A Domain-Specific Language for Generating Dataflow Analyzers

Jia Zeng [2]

Department of Computer Science
Columbia University, New York

Chuck Mitchell [3]

Microsoft Corporation
Redmond, Washington

Stephen A. Edwards [4]

Department of Computer Science
Columbia University, New York

Abstract

Dataflow analysis is a well-understood and very powerful technique for analyzing programs as part of the compilation process. Virtually all compilers use some sort of dataflow analysis as part of their optimization phase. However, despite being well-understood theoretically, such analyses are often difficult to code, making it difficult to quickly experiment with variants.

To address this, we developed a domain-specific language, Analyzer Generator (AG), that synthesizes dataflow analysis phases for Microsoft's Phoenix compiler framework. AG hides the fussy details needed to make analyses modular, yet generates code that is as efficient as the hand-coded equivalent. One key construct we introduce allows IR object classes to be extended without recompiling.

Experimental results on three analyses show that AG code can be one-tenth the size of the equivalent handwritten C++ code with no loss of performance. It is our hope that AG will make developing new dataflow analyses much easier.

Key words: Domain-specific language, Dataflow analysis, dynamic class extension, compiler, Phoenix compiler framework

[1] Edwards and his group are supported by an NSF CAREER award, an award from the SRC, and from New York State's NYSTAR program.

[2] Email: `jia@cs.columbia.edu`

[3] Email: `chuckm@microsoft.com`

[4] Email: `sedwards@cs.columbia.edu`

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

1 Introduction

Modern optimizing compilers are sprawling beasts. GCC 4.0.2, for example, tips the scales at over a million lines of code. Much of its heft is due simply to its many features: complete support for a real-world language, a hundred or more optimization algorithms, and countless back-ends. But the intrinsic complexity of its internal structures' APIs and the verbosity of its implementation language are also significant contributors.

We address the latter problem by providing a domain-specific language, AG for "Analyzer Generator," for writing dataflow analysis phases in Microsoft's Phoenix compiler framework. Experimentally, we show functionally equivalent analyses coded in AG can be less than one-tenth the number of lines of their hand-coded C++ counterparts and have comparable performance.

Reducing the number of lines of code needed to describe a particular analysis can reduce both coding and debugging time. We expect our language will make it possible to quickly conduct experiments that compare the effectiveness of various analyses. Finally, by providing a concise language that to allows analyses to be coded in a pseudocode-like notation mimicking standard texts [1], compiler students will be able to more quickly code and experiment with such algorithms.

One contribution of our work is a mechanism for dynamically extending existing classes. In writing a dataflow analysis, it is typical to want to add new fields and methods to existing classes in the intermediate representation (IR) in the analysis. Such fields, however, are unneeded after the analysis is completed, so we would like to discard them. While inheritance makes it easy to create new classes, most object-oriented languages do not allow existing classes to be changed. The main difference is that we want existing code to generate objects from the new class, which it would not otherwise do.

The challenge of extending classes is an active area of research in the aspect-oriented programming community [7], but their solutions differ from ours. For example, the very successful AspectJ [6] language provides the intertype declarations that can add fields and methods to existing classes. Like ours, this technique allows new class fields and methods to be defined outside the main file for the class, it is a compile-time mechanism that actually changes the underlying class representation, requiring the original class and everything that depends on it to be recompiled. In AG, only the code that extends the class must be recompiled when new fields are added.

MultiJava [3] provides a mechanism that is able to extend existing classes without recompiling them, much like our own, but their mechanism only allows adding methods, not fields, to existing classes.

In AG, we provide a seamless mechanism for adding annotations to existing IR classes. In AG code, the user may access such added fields with the same simple syntax as for fields in the original class. Adding such fields does not require recompiling any code that uses the original classes.

We implemented our AG compiler on top of Microsoft's Phoenix, a framework for building compilers and tools for program analysis, optimization, and testing. Like the SUIF system [13], Phoenix was specifically designed to be extensible and provides the ability, for example, to attach new fields to core data types without having to recompile the core. Unfortunately, implementing such a facility in C++ (in which Phoenix is coded) has a cost both in the complexity of code that makes use of such a facility and in its execution speed. Experimentally, we find the execution speed penalty is less than factor of four and could be improved; unfortunately, the verbosity penalty of using such a facility in C++ appears to be about a factor of ten. Reducing this is one of the main advantages of AG.

