Context Free Languages – An Application to Recursive Programs Analysis

CRISTINA ELENA ARON, EMIL MARIAN POPA, MIRCEA ADRIAN MUSAN
Department of Computer Science and Economic Informatics
University Lucian Blaga Sibiu, Faculty of Sciences
Str. Dr. Ion Ratiu 5-7, 550012, Sibiu
ROMANIA

Abstract: - Consider one approach to the analysis of a model of recursive computer programs. This version of a model is referred to as program schemes. With this model, programs are viewed from a higher level of abstraction. The goal is to compare the "power" of language features based on the analysis of the control and data flow characteristics of programs. Programming techniques complicate such comparisons. For instance, one can represent two independently manipulated integer values *x* and *y* with a single integer value; if a value of the form 2^x3^y is retained, then incrementing *x* is accomplished by multiplying the stored value by 2, and incrementing *y* is accomplished by multiplying the stored value by 3, etc. So for example, when asking if it's possible for every computer program that uses two integer variables to be replaced by another program that uses a single integer variable, the answer would be yes. And, if two variables can be replace by one, three variables could be replaced by two and hence then by one, etc. So the theoretical conclusion is that there is no need for more than a single integer variable although no one would recognize that he would write its programs this way.

Key-Words: - recursive program schemes, computation, interpretation, Herbrand interpretation.

1 Introduction

The program scheme model has proven useful in the analysis of major features of languages and programs. It avoids making any assumptions about the specifics of what is being computed. For instance, a scheme delimits the repetitive structure in a program, but does not include information to determine how many repetitions are performed. One of the primary kinds of issues that this model helps to analyze is whether a transformation on a program always gives a new program that computes the same results - that is, an equivalent program.

For one simple example, in compiler optimization of programs, if an assignment X := f(Y) is found within a loop and nothing in the loop ever changes Y, then the same effect can be obtained more efficiently by moving this assignment after the loop, so it is performed only once. This is independent of just what computation the function f performs. Of course, there are some conditions - subprogram f must be a "true function" whose value depends only on its arguments and it can have no side - effects. But these are conditions on the data and control flow of the program, not the computed values. Therefore some analysis is simplified by suppressing the details of the computation, and this is a primary characteristic of scheme models.

2 Problem Formulation

Let's examine how formal languages can help in analysis of programs in this context. Consider a simple functional form of programs.

2.1 Program schemes

2.1.1 Definition 1:

A **recursive program scheme** $\Re$ over **defined function names** $\{F_1, F_2, ..., F_m\}$ consists of $m \geq 1$ mutually recursive definitions of the form:

$F_k(x_1, x_2, ..., x_{i_k}) \Leftrightarrow$ **if** <predicate term>
then <function term>
else <function term>

one for each $k(1 \leq k \leq m)$, where $i_k \geq 1$ is the arity (number of arguments) of F_k. The constructions <predicate term> and <function term> are defined as follows: assume an unlimited number of **constant symbols,** for example a, b, c, … (possibly with subscripts); assume an unlimited number of

parameter identifiers (or **variables**), for example letters from the end of the alphabet (possibly with subscripts); assume for each $n \geq 1$ an unlimited number of n-ary **basic function symbols** (or **names**), for example f, g, h, … (possibly with subscripts), assume for each $n \geq 1$ an unlimited number of n-ary **predicate** (or **test**) **symbols** (or **names**), for example f, g, h, … (possibly with subscripts). A <**function term**> is defined inductively as either a constant or a variable, or $f(t_1, t_2, ..., t_n)$, where *f* is a n-ary basic or defined function name and $t_i (1 \leq i \leq n)$ is a <function term>. A <**predicate term**> is of the form $p(t_1, t_2, ..., t_n)$, where *p* is a n-ary predicate name and defined functions are all unary.

In a recursive program scheme F_1 will be regarded as the *function of interest* with the others serving as auxiliary functions supporting this definition.

The meaning of the defnitions in a scheme is formally described below. The basic function names are unspecified primitives, and the defined functions are expressed recursively using conditionals that make tests using primitive predicates. The facilities introduced only permit the writing of functional program structures, but this approach has been utilized for other styles of programming as well. Program schemes incorporate only the names of the primitive operations that are performed and information about the nature of the data and the effect of the primitives is completely omitted. A scheme provides an outline form of a program (actually many programs), describing its general structure but not the specifics of the computation.

