It is his dimension unification algorithm that is used in our extension.

Allen et al. [2] provide a solution that differs from ours and the others described above. They first add meta-programming facilities (meta-classes) to an extension of Java called MixGen. They then use those to implement dimension and unit analysis. Their extension of dimension types integrates well with generics and subtypes, something not investigated in our extension.[2] This approach also supports composition of extensions since a programmer can use meta-classes from different libraries. It does not however allow the extension writer to create new syntactic constructs that reflect the notation of the domain. This is less critical for extensions that add new numeric types for dimension analaysis, but it is critical for language extensions that, for example, extend Java with SQL to support static checking for syntax and type errors [17].

Extensible languages: Mechanisms for specifying and implementing programming languages have been an active research area for many years and thus there is much work on this topic in general and, more specifically, on the topic of extending Java with different language features. We are thus necessarily very brief here. A more complete description of related work can be found in previous work [17, 16].

In the attribute grammar community there have been many investigations into the modular specification of languages [8, 10, 7, 1, 14], to cite just a few. Of particular interest is the JastAddII [6] system and its implementation of Java 1.5. Where as we use forwarding to add a form of (non-destructive) transformation to attribute grammars JastAddII adds destructive rewriting. Both allow for the implicit specification of semantics (that is attributes) through some transformation technique. Destructive rewriting has the advantage of not keeping both trees in memory at the same time and thus uses less memory than forwarding. It can also be used for traditional optimizations implemented as rewrite rules. Forwarding, on the other hand, has the advantage of allowing attributes to be computed on either the original extension AST or the forwarded-to host language AST. This supports a more modular specification of languages.

JavaBorg is an extensible Java tool that uses MetaBorg [4], an embedding tool that allows one to extend a host language by adding concrete syntax for objects. It is based on term rewriting and uses conditional rewriting of the AST to process programs. Thus, one must encode semantic analysis as rewrite rules.

5 Conclusion

There are several features that we are in the process of adding to our extension. The first adds the notion of units so that these can also be checked. This extension is straight forward since unit specifications like `ft` (for feet) might be used instead of `L` in dimension expressions. The unification algorithm must be extended to track units and multiplication must be checked for consistent use of units as well. A second may add the automatic conversion of non-dimension types (like `int`) to appropriate dimension types. Although more convenient, this

[2] Our apporach supports extensions that add new types and sub-type relationships [17].

is less safe than the current implementation. In extending ableJ 1.4 with Java 1.5 features we will investigate how new extension-introduced types such as the dimension types presented here can integrate with Java generics.

The language extension presented here adds dimension analysis to Java. Although it is less complete than some previous work on dimension analysis, it is distinguished in that it is added to Java as a *composable* language extension that also introduces new syntactic constructs. Thus, several extensions may be simultaneously added to Java so that the composed language contains new language features from more than one domain.

Acknowledgements

We thank Lijesh Krishnan for his work on the ableJ system.

References

1. S. R. Adams. *Modular Grammars for Programming Language Prototyping.* PhD thesis, University of Southampton, Department of Elec. and Comp. Sci., UK, 1993.
2. E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, and J. Guy L. Steele. Object-oriented units of measurement. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 384–403, New York, NY, USA, 2004. ACM.
3. J. T. Boyland. Remote attribute grammars. *J. ACM*, 52(4):627–687, 2005.
4. M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proc. of OOPSLA '04 Conf.*, pages 365–383, 2004.
5. A. V. Delft. A java extension with support for dimensions. *Software-Practice and Experience*, 29(7):605–616, June 1999.
6. T. Ekman and G. Hedin. Rewritable reference attributed grammars. In *Proc. of ECOOP '04 Conf.*, pages 144–169, 2004.
7. R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars. In *19th ACM Symp. on Prin. of Prog. Lang.*, pages 223–234, 1992.
8. H. Ganzinger. Increasing modularity and language-independency in automatically generated compilers. *Science of Computer Programing*, 3(3):223–278, 1983.
9. R. T. House. A proposal for an extended form of type checking of expressions. *The Computer Journal*, 26(4):366–374, Nov. 1983.
10. U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.
11. A. Kennedy. Dimension types. In D. Sannella, editor, *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming*, volume 788, pages 348–362, Edinburgh, U.K., April 1994. Springer.
12. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in **5**(1971) pp. 95–96.
13. NASA. Mars climate orbiter - mishap investigation report. Technical report, November 1999.
14. J. Saraiva and D. Swierstra. Generic Attribute Grammars. In *2nd Workshop on Attribute Grammars and their Applications*, pages 185–204, 1999.

15. E. Van Wyk, D. Bodin, L. Krishnan, and J. Gao. Silver: an extensible attribute grammar system. In *Proc. of LDTA 2007, 7th Workshop on Language Descriptions, Tools, and Analysis*, 2007.
16. E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th Intl. Conf. on Compiler Construction*, volume 2304 of *LNCS*, pages 128–142, 2002.
17. E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute grammar-based language extensions for Java. In *European Conf. on Object Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 575–599. Springer-Verlag, 2007.
18. E. Van Wyk and A. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Intl. Conf. on Generative Programming and Component Engineering, GPCE 2007*. ACM Press, October 2007.
19. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-order attribute grammars. In *ACM PLDI Conf.*, pages 131–145, 1990.
20. M. Wand and P. O'Keefe. Automatic dimensional inference. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 479–483, 1991.