2 Related Work

The theory of dataflow analysis is well-studied. Kildall [8] was one of the first to propose a unified lattice-based framework for global program analysis. Later, Kam and Ullman [5] addressed the iterative approach and made the theory more concrete.

Wilhelm [12] notes that there are many generic theories for dataflow analysis, but few tools are built on these theories and even fewer are widely accepted. One big reason is the lack of a standard mid-level program representation. We expect the Phoenix compiler framework to address this problem, at least for object-oriented imperative languages. Another reason for the lack of tools is their complexity. Thus the focus of our work is to provide a simple language and tool for writing dataflow analyses.

Tjiang's Sharlit [10] is a tool for building iterative dataflow analyzers and optimizers. It is built on the SUIF [13] generic compiler construction framework. However, Sharlit did not introduce a new language. It uses C++ and provides some APIs, much like the Phoenix environment, and its focus was mostly on its efficiency, not its simplicity. While it makes an implementation of an analysis much more modular, it remains difficult to use.

A few tools require an explicit definition of the lattice used in dataflow analysis. Examples include Alt and Martin's PAG [2], Venkatesh and Fischer's SPARE [11], and the flexible architecture presented by Dwyer and Clarke [4]. PAG is well-known and has been used in industry. There are many similarities between AG and PAG: both use basic blocks and unchanged-pre-condition checking to improve the speed of the generated analyzer. Both provide a "set" data type. Unlike AG, PAG requires the user to specify the lattice used during analysis, which provides more optimization choices, like widening and narrowing, and makes it easier to verify the algorithm's correctness, but this makes PAG descriptions larger and more complex.

Some tools specifically address interprocedural analysis, such as Yi and Harrison's auto-generation work [14]. We focus only on intraprocedural analysis, although many of our ideas should carry over to inter-procedural problems.

3 The Design of AG

Ag is a high-level language that provides abstractions to describe iterative dataflow analyses. The ag compiler translates an ag program into C++ source and header files, which are then compiled to produce a Dynamically-Linked Library (dll) file. (Figure 1) This dll can then be plugged in to the Phoenix compiler and invoked just after a program is translated into Phoenix's Middle Intermediate Representation (mir).

Our generated plug-in extends ir objects to collect information and invokes a traversal that is part of the Phoenix framework to perform iterative analysis. This traversal function invokes computations defined in the ag program.

We follow the classical dataflow analysis approach. An ag program implicitly traverses the control-flow graph of the program and considers a basic block at a time. Inside each block, the analysis manipulates its constituent instructions and operands. We thus chose to make blocks, instructions, and operands basic objects in ag. Phoenix, naturally, already has such data types, but ag makes them easier to uses since our language has a deeper understanding of them.

One of the main contributions of ag is the ability to add attributes and computations to these fundamental data types. This facility relies on mechanisms already built in to Phoenix, but because of the limitations of C++, making use of such mechanisms is awkward and tedious to code. Ag makes it much easier.

To simplify the description of computation functions, we included new statements in ag such as *foreach* and data-flow equations like those found in any compiler text. We also introduced a *set* data type since data collected during dataflow analysis usually takes the form of sets.

Ag relies on the Phoenix Traverser class. This is an iterative traverser that does not guarantee boundedness. See Nielson and Nielson [9] for a discussion of the issues in guaranteeing boundedness.

4 The AG Language

The AG language is designed for dataflow analysis. It provides abstractions for the common features of iterative intraprocedural analysis. For user convenience and adaptability, we chose a syntax similar to that of C++ and added a variety of new statements and constructs.

4.1 Program Structure

Figure 2 shows the structure of a typical ag program to describe an analyzer. It defines a new, named phase, extends a number of built-in Phoenix classes with new fields and methods to define what information to collect, and finally defines a transfer function for the dataflow analysis.