2.1.2 Example 1:

The scheme $\Re_1$ below illustrates a structure that could be common to many programs, because it leaves computational details unspecified.

$F_1(x) \Leftrightarrow$ **if** $p(x)$ **then** c **else** $f(F_1(g(x), h(x)))$

This scheme involves unary test identifier *p*, unary basic function names *g* and *h*, binary basic function name *f*, and unary defined function name F_1, constant symbol *c*, and argument *x*. It exposes explicit control flow and data flow. It does not associate any explicit range of values for the arguments, specific constant value and leaves unspecified what functions or tests are associated with the basic names that appear.

This scheme describes computations that test the argument *x* with a basic predicate *p* and either return the value of the constant *c* if the test is true, or return the result of basic function *f* applied for two arguments that result from a recursive call with new argument *g*(*x*), and application of basic function *h*(*x*), if the test is false.

2.2 Interpretations

2.2.1 Definition 2:

An **interpretation *I*** of a program scheme $\Re$ provides:

- a domain of data values and
- assignments of:
  - a. each parameter variable of F_1 to a value in the data domain
  - b. each constant symbol to a value in the data domain
  - c. each basic function name to a n-ary function on the data domain and
  - d. each n-ary predicate name to a n-ary predicate, i.e., Boolean-valued function on the data domain

The pair $< \Re, I >$ is referred to as a **program.**

So when a program scheme is coupled with an interpretation, we have all the details of a fully specified program. The scheme captures the "structural" aspects of control and data flow and provides an abstraction of many programs. One scheme can be connected to any number of actual programs by the association of various interpretations.

2.2.2 Example 2:

This is an example of interpretation. This interpretation illustrates one of the programs associated with the scheme $\Re_1$ of Example 1.

Let interpretation I_1 consist of a data domain of lists (ordered sequences) of natural numbers, plus the correspondence of constant names, predicate names and basic functions names to concrete functions on this domain, as follows:

I_1:

< $p(V) \leftrightarrow V =$ null (as lists)
$f(V,W) \leftrightarrow app(V,W)$
$g(V) \leftrightarrow tail(V)$
$h(V,W) \leftrightarrow head(V)$
$c \leftrightarrow$ null

$x \leftrightarrow [1,2,3]$

> where:

app: List-of-Nat $\times$ Nat $\rightarrow$ List-of-Nat is the 2-argument function that yields a new list consisting of the second argument appended at the end of the first argument.

head: List-of-Nat $\rightarrow$ Nat returns the head (first item) of a list.

tail: List-of-Nat $\rightarrow$ List-of-Nat produces a new list by removing the head of its argument.

null: $\rightarrow$ List-of-Nat is the empty list constant (it is convenient to regard constants as 0-ary functions - a function that takes no arguments must always yield the same result).

Combined with this interpretation, the scheme is specialized to the program $< \Re_1, I_1 >$ below:

$F_1(x) \Leftrightarrow$ **if** x=null
then null
else app(F_1 (tail(x),head(x)))

It can be seen that this program computes the reversal of the list argument x. With interpretation I_1:

$F_1([1,2,3]) \Rightarrow app(F_1([2,3]),1) \Rightarrow$
$app(app(F_1([3]),2),1) \Rightarrow$
$app(app(app(F_1([]),3),2),1) \Rightarrow$
$app(app(app([],3),2),1) \Rightarrow$
$app(app([3],2),1) \Rightarrow app([3,2],1) \Rightarrow [3,2,1]$

2.3 Equivalent program schemes

2.3.1 Example 3:

This example provides a second interpretation of the scheme of Example 1. Let interpretation I_2 consist of the natural numbers as the data domain, plus the correspondences of constant names, predicate names and basic function names to concrete functions on this domain as follows:

I_2:
< $p(V) \leftrightarrow V = 0$
$f(V,W) \leftrightarrow V * W$
$g(V) \leftrightarrow V - 1$
$h(V) \leftrightarrow identity(V)$
$c \leftrightarrow 1$
$x \leftrightarrow 4$

> where all symbols represent the usual operations for $\mathbb{N}$.

When combined with this interpretation, the earlier scheme is specialized to the program $< \Re_1, I_2 >$ below:

$F_1(x) \Leftrightarrow$ **if** x=0
then 1
else F_1(x-1) * x

With the interpretation I_2 it can be seen that the resulting program computes the factorial of natural number argument x. With interpretation I_2:

$F_1(4) \Rightarrow F_1(3) * 4 \Rightarrow F_1(2) * 3 * 4 \Rightarrow$
$F_1(1) * 2 * 3 * 4 \Rightarrow F_1(0) * 1 * 2 * 3 * 4 \Rightarrow$
$1 * 1 * 2 * 3 * 4.$

These first three examples illustrate that common structures may be shared by quite different programs. With the scheme model, seek to analyze the properties that are determined by this structure alone. Passing from the specifics of the programs to the associated scheme raises the level of abstraction, and discarding computational details helps us to focus more clearly on characteristics implicit in the structure.

2.3.2 Definition 3:

Given a recursive program scheme $\Re$ and interpretation *I*, a **computation** is determined in the usual way - the interpretation specifies the initial values from the data domain for the arguments to F_1, the predicate is evaluated, and depending the result is true or false, either the then-term or the else-term is evaluated. Arguments are transmitted "by-value"- that is it's assumed that all arguments are evaluated before the invocation of any function. The value resulting when the program halts (if ever) is denoted by $val(\Re, I)$, and may either represent a value from the data domain or is undefined if $< \Re, I >$ does not terminate. Two schemes $\Re$ and $\Re'$ are **equivalent** if for all interpretations *I*, either both $val(\Re, I)$ and $val(\Re', I)$ are undefined, or both are defined and $val(\Re, I) = val(\Re', I)$. Two classes of program schemes *C* and *C'* are **equivalent** if for each $\Re \in C$ there is an equivalent $\Re' \in C'$

and conversely, for each $\Re' \in C'$ there is an equivalent $\Re \in C$.

2.4 Herbrand interpretation

2.4.1 Definition 4:

An interpretation *I* is called a **Herbrand interpretation** if:

- the data domain is the collection of formal terms over basic functions, called the **Herbrand universe**; for technical reasons adopt the assumption that the data domain of terms in a Herbrand universe have an adequate supply of constants (i.e., 0-ary functions) available, so automatically add as constants in the Herbrand universe as many unused symbols as the arity of F_1, and
- for each n-ary basic function name *f*, the function assigned to this name is the one where for argument terms $t_1, t_2, ..., t_n$ the result is the term $f(t_1, t_2, ..., t_n)$.

With a Herbrand interpretation, the values are purely symbolic representations, just formal terms, and the basic functions operate on argument terms by simply building a compound term that records the arguments and the function name. The only options for a Herbrand interpretation are in the selection of values assigned to parameters and in the selection of testing functions assigned to the predicate symbols. Using the formal terms for the data domain and having basic functions preserve the maximal amount of information about a computation. For example, don't simply end up with a value such as, say, 4; instead obtain a term that is a formal description of the computation that yields the results, such as plus(1,3), or plus(2,2) or times(2,2).

2.4.2 Example 4:

Consider program scheme $\Re_1$ of Example 1 with the Herbrand interpretation $x \leftrightarrow c$, and for *p*, $p(g(c)) =$ **false** and $p(t) =$ **true** for all other terms *t*. We then obtain the computation:

$$F_1(c) \Rightarrow f(F_1(g(c)), h(c)) \Rightarrow$$
$$f(f(F_1(g(g(c))), h(g(c))), h(c)) \Rightarrow$$
$$f(f(c, h(g(c))), h(c)).$$

The final result (if any) of a computation in a computation in a Herbrand interpretation shows the history of the computation that produced the result from the given arguments, since each step (i.e., application of a basic function) is recorded as a subterm. Herbrand interpretations thereby maximize the retention of information about the computation that produces a result. This observation is reflected in the following theorem.

3 Problem Solution

Theoreme:

Two program schemes that yields equivalent programs for all Herbrand interpretations are equivalent (i.e., yield equivalent programs for all interpretations).