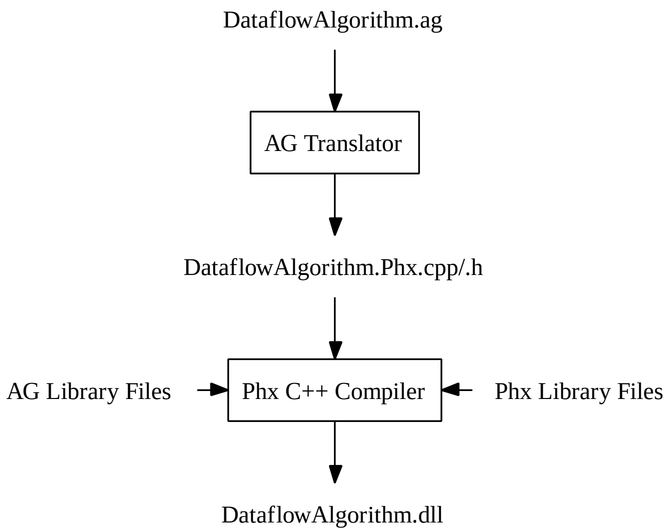

Fig. 1. The operation of the AG framework

```
Phase name {
    extend class name {
        field declarations...
        method declarations...
        void Init() { . . . }
    }
    ⋮
    type TransFunc(direction) {
        Compose(N) { . . . }
        Meet(P) { . . . }
        Result(N) { . . . }
    }
}
```

Fig. 2. The structure of an AG program

An *extend class* defines a new IR class that uses the Phoenix dynamically extensible IR class system. New fields and methods declared in an extend class are added as new class members. The user may directly refer to them as if they were members of the original class (our compiler identifies such fields and generates the appropriate Phoenix code to access and call members of such extended classes). Notice the methods declared in an extend class are "private," i.e. they can only be applied to the corresponding extend object, or in other methods declared under the same extend class. Currently, we only support extending Block, Instr, and Opnd classes.

In each extend class, the Init method behaves (and is executed as) an initializer just after the constructor for the extended class.

Each phase has a single TransFunc that defines the return type and iteration direction (backward or forward) of the analyzer and, more importantly, the equations applied during the analysis. The body of a TransFunc may define functions, especially three reserved functions: Compose, Meet, and Result. Compose and Meet functions are applied when the traverser iterates on every blocks. The Compose function defines the computation inside a block using global data. The Meet function defines the computation performed between blocks, i.e., to merge data from the exit of the predecessor to the entry of the successor. The Result function defines operations to be performed just after the iteration. It usually propagates information to the objects that make up the blocks, such as instructions. Other functions may be declared in the TransFunc; they can be called by the three reserved functions or each other.

The user may embed arbitrary C++ code in the body of these methods. Such code segments are transparent to AG compiler, which simply includes them verbatim in the generated code.

4.2 AG Syntax

We derived the syntax of AG from C++. We present its complete syntax in the appendix; Table 1 provides a summary. Below, we provide some details about its design.

Set is a data type similar to *set* in the C++ standard library. It can only apply to the reserved classes and actually refers to a set of IDs. For example, "Set<Instr>" will be translated into a bit-vector mapped on IDs of instructions in implementation. The *Map* type is similar.

During the analysis, the most relevant data are those with information for the entry and exit points of each block, so we introduced the *In* and *Out* data set as built-in variables.

Except for the two logical operators, the operators in Table 1 can be applied both to integers and *Set*-valued variables. Using the +, −, and * operators generate code that perform Or, Minus, and And operations on bit vectors.

In dataflow analysis, one often needs to iterate over a subset of objects, so we added a *foreach* statement to do this. *Foreach* is a predicated iterator, meaning that it steps through the members of a set and performs actions on only selected members of the set. The user does not have to declare an iterator specifically, just a variable of the type over which the iteration is occurring and the set on which to iterate. The user may also specify a condition that acts as a filter and a direction (Forward/increase or Backward/decrease). The condition is described with the *where* keyword. The syntax is shown in Table 1.

The *type*, *range* and *condition* allowed are listed in the attached syntax table. The "where *condition*" and "*direction*" parameters are optional.

data types	Set Map int bool void
special variables	In Out
operators	+ − ∗ = += −= ∗= && ‖
built-in classes	Opnd Instr Block Alias Expr Func Region
special methods	Init Compose Meet Result
built-in functions	DstAliasTable SrcAliasTable Print
built-in constants	Forward Backward
declarations	Phase *identifier* (*parameter list*) { ... }
	extend class *type* { ... }
	type TransFunc (*direction*) { ... }
statements	*lvalue* = *expression*;
	if (*expression*) { ... } else { ... }
	/% *arbitrary C++ code* %/
	foreach (*type var* in *range* where *cond. direction*) { ... }
	phoenix-iterator (...) { ... }

Table 1
AG Syntax Summary

Such *foreach* statements are translated to conditional for loops in the C++ and use the iterator macros in the Phoenix framework. Note that the *foreach* statement, especially the predication, is not strictly necessary (an additional *if* is sufficient), but the same can be said of C's *for* statement.

If the *range* is a Set, the *type* must match its content. Otherwise, if the *range* is a class, the *type* must match one of its members. For example, each instruction contains a list of operands, so we can specify a *type* of Opnd and a *range* of an instruction. Also, the user may specify a *condition* of "dataflow && dst" to iterate over dataflow-related destination operands in the list.

Phoenix provides a number of iterator macros, which can be used in AG almost verbatim (see Figure 3 Line 12). The only difference is that in C++, a matching "next" macro must follow the use of each iterator macro (see Figure 4 Line 26); this is not necessary in AG.

DstAliasTable is a reserved function that takes an alias tag x as parameter and returns a set of destination operands whose alias-tag is x. Similarly, *SrcAliasTable* returns all source operands with the same alias-tag.

5 An Example

To illustrate AG, we present a complete example: the classical "reaching definitions" dataflow analysis. The complete AG source is in Figure 3.

This algorithm computes the sets of definitions that reach the entry and exit points of each basic block in a program. Following the Dragon book [1], a definition of a variable is the operand in an instruction that may assign to the variable. In the Phoenix IR, each instruction has source operands and destination operands. For reaching definitions, we are concerned mostly with the destinations.

The whole analysis is defined as a phase called *ReachingDefs* (line 1 of Figure 3). The rest of the analysis consists of extend classes that add fields and computations to the built-in data types for operands, instructions, and basic blocks, and description of transfer functions.

5.1 Extend Classes

Extend classes augment existing data types with additional fields in which to collect information and procedures for collecting it. This is similar to extending a base class in an object-oriented language, but differs because the new attributes are actually attached to objects of the "base class" itself at the language level, not just in objects of derived classes (the C++ code we generate from AG actually uses class inheritance). But a user can refer to new attributes as if they were already in the original class. Consider the Opnd extend class (lines 3–20). This adds two attributes to each operand, operand sets named *Gen* and *Kill*. As usual, the *Gen* set contains operands that are defined within the block and available immediately after it in the source code.

The *Init* function initializes the values of the *Gen* and *Kill* fields. The two sets are implemented as bit vectors—see Lines 2–12 in Figure 4 for the declaration of *Gen*; Lines 14–29 show the translation of the *Init* function. The body of *Init* adds destination operands to the *Gen* set. Similarly, all other destination operands in the built-in destination-opnd-map-to-alias-tag table (DstAliasTable) that have the same alias tag as the operand (i.e., when both modify the same memory location) are added to the *Kill* set (Lines 12–17).

The *Instr* and *Block* extend classes add *Gen* and *Kill* sets to each of their classes and populate these sets with data from *Opnd* and *Instr* objects respectively. Lines 47–72 in Figure 4 call the three *Init* functions (the translation of the other two are not shown). Note that this function is synthesized completely from how this data is used in the analyzer, not from explicit code in the AG source.