Proof:

Suppose schemes $\Re$ and $\Re'$ yield equivalent programs for all Herbrand interpretations, and let *I* be an arbitrary interpretation. Since $\Re'$ is equivalent to $\Re$ for this Herbrand interpretation, it yields the same terms, and hence when these terms are interpreted with *I*, all the same values are produced.

Define a Herbrand interpretation *I'* based on *I*, that has the same computation steps in $\Re$ as I does, and when *I* is applied to the term $val(\Re, I')$ the result is $val(\Re, I)$. The data domain for *I'* is the collection of all basic function terms for function symbols occurring in $\Re$ and $\Re'$ (augmented by suitable constants).To define the predicate functions for *I'* we proceed inductively using the interpretation *I*.

Suppose the interpretation *I* makes the assignments $I(x_j) = a_j (1 \le j \le i_k)$ of parameters of F_1 to domain values; then, for Herbrand interpretation *I'*, the assignments are $I'(x_i) = c_i$, where $c_1, ..., c_{i_k}$ are the added constants in the Herbrand universe. For convenience, extend the definition of *I* to these added constants by $I(c_j) = a_j (1 \le j \le i_k)$. For atomic terms this anchors an inductive defi nition for applying the interpretation *I* to terms of the Herbrand universe to obtain a value in the data domain of *I*. For each compound term $f(t_1, t_2, ..., t_m)$, *I* is inductively extended via function *I(f)*, by defining $I(f(t_1, t_2, ..., t_m)) = I(f)(I(t_1), I(t_2), ..., I(t_m))$. Similarly for each predicate symbol *p* of arity m the test function $I'(p)(t_1 t_2, ..., t_m)$ is defined to be

$I(p)(I(t_1), I(t_2), ..., I(t_m))$, for argument terms $t_1, t_2, ..., t_m$.

With this definition of the predicates at each step the program $<\Re, I'>$ makes the same evaluation as the program $<\Re, I>$, but with term values whose evaluation under *I* yields the same value as in program $<\Re, I>$. Therefore program $<\Re, I>$ halts if and only if program $<\Re, I'>$ halts, and $I(val(\Re, I')) = val(\Re, I)$. But $<\Re', I'>$ is equivalent to $<\Re, I'>$, $val(\Re', I') = val(\Re, I')$, and hence $val(\Re', I) = I(val(\Re', I')) = I(val(\Re, I')) = val(\Re, I)$.Thus $\Re$ and $\Re'$ agree for every interpretation.

4 Conclusion

The scheme model is useful in providing an analytic comparison of many programming language features whose essential nature is unclear when analysis is oriented to programs rather than schemes. The deliberate absence of information about the behavior of the basic functions and tests, forces reliance on the features available to prevent the use of programming "tricks" (e.g., storing several independent variables in one).

References:

[1] Arthur Fleck, *Formal Models of Computation - the Ultimate Limits of Computation*, University of Iowa, USA, 2001.

[2] A.V.Aho, R.Sethi, J.D.Ullman, *Complilers principles, Techniques and Tools*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[3] B. Courcelle, M. Nivat, *The Algebraic Semantics of Recursive Program Schemes.* J. Winkowski, editor, Mathematical Foundations of Computer Science, LNCS64, Zakopane, 1978

[4] M.L. Lowry, R.D. McCarthy, editors. *Automatic Software Design*, MIT Press, Cambridge, 1991

[5] Theodor Rus, *Algebraic Processing of Programming Languages*, Theoretical Computer Science, Volume 199, No. 1-2, April 1998

[6] U. Schmid, F. Wysotzki, *Applying Inductive Program Synthesis to Macro Learning.* Proccedings of 5th International Conference on Artificial Intelligence, Planning and Scheduling (APIS 2000), AAAI Press, 2000

[7] R.Wilhem, D.Maures, *Compiler Design*, Addison-Wesley Publishing Company, 1995.

[8] http://citeseer.ist.psu.edu/360032.html

[9] http://citeseer.ist.psu.edu/115591.html

[10] www.ki.cs.tu-berlin.de/publications/Schmid

[11] www.springerlink.com

[12] http://portal.acm.org/citation.cfm?id=22584.24073