After collecting *Gen* and *Kill* sets for blocks, the algorithm specifies some details of the main analysis iteration. At the beginning of the transfer function *TransFunc*, the iteration is declared to proceed in the forward direction and return a set of *Opnd* objects.

```
Phase ReachingDefs {

  extend class Opnd {
    Set<Opnd> Gen;
    Set<Opnd> Kill;

    void Init() {
      Opnd opnd = this;
      if (opnd->IsDef) {
        opnd->Gen += opnd;

        foreach_must_total_alias_of_tag(alias_tag, opnd->AliasTag, AliasInfo) {
          opnd->Kill += DstAliasTable(alias_tag);
        }
        opnd->Kill -= opnd;
      }
    }
  }

  extend class Instr {
    Set<Opnd> Gen;
    Set<Opnd> Kill;

    void Init() {
      Instr instr = this;

      foreach (Opnd dstOpnd in instr where (dataflow && dst)) {
        instr->Gen += dstOpnd->Gen;
        instr->Kill += dstOpnd->Kill;
      }
    }
  }

  extend class Block {
    Set<Opnd> Gen;
    Set<Opnd> Kill;

    void Init() {
      Block block = this;

      foreach (Instr instr in block) {
        block->Gen = instr->Gen + (block->Gen - instr->Kill);
        block->Kill = block->Kill + instr->Kill - instr->Gen;
      }
    }
  }

  Set<Opnd> TransFunc(Forward) {
    Compose(N) {
      Out = In - N->Kill + N->Gen;
    }

    Meet(P) {
      In += P->Out;
    }
  }
}
```

Fig. 3. A Complete AG analysis: Reaching Definitions

```
class OpndExtensionObject :
   public Phx::RbagGenTest::AG::OpndExtensionObject
{
  PHX_DECLARE_PROPERTY(Phx::BitVector::Sparse *, Gen);
  __PHX_DEFINED_VIRTUAL_GET_PROPERTY(Phx::BitVector::Sparse *, Gen) __const;
  __PHX_DEFINED_VIRTUAL_SET_PROPERTY(Phx::BitVector::Sparse *, Gen);

  Phx::BitVector::Sparse * _local_Gen;
}

void OpndExtensionObject::Init( Phx::FuncUnit *func_unit,
                                Phx::BitVector::Sparse *PHX_ARRAY(dst_alias_table))
{
  Phx::IR::Opnd *opnd = _this;
  if(opnd->IsDef) {
    this->Gen->SetBit(this->uid);
    foreach_must_total_alias_of_tag(alias_tag, opnd->AliasTag, func_unit->AliasInfo) {
      this->Kill->Or(dst_alias_table(alias_tag));
    }
    next_must_total_alias_of_tag;
    this->Kill->ClearBit(this->uid);
  }
}

void IterateData::Merge(
  Phx::DataFlow::Data *dependent_block_data,
  Phx::DataFlow::Data *effected_block_data,
  Phx::DataFlow::MergeFlags flags) {
  IterateData * dep_block_data = PTR_CAST(IterateData *, dependent_block_data);
  Phx::BitVector::Sparse * Out = dep_block_data->Out;

  if(flags & Phx::DataFlow::MergeFlags::First) In = Out->Copy();
  else In->Or(Out);
  dep_block_data->Out = Out;
}

void Traverser::InitData(Phx::BitVector::Sparse *PHX_ARRAY(dst_alias_table))
{
  foreach_block_in_func(block, funcUnit) {
    foreach_instr_in_block(instr, block) {
      foreach_dataflow_dst_opnd(dstopnd, instr) {
        OpndExtensionObject *ext_dstopnd =
           OpndExtensionObject::GetExtensionObject(dstopnd);
        ext_dstopnd->Init(funcUnit, dst_alias_table);
      }
      next_dataflow_dst_opnd;
      InstrExtensionObject *ext_instr =
         InstrExtensionObject::GetExtensionObject(instr);
      ext_instr->Init(funcUnit->Lifetime);
    }
    next_instr_in_block;
    BlockExtensionObject *ext_block =
        BlockExtensionObject::GetExtensionObject(block);
    ext_block->Init(funcUnit->Lifetime);
  }
  next_block_in_func;
}
```

Fig. 4. Part of the Phoenix (C++) code generated by the AG compiler for the reaching definitions example

The extend classes are based on original IR classes. The example in Figure 3 shows that the user may refer to fields from the extend class (e.g., Figure 3, Line 10, "`opnd->Gen`") using the same notation as for those in the base class (e.g., Figure 3, Line 13: "`opnd->AliasTag`"). These two references generate very different C++ code (c.f. Figure 4, Lines 21 and 23).

5.2 Transfer Function

As usual, we assume there are unique entry and exit points in the control flow graph for each block. "In" and "Out" are two built-in data sets related to the entry and the exit points respectively. The definition for TransFunc head declares the type of "In" and "Out" sets as holding operands. These two sets are usually used in the transfer function to pass data.

Compose and Meet are the two main functions for defining the transfer function. In this program, they specify the two groups of dataflow equations in the standard way [1, Eq. 10.9]:

$$in[B_i] = \bigcup_{B_j \text{ a predecessor of } B_i} out[B_j]$$

$$out[B_i] = gen[B_i] \cup (in[B_i] - kill[B_i]).$$

The first equation is exactly and simply included in the Meet function (Line 59), which computes the effect of the exit-point data from predecessors to the entry-point data of the current block in the iteration. *In* is related to the current block being visited, while *Out* is related to the block P that is passed to the *Meet* function. By default, the argument for the *Meet* function is a basic block that represents an arbitrary predecessor of the current block. As shown in Figure 4 lines 31–45, the data equation is translated into bit-vector manipulations.

The second dataflow equation is included in the *Compose* function (Line 55), which computes the data transformation globally from the entry point to the exit point for a single block. Declared as an argument to the *Compose* function, variable N is an extended object of the block by default. Since *Gen* and *Kill* are fields that have been added to the Block class (lines 38 and 39), they can be referred to as members of N.

5.3 Wrap up: Phase and Traverser

A complete AG program is translated into a C++ program that is compiled as a plug-in phase that can be invoked as part of the Phoenix compilation processes. It initializes all extended objects first, then executes the forward traverser, which applies the dataflow equations to iteratively compute on the blocks following the structure of the control-flow graph until the *In* sets converge for every block. The generated code uses the machinery built into the Phoenix framework to do this; an AG user does not write code for this.

	Reaching Definitions	**Live Variables**	**Uninitialized Variables**
C++ LOC (manual)	791	303*	108†
AG LOC (manual)	64	55	94
C++ LOC (generated)	626	519	682
C++ runtime	7.3s	0.8s	†
AG runtime	7.4s	3.1s	13.6s

*The manually-coded live variable analysis uses hard-coded fields, which makes it simpler at the expense of being far less modular.

†The manually-coded uninitialized variables analysis relies on the Phoenix SSA library not included in this count. This is a very different architecture than the code generated by AG.

Table 2
Experimental results: size and speed of AG-generated code vs. handwritten.

6 Experimental Results

We tested AG on three analyses: reaching definitions, live variables, and uninitialized variables. We chose these three examples because a hand-written version of each, done by experienced programmers, already existed in Phoenix. We compared the size and speed of the generated code with the manually written version for the first two examples because, like our generated code, they use the Traverser class in Phoenix. The manually-written version of uninitialized variables used Phoenix's static single-assignment code, which AG does not take advantage of, so we did not experiment with it.

Table 2 shows our results. "LOC" indicates the number of lines of code excluding comments; times are in seconds. We computed the average run times of these plug-ins by running compiler with the plug-in, running the compiler without the plug-in, and subtracting these two running times. The times are thus a little suspect because they also include the time to load and initialize the plug-in itself.

In each test case, the C++ code generated by the AG compiler is more than six times the size of the AG source. Even better for AG, the manually-written code for reaching definitions is even larger than the generated code. That is because the AG library files include commonly used code and default methods, for example, the constructor of the phase.

The manually-written live-variables code is smaller than the generated C++ code for that analysis, but this is because the manually-written code does not use the (verbose) Phoenix extend objects.

We ran the generated Phoenix C++ code on a laptop with a 2.0 GHz Pentium-M processor running Windows XP. The benchmark is the Phoenix

Microsoft Intermediate Language reader, which can generate high-level intermediate representations for a variety of targets. It is about five hundred thousand lines of code.

The AG-generated code for the reaching definitions analysis runs just as fast as the manually-written code on the MSIL reader. Unfortunately, the live variable analysis code runs about one-fourth as quickly, but there is a good reason for this: the manually-written C++ version does not use the Phoenix object-extension facility. Instead, it simply recomputes the desired data every time it traverses a block. Thus, the speed difference here more illustrates the cost of using extension objects instead a more brute-force approach. Evidently in this example, the computation is cheap enough so that repeating it is less costly than saving and recovering it later. We include the runtime for the AG code for uninitialized variables, but do not give a time for the manually-written code because it uses a completely different algorithm.

7 Conclusions

We presented a domain-specific language, AG, for writing dataflow analysis phases in Microsoft's Phoenix framework. Experimental results show that manually-written AG code can be less than one-tenth the size of the equivalent manually-written C++ with similar performance. A key enabler for the simplicity of AG code is its mechanism for extending existing IR classes, which makes it possible to extend existing classes without recompiling them and allows user-level code to access these fields as easily as typical ones.

As a small, domain-specific language, AG has some weaknesses. Minimizing verbosity was our focus, and we did so at the loss of some flexibility. The most obvious is that the user is forced to use the iterative analysis framework, even though Phoenix has other options, such as lattice and static single-assignment frameworks. Although AG has some high-level types such as sets and maps, its type system is limited and does not support strings, arrays, arbitrary iterators, and so forth.

AG is also currently limited to analyses running on the medium-level intermediate representation (MIR), although it could be extended to handle others. Furthermore, AG programs currently only handle user-defined variables; the many implicit temporary variables in the MIR are currently ignored. For example, the C statement on the left is dismantled as shown on the right. AG code currently ignores the temporary `t1`.

```
                                t1 = y + 3;
x = y + 3;        ⟶            x = t1;
```

As with many domain-specific languages, debugging AG is somewhat problematic. While we provide a print statement, AG does not have a dedicated debugger, IDE, or any of the other now-standard features in a development environment. All these could be added, but not without a fair amount of work.

AG is constructed as a translator, so in theory most weaknesses could be fixed by extending AG, provided the new features were supported by Phoenix. It could be extended, say, to describe region-based dataflow analyses, or to describe optimizations. But it is difficult to say at what point AG would cease to be a domain-specific language and balloon into C++.

Nevertheless, we believe that a factor of ten in code-size reduction justifies the extra challenges in using a small language.

For more information about Phoenix, see its official website: `http://research.microsoft.com/compilers`.

Acknowledgments

We would like to thank Al Aho for his enlightening discussions and interesting suggestions on this project. We also wish to thank the whole Phoenix group for their kind help and support.

References

[1] Aho, A. V., R. Sethi and J. D. Ullman, “Compilers, principles, techniques, and tools,” Addison-Wesley, 1988.

[2] Alt, M. and F. Martin, *Generation of efficient interprocedural analyzers with PAG*, in: *Proceedings of the Second International Symposium on Static Analysis (SAS)* (1995), pp. 33–50.

[3] Clifton, C., G. T. Leavens, C. Chambers and T. Millstein, *MultiJava: Modular open classes and symmetric multiple dispatch for Java*, in: *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Minneapolis, Minnesota, 2000, pp. 130–145.

[4] Dwyer, M. B. and L. A. Clarke, *A flexible architecture for building data flow analyzers*, in: *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, Berlin, Germany, 1996, pp. 554–564.

[5] Kam, J. B. and J. D. Ullman, *Global data flow analysis and iterative algorithms*, Journal of the ACM **23** (1976), pp. 158–171.

[6] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An overview of AspectJ*, Lecture Notes in Computer Science **2072** (2001), pp. 327–355.

[7] Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, *Aspect-oriented programming*, in: M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming* (1997), pp. 220–242.

[8] Kildall, G., *A unified approach to global program optimization*, in: *Proceedings of Principles of Programming Languages*, 1973, pp. 194–206.

[9] Nielson, H. R. and F. Nielson, *Bounded fixed point iteration*, in: *Proceedings of Principles of Programming Languages (POPL)*, Albuquerque, New Mexico, 1992, pp. 71–82.

[10] Tjiang, S. W. K. and J. L. Hennessy, *Sharlit: a tool for building optimizers*, in: *Proceedings of Programming Language Design and Implementation (PLDI)*, New York, New York, 1992, pp. 82–93.

[11] Venkatesh, G. A. and C. N. Fischer, *Spare: A development environment for program analysis algorithms*, IEEE Transactions on Software Engineering **18** (1992), pp. 304–318.

[12] Wilhelm, R., *Program analysis—a toolmaker's perspective*, ACM Computing Surveys **28** (1996), p. 177.

[13] Wilson, R. P., R. S. French, C. S. Wilson, S. P. Amarasinghe, J.-A. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam and J. L. Hennessy, *SUIF: An infrastructure for research on parallelizing and optimizing compilers*, ACM SIGPLAN Notices **29** (1994), pp. 31–37.

[14] Yi, K. and W. L. Harrison III, *Automatic generation and management of interprocedural program analyses*, in: *Proceedings of Principles of Programming Languages (POPL)*, Charleston, South Carolina, 1993, pp. 246–259.

AG Syntax

ag-phase:
`Phase` *identifier* `(` *parameter-list*$_{opt}$ `)` *compound-statement*

parameter-list:
parameter
parameter-list `,` *parameter*

parameter:
type identifier

type:
basic-type
extensible-class-type
`Set <` *type* `>`
`Map <` *type* `,` *type* `>`

basic-type: one of
`int  bool  void`

extensible-class-type: one of
`Alias  Opnd  Instr  Block  Region  Func`

compound-statement:
`{` *statements* `}`

statements:
statement
statements statement

statement:
variable-declaration
function-definition
extend-class-definition
assignment-expression$_{opt}$ `;`
if-else-statement
foreach-statement
phoenix-foreach
`continue ;`
`break ;`
`return` *expression*$_{opt}$ `;`
cpp-code-segment
compound-statement

variable-declaration:
type variable-declaration-list `;`

variable-declaration-list:
variable
variable-declaration-list `,` *variable*

variable:
identifier
identifier `=` *expression*

function-definition:
basic-function-definition
transfer-function-definition
compute-function-definition

basic-function-definition:
type identifier `(` *parameter-list*$_{opt}$ `)` *compound-statement*

transfer-function-definition:
type `TransFunc` `(` *direction* `)` *compound-statement*

compute-function-definition:
compute-function-name `(` *identifier*$_{opt}$ `)` *compound-statement*

compute-function-name: one of
`compose` `meet` `result`

extend-class-definition:
`extend class` *extensible-class-type compound-statement*

assignment-expression:
variable-or-field assignment-operator expression
expression

variable-or-field:
variable-or-field `->` *identifier*
identifier

expression:
 numeric-literal
 variable-or-field
 expression binary-operator expression
 `!` *expression*
 `-` *expression*
 variable-or-field `(` *variable-list*$_{opt}$ `)`
 `(` *expression* `)`

variable-list:
 variable-or-field
 variable-list `,` *variable-or-field*

binary-operator: one of
 `+  -  *  <  >  &&  ||  <=  >=  !=  ==`

assignment-operator: one of
 `=  +=  -=  *=`

if-else-statement:
 `if (` *expression* `)` *statement*
 `if (` *expression* `)` *statement* `else` *statement*

foreach-statement:
 `foreach (` *type identifier* `in` *expression* `where`$_{opt}$ *direction*$_{opt}$ `)` *compound-statement*

where:
 `where` *expression*

direction: one of
 `forward  backward`

phoenix-foreach:
 phoenix-foreach-keyword `(` *parameter-list*$_{opt}$ `)` *compound-statement*

cpp-code-segment:
 `/%` *C++-program-text* `%/`